Software Quality

Carlo van Kessel
S-DB-IP3 and S-DB-GP3
S3-DB03
Hans van Heumen, Marc van Grootel
09-01-2023
V3

Version control

Version	Changes
1	Initial version.
2	Added backend tests.
3	Added load testing.

Contents

Frontend

Unit testing

What did I use?

For unit testing my Vue code I used Vue Test Utils (VTU). With VTU you can easily make unit tests because it isolates the component.

How did I use it?

I created a spec.js file in the automatically created tests/unit folder. In this file I can then easily import a component and use the mount from VTU. Now I can make a test:

```
test('Should render input for a new workout', () => {
    //Arrange
    const wrapper = mount(startNewWorkout)

    //Act
    const newWorkoutInput = wrapper.get('.form-control')

    //Assert
    expect(newWorkoutInput.element).toBeTruthy();
})
```

Here I test if the input on the startNewWorkout component is rendered. With the get I get the input element with its class name called form-control. Then I expect the element to be truth which means it has rendered.

Results

When running all the test with the command: 'npm run unit:test'. It runs all the tests and gives me the results:

```
Test Suites: 2 passed, 2 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        2.794 s
Ran all test suites.
```

I added that when it runs it gives me a coverage:

```
---------------------------|---------|----------|---------|---------|-------------------
File                       | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
---------------------------|---------|----------|---------|---------|-------------------
All files                  |    52.8 |     12.5 |    6.52 |    52.8 |
 src                       |   30.33 |        0 |    5.26 |   30.33 |
  App.vue                  |       0 |      100 |       0 |       0 | 4-55
  ErrorPage.vue            |     100 |      100 |     100 |     100 |
  HomePage.vue             |   88.88 |      100 |       0 |   88.88 | 59
  StatsPage.vue            |   88.88 |      100 |       0 |   88.88 | 21
  main.js                  |       0 |        0 |       0 |       0 | 14-81
  registerServiceWorker.js |       0 |        0 |       0 |       0 | 5-29
 src/components            |   68.29 |    18.18 |     7.4 |   68.29 |
  AppFooter.vue            |     100 |      100 |     100 |     100 |
  AppHeader.vue            |       0 |        0 |       0 |       0 | 4-63
  StartNewWorkout.vue      |   85.71 |    66.66 |   33.33 |   85.71 | 85-88
  Workout.vue              |   77.19 |        0 |       0 |   77.19 | 316-365
  WorkoutsTable.vue        |   78.26 |        0 |       0 |   78.26 | 108-120
 src/router                |     100 |      100 |     100 |     100 |
  index.js                 |     100 |      100 |     100 |     100 |
---------------------------|---------|----------|---------|---------|-------------------
```

CI/CD

In my CI I added the unit tests:

```
      - name: run unit and integration tests
        run: npm run unit:test
```

When the tests fail the CI fails and the developer needs to make changes.
CI file

Integration testing

In the frontend I don't have integration tests. This is because you can't access my application without login in with Keycloak. Since this will then be called an end-to-end test, I don't have an integration test.

End-to-end testing

What did I use?

For end-to-end testing I used Cypress. This allows you to test you application just as a user would. You can test the full application.

How did I use it?

I created 2 *.cy.js files, these files are the test files. At the start I use a before each:

```
beforeEach(() => {
    cy.visit('http://localhost:8081/workout/5498f676-d9ec-4dba-9e16-
164dc782bfa5')

    cy.get('#username')
        .type(Cypress.env('CYPRESS_USERNAME'))

    cy.get('#password')
        .type(Cypress.env('CYPRESS_PASSWORD'))

    cy.get('#kc-form-login').submit()
})
```

