QUALITY ASSURANCE

S3

Welles,Nick N.H.M.

2022 - 2023

Table of content

[What is software testing 2](#_Toc122612683)

[Front-end 2](#_Toc122612684)

[Unit/integration tests 2](#_Toc122612685)

[End-to-end tests 3](#_Toc122612686)

[Static code analysis 4](#_Toc122612687)

[Performance tests 5](#_Toc122612688)

[Stress/load tests 6](#_Toc122612689)

[6](#_Toc122612690)

[Backend 6](#_Toc122612691)

[Unit tests 6](#_Toc122612692)

[Integration tests 7](#_Toc122612693)

[Security tests 7](#_Toc122612694)

# What is software testing

Software testing is the most common way of assessing and checking that a software product or application does what it should do. The advantages of testing incorporate forestalling bugs, decreasing development costs and further developing performance.

For a more detailed explanation about what software testing is. View the research report of Software tests.

[View file](Research/Software%20tests.docx)

# Front-end

## Unit/integration tests

In the front-end, I mainly used integration and unit tests. This involves checking whether different components work well with each other. Unit tests check the integrity of the data.

In the code snippet below is a small example of an integration test.

Afbeelding met tekst

Automatisch gegenereerde beschrijving

This small test makes sure that the component is generated. Without the application would crash and not display the wanted data.

In the code snippet below there is a unit test that checks whether the mock-data (in this case the let products) is found in the components html file.

Afbeelding met tekst

Automatisch gegenereerde beschrijving

[View product-detail test file](https://github.com/Instruweb/Front-end/blob/main/src/app/products/product-detail/product-detail.component.spec.ts)

I’ve tested almost all components this way (the ones who pass and receive data from the API)  
But to actually test whether the data from the *real* API is found on the page I wrote some end-to-end tests with Cypress.

## End-to-end tests

The purpose of end-to-end testing is to test whether the data from the database, through the API is also displayed on the front-end page. in other words, from head to tail, testing is done. I made docker compose of the services needed to run end-to-end tests. Keycloak, MariaDB and Quarkus.

[View docker-compose file](https://github.com/Instruweb/Front-end/blob/development/docker-compose.e2e.yml)

Since I am using an external service (keycloak) to allow users to log in, the example below checks whether the specified username and password can also be used to log in via the SSO provider.  
Cypress will visit the home page, then click on the login button in the navigation bar, then it wil fill in the username and password set in the environment variable. Then Cypress will click the login button. If every step of this is successfully achieved then the end-to-end test is passed.

Afbeelding met tekst

Automatisch gegenereerde beschrijving

In the code snippet below the test checks whether the user can navigate from the home page, though a category and a product. In this case the test opens the navigation bar, then starts monitoring the performance, then it will click the ‘Gitaren’ category and at last it will click the ‘More info’ button on that page. If the page loaded after this contains the specified title, the test is successful.

Afbeelding met tekst

Automatisch gegenereerde beschrijving

In the screenshot below, confirmation that all written end-to-end tests passed.  
Afbeelding met tekst, monitor, scherm, schermafbeelding

Automatisch gegenereerde beschrijving

Of course, I tried to automate this process by putting it into the CI/CD pipeline. Unfortunately, this failed due to a Proxy error message.

To avoid a CORS error message with requests to the API, I used a Proxy in my Angular front-end application. In the time I have been working on this, I have not come across a solution that fixes this particular problem.



## Static code analysis

To check if the code is bug-free, no security flaws, etc, I ran a static code analysis. This analysis will analyse al code smells, parts that are covered by unit test and how much of the complete code that is. To do this I made use of SonarCloud (the cloud variant of SonarQube) This process is completely automated in the CI/CD pipeline. In the screenshot below are the results of a run.

Afbeelding met tekst

Automatisch gegenereerde beschrijving

Note that the code coverage in the front-end application is on 0.0%. For some reason SonarCloud did not recognize that the unit tests actually covered parts of the code. When I run the coverage locally via npm, we get this result:

Afbeelding met tekst

Automatisch gegenereerde beschrijving

[View CI file](https://github.com/Instruweb/Front-end/blob/development/.github/workflows/ci.yml)

