COMPX341 Assignment 4

Stress-Testing Containerized Microservices

Table of Contents

[GitHub 1](#_Toc10809406)

[Unit Testing 1](#_Toc10809407)

[REQ-1 Testing 1](#_Toc10809408)

[REQ-2 Testing 1](#_Toc10809409)

[Load Testing 2](#_Toc10809410)

[Initial Docker Compose File Config 2](#_Toc10809411)

[Test’s Run 2](#_Toc10809412)

[Scenario Specifications 2](#_Toc10809413)

[Test Specifications 2](#_Toc10809414)

[Results + Discussion 3](#_Toc10809415)

[Appendix of Figures 5](#_Toc10809416)

# GitHub

<https://github.com/Woodsy1FD/StressTestingContainerizedMicroservices>

# Unit Testing

These test cases were generated from both the specifications provided, for black box testing, and from the code produced, for white box testing. The specifications can be seen in the assignment specification in the GitHub repo.

## REQ-1 Testing

|  |  |
| --- | --- |
| Equivalence Partitions | Description |
| Exception | Cannot find whether the input is either prime or not prime |
| Returns | Returns whether the number is prime or not prime |

|  |  |  |
| --- | --- | --- |
| Test Cases | Expected Partition | Expected Returns |
| 0 | Returns | 0 is not prime |
| -1, -2, null, “string” | Exception | 404 Error |
| 1, 2, 3 | Returns | {number} is prime |
| 4, 6, 8 | Returns | {number} is not prime |
| INT\_MAX | Returns | {INT\_MAX} is prime |
| “12” | Returns | 12 Is not prime |

These test cases cause 100% code coverage of the is\_prime method in app.py, as well as covering all possible test cases for the route /isPrime/<number>.

## REQ-2 Testing

|  |  |
| --- | --- |
| Equivalence Partitions | Description |
| Exception | Cannot find redis server |
| Returns | Can find redis server |

This should always return, even if the set that is stored in redis is empty. The only way this fails is if the client cannot connect to the redis image. There are no inputs to test here.

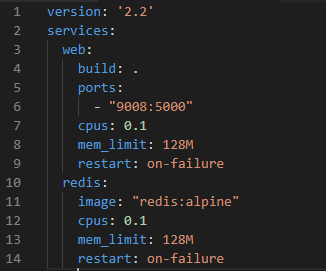
|  |  |  |
| --- | --- | --- |
| Test Cases | Expected Partition | Expected Returns |
| 2 of the same prime numbers | Returns | Size of list returned should be 1 (No duplicates) |
| 1 prime number and 1 non-prime number | Returns | Size of list returned should be 1 |
| 1 non-prime number | Returns | Size of list returned should be 0 |
| 2 prime numbers | Returns | Size of list returned should be 2 |

These test cases cause 100% code coverage of get\_primes\_redis in app.py.

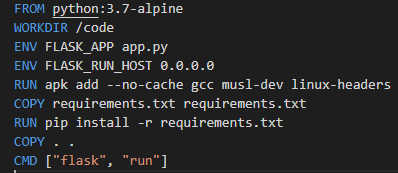
These test cases were all ran manually against the application, and all returned as expected. The application has been fully tested. An addition in future would be to add an automated testing environment using Flasks built in test suite.

# Load Testing

## Initial Docker Config



*Figure 1: Docker Compose file*



*Figure 2: Dockerfile for web container*

The code in Figure 1 is the exact docker compose specification used by this application, and Figure 2 being the Dockerfile configuration for the web container.

In the tests run in the next section, the cpus field in the web service is the only field that is changed in the docker compose file between tests.

## Test’s Run

### Scenario Specifications

|  |  |
| --- | --- |
| Scenario | Description |
| 1 | Repeatedly decides if the number 2147483647 is prime by invoking the app’s isPrime URI |
| 2 | First invokes the isPrime URI for all numbers between 1 and 100; then, it repeatedly invokes the primesStored URI of the app |

Scenarios are in the GitHub repo as Scenario1.jmx and Scenario2.jmx.

### Test Specifications

|  |  |
| --- | --- |
| Test | Description |
| 1 | Initial Scenario 1 Test Config |
| 2 | Initial Scenario 2 Test Config |
| 3 | Scenario 1 Test Config with 20% CPU on Web container |
| 4 | Scenario 2 Test Config with 20% CPU on Web container |
| 5 | Scenario 1 Test Config with 50% CPU on Web container |
| 6 | Scenario 1 Test Config with 50% CPU on Web container |
| 7 | Scenario 1 Test Config with 50% CPU on Web container and 300ms delay |
| 8 | Scenario 2 Test Config with 50% CPU on Web container and 300ms delay |
| 9 | Scenario 1 Test Config with 50% CPU on Web container and 1s delay |
| 10 | Scenario 2 Test Config with 50% CPU on Web container and 1s delay |

The specific test configs for Tests 7 through 10 are also in the GitHub repo as Test\*.jmx.

These 10 experiments were run against the environment to see how the changing variables would affect the average response time and through put of the application.

## Results + Discussion

*Figure 3: Results for Scenario 1 Testing*

For Figure 1, we can clearly see that the average response time is decreasing for each of the tests run, however, it is very interesting to see that the throughput doesn’t increase after CPU usage stops increasing (after Test 5). This would imply that the limiting factor for throughput in Scenario 1 is the sheer amount of brute force the system can apply.

Interestingly, the response time of the system continues to increase as a timer delay is added to the system. We can model this using Littles Law:

e.g. Using Test 5 data of a throughput λ of 2600 req/s, and an average response time W of 1.136s

L = λW = 2600 \* 1.136 = 2953 requests in the system.

Now using Test 9 data where 1s of wait time was added to the system.

