Testing course documentation

This is the documentation for Testing course semester project. The testing course is part of the Software Developer course on Copenhagen Business Academy Lyngby.

Authors

- Nicolai Vikkelsø Bonderup
- Dennis Michael Rønnebæk
- Ebbe Vig Nielsen
- Rune Vandall Zimsen
- Emil Rosenius Pedersen

Introduction

Our wish for this project was to cover all of this semesters testing course subjects. Furthermore, we combined our Testing course project with the Database course project Gutenberg. In this document we have written about how this project was setup and all the relevant testing tools and methods we have used.

Below you can find all relevant links to this project:

Links

Assignment links

- Learning goals
- Database course documentation

GitHub Repositories

- Backend Repository
- Frontend Repository

Web application

- Front-end application
- MySQL endpoints
- MongoDB endpoints

Code documentation

Back-end

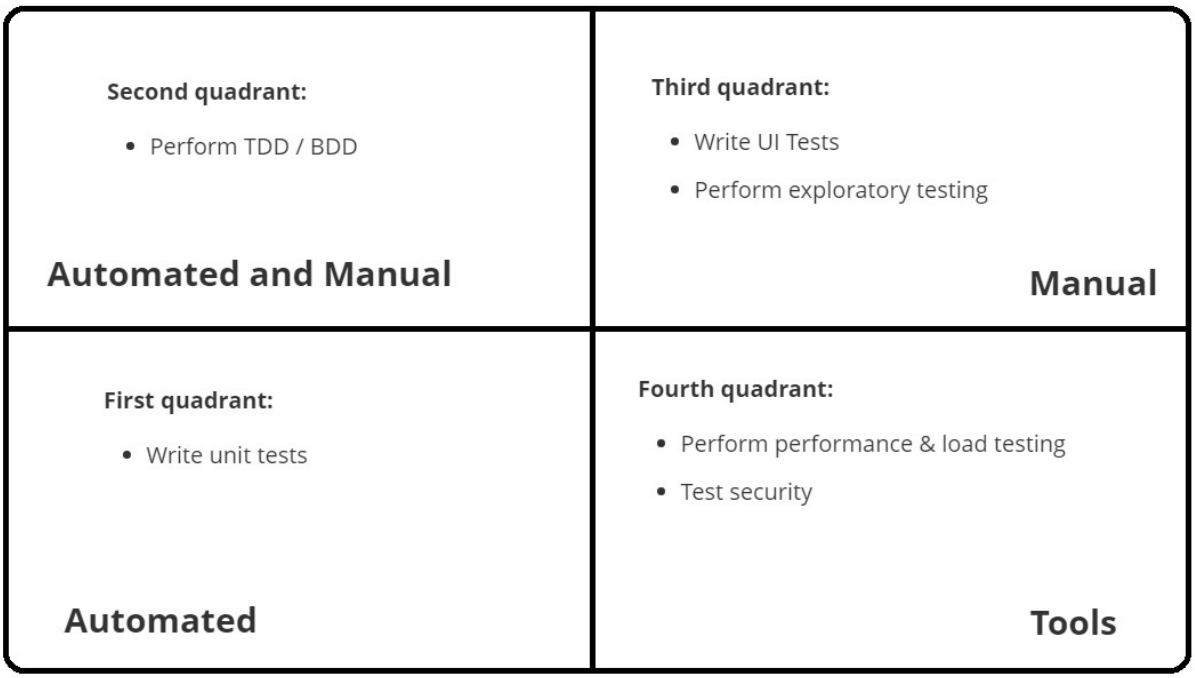
- JavaDoc
- Code Coverage
- Test Analytics

