Most commonly used terms in

Author: Anshita Bhasin
LinkedIn: https://www.linkedin.com/in/anshita-bhasin/

K6 is a widely-used open-source load testing tool written in Go. It is designed for conducting load tests on APIs and Web browsers (experimental mode). It has gained significant popularity due to its straightforward installation process and user-friendly learning curve, distinguishing it from other tools in the market.

In this cheat sheet, I will cover the most used terms and terminologies used in K6

(1) Test Lifecycle in K6

In the K6 test lifecycle, a script always runs through the below 4 stages:

- ***Init***

init stage is where you write all your import files

- ***Setup function***

In the setup function, You write all the options where you set the test environment in terms of configuring the number of virtual users, defining the number of iterations, etc

- ***VU***

In this stage, you run the actual code, i.e. you define calling the HTTP requests/ opening browsers and validating the responses.

- ***Teardown function***

This is the final stage where you define the result of the setup code. Example- defining handleSummary function to generate the HTML report at the end of test execution.

(2) Virtual User (VU)

A virtual user represents a concurrent user that interacts with the application during the load test. Each VU executes the predefined test script. It is an integer value and takes 1 as the default value.

Example

```
export const options =
 {
vus: 10,
};
```

(3) Iterations

It refers to the number of times a scenario or a set of actions is repeated during a load test. For example, if you set the iterations to 10, the scenario will be executed 10 times. It takes an integer value and its default value is also 1.

Example

```
export const options = {

iterations: 10,

 };
```

(4) Duration

It is the total time for which a load test is run. It specifies the length of time the virtual users will be executing the scenario. It takes an integer value and its value can be in seconds/minutes.

Example

```
export const options = {
duration: '10s'
};

export const options = {
duration: '1m'
};
```

(5) Stages

It defines the different phases or steps of a load test, with each stage having its own duration and virtual user count. It allows the simulation of realistic load patterns during the test.

Example

```
export const options = {
stages: [
{ duration: '1m', target: 100 }, // 1 minute with 100 virtual users
{ duration: '2m', target: 200 }, // 2 minutes with 200 virtual users
{ duration: '1m', target: 0 }, // 1 minute ramping down to 0 users
], };
```

(6) Target

It specifies the number of virtual users to ramp up or down to for a specific period.

Example

```
export const options = {
 stages: [
{ duration: '3m', target: 10 }, // 3 minutes with 10 virtual users
{ duration: '5m', target: 10 },
{ duration: '10m', target: 35 },
{ duration: '3m', target: 0 },
],
};
```

(7) 90 Percentile

The 90th percentile refers to the response time below which 90% of the requests fall. It is a measure of response time performance where lower values indicate better performance. In easy terms, it represents 90% of the requests that are completed within the specified time, while the remaining 10% may take longer.

(8) 95 Percentile

The 95th percentile refers to the response time below which 95% of the requests fall. It is a measure of response time performance where lower values indicate better performance.In easy terms, it represents 95% of the requests that are completed within the specified time, while the remaining 10% may take longer.

(9) Checks

Checks are used to verify specific conditions or expectations in the response received from an HTTP request. They help validate that the application is behaving as expected during the load test.

The only difference in the case of checks is that failed test case does not halt the execution.

Example

```
export default function () {
const res = http.get('http://test.k6.io/');
check(res,
{
'Response status is 200': (r) => r.status === 200,
});
}
check(page,
{
'Verify user is logged In': () =>
page.locator('.breadcrumb-item.active').textContent() == 'Account'
});
```

