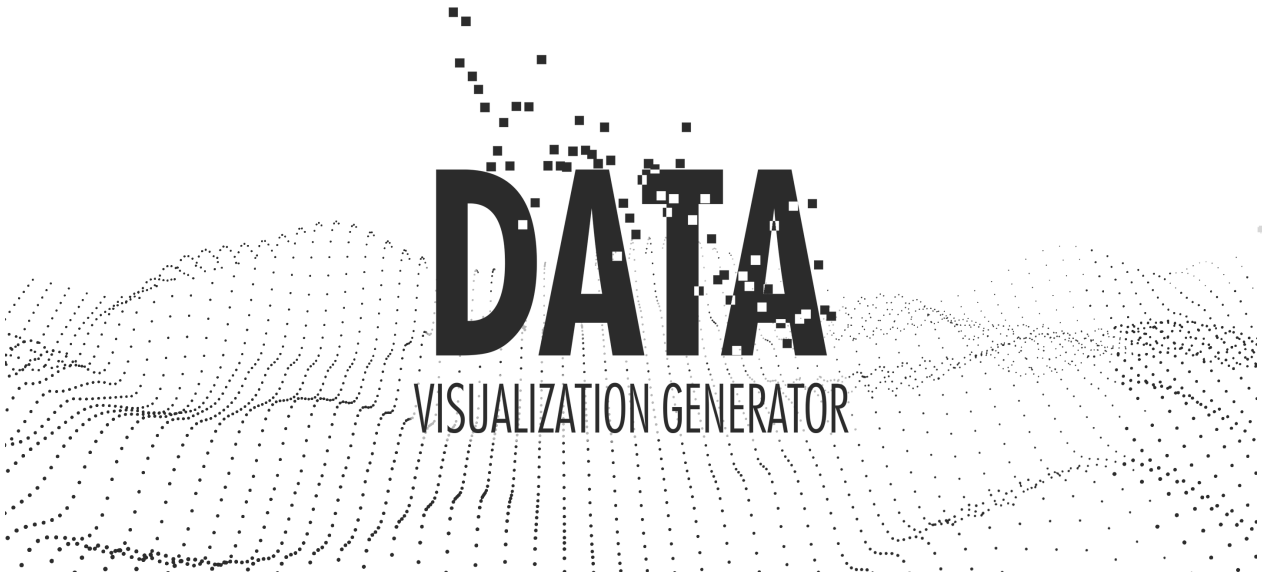
Data Visualization Generator

Testing Policy

Doofenshmirtz Evil Inc

COS 301 - 2020

Marco Lombaard u18026975
Elna Pistorius u18010319
Phillip Schulze u18171185
Byron Tomkinson u18042717
Gian Uys u18052569



Contents

1 Introduction

This document will outline the testing policies that were followed throughout the lifetime of the project. It will show the types and methods used for testing the code as well as give examples of some tests.

2 Objectives

Through the course of developing the application, testing had to be done and enforced by everyone on the team. Testing the application will ensure all features work as expected and if something went wrong, the tests will help to find the problem and fix it.

Tests need to pick up on errors that were not found in the development of the functions. It needs to be extensive and it needs to test all aspects of a function to find areas where it might not function properly.

3 Types of testing

The different types of tests we have implemented for our software application is the following.

- **Unit Testing** is when individual components (units) of the system gets tested to ensure that the basic logic of the unit works. The tests use stubs/mock data as parameters and/or outputs. Unit tests can be run at different granularities ranging from testing the program/class to testing individual functions.
- **Integration Testing** is when different components (units) gets tested together to see if they integrate as expected. Integration Testing takes place after all related unit tests are done, to ensure that the individual units are not causing the problems.
- **Continuous Testing** is when tests are run automatically as part of the development process. As code are put/merged together, the tests are run to get immediate feedback on the success of the merge.
- **Non-functional Testing** is the testing of a software application or system for its non-functional requirements (performance, usability, reliability, security, etc): the way a system operates, rather than specific behaviours of that system.

4 Methods

A conjunction of **Agile testing**, **Black-Box testing** and **White-Box** testing was used in the development of the application. The Agile design principle was followed hence all software were tested as they were developed. Every time a user story is created an acceptance criteria is defined, this drives testing and validation of the user stories, **Test-Driven Development (TDD)** is a type of Agile testing method. TDD is typically used on unit and component tests — which can be done with automated testing tools. TDD makes sure the features are working as they should be. Examples of this type of testing is Continuous testing and Unit testing.

White box testing was used to test the internal structure, design and coding of the software and verifies the flow of input-output, and also used to improve design, usability and security. Examples of White box testing is Unit Testing.

Black box testing was used to examine the functionality of the application without peering into its internal structures or workings. Examples of Black box testing is Integration testing and also Non-functional testing, this type of black box testing is not related to testing of specific functionality, but non-functional requirements such as performance, reliability, usability and security..

5 Coverage Criteria

- **Function coverage** - Unit tests need to cover all the functions to ensure that each function works as expected.
- **Statement coverage** - Tests need to cover all the source code to ensure that no code goes untested.

- **Condition coverage** - Tests should cover all possible scenarios of conditions. For example, the ‘if’ and ‘else’ part should be tested.

