Published in ITNEXT

Javier Ramos Follow
Jan 9 · 11 min read · ✨ · Listen

How to run distributed performance tests in Kubernetes with K6

If you love writing high performant code and also love the power of Kubernetes, I think you are going to find interesting this article where I will explore **K6**, an open source easy-to-use load testing framework which has the best support for **Kubernetes**. In this article, I will show you how to perform **load testing** natively on a Kubernetes cluster using multiple pods simulating real world traffic to test a **ElasticSearch** cluster deployed using the **ECK Operator**.

Photo by Marc-Olivier Jodoin on Unsplash

This is a hands-on article where we are going to test the performance of **ElasticSearch**. You can find the source code of this article in this **repo**.

What is K6?

K6 is part of Grafana Labs, and it is a free open source tool to perform load testing. It uses JavaScript to easily write test cases and provides many customizations to run the test cases; including running them on **Kubernetes**. The test results can be easily exported to many places like cloud storage, Prometheus, Datadog and common formats such as CSV and JSON. k6 is free, developer-centric, and extensible.

> *Using k6, you can test the reliability and performance of your systems and catch performance regressions and problems earlier. k6 will help you to build resilient and performant applications that scale.*

K6 follows the **Load Testing Manifesto**:

- Simple testing is better than no testing
- Load testing should be goal oriented
- Load testing by developers
- Developer experience is super important

- Load test in a pre-production environment

K6 can be used by developers, QA engineers or SREs for a wide range of use cases including:

- **Load Testing**: K6 is easy to use and uses very little resources. It is designed for running high load tests and includes features such as spike, stress and soak tests.
- **Browser Testing**: Using xk6-browser.
- **Chaos Testing**: K6 can simulate traffic and inject different types of faults in Kubernetes with xk6-disruptor.
- **Performance Monitoring**: With k6, you can automate and schedule to trigger tests very frequently with a small load to continuously validate the performance and availability of your production environment.

Installing K6

Installing K6 is very **easy**, it supports **Docker** and all operating Systems.

For **Mac** just run `brew install k6`.

On **Windows** run `choco install k6`.

For **Linux**:

```
sudo gpg --no-default-keyring --keyring /usr/share/keyrings/k6-archive-keyring.
echo "deb [signed-by=/usr/share/keyrings/k6-archive-keyring.gpg] https://dl.k6.
sudo apt-get update
sudo apt-get install k6
```

Or just run:

```
docker run grafana/k6
```

If you want to use any **extension**, go to this page.

Running K6

We write script using **JavaScript** which makes K6 very easy to use, event for people who do not know JavaScript. As an example, you can open an editor add create this file:

```
import http from 'k6/http';
import { sleep } from 'k6';

export default function () {
  http.get('https://test.k6.io');
  sleep(1);
}
```

In here we just do an HTTP GET requests and then sleep for 1 second. Quite simple!

To run it simple type `k6 run script.js` using the name you used to save the file. If you did not install K6 you can use docker:

```
docker run --rm -i grafana/k6 run - <script.js
```

You can add other parameters, for example, to run the test for 15 seconds using 10 virtual users, you will run:

```
k6 run --vus 10 --duration 15s script.js
```

You can learn more by checking the getting started section. You can implement *init* functions to setup whatever is required before running the tests. To learn more about the test life cycle check this page.

Hands-on!

In this section we are going to deploy **ElasticSearch**, create an index and test the **ElasticSearch** performance using K6. We will do this in a Kubernetes cluster!

Deploy ElasticSearch on Kubernetes

First, you will need a **Kubernetes** cluster. If you do not have acccess to a cloud service you can run one locally, for this I recommend **K3D** which is a very **lightweight** distribution.

