Performanse Testing using K6

For using k6, you can follow these steps:

1.Install k6: k6 is a command-line tool, so you need to install it on your machine. You can download the appropriate binary for your operating system from the official k6 GitHub repository (https://github.com/k6io/k6/releases) or use a package manager like Homebrew (for macOS/Linux) or Chocolatey (for Windows).

2.Write your test script: Create a JavaScript file that contains your test script. You can use any text editor or integrated development environment (IDE) to write the script. Make sure to import the necessary k6 modules and define the test logic within the default export function.

3.Run the test script: Open a terminal or command prompt and navigate to the directory where your test script is located. To run the test, use the following command:

```
k6 run your_test_script.js
```

Replace "your_test_script.js" with the actual file name of your test script.

5.View test results: Once the test starts running, k6 will display real-time progress and statistics in the terminal or command prompt. This includes information about the number of virtual users (VUs), requests per second (RPS), response times, and any checks or thresholds you have defined.

5.Analyze test results: After the test completes, k6 will provide a summary of the test results, including statistics such as minimum, maximum, and average response times, and the number of successful and failed requests. You can also export the results to various formats, including JSON, InfluxDB, and more.

6.Parameterize and customize your test: k6 provides various options for parameterizing and customizing your test. You can specify the number of virtual users, duration of the test, custom headers, request payloads, checks, and thresholds. Refer to the k6 documentation (https://k6.io/docs/) for detailed information on all the available features and options.

Common Use Cases:

We anticipate that most of the users are customers who log in as guests. As time goes by, the guests will fall in love with the simplicity and beauty of the system and register as users. Many users prefer to skip the registration process initially and just get what they want. However, when they reach a point where they need other capabilities rather than guest options, they will have to register.

That's why most of our use cases focus on this scenario. As the system grows, we will add more coverage for registered customers as well.

Another important use case is when many different users try to buy the same product that just became available. For example, a new Nike Air Jordan shoe will be released in our market, causing lots of customers to try to purchase it at the same time. The "EnterAsGuestAndThenCheckOut.js" script can be a perfect test for that scenario.

Some of the test use cases can include:

Enter as a guest -> add to cart -> purchase shopping cart.

Enter as a guest.

Enter as a guest -> register -> create shop -> add product.

Enter as a guest -> registration.

Enter as a guest -> register with the same username (only one success).

Enter as a guest -> register -> get info about the market.

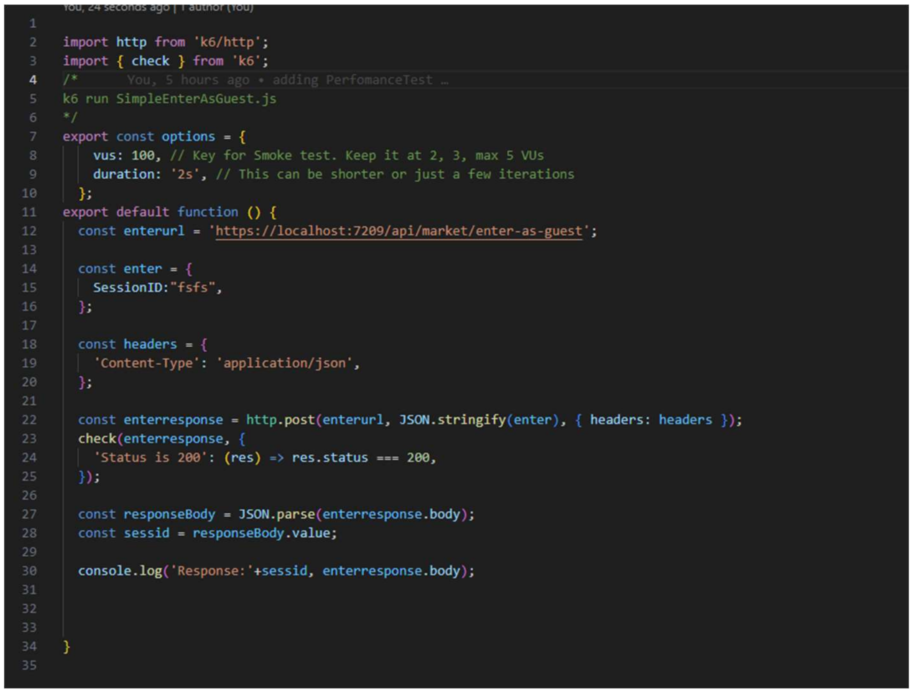
Enter as a guest -> register -> add a review to the same shop.

