Karina Raiküla

Implementation of Automated End-To-End Testing in Web Applications

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communication Technology

Bachelor’s Thesis

13 April 2023

Abstract

Author:	Karina Raiküla
Title:	Implementation of Automated End-To-End Testing in Web Applications
Number of Pages:	35 pages
Date:	13 April 2023
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communication Technology
Professional Major:	Media Technology
Supervisor:	Matti Peltoniemi, Senior Lecturer

Web automation is a growing practice and having a high-quality product is crucial, as the market is greatly saturated with web applications. This thesis follows the implementation process of an automated Cypress testing environment for a Finnish sports competition organisation Reppi. End-to-end testing is used as it tests the application from the viewpoint of the end user, thus making sure the application delivers exactly what real world users need.

Three test suites for the most popular functionalities were outlined and the tests for these were created. Tests were optimized by including custom commands, sessions, and tools. Finished tests will be included into daily programming practices, to ensure a high standard for the final product, customer satisfaction and user retention.

The work argues for the benefits of implementing an automated testing framework into development practises. Though this project is focused on End-to-end testing with Cypress, other frameworks and methods are outlined, as the technology of choice is unique to each development team and should be implemented regardless of chosen framework.

Keywords: Test Automation, Web Testing, End-To-End, Cypress, Web Application, Angular, AWS

Tiivistelmä

Tekijä:	Karina Raiküla
Otsikko:	Automatisoidun päästä päähän -testauksen käyttöönotto verkkosovelluksissa
Sivumäärä:	35 sivua
Aika:	11.4.2023
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Mediatekniikka
Ohjaaja:	Lehtori Matti Peltoniemi

Insinöörityön tarkoitus oli luoda ja ottaa käyttöön automatisoitu Cypress-testausympäristö suomalaiselle urheilukilpailujärjestö Repille sekä tutkia automaatiotestauksen menetelmiä ja käytäntöjä. Reppi halusi tuotteelleen automaatiotestausta, koska se mahdollistaa laadukkaan tuotteen kehittämistä ja sen systemaattista ylläpitoa. Korkealuokkainen verkkosovellus edesauttaa yrityksiä erottumaan massasta ja varmistamaan käyttäjien uskollisuuden.

Insinöörityössä luotiin Cypress-testiympäristö toimeksiantajalle, Repille. Tärkeimmiksi toiminnoiksi valittiin kolme kokonaisuutta, ja niille luotiin luotettavat testit. Työssä todettiin automatisoidun Cypress-testausympäristön helppo ja nopea käyttöönotto, sekä monipuoliset komennot, istunnot ja työkalut. Valmiit testit sisällytetään päivittäisiin ohjelmointikäytäntöihin, jotta varmistetaan korkealaatuinen tuote, asiakastyytyväisyys ja käyttäjien pysyvyys.

Työssä käytetty päästä päähän -testaus tarkastelee sovellusta loppukäyttäjän näkökulmasta ja varmistaa, että sovellus tarjoaa juuri sitä, mitä tosielämän käyttäjät tarvitsevat. Vaikka insinöörityössä keskityttiin Cypress-testaukseen, muitakin ohjelmia on olemassa, kuten Selenium ja Appium. Päästä päähän -testauksen lisäksi on mahdollista soveltaa valkoisen tai mustan laatikon testausta, toiminnallista testausta ja regressiotestausta. Valitusta menettelytavasta tai ohjelmasta riippumatta automaatiotestaus on hyvä ottaa käyttöön jokaisessa kehitysympäristössä.

Avainsanat: Automaatiotestaus, End-To-End, Cypress, verkkosovellus, Angular, AWS

Table of Contents

1 Introduction

Testing a web application is a critical part of the development lifecycle. It is the process of determining if the product meets its working requirements. The product must satisfy consumer needs as well as technical demands to ensure an optimal working product. A well working, high-quality product can be a deciding factor when securing users' loyalty. The testing process of a web application includes the creation of test cases, overseeing and validating their successes and failures, and maintaining the validity. Automating the testing process saves time and resources in the long run, but it does require a significant initial investment of both time and resources.

The topic of this thesis was given by Reppi. Reppi is a sports competition and results platform, with a focus on CrossFit. It was founded in 2018 when competition organisers noticed a gap in the results services on the market. They wanted an easy-to-use platform and decided to create one themselves. The platform was originally meant for CrossFit competitions and at its core, it is designed as such. Nevertheless, it is also used for other sports such as weightlifting, functional fitness, and strongman and strongwoman competitions.

The goal of this thesis is to set up an automation environment for Reppi and automate repetitive tasks in advance of their front-page reform. The chosen framework for testing is Cypress, and three test suites for an initial testing plan are outlined as: single-athlete competition functions, pair competition functions and team competition functions. These functions are repetitive and include signing up for competitions and modifying personal details.

The goal for this project is to create an efficient End-to-end testing environment with Cypress. Three testing suites that can be used during development and for continuous integration will be developed. The finished work is intended to be integrated into daily practices to ensure the application works as expected.

2 Importance of Testing in Web Applications

Every application is tested before it is published, sent to the customer, or redeployed. A well-working application has a higher rate of user retention than one filled with flaws. In smaller web applications testing can easily be done manually by going through the application and ensuring that all features work as intended, and no errors arise. However, as applications scale up in size, manual testing becomes time-consuming, costly, and prone to errors.

Manually testing a web application during development, to see if it behaves as expected, seems simple, since you only interact with the application as any end-user would. However, this approach is unreliable and tedious, especially if done multiple times a day. Small changes to functions might seem insignificant, and not worth manually testing the whole application for, which can later cause issues ranging from simple inconveniences and user dissatisfaction to economic problems and loss of business [1 p.195]. Implementing testing from the early stages of development reduces issues, bugs, and the possibility of complete failure upon implementation [2 p.126].

Validation refers to the evaluation of the application at the end of its development to ensure compliance with the intended usage. Verification is the process of determining whether the product fulfils the technical requirements that have been established. Verification is usually more technical and focuses on the technology behind implementation, such as software requirements and specifications, whereas validation assesses if the product works as intended for the end user and depends heavily on domain knowledge. [3 pp.8-9.] As the verification and validation of a conventional testing process are usually performed by different persons, efficient automation provides a better relation between verification and validation [2 p.126].

Many teams automate their web application testing using several different methods. The main benefits of automated testing come from its fast and unattended testing, finding issues in the early stages of development, streamlining validation and verification, and reusing test code for regression testing at all stages of the application’s lifetime [1 p.195].

2.1 Disadvantages of Automated Testing

Taipale et. al [4], in their qualitative study of the benefits and drawbacks of test automation, found that the main disadvantages of automated testing are the associated costs. Costs include implementation costs such as direct investment, time, and human resources, as well as maintenance, and training costs [4 p.123]. Leotta et. al [1] also argue the costs associated with the maintenance of fragile tests, are the biggest setback of automated testing. Automating the testing process does not exclude human involvement and can require retraining of testing staff. The reusability of automated tests is therefore crucial for ensuring the profitability of testing. [4 p.123].

Tests are fragile when they are easily broken with a slight modification to the application. Broken tests might be unable to locate links, input fields or correct paths and therefore yield a failed test. Repairing broken tests is a time-consuming task that must be performed by a Testing Engineer. Repair activities of a web application can be categorised into two types: logical and structural, depending on the kind of maintenance that is needed [5 p.274].

Logical repairs are needed after a functionality of the web application has been modified. Tests may not be able to find the correct path to a web page anymore, or the target functionality has changed, therefore generating a failed test. [5 pp.274-275.] A simple example of this would be adding a new, compulsory, field to a form, which has not been included in the previous tests. As the test was not written to include this new field, form submission will be unsuccessful, and the test will fail. The logic of the test must now be updated to include the new functionality for it to become accurate again.

Structural repairs only involve changes in the layout or structure of the web page and are significantly easier to repair than logical changes [5 p.275]. These repairs might simply include updating the test locator ID from ‘name’ to ‘username’. However, depending on the size of the application and the framework that is being used for testing, it is better to use custom locator attributes, instead of relying on IDs or class names, to minimise fragility. Targeting dynamic elements is considered to be bad practice.

Testing must be supervised in case of failed tests, maintained as old features are refactored, and developed when new functions are added. [4 p.122.] Therefore, despite being automated, testing still needs human involvement. Automated tests are excellent for repetitive tasks but not as useful for non-standardised tasks. Manual testing is a better option for functions that are not repetitive. The choice of what to test manually and what to automate should be decided early in the project, to maximise the efficiency of testing practices. [4 p.114.]

2.2 Testing Methods

There are many available approaches to testing depending on what the purpose of testing is. Application testing can roughly be divided into White box and Black box testing. White box testing can be described as Functional testing and has four levels: Unit, Integration, System and Acceptance testing, as shown in figure 1 [6].

