Behavior Driven Platform-Independent Testing for Mobile Apps

Bachelor Thesis

by

R. J. Vennik & W. F. de With

Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Coach:	Dr. ir. F. Palomba,	TU Delft
Supervisor:	Ir. W. Van,	bunq

Abstract

Software testing is widely recognised as a crucial activity for the development of modern software systems, since it points out defects and errors that were made during the development. Testing also improves the reliability and consistency of an application.

However, testing behaviour on mobile devices is not easy because of a number of reasons: (i) manual testing of behaviour requires a lot of valuable time of developers, (ii) the problem of covering all possible operating systems, (iii) covering all possible mobile devices, (iv) and frequently releasing new versions of apps is the industry standard. In addition bugs in mobile apps can cause the reputation of the app on the store to decrease and thereby the the number of downloads. Hence the following question is central to our research: can we automate user interface testing for mobile apps and multiple platforms without human interaction?

To overcome the limitation of multi-platform testing, as well as to help non-developers in the creation of tests, we introduce a novel tool that is able to test behaviour on mobile devices by defining expected scenarios. These scenarios consist of simple human readable instructions that are easy to write and understand by non-developers. These scenarios can then be tested against multiple operating systems and devices by simply connecting a device and pressing a button.

Preface

In front of you is the Bachelor Thesis on behaviour driven platform-independent testing for mobile apps. It is written as part of the Bachelor Project at the end of the Computer Science Bachelor at the Delft University of Technology. This is the result of 10 weeks of hard work and dedication.

We would like to thank everyone who has helped and supported us during our project. Especially Wessel Van for supervising us at bunq, Fabio Palomba for coaching us, Peter Jansen for giving us valuable feedback on the testing framework, Ali Niknam for reviewing our code, and finally everyone at bunq who supported us.

R. J. Vennik & W. F. de With
Amsterdam, June 2017

Contents

1 Introduction

Software testing is widely recognised as a crucial activity for the development of modern software systems, since it points out defects and errors that were made during the development. Testing also improves the reliability and consistency of an application.

However, testing behaviour on mobile devices is not easy because of a number of reasons: (i) manual testing of behaviour requires a lot of valuable time of developers, (ii) the problem of covering all possible operating systems, (iii) covering all possible mobile devices, (iv) and frequently releasing new versions of apps is the industry standard. In addition bugs in mobile apps can cause the reputation of the app on the store to decrease and thereby there the number of downloads. Hence the following question is central to our research: can we automate user interface testing for mobile apps and multiple platforms without human interaction?

To overcome the limitation of multi-platform testing, as well as to help non-developers in the creation of tests, we introduce a novel tool that is able to test behaviour on mobile devices by defining expected scenarios. These scenarios consist of simple human readable instructions that are easy to write and understand by non-developers. These scenarios can then be tested against multiple operating systems and devices by simply connecting a device and pressing a button.

The proposed tool was born in collaboration with bunq, a Dutch FinTech bank that provides financial services exclusively through their mobile apps.

This thesis is organised as follow: First, in chapter 2, we define the problem that bunq gave us to solve. Then, in chapter 3, we analyse the given requirements and chapter 4 looks into known (partial) solutions to the problem for which we explain whether they are applicable. The resulting product with its features is shown in chapter 5. In chapter 6 and 7 the actual design and implementation of the final product are described. And in the same chapter some interesting parts of the implementation are highlighted. Chapter 8 reflects on the software development process. Finally, in chapter 9 we reflect on how the problem was solved and we do some recommendations to extend the project even further.

2 Problem Definition

Currently when an automated behavioural mobile testing for non-developers is needed, there are no good tools that facilitate this. Some companies even started to rely on manual testing. To automate mobile testing while still covering multiple operating systems and devices a testing framework is required.

Since the testing should be usable by non-developers it should not require programming knowledge. In addition, when the structure of the app on multiple operating systems are mostly the same, tests should be easily reusable between systems. Only when the structure is different multiple tests should be needed.