To install **K3D** simply run:

```
curl -s https://raw.githubusercontent.com/k3d-io/k3d/main/install.sh | bash
```

Next, let's install **ElasticSearch** in the cluster. We will follow best practices and use a **Kubernetes Operator**. ElasticSearch has the **ECK Operator** which has all the necessary features to deploy and maintain ElasticSearch clusters in Kubernetes. You can follow the quick start guide. In a nutshell, you need to install the **CRDs** first:

```
kubectl create -f https://download.elastic.co/downloads/eck/2.5.0/crds.yaml
```

Then, install the operator itself:

```
kubectl apply -f https://download.elastic.co/downloads/eck/2.5.0/operator.yaml
```

Run the following command to verify that the operator is running:

```
kubectl -n elastic-system logs -f statefulset.apps/elastic-operator
```

Now that we have the Operator, let's create an **ElasticSearch cluster**. With the **ECK** operator it couldn't be easier. You just need to create a manifest file with the cluster details. For local testing we can create a simple single node cluster:

```
cat <<EOF | kubectl apply -f -
apiVersion: elasticsearch.k8s.elastic.co/v1
kind: Elasticsearch
metadata:
  name: quickstart
spec:
  version: 8.5.3
  nodeSets:
  - name: default
    count: 1
    config:
      node.store.allow_mmap: false
EOF
```

You can run `kubectl get elasticseark` to check if the cluster was created.

Feel free to customize your installation to meet your needs.

Awesome!, we have now an ElasticSearch cluster running on Kubernetes. Let's now install Kibana.

Deploy Kibana

To deploy **Kibana** we also use the **Operator**. You can run:

```
cat <<EOF | kubectl apply -f -
apiVersion: kibana.k8s.elastic.co/v1
kind: Kibana
metadata:
  name: quickstart
spec:
  version: 8.5.3
  count: 1
  elasticsearchRef:
    name: quickstart
EOF
```

Check the documentation for customization options.

Run `kubectl get kibana` to check if Kibana is running. Wait until the status goes green.

Now we need to connect to the Kubernetes service and port-forward the traffic. Run:

```
kubectl port-forward service/quickstart-kb-http 5601
```

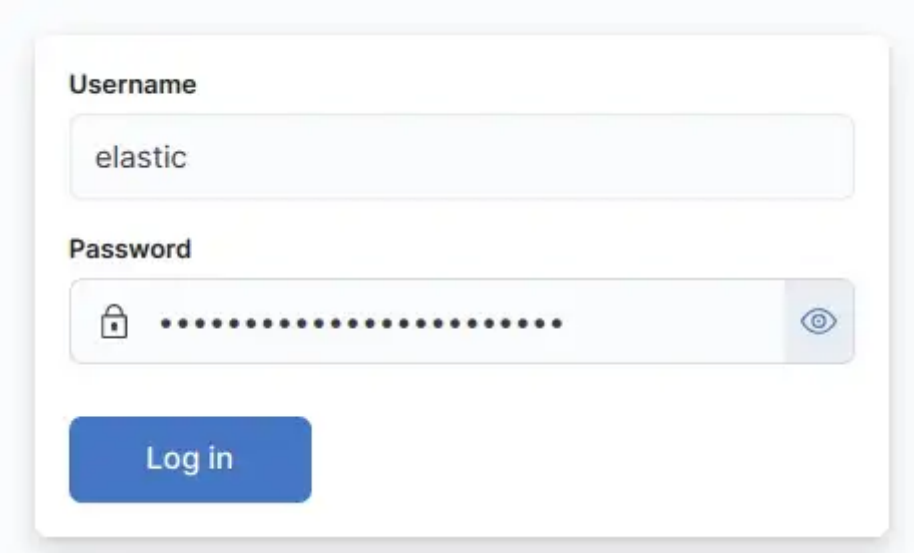
Now you can open Kibana in your browser:

https://localhost:5601

Kibana should discover the **ElasticSearch** cluster and connect to it. If you haven't customize you ElasticSearch manifest file, a default user `elastic` should have been created and the password stored in a secret. To retrive it run:

```
kubectl get secret quickstart-es-elastic-user -o=jsonpath='{.data.elastic}' | b
```

