Guides

API Load Testing: A Beginner's Guide

Introduction

This book is intended to be a reference book, so while I would recommend that complete beginners to load testing read it cover to cover, I encourage readers with some experience in testing to flip to sections of interest, skip some altogether, go back and forth, and generally use it like the guidebook that it is. Here's a summary of what we'll go over.

This book is split into 7 major sections.

In the **Introduction**, we'll start with an explanation of what exactly an API is and why it's even worth testing. To help you figure out whether this is something that might apply to your situation, check out the sections on what API load testing can't do for you and what it can.

Then we'll jump straight into **Planning** for your load testing. You'll learn how to formulate nonfunctional requirements that are SMART to make it clear up front what kind of behaviour you expect from your application under load. You'll set your testing scope and entry criteria, learn how to create a workload model that will most closely resemble production, set up monitoring on your servers, stub out components, and design test scenarios that will recreate the situations your application is most likely to find itself in.

In **Scripting**, we'll discuss how to choose among the most popular and robust tools on the market today. I won't discuss how to script in every tool, but I will cover JMeter and Gatling basics and then point you to advanced resources. We'll also talk about some tips for how to script so that your tests are as realistic as possible and simulate real users' behaviour.

Then it will be time to start **Execution**. We'll discuss the benefits of having execution checklists for larger teams, a run list to keep track of test purposes and results, what to monitor during runtime, and how long you should run your tests for.

Analyzing Results and Reporting is the part of the testing process that management will pay most attention to. We'll go over which performance metrics you want to pay attention to, how each one can help you identify a performance bottleneck, and tips for presenting this to key stakeholders.

But you’re not done yet— we’ll also talk about the benefits of **Incorporating your API Load Testing into Your Continuous Integration Framework** so that you can set it up yourself and reap the benefits of having every release load tested. We’ll emphasise the continuous testing part of the equation with an argument for testing in production.

We’ll wrap up with **Other Considerations for API Load Testing**, going over other tests you can run to complement it and what else we can do to get a better idea of overall system performance.

Let's get started!

What is API load testing?

What's an API, anyway?

API stands for **A**pplication **P**rogramming **I**nterface. In simple terms, **an API is a set of rules for *how* to communicate with an application.**

These rules are necessary so that the application server can understand requests and process them accordingly. When a user accesses a web app through a browser, his or her clicks on the page and other interactions actually send requests to the server. APIs allow you to bypass those clicks in the UI and just get the information directly by sending requests.

How do you know what format you can use to formulate your requests in? That's where API documentation comes in. Let's have a look at an example.

Poké API is a collection of data about Pokémon, and their site includes a link to the Poké API documentation:

```
GET <https://pokeapi.co/api/v2/pokemon/>{id or name}
```

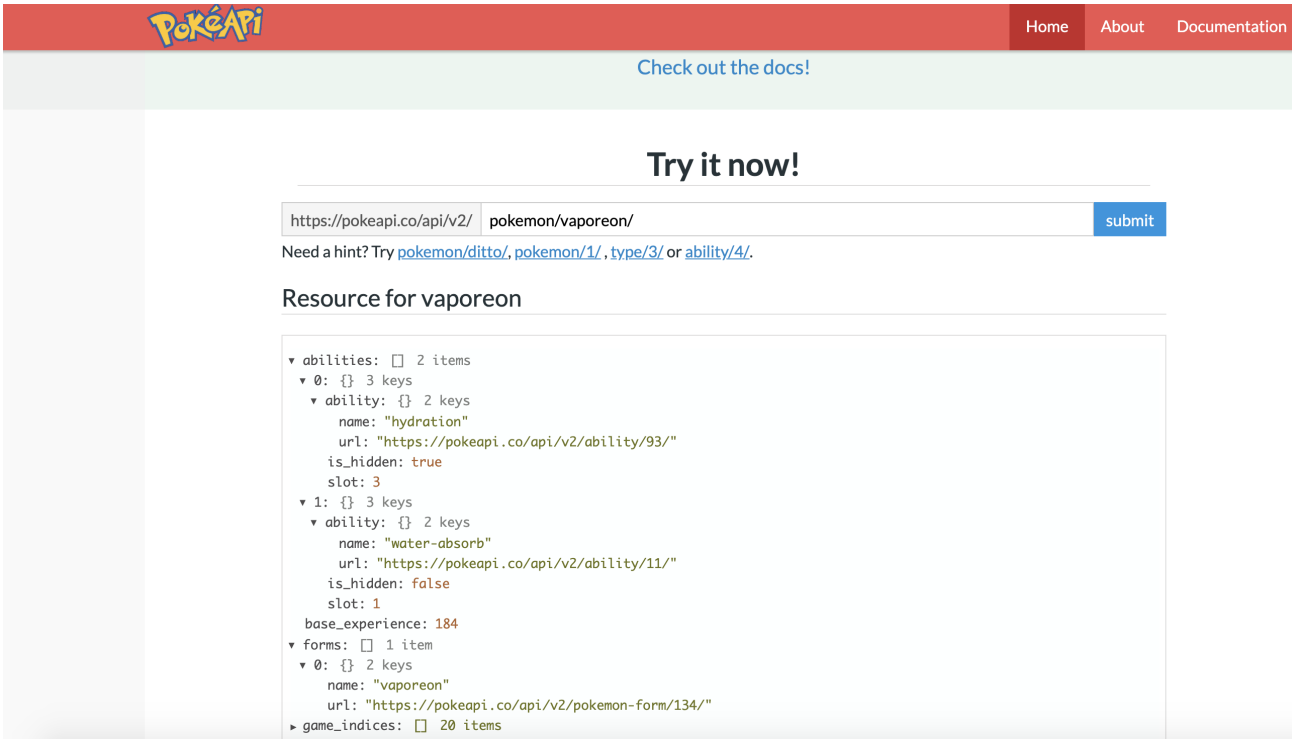
This is how you should formulate a request in order to retrieve information from their database. In this example:

- GET is the method (or "verb") that describes what kind of request you're making
- https://pokeapi.co/api/v2 is the base URL of the API
- /pokemon is the endpoint we're using and describes where the data we want is kept
- {id or name} is extra information we want to send along with our request

There are a few ways to actually consume (use) this API. In this case, let's say we want to do a search in the database for details on the pokemon Vaporeon. Using the format of the request above, we can use this request:

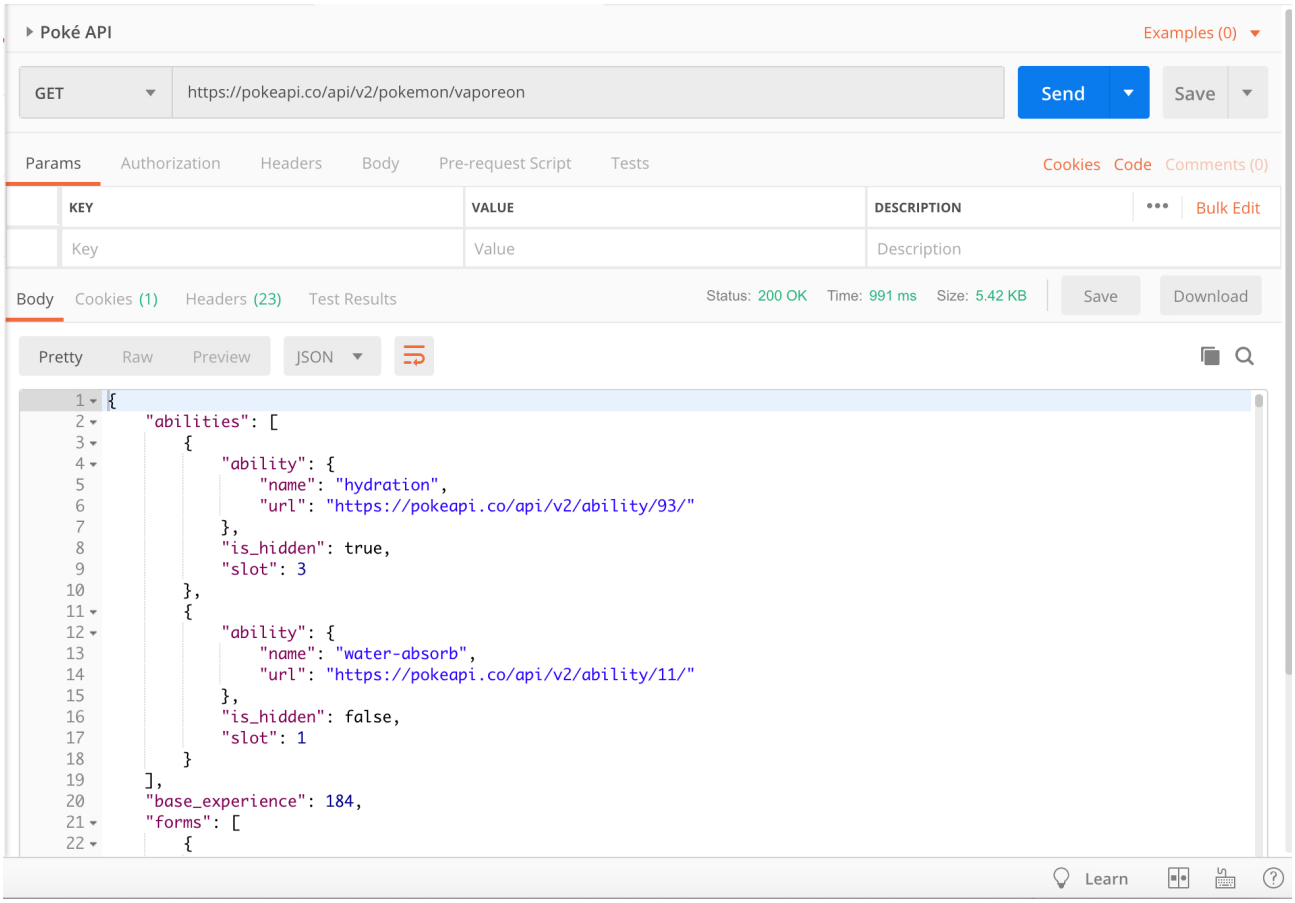
```
GET <https://pokeapi.co/api/v2/pokemon/vaporeon
```

You can test a simple GET request like this by typing https://pokeapi.co/api/v2/pokemon/vaporeon in your browser's URL bar and hitting enter. You'll likely see a LOT more information about Vaporeon than you ever wanted to know. To see a formatted version of all this data, use Poké API's simple interface and type in /pokemon/vaporeon:



This returns the same data, but formatted better.

You can also make the same requests using cURL via the command line or Postman for a friendlier interface.



Using Postman to get information about Vaporeon

Because the request is in a format that the application server understands, the application server replies and responds with the resources for that page. So **the API defines a language and syntax that clients (such as web browsers) need to use in order to interact with an application**.

There are several types of APIs, including SOAP, REST, JSON-RPC, and XML-RPC. Each one has a different standardized format for communicating with the web application. RESTful APIs (REST stands for REpresentational State Transfer) are particularly popular because of their simplicity. The Poké API is a RESTful or REST API. However, the principles discussed in this book will apply to load testing any API.

How to do API load testing

APIs can also present a more efficient way to load test your web application.

Load testing is determining the traffic that you can expect from production and applying it systematically to your application servers in order to determine how the application behaves. It's a way to check whether your application is robust enough to handle the traffic you want it to handle. Load testing is one type of performance testing, which is sometimes referred to as non-functional testing.

In API load testing, instead of your browser sending messages like `GET <https://pokeapi.co/api/v2/pokemon/vaporeon>`, you can use testing tools that will do that for you at scale. There are commercial and open source tools that you can use to create test scripts that you can use to tell those tools which messages to send and when. With API load testing, you can simulate multiple concurrent users sending requests to your server at the same time.

There are several steps to executing an API load test. Feel free to jump around to the respective sections for each step if you would prefer.

It's also important to be clear up-front about the advantages and disadvantages of API load testing so that you know whether it's the type of testing you need.

Disadvantages of API load testing

API load testing does not simulate real users interacting with elements of your webpage.

In contrast to a customer opening up a browser and filling out forms on your site, an API load testing script will consist only of the underlying requests to the server that are made by clicking on those on-screen elements. Exactly how the customer triggers those requests, and what buttons they've pressed, is irrelevant. It's all about the raw requests.

It doesn't give you an idea of how user-friendly your application is.

API load testing scripts don't give you feedback on how long your pictures took to render on your users' browsers or whether that "SUBMIT" button is in an obvious spot.

It doesn't measure front-end performance or how quickly pages render in different browsers.

While all the resources that the server returns can be downloaded by your API load testing tool, there is no browser on which to run them.

API load testing doesn't run client-side scripts.

It isn't what you're looking for if your application is a single-page web app that relies heavily on JavaScript or AJAX to dynamically populate and update the page. Your load testing tool will download the scripts, but not execute them.

This includes scripts like those required to trigger Google Analytics, which prompts many an engineer to question the results of an API load test due to the inability to see the traffic come in on Google Analytics. API load testing won't help you with that. Running browser-

level load tests using tools like Flood Element, Selenium or Tricentis Tosca may be more useful.

Advantages of API load testing

Now that we've talked about what API load testing isn't good at, let's talk about what it IS good at.

It allows you to load test specific servers rather than the whole stack.

API load testing allows you to tailor your load testing by applying load only on particular servers. This is particularly useful in more complicated applications that involve several components and would require substantial effort to reproduce in a test environment. Using APIs, you can test only the functions you want to test.

It is well-supported.

API load testing has been around for decades. There are lots of robust tools you can choose from, both commercial and open-source, and many of these tools have large communities and extensive documentation around how to script the most common cases. It's a far cry from the browser-level testing space, which is relatively new and sparsely populated by comparison.

It's not as resource-intensive as browser-level testing.

You can simulate more users with API load testing than with browser-level load testing. Since most interactions on the browser-level get translated to requests anyway, generating load this way is incredibly efficient, allowing you to hit your server with requests while bypassing the UI layer and the resource overhead that that entails.

For example, at Flood we've baselined some of the tools that we support to see how many users we can run on an AWS m5.xlarge instance:

Selenium: 5 users
Flood Element: 20-40 users
JMeter: 1000 users
Gatling: 1000 users

You'll see that the browser-level test tools, Selenium and Flood Element, can run significantly fewer users on the same sized node as can the protocol-level tools, JMeter and Gatling. This isn't due to an inefficiency in the tools but rather to the fact that browser-level tools inherently need more CPU and memory to start separate instances of browsers per user and render pages graphically. You can run more users per node (or machine) by carrying out API load testing with protocol-level tools.

It's cheaper.

The efficiency in resource utilisation translates directly into cost savings because every node that you don't need to execute your tests on is one fewer node that you'll have to pay to provision (whether on premises or in the cloud).

For this reason, API load testing is one of the most cost-efficient ways you can get started with load testing, allowing you to scale up your load relatively cheaply while getting immediate results.

Good API load testing will generate load that is a good representation of load you could expect to see in production, giving you an accurate estimate of how your application architecture will behave and highlighting any performance bottlenecks that can be fixed prior to go-live.

The rest of the book

In this book, we'll go over everything you need to know to plan, script, execute and analyse your API load tests and even how to add load testing into your Continuous Integration framework.

Planning for API load testing

Requirements

Every good test begins with requirements.

You may think, “But I’m a tester! I’m not a business analyst!”, and I hear you. But as testers, especially on smaller teams or shorter projects, it’s still part of our job to make sure we know what our clients want. Otherwise, how do we know whether we’ve succeeded or failed?

I often speak to project teams who ask for help interpreting reports. Most of those times, the problem is not that they don’t understand the metrics or what they measure. The problem is that they didn’t set pass or fail criteria in the first place, and so they don’t have anything to judge the results against. This is always a big warning sign.

Requirements inform every step of the load testing process. Why are we doing load testing? What exactly do we want to test? How will we know when a test has passed or failed? How will we know if application performance is good enough to go into production? What does “good enough” mean?

Tim Koopmans, co-founder of Flood, coined the acronym SPEAR to describe the different aspects of a performance that should be considered in load testing. This is a great starting point when thinking about what we want our nonfunctional requirements to cover.

Scalability

Scalability is the application’s ability to cope with increasing demands by increasing the amount of server resources. This could mean scaling up (increasing the resources of the dedicated server) or scaling out (adding more nodes to shoulder the load). What happens when more users than expected sign up in response to a promotion on your site?

Performance

The most common performance metric is page response time, but there are other considerations here, such as throughput (requests per minute) and the number of

concurrent sessions that need to be supported. Things like the total size of the resources on the page, whether or not a CDN is being used, and what to cache are also worth discussing.

Elasticity

Elasticity is a relatively newer aspect to performance testing brought about by advances in the cloud that allow application infrastructure to adapt to changes in load. Unlike scalability, elasticity emphasises scaling down as much as it does scaling up. Testing that virtual machines scale up when load increases is important, but testing that virtual machines also scale down when load decreases can also help save on unnecessary costs.

Availability

To test for high availability, ask yourself what would happen when (not if) your application's server fails. Is there another server that the load balancer will seamlessly send traffic towards? Does the throughput fluctuate wildly? If users are connected to one server that fails, is your application smart enough to make new connections to another server? Or will it simply serve up an error page that users won't know what to do with? Disaster recovery is best tested when there's no disaster imminent.

Reliability

Reliability encompasses a lot of scenarios, but they all have to do with whether or not your application returns expected responses. Does your error rate increase when you increase the duration of your load test? Are you adding verification steps to your load testing scripts to check whether or not the HTTP 200 response that the application returned is not actually an error page?

What should my application's response time be?

Our clients frequently ask us what the industry standard is for response times, wanting to make sure their applications measure up. The answer, however, is more complicated than a single number.

Industry standards for response time are only useful when applications are very similar. Constantly changing technologies used in web development as well as innate differences in business processes, however, make it very difficult to extrapolate a single number that will apply to all, or even most, applications in a certain industry.

The home page of one e-commerce app, for instance, might be several seconds slower than that of their main competitor. However, that doesn't take into account the fact that their app loads a video showcasing new products. Does that mean that the development team should remove the video in order to fall in line with their competition?

Well, maybe. But not necessarily. It's a business decision that needs to be made after perhaps using focus groups to determine the impact of the video, forecasting changes in conversion rate due to it, and comparing its projected value to the effects of being slower than the competition. A/B experiments could be used to test these assumptions and gather quantifiable data to support the team decision.

These factors are often not considered in the search for one number to rule them all, which is why a fixation on that number can be detrimental. Instead, I encourage project teams to brainstorm and come up with their own numbers for all metrics that would be more appropriate for their application. Gathering comparative metrics from a competitor may be part of this process.

What is a good API load testing requirement?

A good requirement, just like any good goal, is SMART:

Specific. Vagueness in a requirement leads to vagueness in results interpretation.

Instead of: "The performance of the web application"

Try: "The average response time of the Login transaction"

Measurable. Make sure there is a quantifiable way to know whether requirements have been achieved.

Instead of: "Decrease user frustration"

Try: "The error rate must be below 3% at peak load"

Agreed Upon. Have the appropriate stakeholders been involved?

Instead of: "The system must be able to generate emails as soon as users register"

Try: “The system must be able to generate a maximum of 100 emails an hour, after which emails are queued”

A good example for this is a project I was involved in that aimed to increase the speed with which emails were generated. Unfortunately, the particular email being sent included a big change that many customers were expected to contact Support about, and nobody had looped in the Customer Support team. They quickly raised their concern that they would not be able to handle the expected volume of emails unless the emails were staggered. This could have been avoided if they had been brought into discussions from the very beginning, in the requirements gathering phase.

Realistic. Can we meet this requirement given the resources available?

Instead of: “All requests must be returned within 5 ms”

Try: “90% of the Catalog page requests should be returned within 3 seconds”

Timely. Especially for nonfunctional testing, consider adding a timeframe to requirements.

Instead of: “The digital code is sent by SMS upon successful client log in at peak load”

Try: “The digital code will be sent by SMS no later than 5 minutes after successful client log in at peak load”

Start with why

Requirements represent a great opportunity to think things through and make sure everyone on the project team is on the same page about the goals for your load testing. Too often projects skip this phase, only to realise much later that the tests that were executed didn’t address a key stakeholder’s concerns.

In load testing, as in many things: when in doubt about what to do, start with why.

Scope

The discussion of scope among stakeholders should also occur during the requirements gathering process. There’s a big pressure to fit in as much as possible into a sprint, but it’s

important to stop and think about what amount of work is realistic to include.

Business priorities need to be weighed against the resource limitations (number of people available to do the work and time available) in order for testing to deliver maximum value.

Some considerations for scope include:

- Specific features or key transactions to be tested

- Types of tests included (component test vs end-to-end test)

- Test scenarios (peak load test vs disaster recovery)

- Applications included in testing

It's also a good idea to add things that will not be tested.

As with most things in the planning phase, scope is something that can change during the test when unexpected circumstances arise or when priorities change.

Entry criteria

Entry criteria are conditions that you need to be fulfilled before the testing actually begins. It's a good idea to have these conditions communicated beforehand so that everyone is clear on what needs to be set up before you can do your job.

For nonfunctional testing, there are several general conditions you might want to include in your entry criteria.

Functional testing

Load testing cannot realistically be carried out until at least the core functionality has been tested and high-severity defects have been fixed. Depending on the kind of load test you want to execute, you may also want to specify that user-acceptance testing (UAT) has been executed, as there's no point doing an end-to-end load test with 1000 users if it doesn't work for one user.

Environment

Nonfunctional testing has stricter requirements for an environment than does functional testing, and you may have to champion this cause. For load testing, it is not enough to have an application staging environment that is a virtual machine that is a quarter of the size of the production environment. It's important to get as close to a production-like environment as possible in terms of capacity (memory, CPU), codebase (the actual build that will be deployed), and integrations with other environments or servers (if within test scope).

Load testing is not linear: a response time of 5 seconds on a server with half the capacity of the production server does not necessarily equate to a response time of 2.5 seconds in production.

Once you have as close a copy of the production environment as possible, keep in mind that it's still a clean copy, which may not be realistic. If there are databases in production, how much data do they contain? The application server may respond differently when your test database is empty compared to when it must contend with gigabytes of data in the production database.

This is also the time to think about your load injectors. Will they be on-premises, or in the cloud? A good entry criterion is the availability of the machines in the right network and with the right tools installed. If you're using commercial tools, license provisioning should be a criterion. What sort of capacity will your load testing scripts require?

Monitoring tools that you use during execution will also fall under environment criteria, but we'll discuss that a little bit more in detail later.

Application Teams to Be Involved

Load testing is a team activity. When a load test involves multiple application teams, it's important to request availability of key persons on those teams during the test. Often as load testers, we are seen as working independently, but the truth couldn't be farther from that. Load testing is a team sport. We need support from:

- business analysts who will be able to tell us how things are expected to work and what the current priorities are

- developers whom we can consult when poorly performing code needs to be optimised

- functional testers who can show us how the application works

- DevOps engineers who can help us provision and monitor servers

and many more!

There should be a test schedule drafted with the input of all key resources.

Test Data

Test data should be examined in conjunction with the determination of the key business processes to be tested. For example, if part of the requirements involves testing user logins, where will the user credentials come from?

In general, there are three ways to get test data:

1. Take it from production. One way to do this would be to take a copy of the production data and use it for your test environment. Note that you may have to scrub sensitive data (such as customer information or passwords) before you can do this. Email addresses should particularly be scrubbed and ideally replaced with a fake domain (such as "xxxx@companyperformancetest.com") in case there's a chance that notifications are generated by the test itself that sends emails to real clients.

An advantage of this is that it's more production-like and may expose issues with non-Latin characters, for example, that may not otherwise have been tested.

2. Inject records into the database. This can often be the easiest way to generate test data. Your friendly DBA can create a script to randomly or sequentially generate the data from the back-end.

3. Write a script to generate the data yourself. Sometimes neither of the first two options is feasible, and in those cases your only other recourse is to create the test data yourself. A disadvantage to this approach is that it will take more time to create the data before you can run your test, but an advantage is that it's another functionality that you've scripted that can perhaps be included in the test suite.

When running a test that either requires a significant amount of data or consumes the data in the course of the test, it's a good idea to take a backup of the relevant databases and create a restore point after the data has been created but before the test is executed. That

way you can save some time by falling back to that restore point after the test so that the data is in a known state each time.

Workload modelling

A workload model is a schema describing the load profile for a given test scenario, and it involves determining what (the key transactions), how much (the load distribution among the transactions) and when (timing of the load) to test.

Workload modelling can be the most difficult part of the testing process because it involves finding out how load test scripting can best mimic what is actually happening in production. It can also be the most critical.

Imagine a project that runs load tests of the guest checkout process on their e-commerce site. Despite having tested up to 1,000 concurrent users on their production-like environment and getting sub-two-second response times, they get response times of greater than a minute in production before their servers start to go belly up. What could have gone wrong?

Well, a lot of things, but one thing they may not have taken into account is the standard user path. Perhaps they assumed that most customers would checkout as guests without logging into their accounts, but in reality, 90% of their customers log in before checking out. This means that there's a big gap in their testing: user login. Perhaps it wasn't the main application server that ran into issues at all; it could have been their authentication servers.

Workload modelling ensures that you're testing what you need to test, but it can be more complicated than it sounds.

You can begin building a workload model by gathering data from production about what users are actually doing on your application. Real historical data on the traffic over a wide enough time period (a year or six months) is ideal, so that you get a broad view. If you have something like Google Analytics or other tools such as New Relic, even better.

But what if your application, or the specific feature you're launching, is new, and you don't have any historical data to look through? In this case, the answer is to involve business analysts as well as developers and estimate the traffic patterns. If you have data for other

similar transactions, you can analyse them and extrapolate best guesses from them about the new transactions.