6 Required Resources

- Required Software for the tests to run:
  - JEST
  - NodeJS
  - Arachni
  - Apache JMeter
- Time
  - Tests should not run longer than 5 minutes on localhost
  - Due to CircleCI running tests on pull requests across the internet, it should not take more than 10 minutes to complete the tests, depending on the server load.

7 Description of the Test Process

After a specific function is coded, the person responsible for the function is in charge of writing several unit tests for that function and ensuring that the test passes.

All tests that will be written will comply with a strict and uniform structure that will give enough detail of what the test is actually testing. All tests should be short, concise and to the point, while still providing enough information in the report so that people running the tests are sure which tests are run and on what part of the software it is running.

Testing will be done through the JEST testing framework since JEST has a wide variety of available tests and works well with React (our front-end framework). Tests will also be run automatically by CircleCI on pull requests to ensure that the software is in an acceptable state before merging into a different branch. This will ensure that bugs are identified before merging it into a branch that should be in a deployable state at all times.
The logs of CircleCI tests can be found here CircleCI. Non-functional tests will be automated using JMeter and Arachni, this will test the Performance, Reliability, Availability and Security of our system.

When doing tests, it should be clear that the test is either a Unit, Integration or a Non-Functional test. We have divided the tests in to three directories, **int**, **unit** and **nf**.

8 Structure of Tests

8.1 Unit Testing

Unit testing is a WhiteBox testing technique performed by developers, where individual units or software components are tested. This type of testing isolates a section of code and verifies that it performs correctly and as expected. A unit may be an individual function, method, procedure, module, or object.

Unit tests for the Node.js server is performed using the JEST library. A typical unit test would look like:

```
describe('Test API-Key generation', () => {
    test('Test properties of generated key', () => {
        const apikey = Authentication.generateApiKey();
        expect(apikey.length).toBe(Authentication.apikeyLength);
        apikey.split('').forEach((char) => expect(Authentication.apikeyCharacters.includes(char)));
        expect(apikey).toEqual(expect.stringMatching(/^([a-zA-Z0-9_-]){20}$/));
    });

    test('Test uniqueness of generated key', () => {
        const apikey = Authentication.generateApiKey();
        let keyCounter = 0;
        Object.keys(Authentication.loggedUsers).forEach((key) => {
            if (key === apikey) keyCounter++;
        });
        expect(keyCounter).toBe(0);
    });
});
```

Node.js Unit test that passed:

```
phillipstemmlar:~/cos/capstone$jest unit/authentication
 PASS  tests/unit/authentication.test.js
  Test User Authentication Process
    ✓ Test Single-User authenticate and unauthenticate (1 ms)
    ✓ Test Multi-User authenticate and unauthenticate (1 ms)
    ✓ Test Pruning All user-Sessions (2 ms)
    ✓ Test Pruning of invalid User-Sessions
    ✓ Getting Authenticated User info from Multiple Users (2 ms)
  Test API-Key generation
    ✓ Test properties of generated key (3 ms)
    ✓ Test uniqueness of generated key
  Test validation Functions
    ✓ Test Email Validation (1 ms)
    ✓ Test Name Validation (1 ms)
    ✓ Test Password Validation (3 ms)

Test Suites: 1 passed, 1 total
Tests:       10 passed, 10 total
Snapshots:   0 total
Time:        0.67 s
Ran all test suites matching /unit\/authentication/i.
phillipstemmlar:~/cos/capstone$
```

Node.js Unit test that failed:

```
phillipstemmlar:~/cos/capstone$jest unit/authentication
 FAIL  tests/unit/authentication.test.js
  Test User Authentication Process
    ✓ Test Single-User authenticate and unauthenticate (2 ms)
    ✓ Test Multi-User authenticate and unauthenticate (1 ms)
    ✓ Test Pruning All user-Sessions (1 ms)
    ✓ Test Pruning of invalid User-Sessions (1 ms)
    ✓ Getting Authenticated User info from Multiple Users (2 ms)
  Test API-Key generation
    × Test properties of generated key (4 ms)
    ✓ Test uniqueness of generated key
  Test validation Functions
    ✓ Test Email Validation (2 ms)
    ✓ Test Name Validation
    ✓ Test Password Validation (1 ms)

  ● Test API-Key generation › Test properties of generated key

    expect(received).toEqual(expected) // deep equality

    Expected: StringMatching /^([a-z0-9_-]){20}$/
    Received: "Gd4CgSjLUPb65lTmev1s"

      147 |             expect(apikey.length).toBe(Authentication.apikeyLength);
      148 |             apikey.split('').forEach((char) => expect(Authentication.apikeyCharacters.includes(char)));
    > 149 |             expect(apikey).toEqual(expect.stringMatching(/^([a-z0-9_-]){20}$/));
          |                            ^
      150 |     });
      151 |
      152 |     test('Test uniqueness of generated key', () => {

      at Object.<anonymous> (tests/unit/authentication.test.js:149:18)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 9 passed, 10 total
Snapshots:   0 total
Time:        0.494 s, estimated 1 s
Ran all test suites matching /unit\/authentication/i.
phillipstemmlar:~/cos/capstone$
```