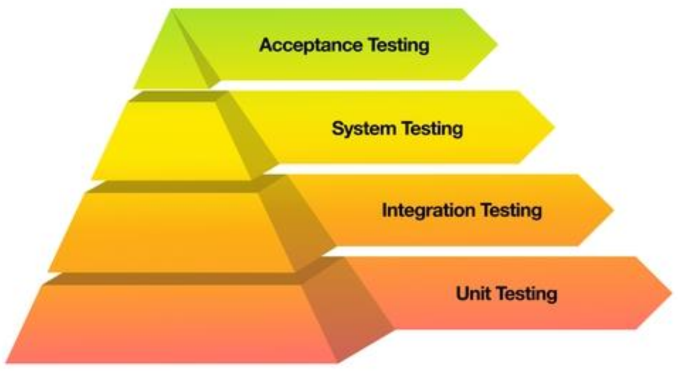


Figure 1. Functional testing pyramid.

At each level, a different approach to testing is implemented. Black box testing looks at the application, without knowing its inner workings. End-to-end testing, which looks at how the application performs from the users' perspective, whilst validating backend functionalities, is also a type of Black box testing. Regression testing tests how new features work together with existing ones and is a good practice to implement during development.

2.2.1 Functional Testing

Unit testing happens at the individual unit levels and tests methods and classes. As this is done at the code level, it is usually done by developers. Unit tests are done on low-level components to verify their desired functionality. Testing multiple units as a group, or module testing, can be automated for maximum efficiency. [7 p.146.]

Integration testing is performed after integrating two or more modules that interact with each other. Again, this is done on the code level, and not from the perspective of the end user. Units and modules are tested again early on in this step to minimise possible fixing costs. Testing the whole application from the user's perspective, System testing, verifies that the system is working as intended and up to user requirements and needs. System testing includes Functional and non-Functional testing, as well as Regression testing. [7 p.145.]

At the last level of Functional testing lies Acceptance testing. At this time, the product is tested in its production environment to check whether it works as expected. [7 p.145.] This is sometimes done by users or individual testers, instead of developers.

2.2.2 White and Black Box Testing

White box testing ensures that the application code performs correctly, and it requires comprehensive knowledge of the logic and internal structure of the application code. Black box testing does not require the tester to have any knowledge of the inner logic of the application and only tests the application functions from the end user's perspective. Black box testing can be applied during any development life stage from Unit to Acceptance testing. White box method can be applied during Unit, Integration and System testing however, it is most often used at the Unit level. [3 p.26; 7 p.12.]

Drawbacks to White box testing include high testing costs, as an experienced software developer is usually needed to perform these tests. Due to the demanding testing process, not all functionalities can be tested for hidden errors and maintaining the manual testing of these can be difficult. [8 p.13.] However, automating some of the repeated functionalities would decrease the costs and effort of this structural testing significantly.

Disadvantages of Black box testing include its test coverage. Test coverage can be poor if not all use case scenarios are tested. Black box tests need to be designed well for the coverage to be sufficient. [8 p.13.] Black box and End-to-end testing are used in the work done for this thesis.

2.2.3 End-To-End Testing

In End-to-end (E2E) testing, the software is tested from the end user’s perspective, whilst validating its backend functionalities. E2E is a type of Black box testing. It is a method that evaluates the entire application flow and ensures that all features work as expected in a real-world scenario. The purpose of this testing is to validate the overall behaviour of the application as well as its functionality, reliability, and performance. By testing the whole application from the intended point of view, its backend functionality, database, and network communication a comprehensive understanding of how different parts of the application work together, will be gathered. E2E testing is usually performed after Integration testing to ensure that all modules are working together seamlessly and meet expectations. After E2E testing, Acceptance testing is performed to ensure that the application meets real-life user requirements and expectations. [9.]

Benefits of End-to-end testing range from quality assurance to reduced costs. E2E testing helps to ensure that all aspects of the application are working as intended and are meeting their technical requirements. As the software is tested from the viewpoint of the end user, it increases confidence that the end product will work well once published. Software that meets its technical requirements is more likely to be also accepted by the intended users. The cost associated with defects and bugs that were detected late into the development process can be eliminated by implementing E2E testing early on and continuously throughout the development. [9.]

2.2.4 Regression Testing

Regression testing is done after any code changes or improvements to ensure that the application still functions properly, and no old bugs have resurfaced. It is a method responsible for the overall functionality and stability of existing features in the application. Regression testing must be applied after any code changes to ensure that the system stays sustainable throughout the development process.

Changes to code include new requirements to existing features, new features, code optimization, patch fixes, added dependencies or changes in configuration. The role of regression testing also includes ensuring that previously used test code remains operational after these changes. An application usually goes through multiple tests before it is integrated into the main development branch. [10.] Regression testing is usually done throughout the development progress and especially after integrating any units or changing any code. As functional tests only look at the behaviour of newly added features, they tend to overlook their compatibility with existing features. Therefore, regression testing helps prevent these time-consuming future issues. [10.]

2.3 Available Testing Frameworks

Framework selection is usually done depending on the application under development. The selection might depend on test coverage, ease of use, compatibility, and scalability. As the focus of this thesis is on End-to-end testing, three main frameworks for E2E testing will be introduced: Cypress, Selenium and Appium. Appium, as the name might suggest, started as a testing framework for mobile applications, but has since developed into a full testing environment for mobile and desktop applications. Selenium offers a wide range of customisation depending on the kind of testing that needs to be done, and the browser in which the testing will be performed. Cypress has the advantage of being easy to implement in web development teams, since it does not require any configuration to get the test writing process going. It is designed to only test web-based applications.

Appium is an open-source UI testing framework that automates mobile testing on native and hybrid mobile apps. Native applications are only made for either Android or iOS, whereas Hybrid applications work on both. Appium allows developers to test on physical devices as well as in emulators or simulated environments. Cross-platform testing is also possible for example for both Android and iOS platforms using the same test. [11.]

Selenium is a versatile testing framework that can be used for End-to-end, Functional and Regression testing. It supports most browsers and programming languages, enabling developers to create their personalised testing environment. Selenium utilises WebDriver and HTTP network requests to interact with the page. Before you can start using Selenium, you must install the desired language binding libraries, the browser, and the driver for that browser [12].

Cypress is a testing tool that runs directly in the browser and in the application, which is the biggest difference between Cypress and Selenium. Installing Cypress also instals all the tools needed to start writing tests immediately.

Cypress will be further introduced in the next chapter, as it is the chosen testing framework for this thesis.

3 Technologies Used in the Project

Reppi uses a technology stack of Angular and Amazon Web Services. Angular is a popular platform built on TypeScript for building web applications utilising components to develop scalable applications. The use of Angular does not hold much significance for the testing in this project, other than Cypress is recommended for its automated testing. Amazon Web Services (AWS) provides on-demand delivery of technology services through the Internet with pay-as-you-go pricing as it is a cloud computing platform [13]. For the testing of the main features, Cypress will be used.

3.1 Cypress

Cypress is an End-to-end testing framework. It focuses on testing web applications specifically and is designed to be used by developers and Quality Assurance Engineers. Cypress only supports JavaScript (JS) as that is the only language for automated web testing. As most front-end teams use JS in their development, Cypress has a quick learning curve and is beginner friendly. Installing Cypress only takes one line command, and the installation comes as a bundle with everything needed to start writing tests immediately. [14 p.16.]

Cypress comes packed with tools and features that make its usage enjoyable and seamless. The setup bundle includes Cypress Test Runner, a browser, and an assertion framework. These elements make it easy for developers to start testing their applications.

Cypress Test Runner is an interactive user interface that comes by default with the installation of Cypress. It enables real-time, step-by-step viewing of tests and shows the application that is being tested as Cypress interacts with it. Test runner shows each test command, its passes and failures and provides feedback to failed tests in its built-in command log. The command log highlights each failed test giving feedback on exactly which part of the command has failed. Again, as Cypress is built by JS developers for JS developers, it is easy to identify errors and fix them, as the language and errors are already familiar. [14 p.25.]

Cypress utilises a universal driver and runs directly in the browser. Running in the browser gives it an advantage over other testing frameworks, as it makes Cypress significantly faster. Furthermore, it comes with automatic wait sequences that would otherwise need to be defined in the tests. Cypress interprets commands, such as a network request, and adds a necessary wait time. [14 p.17.] Despite running in the browser, Cypress also has access to everything the application has access to, such as the DOM (Document Object Model) and any processes and methods [14 p.20]. DOM is created by the browser when a web page is loaded. It enables JavaScript to change the HTML elements, events, and attributes and is a standard for interacting with the HTML.

Running directly in the browser gives Cypress an advantage, but it also has limitations that cannot be changed due to this architecture. The only language that Cypress can ever support is JavaScript, as the test code will always be checked in the browser. Even though it is possible to have Cypress receive data from outside the browser, for example from a database, this can be cumbersome. [14 p.21.] In addition, testing in more than one browser tab at once is not permitted with Cypress. If this is needed, Cypress can be configured to use other tools to achieve this. In the same regard, within one test, Cypress only supports visits to URLs that are of the same origin. For example, navigating from GitHub to Facebook would yield an error, but navigating within any GitHub page is permitted. [14 p.22.]