Enter as a guest -> register -> add an item to their basket.

Enter as a guest -> register -> create a shop.

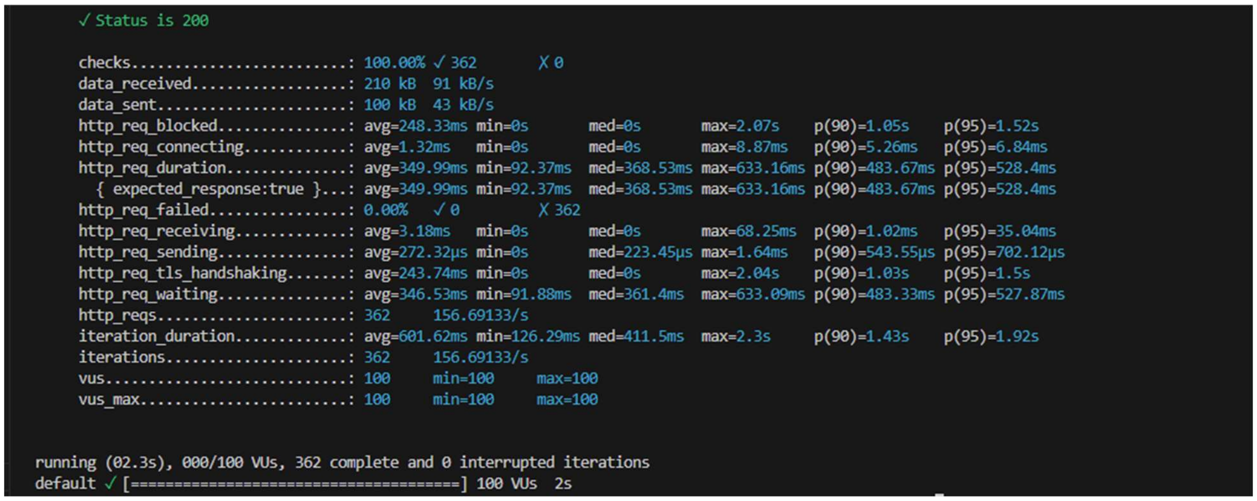
Remember, these are just examples, and you can customize and parameterize the tests according to your specific requirements and scenarios.

Explanation of how its done:
lets try to run a simple enter as guest:



```
You, 24 seconds ago | 1 author (You)

import http from 'k6/http';
import { check } from 'k6';
/*       You, 5 hours ago • adding PerfomanceTest …
k6 run SimpleEnterAsGuest.js
*/
export const options = {
    vus: 100, // Key for Smoke test. Keep it at 2, 3, max 5 VUs
    duration: '2s', // This can be shorter or just a few iterations
  };
export default function () {
  const enterurl = 'https://localhost:7209/api/market/enter-as-guest';

  const enter = {
    SessionID:"fsfs",
  };

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

  const enterresponse = http.post(enterurl, JSON.stringify(enter), { headers: headers });
  check(enterresponse, {
    'Status is 200': (res) => res.status === 200,
  });

  const responseBody = JSON.parse(enterresponse.body);
  const sessid = responseBody.value;

  console.log('Response:'+sessid, enterresponse.body);



}