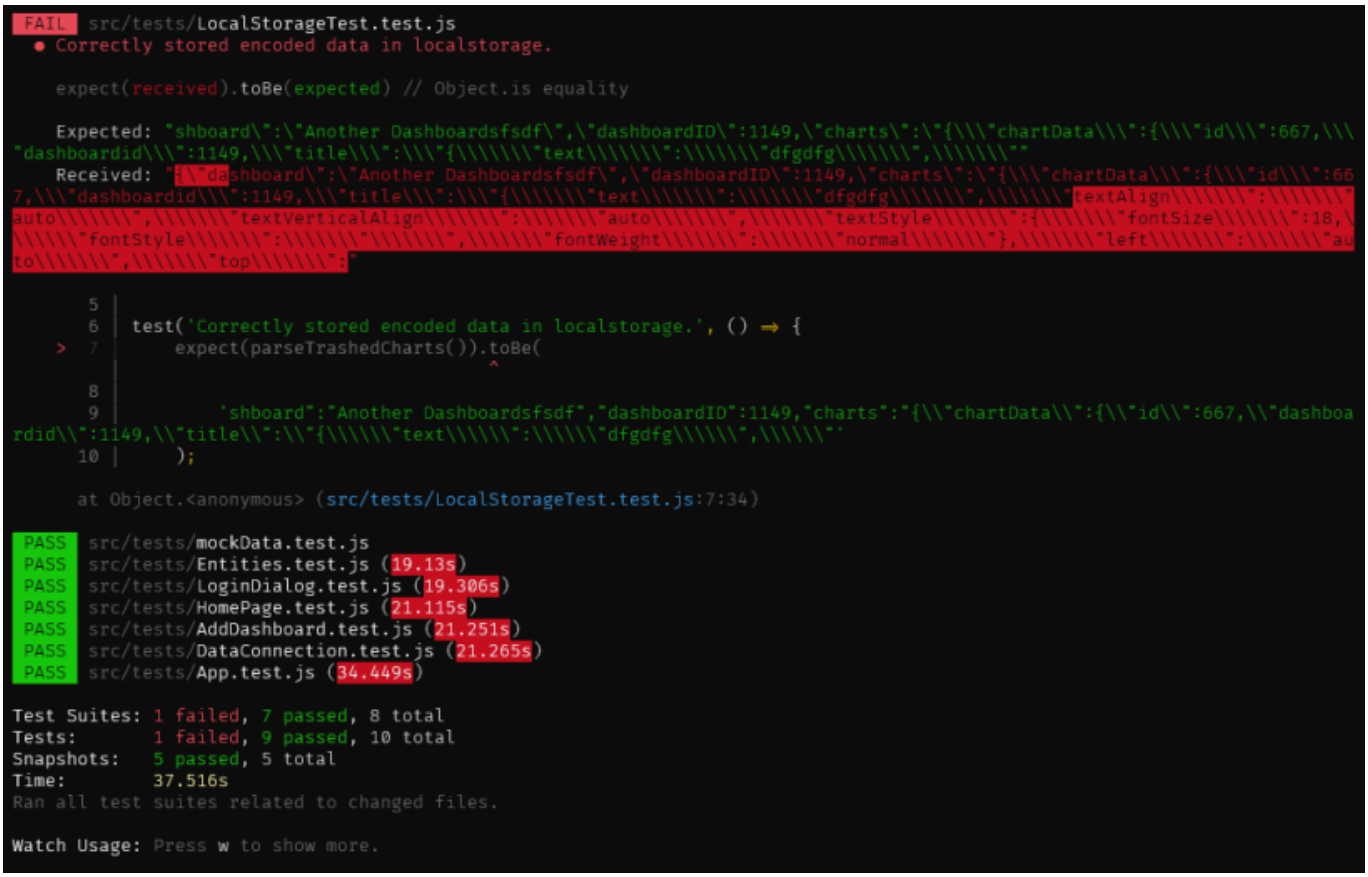
Unit tests for the React Web Application is performed using the JEST library and the JEST-DOM library. A typical unit test would look like:

```
import React from 'react';
import AddDashboard from '../components/AddDashboard';
import renderer from 'react-test-renderer';


Object.defineProperty(window, p: 'matchMedia', attributes: {
    writable: true,
    value: jest.fn().mockImplementation( fn: query => ({
        matches: false,
        media: query,
        onchange: null,
        addListener: jest.fn(), // deprecated
        removeListener: jest.fn(), // deprecated
        addEventListener: jest.fn(),
        removeEventListener: jest.fn(),
        dispatchEvent: jest.fn(),
    })),
});

describe('AddDashboard component', () => {
    it('AddDashboard renders correctly', () => {
        const rendered = renderer.create(<AddDashboard/>);
        expect(rendered.toJSON()).toMatchSnapshot();
    });
});
```

React Web Application Unit test that passed:

```
 PASS  src/tests/mockData.test.js
 PASS  src/tests/LocalStorageTest.test.js (9.451s)
 PASS  src/tests/HomePage.test.js (13.27s)
 PASS  src/tests/AddDashboard.test.js (13.434s)
 PASS  src/tests/Entities.test.js (13.537s)
 PASS  src/tests/LoginDialog.test.js (14.286s)
 PASS  src/tests/DataConnection.test.js (13.791s)
 PASS  src/tests/App.test.js (33.664s)

Test Suites: 8 passed, 8 total
Tests:       10 passed, 10 total
Snapshots:   5 passed, 5 total
Time:        36.431s
Ran all test suites related to changed files.

Watch Usage: Press w to show more.
```

React Web Application Unit test that failed:



8.2 Integration Testing

Integration testing is a BlackBox testing technique where software modules are integrated logically and tested as a group. The purpose of this level of testing is to expose defects in the interaction between these software modules when they are integrated. Integration Testing focuses on checking data communication amongst these modules.