(10) Metrics

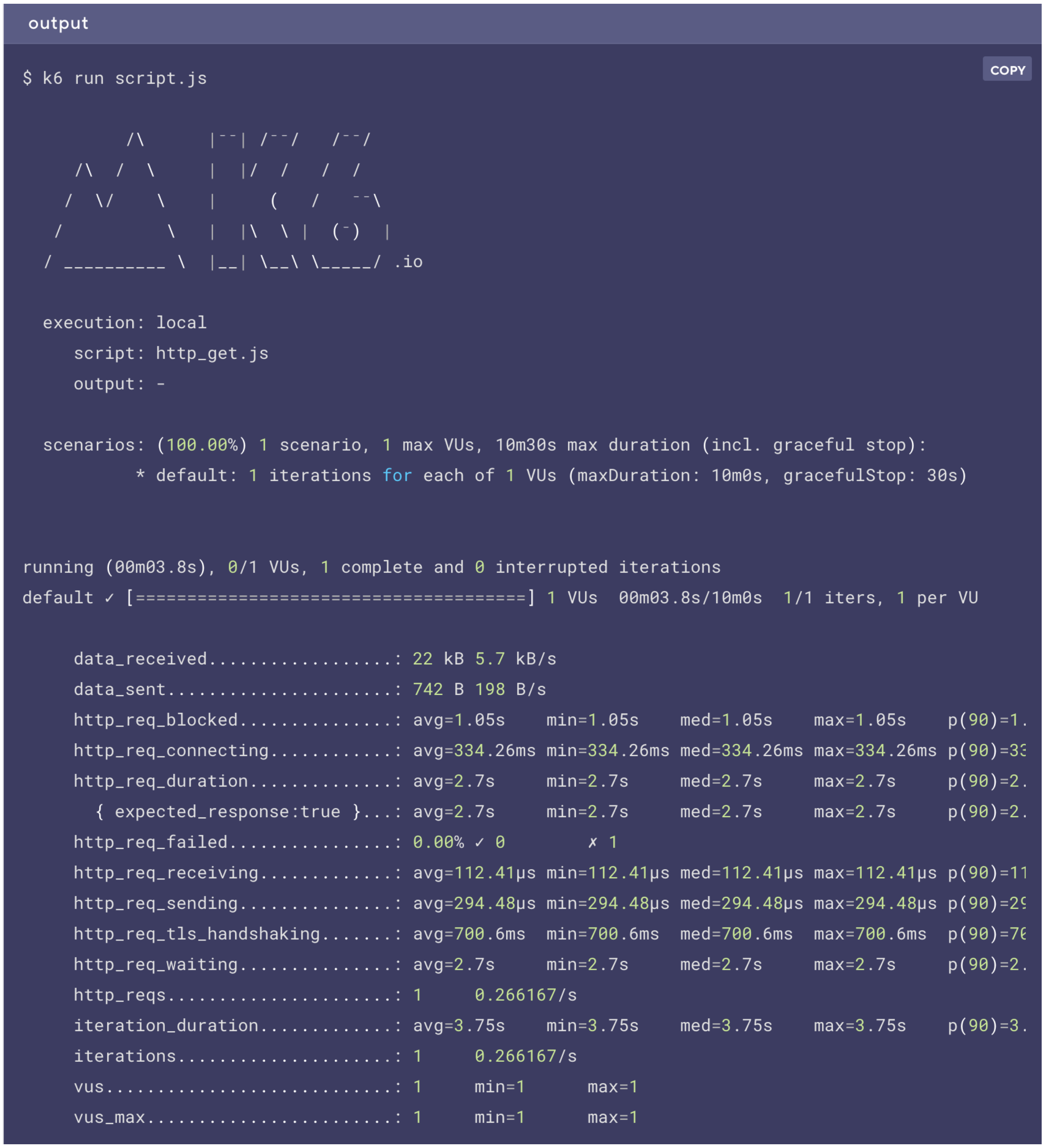
Metrics are measurements collected during the load test of the system under test. K6 provides various built-in metrics such as response time, throughput, error rate, and more.

Metrics fall into four broad types:

- ***Counters*** sum values.
- ***Gauges*** track the smallest, largest, and latest values.
- ***Rates*** track how frequently a non-zero value occurs.
- ***Trends*** calculate statistics for multiple values (like mean, mode, or percentile).

```
import http from 'k6/http';

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



```
$ k6 run script.js

          /\      |‾‾| /‾‾/   /‾‾/
     /\  /  \     |  |/  /   /  /
    /  \/    \    |     (   /   ‾‾\
   /          \   |  |\  \ |  (‾)  |
  / __________ \  |__| \__\ \_____/ .io

  execution: local
     script: http_get.js
     output: -

  scenarios: (100.00%) 1 scenario, 1 max VUs, 10m30s max duration (incl. graceful stop):
           * default: 1 iterations for each of 1 VUs (maxDuration: 10m0s, gracefulStop: 30s)


running (00m03.8s), 0/1 VUs, 1 complete and 0 interrupted iterations
default ✓ [======================================] 1 VUs  00m03.8s/10m0s  1/1 iters, 1 per VU