To support multiple operating systems and devices the framework should be extendable for new operating systems. Since apps are frequently released and tested, automated testing for multiple platforms and devices should not be slower than manual testing.

The current checklist that bunq uses to verify correct behaviour after an update can be found in appendix B. See appendix A as well for the original project description.

3 Requirements

This chapter describes the functional and non-functional requirements of the project.

3.1. Functional requirements

The functional requirements are listed according to the MoSCoW model.

3.1.1. Must have

1. The user shall be able to specify tests by creating sequences of test instructions.
2. The system shall be able to perform user interface actions on mobile apps.
3. The system shall be able to verify the user interface state of a mobile app.
4. The system shall test behaviour (black-box testing), not implementation (white-box testing).
5. The system shall be usable for non-developers, which means that the tests should be readable for and writable by non-developers.
6. The user shall be able to test apps on the Android operating system.
7. Tests shall be repeatable and deterministic.
8. The system shall be extensible to support multiple platforms.
9. The system shall not require changes to existing apps.

3.1.2. Should have

1. The user shall be able to write test instructions in a browser application.
2. The current manual testing checklist (see appendix B) items shall be testable by the system.
3. The user shall be able to execute tests on multiple devices concurrently.
4. The system shall autocomplete test instruction input fields in the browser application.
5. The user shall be able to reuse sequences of test instructions in multiple tests.
6. The user shall be able to run a subset of the written tests.

7. The user shall be able to add comments to tests.

3.1.3. Could have

1. The user shall be able to test apps on the iOS operating system.
2. The user shall be able to involve multiple devices in one test.
3. The system shall support multiple users adding tests and executing them at the same time.
4. The system shall support running tests from the command line.

3.1.4. Won't have

1. The system shall support verifying behaviour in external (non-mobile) applications, for example: the back-end the app under test communicates with.
2. The user shall be able to record user interface actions and convert those to tests.
3. The user shall be able to select user interface elements for tests using a visual representation.

3.2. Non-functional requirements

The following section will describe and explain the non-functional requirements of the project.

1. The code shall be tracked with Git.
2. The provided GitLab server shall be used to host the project.
3. The provided Jenkins server shall be used for continuous integration.
4. Cloud-based services shall not be used for this project.
5. A pull based development model shall be used, where a feature shall be developed in a feature branch and a merge request shall be made when the feature is completed.
6. Merge requests will be build and tested by the CI server.
7. The back-end shall be written in Java.
8. The front-end shall be written in JavaScript using the React library.
9. The Android implementation shall be written in Java.
10. The iOS implementation shall be written in Swift.
11. Gradle shall be used as build tool for both the back-end and the Android implementation.
12. The project shall be tested with at least 80% coverage.
13. The project shall be completed in 11 weeks.
14. A working sample shall be produced each week.

3.2.1. Motivation

bunq prefers PHP as a language for the project, but we will need to write parts in Java and Objective-C or Swift for the Android and iOS integration respectively. bunq allowed us to choose Java, because we are more experienced with it, and it eliminates the need to introduce another language. Besides, the whole project will be maintained by Android and iOS developers when it is completed, and they do not use PHP anyway.

During our research, Java was the only supported language on the Android platform, so we had no choice but to use it. It is possible to use other JVM compatible languages on Android, but not only subtle incompatibilities may occur, they are not supported officially, and bunq would preferably not depend on a third party for support. Besides the apps are written in Java anyway, so that's what bunq is already familiar with. However, during the development phase of the project, Kotlin became an officially supported language on the Android platform. We both like the language and have experience with it, and bunq was enthousiastic about it as well. Unfortunately, this announcement came too late, as we had already written quite some code. We would probably have chosen Kotlin for both the main part, and the Android integration if the announcement had come earlier. bunq even allowed us to rewrite the project in Kotlin (which is not that work-intensive, because it's possible to convert most Java code automatically) but we decided against it because it would still consume too much of our time. Especially since the bachelor project timeframe isn't that long, this time was better spent elsewhere. As for iOS, the options are Swift and Objective-C, but seeing that Swift is meant to succeed Objective-C, even though the apps are written in Objective-C, we chose Swift.