Integration tests for the system is performed using the JEST library. A typical integration test would look like:

```
test('Test user authentication', () => {
    return Rest.registerUser(
        F_NAME,
        L_NAME,
        EMAIL,
        PASSWORD,
        (user) => {
            return Rest.loginUser(
                EMAIL,
                PASSWORD,
                (user) => {
                    expect(user.email).toBe(EMAIL);
                    expect(user.firstname).toBe(F_NAME);
                    expect(user.lastname).toBe(L_NAME);
                    expect(user).toHaveProperty('apikey');
                },
                (err) => {}
            );
        },
        () => {}
    );
});
```

Integration test that passed:

```
phillipstemmlar:~/cos/capstone$jest int/sequential/restController
 PASS  tests/int/sequential/restController.test.js (16.496 s)
  Testing user management
    ✓ Test registration of a new user (2712 ms)
    ✓ Test registration when the user already exists (3375 ms)
    ✓ Test user authentication (842 ms)
  Testing with an existing user
    Test data source management
      ✓ data source list defaults to empty
      ✓ Adding a data source (1 ms)
      ✓ Removing a data source
    Test dashboard management
      ✓ dashboard list defaults to empty
      ✓ Adding a dashboard (1 ms)
      ✓ Updating a dashboard
      ✓ Removing a dashboard (1 ms)
    Test graph management
      ✓ graph list defaults to empty
      ✓ Adding a graph (1 ms)
      ✓ Updating a graph
      ✓ Removing a graph

  console.log
    Clossing DB connecion...

      at Function.close (controllers/database/databaseController.js:83:11)

  console.log
    DB connecion is now closed.

      at controllers/database/databaseController.js:86:24

Test Suites: 1 passed, 1 total
Tests:       14 passed, 14 total
Snapshots:   0 total
Time:        16.524 s
Ran all test suites matching /int\/sequential\/restController/i.
phillipstemmlar:~/cos/capstone$
```

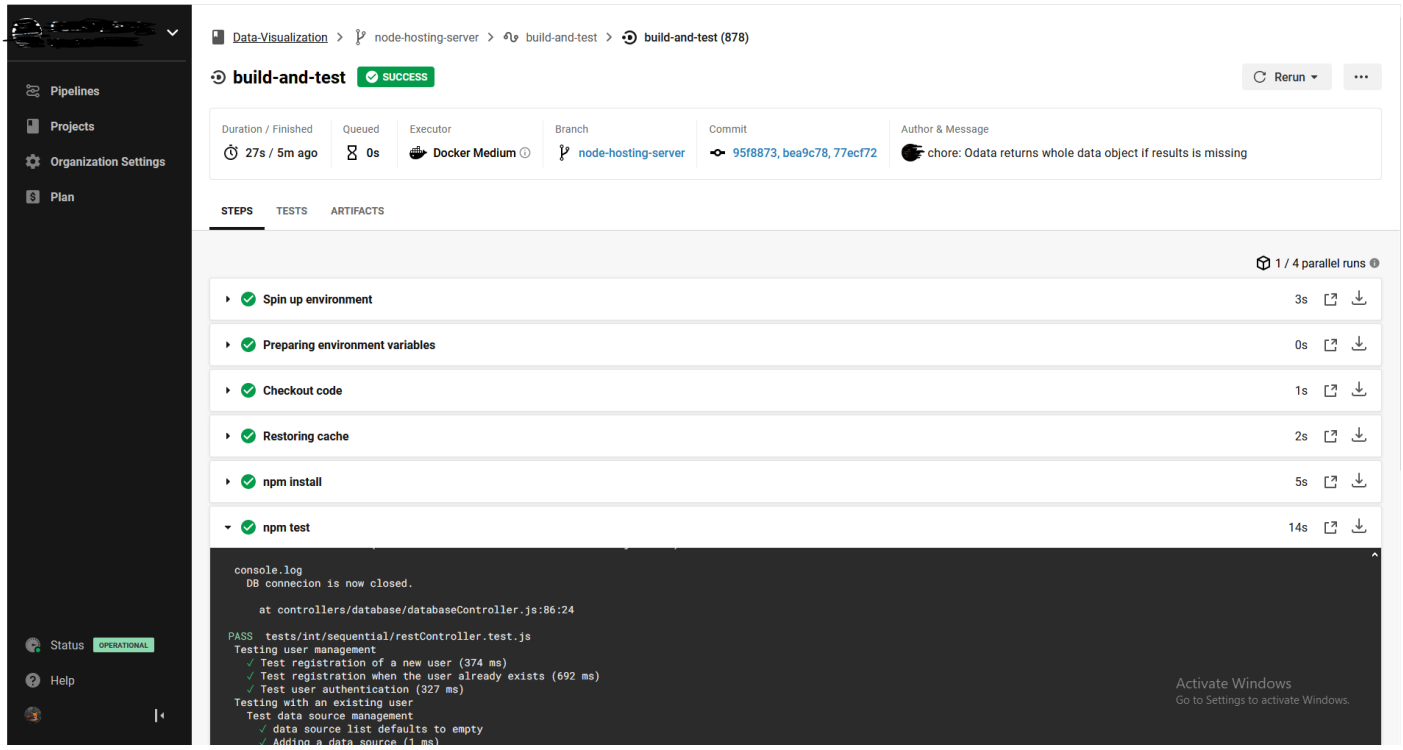
Integration test that failed:

```
phillipstemmlar:~/cos/capstone$jest int/sequential/restController
 FAIL  tests/int/sequential/restController.test.js (13.577 s)
  Testing user management
    ✓ Test registration of a new user (2482 ms)
    ✓ Test registration when the user already exists (3289 ms)
    × Test user authentication (542 ms)
  Testing with an existing user
    Test data source management
      ✓ data source list defaults to empty (1 ms)
      ✓ Adding a data source
      ✓ Removing a data source (1 ms)
    Test dashboard management
      ✓ dashboard list defaults to empty
      ✓ Adding a dashboard (1 ms)
      ✓ Updating a dashboard
      ✓ Removing a dashboard
    Test graph management
      ✓ graph list defaults to empty (1 ms)
      ✓ Adding a graph
      ✓ Updating a graph (1 ms)
      ✓ Removing a graph

  ● Testing user management › Test user authentication

    expect(received).toBeTruthy()

    Received: false

      114 |
      115 |     test('Test user authentication', () => {
    > 116 |             expect(false).toBeTruthy();
          |                           ^
      117 |             return Rest.registerUser(
      118 |                     F_NAME,
      119 |                     L_NAME,

      at Object.<anonymous> (tests/int/sequential/restController.test.js:116:17)

  console.log
    Clossing DB connecion...

      at Function.close (controllers/database/databaseController.js:83:11)

  console.log
    DB connecion is now closed.

      at controllers/database/databaseController.js:86:24

Test Suites: 1 failed, 1 total
Tests:       1 failed, 13 passed, 14 total
Snapshots:   0 total
Time:        13.605 s, estimated 15 s
Ran all test suites matching /int\/sequential\/restController/i.
phillipstemmlar:~/cos/capstone$
```