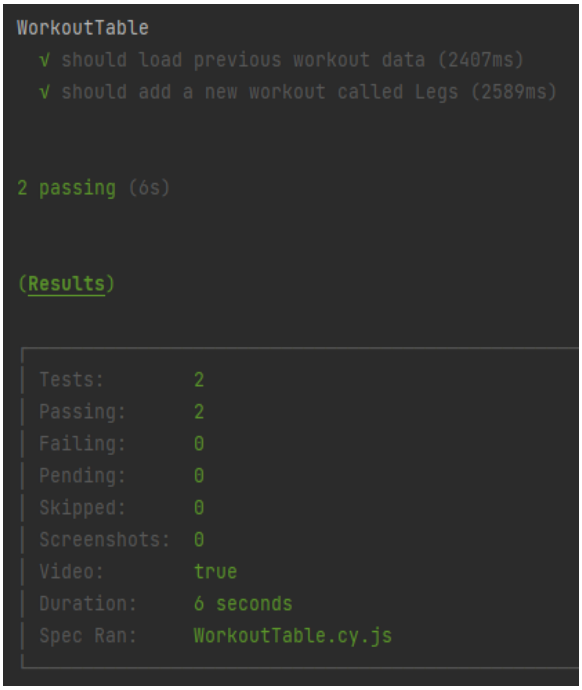
Here I login into the application. Without logging in you can't access the application.
Then I test the application:

```
it('should update set', function () {
    cy.get('.setsInput')
        .type('2')

    cy.get('.repsInput')
        .type('8')

    cy.get('.weightInput')
        .type('120')

    cy.get('.restInput')
        .type('5')

    cy.get('.updateSet').submit()

    cy.get('.setsInput').should('have.value', '2')
    cy.get('.repsInput').should('have.value', '8')
    cy.get('.weightInput').should('have.value', '120')
    cy.get('.restInput').should('have.value', '5')
});
```

This tests if I can update a set.

Results

When running all the tests with the command: ‘npm run cypress:test’. It runs all the tests in the folder and here are the results:



CI/CD

I tried to automate this process in my CD. Unfortunately, I did not get docker to work with my database, Keycloak, and backend. This results in me getting 401 and 403 messages. What I did do is commented it out in my CD. So, when I eventually get docker to work I can just uncommand the end-to-end part in my CD, and it should automatically work.
CD file

Performance testing

What did I use?

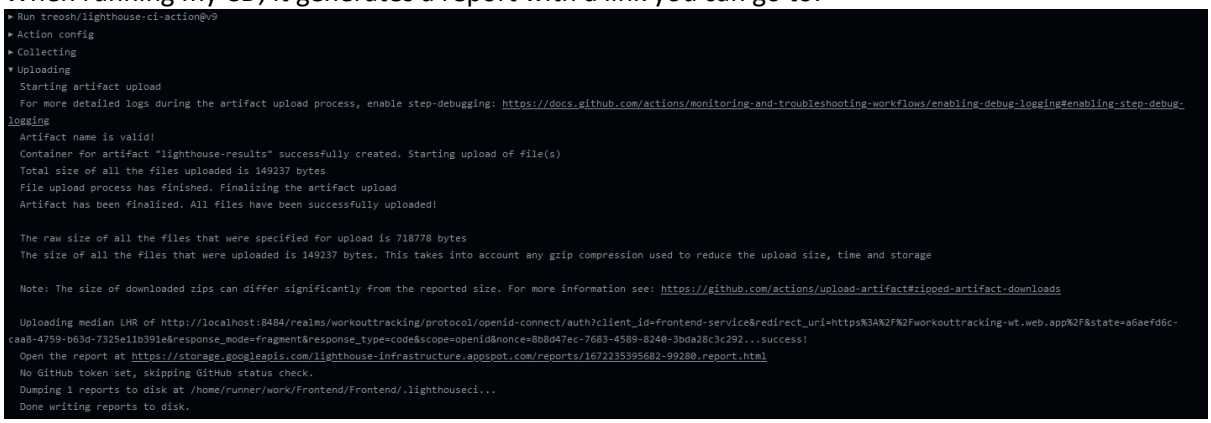
For my performance test I used Google Lighthouse.

How did I use it?

I used it in my CD. I did not write single tests because my application you need to login to even start. So that would then be an end-to-end test.

Results

When running my CD, it generates a report with a link you can go to:



In this report you can see that PWA is not installable. This is because it links to Keycloak: Report. But when I did a run earlier, when Keycloak was down I get this: Report. Here you can see that PWA is installable.

Load testing

What did I use?

For load testing I used Artillery. Artillery is the most advanced load testing in the world and is used by big companies like Gymshark, Brave, and Amity.

How did I use it?