## Performance tests

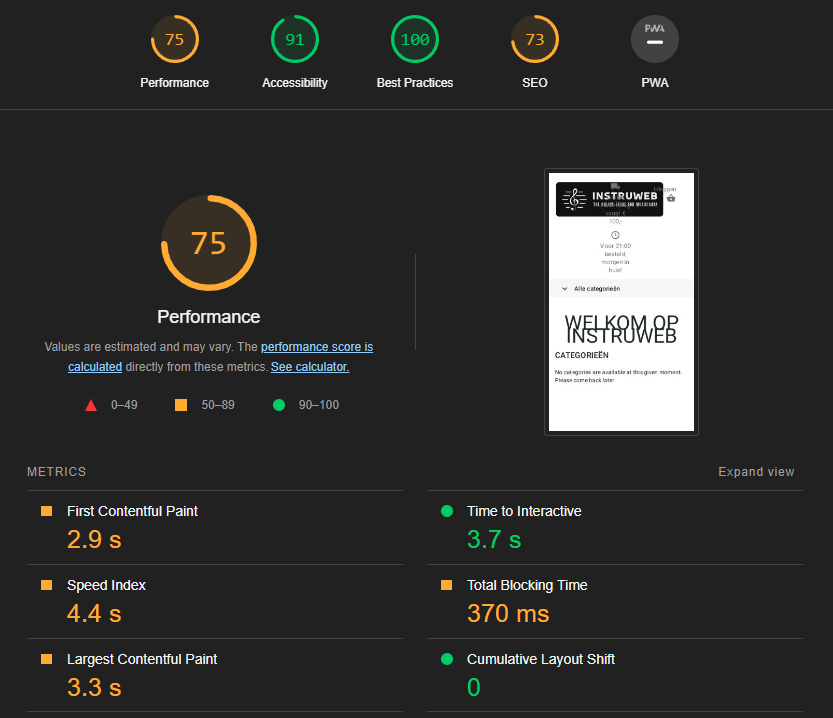
To test whether the website actually loads quickly, I applied performance testing. By installing an npm package from Google called Lighthouse, you can easily test the performance of your application.

The code below makes sure there is a headless Chrome browser running. Lighthouse will navigate to /products/detail/1, then it will wait until the page is fully loaded and dumps everything in a lhreport.html file.

Afbeelding met tekst, scherm, schermafbeelding, sluiten

Automatisch gegenereerde beschrijving

I automated this process in the CI/CD pipeline. Here I used a GitHub action. This shows a URL at the end where you can find your report.



[View CD file](https://github.com/Instruweb/Front-end/blob/development/.github/workflows/cd.yml)

## Stress/load tests

To check whether the application will still stand under heavy load, stress tests will be applied. I used the npm package Artillery. This is a package that turns a .yml file into a script.

This script will visit the localhost instance with url /products/detail/1. It has 4 phases. A warmup phase; this one is under normal load. A ramp up phase; this one is more or less a hefty load. And at last the Sustained load; this one will run a sustained load for 10 minutes. If there is one error the test fails.

# Afbeelding met tekst Automatisch gegenereerde beschrijving

The end result:

# Backend

## Unit tests

In the backend, it is definitely important to see what the code will do with mock-data. Hence, a fair amount of unit tests have been created for this.

In Quarkus I’ve made use of the @QuarkusTest dependency to define that specific function as a test.

In the example below, I checked whether the mock product I want to create is actually returned by the response of the function being called.  
Afbeelding met tekst

Automatisch gegenereerde beschrijving

## Integration tests

Since keycloak is embedded in the Quarkus backend service we need to make sure it is well integrated with the written code.

In the example below the function updates the user given by the @TestSecurity. The function called in this integration test returns a message, and that is the message we expect to get if we update the user. If the String username was a different user, we would not be allowed to update that user since the @TestSecurity has ‘nickwelles’ as given user. This also has something to do with security tests.

Afbeelding met tekst

Automatisch gegenereerde beschrijving

## Security tests

To make sure your application is secure you need to perform security tests. In Quarkus, together with keycloak we can check whether a user is allowed to access an endpoint or not.

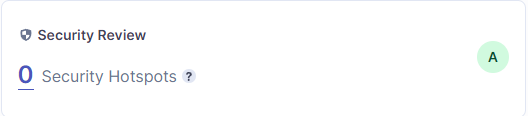
In this example the first function (testUpdateUser) updates the ‘nickwelles’ user that is given in the path in the put() function. In the function below that (testUpdateAnotherUser) we want to update ‘anotherusername’ given in the put() function, but the user the security dependency gives us is ‘nickwelles’, so that’s why the expected status code is 401 unauthorized.

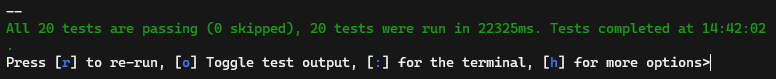
Afbeelding met tekst

Automatisch gegenereerde beschrijving

[View test files](https://github.com/Instruweb/Backend/tree/main/src/test/java/com/instruweb)

SonarCloud also provides lots of functionality to check if there is a security flaw. For example, someone left a token or a password in the code and committed this to the repository. SonarCloud will recognize this and your workflow job will fail.





Both these processes are automated using the CI/CD pipeline from GitHub.

[View workflow files](https://github.com/Instruweb/Backend/tree/main/.github/workflows)