8.3 Continuous Testing

For continuous testing we used **CircleCI**. This allows us to test any code after it has been committed, and to see which of our tests fail. To do this, we set up a config.yml file in a .circleci folder which is read by the CircleCI service in order to perform testing. As soon as changes are committed to any branch, CircleCI will run these tests and display whether or not they succeeded via a flag on the commit details. This flag will be a "check" for success and a "cross" for failure.

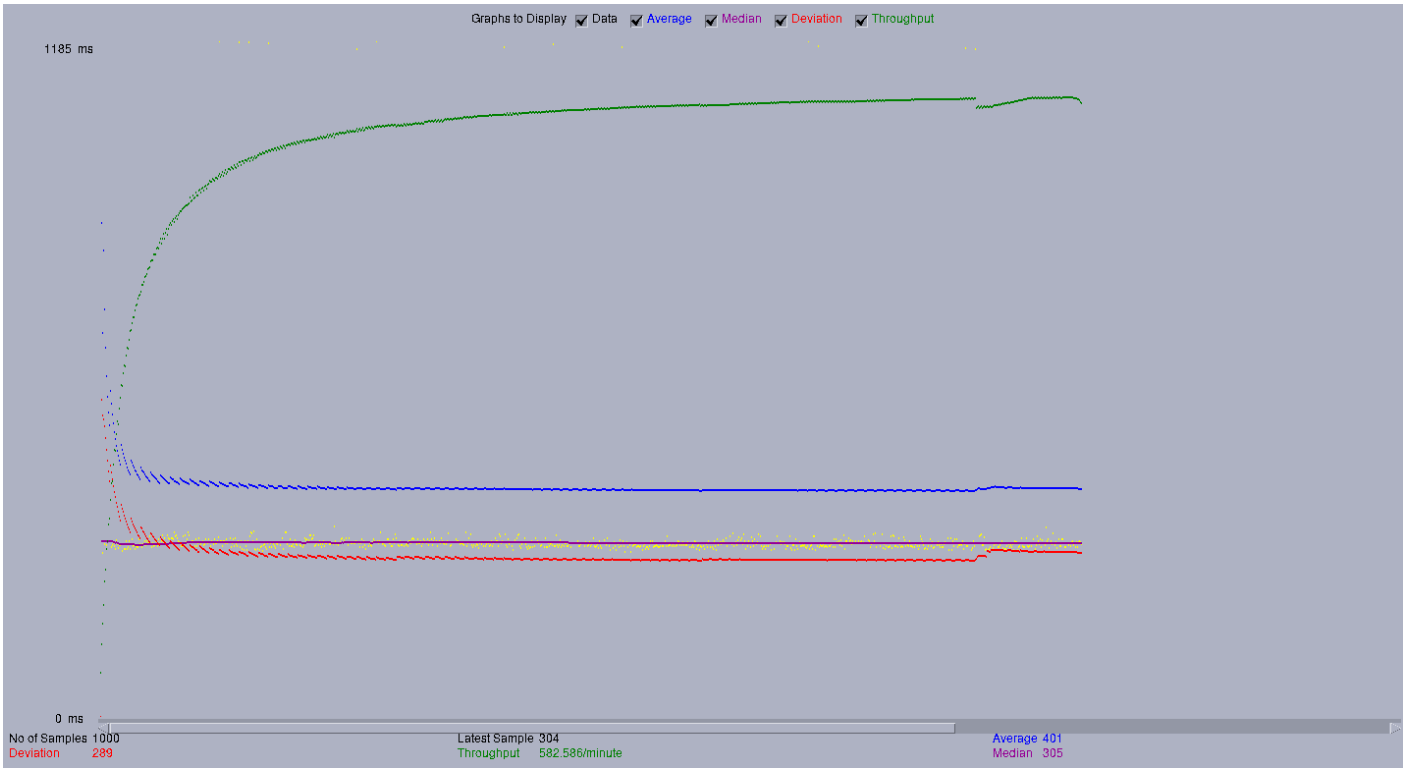
In order to see more details about the tests, the CircleCI app can be accessed, where all tests for all branches will be displayed. Clicking on a test will bring you to a details screen, where you can view each step in the testing process and see where tests failed, if any.



