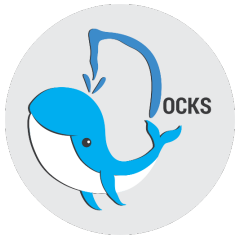

Testing Policy for Docks

Team: TripleParity
Client: Compiax

Team Members
Francois Mentz
Connor Armand du Plooy
Raymond De Vos
Evert Geldenhuys
Anna-Mari Helberg
Paul Wood

Contents

1 Testing Process **2**
1.1 Peer Review . . . 2
1.2 Automated Testing and Continuous Integration . . . 2
1.2.1 Travis CI . . . 2
1.2.2 Docker Cloud . . . 3
1.3 Docks UI . . . 4
1.3.1 tslint . . . 4
1.3.2 karma . . . 5
1.3.3 Protractor . . . 5
1.4 Docks API . . . 5
1.4.1 Eslint . . . 5
1.4.2 Jasmine . . . 6
1.4.3 Jest . . . 6
1.4.4 CodeClimate . . . 7

1 Testing Process

1.1 Peer Review

Before code can be merged into the develop branch a Pull Request has to be created. Three reviews are required from other developers before the pull request can be merged into develop.

During peer reviews the following should be checked:

- Architectural problems - will this cause problems in the future?
- Compliance with requirements and design - is that what we need?
- Coding Standards - proper code formatting and security standards?

Figure 1: At least 3 peer reviews are needed for a feature to be merged

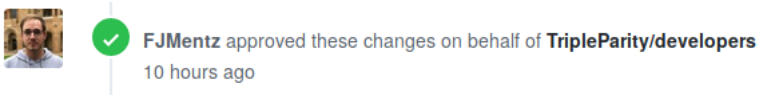

Figure 2: An example of a positive peer review

1.2 Automated Testing and Continuous Integration

When a commit is made to the 'docks-ui' or 'docks-api' repositories on GitHub two processes are started:

- Travis CI Builds the repository (https://travis-ci.org/TripleParity)
- Docker Cloud builds the repository and creates a Docker Image that can be deployed in production (https://hub.docker.com/u/tripleparity/)

1.2.1 Travis CI

The history of test reports for Travis CI can be viewed at https://travis-ci.org/TripleParity/docks-ui/branches. If the build was not successful it will be marked as 'failed'

Travis CI is used for running tests associated with each repository.

For the frontend Angular generates a set of tests for each component to verify that the component was successfully created. These tests run inside a headless (no screen required) Chrome browser.

The backend (docks-api) also has a suite of unit tests that are executed on Travis CI.