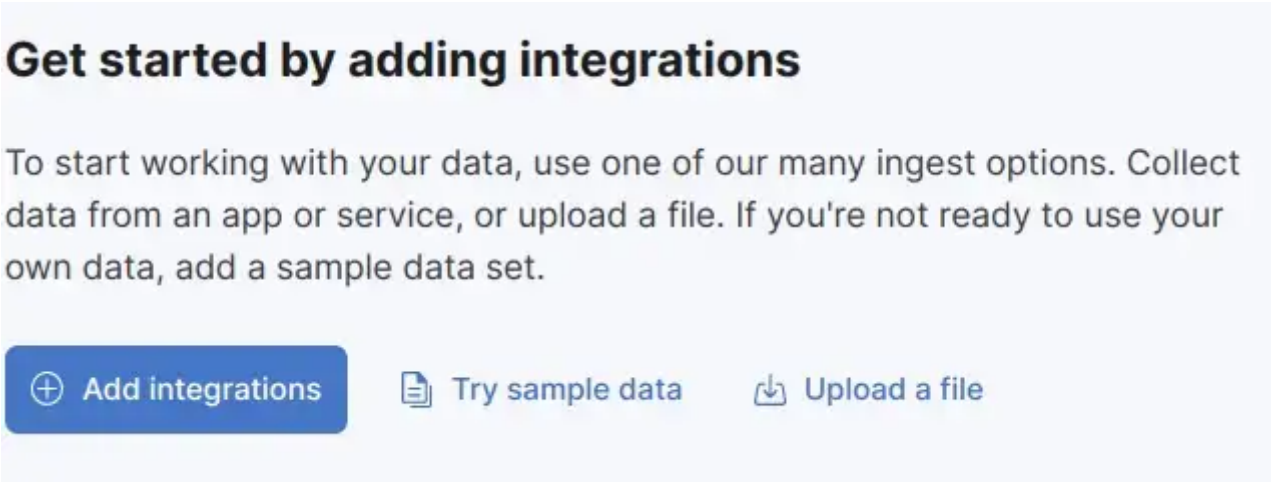
Copy the password and login into Kibana:



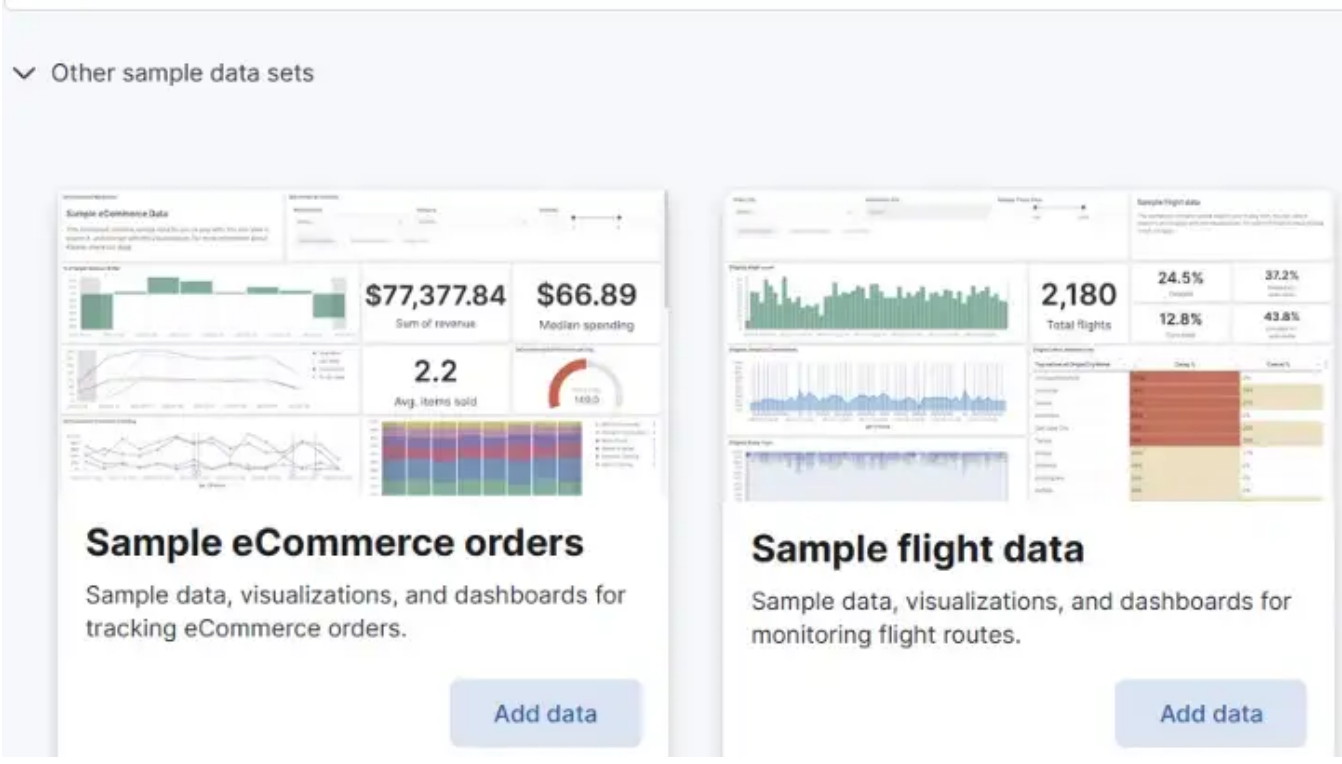
You should see now the Kibana dashboard, feel free to explore it on your own!