8.4 Non-Functional Testing

8.4.1 Performance

To test the performance and of our system we used **Apache JMeter**, that tests the functional behavior and measures performance of an application. It works by simulating a group of users sending requests to a target server, and returns statistics that show the performance/functionality of the target server/application via tables, graphs, etc. Below is an example of running a performance test on one of our systems endpoints, in this example we ran 100 threads (number of users connected to the web app).



- **Blue** indicates the average value for the total number of samples sent.
- **Red** indicates the standard deviation.
- **Green** indicates the throughput rate (This is responsible for the total number of requests per minute that the server handled)

In this test, the throughput of the our application is **582.586/minute**. This means that our application can handle **582.586 requests per minute**. A higher throughput means better system performance, and stated the amount of concurrent transactions a system can process per second. If a system can handle this 582.586 requests per minute it means that the system is available for each of these requests. The deviation of our system is **289**, the lower the deviation is from the average the better the server performance.

8.4.2 Usability tests

Usability testing was done to evaluate the usability of our product with real users. We selected three experienced users and two novice users to test the system and tasks were given to them to complete. We observed where they encountered problems with the tasks and then similar problems that were encountered were noted.

Usability Tests

Tasks

1. View charts suggestions
2. Create Dashboard with charts
3. Restore deleted chart

Users

1. Experienced users (3)
2. Novice users (2)

Questions to ask

1. Rate difficulty on scale 1 to 10
2. Anything you found difficult?
3. Would you change anything and if so, how and why?

Experienced User Testing Data

Experienced User 1

View chart suggestions

Difficulty?

2

Experience?

Nothing, said it was intuitive enough

Change? If yes, how and why?

N/A

Create Dashboard with charts

Difficulty?

5

Experience?

Thought you could add new graphs from the dashboard

Change? If yes, how and why?

Have option to add new charts from the dashboard menu

Restore deleted chart

Difficulty?

4

Experience?

Didn't see the trash page but restoring was easy.

Change? If yes, how and why?

Make trash section on dashboards.

Experienced User 2

View chart suggestions

Difficulty?

5

Experience?

Don't like the selecting of entities

Change? If yes, how and why?

Maybe have a default selection where I can immediately generate suggestions

Create Dashboard with charts

Difficulty?

3

Experience?

Nothing wrong

Change? If yes, how and why?

Maybe give more information on the dashboard.

Restore deleted chart

Difficulty?

1

Experience?

Easy

Change? If yes, how and why?

N/A

Experienced User 3

View chart suggestions

Difficulty?

2

Experience?

Couldn't select the generate suggestions button properly

Change? If yes, how and why?

Would make that easier to see and select.

Create Dashboard with charts

Difficulty?

3

Experience?

Dashboards were easy to access.

Change? If yes, how and why?

N/A

Restore deleted chart

Difficulty?

5

Experience?

Did not see it on the sidebar. Was looking on the dashboards page.

Change? If yes, how and why?

Maybe trash icon on dashboards section.

Novice User Testing Data

Novice User 1

View chart suggestions

Difficulty?

4

Change?

Doesn't like all the steps to get to suggestions.

Was a bit slow in figuring what the data source page was about.

Change? If yes, how and why?

Make instructions about what to do on data source page clearer

Create Dashboard with charts

Difficulty?

4

Experience?

Would like more colour and make graphs look more appealing.

Change? If yes, how and why?

Just add more colour.

Restore deleted chart

Difficulty?

1

Experience?

Not difficult

Change? If yes, how and why?

N/A

Novice User 2

View chart suggestions

Difficulty?

4

Change?

Confused at data sources.

Change? If yes, how and why?

Make instructions at data sources clear.

Create Dashboard with charts

Difficulty?

3

Experience?

Easy to access.

Change? If yes, how and why?

N/A

Restore deleted chart

Difficulty?

7

Experience?

Also missed the icon on the side, took him a while to find it.