Git will be used as version control, to track changes to the software. Git was the logical choice, because we have experience with it, and bunq provided us with a GitLab server. We will also use a pull based development model, where one team member will push their changes to a branch and make a pull request, which the other team member has to verify and approve. The merge requests will be tested by a CI server that builds the code and runs the test code.

A build tool like Gradle[14] or Maven[16] is necessary for reproducing builds. We chose Maven for the project because we were familiar with it, and it's a proven tool. For Android, we needed to use Gradle because only Gradle is supported there.

The project should also be tested. We aim for 80% coverage, because more test coverage doesn't necessarily mean that the test suite is better[29].

The timeframe for the project is 11 weeks, as that is the length of the Bachelor Project.

We will apply the agile software development methodology to this project, which means that a working sample will be produced each week, and we plan for each week in order to be able to react to changes and setbacks or breakthroughs. bunq requires a status update about what we did each day to keep track of our progress. Every few days we will have a meeting with the project supervisor to discuss progress and functionality, and the other stakeholders, for example the test owner, are updated when relevant. This is possible because we will be in the office every day. For our TU Delft coach, we will have a meeting every two weeks, unless problems occur, and send as much as possible beforehand to review.

Since bunq is a bank, and everything is under strict non-disclosure agreements, we cannot use cloud services for our project files.

4

Analysis of Existing Tools and Selection of the Solution

We analysed various existing solutions to the problem of automated UI testing. In the following section we will outline a few full solutions and why those do not satisfy our requirements, and some solutions to parts of the problem, and if those are suitable for the project.

4.1. Full testing solutions

This section focuses on solution that will either solve the entire problem or parts of it.

4.1.1. Espresso

Espresso[11] is Google's own white box Android UI testing framework. Tests are written in Java, and compiled in the Android package, and it uses JUnit to run those tests. Espresso does not cover the requirement of being cross platform, and the Java code isn't easy to use for non-developers. It's also white box testing, because it's integrated in a specific app, which makes it not usable for general app testing.

4.1.2. EarlGrey

EarlGrey[10] can be defined as 'Espresso for iOS'. It is also made by Google, it uses Apple's XCTest and XCUITest as testing framework. It has the same shortcomings as Espresso, tests are defined in either Objective-C or Swift, it is not cross platform, and it's also white box testing.

4.1.3. Calabash

Calabash[7] is a cross platform UI testing framework written in Ruby that uses Cucumber[9] to specify tests. This came the closest to what we needed, but it is no longer under active development[8]. Furthermore, bunq prefers working with a language stack they already have experience with and the architectural choices regarding the Android and iOS modules in Calabash aren't up to date, so it would probably need a large scale rewrite, at which point it is more productive to write a new solution to reevaluate various architectural choices, and maintaining code we don't need and use is not desirable.

4.1.4. Appium

Appium[6] is another cross platform framework for UI testing. It uses the WebDriver[26] specification for specifying tests for both Android and iOS. While the cross platform functionality does cover our requirements, WebDriver is designed for testing web applications, and Appium will therefore have to adapt to the protocol, instead of adapting the protocol to the use case. To implement Appium tests, it's necessary to use a WebDriver implementation in a programming language, which violates the requirement of being easy to use for non-developers. It may also require major refactoring in the tests if the apps are refactored, as it's not fully black box testing because it requires to specify parts of the implementation of the apps.

4.2. Partial testing solutions

This sections shows frameworks that can be used to test Android or iOS. These frameworks will not result in a standalone solution and need implementation.

4.2.1. UI Automation

UI Automation was Apple's UI automation framework. It can be used to automate UI interactions in iOS apps. It is now deprecated in favor of XCUITest (see below), so it is not relevant for our solution anymore.