Cypress is a tool designed specifically for web application testing. Though it has its limitations, it provides a good framework for quick and easy test setup. It comes packed with tools and features other testing frameworks do not have that make the testing process enjoyable for testers.

3.2 Amazon Web Services

Cloud computing delivers IT resources over the Internet with pay-as-you-go pricing. It enables you to access storage and databases without buying and maintaining physical servers. Cloud computing ensures that the application can handle peak levels of activity as you can scale the resources instantly for any application needs. Scalability ensures that you do not need to pay for storage that you do not consume, resulting in cost savings. In addition, AWS has infrastructure all over the world, allowing applications to be deployed in close proximity to end users to improve their experience. [15]

Amazon DynamoDB is a fully managed NoSQL database service provided by Amazon Web Services. It promises fast and predictable performance with seamless scalability. It can be used to create database tables that store and retrieve any amount of data and are capable of handling any degree of request traffic. [13]

4 Setup of Testing Environment

Testing environment consists of the application itself, AWS database and the Cypress testing interface. Firstly, the application was installed locally, database access was granted after, followed by installation of Cypress.

4.1 Local Installation of Application

The first challenge was to set up the Reppi application locally. After being added to Reppi’s version control system, the application was cloned through a code editor. Writing rights were enabled for the testing repository and viewing rights for the Reppi application repository. For everything to function as needed, several rights were added to local files. Npm had to be installed and the application was then functioning locally.

Npm (Node Package Manager) is a free-to-use JavaScript software library, package manager and installer. It is used to manage development or share software packages. [16.] To install npm, Node.js server environment must be installed, which can be easily done from their webpage. Npm is used to download, share, or run tools and packages. Packages can be run without downloading them locally with npx (Node Package Execute). Both npm and npx will be used during the testing process to install and use the Cypress library.

4.2 Access to Database

Reppi uses Amazon's DynamoDB as their database service and as they are a fully established product, it was essential to be careful with which rights were appropriate to grant for the testing process. Amazon DynamoDB data tables were created specifically for the testing environment, and writing rights to those were granted. DynamoDB and all Amazon services were not familiar prior to starting the testing process, but this did not hinder the process at all, as enough assistance was given to testing tasks that needed database modification.

4.3 Installing Cypress

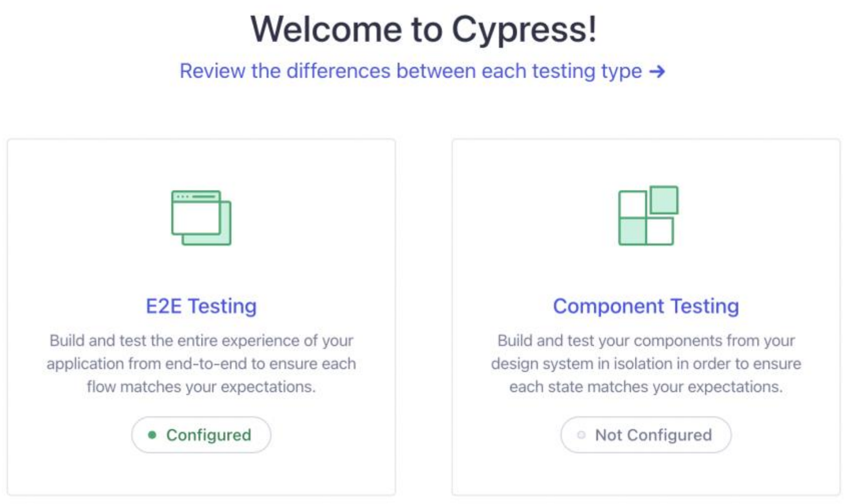
Cypress is extremely simple to install as prefaced in the earlier chapters. Only one line command is needed to fully install the whole framework using npm. Npm must be installed prior to using it to install Cypress.

```
npm install cypress --save-dev
```

After the installation is complete, Cypress can be opened, and the testing process can begin.

```
npx cypress open
```

This will open the Cypress Launchpad and the application will guide you through your testing process. It will be possible to choose between End-to-end testing and Component testing, as shown in figure 2. For the purpose of this thesis, E2E testing was chosen.



Welcome to Cypress!

Review the differences between each testing type →

E2E Testing

Build and test the entire experience of your application from end-to-end to ensure each flow matches your expectations.

● Configured

Component Testing

Build and test your components from your design system in isolation in order to ensure each state matches your expectations.

○ Not Configured

Figure 2. Cypress Launchpad testing type choice.

Launchpad then shows Cypress configuration files and asks for modifications. None were needed in this case and files stayed as default. A preferred browser is then chosen, in this case, Google Chrome (figure 3). All these decisions can be changed later if desired.

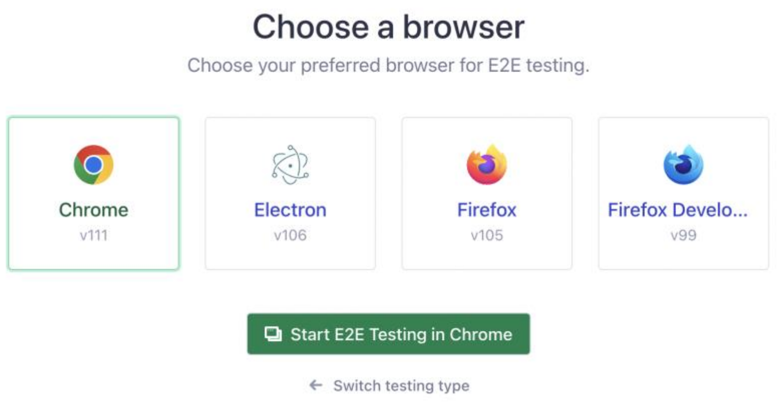


Figure 3. Cypress Browser choice.

It is time to create a test. Cypress will ask if you want to create a new spec and by choosing so, it will ask you to name it. The newly generated test, a spec, will now show up in your code editor of choice, and the test writing process can begin.

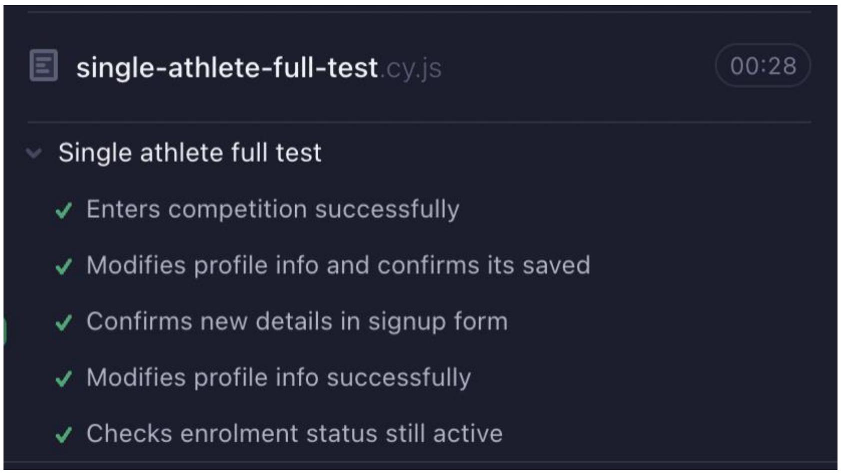
5 Test Specification

Testing was asked to be done for repetitive tasks, such as competition sign-up and athlete information modification. The testing started with competition sign-ups for individual athletes. This included finding an available competition and filling out its sign-up form. As some competitions are for individual participants and others are completed in pairs, it affects their sign-up process and a test written for one participant will not be viable for a group competition.

Once the testing was underway, it became clear that the three test suites would include individual athletes, pairs, and teams. Therefore, three tests were outlined as individual athlete competition sign-up and information editing, pair sign-up and information editing and team sign-up and editing. The goal was to make the tests as resilient to changes as possible, to ensure that they would not break in case of any modifications.

5.1 Single Athlete Full Test

The first full test consisted of a single athlete signing up for a competition and modifying their details. The goal was to make sure that the modified details were saved and auto-filled correctly in future competition enrolments and didn’t affect the enrolment status. Figure 4 shows the five objectives that “Single athlete full test” consists of besides logging in: “Enters competition successfully”, “Modifies profile info and checks if saved”, “Checks new details in signup form”, “Modifies profile into” and “Checks enrolment status”.



single-athlete-full-test.cy.js 00:28

Single athlete full test

- ✓ Enters competition successfully
- ✓ Modifies profile info and confirms its saved
- ✓ Confirms new details in signup form
- ✓ Modifies profile info successfully
- ✓ Checks enrolment status still active

Figure 4. Single athlete test objectives.

Two Test User data objects are used throughout this testing process, Test User A and Test User B. The first step of this test was to log the user in. The same user login credentials are used for all the tests.