With Artillery it is easy to make tests. I installed Artillery and created an yml file:

```yaml
config:
  target: "http://localhost:8081"
  phases:
    - duration: 60
      arrivalRate: 5
      name: Warm up
    - duration: 120
      arrivalRate: 5
      rampTo: 50
      name: Ramp up load
    - duration: 600
      arrivalRate: 50
      name: Sustained load
  ensure:
    maxErrorRate: 1
    max: 500

scenarios:
  - name: "Go to home"
    flow:
      - get:
          url: "/"
```

This script will go to the home page. It has 3 phases: warm up, ramp up, and sustained. The warmup phase is a phase where the site is under normal load. The ramp up phase is a more loaded phase where the site is more tested. The sustained phase is a phase which runs longer for more reliability. If there is one error the test fails.

Results

```
All VUs finished. Total time: 13 minutes, 2 seconds

------------------------------
Summary report @ 12:44:20(+0100)
------------------------------

http.codes.200: ..... 33562
http.request_rate: ..... 43/sec
http.requests: ..... 33562
http.response_time:
  min: ..... 0
  max: ..... 43
  median: ..... 1
  p95: ..... 2
  p99: ..... 3
http.responses: ..... 33562
vusers.completed: ..... 33562
vusers.created: ..... 33562
vusers.created_by_name.Go to home: ..... 33562
vusers.failed: ..... 0
vusers.session_length:
  min: ..... 1.7
  max: ..... 54.7
  median: ..... 3.6
  p95: ..... 6.5
  p99: ..... 25.8
```

CI/CD

I have not added this test to my CI, this is because then my CI would take about 20 minutes. Now I can run this local in the background.

Static code analysis

What did I use?

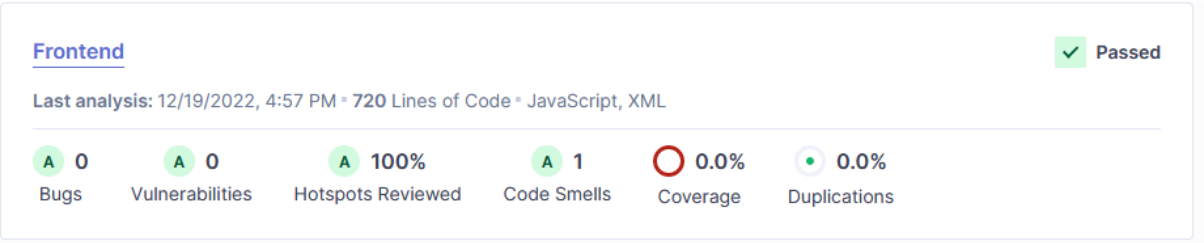
For my static code analysis, I used SonarCloud. SonarCloud is a code analysis platform that helps with detecting code quality and security issues within code.

How did I use it?

To install sonarcloud you need to add a sonar-project.properties file. Here you fill in some credentials:

```
sonar.projectKey=WorkoutTracking_Frontend
sonar.organization=workouttracking
sonar.projectName=Frontend
sonar.host.url=https://sonarcloud.io
```

Results



Here you can see all the results of the scan.
The coverage does not work. This is because you need to make report files of the tests. It tried to get it to work but SonarCloud would not recognize my coverage files. So that is way is show coverage in my unit test above.

CI/CD

I automated this scan in my CI:

```
INFO: ANALYSIS SUCCESSFUL, you can find the results at: https://sonarcloud.io/dashboard?id=WorkoutTracking_Frontend&pullRequest=53
INFO: Note that you will be able to access the updated dashboard once the server has processed the submitted analysis report
INFO: More about the report processing at https://sonarcloud.io/api/ce/task?id=AYVY_XCqn6WCtpJEqXiS
```

Then in my SonarCloud dashboard I can see the last scan.

Backend

Unit testing

What did I use?

For unit testing I used the test implementation of Junit. With @QuarkusTest above the class its recognized as a test class.

How did I use it?

```
@Test
@TestSecurity(user = "carlovankessel@yahoo.nl", roles = "user")
public void When_Get_Workouts_By_User_Is_More_Than_Zero() {
    // Arrange
    String email = "carlovankessel@yahoo.nl";
    int lowestSize = 0;

    // Act
    List<Workout> workouts = workoutService.allWorkoutsByUser(email);

    // Assert
    Assertions.assertNotEquals(lowestSize, workouts.size());
}
```

Here as the name suggests I check if the workouts I get back are not equal to 0. In this test I test the unit allWorkoutsByUser in the workoutService.