Front-end

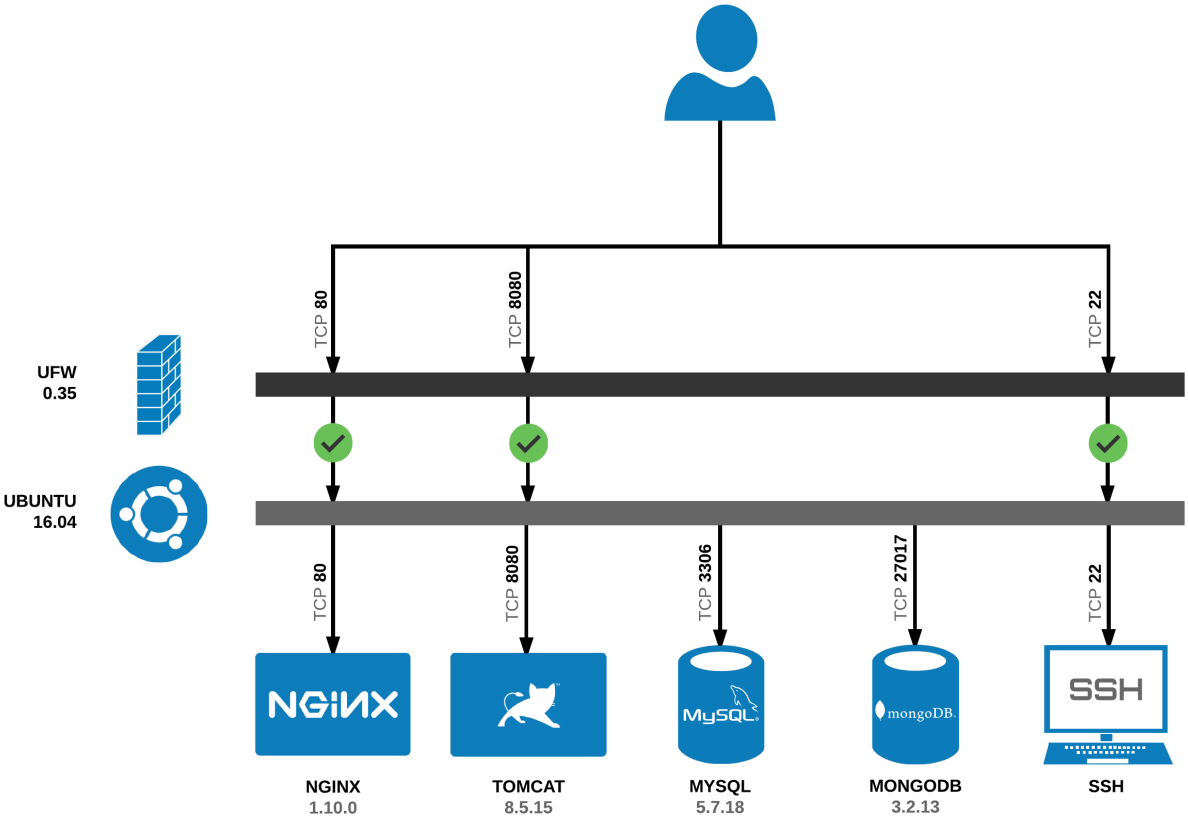
- Code Coverage
- E2E results

Agile Quadrant

To gain a better understanding of the agile quadrant, we have planned our project testing methods and tools from the four quartiles from the test quadrant, as seen below:



Application overview



Our application resides on a Digital Ocean droplet with Ubuntu 16.04, which serves the following applications:

- NGINX 1.10.0
- Tomcat 8.5.15
- MySQL 5.7.18
- MongoDB 3.2.13

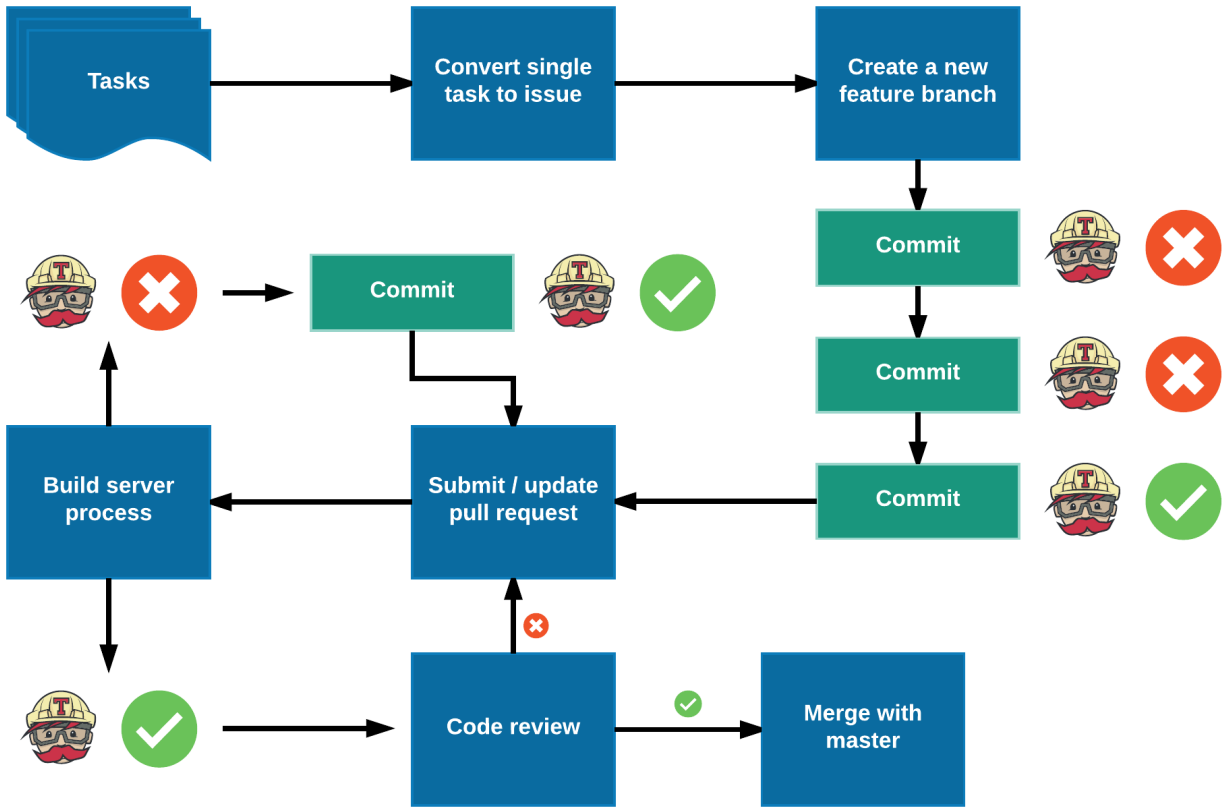
NGINX serves our front-end application, as well as our "CDN". The "CDN" hosts the text files of the books, which can be downloaded through the front-end application.