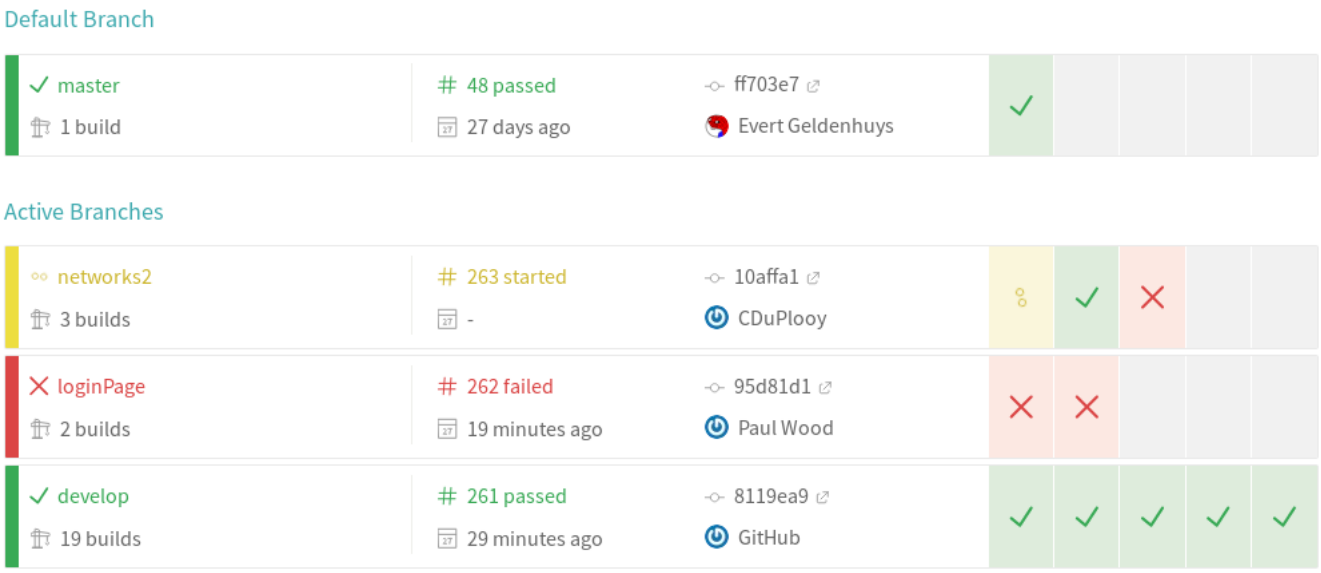

Figure 3: Screenshot of Travis CI branch build history

```
2773 The command "ng build --prod --no-progress" exited with 0.
2774 $ ng test --no-progress
2775 (node:5873) ExperimentalWarning: The fs.promises API is experimental
2776 10 05 2018 20:06:46.172:INFO [karma]: Karma v2.0.2 server started at http://0.0.0.0:9876/
2777 10 05 2018 20:06:46.174:INFO [launcher]: Launching browser ChromeHeadless with unlimited c
2778 10 05 2018 20:06:46.179:INFO [launcher]: Starting browser ChromeHeadless
2779 10 05 2018 20:06:56.045:INFO [HeadlessChrome 0.0.0 (Linux 0.0.0)]: Connected on socket yos
2780 HeadlessChrome 0.0.0 (Linux 0.0.0): Executed 0 of 18 SUCCESS (0 secs / 0 secs)
2781 e 0.0.0 (Linux 0.0.0): Executed 1 of 18 SUCCESS (0 secs / 0.036 secs)
2782 e 0.0.0 (Linux 0.0.0): Executed 2 of 18 SUCCESS (0 secs / 0.046 secs)
2783 e 0.0.0 (Linux 0.0.0): Executed 3 of 18 SUCCESS (0 secs / 0.057 secs)
2784 e 0.0.0 (Linux 0.0.0): Executed 4 of 18 SUCCESS (0 secs / 0.066 secs)
2785 e 0.0.0 (Linux 0.0.0): Executed 5 of 18 SUCCESS (0 secs / 0.076 secs)
2786 e 0.0.0 (Linux 0.0.0): Executed 6 of 18 SUCCESS (0 secs / 0.086 secs)
2787 e 0.0.0 (Linux 0.0.0): Executed 7 of 18 SUCCESS (0 secs / 0.165 secs)
2788 e 0.0.0 (Linux 0.0.0): Executed 8 of 18 SUCCESS (0 secs / 0.225 secs)
2789 e 0.0.0 (Linux 0.0.0): Executed 9 of 18 SUCCESS (0 secs / 0.248 secs)
2790 e 0.0.0 (Linux 0.0.0): Executed 10 of 18 SUCCESS (0 secs / 0.296 secs)
2791 e 0.0.0 (Linux 0.0.0): Executed 11 of 18 SUCCESS (0 secs / 0.307 secs)
2792 e 0.0.0 (Linux 0.0.0): Executed 12 of 18 SUCCESS (0 secs / 0.334 secs)
2793 e 0.0.0 (Linux 0.0.0): Executed 13 of 18 SUCCESS (0 secs / 0.353 secs)
2794 e 0.0.0 (Linux 0.0.0): Executed 14 of 18 SUCCESS (0 secs / 0.372 secs)
2795 e 0.0.0 (Linux 0.0.0): Executed 15 of 18 SUCCESS (0 secs / 0.38 secs)
2796 e 0.0.0 (Linux 0.0.0): Executed 16 of 18 SUCCESS (0 secs / 0.406 secs)
2797 e 0.0.0 (Linux 0.0.0): Executed 17 of 18 SUCCESS (0 secs / 0.47 secs)
2798 e 0.0.0 (Linux 0.0.0): Executed 18 of 18 SUCCESS (0 secs / 0.525 secs)
2799 e 0.0.0 (Linux 0.0.0): Executed 18 of 18 SUCCESS (0.565 secs / 0.525 secs)
2800
2801
2802 The command "ng test --no-progress" exited with 0.
```