```

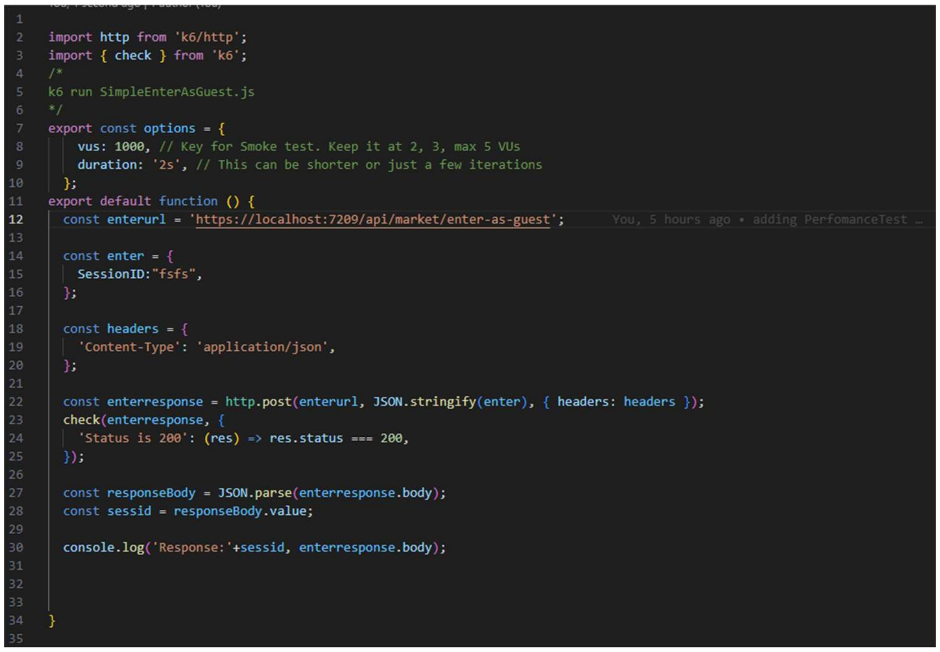
For running this we got these results:



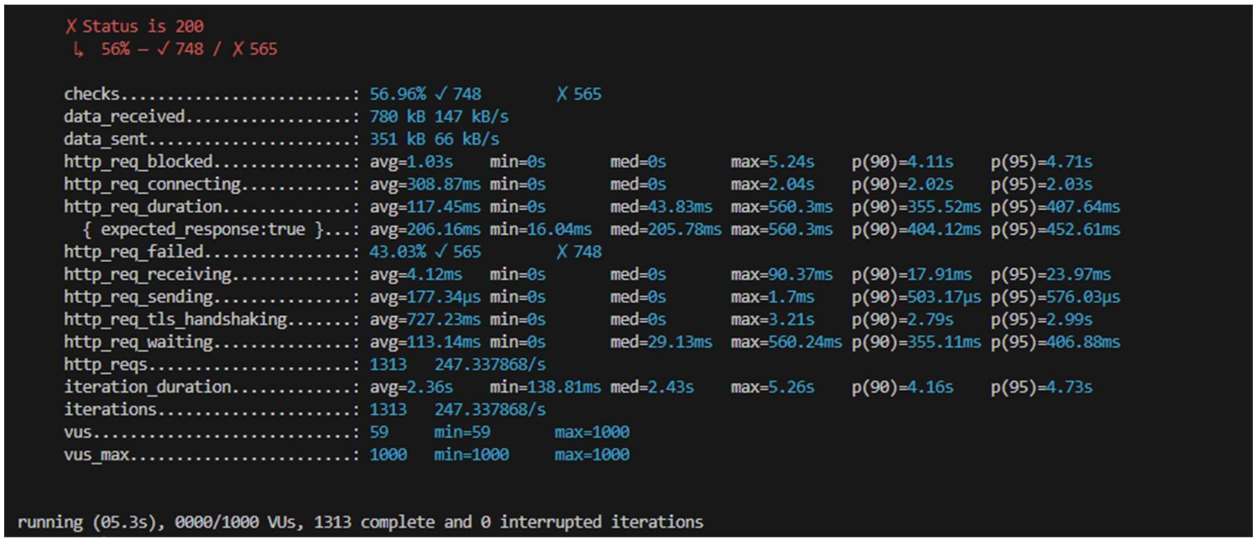
```
     ✓ Status is 200

     checks.........................: 100.00% ✓ 362       ✗ 0
     data_received..................: 210 kB  91 kB/s
     data_sent......................: 100 kB  43 kB/s
     http_req_blocked...............: avg=248.33ms min=0s       med=0s       max=2.07s    p(90)=1.05s    p(95)=1.52s
     http_req_connecting............: avg=1.32ms   min=0s       med=0s       max=8.87ms   p(90)=5.26ms   p(95)=6.84ms
     http_req_duration..............: avg=349.99ms min=92.37ms  med=368.53ms max=633.16ms p(90)=483.67ms p(95)=528.4ms
       { expected_response:true }...: avg=349.99ms min=92.37ms  med=368.53ms max=633.16ms p(90)=483.67ms p(95)=528.4ms
     http_req_failed................: 0.00%   ✓ 0         ✗ 362
     http_req_receiving.............: avg=3.18ms   min=0s       med=0s       max=68.25ms  p(90)=1.02ms   p(95)=35.04ms
     http_req_sending...............: avg=272.32µs min=0s       med=223.45µs max=1.64ms   p(90)=543.55µs p(95)=702.12µs
     http_req_tls_handshaking.......: avg=243.74ms min=0s       med=0s       max=2.04s    p(90)=1.03s    p(95)=1.5s
     http_req_waiting...............: avg=346.53ms min=91.88ms  med=361.4ms  max=633.09ms p(90)=483.33ms p(95)=527.87ms
     http_reqs......................: 362     156.69133/s
     iteration_duration.............: avg=601.62ms min=126.29ms med=411.5ms  max=2.3s     p(90)=1.43s    p(95)=1.92s
     iterations.....................: 362     156.69133/s
     vus............................: 100     min=100     max=100
     vus_max........................: 100     min=100     max=100


running (02.3s), 000/100 VUs, 362 complete and 0 interrupted iterations
default ✓ [======================================] 100 VUs  2s
```