Whether or not you have the data, here are a few things you will want to determine as a team.

Key transactions

This, together with the scope and business need, will determine what you script and what you actually test. Load testing should not be exhaustive. It may sound good to be able to script absolutely everything, but not everything needs to be load tested. Here's how to determine what does:

- Business-critical or high risk transactions. Include anything that is vital to the success of the release. An example for this would be the final purchasing page on an ecommerce site or the submission of a contact details form for a request for quote. If it leads directly to conversion, script it and test it.

- Known pain points. A good place to start is any pages that your Customer Support team have reported as a pain point for customers. Let customers tell you what they want.

- Transactions that are technically complex. These transactions sometimes go through several application servers and are processed several times before a response is returned to the user. This means that they're also the most likely to present issues when one or all of those servers are under load.

- High-traffic transactions. These are the pages that customers use the most. This could be a landing page or the product catalog page that may not be the most complex but is frequently visited. There may be some surprises in this— the landing page might have been obvious, but perhaps you'll find that the many customers browse to the Contact Us page as well.

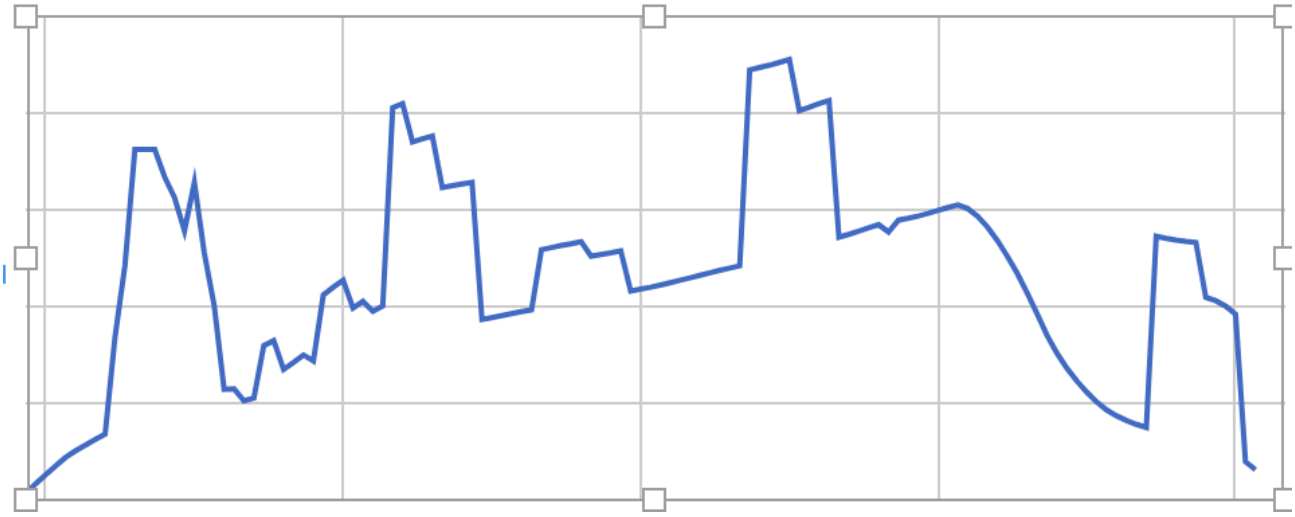
Load profile

The load profile will tell you how the load is distributed over time. Since applications are used in different ways, it's not enough to get an average hits per second over the last year and test at that level. You'll need to understand which transactions those hits are going to.

Look specifically for trends over time. These could be yearly, seasonal, monthly or even daily patterns. Does traffic pick up in the month before the tax filing deadline? Do students access their enrolment portal just before the enrolments are finalised? Does management run a series of reports that take their toll on the server at the beginning of every month? Do employees usually navigate to the intranet page at the beginning of the day, and then again after lunch?

One of the most marked examples of this is the load testing required for a sports event. One of our favourite clients at Flood, Hotstar, simulated video streaming traffic to their site in preparation for the Indian Premier League in 2018.

After analysing the data they had from previous years, they decided on a load test that had this load profile:



The spikes in load correspond to the start of cricket games. This spiky profile means that they had to test higher levels of load for a short period of time, instead of testing with a lower user count for a longer time. In this case, using the average number of users throughout the day would have drastically oversimplified things. Given that their load tests required 4 million concurrent users, that small change in the workload model could have negatively impacted their results and performance during the games in production.

You can only determine what the “Peak” in traffic is for your application after you’ve seen historical data. If you’re like Hotstar and you want to test the performance on game day, don’t pick traffic from some other month where there are no games— take a look at your highest season traffic-wise and test for that. You may even want to increase those figures by 10% or so in order to leave some room for growth.

Geographical location

So, now that you know what you're testing and when you're testing, the next question is where you're testing from.

The geographical location of your load test generators affects the response time you'll get. If you use on-premises machines to generate the load and see 1-second response times, the machines are on the same corporate network as your application servers and have less latency. This means that the response times will be a lot faster than those of a client using your application from across the world.

Depending on where your customers live, latency can change the reported response time. In general, you would want to have a look at your analytics and see where most of your customer base is. Ideally, you'd want to generate load in those regions— remember, we're trying to make your load testing as realistic as possible.

This is one big reason to switch from on-premises load generators to load generators in the cloud. Service providers like Amazon, Azure and Google allow you to provision machines with a few clicks and even select their location. Load testing on the cloud can be significantly cheaper (because you don't have to provision or maintain physical machines and you only pay for when you use them) and more realistic, because they allow you to approximate the effect of distance on your application response times. It also allows you to test the effect of Content Distribution Networks (CDNs), if you're using any.

Bandwidth emulation

With more and more people browsing the internet on their mobile devices, it's worth considering how big an effect users' networks have on their experience of your application. If your app is an internal company portal, this probably won't matter, but if it's a web app that is meant to be accessed both on desktops and mobiles, you can't discount the effect of slow 3G networks.

While you can't control users' network speeds, what you can do is simulate them by throttling the bandwidth available so that you can see how quickly your application would respond for them. Most load test tools can do this, and you may be able to determine how many of your users access your app on slower connections by looking at your analytics. If you decide that this is in scope, you can then build it into your scripts so that your tests will also show you response times on different types of networks.

Network Bandwidth throttling in JMeter

In JMeter, you can control this by adding the following line to your user.properties or jmeter.properties file:

httpclient.socket.https.cps=19200000

This sets the “characters per second” and, when set to anything greater than zero, will allow you to simulate different speeds. Here’s a way to calculate the value to set here:

cps = (target bandwidth in kbps * 1024) / 8

If you’re running your JMeter test through Flood, there’s no need to modify your properties file. You can simply add this line to the Advanced Parameters section when you’re editing your stream:

-Jhttpclient.socket.https.cps=19200000

For more information about this, check out Apache’s documentation.

Network bandwidth throttling in Gatling

With Gatling, this is also indirectly possible to simulate by throttling the number of requests per second. It’ll be a little more difficult to correlate requests per second to network bandwidth, but seeing historical data of real mobile users of your application should help here. Then you can multiply that by the number of users in your simulation and set up the throttling using something like this:

```
setUp(scn.inject(constantUsersPerSec(100) during (30 minutes))).throttle(
  reachRps(100) in (10 seconds),
  holdFor(1 hour)
)
```

Here’s Gatling’s documentation for more information on this.

Server monitoring

Most load testing tools will display some standard load testing metrics such as response times, throughput, error rate, and others, but you'll also need to set up monitoring on the application components that you're testing.

Executing a load test without monitoring server health is like flying blind. You'll know when you land safely and you'll know when you crash, but even if you do crash, you won't know why—or how you can avoid it next time. Monitoring server health is the black box that will tell you what went wrong.

What Metrics to Monitor

There are a lot of metrics that you can monitor. Here are just a few:

- ****Processor Time** - how much the processor is being utilised
- **Processor Interrupt Time** - how much time the processor is spending to handle interrupts
- **Processor Privileged Time** - the time the processor spends handling overhead activities
- **Processor Queue Length** - the number of threads that are waiting to be executed
- ****Memory (Available Bytes)** - unused memory available to process new requests
- **Memory Cache Bytes** - the size of the data stored in memory for quick retrieval
- **Disk I/O** - reads and writes to the disk during the test
- **Disk Idle Time** - time that disks are not doing work
- **Disk Transfer/sec** - average number of seconds that an I/O request takes to complete
- **Disk Write/sec** - average number of seconds that a write request takes to complete
- **Network I/O** - bytes sent and received

And that's just a small sample! How do you determine which one to use?

If you're not sure where to start: **At a minimum, you'll need the CPU and memory utilisation** (with asterisks in the list above) of every major component that's involved in the processing of requests. These two metrics are vital and if either of these is consistently maxing out at (or close to) 100%, that's a sign that the component is struggling with the number of requests. CPU and memory over-utilisation is a very common reason for less-than-ideal response times.

How to Monitor

How you get these metrics depends on your budget and your operating system. I'll start with the free or lower cost methods and work my way up to enterprise solutions.

If your servers are running Windows, **PerfMon** is a good alternative. It's built into Windows and the interface allows you to choose the counters that you want to measure and start recording.

There's also a JMeter server agent that you can download, also confusingly called "PerfMon", that you can install on your (Windows or Linux) server and will gather metrics. It's normally used in conjunction with the PerfMon JMeter plugin.

Nagios Core is another good open-source way to not only collect metrics but also set up alerts when values exceed certain thresholds that you can set.

Moving from budget options to more enterprise solutions, DynaTrace is a powerful tool that allows you to track not just server health but can actually be used to trace individual requests using a custom header.

AppDynamics is another fantastic tool that allows you to really drill down to specific SQL queries that take a long time to execute, for example, feeding you important information to give to your DBAs.

NewRelic is is great for monitoring production as well as test environments.

Flood integrates with DynaTrace, AppDynamics and NewRelic.

Other noteworthy tools are the Microsoft System Center Operations Manager, the Oracle Enterprise Manager, and BlueStripe FactFinder.

Stubbing

Service virtualisation involves creating simpler versions of existing systems and services that are sufficiently realistic as to be able to replace the original component for testing purposes.

A stub is that part that replaces a complicated component that is not within scope. It's a "dumber" version that responds to requests enough to allow you to go on with your load testing without actually requiring that component.

Let's say your application involves customers entering their credit card information upon checkout. That data is then sent to a payment gateway that then sends a message to your app to confirm receipt of payment. Often this payment gateway is outsourced to a third party, but you still want to test that your application saves the order information in your database and shows a "Thank you for your order" message after receiving payment confirmation.

It's often not feasible to include your third party payment provider in your load testing, unless you want to actually be using real credit card numbers. Instead, what you can do is create a stub that will perform this function.

You likely don't need "real" data, so the stub could take the input of a credit card number and send back a "Payment accepted; order number 123456" as the output to your application, allowing your load tests to continue without the payment gateway.

There are much more complex and full-featured enterprise solutions for this in the market, such as Tricentis Tosca Orchestrated Service Virtualisation (OSV). However one good, open-source tool I've found that is easy to set up and not too well-known is Mountebank.

If you are willing to put the time into creating a stub, you can drastically reduce the amount of resources you need to set up an environment and isolate components. Reducing variables in your test allows you to more quickly determine where performance bottlenecks lie.

Test scenarios

Choosing your test scenarios means deciding which situation is most likely to yield the data that you require. Employing several different types of scenarios will give you a greater understanding of your application's capabilities. You should feel free to create your own scenarios that are uniquely tailored to your requirements, but here are some common scenarios to start out with. Take the number of users and durations mentioned as guidelines and not rules.

Shakeout Test

I often see customers testing out their new load testing script by running it with 10,000 users. Such an approach exposes them to unnecessary costs in the provisioning of the machines for a test that might fail, which it often does.

Test out your scripts with a single user on your local machine. If that works, try it with two users. Then try it with ten users, with each one doing multiple iterations of the script. If that works, then you can consider doing a shakeout test on the cloud.

A shakeout test is a low-load test that is intended to be a quick check to see that the script, the environment, and the entire set-up, are working as expected. Depending on the application, a starting shakeout test could consist of anywhere from 1 to 100 users for about 10 minutes. The goal of a shakeout test is not to expose performance bottlenecks; it is a chance to verify that:

- the scripts are hitting the functionalities that were agreed upon with a low error rate
- the environment is fully integrated and functional
- the server monitoring is operational
- the right people for the involved application teams are on board and watching the test

It's a rehearsal before the real tests begin.

Depending on your comfort level and the responsiveness of the application, you can increase the number of users gradually until you're ready to ramp it up.

After shakeout tests are successfully executed (and no major errors are discovered), it's time to move on to other test scenarios.

Peak Load Test

The peak load test involves the simulation of the number of users that you expect to see on your application in production during your busiest times. Unlike shakeout tests, it is likely to give you valuable information about performance bottlenecks.

The number of users for a peak load test will vary, but in general its duration will be about 30 minutes to 1 hour.

Soak Test

Instead of simulating your busiest time in terms of traffic, like a peak load test, a soak test simulates the effect of a lower but sustained load level. This would be the number of users on your application over several hours.

A soak test will typically involve fewer users than a peak load test, but it will usually run from 3 hours to a few days. The goal of a soak test is to see if there is any performance degradation in the application performance over longer periods of time. A common finding from soak tests is that the application has a memory leak, which causes response times to degrade over time as the servers begin to struggle and become more sluggish. This is not something that is always apparent during the quicker peak load tests.

Stress Test

A stress test is usually done after at least a peak load test, and maybe even a soak test, has successfully passed according to the requirements. It involves subjecting the application to more load than you really expect it to ever need. The goal of a stress test is to determine the bounds of an application's capacity, with the ideal result being that it can support far more load than it needs to right now.

One good way of doing a stress test is doing a stepped load, where users ramp up to the expected peak load level, stay there for maybe 30 minutes, ramp up to another 100 or so users, sustain that for another 30 minutes, and so on until the application fails. Application failure can be judged to occur at the point that it no longer meets the nonfunctional requirements. The number of users that it comfortably maintained while meeting the requirements describes the outer bounds of its capacity.

Failover/Resiliency Test

This type of test scenario assumes the worst: that one or some of your application servers have failed and are unavailable. The point of this test is to determine how gracefully your application recovers and how resilient it is to unexpected failures.

A typical use case for this is for two application servers that share the user load. Apply load as normal, at peak load level or lower, and take note of the number of connections on each server. Each should be shouldering around half of the load. Then, shut down one of the application servers on purpose. The number of users should halve temporarily as the

system struggles to recover, and the users connected to the failed server hopefully see a nice error page asking them to try again (you can check for this in your scripts) rather than some unfriendly error page. After a few minutes, those users should be redirected to the one functioning server, the users should begin to be able to carry out their tasks again, and the number of connections on the one server should match the number of connections at the start of the test before the shutdown.

Unlike other types of tests, the error rate is less important in the failover test— having a server shut down in the middle of a test is going to produce errors for even the most hardy application. The test is how well and how quickly your system recovers.

For those wanting to go even further and test other catastrophic events, I really like Netflix's Simian Army approach, which conjures up the image of monkeys being set loose in a server room (basically a DevOps engineer's nightmare). The Chaos Monkey, for instance, randomly shuts down a node at random.

A note about concurrency

So far in this book, I've talked a lot about the number of users as if it were a measure of throughput (how much load is being generated), but that's actually a bit of an oversimplification.

More users doesn't necessarily mean more load. For example, 1000 users could send a combined total of 100 requests per minute, while 100 users could send 1000 requests in the same amount of time. A user that clicks every link on the page and trigger requests to multiple servers will have a different load profile to a user that just navigates to one page and refreshes it.

So there's a missing variable here, and that's the number of requests per second, which is the more accurate measure of test throughput. Most of the time, when people talk about user concurrency, what they're actually looking for is a way to express how much load they want to apply.

If this is the case, the savvy load tester (that's you) can reduce the number of users and increase the throughput of each user. This is because most load testing tools require more resources to increase the number of users than to increase the throughput. Reducing the

number of users while increasing throughput will maintain the expected load on the server while reducing the amount of machines that need to be provisioned (and paid for).

However, there are some situations where the number of users actually does matter.

User Concurrency Test

This is one situation in which using fewer users with increased throughput would not be sufficient. This tests how the system handles a certain number of users that log in and periodically refresh a page in order to maintain a connection. This is a specific type of scenario that will require a different script to execute, since it's less about generating load through transactions and more about just maintaining the user sessions on the server. This type of script can also be included as part of the testing suite for the other scenarios, but it can be useful to run this independently to measure just the effect of the concurrent user sessions on the server.

In the next section we'll actually begin scripting your load tests.

Scripting an API load test

Tool selection

The most important consideration for choosing a load testing tool to use is of course whether they support the protocols that you want to use. Luckily, API protocols tend to be supported by most tools due to their popularity, so this will be less of an issue. The second most important consideration is what resources you have available.

By resources, I mean the technical expertise and experience of the people on the testing team. Would your team be more comfortable with a nice UI, or do you feel at home coding away in a text editor or IDE? How much time do you have available to learn any tools that may be new to you?

A short proof of concept for the most promising tool should be carried out before building an extensive suite of scripts, and this should entail scripting a small sample of the requests. A proof of concept will show whether the tool is appropriate for the job.

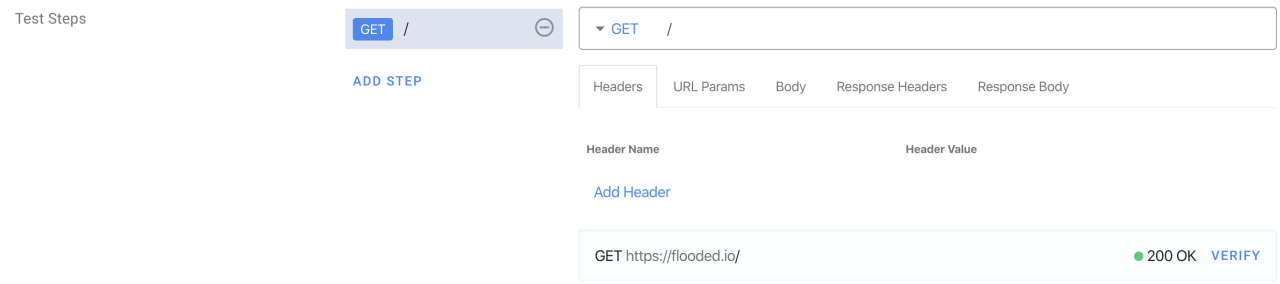
Let me run through some of the most popular API load testing tools, starting from the tools that require less technical experience and going to those that require more.

Test Builder

A good place to start for those with no coding experience is a tool with a user interface. I'm going to use Flood's own Test Builder as an example, just because that's the one I've had most experience with.

The **Test Builder** is an interface built on top of JMeter that gives less experienced load testers a way to get started quickly. There's nothing to install, there's no code to write, and the tool actually creates a JMeter script for you in the background, so you won't get stuck with a script that can't be used anywhere else.

To start, you enter the domain you want to test (in this example, I've used https://flooded.io) and then add a step for each request that you'd like to send.



An advantage of this method is that running through Flood means you don't have to worry about provisioning load generators or monitoring them, because Flood does that for you. It will also give you results for your test and is a great way to get up and running in a few minutes without much preparation.

One disadvantage is that it's not free— Flood is a paid service with a free trial, so if you're looking to go completely open-source, this isn't for you. Another thing to keep in mind is that the Test Builder is meant for simpler scripts; you don't get as much control over parameterisation or think time like you would for other tools.

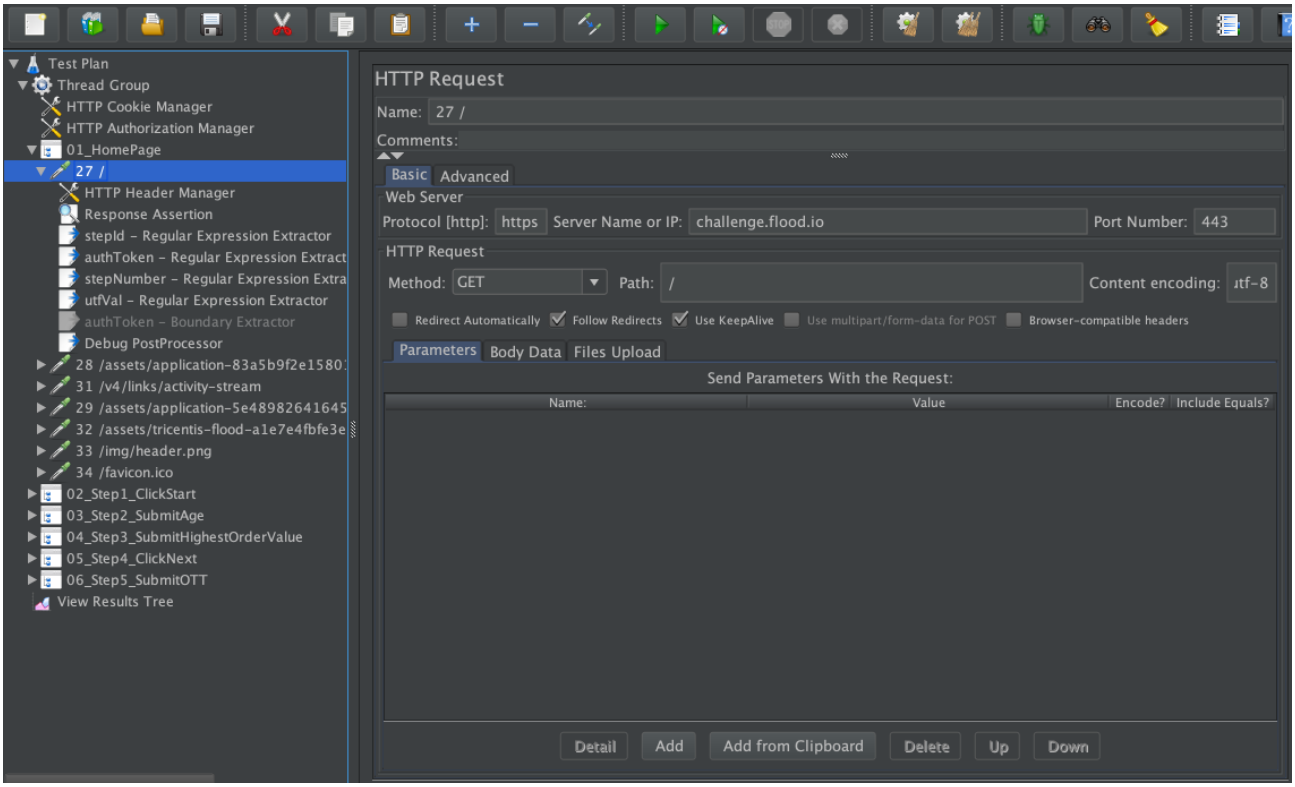
Here's a tutorial on how to use Test Builder with Flood.

JMeter

JMeter is an extremely popular open-source load testing tool, and for good reason. It's got a solid history of being able to deliver results and is an industry standard.

To start with JMeter load testing, you'll need to make sure you've downloaded Java as well as JMeter itself. Unlike the Test Builder, where you did your scripting in a web interface, you'll be doing your scripting within the JMeter program.

One big reason to use JMeter is its UI. While not the best looking, it is relatively easy to use. It's a step above the Test Builder in complexity, because there's still a learning curve here, but it's still a step below other tools where scripts are written entirely in code.



You can use JMeter to create robust load testing scripts without coding experience, but your JMeter load testing script can also include code in the form of Beanshell or Groovy post- and pre-processors. This flexibility means you can do a lot more with JMeter than you can with Test Builder alone.

Because JMeter is so popular, it's very easy to get support online. Regardless of what you're trying to do in JMeter, typing "JMeter load testing" in the search engine is likely to return a lot of results for you to learn from. It is also very well-supported by the community: the JMeter project on GitHub boasts 15,722 commits in the 20 years it's been around, and the number of custom plugins for JMeter load testing seems to increase every day.

Gatling

Gatling is another big player in the load testing tool space. Gatling load testing may not be as popular as JMeter load testing, but it definitely has its own share of avid fans, especially among developers.

The most divisive characteristic of Gatling is that it has no UI. Instead, you'll write your Gatling load testing scripts in a text editor using pure code like this:

```
import scala.concurrent.duration._

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

class gatling3newscript extends Simulation {
  // Optional, Tricentis Flood will pass in threads, rampup and duration properties from UI
  val threads   = Integer.getInteger("threads",  1000)
  val rampup    = Integer.getInteger("rampup",   60).toLong
  val duration  = Integer.getInteger("duration", 300).toLong

  val httpConf = http.baseUrl("https://flooded.io/")

  val scn = scenario("scenario")
    .during(duration seconds) {

      exec(
        http("Flooded - Home")
          .get("/")
          .check(status.is(200))
      )
      .pause(2)

      .exec(
        http("Flooded - Search")
          .post("/random?f=test123")
          .check(status.is(200))
      )

      .pause(1000 milliseconds, 3000 milliseconds)
    }

  setUp(scn.inject(rampUsers(threads) during (rampup seconds))).protocols(httpConf)
}
```

This may turn off some people, but others may be charmed by Gatling's easy, simple approach to load testing. Gatling is written in Scala, which is a relatively user-friendly language, and it's a lot easier to hook up to continuous integration tools rather than other tools that require an interface to edit a script.

Gatling does have a smaller community behind it, so while there is still quite a bit of training material and plugins for Gatling, it can't match the information that's out there for JMeter.