Figure 4: 18 UI tests running on Travis

1.2.2 Docker Cloud

Docker images are build on Docker Cloud. These image can then be deployed in a production environment or for development and testing. Images can be viewed at https://hub.docker.com/u/tripleparity/

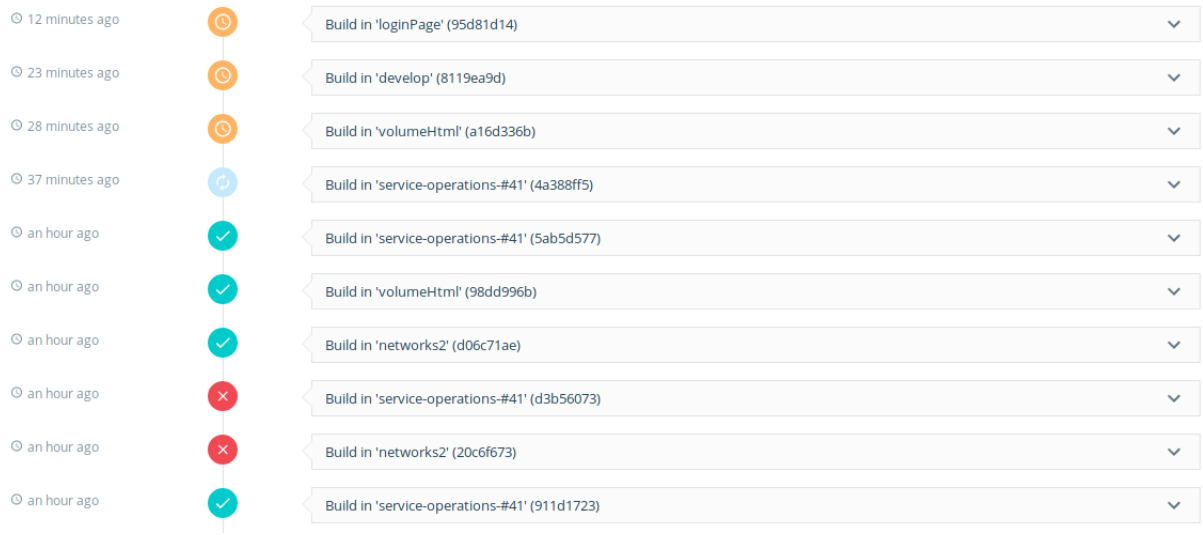

Figure 5: Screenshot of Docker Cloud Build History

1.3 Docks UI

The tests for the frontend could briefly be described as a series of steps. These tests are run on every commit, and on each pull request. Usually each team member will run the tests before comitting to ensure that the build does not break.

1. Run linter to check for type errors/typos in code structure.
2. Create a production build of the web application and ensure there are no errors.
3. Run the internal test cases.

The **test configuration** can be viewed at https://github.com/TripleParity/docks-ui/blob/master/karma.conf.js. The **tests** are inside a variety of files, in particular any file with a keyword "spec" inside https://github.com/TripleParity/docks-ui/tree/master/src/app. The **test history** is on Travis CI (https://travis-ci.org/TripleParity/docks-ui.