Add Sample Data

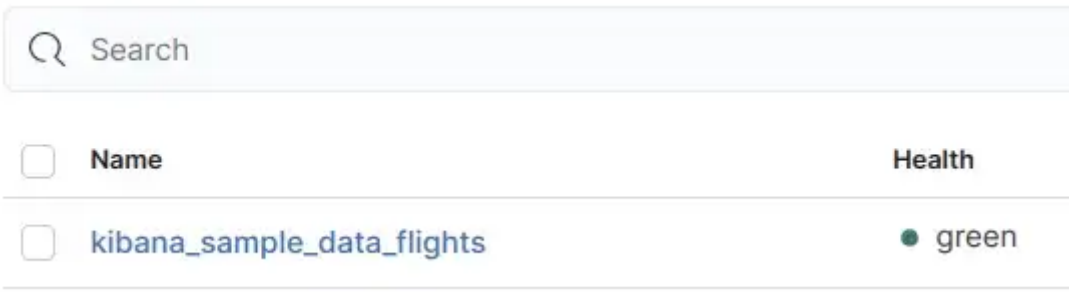
Let's add some index with some sample data for our performance tests. In the dashboard click "*Try sample data*":



Expand "*Other sample data sets*" and select "*Sample flight data*". Click "A*dd data*":



Now go to the menu and click "*Stack management*", then, in the data section click "*Index Management*". You should see an index with data:



Congratulations, you know how **ElasticSearch** + **Kibana** running on your **Kubernetes** cluster.

Let's do a quick test and run a query using Kibana Dev Tools. In the menu, scroll down to "*Management*" and click "*Dev Tools*". This will show an UI, where you can enter queries. Enter the following to search all documents:

```
GET kibana_sample_data_flights/_search
{
 "query": {
        "match_all": {}
    }
}
```

You should see the documents on the right hand side window:

```
    "hits": [
      {
        "_index": "kibana_sample_data_flights",
        "_id": "2mRPf4UBd1qcrOsWvlI9",
        "_score": 1,
        "_source": {
          "FlightNum": "9HY9SWR",
```

```
            "DestCountry": "AU",
            "OriginWeather": "Sunny",
            "OriginCityName": "Frankfurt am Main",
            "AvgTicketPrice": 841.2656419677076,
            "DistanceMiles": 10247.856675613455,
            "FlightDelay": false,
            "DestWeather": "Rain",
            "Dest": "Sydney Kingsford Smith International Airport",
            "FlightDelayType": "No Delay",
            "OriginCountry": "DE",
            "dayOfWeek": 0,
            "DistanceKilometers": 16492.32665375846,
            "timestamp": "2022-12-26T00:00:00",
            "DestLocation": {
              "lat": "-33.94609833",
              "lon": "151.177002"
            },
...
```


Writing the Test Cases

As we have seen, writing test cases is quite easy and it is done using **JavaScript.** You can find the source code in this **repo**.