One of the big advantages to Gatling load testing is that it is more efficient than JMeter for large-scale load tests. Unlike JMeter, which starts up a thread for every virtual user that needs to be run, Gatling uses a different structure that allows it to run more than one user per thread, minimising the total number of threads used. If you're looking at a test that you hope to scale up to more than 100,000 users, you should consider using Gatling.

Other tools

There are a lot of other commercial tools out there. In this book, I'm going to focus more on JMeter and Gatling because they are open-source and free, aside from being extremely

powerful load testing tools in their own right. However, if you have a bit more of a budget or possibly already have other tool licenses, here's a round-up of other good load testing tools:

Initially created by Mercury, sold to HP and now owned by MicroFocus, LoadRunner has been a load testing staple for decades. While it is notoriously expensive, LoadRunner does have excellent support for a huge swathe of protocols, and it does most things very well. It also comes with the benefit of being able to hook up to the extremely popular Application Lifecycle Management suite.

Neotys NeoLoad is another great commercial alternative, and one that happens to be really good for new load testers. I have a soft spot for NeoLoad as it was the first load testing tool that I scripted with. Its drag-and-drop interface is somewhat similar to JMeter, but the whole package is just generally easy to use.

MicroFocus' Silk Performer has also been a contender in recent years, and its customer support has been stellar for me. However, its future is unclear now that MicroFocus has bought the load testing darling, LoadRunner.

JMeter scripting

In this section you'll learn how to get started with JMeter load testing. We'll go through the steps for creating your first basic JMeter load testing script here, but you can also check your work against this sample script.

First, download and install JMeter as well as Java, which it requires. Then go to the /bin directory where you installed JMeter and double-click either jmeter.bat (Windows) or just jmeter (Linux). The JMeter window will open with your very first project, and the only thing you'll see is an empty Test Plan.

The most basic JMeter load testing script will require these elements:

- Thread Group

- Transaction Controller

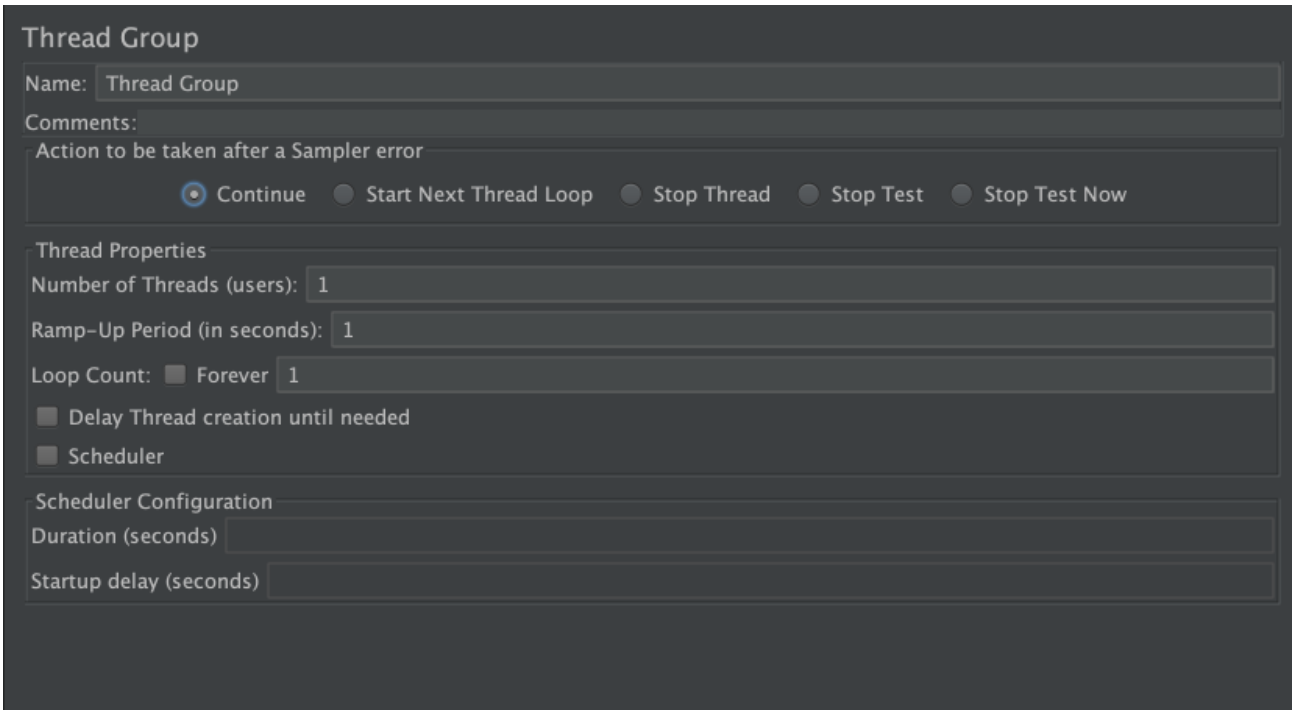
- HTTP Request Sampler

- View Results Tree Listener

- Uniform Random Timer

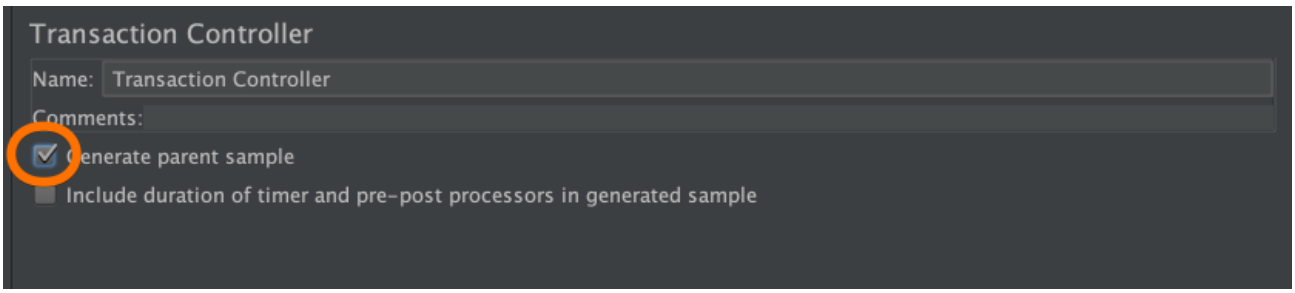
There are alternatives within JMeter for each of these, but for simplicity's sake, I've included the most basic versions.

A **Thread Group** in JMeter-speak is a set of instructions that you want each user to follow. To add it, right click on Test Plan > Add > Threads > Thread Group. You should get a child element under Test Plan that looks like this when you click on it:



These are the default values for a Thread Group. Some of these values need to be changed, which we'll cover later. For now, let's focus on getting a single user to hit one endpoint.

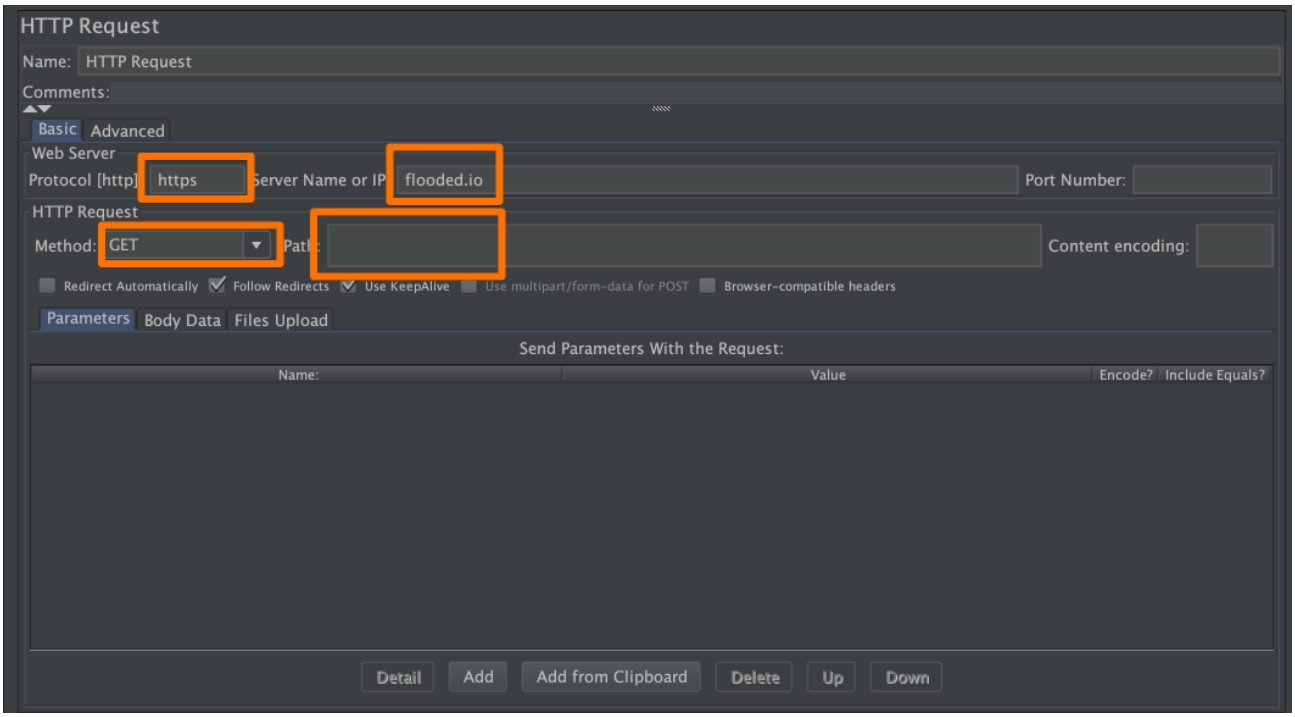
A **Transaction Controller** is a way to organise a group of requests that logically belong together. This has reporting ramifications. For example, you might have 10 requests for resources for the same page. Rather than reporting on each of those separately, it might make more sense to report the total response time for the entire page. To add it, right click on Thread Group > Add > Logic Controller > Transaction Controller.



The only setting I would recommend changing is “Generate parent sample”. The Transaction Controller comes with this unticked by default, but I would tick it so that JMeter reports on the metrics for the transaction, not the individual requests.

The **HTTP Request Sampler** is the heart of our basic test plan. This is where JMeter will actually send the request to your API. To add it, right click on the Transaction Controller > Add > Sampler > HTTP Request. A “sample” is a request that has been sent, and a “sampler” is what JMeter calls these elements that allow you to build requests.

Click on the the HTTP Request Sampler and fill in the following fields:



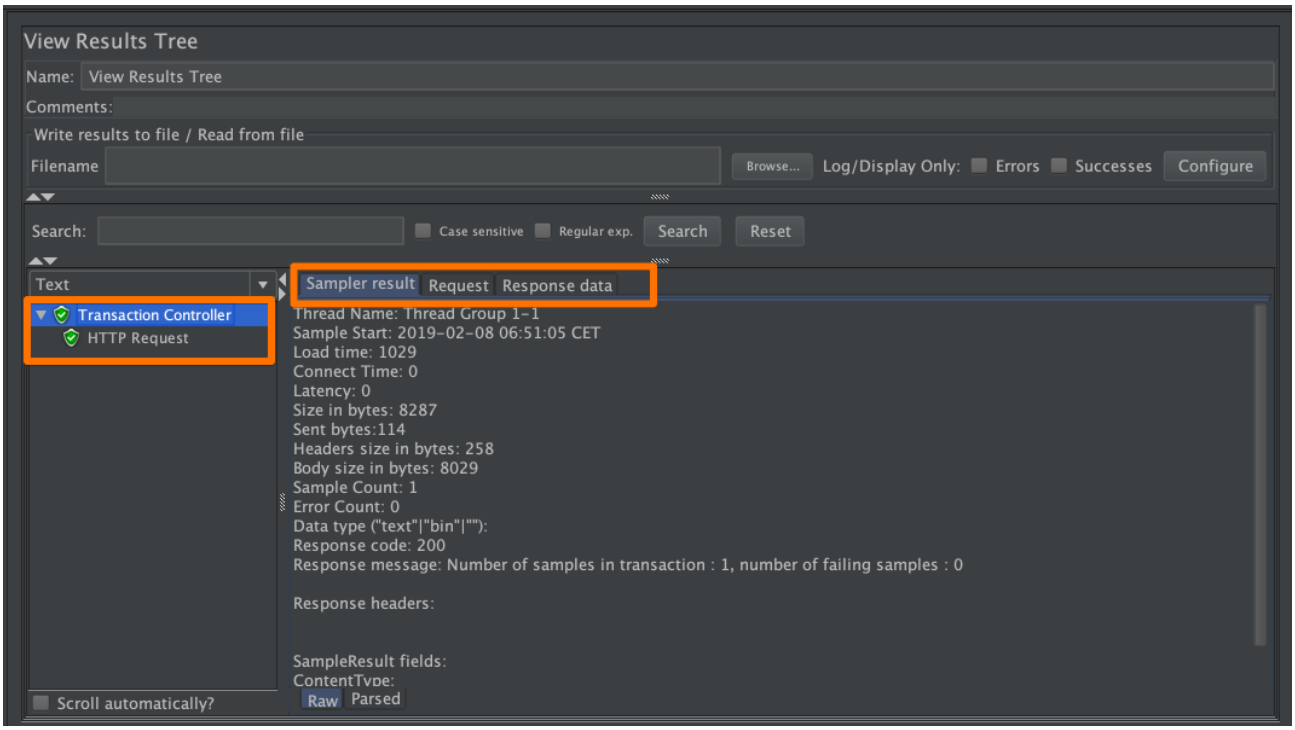
In this example I’m just sending a GET request to https://flooded.io.

To view results while we’re running the test, we’ll first need to add a listener. A listener captures requests sent, as well as their corresponding responses, for us to view later. There

are quite a few listeners, but the most basic one that you'll need is the **View Results Tree Listener.** It records all the details of the request and response pair and is best for debugging. Go ahead and right click on Test Plan > Add > Listener > View Results Tree.

It will look pretty empty until we run a test, so let's do that now. On the JMeter toolbar, you'll see two buttons that look like a play button.

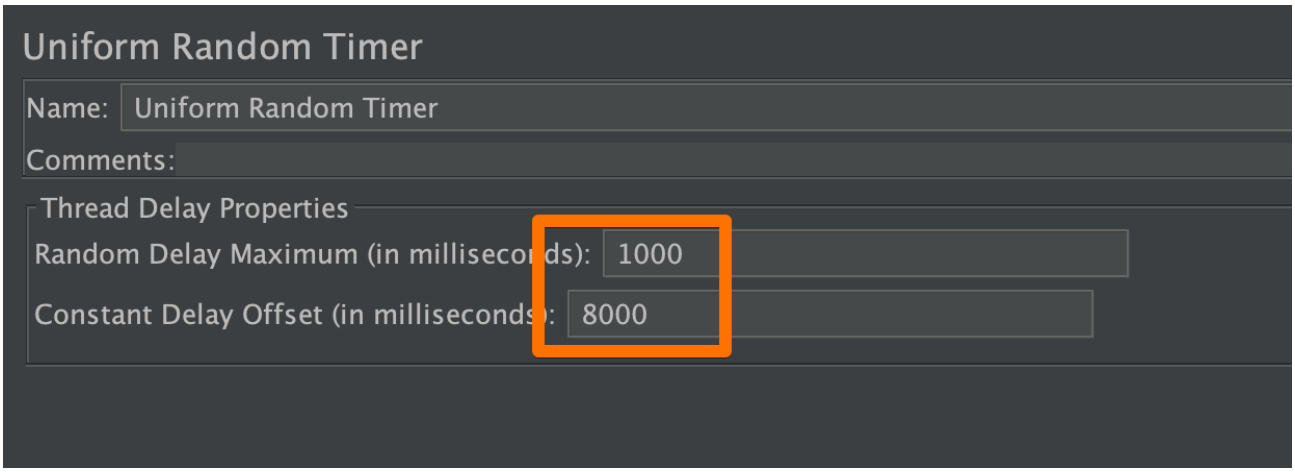
Click on the first one, the solid green play button. Now click on the View Results Tree listener. It may take a few seconds to get a response, but then your listener should look like this:



You'll see that our request, "HTTP Request", under the Transaction Controller, has been sent. By default, we're on the Sampler result tab, which will show us some quick metrics about our request. Clicking on the Request and Response data tabs will allow us to see the raw request and response for that request as well.

The last basic element is the timer. For this tutorial, I've chosen the **Uniform Random Timer** because it's easy to understand. A timer in JMeter is an artificially introduced delay. These delays space out the requests to make them more realistic— more on that later. For now, go

ahead and add the timer by right clicking on Test Plan > Add > Timer > Uniform Random Timer.



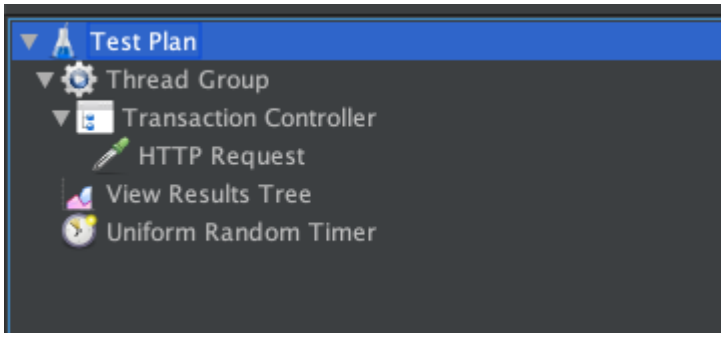
Enter values in the Thread Delay Properties. A good place to start is by having a Constant Delay Offset of 10,000 ms and a Random Delay Maximum of 1,000 ms. This means that the delay will be at least 10,000 ms long, plus a variable amount of up ot 1,000 ms. JMeter automatically removes timers when calculating the response time.

Now let's click on that second green play button on the toolbar.

The first green play button runs the script including the timers, and the second one runs the script without any timers. For the most part, while we're scripting and running just one user, we'll want to use the second play button to save a little bit of time. Go ahead and play around with those now. Regardless of which one you use, the response time JMeter reports in the View Results Tree Listener will be the actual response time of the request, not including the timer delays.

Another thing to note is that timers are always applied before the requests, not after them.

If you've been following along, your test plan should look like this:

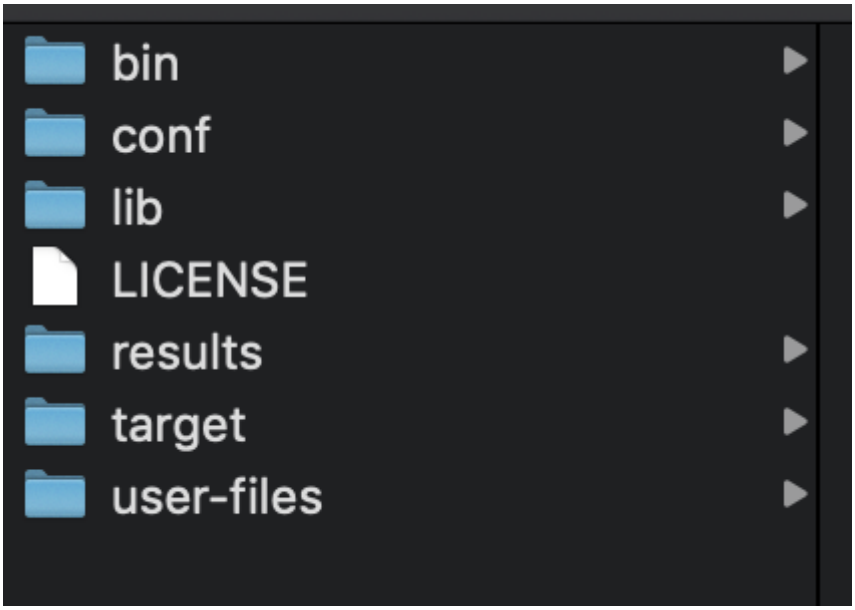


One final note about elements in JMeter before we move on to Gatling: where you put the element matters. For example, since we put the timer as a child of the Test Plan, we're applying the timer to all samplers in that context. If we wanted to apply it only to one HTTP request, we'd need to put it as a child element of that HTTP Request Sampler.

Gatling scripting

First, download Gatling. You'll want to get the standalone tool for this basic tutorial. You'll also need Java to run Gatling.

Unzip Gatling. You'll see that it comes with a few directories:



bin contains Gatling's execution engine and a recorder that will help you create scripts

conf contains files you'll modify to change the default Gatling configuration settings

lib contains extra files you may require to run your simulations

results will contain data from previous test runs

target will contain simulations that you've successfully compiled

user-files contains your simulations and scala files

Gatling comes with a sample script by default, which you can find in /user-files/simulations/computerdatabase/BasicSimulation.scala, but let's take a step back and create our own Gatling simulation to run. Create a folder in /user-files/simulations called gatlingsample and create a file within that folder called BasicSimulation.scala. You can download a copy of this script to follow along.

Here's the most basic version of the script:

```
package gatlingsample
import io.gatling.core.Predef._
import io.gatling.http.Predef._
import scala.concurrent.duration._
class SampleSimulation extends Simulation {
  val httpProtocol = http.baseUrl("https://flooded.io") // This sets your
  val scn = scenario("Basic") // A scenario is a chain of requests and pau
     .exec(http("01_Home") // This sets the transaction name
       .get("/")) // method and relative path to retrieve
 .check(status.is(200), substring("Smooth Scaling").exists) // Verify that
     .pause(7) // think time
  setUp(scn.inject(atOnceUsers(1)).protocols(httpProtocol))
}
```

This script contains one transaction called "01_Home" with the scenario "Basic" that will start one user and make a single GET request for https://flooded.io. It will then look at the data returned, check for the text "Smooth Scaling", and then pause for 7 seconds before finishing.

To run this script, open up your terminal. Unlike JMeter, Gatling doesn't have a UI, so you'll need to get comfortable with the command line in order to use it.

Change directory to Gatling's bin folder and type ./gatling.sh (Unix) or ./gatling.bat (Windows). Gatling will compile all simulations it finds. Once that's done, it'll then ask you which one you'd like to run.

```
^Crevinejo:bin nvanderhoeven$ ./gatling.sh
GATLING_HOME is set to /Users/nvanderhoeven/gatling/gatling-charts-highcharts-bundle-3
.1.3
Choose a simulation number:
     [0] computerdatabase.BasicSimulation
     [1] computerdatabase.advanced.AdvancedSimulationStep01
     [2] computerdatabase.advanced.AdvancedSimulationStep02
     [3] computerdatabase.advanced.AdvancedSimulationStep03
     [4] computerdatabase.advanced.AdvancedSimulationStep04
     [5] computerdatabase.advanced.AdvancedSimulationStep05
     [6] gatlingsample.SampleSimulation
6
```

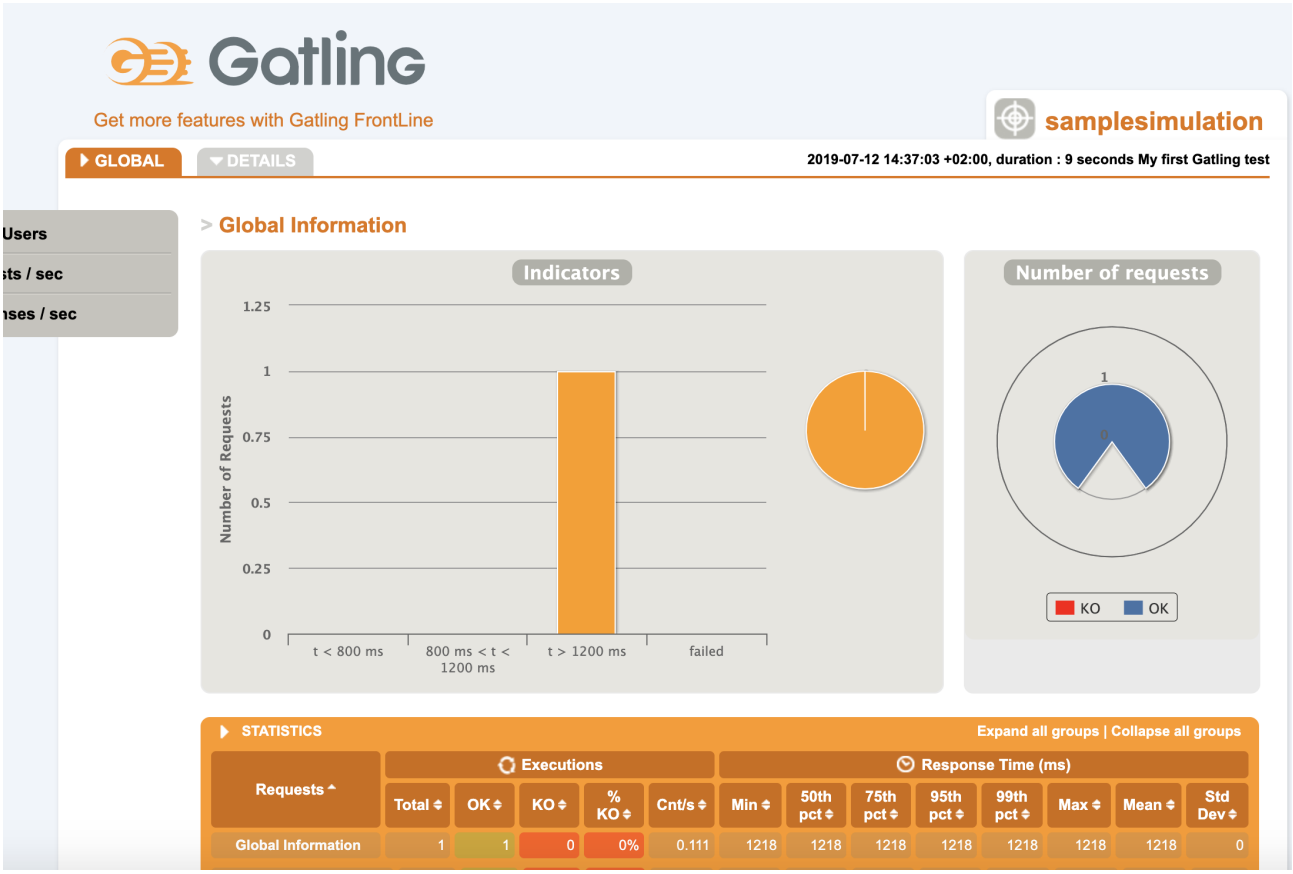
Type 6 and hit the ENTER key. Enter an optional test description and hit ENTER, and Gatling will run the simulation. After the test has finished, you'll see a cursory report:

```
Generating reports...

================================================================================
---- Global Information --------------------------------------------------------
> request count                                          1 (OK=1      KO=0     )
> min response time                                   1218 (OK=1218   KO=-     )
> max response time                                   1218 (OK=1218   KO=-     )
> mean response time                                  1218 (OK=1218   KO=-     )
> std deviation                                          0 (OK=0      KO=-     )
> response time 50th percentile                       1218 (OK=1218   KO=-     )
> response time 75th percentile                       1218 (OK=1218   KO=-     )
> response time 95th percentile                       1218 (OK=1218   KO=-     )
> response time 99th percentile                       1218 (OK=1218   KO=-     )
> mean requests/sec                                  0.111 (OK=0.111  KO=-     )
---- Response Time Distribution ------------------------------------------------
> t < 800 ms                                             0 (  0%)
> 800 ms < t < 1200 ms                                   0 (  0%)
> t > 1200 ms                                            1 (100%)
> failed                                                 0 (  0%)
================================================================================

Reports generated in 1s.
Please open the following file: /Users/nvanderhoeven/gatling/gatling-charts-highcharts
-bundle-3.1.3/results/samplesimulation-20190712123703370/index.html
revinejo:bin nvanderhoeven$
```

If you copy that URL into your browser, you'll see Gatling's standard HTML report, which you can use to find out more detailed information about your test run:



Each run will generate a new report.