Results

```
All 8 tests are passing (0 skipped), 8 tests were run in 39385ms. Tests completed at 13:53:28.
```

When running the tests local it gives all green.

CI/CD

I automated this process in my CI. When building the application with ./gradlew build the application runs these tests automatically.

integration testing

What did I use?

For integration tests I again used Junit with @QuarkusTest.

How did I use it?

```
@Test
@Transactional
@TestSecurity(user = "carlovankessel@yahoo.nl", roles = "user")
public void When_Post_Workout_Succeeds() {
    given()
            .header(HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION_JSON)
            .header(HttpHeaders.ACCEPT, MediaType.APPLICATION_JSON)
            .when().post("/" + "Legs" + "/" + "carlovankessel@yahoo.nl")
            .then()
            .statusCode(201);

    given()
            .when().get("/user/carlovankessel@yahoo.nl")
            .then()
            .statusCode(200)
            .body("$.size()", is(3));
}
```

The first givin adds a workout to the user. It checks this automatically because of the status code 201 which means it is created. Then with the second given it check if it is actually created by checking if the user has 3 workouts.

Results

The results are the same with the unit tests.

CI/CD

The CI is the exact same as the unit tests.

Security testing

To make sure the user who tries to access the backend endpoint needs to be verified by Keycloak. This is done with the @TestSecurity I give the user email. Then it will check automatically if it is valid.

```
@Test
@TestSecurity(user = "admin@yahoo.nl", roles = "user")
public void
When_Get_Exercises_By_Workout_And_User_Fails_Because_TokenEmail_And_Email_A
rent_From_The_Same_User() {
    // Arrange
    String email = "carlovankessel@yahoo.nl";
    UUID workoutId = UUID.fromString("32b5646c-cecb-4518-ae17-
bcd296990db7");

    // Assert
    Assertions.assertThrows(IllegalArgumentException.class, () -> {
        exerciseService.findExercisesByWorkoutId(workoutId, email);
    });
}
```

As you can see here when trying a different email than the one in the token it fails.

Static code analysis

What did I use?

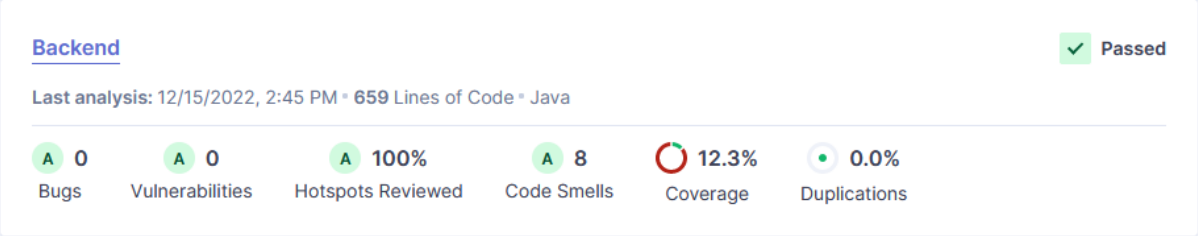
For the backend I used the same as the frontend SonarCloud.

How did I use it?

In Quarkus to add SonarCloud you need to add the credentials in the build.gradle file:

```
sonarqube {
    properties {
        property "sonar.projectKey", "WorkoutTracking_Backend"
        property "sonar.projectName", "Backend"
        property "sonar.organization", "workouttracking"
        property "sonar.host.url", "https://sonarcloud.io"
    }
}
```

Results



Here you can see the results of the scan. For my backend the code coverage does work. I used the JacocoTestReport to make a report that SonarCloud can read to get the coverage.

CI/CD

I automated this scan in my CI, which gives me the exact same thing as the frontend.

Test files

Frontend

1. Unit test files
2. End-to-end files
3. Load test file

Backend

1. Unit test files
2. Integration test files
3. Security test files

Sources

https://www.youtube.com/watch?v=QIDhzBg5eWY