Tomcat serves our Java Application which exposes a RESTful API. The Java Application uses MySQL and MongoDB to store and query the data.

Operations on the server have been done through an SSH connection through the terminal.

Setup and tools

Continuous Integration & Static testing techniques



Build process

The above picture creates a visual representation of our agreed-upon build process, which is a customized simplification of the git flow strategy. The build process implements a number of system development concepts such as SCRUM-like task management, static testing techniques and continuous integration.

After user stories have been written, they are converted to tasks which are placed in GitHub's planning tool, which is basically a SCRUM Board similar to that of Trello. The tasks are then picked by members of the group and converted to issues, and a feature branch for the selected task is then created. The person working on the task is then committing to that feature branch until the feature has been completed. Each commit triggers a Travis build, which verifies whether the feature is shippable. When the feature is ready for review, the person working on the feature then submits a pull request, which triggers a build for the merge into the master branch. If the build succeeds, static testing techniques are applied where the pull-request is reviewed by another group member. If the other group member approves the feature, it is then merged into the master branch.

CI Setup

For our continuous integration process we have used Travis-CI - an on-the-fly CI tool integrated with our GitHub repositories, so that each commit or pull-request will be tested automatically.

Back-end CI setup

Our CI setup for the backend is configured in the .travis.yml which is located in the root of our back-end repository.

We tell the Travis service that we want to run our build in a environment that supports Java, and that Java should use version 8 of the Java Development Kit.

```
language: java
jdk: oraclejdk8
```

We then tell the Travis service, that we want to change the access permissions for our deploy script so we can execute it on the build server.

```
before_script:
- chmod +x ./deploy.sh
```

Lastly, we tell the Travis service, that we want to execute our custom deploy script.

```
script:
- ./deploy.sh
```

The reason why we have made a custom deploy script, is to differentiate between whether to execute the simple `mvn verify` command, or `mvn clean verify site -X`. The latter also generates documentation. This is determined by checking whether the current build is started from a merge request to the master branch. The git meta data is available through environment variables on the Linux system that Travis provides.

```
if [ $TRAVIS_BRANCH == 'master' -a $TRAVIS_PULL_REQUEST == 'false' ]; then
    mvn clean verify site -X;
else
    mvn verify;
fi
```

We go into more detail with the behavior of the `mvn` commands in the Test Analytics section.

Front-end CI setup

Our CI setup for the front-end is configured in the .travis.yml. which is located in the root of our front-end repository.

We tell the Travis server that we want to deploy the web application in a Docker container in a Linux environment:

```
os:
  - linux
services:
  - docker
```

Since the front-end application is built with Angular-CLI, we need the latest stable distribution of NodeJS:

```
language: node_js

node_js:
  - "6.9"
```

In order to run our E2E tests, we need to be able to run a browser on the Travis machine. We have exploited Docker as a way to be able to install and run Google Chrome on Travis:

```
addons:
  apt:
    sources:
    - google-chrome
    packages:
    - google-chrome-stable
```

In theory, the E2E tests could have been run in a headless browser like PhantomJS, like our unit-tests, however running E2E tests in PhantomJS does not create an exact simulation of a real user.