We just need to create a JS file and add the test cases, in our case we will create the following `run.js` file:

```
import http from 'k6/http';
import { check } from 'k6';
import { Rate } from 'k6/metrics';

const username = __ENV.ES_USERNAME || 'elastic';
const password = __ENV.ES_PASSWORD || 'password';
const index = __ENV.INDEX || 'test';
const url = __ENV.URL || 'localhost:9200' ;

console.log("index", index, "user", username, "url", url)

const params = {
    headers: {
      'Content-Type': 'application/json',
    },
};

export let options = {
    vus: 200,
    duration: '30s',
```

```
    insecureSkipTLSVerify: true,
};


export const errorRate = new Rate('errors');

export default function () {
  const credentials = `${username}:${password}`;

  const reqUel = `https://${credentials}@${url}/${index}/_search`;

  console.log("Starting Load Test. URL: ", reqUel)

  const p1 = JSON.stringify(
    {
      "query": {
            "match_all": {}
       }
     }
  );

  let res = http.post(reqUel, p1, params);

  check(res, {
    'Q1 status is 200': (r) => r.status === 200
  }) || errorRate.add(1);

  const p2 = JSON.stringify(
    {
        "query": {
            "query_string": {
                "query": "OriginCityName:Frankfurt*"
            }
        }
    }
  );

  res = http.post(reqUel, p2, params);

  check(res, {
    'Q2 status is 200': (r) => r.status === 200
  }) || errorRate.add(1);

  const p3 = JSON.stringify(
    {
        "query": {
            "query_string": {
                "query": "Rain"
            }
        }
    }
  );
```

```
    res = http.post(reqUel, p3, params);

    check(res, {
      'Q3 status is 200': (r) => r.status === 200
    }) || errorRate.add(1);

    const p4 = JSON.stringify(
      {
          "query": {
              "query_string": {
                  "query": "test"
              }
          }
      }
    );

    res = http.post(reqUel, p4, params);

    check(res, {
      'Q4 status is 200': (r) => r.status === 200
    }) || errorRate.add(1);

    const p5 = JSON.stringify(
      {
        "query": {
            "query_string": {
                "query": "OriginCountry:DE"
            }
        }
    }
    );

    res = http.post(reqUel, p5, params);

    check(res, {
      'Q5 status is 200': (r) => r.status === 200
    }) || errorRate.add(1);



    console.log("Completed!")
  }
```

First, we just read some environment variables and use some sensible defaults.

Then, we use Basic Auth and setup the URL based on the environment variables parameters:

```
const credentials = `${username}:${password}`;
const reqUel = `http://${credentials}@${url}/${index}/_search`;
```

Next, we set the **parameters:**

```
export let options = {
```


In our case, we are going to use 200 Virtual Users and we will run the tests for 30 seconds.

We set `insecureSkipTLSVerify: true` because the operator has created self signed certificates by default and we need to skip the check, do not do this in production.

Finally, we write some HTTP POST requests to the ElasticSearch search API:

```
const p1 = JSON.stringify(
    {
      "query": {
            "match_all": {}
        }
     }
  );

  let res = http.post(reqUel, p1, params);

  check(res, {
    'Q1 status is 200': (r) => r.status === 200
  }) || errorRate.add(1);
```

First we create the body of the request using the **ElasticSearch** API. Then, we send the POST request. After that, we use K6 *check* method to make sure the status is *200*, if not we increase the Error rate that will be displayed in the report. Easy, right?

Running on Kubernetes

Now let's run our test cases in Kubernetes. First you need to install the K6 Operator.

Clone the repository and run `make deploy` to install it in your cluster:

```
git clone https://github.com/grafana/k6-operator && cd k6-operator
make deploy
```


This will install the K6 CRD that we can use to run tests in Kubernetes.

Next, we need to upload our Script as a Config Map and stored in out cluster. To do this run:

```
kubectl create configmap es-perf-test --from-file run.js
```

Or create the manifest file manually and apply it.

Create a new K6 resource using YAML:

```
apiVersion: k6.io/v1alpha1
kind: K6
metadata:
  name: k6-es-perf-test
spec:
  parallelism: 2
  script:
    configMap:
      name: es-perf-test
      file: run.js
  runner:
    env:
      - name: ES_USERNAME
        value: "elastic"
      - name: ES_PASSWORD
        value: "71GSm51vV7EC3E4Lk3p85zhY"
      - name: INDEX
        value: "kibana_sample_data_flights"
```

```
        - name: URL
          value: "quickstart-es-http:9200"
```

Enter the Environment Variables parameters that apply to you. To get the password run:

```
kubectl get secret quickstart-es-elastic-user -o=jsonpath='{.data.elastic}' | b
```

`quickstart-es-http` is the Kubernetes Service name generated when deploying **ElasticSearch** with the Operator.