Competition sign-up is a process used by almost all Reppi users and was therefore the main testing target. The test included choosing an upcoming competition “Turun Tuomiopäivä” from a list of all available competitions. As there are multiple different competitions available, the test must choose the specific competition that meets the requirements of testing objectives. The competition should be for individual athletes only and, in this case, be free of charge. This is not specified in the test itself. The competition choice is manually written into the test.

Competitions can sometimes have different divisions depending on age or gender. The options are not always the same, as they are unique to each event organiser, so the selection must be written in the test manually. In this case, the test was written for the open male division “Avoin, miehet”, as seen in figure 5.

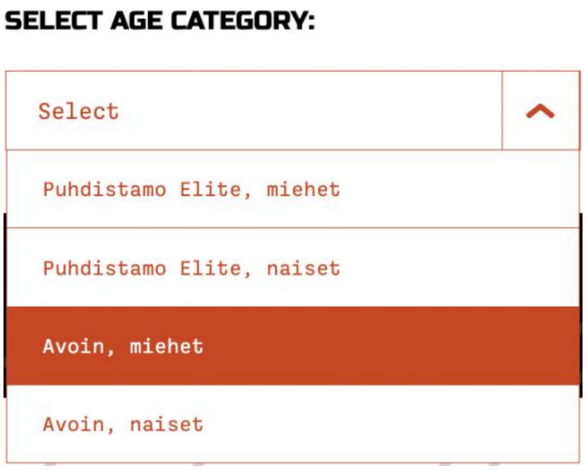
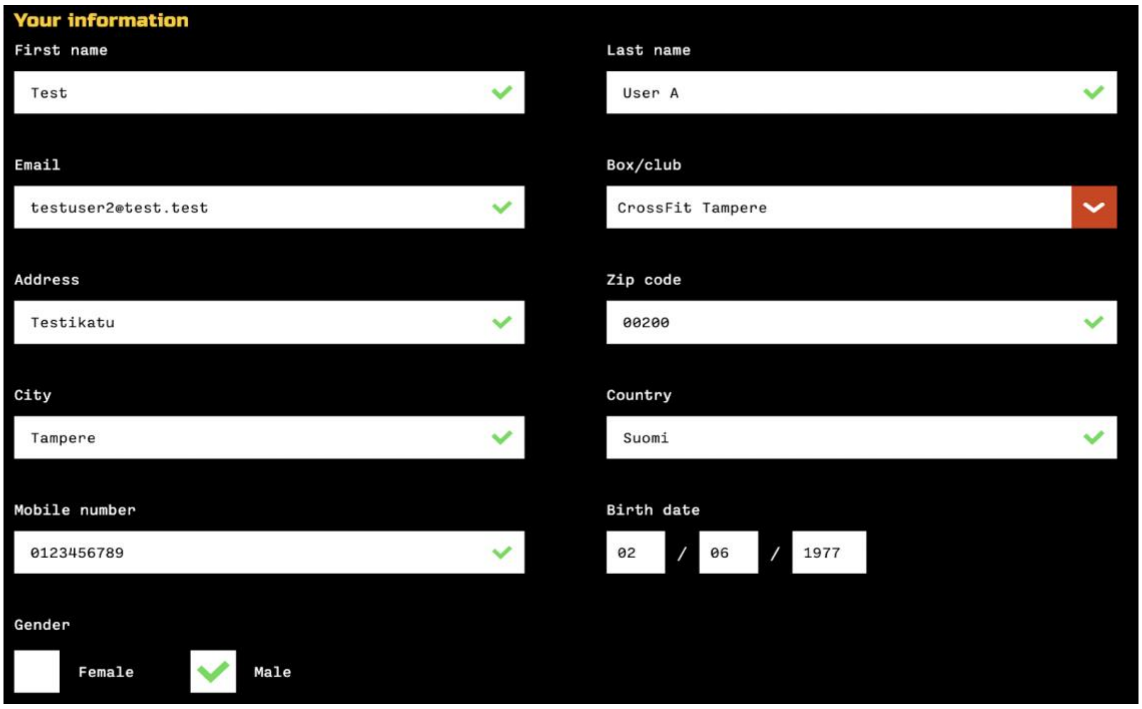


Figure 5. Age category selection.

Figure 6 illustrates the sign-up process which consists of filling out a form with your information, such as first and last name, email, phone number, address details and birthday. Some information is auto filled from the user's profile if it has been added by the user. The competition was signed up using Test User A.

This particular test also included further fields for the athlete's CrossFit ID, food allergies and shirt size.



Your information

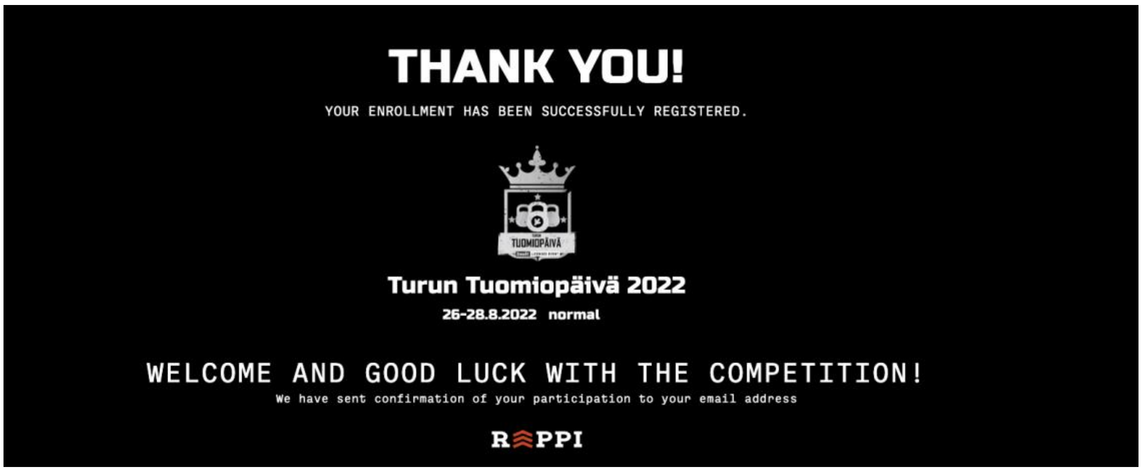
First name	Last name
Test ✓	User A ✓
Email	Box/club
testuser2@test.test ✓	CrossFit Tampere
Address	Zip code
Testikatu ✓	00200 ✓
City	Country
Tampere ✓	Suomi ✓
Mobile number	Birth date
0123456789 ✓	02 / 06 / 1977

Gender

☐ Female [✓ Male]

Figure 6. Personal information form.

The first part of this test “Enters competition successfully” was considered successful if a container named “enroll-success-container” was visible on the page after going through the sign-up process (figure 7). It is worth acknowledging that there are more reliable ways to check whether a form submission has been successful, but with the time constraints of this process, this method of confirmation was chosen instead.


THANK YOU!

YOUR ENROLLMENT HAS BEEN SUCCESSFULLY REGISTERED.

Turun Tuomiopäivä 2022

26-28.8.2022 normal

WELCOME AND GOOD LUCK WITH THE COMPETITION!

We have sent confirmation of your participation to your email address

Figure 7. Successful sign-up banner.

The test is then redirected to the user's profile, as for individual athlete competitions there is no option to edit details through the competition page. Profile info modification is very similar to filling out a competition sign-up form, as it has the same format. Test User A data was replaced with Test User B data and saved. New data is searched for in the profile, and if found in the correct fields, the test “Modifies profile info and checks if saved” has been successful.

As mentioned earlier, autofill is used on some of the competition sign-up forms. Even though the application receives the data for the profile from a database, and having the correct details saved on the profile, should also yield correct details on the sign-up form, it does not hurt to be thorough. The athlete is deleted from the competition they previously signed up for and is signed up again. When the sign-up form comes up again, each field is checked for Test User B data. If all input fields match the test data, the competition is signed up to and “Checks new details in signup form” -test has been successful.

The test is wrapped up by changing the personal details back to Test User A, with a quick check that the profile name has been updated after saving, concluding our penultimate test “Modifies profile data”. Lastly, the enrolment status for the competition is checked, by searching for the competition on the athlete's profile tab “competitions” (figure 8).

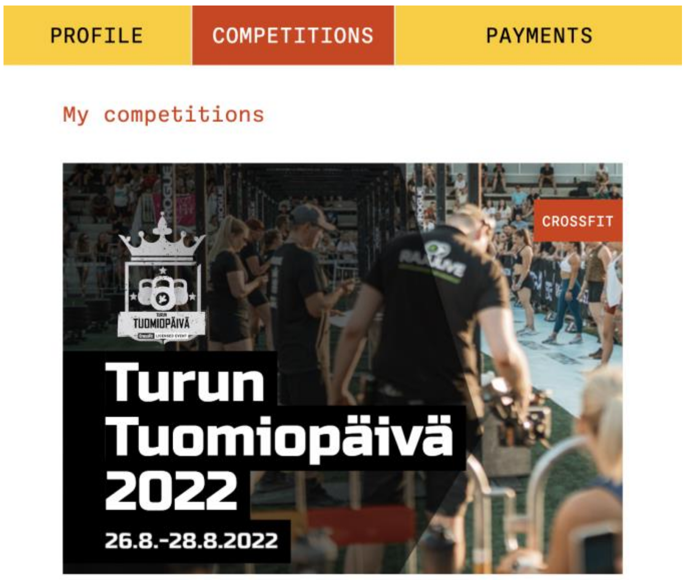


Figure 8. Turun Tuomiopäivä competition found in user's profile.