     data_received..................: 22 kB 5.7 kB/s
     data_sent......................: 742 B 198 B/s
     http_req_blocked...............: avg=1.05s    min=1.05s    med=1.05s    max=1.05s    p(90)=1.
     http_req_connecting............: avg=334.26ms min=334.26ms med=334.26ms max=334.26ms p(90)=33
     http_req_duration..............: avg=2.7s     min=2.7s     med=2.7s     max=2.7s     p(90)=2.
       { expected_response:true }...: avg=2.7s     min=2.7s     med=2.7s     max=2.7s     p(90)=2.
     http_req_failed................: 0.00% ✓ 0        ✗ 1
     http_req_receiving.............: avg=112.41µs min=112.41µs med=112.41µs max=112.41µs p(90)=11
     http_req_sending...............: avg=294.48µs min=294.48µs med=294.48µs max=294.48µs p(90)=29
     http_req_tls_handshaking.......: avg=700.6ms  min=700.6ms  med=700.6ms  max=700.6ms  p(90)=70
     http_req_waiting...............: avg=2.7s     min=2.7s     med=2.7s     max=2.7s     p(90)=2.
     http_reqs......................: 1      0.266167/s
     iteration_duration.............: avg=3.75s    min=3.75s    med=3.75s    max=3.75s    p(90)=3.
     iterations.....................: 1      0.266167/s
     vus............................: 1      min=1      max=1
     vus_max........................: 1      min=1      max=1
```

In the above output, all the metrics that start with http, iteration, and vu are *built-in* metrics, which get written to stdout at the end of a test. For details of all metrics, refer to the Metrics reference.

Author : Anshita Bhasin

(11) Thresholds

Thresholds are the predefined conditions or limits used to determine whether a load test has passed or failed based on specific metrics. If the performance of the system under test (SUT) does not meet the conditions of your threshold, the test finishes with a failed status.

For example, you can create thresholds for any combination of the following expectations:

- Less than 1% of requests return an error.
- 95% of requests have a response time below 200ms.
- 99% of requests have a response time below 400ms.
- A specific endpoint always responds within 300ms.
- Any conditions for a custom metric.

They allow you to set performance targets and define acceptable levels for metrics such as response time, throughput, or error rates.

Example

```
export const options = {
thresholds: {
http_req_duration: ['p(95)<500'], // Set threshold for response time
http_req_failed: ['rate<0.1'], // Set threshold for the failure rate
},
};
```

http_req_duration

http_req_duration is a metric in K6 that measures the duration or response time of an HTTP request. It represents the time taken from when the request was sent until the response was received.

http_req_failed

http_req_failed is a metric in K6 that counts the number of failed HTTP requests. It includes requests that resulted in non-2xx or non-3xx status codes.

(12) Status

Status represents the HTTP status code of a response. It indicates the success or failure of an HTTP request.

Example

```
export default function () {
const res = http.get('http://test.k6.io/');
check(res,
{
' Response status is 200': (r) => r.status === 200,
});
}
```

(13) Error

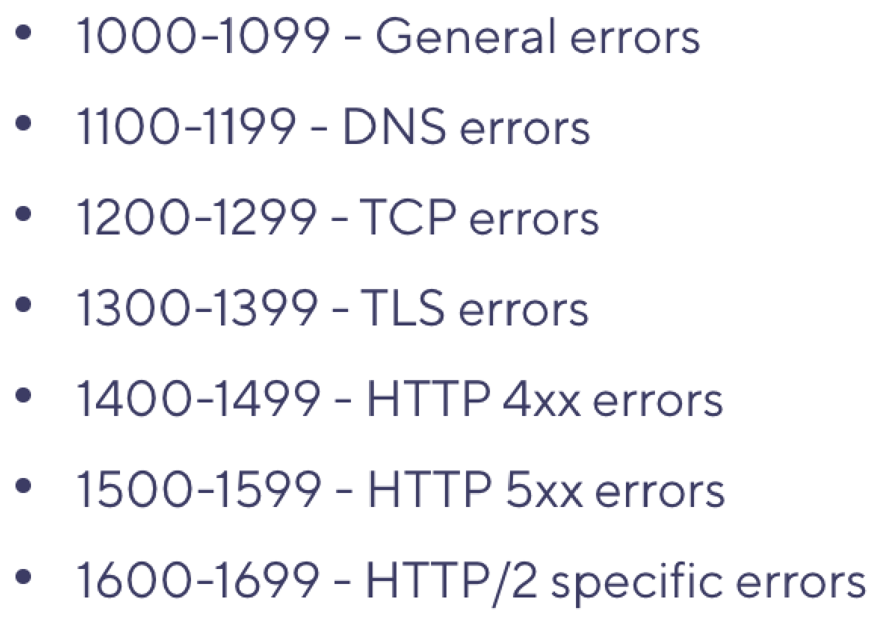
The ***error*** variable is the error encountered during an HTTP request. It includes details like the error message, stack trace, and any additional information available.

(14) Error_Code

Error codes are unique numerical identifiers that simplify the identification and handling of various application and network errors. Currently, K6 supports the error codes that are specific to errors encountered during HTTP requests but it will expand support for other protocols.

When an error occurs, its corresponding code is assigned and returned as the error_code field within the http.Response object. It is also attached as an error_code tag to associated metrics related to that request. Furthermore, the error metric tag and http.Response field retains the actual error message for more detailed information.

Example



- 1000-1099 - General errors
- 1100-1199 - DNS errors
- 1200-1299 - TCP errors
- 1300-1399 - TLS errors
- 1400-1499 - HTTP 4xx errors
- 1500-1599 - HTTP 5xx errors
- 1600-1699 - HTTP/2 specific errors

(15) Scenario

A scenario in K6 represents a specific user journey or flow during a load test. It provides detailed configuration options for virtual users (VUs) and iteration schedules in load tests.

You can define multiple scenarios within the same script, allowing each scenario to execute a unique JavaScript function independently and parallelly.

Example