λ = ~2600 req/s, W = 0.2s

L = λW 2600 \* 0.2 = 520 requests in the system.

From these calculations, we can see that adding a wait time to each thread in the testing, reduces the number of requests in the system, causing the response time to increase dramatically, as the CPU will have more time to process responses as required!

*Figure 4: Results for Scenario 2 Testing*

For Figure 2, we see that similarly to Figure 1, increasing the amount of CPU usage allowed for the web server, increases the throughput and decreases the response time of the application. However, the polarizing difference with this system is that the throughput majorly decreases as wait time is introduced to the system.

Using Little’s Law to calculate L for Test 6:

L = λW; λ = 8362 req/s, W = 0.343s

L = 8362 \* 0.343 = 2688 requests in the system.

Using Little’s Law to calculate L for Test 10:

L = λW; λ = 3011 req/s, W = 0.016s

L = 3011 \* 0.016s = 48.7 requests in the system.

From these calculations using Littles Law, we can see that introducing the wait timer to the system detriments the system, as the number of requests being sent to the system is dramatically lower due to this lower wait time. The system is easily able to service the requests as they come as the CPU is not being utilized as heavily, hence the quick response time. This shows that this is not a very effective load test, but also shows that if the system only required a throughput of 3000 requests per second, this would be an ideal set of parameters. This is because 0.016s is below the considered ‘human noticeable’ response rate of 0.1s.

# Appendix of Figures



*Figure 5: Scenario 1 run with initial commands*

*Figure 6: Scenario 2 run with initial commands*



*Figure 7: Scenario 1 run with 20% CPU on Web*



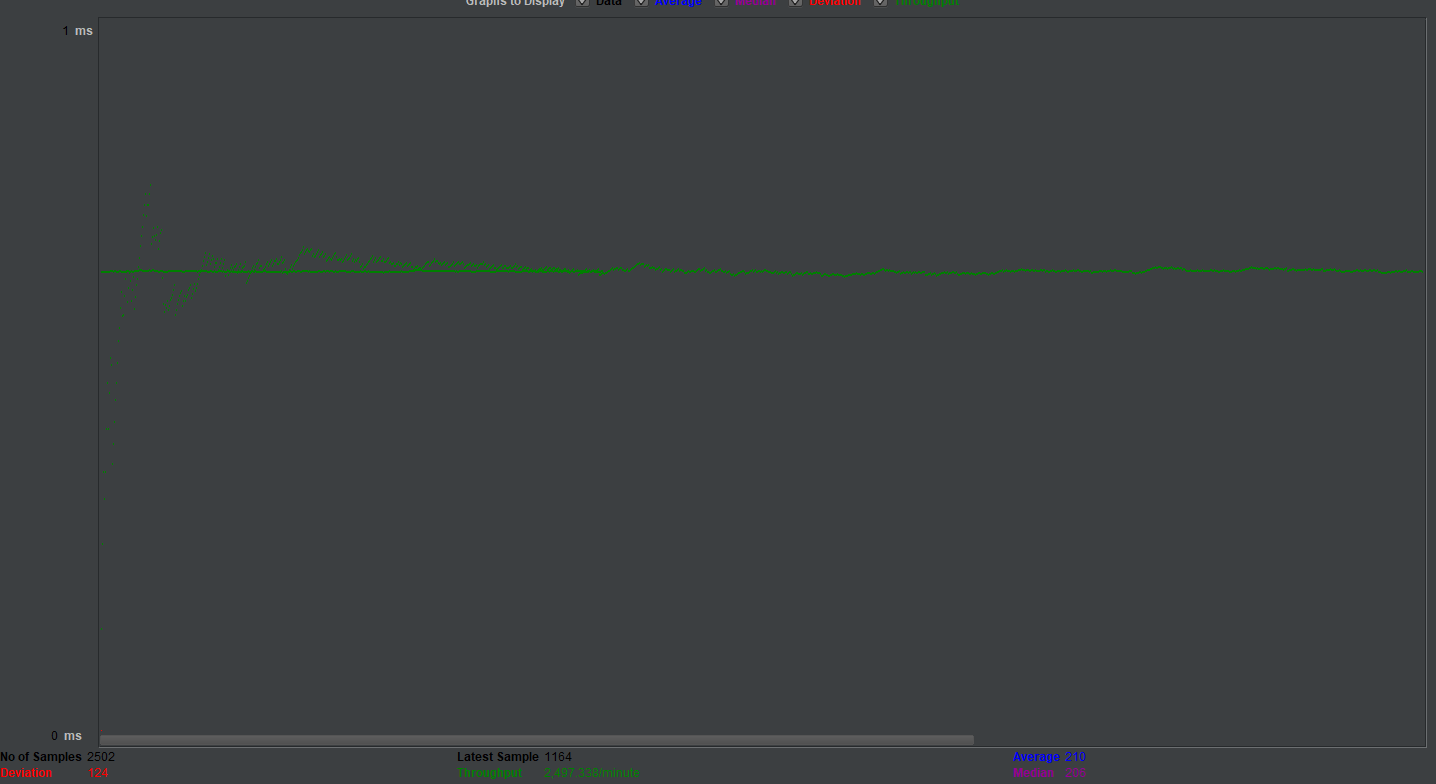
*Figure 8: Scenario 2 run with 20% CPU on Web*

*Figure 9: Scenario 1 run with 50% CPU on Web*

*Figure 10: Scenario 2 run with 50% CPU on Web*

*Figure 11: Scenario 1 run with 50% CPU and 300ms delay time*

*Figure 12: Scenario 2 run with 50% CPU and 300ms delay time*

*Figure 13: Scenario 1 run with 50% CPU’s and 1s delay time*

*Figure 14: Scenario 2 run with 50% CPU’s and 1s delay time*