That script only makes one request, however, so we'll need to instruct Gatling to iterate. In addition, the scenario so far only runs one user. Let's make some adjustments:

```
1  package gatlingsample
2  import io.gatling.core.Predef._
3  import io.gatling.http.Predef._
4  import scala.concurrent.duration._
5  class SampleSimulation extends Simulation {
6    val threads    = 10
7    val rampup     = 30
8    val duration   = 300
9    val httpProtocol = http.baseUrl("https://flooded.io") // This sets your
10   val scn = scenario("Basic") // A scenario is a chain of requests and pau
11   .during(duration seconds) {
12    exec(http("01_Home") // This sets the transaction name
13        .get("/") // method and relative path to retrieve
```

```
14          .check(status.is(200), substring("Smooth Scaling").exists) // Verif
15    )
16    .pause(7) // think time
17    }
18     setUp(scn.inject(rampUsers(threads) during (rampup seconds))).protocols(
19  }
```

This wraps everything in the scenario in a during {} loop, and you may notice that I've also changed the setUp scenario line so that it ramps up users to a certain number and then maintains that amount of users for the whole duration. In this script, I've hardcoded values for those parameters, but if you're running this script on Flood, you can do this to have the script take the values from the Flood UI:

```
1  // Optional, Tricentis Flood will pass in threads, rampup and duration pro
2    val threads   = Integer.getInteger("threads",  1000)
3    val rampup    = Integer.getInteger("rampup",   60).toLong
4    val duration  = Integer.getInteger("duration", 300).toLong
```

Download the final script here.

Making scripts more realistic

At this point, you should already have some load testing scripts that have been functionally shaken out. This means that you have executed the script with at least one user locally and verified on the backend that you're hitting the right components without errors. For now, your script is just an automated way of sending requests. What makes it a load test?

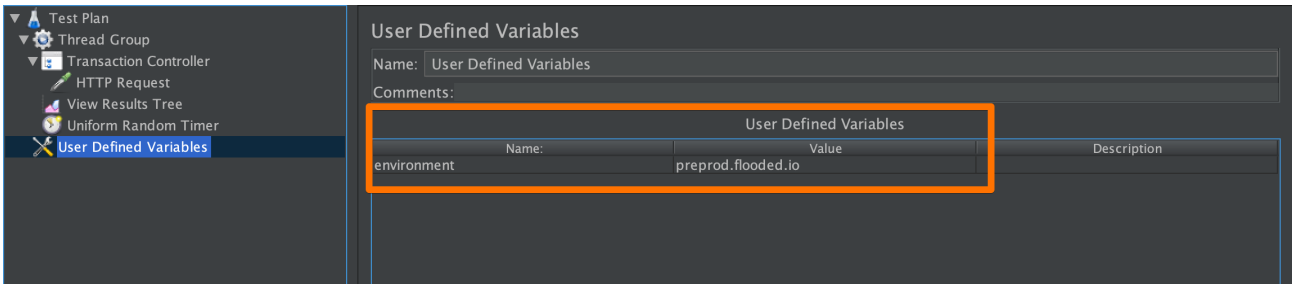
Parameterizing environments

If you have several environments that you might like to test with the same script, it would be a good idea to set up your script so that it's easy to switch between environments.

JMeter

In JMeter, you can accomplish this by creating a variable for your environment. Right click on Test Plan > Add > Config Element > User Defined Variables. Then, click Add and then set

the name and value for your variable.



You'll want the value of the variable to be the URL of the environment you currently want to test.

Then, create an HTTP Request Defaults config element (right click on Test Plan > Add > Config Element > HTTP Request Defaults). In this config element, you'll add ${environment} as the URL. That way, any requests with a blank field for the server name will take the value of the variable you created in User Defined Variables as the default domain.

The last step is to go ahead and delete the domain name of all HTTP requests so that the variable in the default is used.

This way, when you want to change the domain name you're testing, you only need to change it in one place: User Defined Variables.

Gatling

In Gatling, this is done by setting the base URL:

```
val httpProtocol = http.baseUrl("https://flooded.io")
```

Then, succeeding requests will only need to have the path:

```
exec(http("01_Home") // This sets the transaction name

 .get("/") // method and relative path to retrieve
```

To change the environment, all you'll need to change is the base URL.

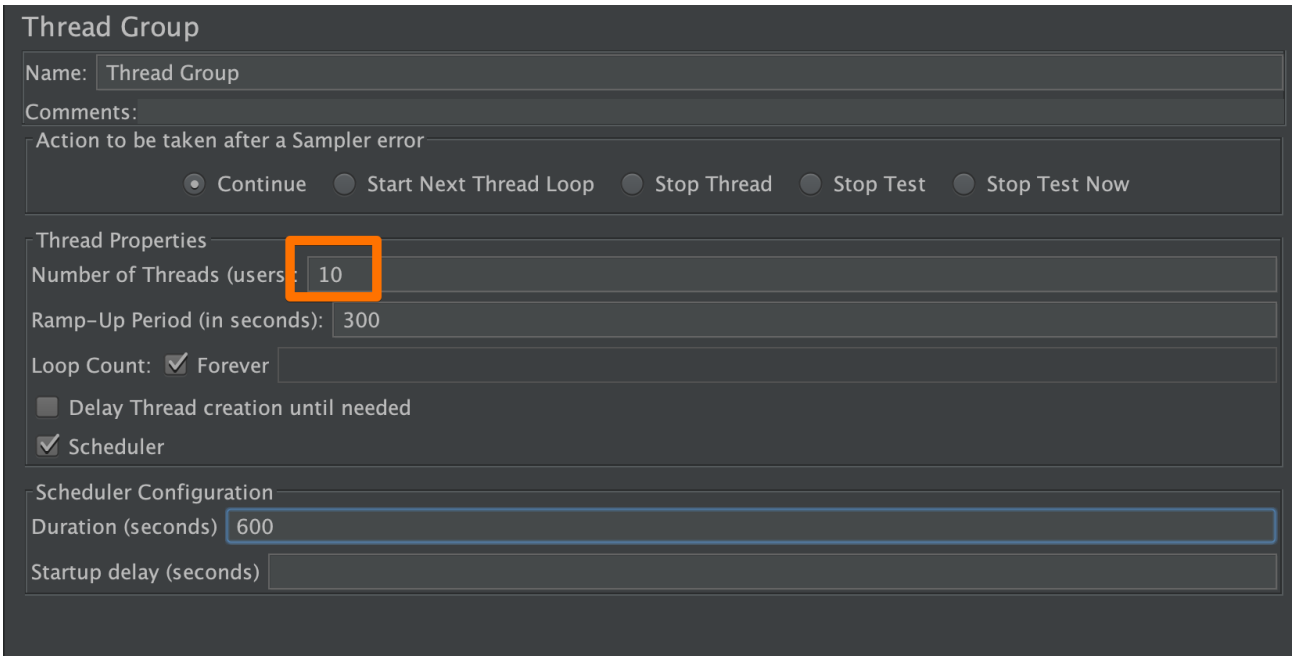
Increasing throughput

The most obvious way to turn a script into a load testing script is to increase throughput. This can actually be done in a few different ways.

Number of users or threads

This is the most obvious way to increase load. The more instances of your script running, the more requests are executed.

You can change this in JMeter by clicking on your Thread Group and changing the “Number of Threads (users)” field:



In Gatling, you can use the line setUp(scn.inject(rampUsers(threads) during (rampup seconds))).protocols(httpProtocol) in your script and replace the value of threads with the number that you’d like to run with.

Think time

Not including think time is a common mistake in load testing scripts. Think time is the amount of time that a user spends “thinking”— that is, the delay between requests. Firing off many requests immediately one after another may sound like a great way to apply some extra load on your server, but this can actually backfire. Not including think time in your scripts is resource-intensive because it causes the node you’re running the script from to work overtime to send the requests. Sometimes this can cause delays within the node as

it struggles to process each request before sending it to the server. This means that the node itself can become a bottleneck, causing some queuing of requests way before your requests are even sent to your servers. This would report high response times that don't necessarily reflect how your application servers handle the load.

To prevent this from happening, add think time to your scripts that reflects a user's normal wait time. This will space out the requests and lead to more accurate results.

However, it's important to note that decreasing think time will also increase throughput, because each thread will be able to send more requests if the time it has to wait decreases. The right balance of think time needs to be struck in order to mimic production behaviour.

In JMeter, this is done by using one of the many built in timers. An example of this can be found in the section for writing a basic JMeter script, but you can really use any of the timers in JMeter to suit your needs. Pay special attention to the context that the timer sits in: JMeter applies the timer to all elements at that level, so you only need to add one timer once. Run the script with pauses to verify that the think time is being applied as you expect.

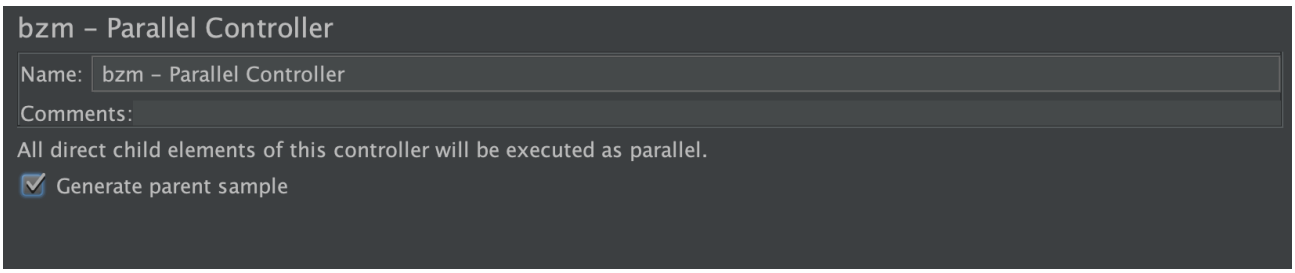
You can simulate think time in Gatling using .pause(1000 milliseconds, 3000 milliseconds) after the request you want to pause after. You can change the numbers to fit your scenario.

Concurrent requests

By default, requests are executed sequentially, as they are written in the script. By contrast, concurrent requests are executed in parallel— they are sent at the same time. In the same way that decreasing think time increases how many requests a user can make in a certain amount of time, using concurrent requests also increases throughput because the requests are made in batches.

Web browsers actually send some requests concurrently, such as embedded resources on a page. Determine how API calls are made in production: are they usually called one after another (sequential) or made at the same time (concurrent)? This will determine whether you incorporate concurrent requests in your script.

This can be accomplished in JMeter by using the Parallel Controller instead of the Transaction Controller. Right click on the Thread Group > Add > Logic Controller > Parallel Controller:



All requests you put underneath this parallel controller will then be executed concurrently.

Gatling uses .resources to accomplish the same thing. Here's an example from the Gatling Documentation:

```
http("Getting issues")
  .get("https://www.github.com/gatling/gatling/issues")
  .resources(
    http("api.js").get("https://collector-cdn.github.com/assets/api.js"),
    http("ga.js").get("https://ssl.google-analytics.com/ga.js")
  )
```

It's meant to be used to simulate the download of resources that are embedded into a page, but it will work for any request that you put in as a resource.

Setting test parameters

Test parameters are the main characteristics of the test and include the number of users, ramp up and duration.

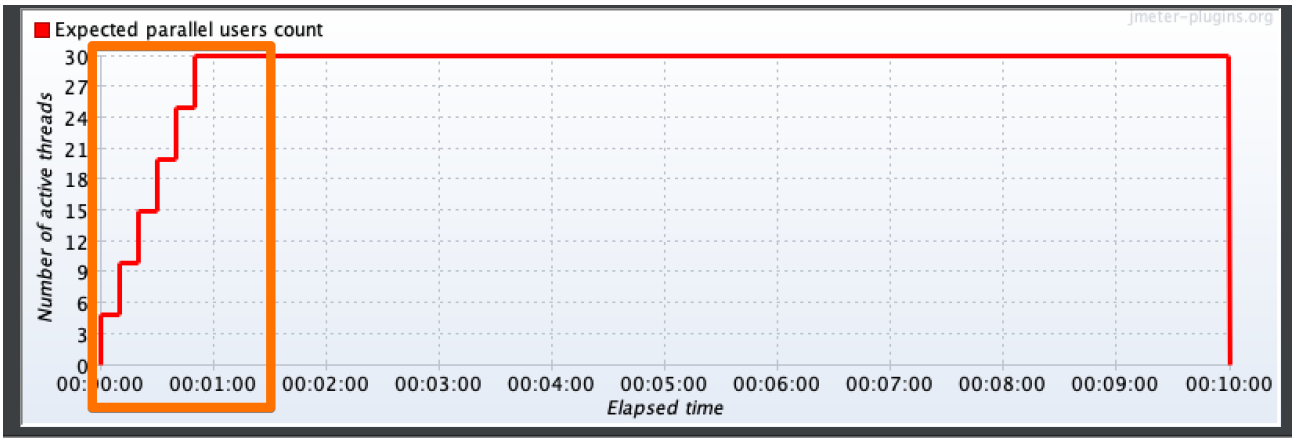
We already discussed the number of users in the previous section, but they also belong here as a key characteristic of the test.

Ramp up

In production, load on servers very rarely goes from 0 to 1000 in one second. Even for cases with a very definitive start time (such as an item going on sale at a particular time), the load

generally increases gradually. This gradual increase can be simulated by adding ramp up times in your script. A ramp up is the amount of time during which new users are added at staggered intervals.

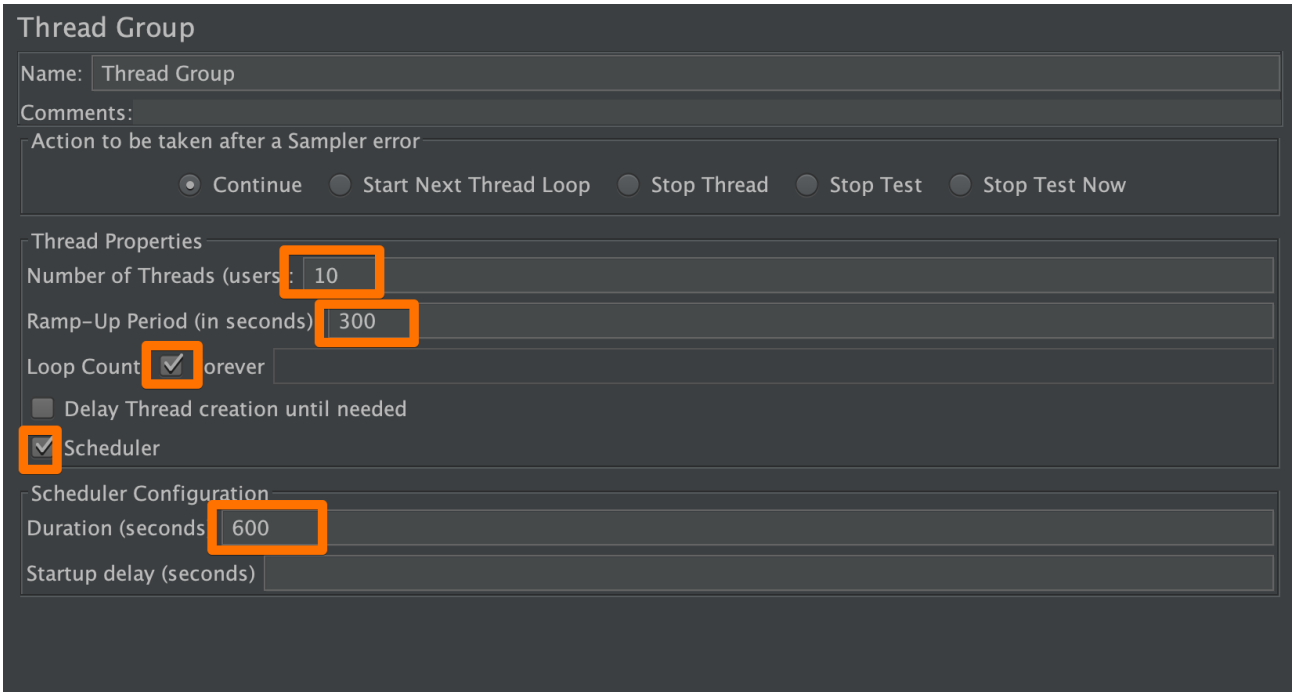
Here's an example of what that might look like:



Duration

Duration is how long the whole test actually lasts.

The simplest way to set these values in JMeter is to use the basic Thread Group. You'll need to change the fields highlighted below:



Note that in order to use the duration, you'll have to select "Forever" next to Loop Count and "Scheduler" as well as type in the number of seconds in the Duration field. Otherwise, you can also run the script according to the number of loops, or iterations, that you want executed. However, since the response times will vary, using the Loop Count as a proxy for duration will yield different results from test to test.

In Gatling, the threads and ramp up are set in the line setUp(scn.inject(rampUsers(threads) during (rampup seconds))).protocols(httpConf), where you replace the parameters threads and rampup with the values you want. You can set the duration by including this line of code:

```
val scn = scenario("scenario")

    .during(duration seconds) {

      exec(

 //Requests here

          )

      )

    }
```

and replacing duration.

Adding test data

Using the same data (such as user credentials) in your test may yield faster-than-average response times due to server caching. To avoid this, you should consider using a more diverse data set. Using a variety of data can also expose issues seen in production that arise due to data being in different states.

For example, a financial institution running a load test for mortgage origination might well get excellent response times when testing the same user on a "happy path": a user who has filled out all the information for a loan and has passed the identity checks. However, using

a larger set of users that is more representative of production may show that a user who does not fill out enough information for the identity checks experiences very slow response times as the application waits for the responses from the identity verification service in the background.

First, you'll need a CSV file (other formats are available, but this one is the most commonly used for this purpose). Here's an example of what this could look like:

username,password

user1,pass1

user2,pass2

user3,pass3

user4,pass4

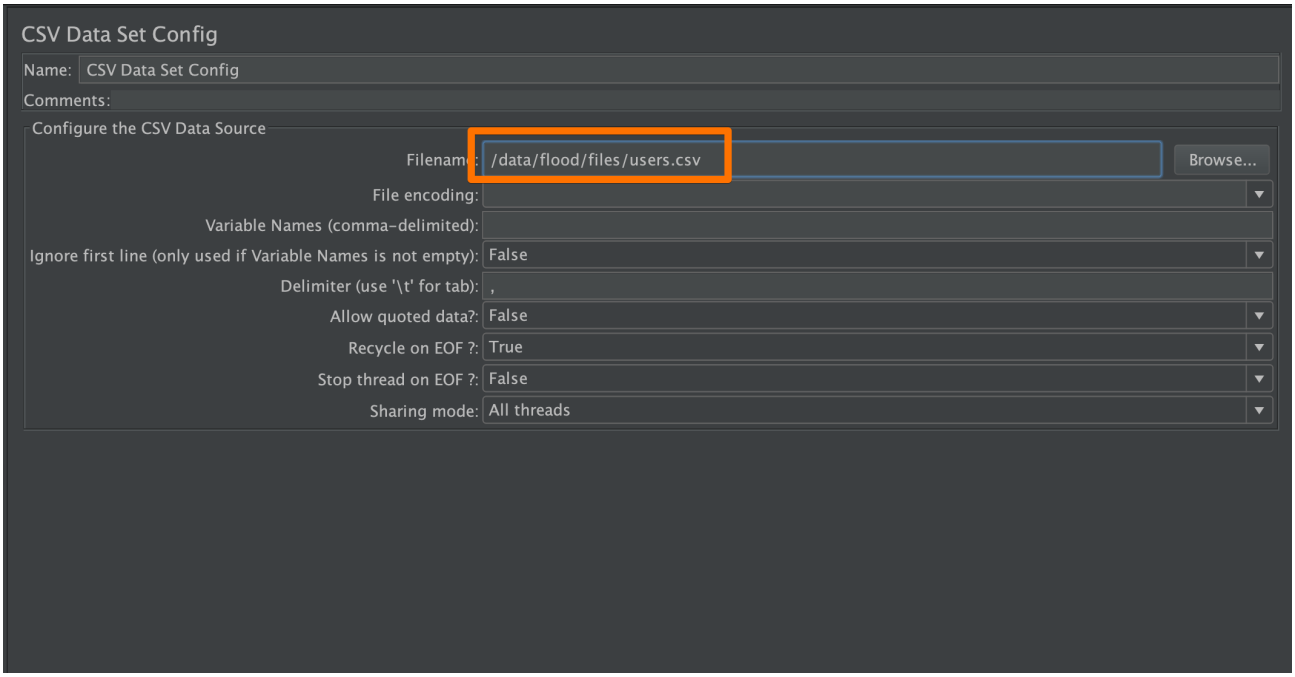
Save this file as users.csv.

Next, you'll need to modify your script to read these values.

JMeter

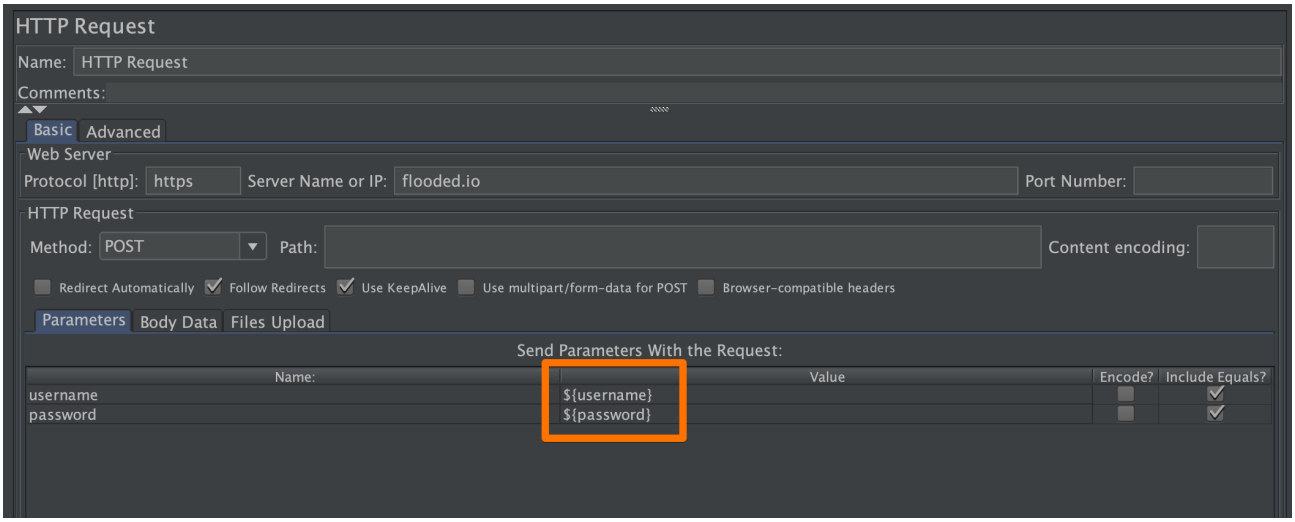
In JMeter, you can add a CSV Data Set Config element to add this functionality to your scripts.

Click on the Test Plan element > Add > Config Element > CSV Data Set Config. Then click on the newly created element. The only thing you'll need to change is the filename to include the path to the CSV file you've just created.