4.2.2. XCUITest

XCUITest[28] is the successor of UI Automation. It is the only way to automate UI interactions in iOS apps, and every full solution mentioned above uses it one way or another. We will need XCUITest to satisfy the requirement of being able to run the tests on on the iOS apps, so there is no choice here.

4.2.3. UI Automator

UI Automator[25] is Android's equivalent of XCUITest. It can be used to automate UI interactions in Android apps. Although it would be possible to use the internal Android API's for sending events to apps, UI Automator conveniently wraps those API's for us, especially because they may be different between Android versions.

4.3. Test language

To test, you have to define tests. These tests should be defined in language writable en readable for non-developers. In this section we show the possible solutions.

4.3.1. Define a custom domain specific language

It is quite easy these days to define a new domain specific language with tools like ANTLR[5]. It provides a generator for a lexer and a parser for the defined grammar. The parser produces an Abstract Syntax Tree, and ANTLR provides tools to interpret this AST based on the visitor pattern. Defining your own language is very flexible, but costs also a lot of time to implement and maintain.

4.3.2. Cucumber

Cucumber[9] is a framework that uses Gherkin[13] to define tests in a natural language. This allows describing the behaviour of tests without detailing how that behaviour is implemented. Gherkin unifies the tests with the test documentation, because the tests are written in a natural language, which means there is no need to keep the test scenarios documented separately.

4.4. Chosen solutions

We have chosen to base our framework on UI Automator and XCUITest.

Cucumber was considered because it will make it possible to define tests in a natural language. This fits our requirement of making tests easily readable and writable by non-developers. It also makes converting existing test checklists easier. That it also can be used as documentation of tests is a bonus. Cucumber supports both Java and PHP, so that was not a problem.

However, we decided that we didn't really need a library for defining our tests, because tests will consist of sequences of steps to execute. While Cucumber does use sequences of steps for testing, a lot of the functionality of Cucumber consists of running these step files as JUnit tests for a specific project, and that's not necessary for this project. Implementing the step matching ourselves won't be more work than adapting Cucumber for our use case.

UI Automator and XCUITest will make it possible to interact with the testable app on a high level without changing or adding any code to the testable app. These are also the least likely components that will be changed, since they are part of their respective platform toolkit. Our hands are more or less forced here, because there are no alternatives.

5 Product Features

This chapter shows what the user can do with the product. The description is given from the front-end, because that is what the user will see and use. The front-end is high performant Single Page Application (SPA) and feels like a native application. Also local changes can be saved or discarded. The descriptive text is supported by the screenshots in appendix H.

5.1. Structure

As seen in figure D.4 a tests consists of a feature. The feature contains scenarios, and the scenarios contain steps. When the user first starts the program a category has to be created. Categories are used to organise all the features. Figure H.1 shows how this looks from a user perspective.

After the user has created at least one category the user can create new features in each category, or navigate to existing features as shown in figure H.2. Also features can be moved to other categories by using the select dropdown.

5.2. Tests

After you click on a feature (or click the button create feature) you can edit the feature as shown in figure H.3. The following things can be done on the feature page:

- Create and remove scenarios
- Create and remove steps
- Add and remove routines
- Duplicate a scenario
- Drag and drop the order of steps
- Edit the title of feature, the feature context, and scenario titles
- Remove the feature

5.3. Autocomplete

To provide the tester with valuable information about what steps and arguments are available an auto-complete is provided. This auto complete detects what steps and what arguments are available, see figure H.4 and H.5. The possible arguments are provided by the keyswords of the mappers defined in the configuration.

5.4. Config

Since the steps can use keywords for `Apps`, `Screens`, and `Elements` that are unknown for the devices. It is possible to create mappers on the config page as shown in figure H.6. These mappers will replace keywords with identifiers and can be configured for Android. It is written with extensibility to iOS in mind. Ideally the tests can run on iOS by only adding iOS identifiers.

