Containerized python application
COMPX341-19A

Pablo Ramón Guevara - 1517675

The objective of this assignment was to create and containerize a python application, with flask and docker.

Project URL: https://github.com/praguevara/containerized-primes

For the application, I used these components:

- Flask as a web framework.
- Waitress as a more performant web server than Flask's default one.
- Sympy for its Miller–Rabin primality test. This is a probabilistic method much faster than repeated divisions to check if the number is a pseudo-prime, but Sympy states that it's valid for all n < 2^64-1, which fits the requirements.
- Redis for caching prime numbers. If a number is determined as prime by Sympy it is stored into Redis, so that if another request asks for the same number it's not necessary to compute it again.
- Docker and docker compose, which are used to containerise the application and configurate system settings.

To install them inside the container it's sufficient to write them line by line into *requirements.txt* and then run `pip install -r requirements.txt` in the container.

The command to build and launch the containers with docker compose is `docker-compose build && docker-compose up`.

Once the application was up and ad-hoc tested I changed the percentage of CPU assigned to 0.1, 0.5 and 1 to run some stress tests.

- 2147483647: repeatidly call */isPrime/2147483647*.
- Random 100: repeatidly call */isPrime/x*, where $x \in \mathbb{N} \cap [1, 100]$.

These are the results measured with JMeter:

```
Throughput: requests per second.
Median: median average response time in seconds.
L: median request count in the queue.
```

- 0.1 CPU:
  - 2147483647:
    - Throughput: 52.4 req s
    - Median response time: 0.904 s
    - L = 52.4 req s / 0.904 s = 58 req
  - Random 100:
    - Throughput: 52.0 req s
    - Median: 0.989 s
    - L = 52.0 req s / 0.989 s = 52.6 req
- 0.5 CPU:
  - 2147483647:
    - Throughput: 377.3 req s
    - Median response time: 0.119 s

        - L = 377.3 req s / 0.119 s = 3171 req
    - Random 100:
        - Throughput: 345.3 req s
        - Median: 0.124
        - L = 345.3 req s / 0.124 s = 2785 req
- 1 CPU:
    - 2147483647:
        - Throughput: 1211.9 req s
        - Median response time: 0.045 s
        - L = 1211.9 req s / 0.045 s = 26900 req
    - Random 100:
        - Throughput: 961.1 req s
        - Median response time: 0.046 s
        - L = 1211.9 req s / 0.046 s = 26300 req

According to Little's Law: L = λ / μ.

We can observe that the throughput doesn't scale linearly:

Test	CPU	Expected	Real	Ratio
2147483647	0.1	121.19	52.4	0.43
Random	0.1	96.11	52.0	0.54
2147483647	0.5	605.95	377.3	0.62
Random	0.5	480.55	345.3	0.72
2147483647	1.0	1211.9	1211.9	1.0
Random	1.0	961.1	961.1	1.0

If we were to scale the application, it would be more efficient to scale it up instead of scaling out.