```
export const options = {
scenarios: {
scenario1: {
vus: 10,
iterations: 200
},
scenario2: {
vus: 20,
 iterations: 15
},
},
};
```

(16) Requests

It refers to the number of HTTP requests made during a load test. It represents the total count of requests executed by virtual users.

Example

```
// GET Request
 import http from 'k6/http';
 export default function () {
 http.get('http://test.k6.io');
}
```

(17) Cookies

Cookies are small pieces of data stored by web browsers and sent with each request to a website. It is utilized by websites and applications to store relevant information on user devices, maintaining a stateful experience.

In K6, you can manage and handle cookies to simulate realistic user behavior during load testing.

Example

```
 import http from 'k6/http';

 export default function () {
```

```
 http.get('https://httpbin.test.k6.io/cookies', {
 cookies: {
 my_cookie: 'hello world',
},
});
}
```

(18) Results Analysis

Results analysis is the process of examining and understanding the metrics, logs, and data gathered during a load test. In K6, you can study different metrics like response times, error rates, and throughput, among others.

Results analysis is valuable for pinpointing performance issues, unusual occurrences, or areas that need enhancement in the application or system being tested.

(19) Load Distribution

Load distribution in K6 means how we divide the work among virtual users during a load test. It's like making sure everyone has a fair share of the workload.

Example

Let's say, we have 100 virtual users and we want to simulate realistic user traffic. We can control how quickly these virtual users make requests or set the number of concurrent users at different points in time.

By distributing the load in this way, we can mimic real-world situations where multiple users are accessing the system simultaneously.

This helps us understand how well the system performs under different levels of concurrent traffic and helps us identify any performance issues or bottlenecks.

(20) Load Testing

Load testing in K6 is a way to check how well a system performs when lots of people are using it at the same time.

Example1

Let’s say we have a website, and we want to see how it handles a large number of users. In K6, we can simulate virtual users who behave like real people and generate a high load on the website.

Example2

If we want to test an online shopping website. We can create virtual users in K6 that add products to their carts, browse through different pages, and complete the checkout process.

By simulating many virtual users doing these actions simultaneously, we can see if the website slows down, crashes, or handles the load smoothly and ensure it can handle the expected load when it goes live.

(21) Test Script

In K6, a test script is a piece of code that defines the actions and behavior of virtual users during a load test. It tells K6 what actions to perform, such as making HTTP requests, interacting with APIs, or simulating user behavior on a website.

The test script is written in ***JavaScript*** and serves as a set of instructions for K6 to follow during the load test.

Example

If we want to test the performance of a login page on a website. In the test script, we would define the steps to simulate user behavior, such as sending a POST request to the login endpoint with valid credentials, measuring the response time, and validating the response received.

The test script allows us to automate these actions and evaluate the system’s performance under load.

(22) Ramping

Ramping refers to gradually increasing the number of virtual users or the load over a specified period. It helps stimulate a realistic user load scenario, where the number of users accessing the system gradually increases over time instead of suddenly hitting the system with a high load.

(23) Ramping VUs

The ramping-vus executor in K6 allows for a flexible number of virtual users (VUs) to perform iterations for a set duration. It’s a convenient way to control the VUs’ behavior during different time periods.

To make things even easier, you can utilize the stages option as a shortcut for implementing this executor.

The ramping-vus executor is especially useful when you want the number of VUs to gradually increase or decrease within specific time intervals.

Example

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

export const options = {
discardResponseBodies: true,
scenarios: {
contacts: {
executor: 'ramping-vus',
startVUs: 0,
stages: [
{ duration: '20s', target: 10 },
{ duration: '10s', target: 0 },
],
gracefulRampDown: '0s',
},
},
};
```

Author : Anshita Bhasin

```
    export default function () {
    http.get('https://test.k6.io/contacts.php');
    // Injecting sleep
    // Sleep time is 500ms. Total iteration time is sleep + time to finish
request.
    sleep(0.5);
```

Bonus Tips

Command to run API test in K6

```
k6 run testName.js
```

Note: You have to be in same directory as your test case

Command to run browser test in K6

```
K6_BROWSER_ENABLED = true k6 run testName.js
```

Note: You have to be in same directory as your test case

Author : Anshita Bhasin

Conclusion:

K6 is an open-source load-testing tool that helps QAs and Developers to test the performance and scalability of their APIs and websites. The above-shared list is the most commonly used terms in K6.

Thanks for reading. Happy Learning! AB

Ref : https://k6.io/docs/

Anshita Bhasin

GitHub | LinkedIn | Twitter | Youtube