5.5. Runner

After the tests are created, it is possible to run the tests on connected devices. Figure H.7 shows the runner. The features of runner are:

- The phone is automatically detected when plugged in by USB.
- Select device
- Select features by category or individual features
- Run multiple test instances asynchronously (on different devices)
- See progress by percentage of steps ran

When the tests are running it looks like figure H.8. Tests can be canceled and live progress can be shown by clicking on a feature. After the feature is done running you can view the results as shown in figure H.9. Results are displayed per individual step. The use of a routine is also indicated.

If the user wants to know more about a certain step, he can click on the step. Then the user will see something like H.10. The step result includes:

- Result: successful, error or skipped
- Arguments parsed from the action
- Message given by the device
- Screenshot of state after the step
- Duration in seconds of the step

When a tests fails the user will see red headers and a red step as shown in figure H.11. The steps after the failed step are skipped, since the state of the device is now unknown.

5.6. User flow

The designers of bunq like to have insights in the user flows. Since the features are used to test the user flows, the screenshots that are retrieved from the tests can be used to document the user flows. After a feature is done running a button 'Show user flow' becomes available. When the user selects the button he can navigate through the user flow as shown in figure H.12. To make navigation easier the previous and next actions also listen to the scroll-wheel of the mouse.

5.7. Routines

Routines are available to help the user with reusable parts of scenarios. If the user want to reuse steps he can define a routine. Routines are created (and removed) on the page shown in H.13. After creating or clicking on an existing routine the user can edit the routine as if it it an scenario. This is shown in figure H.14.

6

Architecture of the System

This chapter gives insight into the architecture of the system. The product is divided into 3 projects: the front-end, the back-end, and the device implementation. In the sections bellow the projects are discussed.

6.1. Front-end

The front-end is responsible for creating the tests, instructing the server to run the tests and to show the tests results. For creating a Single Page Application (SPA) for the web we chose Facebook's React[17] framework for it's excellent scaling and use of the shadow DOM. Figure 6.1 shows the design of the front-end. This design is based on the Flux[12] architecture.

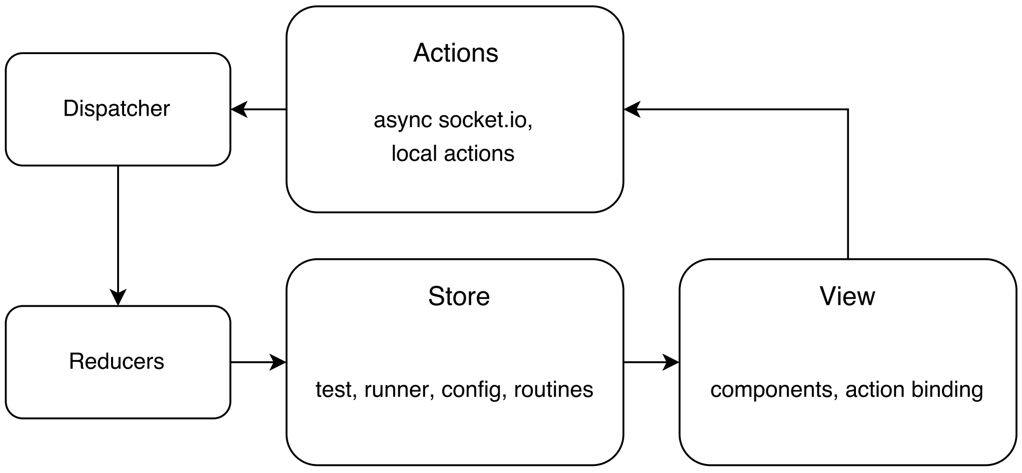


Figure 6.1: Design of the front-end

For the connections with de backend server we chose socket.io[21] for it's performance, reliability and implementation in multiple programming languages.