These are the statistiscs behind the test.
as we can see all the requests have been received by our system and no errors have been ocourd means a 100 useres can send non stop enter as guesr requests to the server successfully.

Let try to cross the limit:



```
import http from 'k6/http';
import { check } from 'k6';
/*
k6 run SimpleEnterAsGuest.js
*/
export const options = {
    vus: 1000, // Key for Smoke test. Keep it at 2, 3, max 5 VUs
    duration: '2s', // This can be shorter or just a few iterations
  };
export default function () {
  const enterurl = 'https://localhost:7209/api/market/enter-as-guest';      You, 5 hours ago • adding PerfomanceTest …

  const enter = {
    SessionID:"fsfs",
  };

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

  const enterresponse = http.post(enterurl, JSON.stringify(enter), { headers: headers });
  check(enterresponse, {
    'Status is 200': (res) => res.status === 200,
  });

  const responseBody = JSON.parse(enterresponse.body);
  const sessid = responseBody.value;

  console.log('Response:'+sessid, enterresponse.body);



}
```



```
     ✗ Status is 200
      ↳  56% — ✓ 748 / ✗ 565

     checks.........................: 56.96% ✓ 748        ✗ 565
     data_received..................: 780 kB 147 kB/s
     data_sent......................: 351 kB 66 kB/s
     http_req_blocked...............: avg=1.03s    min=0s       med=0s       max=5.24s    p(90)=4.11s    p(95)=4.71s
     http_req_connecting............: avg=308.87ms min=0s       med=0s       max=2.04s    p(90)=2.02s    p(95)=2.03s
     http_req_duration..............: avg=117.45ms min=0s       med=43.83ms  max=560.3ms  p(90)=355.52ms p(95)=407.64ms
       { expected_response:true }...: avg=206.16ms min=16.04ms  med=205.78ms max=560.3ms  p(90)=404.12ms p(95)=452.61ms
     http_req_failed................: 43.03% ✓ 565        ✗ 748
     http_req_receiving.............: avg=4.12ms   min=0s       med=0s       max=90.37ms  p(90)=17.91ms  p(95)=23.97ms
     http_req_sending...............: avg=177.34µs min=0s       med=0s       max=1.7ms    p(90)=503.17µs p(95)=576.03µs
     http_req_tls_handshaking.......: avg=727.23ms min=0s       med=0s       max=3.21s    p(90)=2.79s    p(95)=2.99s
     http_req_waiting...............: avg=113.14ms min=0s       med=29.13ms  max=560.24ms p(90)=355.11ms p(95)=406.88ms
     http_reqs......................: 1313   247.337868/s
     iteration_duration.............: avg=2.36s    min=138.81ms med=2.43s    max=5.26s    p(90)=4.16s    p(95)=4.73s
     iterations.....................: 1313   247.337868/s
     vus............................: 59     min=59       max=1000
     vus_max........................: 1000   min=1000     max=1000


running (05.3s), 0000/1000 VUs, 1313 complete and 0 interrupted iterations
```

As we can see when doing the same operation for 1000 users a little bit more than half of the request handled successfully.

Thank you for reading!