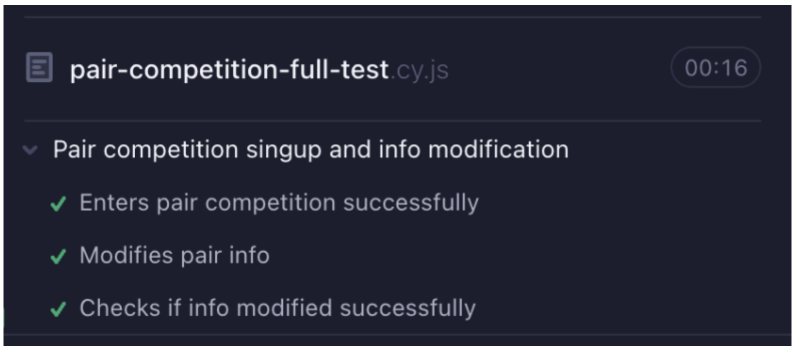
If a correct competition is found from items in the said tab, "Checks enrolment status" is successfully complete.

"Single athlete full test" checks all the most necessary functions that can be performed by a single athlete. From logging in, to enrolling in a competition and editing personal information, this test catches any errors that may arise.

5.2 Pair Competition Full Test

The second test completed was testing the sign-up and detail editing for a pair competition. Pair competition sign-up is very similar to single-athlete competitions, but it has additional input fields for a competition partner.

The main goal of this test is to ensure that detail editing saves successfully. Figure 9 shows that this test consists of three objectives: "Enters pair competition successfully", "Modifies pair info" and "Checks if info modified successfully". This competition was not free to enter and had an additional aspect of paying for entry. Again, two Test User objects were defined, and Test User A was logged in.

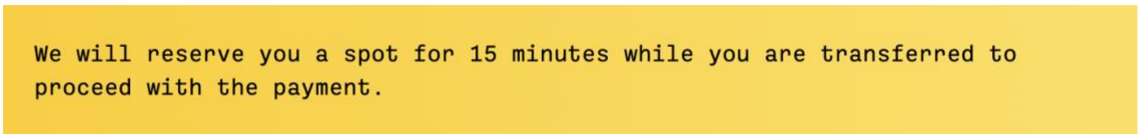


```
pair-competition-full-test.cy.js                00:16
Pair competition singup and info modification
  ✓ Enters pair competition successfully
  ✓ Modifies pair info
  ✓ Checks if info modified successfully
```

Figure 9. Pair competition test objectives.

Much like the first competition sign-up process, an upcoming competition “Häjyin” was chosen. This competition choice must meet the criteria of being open for sign-up and having a pair competition division. Like with the previous competition, this selection was written into the test.

The pair competition in the testing environment was not free of charge and therefore had to include slight modifications to the sign-up test code. The sign-up form is identical to the single athlete sign-up form, with the exception of adding the information of a competition partner. Regarding the payment for entry, for the scope of this test, it is good to note that an entry is valid for 15 minutes before it needs to be paid (figure 10).



We will reserve you a spot for 15 minutes while you are transferred to proceed with the payment.

Figure 10. Reserved entry banner.

During these 15 minutes, the entry can be edited and tested exactly like a paid entry. For the purpose of this project, it is not necessary to test the payment functions, as that will fall out of the scope of the application functionalities. As the scope is focused on competition entry and information editing, this unconfirmed entry is enough. The test finds and clicks a “Pay” button and the test moves on to the competitions tab in the user's profile.

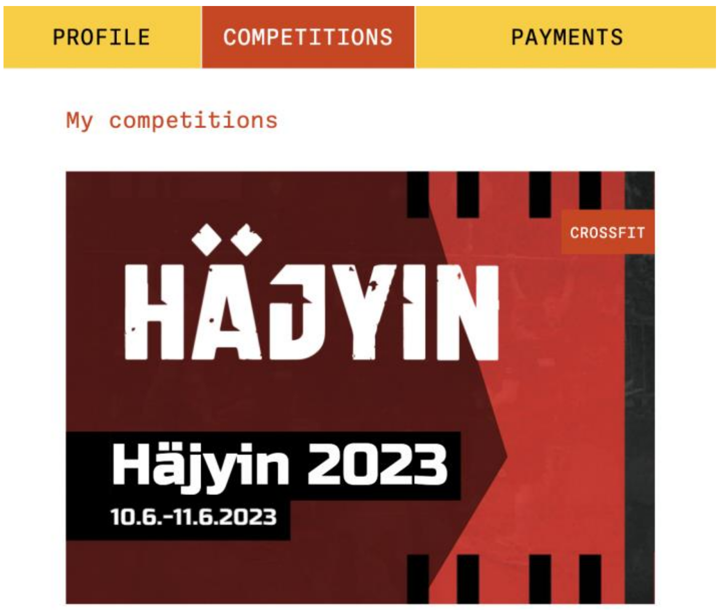
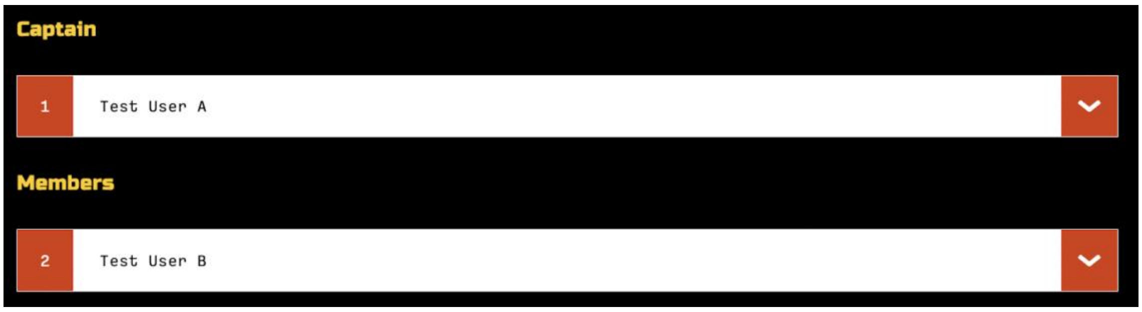


Figure 11. Competition found in user profile.

If the page contains the competition that was just signed up for (figure 11), the entry is considered successful and the objective “Enters pair competition successfully” has been finalised.

Once the pair has been signed up for the competition, the test moves on to editing their details. In this test, Test User A and Test User B were used, and for the purpose of editing their details, they were swapped around. At the start, Test User A is considered the captain of their team, and Test User B is a regular member as seen in figure 12.

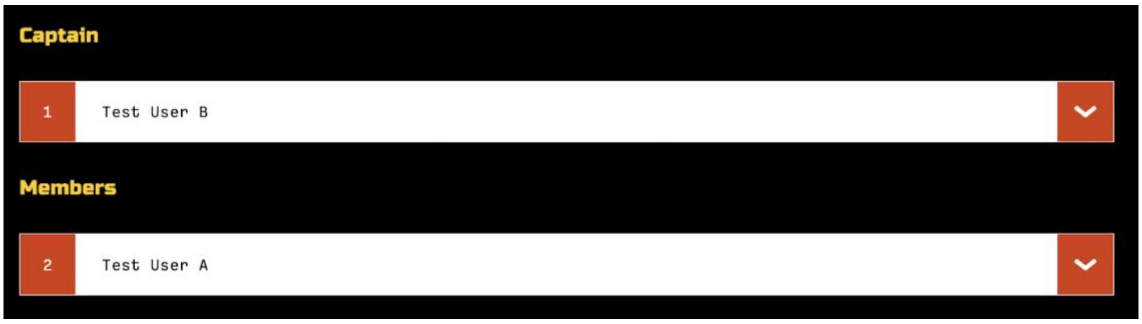

Captain

1 Test User A

Members

2 Test User B

Figure 12. Test User A and B before information editing.


Captain

1 Test User B

Members

2 Test User A

Figure 13. Test Users A and B after information editing.

These rankings do not have any other significance, other than which user is the main contact person for the pair. Their detail editing form is, again, identical to the sign-up form and Test User A details are replaced with Test User B details and vice versa. Changes are saved and the second objective “Modifies pair info” is finalised. Lastly, the changes are checked on the team details page. If Test User B is now considered to be the team captain and Test User A is a regular member, the information editing was successful (figure 13).