1.3.1 tslint

Inside of the UI we additionally use tslint, protractor and karma for testing.
Tslint is a static analysis tool to check typescript code for readability, maintainability and functionality errors. It can be customised with custom linting rules and formatters. This can help catch type errors in the code.

```
fetchVolumes() {
  this.volumeService.getVolumes().subscribe(
    (volumes: Volume[]) => {
      this.volumes = [1, 2, 3];
      this.searchString = [...volumes];
      this.isLoaded = true;
    },
    (err: VolumeError) => {
      this.toastr.error(err.message, 'An error occured');
    }
  );
}
```

Figure 6: Screenshot of tslint linting

In the above example an array of numbers is mistakenly assigned to an array of Volumes. The linter clearly highlights this as a problem.

1.3.2 karma

Karma is a tool which creates easy testing environments for developers; It's main goal is to give developers instant feedback about their test cases.

```
module.exports = function (config) {
  config.set({
    basePath: '',
    frameworks: ['jasmine', '@angular/cli'],
    plugins: [
      require('karma-jasmine'),
      require('karma-chrome-launcher'),
      require('karma-jasmine-html-reporter'),
      require('karma-coverage-istanbul-reporter'),
      require('@angular/cli/plugins/karma')
    ],
    client:{
      clearContext: false // leave Jasmine Spec Runner output visible in browser
    },
    coverageIstanbulReporter: {
      reports: [ 'html', 'lcovonly' ],
      fixWebpackSourcePaths: true
    },
    angularCli: {
      environment: 'dev'
    },
    reporters: ['progress', 'kjhtml'],
    port: 9876,
    colors: true,
    logLevel: config.LOG_INFO,
    autoWatch: true,
    browsers: ['ChromeHeadless'],
    customLaunchers: {
        ChromeHeadlessNoSandbox: {
            base: 'ChromeHeadless',
            flags: ['--headless', '--disable-gpu','--disable-translate', '--disable-extensions', '--remote-debugging-port=9223', '--no-sandbox']
        }
    },
    singleRun: true
  });
};
```

Figure 7: Screenshot of karma config

In particular we run a headless instance of chrome with all extensions disabled to test our code; Istanbul is also used to generate a code coverage report. The test cases are run only once but can be configured such that the tests are run every time a file is changed.
When a commit is made to a branch, these tests are also run.

1.3.3 Protractor

Protractor is an end-to-end test framework. Tests are run in a browser which automatically interacts with the page as a user would. We decided to use protractor because it was specifically created with Angular (our front end framework) in mind.

1.4 Docks API

The tests for the API run in the same way and at the same time, as the front end; Which is to say on every commit and pull request. They can also be run manually. First unit tests are performed with jasmin, and then integration tests are done with jest.

The tests are defined in https://github.com/TripleParity/docks-api/tree/master/spec. The jasmine tests are defined in https://github.com/TripleParity/docks-api/blob/master/__tests__/api/api_spec.js. The test history is on Travis CI (https://travis-ci.org/TripleParity/docks-api.

1.4.1 Eslint

Eslint is essentially the same as tslint but for javascript.

1.4.2 Jasmine

A behaviour driven testing framework. We use it because it does not require any other JavaScript and is easy to write.

```
describe('Authentication', function() {
  it('should not serve Docker API requests without a JWT', function() {
    return frisby.get(host + '/docker/containers/json').expect('status', 401);
  });

  it('should serve JWT for valid ("admin/admin") credentials', function() {
    return frisby
      .post(tokenURL, credentials)
      .expect('status', 200)
      .expect('bodyContains', 'jwt')
      .expect('jsonTypes', 'jwt', Joi.string().required())
      .then(function(res) {
        let jwt = res.json['jwt'];

        return frisby
          .fetch(host + '/', {
            method: 'GET',
            headers: {
              Authorization: 'Bearer ' + jwt,
            },
          })
          .expect('status', 200)
          .expect('bodyContains', 'Welcome to Express');
      });
  });
```

Figure 8: Screenshot of jest test

1.4.3 Jest

Jest is a javascript testing framework. It has some nice features such as instant feedback meaning failed tests run first. This means we can easily tend to issues.

Jest is used in the backend to run tests on the API itself.