Note that the *configMap* name needs to match the one you used when creating the ConfigMap.

The `parallelism` attribute is used to split the number of Virtual Users (**VUs**) among different pods. For instance, if the script calls for 40 VUs, and parallelism is set to 4, the test runners would have 10 VUs each. This is based on the **execution segments**. In our case, since we have 200 VUs and parallelism set to 2, each pod will run 100 VUs.

Save the file as `k6test.yaml` .

To run the tests in the cluster apply the manifest file:

```
kubectl apply -f k6test.yaml
```

Run `kubectl get K6` to check that the K6 resource was created.

Run `kubectl get pods` to see the 2 pods created:

```
k6-es-perf-test-initializer-nwnlp   0/1     Completed   0          19m
k6-es-perf-test-starter-49jnd       0/1     Completed   0          19m
```

```
k6-es-perf-test-1-7ltk7              0/1     Completed   0          19m
k6-es-perf-test-2-bm7pm              0/1     Completed   0          19m
```

These pods are created by the **Operator** when we apply the manifest file. It also creates some pods to initialize the tests. Note how 2 pods were created to distribute the 200 VUs in the cluster. This allows us to run distributed load tests across the cluster using many nodes, this is great for performance testing and chaos testing.

You can check the logs of one of the Pods to get the summary and the performance results.

```
     checks.........................: 100.00% ✓ 89335       ✗ 0
     data_received..................: 655 MB  22 MB/s
     data_sent......................: 28 MB   928 kB/s
     http_req_blocked...............: avg=236.81µs min=552ns    med=1.82µs  ma
     http_req_connecting............: avg=19.16µs  min=0s       med=0s      ma
     http_req_duration..............: avg=21.73ms  min=445.08µs med=14.31ms ma
       { expected_response:true }...: avg=21.73ms  min=445.08µs med=14.31ms ma
     http_req_failed................: 0.00%   ✓ 0           ✗ 89335
     http_req_receiving.............: avg=719.48µs min=7.32µs   med=23.34µs ma
     http_req_sending...............: avg=115.73µs min=3.78µs   med=10.4µs  ma
     http_req_tls_handshaking.......: avg=197.21µs min=0s       med=0s      ma
     http_req_waiting...............: avg=20.89ms  min=419.32µs med=13.72ms ma
     http_reqs......................: 89335   2972.151376/s
     iteration_duration.............: avg=167.87ms min=9.08ms   med=155.1ms ma
     iterations.....................: 17867   594.430275/s
     vus............................: 100     min=0         max=100
     vus_max........................: 100     min=100       max=100
```

You may be wondering how to get an aggregate metrics instead of checking the results of each pod individually. Metrics generated by running distributed k6 tests using the operator won't be aggregated by default, which means that each test runner will produce its own results and end-of-test summary.

To be able to **aggregate** your metrics and analyse them together, you'll either need to:

- Set up some kind of monitoring or visualisation software (such as Grafana) and configure your K6 custom resource to make your jobs output there.

- Use logstash, fluentd, splunk, or similar to parse and aggregate the logs yourself.
- You can also rely on k6 Cloud output for aggregations.

For more information check the documentation.

And that's it, we have run a cluster wide load test distributed across many pods in our Kubernetes cluster. This is only the beginning, K6 can do much more than that such as chaos testing, stress testing, API tests and can support very large tests.

Conclusion

In this article we have seen how to create a local Kubernetes cluster using **K3D**, deploy the **ECK** Operator, ElasticSearch, Kibana and the **K6** Operator. We used K6 to run distributed tests inside the cluster to test the performance of our **ElasticSearch** queries.

You can find the source code on this **repo**, you can use it as a template to test any other project in Kubernetes.

*Remember to **clap** if you enjoyed this article and **follow me** or **subscribe** for more updates!*

Subscribe to get **notified** when I publish an article and **Join Medium.com** to access millions or articles!

Elasticsearch Testing Kubernetes K 6 Load Testing