Instead of executing the tests on a headless browser we use a real browser, whilst instead mocking out the entire display. We add the `CHROME_BIN` environment variable pointing to our Google Chrome install path. We then point our `DISPLAY` environment variable to `:99.0`, and execute an X Virtual Framebuffer server which uses the `DISPLAY` environment to emulate a framebuffer server in virtual memory. This allows Google Chrome to perform all graphical operations on the network layer in the virtual framebuffer without showing any screen output, thus creating a realistic in-memory representation of a UI test:

```
before_install:
  - export CHROME_BIN=/usr/bin/google-chrome
  - export DISPLAY=:99.0
  - sh -e /etc/init.d/xvfb start
```

Before we execute any scripts we tell the Travis service to install `@angular/cli` globally which installs Angular CLI globally on the system and adds the `ng` commands to our path variable.

```
before_script:
  - npm install -g @angular/cli
```

We then build the web application, run the unit tests with a verbose logging level and code coverage, and lastly we execute our E2E tests:

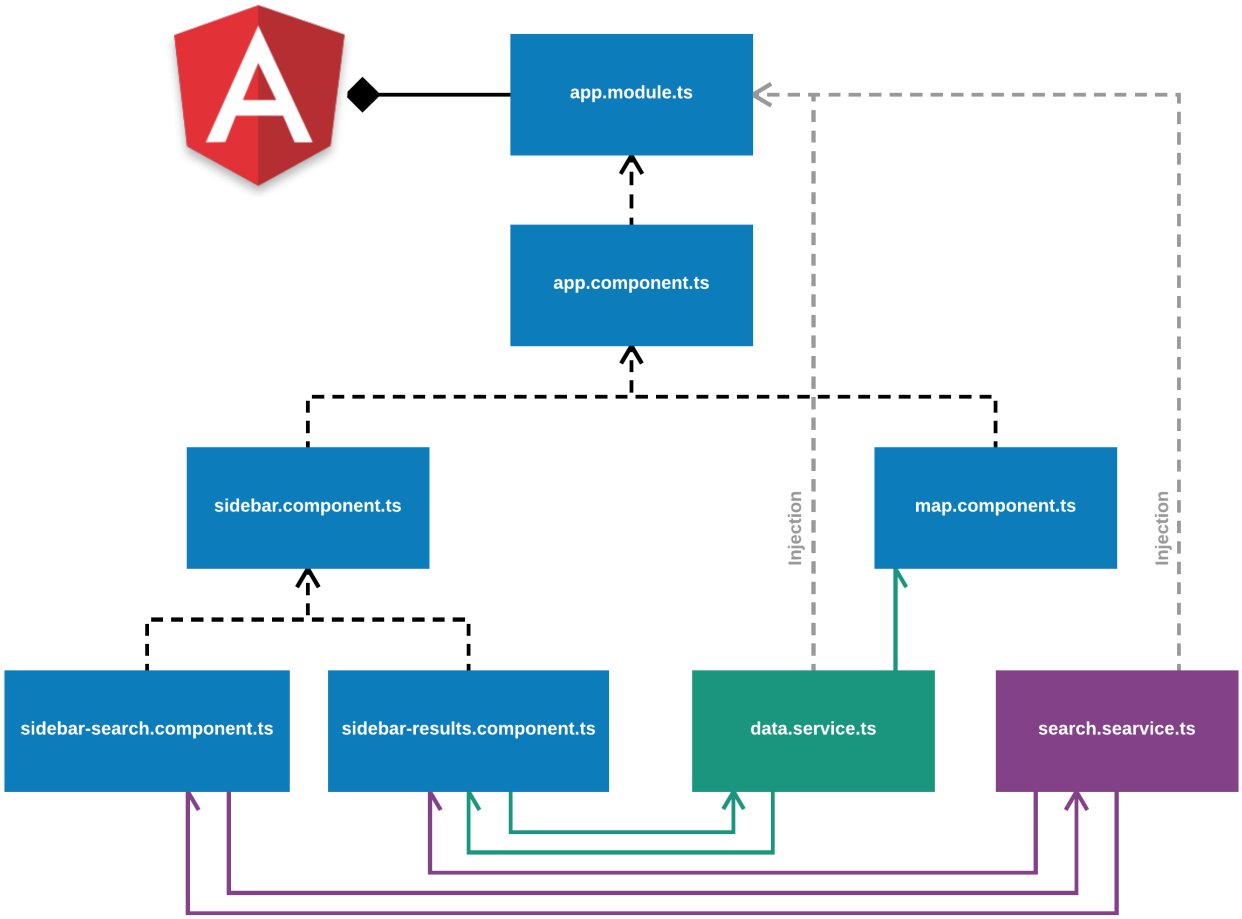
```
script:
  - ng build --prod
  - ng test --code-coverage --watch false --log-level verbose
  - ng e2e
```

Test Analytics & Code Documentation

Testing analytics have been part of our build process for both the front-end and the back-end, ensuring that the appropriate testing analytics have been deployed to a remote location upon every successful build. This has allowed each and every member of the team to continuously keep themselves up to date with the current state of the application in terms of testing analytics and code coverage.