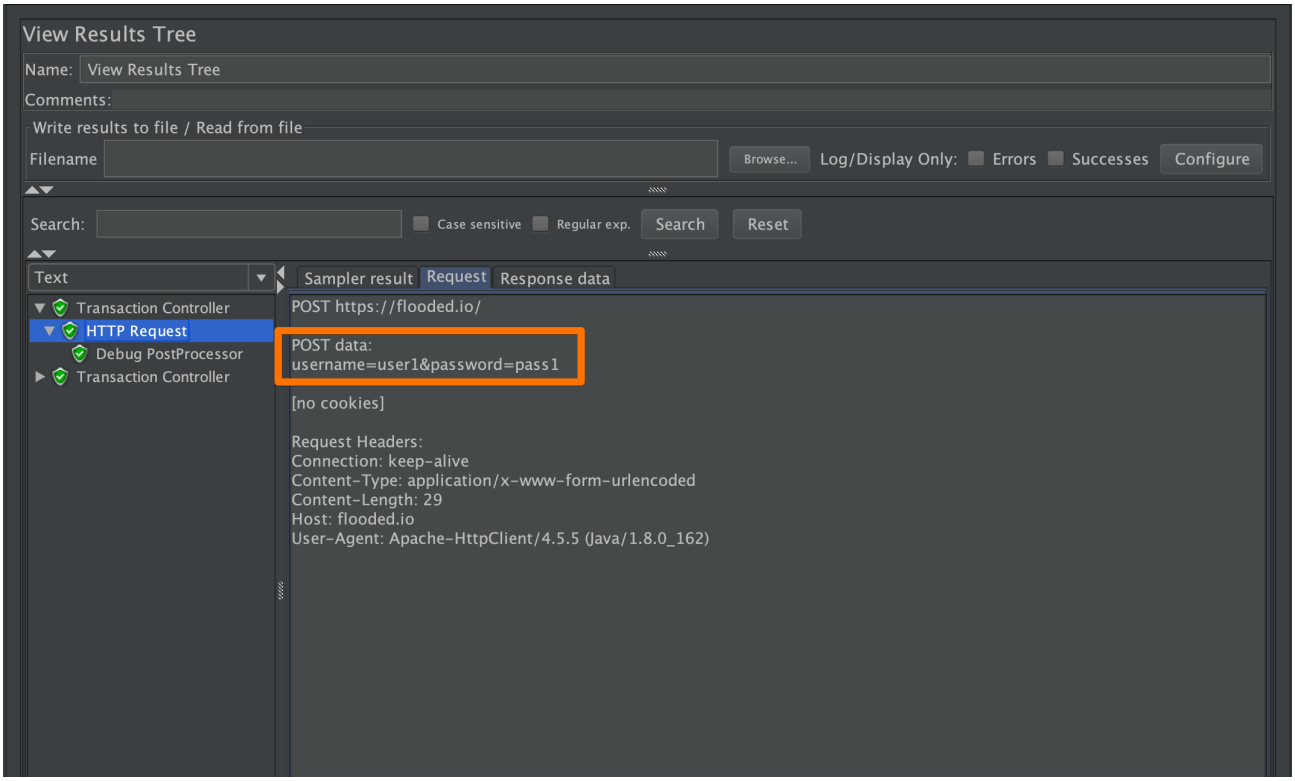


If you don't put anything in the "Variable Names (comma-delimited)" field, JMeter will by default use the first line of the CSV file as the variable names.

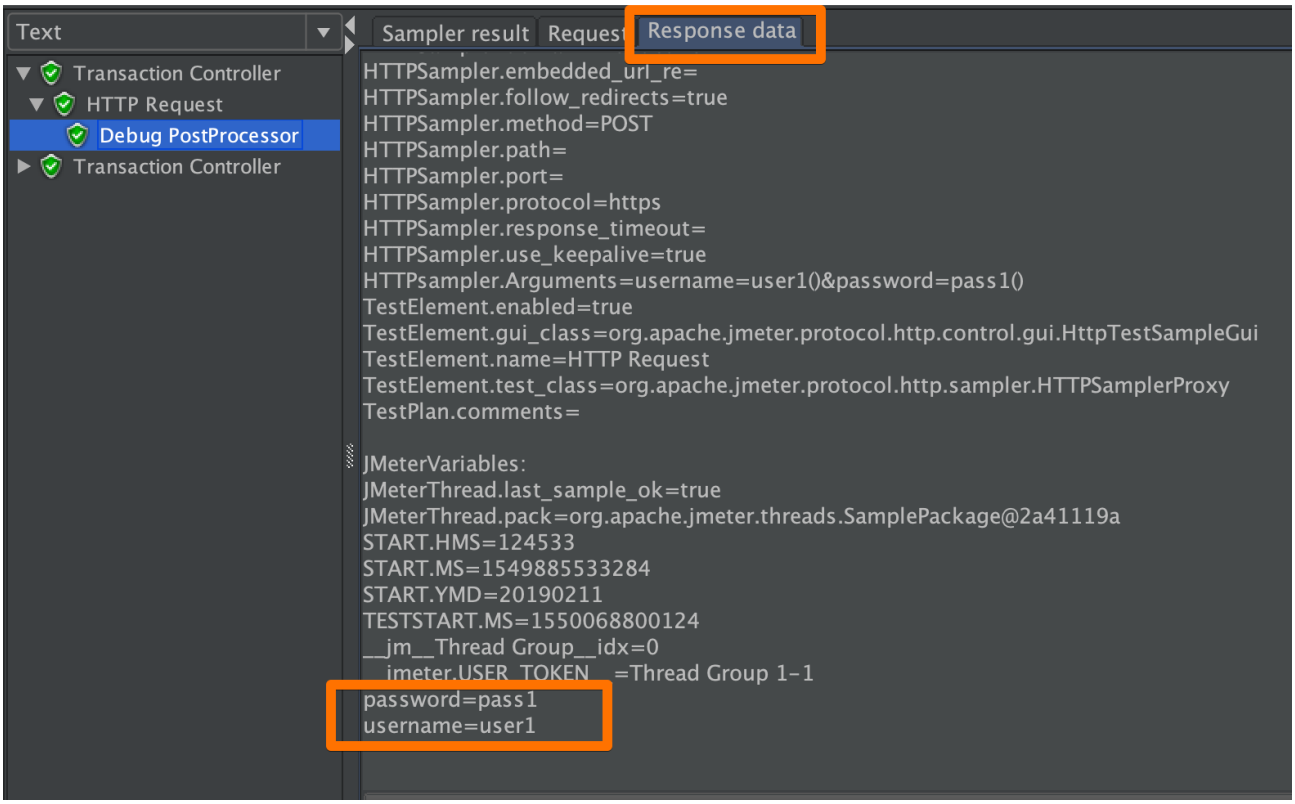
Then go to the request where you want to use those values and replace the hardcoded values with the variables you want to use— in this case, ${username} and ${password}. Note that you wouldn't really pass these in the clear like this in your application; this is just a simple example to get you started.



When you run your script, you can confirm whether the request has picked up these values by clicking on the View Results Tree listener, clicking on the HTTP request and clicking the Request tab. You can see below that the script has automatically used the values from the first line of our users.csv file.



If you have a lot of test data that you're using this way, one way to see at a glance what the current values are is to add a Debug PostProcessor by clicking on the sampler (HTTP Request) > Add > post Processors > Debug PostProcessor. Next time that you run your test, you can click on this post processor and go to the Response data. It will have all the current values of the variables you're using. This can be a great way to debug between multiple iterations. Just remember to disable this while you're running your full-scale load test.



Gatling

We're going to use the same users.csv file that we created earlier, but in order for Gatling to recognise it, it will need to be in a particular folder. The default path for this folder can be viewed or changed in /conf/gatling.conf:

#data = user-files/data # Directory where data, such as feeder files and request bodies are located (for bundle packaging only)

It's worth it to check this as some later versions of Gatling may use user-files/resources instead. Assuming you have this set as above, create a data folder under user-files. Then transfer the CSV file into /user-files/data.

In Gatling, you'll need to first set up what's called a feeder, which is basically a way to get data from a file and bring that data into the script:

val csvFeeder = csv("users.csv").circular

Note that this line is for CSV files particularly, although there are other types of feeders that you can use with Gatling. If your users.csv is not in the default data directory, you will need

to include the filepath. For example, when running this on Flood you'll need to change this line to:

val csvFeeder = csv("/data/flood/files/user-files/data/users.csv").circular

But for now, since we're just running this locally, you can leave it as is.

The circular at the end of the line tells Gatling that when the script runs out data to consume (such as when you have five users and only four lines of data), it should go back to the beginning and re-use the same data.

Now, to use the data, you'll need to refer to it by the headers on the data file itself. In our case, it's username and password, so here's how we could use that data to pull in a line of values into a post request:

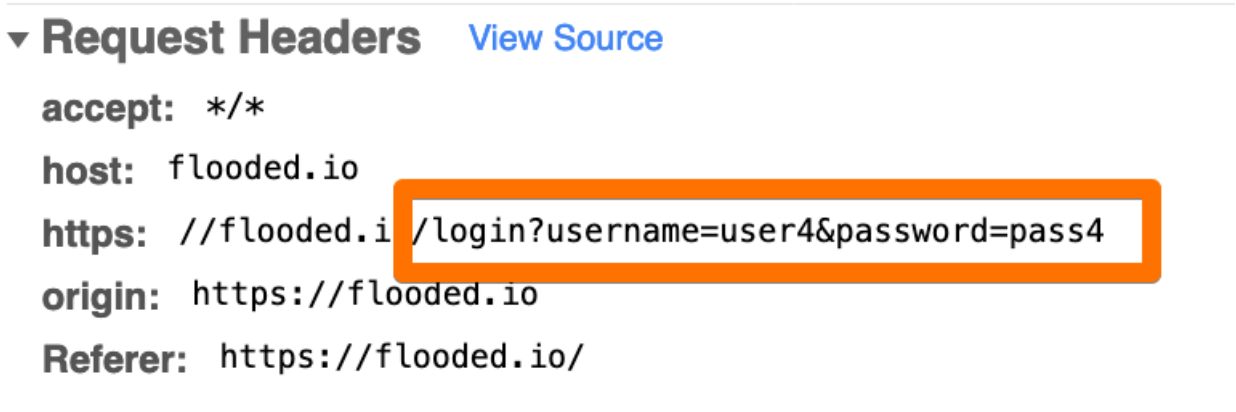
```
.feed(csvFeeder)

.exec(http("02_Login")

   .post("/login?username=${username}&password=${password}")

)
```

Of course, this is a simplified version of a login, but the principles remain the same. Here's what that looks like on Flood:



To follow along, you can download this sample Gatling script here.

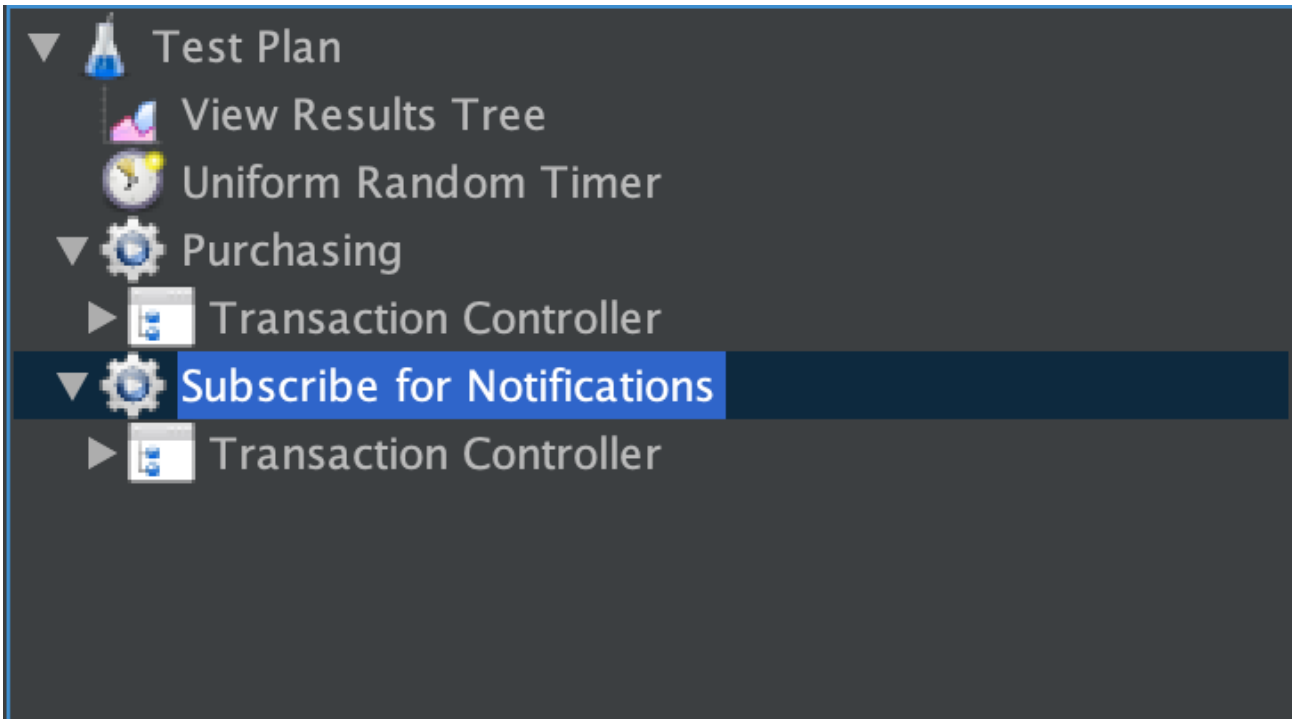
Using different user paths

Your basic script will likely have one business process that all your users follow. This could be, for example, a path that a user takes from landing on the home page to browsing a catalog of items for sale to actually adding the item to a cart and checking out. While this path actually exists in your application, one thing you might consider is alternative user paths.

For example, perhaps instead of browsing, a user searches for a particular item, finds that it is out of stock, and signs up to be notified when the item is back in stock. It might be worth thinking about scripting a separate flow to account for users like this, depending on how common this situation may be.

But how do you add that into your load testing?

In JMeter load testing, one common way to do this is by simply adding another thread group and then scripting another path inside that thread group to run alongside your first thread group.



An advantage of this is that you can set the number of users for each thread group separately, so you can control the ratio of the different paths that you decide to script.

Executing an API load test

Preparing for execution

You have requirements to measure your results against, a workload model based on historical data, and scripts that should simulate user journeys through your application as realistically as possible. You're ready to execute!

Here are some things you might want to do before you officially kick off a test.

Go through the Execution Checklist. If you don't already have one, now is a good time to create one. An Execution Checklist is just a list of steps that need to be followed anytime anyone kicks off a test and will include preparatory activities such as those listed below. This is handy even if you're the only one that will be running tests, because it's easy to forget something and skip a step, but it's especially important when there is a team of testers that may need to execute load tests.

Send out a notification to all stakeholders. If there are teams that you'll need to have on call, either to monitor the test or to quickly resolve any issues that might come up, make sure to check in with them before running a test to confirm that they're willing to support you. Email is a great way to send pre-test communications out to people, and should include at least the following information:

- What you're testing: what components do you expect to hit? Where will the traffic likely come from?

- When you're testing: what date and time are you starting the test and what is its expected duration?

- Why you're testing: what is the purpose of this test?

- How much you're testing: how many users or requests per second is your test scenario expected to hit?

Set up monitoring. If you're monitoring components yourself, start the PerfMon counters, deploy monitoring agents, and bring up the dashboards you will need to monitor the test. If

you're relying on another team to monitor some components for you, reach out to them to confirm that everything is set up and ready to go.

Update the Execution Log. This is simply a record of all tests that have been carried out and should contain this information:

- Run ID: Give each test a unique ID to be able to refer to it later
- Date, time and duration of each test so that you can look back at metrics later
- Purpose of the test
- Load profile
- Any other configuration tweaks from the default that were used for this particular test (such as running with a different set of users, or running against a specific database
- A quick summary of results (after the test)

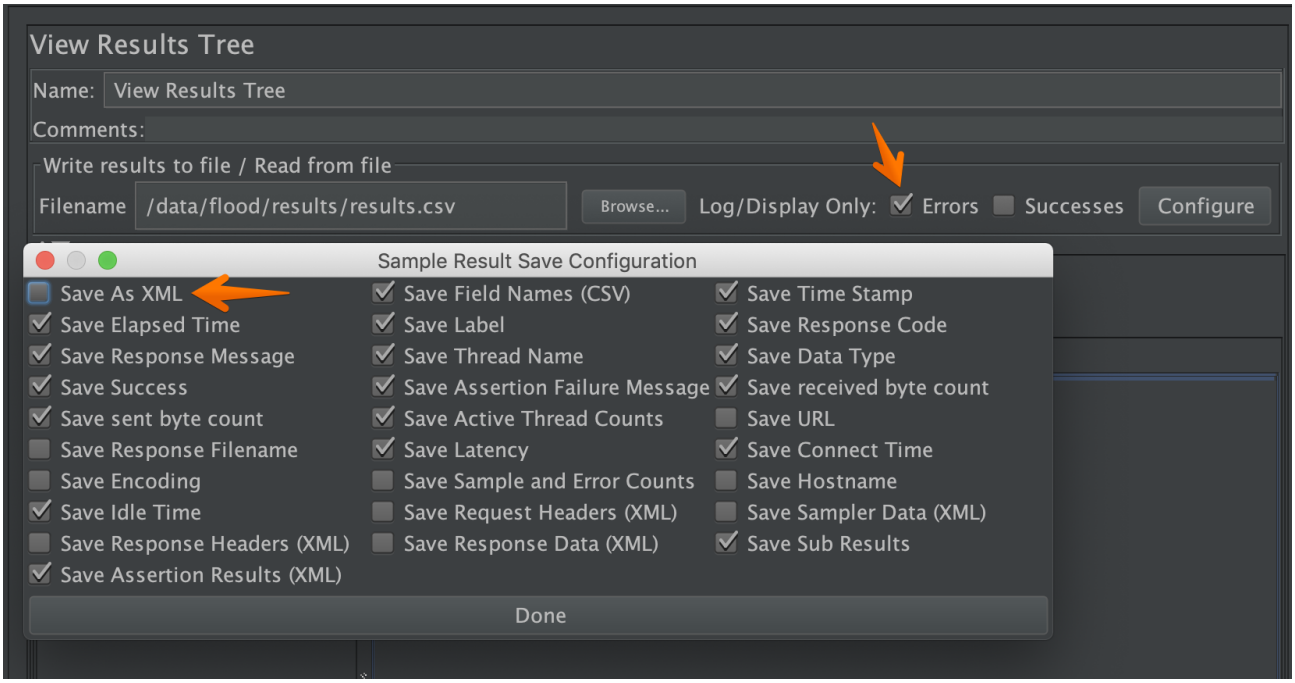
Unlike the communications sent out to stakeholders, the Execution Log is for the testers; it will help them determine what has already been done and provide them an easy way to reference tests and remember which one is which.

Shake out the environment by running a load test with fewer users to make sure everything is working as expected.

Shake out the data. You may have run your script a few times with some of the data, but if you have a large data set, you may want to double-check that the data is in the state you expect. For instance, if your test scenario includes users with certain permissions doing a privileged action, ramping up without shaking out your test data may lead to a lot of data-related errors if a lot of the users turn out not to have the right privileges. To test this without ramping up and incurring the associated costs, start a smaller-scale test (on a smaller sample of nodes) and leave the script running overnight, just to loop through all of the data (at a very low load, or even with just one thread) and determine whether the data is in the expected state.

When you're ready to begin execution, **check log settings**. Debug mode is for shakeouts, not for execution. Logging takes up a significant amount of memory and space, which can severely impact the consistency and accuracy of your results. Having excessive logs while ramping up to full load will also mean that there are a lot of logs generated, leading to large results files that may be unwieldy to transfer after the test.

Watch out for listeners in JMeter that are configured to save more results than are necessary. For instance, when using the View Results Tree listener, make sure that "Log/Display Only: Errors" is checked; otherwise you will also log successes. Another common mistake with this listener is leaving the "Save as XML" option checked in configuration settings, which will save all responses as XML if left checked. Here's a good example of what the configuration for this listener should look like:



This ensures a minimal footprint for the logs while still logging a sufficient amount of errors.

Custom logging can also be dangerous. Instead of writing code (for Gatling or via JMeter's Beanshell or Groovy samplers) to capture responses, consider anticipating errors and checking whether they exist. For example, when testing a login scenario, consider adding assertions that the response doesn't contain "Username or password invalid", "Account blocked after 5 unsuccessful attempts", "Account suspended", "Account already logged in", and other errors that may come up. You'll be dealing with a larger data set than normal, so some data-related errors are bound to creep in. Adding a way for your tool to identify these errors will save you time in determining the causes of errors.

Avoid high transaction cardinality. Recall that transactions are an action or group of actions that you want to measure in a load test. The names of these transactions are set in your script. Every user or thread that runs that script will output the response to the requests within your transaction and save it under the name of the transaction. Every user and

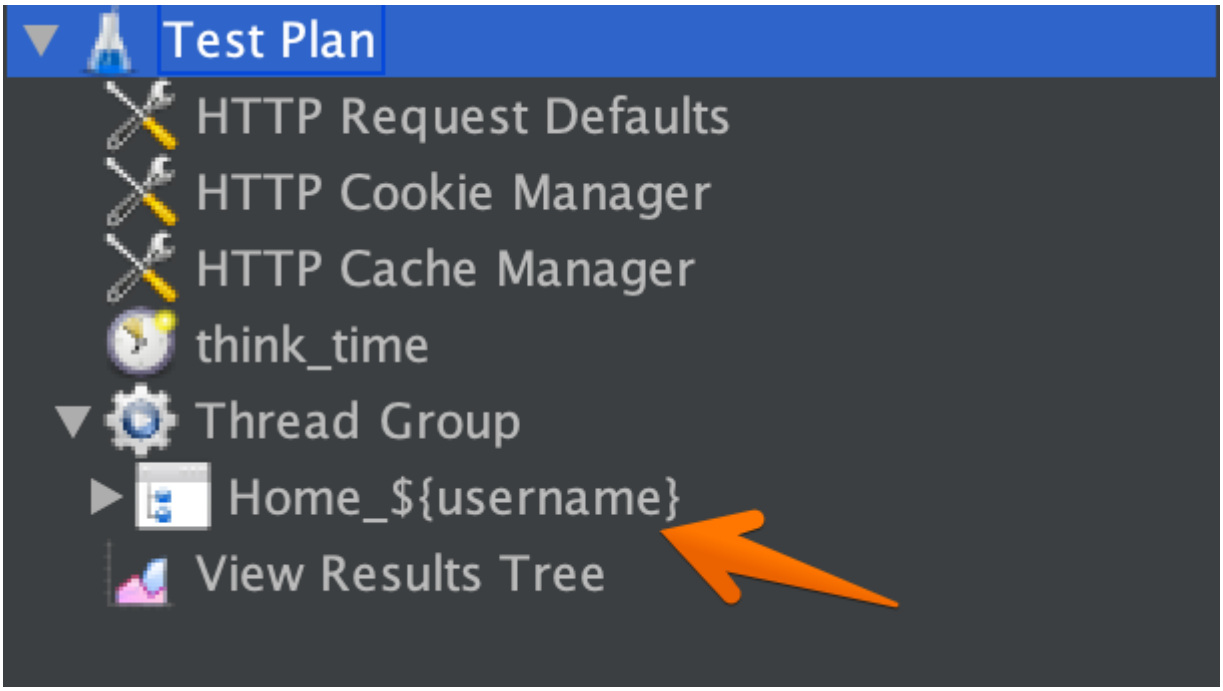
iteration will have the same transactions, and in a load test, all of the results are collated and tallied. When users are iterating over the same script in a load test, it doesn't add much information to know User 1's response time for the Purchase transaction compared to User 2's (unless the accounts used differ in a way that would change the load profile on the server, such as having different roles). What matters is the response time of the Purchase transaction across all users.

High transaction cardinality occurs when transactions have names that are too unique. Normally this is an effect of using dynamic variables in transaction names.

Here's what it looks like in Gatling:

```
exec(
    http("Home_" + username)
      .get("/")
      .check(status.is(200))
  )
```

and in JMeter:



In both cases, the variable username is often the account username that the script signs up with or a counter, which means that instead of having a single transaction called “Home” that you can compare against multiple users, you’ll end up with “Home_user1”, “Home_user2”, etc. This will make result compilation unnecessarily tedious. Don’t do this.

Run your test in CLI mode. JMeter’s GUI mode is great for writing and debugging test scripts, and potentially even for shakeouts, but all load tests should be executed in CLI or non-GUI mode. With JMeter, you can do this by executing this command in your terminal:

```
jmeter -n -t /filepath/test.jmx -l /filepath/log.jtl
```

If you get the error jmeter: command not found , this means that you’ll need to add JMeter to your PATH environment variable. To do this in macOS, execute this command:

```
export PATH=$PATH:/Users/nvanderhoeven/jmeter/apache-jmeter-5.0/bin
```

This will allow you to execute the previous command successfully and run your JMeter script.

Baselining

Finding a baseline is one of the primary goals you’ll have as you start executing load tests. In order to be able to assess how changes in the environment or code affect application performance, you’ll need a stable point of comparison. In order to compare two states of the application that hinge on a variable (say, before a code change and after), it’s best to keep as much of the other circumstances surrounding the test as fixed as possible. The baseline is your **unchanging** test scenario that you can use as a point of comparison to assess future performance.

For example, consider two tests A and B. After the results of A were shown to the team, the developers made a change in how the server caches requests.

A: 1000 users, 1 hour, with an average response time of 3 seconds.

B: 100 users, 10 minutes, with an average response time of 1 second.

Can you make any conclusion about whether or not the change affected performance? The answer, of course, is no— there are too many variables in the two tests to be able to say for sure. The change could have decreased the response time, but that response time decrease could have also been caused by the smaller number of users or the shorter duration.

Instead, what you want to do is run the same test:

A: 1000 users, 1 hour, with an average response time of 3 seconds.