6.2. Back-end

The back-end is responsible for interaction with the database, the front-end, and the devices. With the main purpose of running and executing the tests. The back-end contains 4 modules: webserver, (test) runner, (test) executor, and device connector. See figure 6.2 for a dependency graph. The back-end uses the Spring Framework[23] for Dependency Injection and Spring Boot[24] to start the application.

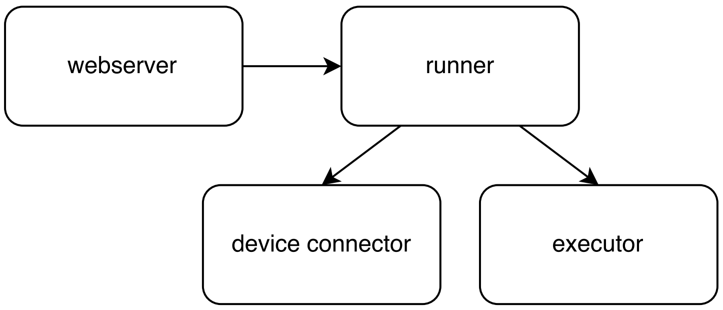


Figure 6.2: Dependency graph of back-end modules

6.2.1. Test runner

The test runner receives instructions from the webserver on what tests to run. The test runner works with observable streams from RxJava[20] and keeps track of running tests and results. Only the executions of the steps are delegated to the test executor.

The test runner is based on the Spring Framework[23] architecture with controllers, services, and repositories. The controllers are the runner components and webserver. The controllers use the services to make use of repositories without having to interact with the database themselves. An overview of this architecture is found in figure 6.3.

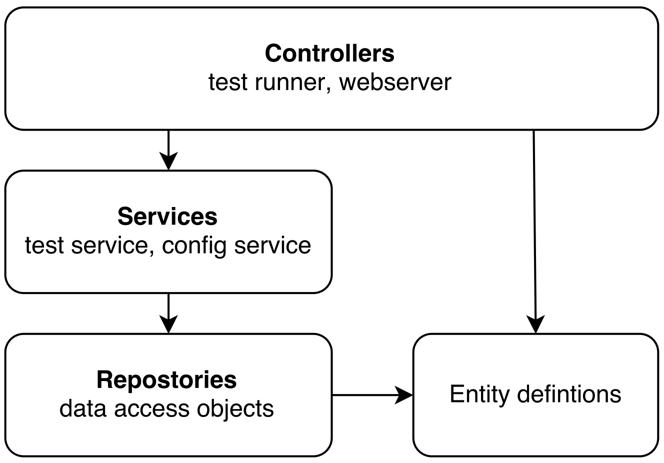


Figure 6.3: Architecture of test runner

6.2.2. Webserver

The webserver handles socket.io[21] events and is the only access point for the front-end. To interact with the persistence layer the webserver uses the services from the test runner. Also the payloads used for the socket.io events are defined here. Spring Boot[24] is used to start the webserver and all of its dependencies.

6.2.3. Test executor

The test executor receives the individual steps from the test runner. The test runner also instructs the test executor when to start and stop a scenario. The steps it receives look like: `Given I start the <app> app` or `When I type <text>`. The parts between <> are wildcards, i.e. they can be anything. Those wildcards can also be assigned a preconfigured value that will be replaced with the specific value. These specific values can even be configured per supported platform. For example: in `Given I start the bunq app`, `bunq` may be mapped to `com.bunq.android` on Android. This allows abstracting the actual identifiers away, to make tests more user friendly and readable. After this the executor looks up the right step and tries to replace possible mappings. The result is send to the predefined API service (with device url included). Finally the result is processed into a step response and given back to the test runner. The common data flow through this module is shown in figure 6.4