Front-end

Below is a visual representation of the components and services of our front end application.



Unit Testing

Our unit-testing process on the front-end application is handled by the Karma test-runner. Our Karma configuration utilizes a few node packages, which together gives us the desired functionality when it comes to auto-documentation of our testing analytics.

Here we use karma-coverage-istanbul-reporter (Istanbul) for automatically generating a code coverage report for unit tests if the `--code-coverage` flag is set on our CLI command:

```
reporters: config.angularCli && config.angularCli.codeCoverage
             ? ['spec', 'coverage-istanbul']
             : ['spec']
```

Istanbul outputs the code coverage report in the terminal, together with the test results, and generates a static HTML output in a folder, which has been manually deployed to our NGINX web server under /documentation/unit.

```
coverageIstanbulReporter: {
     reports: [ 'text', 'html' ],
     fixWebpackSourcePaths: true
}
```

This could have been done automatically in the Travis CI build process, however due to lack of time, and since the code coverage is also displayed in the terminal, both locally and on Travis-CI, this was not prioritized.

End to end testing

Our e2e testing process is handled by the Protractor framework. Our Protractor configuration also utilizes a few node packages, which gives us the desired functionality when it comes to auto-documentation of our e2e tests.

Here we use protractor-jasmine2-html-reporter for automatically generating a report of our e2e tests. This package will generate a report documenting all of the e2e tests with appropriate screenshots taken at the end of each test. The report is generated as static HTML in a folder, which is manually deployed to our NGINX webserver under /documentation/e2e. (For an explanation of why this is deployed manually, please see the above section about testing analytics for unit testing.).

```
onPrepare() {
   jasmine.getEnv().addReporter(
      new Jasmine2HtmlReporter({
         savePath: 'target'
      })
   );
}
```

Back-end

Our back-end uses the JaCoCo code coverage tool to generate a report of the code coverage along with a calculation of the cyclomatic complexity for each method. Along with the code coverage report we also generate a report on the unit and integration tests with the `mvn site` command which checks for the dependencies like JaCoCo, FailSafe, SureFire etc. and generates a report with test results and the code coverage report.

These reports were automatically pushed to GitHub pages by Travis CI each time a merge was done with the master branch. By differentiating between pushes to the issues branches and the master branch we could avoid the Maven Site on regular pushes to speed up the build time on Travis. In order to do this we added a bash script to our `.travis.yml` that checked whether it was a merge with the master branch or not.

```
if [ $TRAVIS_BRANCH == 'master' -a $TRAVIS_PULL_REQUEST == 'false' ]; then
    mvn clean verify site -X;
else
    mvn verify;
fi
```

In the case it was not a merge with the master branch it would run mvn verify which launches the Surefire/Failsafe unit and along with code coverage, we also generate a report on the Javadoc coverage of the Java project. This way we can document that we have properly covered every method in the project.

Behavior driven development

Behavior driven development is a way of defining the test cases based upon the behavior of the program instead of a technical description of a functionality. We used this approach mainly in the front end testing with Jasmine, as this framework is built upon the `describe`, `it`, `should` syntax and makes for creating the tests in a very humanly descriptive way eg.

```
describe('SearchService', () => {
('should sort data', inject([SearchService], (service: SearchService) => {
    let sorted = service.sortData(["B", "C", "A", "C"]);
    expect(sorted[0]).toBe("A");
    expect(sorted[1]).toBe("B");
    expect(sorted[2]).toBe("C");
    expect(sorted[3]).toBe("C");
  }));
}
```

As you can see in the above example the test is focused on the behavior of the program when describing the test which makes it more readable for non-technical people. The Jasmine tests reside in the folder of each of the components and services.

> app > components / services

Here is a link to one of the component test suites.