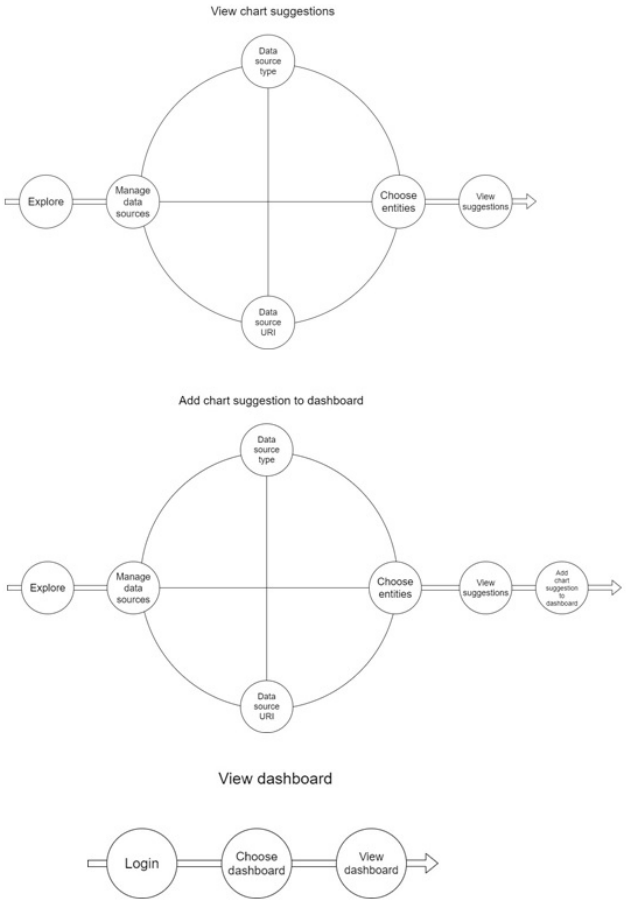
Change? If yes, how and why?

Move 'trash' icon.

Pain points

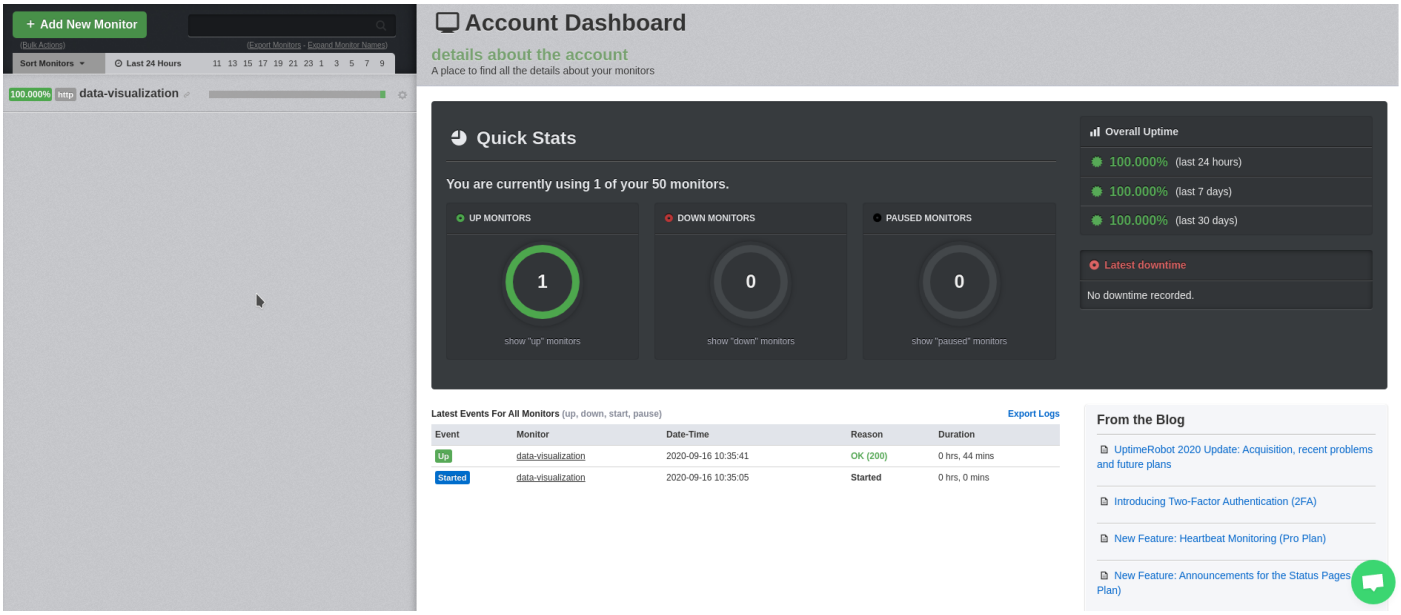
- The process to generate graph suggestions is a bit lengthily.
- Not enough functionality on dashboards page
- 'Trash' icon not easily noticed.