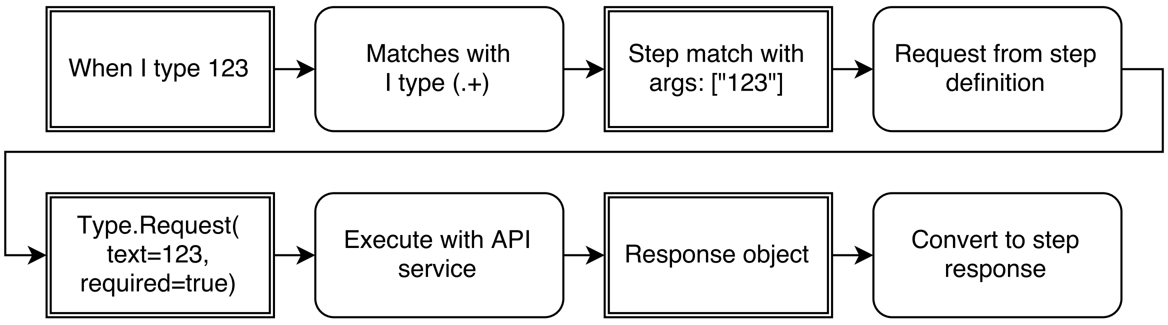


Figure 6.4: Data flow in text executor

6.2.4. Device connection

The device connection module consists of an interface that represents a device, and an interface that represents a connection to a platform. The platform connection interface is then used to get all available devices for that platform, and the device interface to get information from devices and to start and stop the test executor on the device. RxJava is used to provide a constant stream of currently available devices. This allows easy extension for other platforms like iOS, because it's only necessary to implement two interfaces. See figure 6.5 for the class diagram with the Android implementations.

Android

For Android we used the Android Debug Bridge[2] (ADB) to communicate with the devices. Every method in the device interface is implemented using ADB commands. Those ADB commands are sent using the `adb` command line tool, even though it would have been possible to implement the

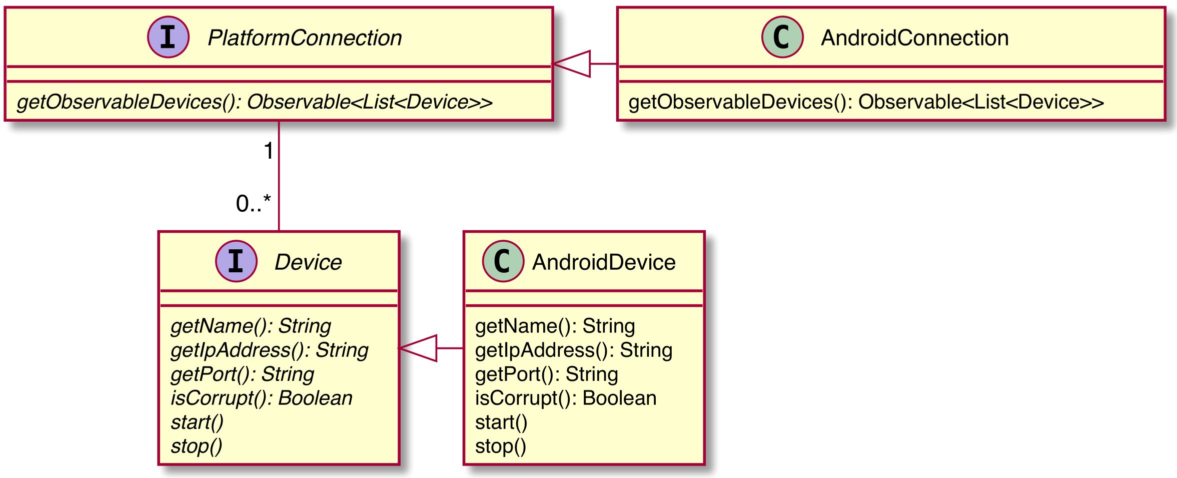


Figure 6.5: Platform and device connection class diagram

ADB protocol in Java. This decision was made because the ADB protocol isn't documented, and not guaranteed stable either. Unfortunately, this requires the Android Platform Tools to be installed on the server where the back-end runs, and executing an external process.

6.3. Platform specific design

This section will discuss the design of sending instructions to a device. Because this was only implemented for the Android platform, it will also discuss the Android specific design choices.