Test driven development

We used test driven development (TDD) for the back-end as the IntelliJ IDE, just like Netbeans and Eclipse comes with the ability to easily write tests and then auto generate classes and functions based on these tests, taking the correct set of parameters. This makes it fairly easy to do TDD in the back-end utilizing this functionality. As some parts of the back-end are more focused on configuration rather than raw code, these parts were not done using TDD, such as the Rest API. They could have been done before hand, but it would have demanded the configuration and generation of these methods first, then writing the test, and afterwards going back to write the code. But for the majority of the back-end we have used TDD.

In parallel to the behavior driven development strategy, the tests written for the back-end are much more focused on classical unit tests as it's less descriptive and more code derived.

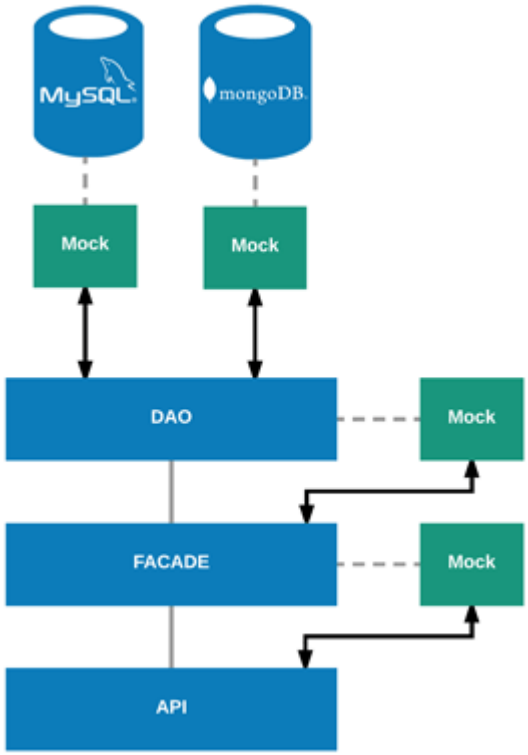
```
@Test
public void testConstructorWithUID() {
  author = new Author(2L, "Ole");
  assertThat(author.getUID(), is(2L));
  assertThat(author.getName(), is("Ole"));
}
```

The above test example is a unit test that tests the constructor of one of the DTO's. As we can see there is no descriptive language but focus on the code.

Unit testing and mocking

Unit testing

Both for the back-end and for the front-end we have unit tests that tests the different methods separately in order to secure the functionality of each specific unit. As we used TDD for the back-end we have around 95% code coverage, which means that 95% of the lines of code in the whole application are tested. As the application primarily focuses on retrieving data from the database, mapping it as a data transfer object and parsing it to the client through our Rest API, there are no particular complex functions in the application. The tests are mainly testing that the objects are created correctly and that the methods return the expected output from a given input. In order to check these methods isolated we have used mocking.