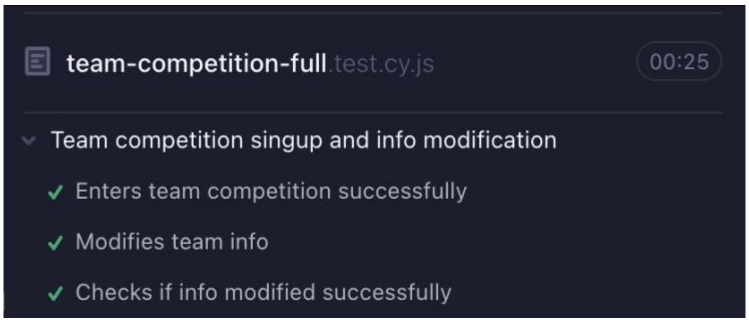
“Checks if info modified successfully” is now complete, as is the pair competition sign-up and info modification test “Pair-competition-full-test”.

It would later be decided that captain editing through this form and the team editing forms should be removed, as those are the account holder’s details, and should only be editable through the profile tab. Further details of this will be provided in chapter 7.2 Challenges with Testing.

5.3 Team Competition Full Test

The final test that was written for the application, was testing the sign-up function for team competitions. Again, the main objectives of this test are competition sign-up and athlete information editing. The sign-up process is identical to the sign-up of a pair competition, with the addition of two more team members.

This test also consists of three objectives: “Enters team competition successfully”, “Modifies team info” and “Checks if info modified successfully” (figure 14). This competition also had the additional aspect of paying for entry. Four Test User objects were defined, and Test User A was logged in.



```
team-competition-full.test.cy.js                00:25

Team competition singup and info modification
  ✓ Enters team competition successfully
  ✓ Modifies team info
  ✓ Checks if info modified successfully
```

Figure 14. Team competition objectives.

Just like the first two competition sign-up processes, an upcoming competition “Yyteri Sandstorm” was chosen. This competition choice must meet the criteria of being open for sign-up and having a team competition division. Again, this choice was explicitly written into the test. The chosen competition had team entries that consisted of four team members, one of which is the team captain. Other team members are Test User B, Test User C and Test User D. The test code that was used for the pair competition test could also be used for this test, with the addition of two more team members.

The competition is, again, not free to enter, and therefore utilised the same test code as the pair competition. The entry is valid for 15 minutes before needing to be paid, which is enough for the purpose of this test. The team is signed up for the “Yyteri Sandstorm” competition, and the sign-up is checked from the competitions tab in the user's profile (figure 15).

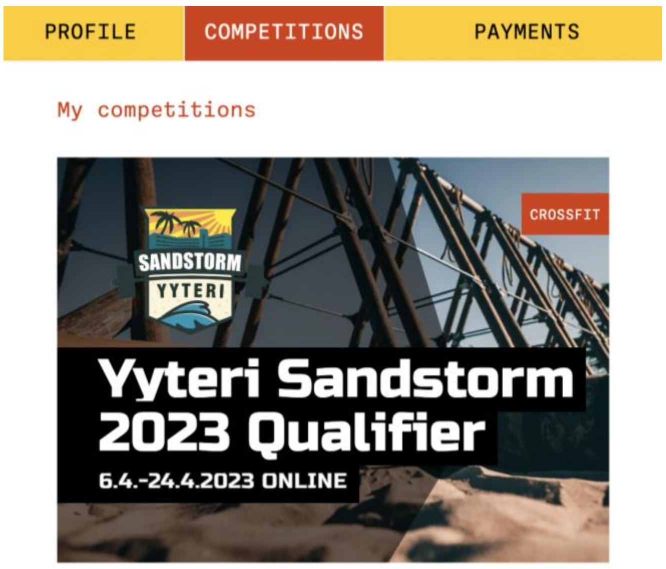
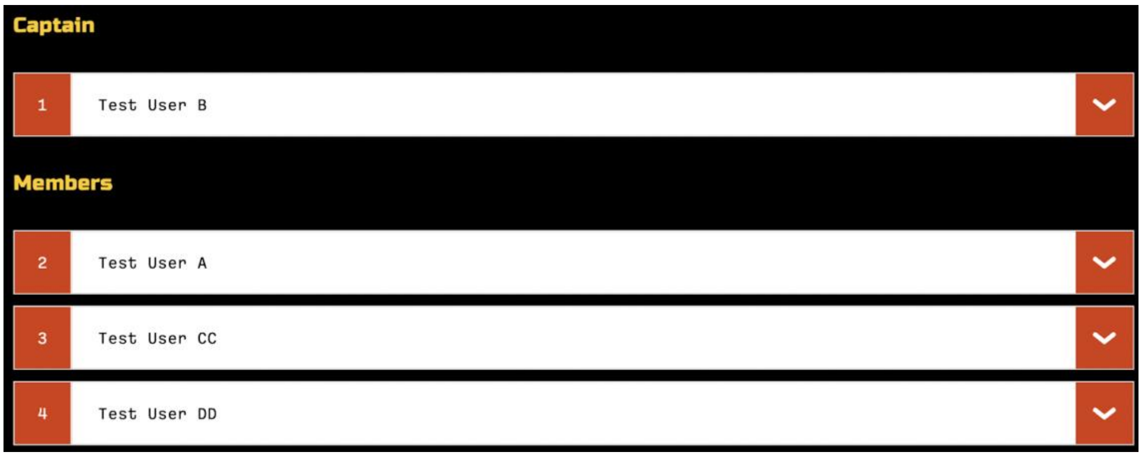


Figure 15. Competition found in user's profile.

If the page contains the competition, the sign-up has been successful, finalising the “Enters team competition successfully” objective.

Information editing follows the same logic as in the previous tests in the objective “Modifies team info”. Captain and the first team member are swapped around, and other team members' last names are edited to include an additional letter. An additional letter is added to the remaining members, rather than swapping the members' places, since in the final view, the members are in alphabetical order, and the result of editing would not otherwise be noticeable.



Captain

1 Test User B

Members

2 Test User A

3 Test User CC

4 Test User DD

Figure 16. Team information after successful editing.

As seen in figure 16, if the test finds that Test User B is the captain, and members include Test User A, Test User CC and Test User DD, the objective “Checks if info modified successfully” is deemed successful. This objective finalises the team testing as well as the whole suite. As prefaced at the end of the previous chapter, team captain’s information editing will be deemed unnecessary and removed later.

6 Execution of Tests

With Cypress, starting the testing process is extremely simple and testing can be jumped in to as soon as the application is up and running. The first step of every test is navigating to the desired webpage, in this case to localhost:4200, where the testing version of the app is running. From there, each test will differ in its objectives, but the overall use of commands, locators and actions will stay the same.

6.1 Cypress Commands and Session

Each test starts with the user logging in. Logging in is a repetitive process with reusable code that is utilised across many different tests. As with any good coding practices, reusing code is commendable, but duplication should be avoided. Duplicating the same lines of code across different files can harm productivity and be time-consuming in case of any arising errors or issues. To shorten the code for each of the tests written for Reppi, logging in was saved into a Cypress Command. As illustrated below in listings 1 and 2, this shortened the code from seven lines to one single line. The longer code is still found in the commands file however, it is only found once, instead of three times across three tests.

```
cy.visit("http://localhost:4200/");
      cy.get('[data-cy="login_button_nav"]').click();
      cy.url().should("include", "/login");
      cy.get('[data-cy="login_email"]').clear().type(loginemail);
      cy.get('[data-cy="login_password"]').clear().type(loginpassword);
      cy.get('[data-cy="login_button"]').click();
      cy.url().should("include", "/competitions");
```

Listing 1. Full login test code.

```
cy.login(loginemail, loginpassword);
```

Listing 2. Login command code.

Saving reusable code as Cypress Commands allows for quicker test writing and more readable code. `Cy.login()` holds the longer code in a separate commands file, and therefore, does not need to be repeated in every test file. Parameters loginemail and loginpassword are defined in each test file, for clarity and to allow quick customisation. Nevertheless, using the UI to log in is not the most effective way and can take a toll on performance and testing time, depending on the structure of the application. As login is the process that oftentimes has the longest loading time, it was saved into a Cypress Session with `cy.session()`, as seen in listing 3.

```
cy.session("login", () => {
      cy.login(loginemail, loginpassword);
 });
```

Listing 3. Login wrapped in Cypress Session.

With wrapping the login command in `cy.session()`, Cypress will save it for the entirety of the testing session, until the test is closed. After running the command 'login' once, it is saved, and a session is created (figure 17).

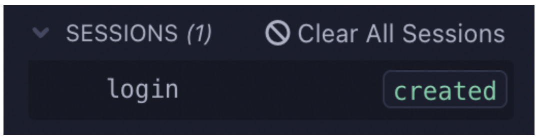


Figure 17. Login session created.

When the test is run again, instead of running the whole command again, Cypress will restore the session (figure 18). This reduces login loading time by roughly five seconds.

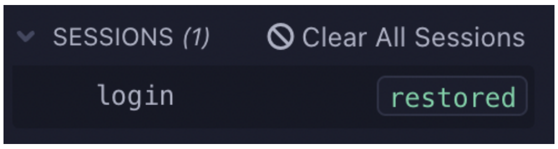


Figure 18. Login session restored.