1.4.4 CodeClimate

An automated code review tool.

Tests can be viewed at https://codeclimate.com/github/TripleParity/docks-api

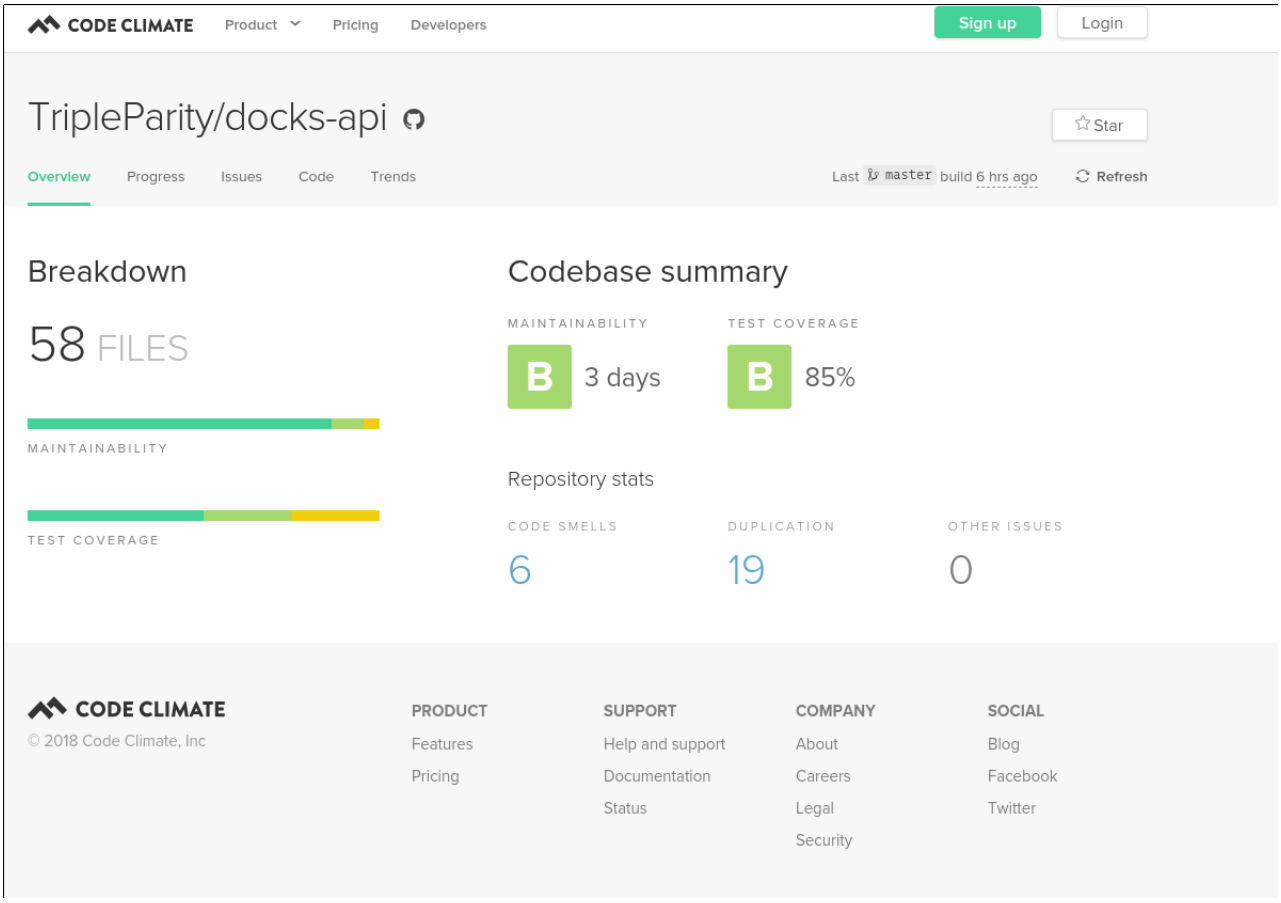

Figure 9: Screenshot of CodeClimate score