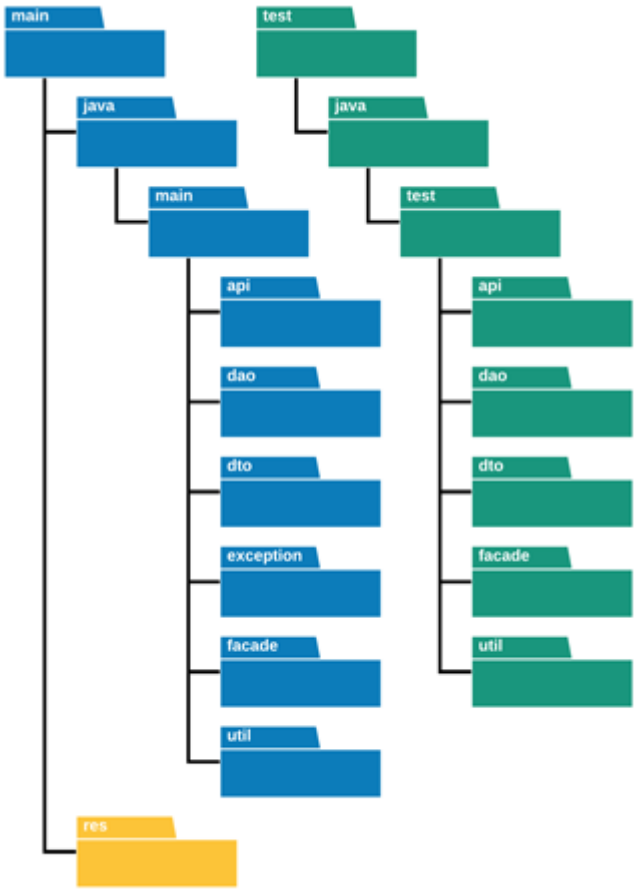


Mocking is used to isolate methods and classes to make sure we test only one component at a time. Mockito was used in the backend. Mockito is effectively made to fake some external dependencies, so that the object being tested has a consistent interaction with its outside dependencies. In the above picture, an overview of the mocking setup is displayed. Unfortunately, we didn't have time to mock out the database, but since the database operations contained no manipulation of the data, we chose to test on the database directly. Ideally this should have been done with a framework like DB Unit.

```
@Test
public void unsuccessfulGetBooksFromLatLongTest() {
  MongoAPI api;
  IBookFacadeMongo facade;

  facade = mock(BookFacadeMongo.class);
  when(facade.getBooksFromLatLong(anyDouble(), anyDouble(), anyInt(), anyInt()))
    .thenThrow(new BookNotFoundException("msg"));

  api = new MongoAPI(facade);
  Response response = api.getBooksFromLatLong(anyDouble(), anyDouble(), anyInt(), anyInt());

  assertThat(response.getStatus(), is(400));
}
```

In this example we use Mockito for the back-end to create a mock of the `BookFacadeMongo`-class and then sets the expectation when calling the `getBooksFromLatLong`-method with any parameter to throw a `BookNotFoundException`. In this way we can unit test the method without triggering the actual call to the database, and assert that the response code is correctly returned when no books are found.

We have structured our back-end tests to follow the same class path as the applications classes, see figure below.



Front-end

For the front-end we have used Jasmine for writing unit tests in Typescript. Jasmine is a JavaScript testing framework which allows us to write behavior driven tests. The Jasmine tests are run through the Karma test-runner, which is configured in the karma.conf.js file. This file sets up Karma with a number of Node packages which gives us our desired testing setup against a PhantomJS browser.

Components

Each component in our web application resides in its own folder containing a number of files:

- component-name.component.html
- component-name.component.scss
- component-name.component.spec.ts
- component-name.component.ts

component-name.component.html

The HTML-file contains the template for the component.

component-name.component.scss

This file contains the styles which are written with SCSS - a CSS preprocessor that provides great improvements to the CSS-language.

component-name.component.spec.ts

This file is where we write tests for the component. In this file, we configure a testbed which initializes the environment for unit testing and provides methods for creating components and services in unit tests. Since unit-testing in Angular 4 is very complicated and requires a complete understanding of the entire lifecycle of an Angular 4 application, we have chosen not to go into detail with the configuration of this testbed in the scope of this report:

```
beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [
        SidebarResultsComponent
      ],
      providers: [
        {provide: DataService, useClass: DataServiceMock},
        {provide: SearchService, useClass: SearchServiceMock},
        SearchService
      ],
      imports: [
        FormsModule,
        HttpModule,
        JsonpModule
      ]
    }).compileComponents();
}));
```

When testing components, we often want to test the functionality that involves transactions between the component itself and a service. For this part, we have chosen to create a stub-class for each service, which contains a mirrored set of methods to the actual service, which instead gives us our desired static output.

In the above code example, the testbed is told to use the stub-class for the service each time the actual service is requested. This means that we can just inject the actual service in our test, and the stub-class will be called instead:

```
it('should give and empty array of books on getBooksFromCity',
    inject([DataService], (dataService: DataService) => {
      component.getBooksFromCity("Copenhagen");
      expect(component.results.type).toBe("getBooksFromCity");
      expect(component.results.cityName).toBe("Copenhagen");
      expect(component.results.data.length).toBe(0);
      expect(component.results.search).toBeFalsy();
    })
  );
```

component-name.component.ts

This file contains the code of the actual component.

Services

Each service in our web application resides in its own folder containing a number of files:

- component-name.component.mock.ts
- component-name.component.spec.ts
- component-name.component.ts

component-name.component.mock.ts

This is the stub-class of the service which is covered in the above section.

component-name.component.spec.ts

This is the file where we test our service. When testing services the principles for the testbed are the same as with components. It is recommended that you read the above section about components before reading this section.