Reducing test loading time by a few seconds does not seem significant, however, it can drastically save time, especially when running the test dozens of times an hour. Once signed in, the test was redirected to the competitions page and the three tests can begin.

6.2 Cypress Locators

As seen in the code examples above, a locator `[data-cy="login_email"]` is used to select the right elements from the HTML. It is best to choose selectors that can stand the test of time and are resilient to changes. To avoid any problems, it is recommended to avoid targeting elements based on their dynamic attributes. These include `class`, `tag`, `ID`, or elements that may change their `textContent`. The worst way one could target an element is by using generic attributes such as ‘button’. Targeting by ID or class is discouraged, as those are highly likely to change. IDs can be changed, and classes added or removed, which causes the test to be fragile and prone to breaking.

Adding a `data-cy` attribute makes it easier to select the right elements. For example, in the login function, an email input field must be selected. A `data-cy="login_email"` attribute is added to the input HTML element and then accessed via `cy.get('[data-cy="login_email"]')` in the Cypress test file. `Cy.get()` usually includes a selector to find matching HTML and DOM (Document Object Model) elements. `Cy.get()` is a query that is usually followed by further action commands, such as `.click()` or `.type()`.

In some cases, a `data-cy` attribute cannot be added to HTML. This can either be because an element is generated to the DOM from a database or a JavaScript file. Though it is possible to get Cypress to select elements from these, due to time constraints, this was not done for a few of the testing objectives. In these cases, `cy.contains()` was used. `Cy.contains()` yields a DOM element based on its text content that a user can see on the page. For example, cy.contains(“Login”) will yield a button with the text “Login” on it. This was used for choosing competitions from the competition array as shown in listing 4, instead through their database tag.

```
cy.get('[data-cy="competition"]').contains("Turun Tuomiopäivä").click();
cy.get('[data-cy="competition"]').contains("Häjyin").click();
cy.get('[data-cy="competition"]').contains("Yyteri Sandstorm").click();
```

Listing 4. Competition selection by text content.

The same method was also used when selecting age categories as shown in listing 5.

```
cy.get('[data-cy="age_select"]').click().contains("miehet").click();
cy.get('[data-cy="age_select"]')
      .click()
      .contains("Parisarja, Miehet")
      .click();
```

Listing 5. Age category selection by text content.

6.3 Cypress Actions

For Cypress to be able to interact with the elements on the page, it must be given actionable commands to do so. These commands include `.click()`, `.clear()`, `.type()`, `.eq()` and many others. These action commands simulate the user's interaction with the application.

`.Click()`, `.clear()` and `.type()` are the most common commands used during this testing process. As the names indicate, `.click()` clicks on a chosen element, `.clear()` clears a field and `.type()` types into a chosen field. These actions, however, must meet a few rules to work properly. For example, Cypress cannot click on an element that is not visible on the page, even if it can find it on the back end. It is not possible to clear fields or type into fields that do not support these actions.

A problem was run into whilst writing the second test, pair-competition-full-test, regarding a form field not being clearable. Clearing some form fields before typing into them was needed when modifying auto-filled information. Despite looking and behaving like regular input fields they were rpi-form-fields. RPI stands for Reading Position Indicator and due to this, Cypress did not see it as a clearable field. This is not an unknown issue amongst test writers and a workaround for this is available, as these fields are indeed clearable. `.Clear()` is an alias for `.type("{selectall}{backspace}")` and using this instead of `.clear()` worked well on these fields.

To fill in team competition sign-up forms, `.eq()` was used. `.Eq()` stand for "equal" and it finds a DOM element at a specific index in an array of elements.

As the necessary number of member input fields are generated automatically and therefore use the same Cypress locator, `.eq()` was used to indicate which of these fields was needed (listing 6).

```
cy.get('[data-cy="signup_member_name"]').eq(0).clear().type(info2.fname);
cy.get('[data-cy="signup_member_name"]').eq(1).clear().type(info3.fname);
cy.get('[data-cy="signup_member_name"]').eq(2).clear().type(info4.fname);
```

Listing 6. Cypress locator .eq() used to select different name fields from a form.

6.4 Custom Tools

For the testing to be successful, it was essential to get access to the database through Cypress. As a user can only sign up for a competition once and cannot drop out of a competition through the application themselves, Cypress needed to remove the athlete from the competition before each test. This enabled the same user to sign up for the same competition repeatedly. To be certain, that the competition is cleared of any athletes altogether, all athletes that were signed up were removed before the beginning of the test. Database clearing can be done before, after or in the middle of each test. It was decided that clearing the database before each test was a better option. This enables the tester to go and interact with the application in the state it was at the end of each test. For example, in case of failed detail editing, the tester can attempt to edit details themselves and search for any console errors, without having to sign up the user manually beforehand.

Two custom tools for clearing out the database were done by a helpful Reppi developer. One for cleaning out a competition of its athletes and another for deleting a user completely. There has not yet been any reason to use the tool for deleting users, however, it does present opportunities for future development.

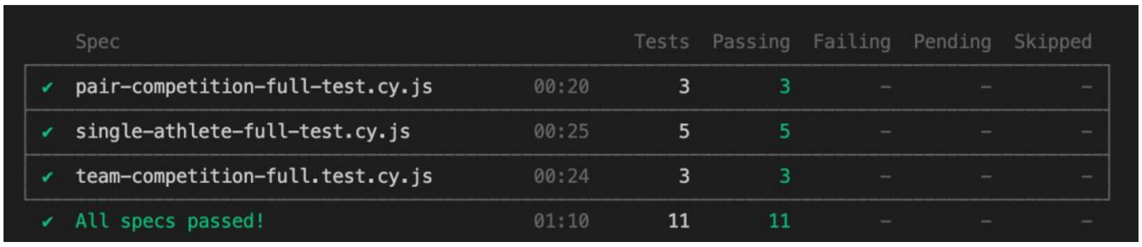
A future development opportunity is also related to editing the database during a test. Sometimes competitions can fill up whilst an athlete is still signing up. The sign-up should then fail and not allow the extra athlete to participate. This feature could be tested by filling the competition during the sign-up process, and therefore failing the sign-up. The test would be successful if the sign-up fails.

7 Results

Three test suites were successfully completed for Reppi. At the moment of writing this thesis, it is unknown when Reppi will integrate the tests into their development process; however, that was the goal for the completed tests. Like any project, this one did not come without its successes and challenges.

7.1 Successes of Testing

As can be seen in figure 19, three working tests were completed according to the requirements that Reppi had set; a test for each of the competition sign-up cases; single athlete, pair competition and team competition.



```
    Spec                                    Tests  Passing  Failing  Pending  Skipped
  ✔ pair-competition-full-test.cy.js  00:20     3        3        -        -        -
  ✔ single-athlete-full-test.cy.js    00:25     5        5        -        -        -
  ✔ team-competition-full.test.cy.js  00:24     3        3        -        -        -
  ✔ All specs passed!                 01:10    11       11        -        -        -
```

Figure 19. Three successful tests.

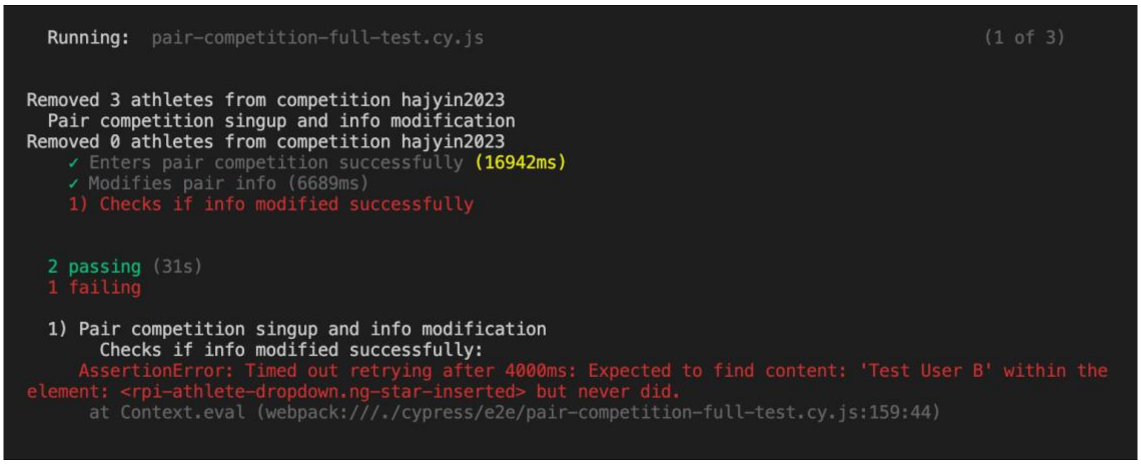
Tests were done by upholding the best practices set by Cypress testing professionals, to the best extent that was possible, considering the time constraint. Writing rights to the Reppi application repository were granted and a separate branch for Cypress was created where Cypress locators were pushed to. Completed tests were pushed to their respective repository one final time.