B: 1000 users, 1 hour, with an average response time of 2 seconds.

C: 1000 users, 1 hour, with an average response time of 1 second.

In this second set of tests, it's a lot easier to see the effect of changes made on the performance. Clearly, the changes applied before executing Test C were more effective at lowering response time than those for Test A.

For this reason, here are a number of things you'll want to settle on for your baseline and then fix:

- number of users

- duration of test

- think time, pacing, and all other waits and delays

- the script (including how requests are broken up into transactions)

User Density

Another thing you'll want to baseline is how many users you can run per node, or user density. As much as we may want to look for industry standards on this, prepare to spend a significant amount of time figuring out this number for yourself up front.

Figuring out user density is essential because all load generators, even virtual ones, have finite resources. This means that each load generator will also have a finite amount of load that it can generate, based on its CPU and memory utilisation, among other things. Trying

to generate too many users on a single node may result in the node itself being the bottleneck for your test.

If you're trying to test how much water a bucket can handle before it overflows, make sure your tap is fully open. In order to accurately assess your application's performance, make sure that the load generators display healthy resource utilisation.

It can be helpful to have a number to start with. At Flood, we've found that we can reliably run 1000 users using JMeter or Gatling with one of our AWS m5.xlarge machines. For reference, an m5.xlarge machine has 4 VCPUs and 16 GB RAM.

If your machine is similarly specced, run a test on a single node with 1000 users. While the test runs, watch the CPU and memory utilisation. If the test finishes without either of those consistently hitting over 80%, you'll know that the node can handle that number of users.

Let me reiterate, though, that you should use the 1000 users figure as a starting point only. You can then figure out your number through trial and error. Increase the number of users past 1000 and watch the resource utilisation again. If that still looks good, add some more users and rerun. When you get to a test where the resource utilisation hovers above 80%, stop and fall back to the previous number of users. You'll now have your number.

Another thing you might want to play with is the think time and pacing in your script. These waits tend to have a huge impact on resource utilisation, so you can expect to be able to run more users per node if you increase your delays.

If you're using JMeter, always run your tests in non-GUI mode. GUI mode is great for debugging, but is unnecessarily resource intensive for real load tests. While you're at it, disable any listeners you may have that you don't need to capture results.

Figuring out the appropriate user density now will save you from getting inaccurate test results. Running as many users as you can without overloading the load generators is also cost-effective, as you'll be making sure that you provision only as many nodes as you need.

Scaling your load test

It's relatively simple to run a load test on one machine, but if you want to run on two or more machines, things quickly get unwieldy. The specifics for how to do this are out of the

scope of this ebook, but I do want to run through your options for achieving this.

1. Upload your tool of choice and script to every machine and kick off each test separately.

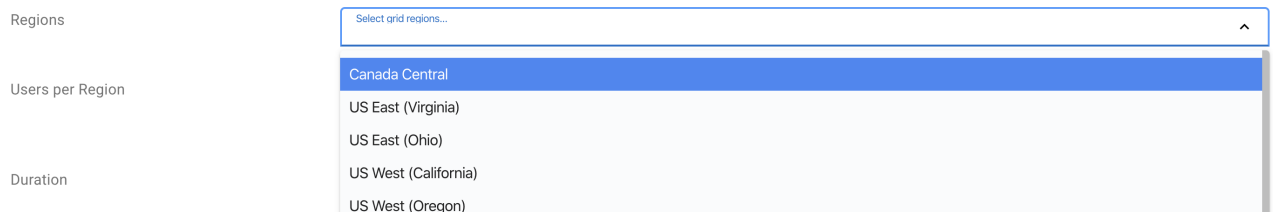
Instead of running one large load test, think of this method as running several smaller ones at (about) the same time. This is relatively easy to set up, but the main disadvantage is that it's tedious. You'll want to ensure that each load generator is as similar as possible to the others in terms of operating system, tool version, and script version. Since each node will kick of a separate test, you're not going to be able to see a real-time combined view of all the load tests, and after execution you'll need to retrieve results files individually and combine them.

2. Use your tool's distributed testing mode. Both JMeter and Gatling have a feature that will allow you to scale. Essentially this will involve setting up agents on each load generator and using scripts to coordinate execution and results collection. This requires a little bit more know-how and time to set up, but it's a little more cohesive than the first method. Here are links on how to set this up for each tool:

Remote Testing with JMeter

Scaling Out with Gatling

3. Use a load testing platform. Distributed load testing platforms like Flood are popular for this because they take away all the setup considerations. This is the easiest option, especially for teams that are new to load testing or are perhaps less technical, because all the setup is done through a UI. Scaling out in this case just means uploading your script, choosing the number of nodes you want to run and in which region, and all the work is done for you in the background.



Metrics from your test are shown in real time and the results from each node are collated in one place for easy download if necessary. A disadvantage of this method is that you'll need to pay for the service.

Monitoring

While you're running the test, monitoring is probably going to be your biggest concern. Here are a few things you need to watch out for.

Load Test Results

Ideally, you'll have real-time stats of how your load test is going. It's possible to run your load test and analyse the results later, but it saves a lot of time if you monitor your test in real time. Here's what you need to be watching during the test:

- Error rate: This is probably the biggest one. If 100% of your transactions fail, there's really no point continuing the test. Stop the test and debug the error before continuing.

- Response times of key transactions: Seeing higher than expected response times from transactions is not necessarily a reason to stop the test, but alerting the relevant team to the issue can save some time figuring out the issue later. Sometimes keeping a test is running is necessary while logs are checked.

- Throughput: Check that the script is hitting the throughput that you expected, taking response times into account. This is an opportunity to see whether your script is falling short of the targeted load and fix it later. Sharp peaks or drops in throughput should be investigated (if they're not part of your test scenario).

Server health

You'll want to be monitoring the health of all servers, and that includes your load generators. In fact, load generator health is critical as over-utilising the load generator itself is a sure-fire way to render your results inaccurate or inconsistent and therefore unusable. The load generator is just like any other machine: attempt to do too much on it and it will become sluggish. Response times are measured from the time that the request is sent by the tool to the time that the tool gets a response back, so they are influenced by sluggishness in the load generator. You don't want your load generator to be the cause of the performance issue, so make sure you monitor every test while you're running it.

You'd also ideally do this for important components in the architecture of your application, if at all possible.

Test duration and quantity

You're ready to kick off the test! But how long should you let each test run? And how many tests should you run for each test scenario identified in the planning phase?

It's all about **sample size**.

Sample size is the number of observations recorded before making a hypothesis. Having a sample size that is too small can drastically affect the results of that hypothesis, because you don't have as much to go on when drawing that conclusion. For example, if you want to know how many people know what load testing is, you may want to ask more than two people. Similarly, when running a load test, we want to make sure that our sample size is large enough that we can draw conclusions from the results of the load test and extrapolate the performance of our application from them.

In load testing, we can look at sample size from two aspects.

The duration of a test affects the sample size because the longer a test runs, the higher the number of transactions that can be executed. The type of test scenario will also affect your test's duration. For instance, a soak test, which aims to measure how your application responds to a sustained load, will be insufficient if you run it for five minutes. The same five minutes, however, might be perfect for a spike test, where you want to test a high amount of transactions within a very short timeframe. In general, though, I would be wary about drawing conclusions from a peak load test that lasts less than thirty minutes.

The number of tests that you run also affects sample size in that each test gives you new observations that you can use to form an educated hypothesis. It's a good idea to run more than one test for each test scenario. Relying too heavily on the results of a single test, especially if it's a short one, may be dangerous because something as simple as a scheduled batch job executing a certain time every day might skew your results.

Analyzing results and reporting

What are test results?

We tend to place a lot of importance on load test scripting or execution, and those things are important, but it's all for nothing if you don't know what the results mean. In order to run good load tests, you'll need to get comfortable dealing with data. The quantitative data from a load test may include:

- Data generated by the load testing tool itself while executing the script, including response times and error rates but potentially also debugging information. You'll get this from your tool's results file as you specified in the Saving Results section or from Flood's dashboard.

- The resource utilization metrics of every load generator, to rule out any execution issues.

- The resource utilisation on your application servers to determine how your application actually performed under load. You'll get the resource utilization metrics from each server.

We'll refer to all of these as "test results". The first step will be to collate all of these results in one place. Then we can delve into actually making sense of those results.

Let's run through the different terms you're likely to come across in your testing. For each one, we'll go over what they mean and what they say about your application performance.

Load testing metrics

Below are some common metrics that are generated by your load testing tool. They pertain to the load test scenario you executed and will give you insight into the user experience (in terms of response time) of your application under load.

Concurrent users are the number of users running at the same time. In JMeter or Gatling speak, this is also referred to as the number of threads. Given the same script, a larger number of users will increase load on your application. Note that user concurrency doesn't say anything about throughput. That is, having 1000 users doesn't necessarily convert to 1000 requests per second. Nor does it mean that those 1000 users are all actively using the application. It only means that there are 1000 instances of your script that are currently

ongoing, and some of those could be executing think time or other wait times that you've scripted.

Response time is the time between when the transaction was sent by the load testing tool on the load generator and the time that a response was received. This is the time to last byte. Since it is measured by the load testing tool, it does include things like latency and is affected by bottlenecks on the generator (such as high resource utilization). Both JMeter and Gatling ignore think time when calculating this. Response time is a useful metric to look at when trying to get an idea of how long your application took to process requests. A high response time means a longer processing time.

The **transaction rate** measures the throughput of your load test. JMeter sometimes reports throughput in terms of samples per second, which is a similar but concept but not the same. Generally speaking, a JMeter sample is a single request, and multiple requests can be grouped into transaction controllers. So samples and transactions are not interchangeable, but they do both describe how quickly your test is sending requests to your application. The transaction rate, more than the number of concurrent users, better describes the load your application is handling. You can expect a higher transaction rate to correspond to higher load.

The **failed rate/error rate** is normally expressed as the number of failed transactions divided by the total number of transactions that were executed. It is often represented as a percentage: an error rate of 40% means that 40% of all transactions failed. Whether a transaction failed or not is determined by the script and can be caused by many issues. A transaction could fail due to a verification on the page not being found due to an unexpected response (such as an error page being returned) or it could also be due to a connection timeout as a result of the load testing tool waiting too long for a response. High error rates are an indication of either script errors or application errors and should never be ignored.

The **passed rate** is similar to the error rate but measures the other side of the coin: it expresses how many of the transactions during the test passed.

Resource utilization metrics

Resource utilization metrics, unlike the load testing metrics in the previous section, are not generated by the load testing tool, although some tools do also report this. They are

measured by the operating system of the machines involved, including both the load generators from which load is generated as well as your application servers, which receive the load.

What you're looking for in resource utilization metrics from the load generators is a way to determine whether or not the load generators were a performance bottleneck. Ideally, you will want all the metrics to show that the load generators were not overutilized, which means the results of your test will be valid. If they were overutilized, you will want to fix the issue and re-run the test.

The resource utilization metrics of your application servers, however, will show you how easily your application handled the load and will give you an idea of how much more it can handle. If you've identified a bottleneck, these metrics will also give you a clue as to where to start your investigation.

CPU utilization is how much of the machine's processing power was being used during the test. This indicates whether a server was struggling with the tasks it was carrying out at the time. In a load generator, you'll want to make sure you stay below 80% utilization for most of the test. In an application server, consistently high utilization may suggest that you need to allocate more CPU towards the server.

Memory utilization is how much of the machine's memory (RAM) is being used up. Sometimes this is measured in percentages (80% memory utilization means 20% of the memory was not used) or this can be expressed in terms of available bytes (the amount of memory that was not used). Consistently high utilization (again, >80%) could point to a memory leak within the server. Memory leaks are often only spotted in longer tests, which is why it may be worthwhile to extend the duration of your tests. High memory utilization could also be a good reason to consider allocating more memory to the server.

Network throughput is similar to the transaction rate in the sense that it tries to measure how much load is being put through the system; however, it does this by measuring the amount of data in bytes that is being delivered by your application servers to the load generator. High network throughput is only a concern if it is equal to or hitting up against the maximum bandwidth of the connection.

For instance, Flood uses AWS nodes of type m5.xlarge by default, which have an advertised bandwidth of up to 10 Gbps. A test with a network throughput of 10 Gbps is a concern

because it means that the bandwidth of the load generator itself is starting to be a bottleneck. In this case, you should attempt to decrease network throughput by incorporating waits into the script so as not to hit this limit.

Latency is the portion of the response time that accounts for the “travel time” of information between the load generator and the application server. It can be influenced by factors such as network congestion and geographical location and is notoriously difficult to measure. While it’s ideal to have low latency, having high latency does not necessarily render a test void; it’s still possible to determine the actual server processing time by subtracting latency from the overall response time.

Disk I/O Utilisation metrics are useful because they measure how quickly data is transferred from memory (RAM) to the actual hard disk drive and back. This can be expressed as a rate (reads/sec or writes/sec), a percentage (busy time is the percentage of time that the disk was actively being used), or even a number of requests (queue length). Requests that cannot immediately be processed are assigned to a queue to be processed later, and a high amount of requests in this queue can be taken as a disk utilization issue on the application server because that means it can’t keep up with the number of read/ write requests. These metrics are more useful for the application server rather than the load generators.

Have a look at the Server Monitoring section to revise your knowledge of these metrics.

Analyzing results

After collating the relevant metrics, you’ll want to start making sense of them. Metrics are useless if their context is not taken into account. Your job is to use those numbers to tell the story of what happened.

It’s impossible to thoroughly explain how to analyze results in this book, but here are some considerations to keep in mind.

First: was the load test valid?

Like any good scientist, your first duty after carrying out an experiment is to determine whether or not the conditions of the experiment accurately recreated the scenario you want to test. Here are some questions to ask yourself:

- Was the load test executed for the expected duration?
- Did the load generators display healthy resource utilization for the duration of the test? Were CPU, memory, and network metrics within tolerance?
- Did your load test hit the throughput (requests per second) that you were aiming for? Is this similar to what you would expect in production? Consider drilling down further into separate transactions: are there business processes that are more common in production than in your load test?
- Was the transaction error rate acceptable? How many of those errors were due to script errors and data in the wrong state?

Next: How did the application handle the load?

Now that you've determined that your load test was a good replication of production load, it's time to figure out how your application fared under that load. Your goal here is to identify whether any performance bottlenecks exist.

- What was the average transaction response time? It's important to drill down into separate transactions here because not all transactions are alike. Which transactions performed worst? Are all the transaction response times pretty close to each other, or are there some transactions that are far and away slower than the others? Also, look at more than just the average: what are minimum and maximum response times, and is there a large gap between those two? What is the 90th percentile response time?
- How much of the transaction error rate was caused by legitimate application errors? Were there any HTTP 5xx responses that were returned by your server? Did these errors occur at the start of the test when the users were still ramping up, right after the full number of users was reached, or at the end of the test? Are there clumps of errors during the test, or were they spread out across the entire test? Do errors occur at regular intervals, and if so, were there any scheduled jobs going on at the same time on the application server?
- When the application failed, did it fail gracefully? Did it display a nice error page, or did it simply offer up an unfriendly error? If your application has load balancing, did the load balancer correctly redirect traffic to less utilized nodes?
- Was the resource utilization healthy on all your application servers? Were your nodes similarly utilized? Were there certain points in the test that display higher utilization than others, and what was going on at that time? Did memory utilization increase as the test went on? Did garbage collection occur?

- Do the server logs display any unusual errors? What was the disk queue depth during the test? Did the server run out of hard disk space during the test, and do you have policies in place for backing up and deleting unnecessary data in production?
- What to watch out for: examples of bottlenecks

Finally: If there were bottlenecks, why did they occur?

This is by far the most difficult part of results analysis, and you may have to liaise between different teams in order to determine why your application didn't behave as expected.

The key here is to go beyond symptoms and actually try to get down to the root cause. If you find yourself saying the following things, it's a good sign that you haven't investigated the issue enough:

"The response time is high because CPU utilization on the server was high."

"The error rate was high because the server returned HTTP 500s."

"The application was slow because all 1000 users had ramped up."

"The login server restarted unexpectedly."

"Load was not even across all nodes, so response times on one server were higher than on the others."

"Memory utilization was high when the identity verification process was triggered."

These statements, as true as they may be, don't really leave you with actionable insights. Instead, ask yourself why several times until you get to the root of the issue.

For instance, take the first statement: "The response time is high because the CPU utilization was high." Why was the CPU utilization high? Well, because the server was busy processing a lot of information at the time. Why was the server processing a lot of information? Because the requests to go to the home page retrieve information from many application components before being returned to the main server. Why does a user browsing to the home page send so many requests? Maybe it shouldn't.

In that case, asking why several times got to the root of the issue: a simple GET of the home page was requesting far more resources and potentially causing higher CPU utilization on the server than was actually necessary.

In figuring out the real reason behind the symptoms you're seeing, you'll come up with tangible steps towards addressing it or also inform management's decision as to whether or not to proceed with a release.

Putting together a report

Communicating the load testing results is almost as important as running the test itself because a good report can sometimes determine whether your findings are actually addressed or whether your testing becomes just a "check-the-box" activity that gets forgotten.

You'll likely need to come up with at least two reports based on the expertise of the stakeholders you're giving reports to.

The Management Report

The management report is one that you intend to give to an audience that is not necessarily technical, so you'll need to tailor the results you show accordingly.

The Executive Summary

This summary should explain in simple terms what you did, why you did it (your goal), how the application behaved, and what you can recommend that would improve performance. This is the highest level of report and, depending on the experience of the manager you're giving this to, may be the only thing that some people read. So make it easy to read, as free of technical jargon as possible, and with very simple tables. If possible, put findings in bullet points.

If your report is a book, the executive summary should be the CliffsNotes version: enough to understand the gist of testing even if you don't get into any of the specifics.

For the rest of the management report, you can then back up a bit and then go into slightly more detail about each point:

- Nonfunctional requirements: what were they?

- Test scenario: what tool did you use for the test? What were the key transactions that you identified? How many times did you run the test, and for how long? Did you run different kinds of tests (load, peak, stress, etc)? How many users did you run?

- Results: what were the response times of the key transactions? Include the top five or ten transactions with the highest response times.

- Fixes: What issues did you find on the application servers? Were there any things that were addressed as part of testing in order to improve testing? Show a before-and-after graph of chart of response times with the differences highlighted.

- Recommendations: What server configuration does your testing suggest is optimal? How did this release compare to other releases in terms of performance (if applicable)? If you had more time to test, what kind of tests would you run, and are there any tests that you would recommend adding to the standard suite of tests in the future?

If appropriate for your role, this is also where you make a GO/NO GO recommendation, along with reasons for that decision.

The Technical Report

While the management report prioritises readability and comprehensibility above precision, the technical report will be geared towards the developers or other technical testers on your team, so you can go ahead and add more detail. This is where you'll go through ALL of the results, but don't forget that you're still telling a story— don't just copy and paste graphs; explain what they mean.

While in the management report you should tend to go for average response times, you can expand on this a bit in the technical report and include some more statistics, such as percentiles, standard deviation, and other ways to display the same data.

You can also include data here that is not necessarily significant — for instance, if the resource utilisation on the load generators was healthy, you will not want to include graphs in the management report, but you should include it in the technical report to anticipate those questions.

API load testing and Continuous Integration

What is Continuous Integration?

Continuous Integration is a software engineering practice in which automation is used to tie the main parts of the software development lifecycle together. This movement started in the development space and advocates improving the process that starts with developers committing code to a repository and ends with that code actually being deployed in production.

As software testing has matured, this definition has involved in order to include testing practices as well.

A modern CI framework still begins with a developer checking code into a repository, which would then automatically kick off functional tests of the application that determine whether or not that code will be allowed to proceed to the next step. Once functional tests are passed, the code could then be automatically deployed into production, with reports emailed out to relevant stakeholders.

A natural extension of this would be to add in a load testing framework. After functional tests pass, the code could potentially be deployed to a staging or similar environment, where we can run our API load testing scripts to see if there have been any performance regression bugs introduced by the code. In this way, load testing could be an extra check before sending out code to production.

This sort of CI framework is preferable to the traditional process, which involves many manual checks and potential bottlenecks as the code is passed among different teams (development to functional testers to nonfunctional testers, with DevOps engineers helping each team along the way). A good CI framework may take a little bit more time to set up in order to come up with the correct path, but it dramatically reduces the amount of time required and encourages a development practice that is iterative, quick, and truly agile.

In this chapter we'll discuss how to add load testing to your CI framework.

Jenkins

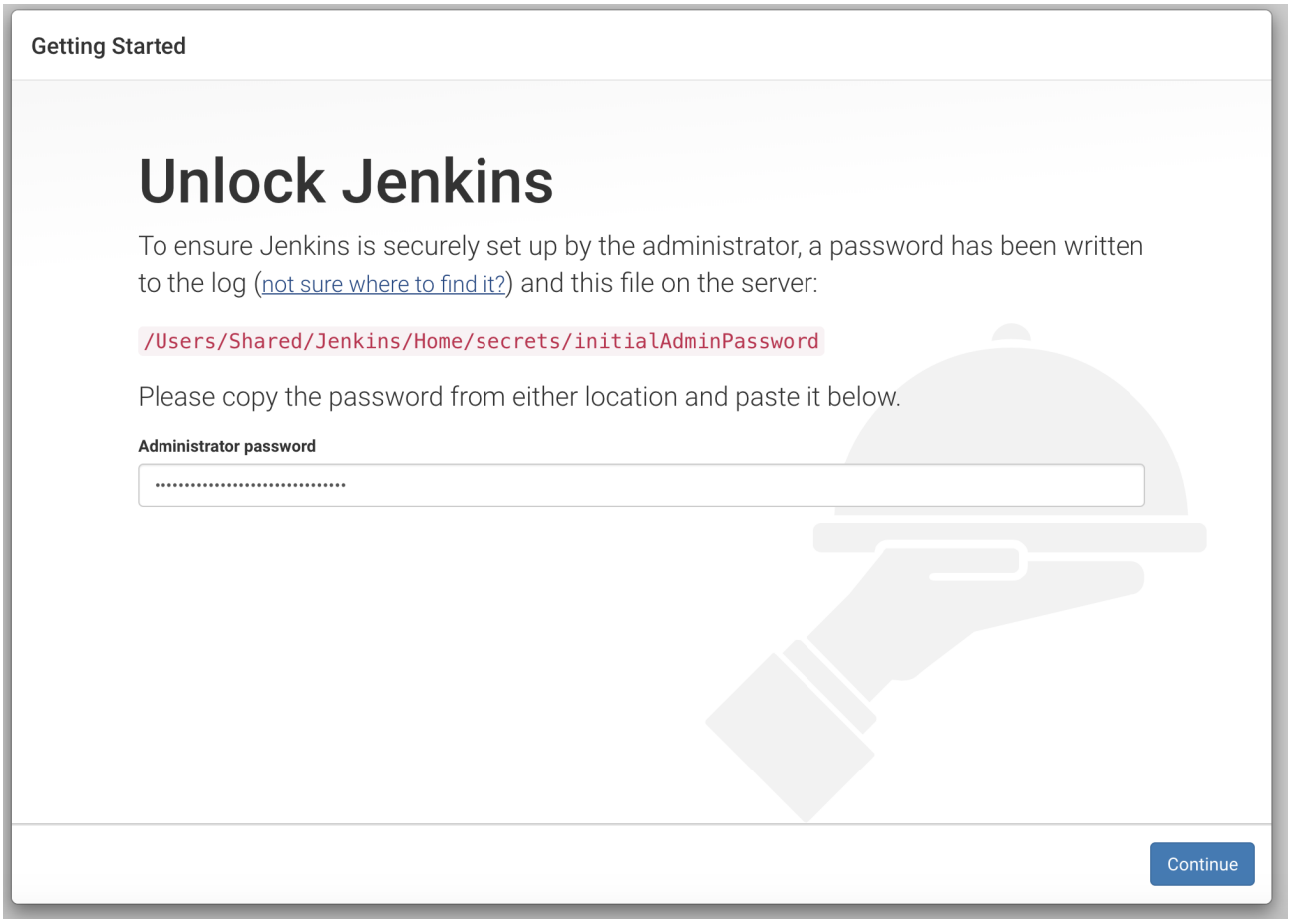
At the heart of your CI framework will be an automation server that is flexible enough to integrate with a variety of tools and applications that you use as part of your development process. There are a few options for this, but one of the most popular and user-friendly ones is Jenkins.

Jenkins is an open-source tool that will handle the interactions between disparate development and testing activities and serve as the traffic cop that will tie everything together. In the succeeding sections, we'll go over the intricacies of installing Jenkins, running your load testing script through Jenkins, and automating test execution through Jenkins.

Installing Jenkins

Download Jenkins here. Choose the appropriate version of Jenkins and download it. Then execute the downloaded file.