Our services often use external modules and providers like the HTTPModule and HTTP. To test the desired functionality in the service, we mock out the XHRBackend, which is used by the HTTPModule to send requests and receive responses. By mocking out the XHRBackend, we can configure each response that we get back, and modify the object that is returned with the promise:

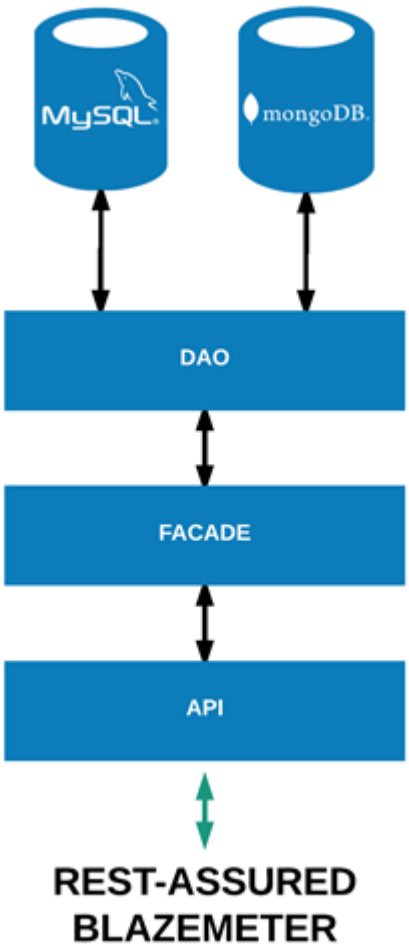
```
beforeEach(inject([XHRBackend], (mockBackend) => {
  mockBackend.connections.subscribe((connection: MockConnection) => {
    let mockResponse = {type: "", data: []};

    if (connection.request.url.indexOf("book") >= 0) {
      mockResponse.type = "book";
    } else if (connection.request.url.indexOf("author") >= 0) {
      mockResponse.type = "author";
    } else if (connection.request.url.indexOf("city") >= 0) {
      mockResponse.type = "city";
    }

    connection.mockRespond(new Response(new ResponseOptions({
      body: JSON.stringify(mockResponse)
    })));
});
```

The method utilizing the HTTPModule can then be tested, as we now have full control over what the XHRBackend returns as its response:

```
it('should return books if type is book.', inject([SearchService], (service: SearchService) => {
  let r;
  service.getSubject().subscribe(results => r = results);
  service.getSearchResults("book", "test").subscribe((books) => {
    expect(books.json().type).toBe("book");
    expect(r.type).toBe("book");
  });
}));
```

Integration testing

Integration testing has been done by testing the API with REST Assured, as seen here. The integration testing is used to test the interaction of methods, classes and modules. Our performance testing has been done by using the tool BlazeMeter, also sending requests to the API to get response times back. It was also a part of the database requirements that we tested the MongoDB vs MySQL DB. The performance test is good for showing where queries need optimization, or a different solution. An overview of a process is seen in figure below.



UI/E2E testing

UI/E2E testing is done by using Protractor, which can be seen here. Protractor is an end-to-end test framework for Angular and AngularJS applications. Protractor runs tests against the application running in a real browser, and therefore it's interacting with the application as a user would.

We run a suite of end-to-end tests for all four of the different query functions available on our API, along with the search methods we use for searching on partial names and titles. This way, we ensure that we get the results we want when we search through the GUI.

Tool-based system-testing

Tool-based system testing was done with JMeter, BlazeMeter and SQLMap. The SQLMap scripts used to test the API can been seen here. Tool-based system testing is a testing discipline where different tools can be used to replicate many users or other things, and it's especially useful for customer requirements test.

Performance Testing

Performance Testing of the API is done with BlazeMeter/JMeter the results can be seen in database documentation. Performance testing is useful to check the all components of the system and can tell if an upgrade of any component is necessary.

Security Testing

We wanted to ensure that the MySQL queries were secure from SQL injections. To ensure this, we performed automated testing through SQLMap, a tool which, when given an endpoint, attempts to send SQL injections to the connected SQL database. We created a `.bat` script for each endpoint, and found that none of our SQL queries were vulnerable to SQL injections.

The scripts we used for SQLMap are located in the repository's resource folder.

Reflections

We chose as a group to include all of the testing course curriculum, which was an ambitious goal, and we have had several long days at the school for 14+ hours to get the project finished. Although it was demanding, we think the result of this project greatly reflects the effort that was put into it.

We wanted to work with continuous integration to get a better experience with this concept. This learning goal has been met as the assignment allowed us to create a continuous integration build, where we have dived deep into some of the areas of Travis-CI.

The agile quadrant's four quantiles have all been covered to a degree that we think is entirely satisfying. We have made sure to follow proper ratios with regards to the Test Pyramid, and have gained insight and experience relating to every quadrant.

We have gained an understanding of the benefits of working with TDD, and have learned that although the initial effort required is high, the benefits of doing so has greatly outweighed the time consumption, especially as every member could work on components of the program whether or not the given component's dependencies were finished. Furthermore, we repeatedly experienced that bugs were found through usage of TDD and static testing.

All in all, we think that the learning goals we have set are met, and we think that the processes positively impacted the entire development process of this application.