7.2 Challenges with Testing

The first challenge that was encountered, was finding the right HTML elements from the application code to add selectors to. As with any large application, it takes time to realise where each component stems from and what it affects. Though Black box and End-to-end testing do not by definition need in-depth knowledge of the application's logic, it inevitably becomes familiar to the tester, to some degree, throughout the testing process. To make the tests as resistant to changes as possible, every single input field needs to be found in the application code for Cypress to type into it. However, when code duplication is avoided and components are reused, can Cypress locators be reused as well. Nevertheless, this also brought up challenges that were mentioned in the 6.3 Cypress Actions chapter. If an input field was not explicitly an HTML input, form or text field, cumbersome workarounds followed from one test to another.

Another challenge was related to limited knowledge of the application's logic. During the final test for team competition sign-up and info editing, the lack of knowledge of how that application functions was the culprit to many failed tests. As in team competitions, there was more than one team member, these were listed together in alphabetical order. This order was not known when writing the tests. At first, it seemed as if the detail editing did not work, as the sequence of Users stayed the same and the test failed as it was looking for a different sequence. Finally, it became clear that detail editing did work, and Users were simply always listed in a specific order, the test logic was easily fixed, and it became successful.

In previous tests after a pair has been signed up for a competition and their details need to be edited, the two User objects were simply swapped around. Captain User A became a member User B and vice versa. However, as the testing was near completion, the tests were run multiple times to ensure that they were successful every time, a rare but persisting problem occurred. Very rarely the detail editing for a team captain failed, as shown in figure 20. The test could not find the swapped details of Test User B in the captain's slot, but correct details would be in the member field.



```
  Running:  pair-competition-full-test.cy.js                                     (1 of 3)


Removed 3 athletes from competition hajyin2023
  Pair competition singup and info modification
Removed 0 athletes from competition hajyin2023
    ✓ Enters pair competition successfully (16942ms)
    ✓ Modifies pair info (6689ms)
    1) Checks if info modified successfully


  2 passing (31s)
  1 failing

  1) Pair competition singup and info modification
       Checks if info modified successfully:
     AssertionError: Timed out retrying after 4000ms: Expected to find content: 'Test User B' within the
element: <rpi-athlete-dropdown.ng-star-inserted> but never did.
      at Context.eval (webpack:///./cypress/e2e/pair-competition-full-test.cy.js:159:44)
```

Figure 20. Team captain editing error.

After discussing this with the developers and asking for advice, the conclusion was drawn that editing the captain's details should not be possible through the team editing form at all, as these are the account holder's details. The error was the result of a previous, lingering, logic that had since been modified. Testing of this was removed from pair and team tests, and the possibility to modify captain's details might be locked in the future altogether. This again illustrated the challenge behind not having the knowledge of the application's logic, or history, but also how the testing process had already uncovered some possible future improvements.

7.3 Future Development Opportunities

Although the tests are done well and by upholding best practices as much as possible, they are not perfect and will need checking and updating as the application develops. As mentioned previously, not all functions were able to be given their own selectors and, for example, competitions are chosen by their name and shirt sizes by their gender and size. This can be developed further, and competitions or shirt sizes could be chosen based on their database tag or

ID, to avoid any language-related changes. Though changing the application language does not currently affect any of the competitions that were tested, it is a possibility that in the future it might. This could be avoided by including language changes in the tests themselves.

Currently, only regular user functions are tested. Administrative functionalities like adding competitions or scores can, and should, also be tested. These tasks are repetitive for the administrative users. As there are significantly less administrative users, than regular users, testing of these features was not the priority during this project, but does present a future development opportunity.

To keep up with the best testing practices, it is necessary that tests don’t depend on other tests to be viable. The state of the previous test shouldn't matter when testing a related feature. Even if tests would always be run one after another, they should not be dependent on each other. For example, detail modification should not be dependent on the outcome of competition sign-up. This can be achieved with database manipulation to ensure that there is always a clear starting base for each testing block. Each test needs to be treated as a separate use case to ensure longevity and resilience to change.

The testing practices that were implemented during this project, are not limited to one company or application. This approach can be used by any development team in any sector. Automated testing with Cypress or any other testing framework could be integrated into development teams with continuous integration practices. In continuous integration, developers integrate their newly written code more often than normally, and this code is tested by automated tests. This way, any errors caused by small code changes are caught early and fixed quickly. In practice, the tests can be run each time code is pushed into a repository.

8 Conclusion

Testing is an integral part of the web application development lifecycle. It should always be done for all web applications. Developers and Quality Assurance Engineers must ensure that the application aligns with the set guidelines: customer needs and technical requirements. Testing can be done manually, or it can be automated to the extent that is fit for purpose. Despite the initial costs, it is recommended that testing for repetitive tasks, such as signing up, logging in, editing details or any other tasks that do not vary from user to user, are automated. Automating repetitive tasks does not fully absolve developers and testers from their testing responsibilities, as tests do need to be observed and maintained. Nevertheless, a well-written test can catch errors more efficiently than manual testing, making the initial investment of time and resources worthwhile, especially in the case of larger web applications.

Web application testing can be done in a plethora of different ways. Each team chooses the best approach for their needs, whether that is Functional testing throughout the applications lifecycle or Regression testing, to ensure all features work as intended after any additions. Moreover, automated testing is a cornerstone of continuous integration. Whichever approach it may be, there is a programme available for their specific needs and expertise. It is up to the team to decide which functions to test and where to start their testing process. Regardless of where they may start, a clear understanding of testing requirements should be outlined for the most efficient process.

Automating web application testing has been gaining popularity for an obvious reason: high-quality end product. Catching errors early on in development, fast test reruns, and eliminating human error are just a few of the examples of how automated web testing can save time and resources when developing a high-quality product. Automating web application testing is a worthwhile progress that yields many benefits throughout the application's lifecycle.

9 References

1 Leotta, M., Clerissi, D., Ricca, F., & Tonella, P. 2016. Approaches and tools for automated end-to-end web testing. In Advances in Computers. Elsevier.

2 Shakya, S. and S., S. 2020. Reliable automated software testing through hybrid optimization algorithm. Journal of Ubiquitous Computing and Communication Technologies, 2(3).
Available at: https://doi.org/10.36548/jucct.2020.3.002.

3 Ammann, P. and Offutt, J. 2016. Introduction to software testing. Cambridge: Cambridge University Press. ISBN: 9781107172012.

4 Taipale, O., Kasurinen, J., Karhu, K., Smolander, K. 2011. Trade-off between automated and manual software testing. International Journal of System Assurance Engineering and Management, 2(2).
Available at: https://doi.org/10.1007/s13198-011-0065-6.

5 Leotta M, Clerissi D, Ricca F, Tonella P. 2013. Capture-replay vs. Programmable web testing: An empirical assessment during Test Case Evolution. 20th Working Conference on Reverse Engineering (WCRE).
Available at: https://doi.org/10.1109/wcre.2013.6671302.

6 Paspelava, D. 2023. What is unit testing in software: Why Unit Testing Is Important, Exposit. Available at: https://www.exposit.com/blog/what-unit-testing-software-testing-and-why-it-important/
(Accessed: April 11, 2023).

7 Vila, E., Novakova, G. and Todorova, D. 2017. Automation testing framework for web applications with selenium WebDriver. Proceedings of the International Conference on Advances in Image Processing.
Available at: https://doi.org/10.1145/3133264.3133300.

8 Ehmer, M. and Khan, F. 2012. A comparative study of white box, black box and grey box testing techniques. International Journal of Advanced Computer Science and Applications, 3(6).
Available at: https://doi.org/10.14569/ijacsa.2012.030603 .

9 What is end to end testing? 2023. BrowserStack.
Available at: https://www.browserstack.com/guide/end-to-end-testing
(Accessed: March 7, 2023).

10 Katalon. What is regression testing? Software testing definition & tools, katalon.com. Katalon.
Available at: https://katalon.com/resources-center/blog/regression-testing
(Accessed: March 7, 2023).

11 Appium. Cross-platform Automation Framework for all kinds of your apps built on top of W3C Webdriver Protocol. GitHub.
Available at: https://github.com/appium/appium
(Accessed: March 8, 2023).

12 Getting started. Selenium. Available at:
https://www.selenium.dev/documentation/webdriver/getting_started/
(Accessed: February 22, 2023).

13 AWS What is Amazon DynamoDB? - Amazon DynamoDB. Available at:
https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html
(Accessed: February 24, 2023).

14 Mwaura, W. 2021. End-to-End Web Testing with Cypress: Explore techniques for automated frontend web testing with Cypress and JavaScript. Packt Publishing Ltd.

15 AWS. What is cloud computing? – AWS.
Available at: https://aws.amazon.com/what-is-cloud-computing/
(Accessed: February 24, 2023).

16 About NPM. npm Docs. Available at: https://docs.npmjs.com/about-npm
(Accessed: April 4, 2023)