If a browser page does not automatically open, open your favourite browser and navigate to localhost:8080. You should see something like this:



You'll need to get an administrator password to unlock Jenkins. You can click on the link in the dialog box to get specific information for your operating system, but here are some instructions for macOS:

1. Navigate to /Users/Shared/Jenkins/Home

2. Right click on the folder secrets and click Get Info.

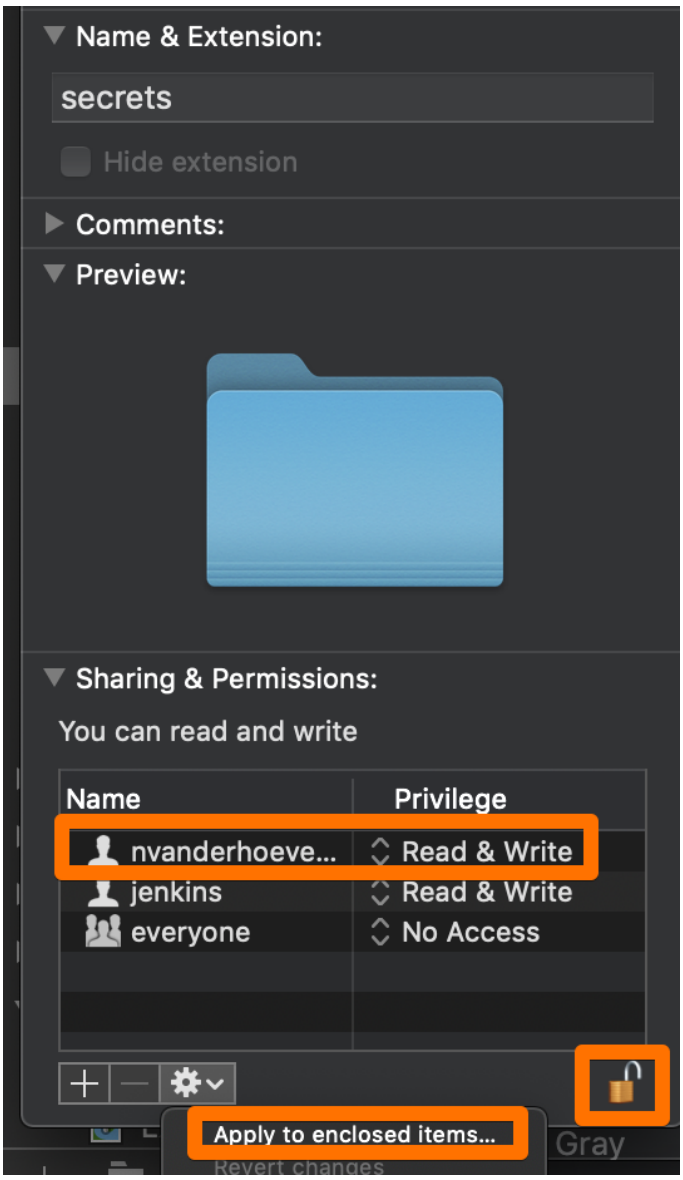
3. Scroll down to the bottom right corner of the dialog box. You may have to expand the last item in order to see it. Then click on the lock image.

4. Enter the password to your machine when prompted.

5. Click on the + and add your local user with the permissions "Read and write".

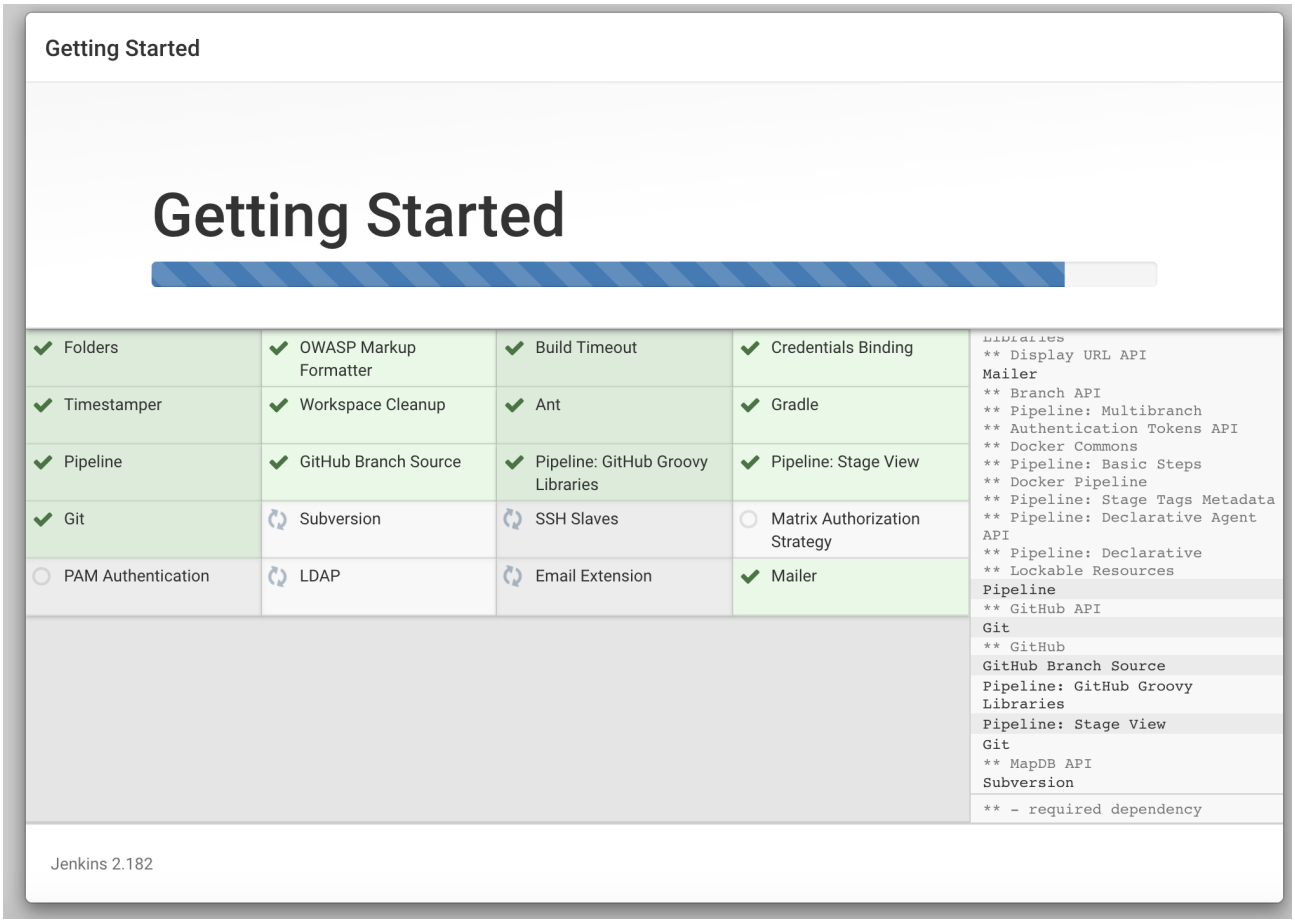
6. Click on gear icon and click "Apply to enclosed items".

7. Double click on the secrets folder and open the file named initialAdminPassword. It will contain your administrator password.

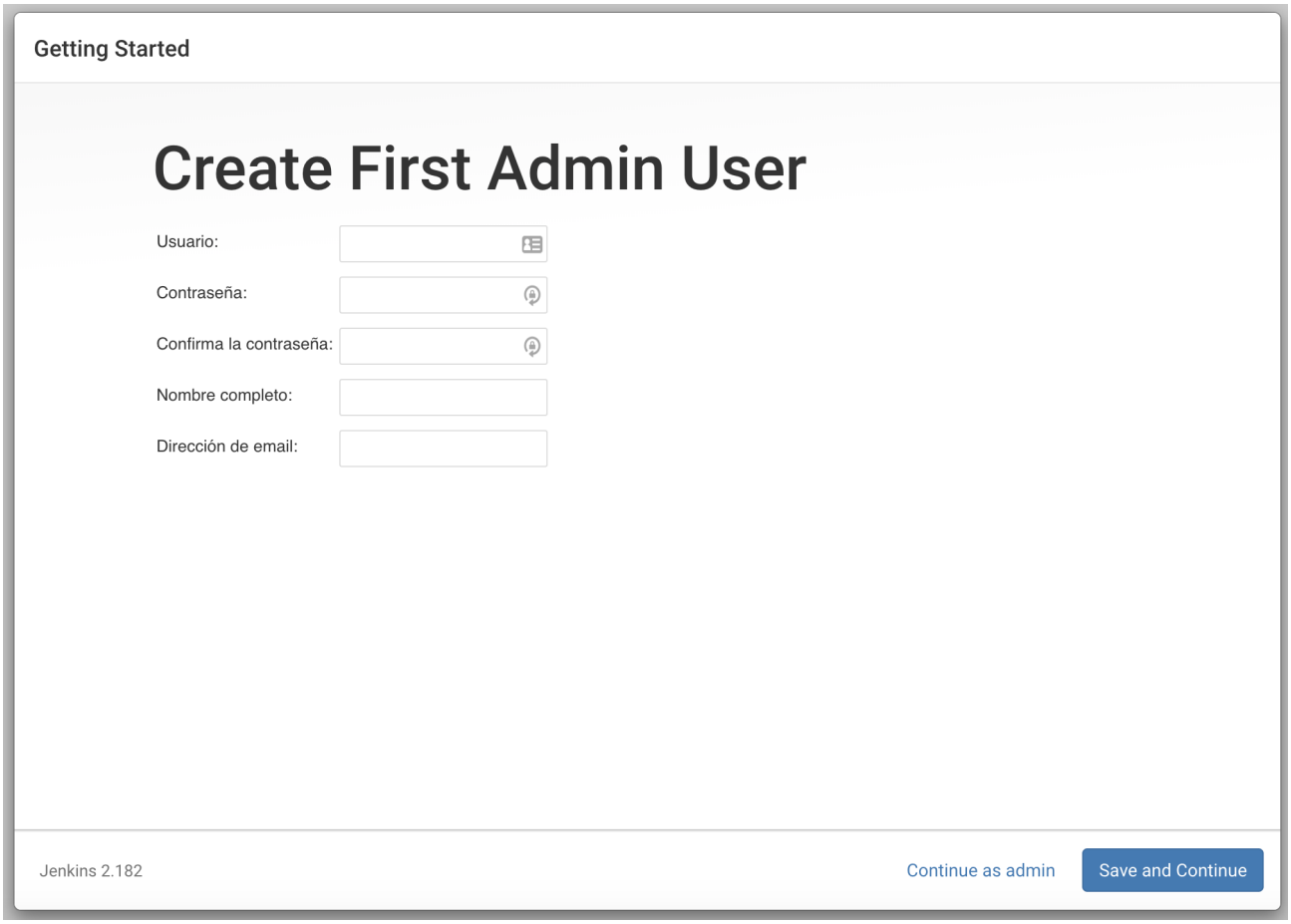


You should then be able to enter your administrator password into your browser window with Jenkins and click Continue.

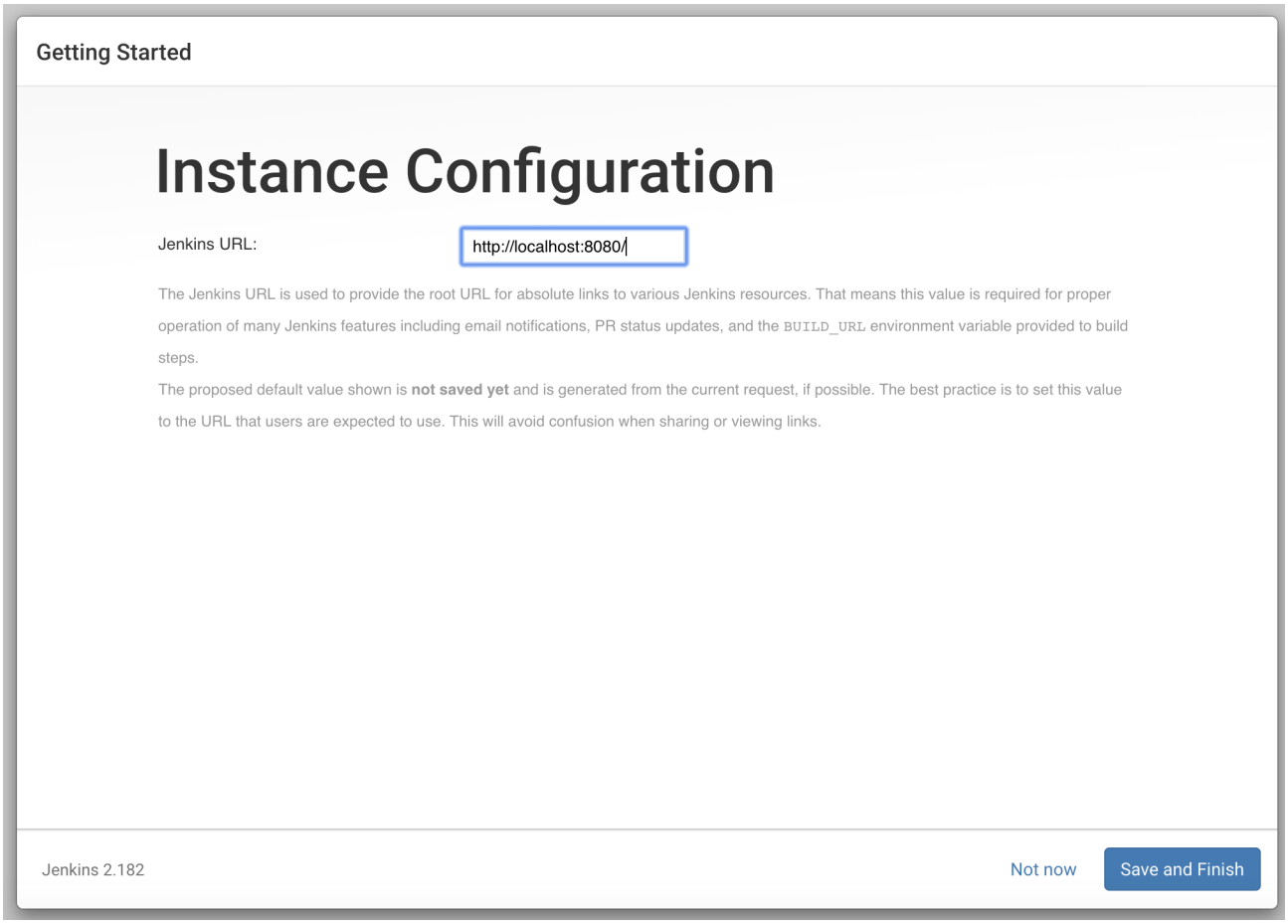
When asked to install plugins, opt to install the recommended plugins and you'll see a screen like this:



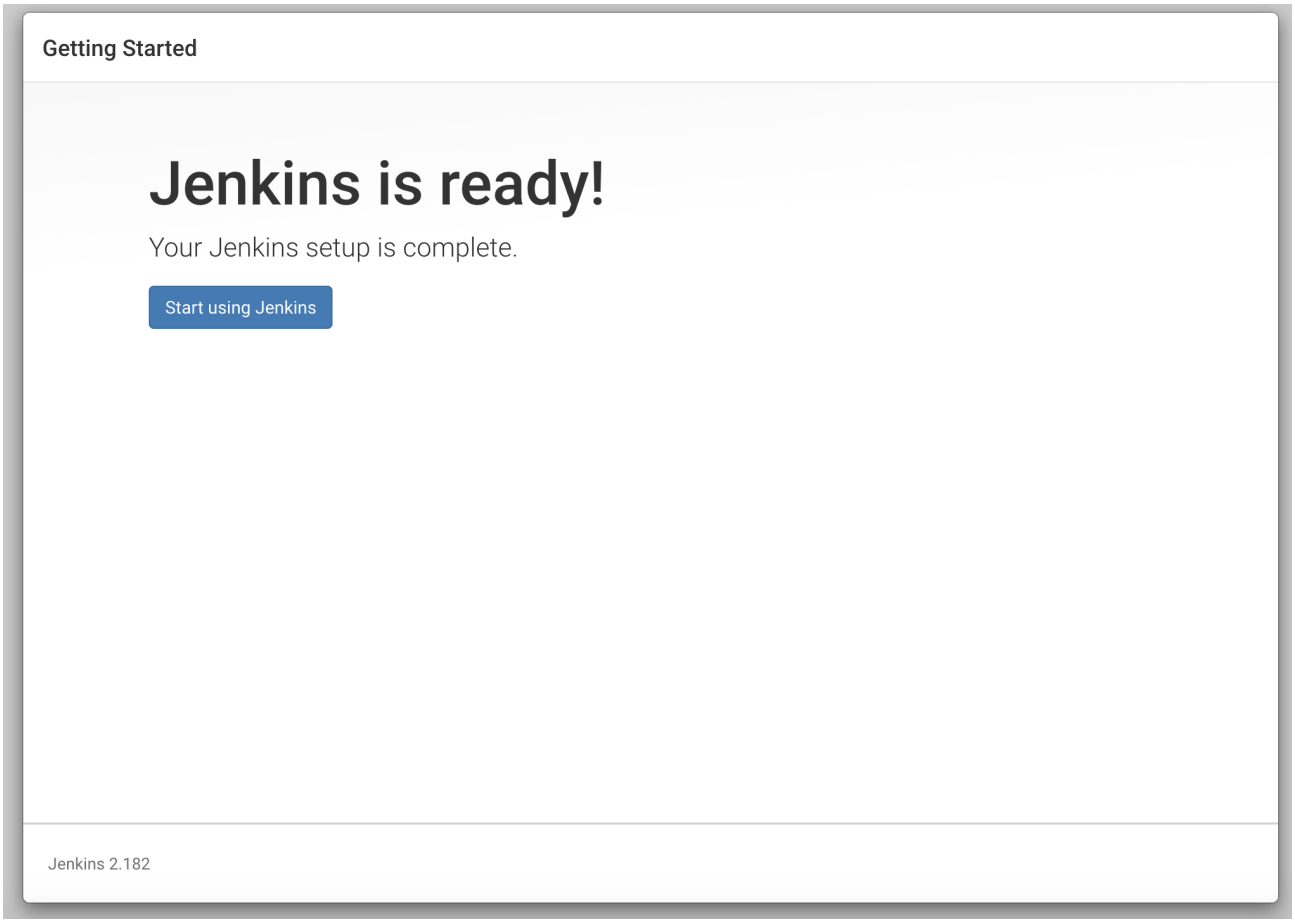
After those plugins are installed, you'll be prompted to create an admin user with which you'll use Jenkins.



Fill out the information and remember your admin credentials. On the next screen, leave the Jenkins URL as the default.

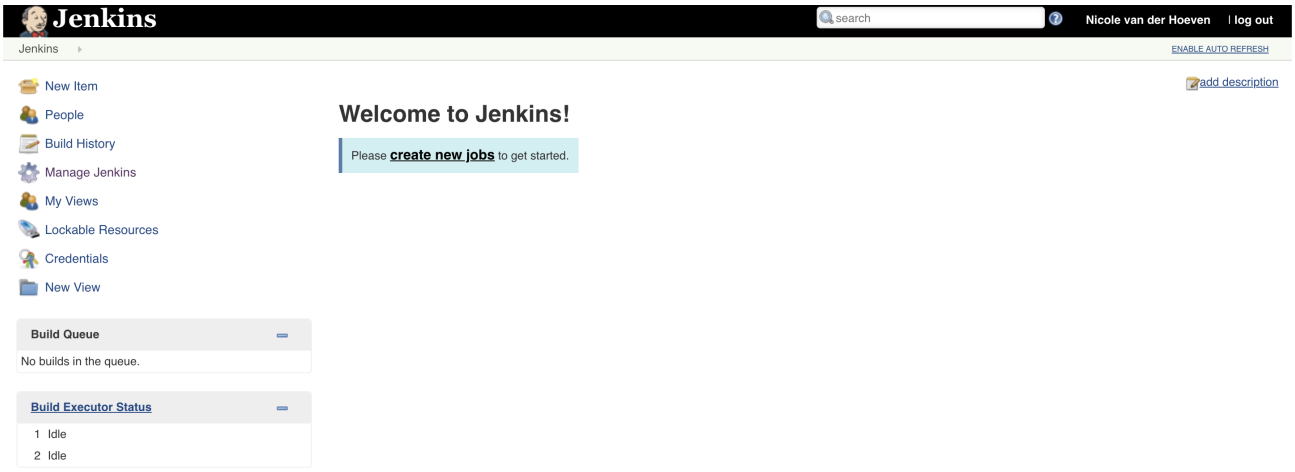


Click Save and Finish.



Click Start Using Jenkins.

When Jenkins starts, login with the username and password you set for your admin user. Click Back to the Dashboard. You should see something like this:



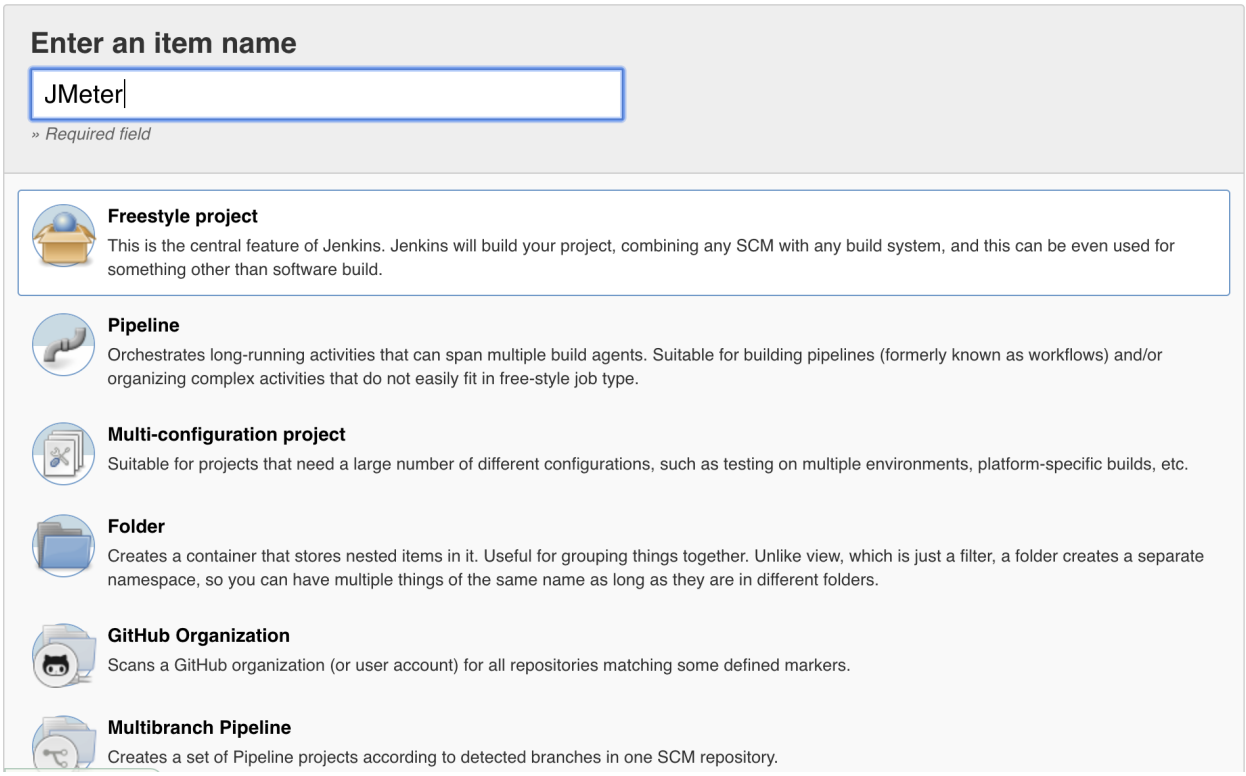
Congratulations! You've installed Jenkins along with the plugins you'll need to run your load testing scripts through it.

For more instructions on installing Jenkins for specific operating systems, check out the Jenkins installation documentation.

Running a JMeter script in Jenkins

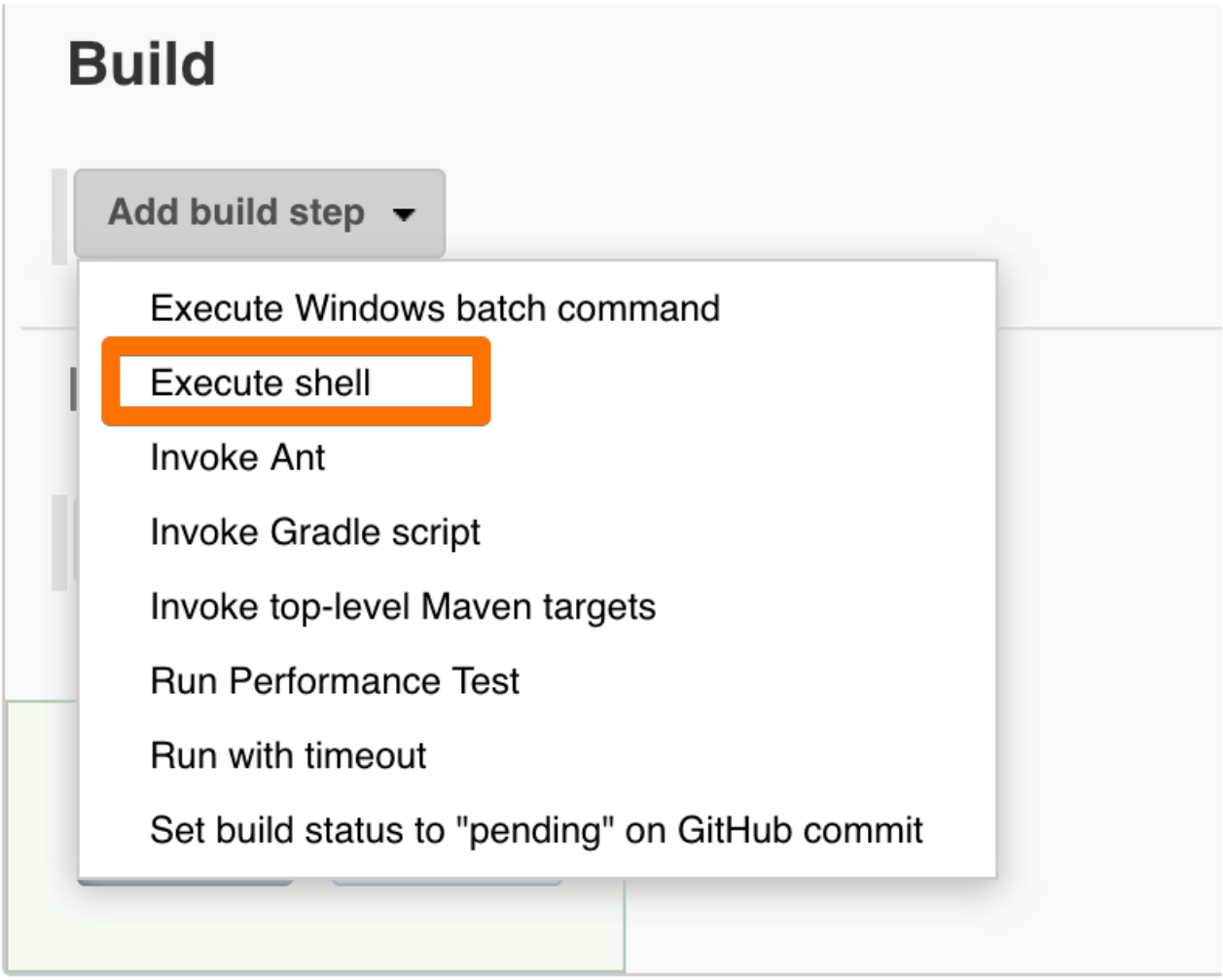
Jenkins has a good guide for running JMeter scripts, but we'll go through the basics here anyway.