Task Models



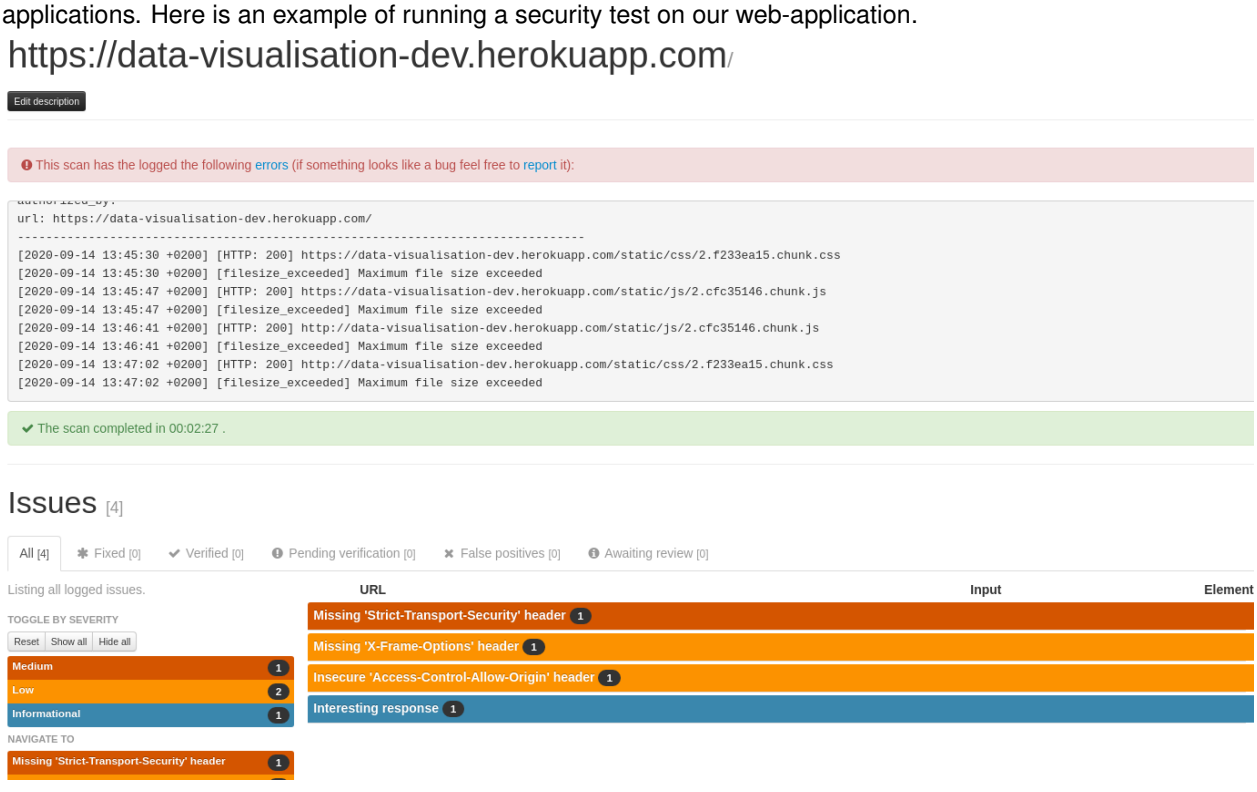
8.4.3 Reliability

Uptime is a measure of system reliability, expressed as the percentage of time a machine, typically a computer, has been working and has been available. We used **Uptime Robot** to monitor our applications uptime. When the website goes down a email will be sent to notify us, so that necessary steps can be taken. Uptime percentage is a common metric used to determine the reliability of a web server. An uptime of 99.9% is a reasonable goal for a web server. Here is a screenshot of our Uptime Robot dashboard.



8.4.4 Security

To test the security of our system we used **Arachni**, which is a tool that allows you to assess the security of web applications. Here is an example of running a security test on our web-application.



As we can see there are three issues, one that has **medium** severity and two that has a **low** severity. So after tests are run and issues are found it is the developers responsibility to look in to these issues as soon as possible to ensure that no malicious attack occur to the system. After the developer has fixed these issues then a security test has to be run again. As can be seen from below security issues were fixed.

https://data-visualisation-dev.herokuapp.com/

Edit description

This scan has the logged the following errors (if something looks like a bug feel free to report it):

```
authorized_by:
url: https://data-visualisation-dev.herokuapp.com/
------------------------------------------------------------------------------
[2020-09-14 15:53:05 +0200] [HTTP: 200] https://data-visualisation-dev.herokuapp.com/static/js/2.66e20dc4.chunk.js
[2020-09-14 15:53:05 +0200] [filesize_exceeded] Maximum file size exceeded
[2020-09-14 15:53:25 +0200] [HTTP: 200] https://data-visualisation-dev.herokuapp.com/static/css/2.f233ea15.chunk.css
[2020-09-14 15:53:25 +0200] [filesize_exceeded] Maximum file size exceeded
[2020-09-14 15:54:09 +0200] [HTTP: 200] http://data-visualisation-dev.herokuapp.com/static/js/2.66e20dc4.chunk.js
[2020-09-14 15:54:09 +0200] [filesize_exceeded] Maximum file size exceeded
[2020-09-14 15:54:28 +0200] [HTTP: 200] http://data-visualisation-dev.herokuapp.com/static/css/2.f233ea15.chunk.css
[2020-09-14 15:54:28 +0200] [filesize_exceeded] Maximum file size exceeded
```

The scan completed in 00:02:10 .

Issues [1]

All [1] Fixed [0] Verified [0] Pending verification [0] False positives [0] Awaiting review [0]

Listing all logged issues.

TOGGLE BY SEVERITY

Reset Show all Hide all

Informational 1

NAVIGATE TO

Interesting response 1

URL	Input	Element
Interesting response 1		