From the Jenkins dashboard, click on Create new jobs.



Enter an item name ("JMeter"), click Freestyle project, and then click OK.

Add a description for your project and then head to the Build section. Click on the "Add build step" dropdown menu and select Execute shell for macOS.



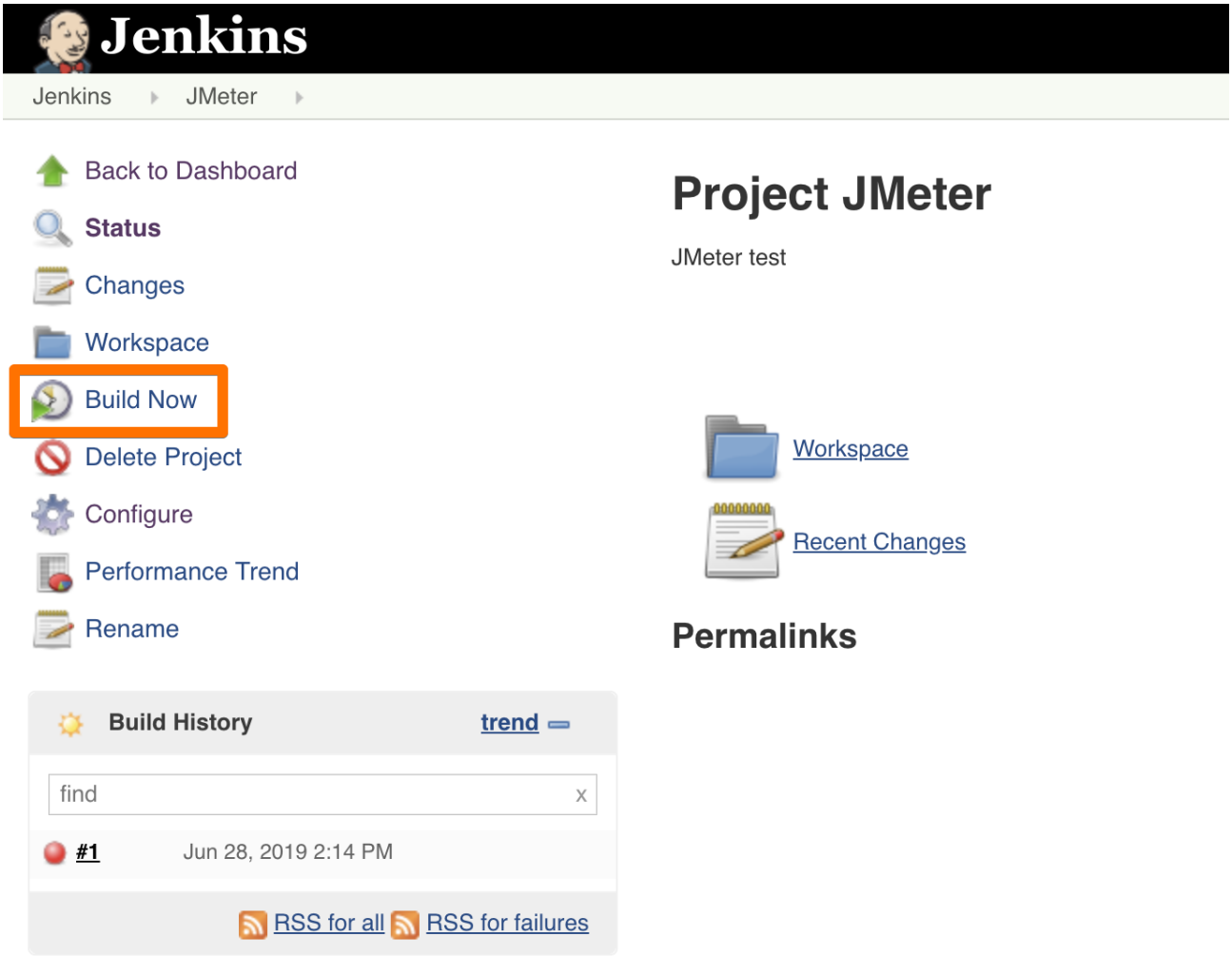
In the Command field, type in the command you would normally use to run JMeter in CLI mode.

Note that you'll need to include the absolute path of JMeter, your script, and your results file such as :

/Users/nvanderhoeven/jmeter/apache-jmeter-5.0/bin/jmeter -n -t /Users/nvanderhoeven/jmeter/jmeter-scripts/challenge_jmeter.jmx -l /Users/nvanderhoeven/Downloads/results.csv

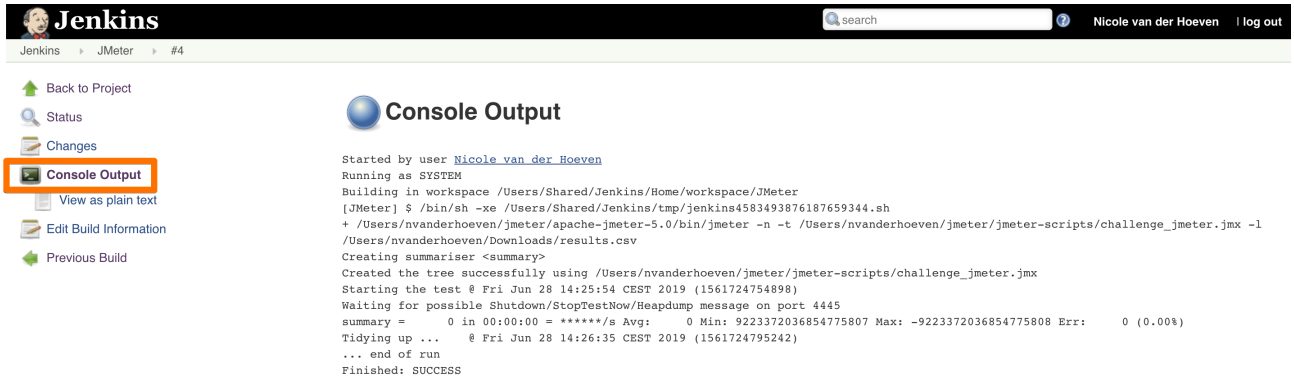
Click Save.

Click Build Now.



Click on the Build number in the “Build History” section that appears.

Then you can click on the Console Output link to see the real-time logs of the test:

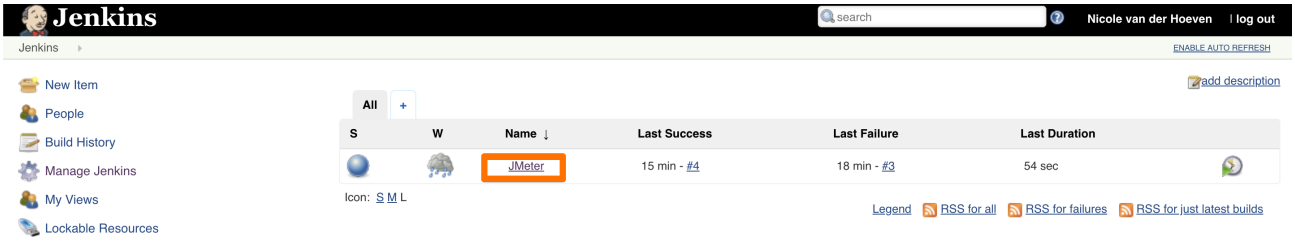


You've just run your very first JMeter script through Jenkins!

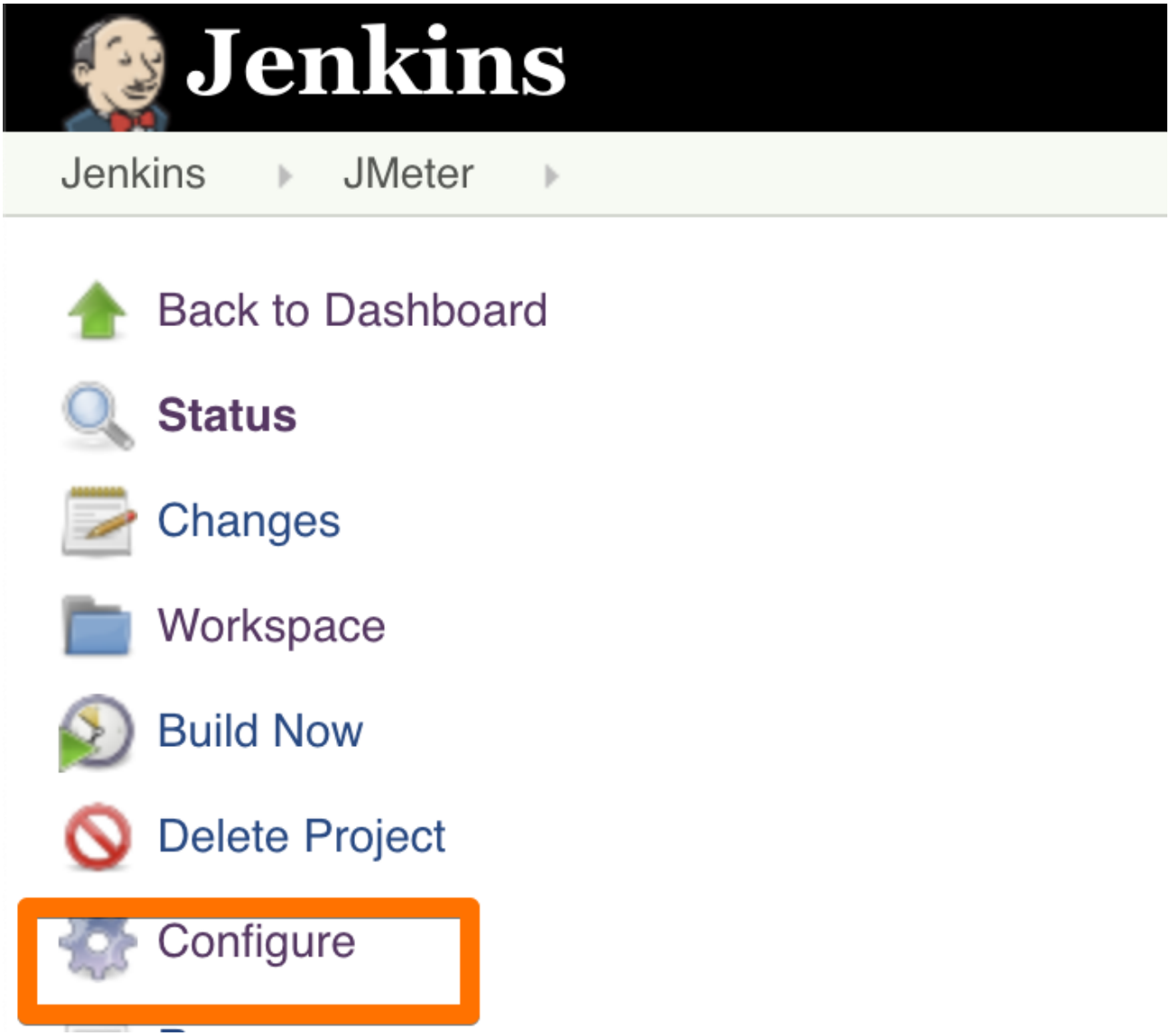
Build triggers

Build Triggers are really where the magic happens. You've managed to run your script manually through Jenkins, but that doesn't really add any more functionality than just running your test through your terminal. The beauty of Jenkins is in setting up build triggers that automatically execute your test once certain triggers are met.

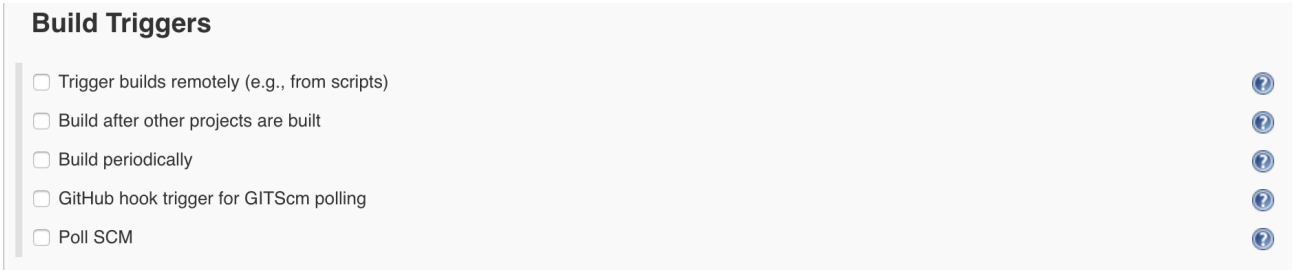
To set these build triggers, go into your project from the Jenkins dashboard.



Click Configure.



This will take you to the same screen where you first set up your project. This time, scroll down to the section labelled “Build Triggers”.



There are a few options here to use as a trigger for your project.

Trigger builds remotely will allow you to create a special URL that you can use to execute the script from another computer. This is especially useful if you want to build this into a

batch job on one of your servers. For example, you could add a command in your script to go to IPADDRESS:8080/job/JMeter/build?token=TOKEN_NAME or /buildWithParameters?token=TOKEN_NAME and your project will automatically be built.

Build after other projects are built will start this project after another Jenkins project has been built. This is handy if you need scripts to run in sequence.

Build periodically will run your test according to a schedule that you define.

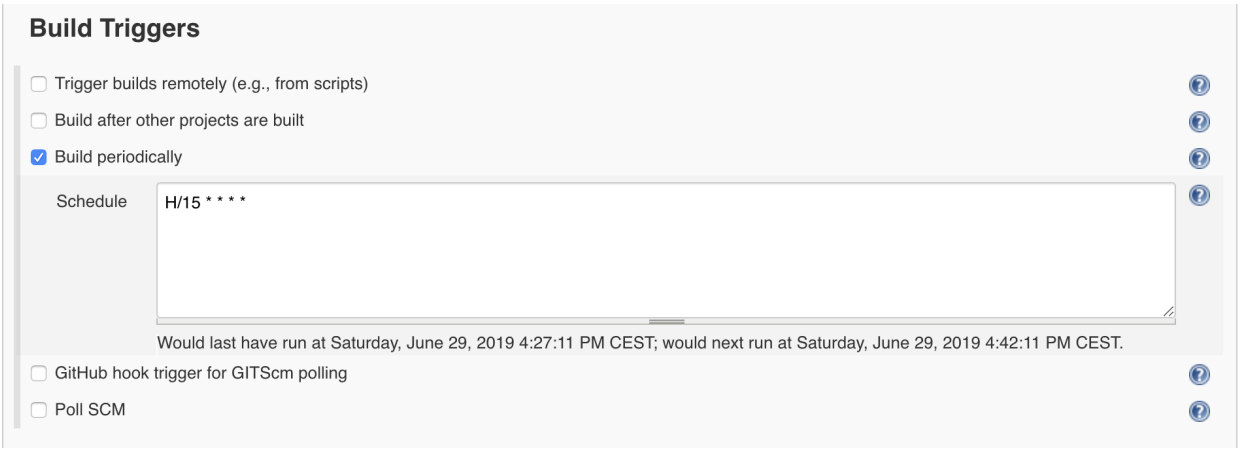
GitHub hook trigger for GITScm polling will allow you to configure your project upon a commit to GitHub. This is handy if you want to trigger load tests after your developers commit new code in a certain repository.

Poll SCM is similar to the GitHub hook trigger, but allows you to kick off Jenkins project builds based on commits in other SCM software.

Schedule your load testing

The most popular build trigger is **Build periodically**, so we'll go over how to set that up here.

First, go to your project in Jenkins and click on Configure. Scroll down to the Build Triggers section and click the checkbox next to **Build periodically**. Then you'll be prompted to enter a schedule for your test.



The schedule will need to be defined in a specific format. You can click on the ? next to the text box in Jenkins, but in general, you'll need to know that Jenkins expects five parameters separated by a whitespace or tab in this format:

minute hour dom month dow

where:

minute is a value from 0-59 that represents minutes within the hour

hour is a value from 0-23 that represents hours within the day

dom is a value from 1-31 that represents the day of the month

month is a value from 1-12 that represents the month within the year

dow is a value from 0-7 that represents the day of the week, beginning and ending with Sunday

So something like 0 0 * * * will run:

minute: every 0th minute

hour: of the 0th hour

dom: every day of the month

month: every month

dow: every day of the week

On top of this, H is a symbol that you can use to represent “hash”. Using it means allowing Jenkins to determine a schedule within your parameters that will optimise the build so that you won’t have multiple projects getting built at the same time. Jenkins recommends that you use H whenever possible, unless your intention is simultaneous execution.

If instead of the example above, we used H H * * *, this means that the build will still be triggered once a day, but the exact hour and minute will vary depending on how many other projects there are with the same schedule. In this way, you can stagger your project builds.

Here are some examples of schedules you can use, along with what they mean:

Value	Schedule
H/15 * * * *	every fifteen minutes (perhaps at :07, :22, :37, :52)
H(0-29)/10 * * * *	every ten minutes in the first half of every hour (three times, perhaps at :04, :14, :24)
45 9-16/2 * * 1-5	once every two hours at 45 minutes past the hour starting at 9:45 AM and finishing at 3:45 PM every weekday.
H H(9-16)/2 * * 1-5	once in every two hours slot between 9 AM and 5 PM every weekday (perhaps at 10:38 AM, 12:38 PM, 2:38 PM, 4:38 PM)
H H 1,15 1-11 *	once a day on the 1st and 15th of every month except December
0 2 * * 1-5	every weekday at 2:00 AM

Once you've chosen and entered your schedule, click Save. Your script will be executed according to your schedule as long as Jenkins is running on your machine.

Cloud-based load testing with Jenkins

Up until this point, I've shown you how to schedule a load test running on your Jenkins server, which may well be your local machine. If so, you'll be running your test on your local machine as well, which may or may not be ideal. What if you want to scale up your load test?

One option is to install Jenkins on each load generator you want to use, setting each one up using the instructions here.

Running on the cloud, though, is another story. Depending on the platform you're using, you may be able to integrate it into Jenkins as well. I'll go over how to achieve this using Flood.

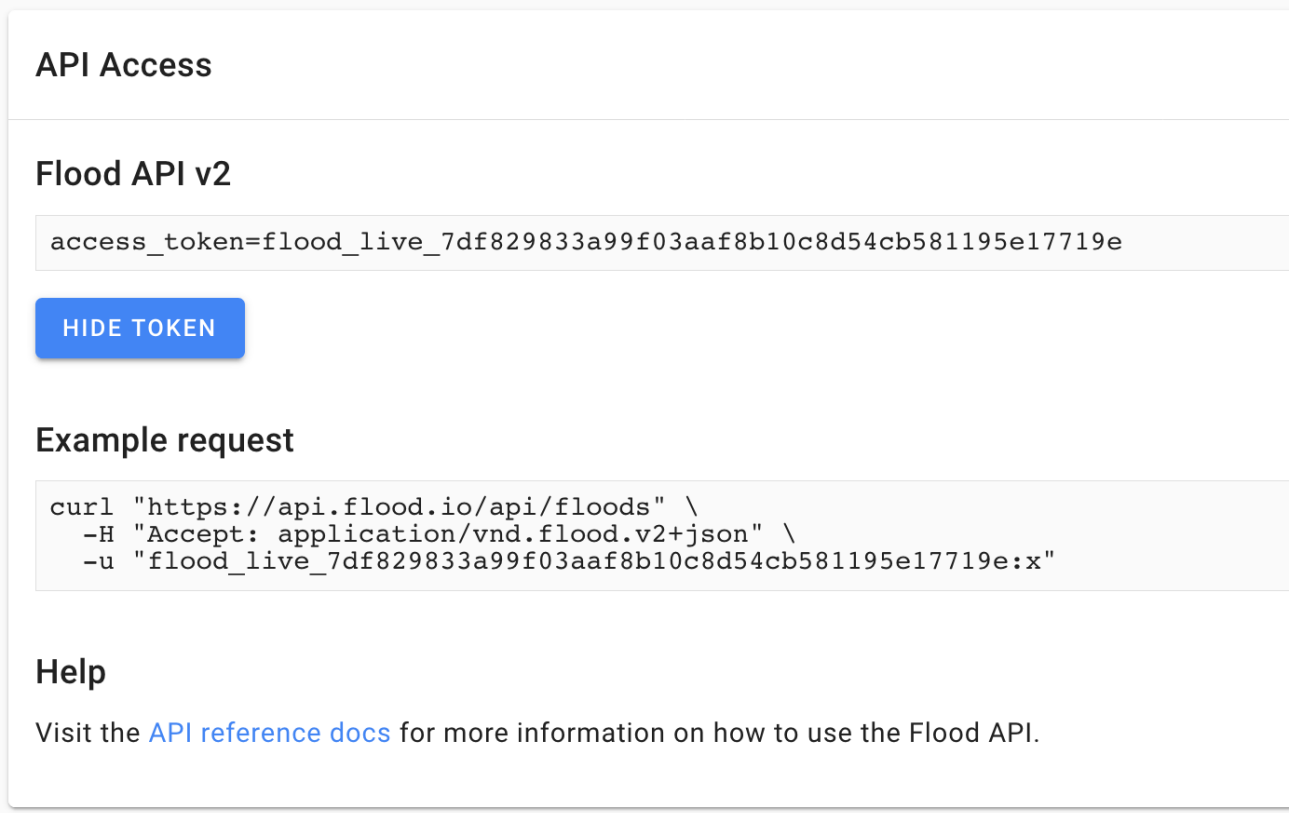
Flood has an extensive API that allows you to start and repeat grids and floods. This makes it really easy to orchestrate through Jenkins. This means that you could conceivably add in

not just a quick single-machine load test before deployment but also a full-blown load test with thousands of users across several geographical regions in the world— all executed without manual intervention.

To do this, you'll need:

1. A Flood account.

2. Your API access token, which you can get by clicking on your profile pictures once logged into Flood and then clicking API Access. Click **Reveal token** and you'll see something like this:



Copy the value of your access token. You'll need it to allow Jenkins to access your Flood account and start floods and grids on your behalf.

3. A shell script to use in Jenkins. In the Configure settings of your project on Jenkins, add a build step with **Execute shell** and enter your calls to the Flood API in that text field. Here's a sample one to get you started:

curl -u ${API_TOKEN}: -X POST https://api.flood.io/floods \

```
-F "flood[tool]=jmeter" \

-F "flood[threads]=10" \

-F "flood[privacy]=public" \

-F "flood[name]=MyTest" \

-F "flood_files[]=@jmeter-with-plugins.jmx" \

-F "flood[grids][][infrastructure]=demand" \

-F "flood[grids][][instance_quantity]=1" \

-F "flood[grids][][region]=us-west-2" \

-F "flood[grids][][instance_type]=m5.xlarge" \

-F "flood[grids][][stop_after]=60"
```

You will need to replace ${API_TOKEN} with the API access token that you revealed in Step 2 above.

Two of my colleagues have already written about this in detail, so for more information, check out:

Jason Rizio's help article on Using Jenkins to Run Continuous Load Test with Flood and

Tim Koopman's blog post on Jenkins Load Testing with Flood.

Other considerations for API load testing

API load testing is a powerful tool in your arsenal that gives you real, concrete data on what actually happens to your application servers under load. When used correctly, API load testing with industry-recognised load testing tools like JMeter and Gatling can accurately identify server-side performance bottlenecks in your application before go-live and allow you some time to fix issues before they even occur.

It's worth noting, though, that API load testing isn't the be-all and end-all of performance testing. In order to see a holistic view of application performance, it's best to pair API load testing with some front-end performance testing.

Front-end performance testing is generally less technical than back-end performance testing, and there are some fantastic tools out there that make it a matter of entering your URL and looking at the recommendations. I'm a big fan of GTMetrix for this sort of quick analysis.

Another option is to complement API load testing scripts with browser-level load testing scripts. Tools such as Flood Element and Selenium can be used to create scripts that better measure the end user experience of your application. One solid strategy is to use API load testing to generate the majority of the load and then simultaneously run a single node of browser-level testing. This hybrid approach will give you the best of both worlds— back-end and front-end performance in a tidy, cost-effective package.

Load testing's value is determined not by executing tests but by actually using the results to improve your application and inform your product roadmap. It's best looked at as a continuous process that doesn't end once a release goes live. Real load testing is an iterative and consistent way of measuring performance that is included in every development activity so that performance is baked into the project team's definition of quality.

Happy flooding!