6.3.1. General

For communication with the devices, a simple HTTP API, with the server on the device and the client in the backend, is used. The HTTP server on the devices listens on port 9229, simply because that number is not assigned by IANA[15]. Executing an instruction consists of making a POST call to the path `/instruction/<name of instruction>`, for example: `/instruction/start_app`, with an instruction specific JSON object containing required parameters. A general JSON object with the result of the instruction will then be returned. See appendix E for details.

6.3.2. Android

The architecture of the application on Android is rather simple. It consists of three parts: an HTTP API endpoint, a foreground service, an instrumentation test entry point and a UI Automator worker thread that executes the test instructions. How these parts work together is best explained with the sequence diagram in figure 6.6. In the following sections, each part will be elaborated upon.

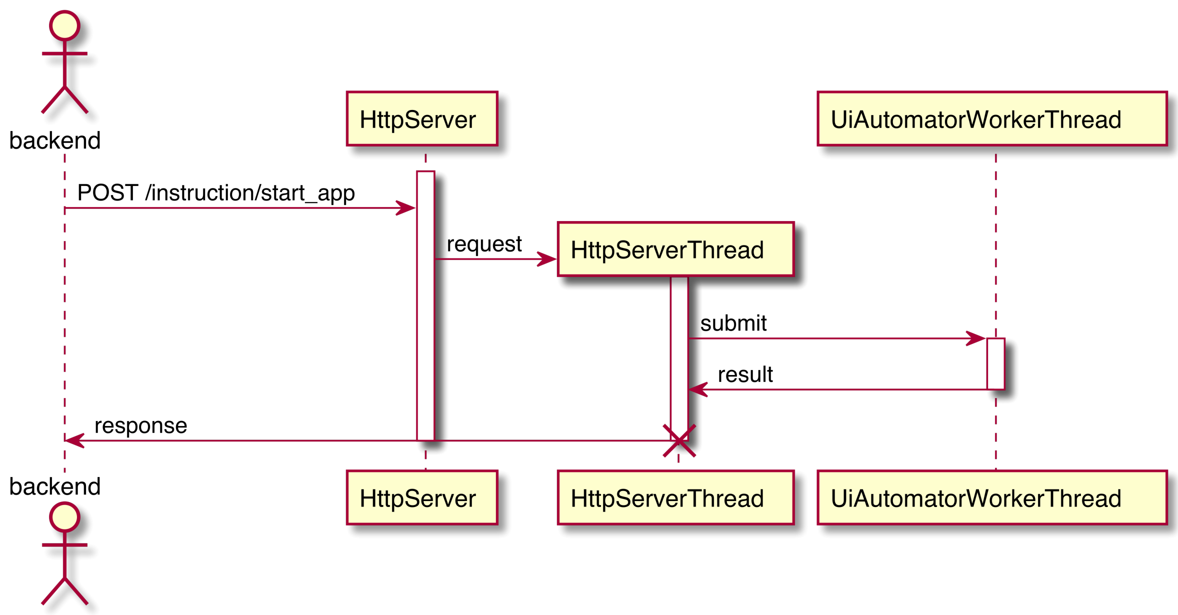


Figure 6.6: Device sequence diagram

HTTP server

To execute an instruction, the backend will send an HTTP POST request with a JSON object (as explained in section 6.3.1) to the HTTP server. The HTTP server will then start a separate thread to handle this response, in which it will schedule a job on the UI Automator worker thread, and wait for a result. This keeps the connection open, until the UI Automator worker thread returns a result, which will then be send back to the backend as response.

Foreground service

The Android threading is a bit different from the standard Java threading, because it is a mobile platform with performance and battery life constraints. On an Android phone, background threads can be killed at any given moment by the Out of Memory (OOM) killer[3], and that is not desirable for the HTTP server. However, by binding the service containing the thread to a notification, the OOM killer will only kill it as a last ditch effort. This technique is used to keep the HTTP server alive.

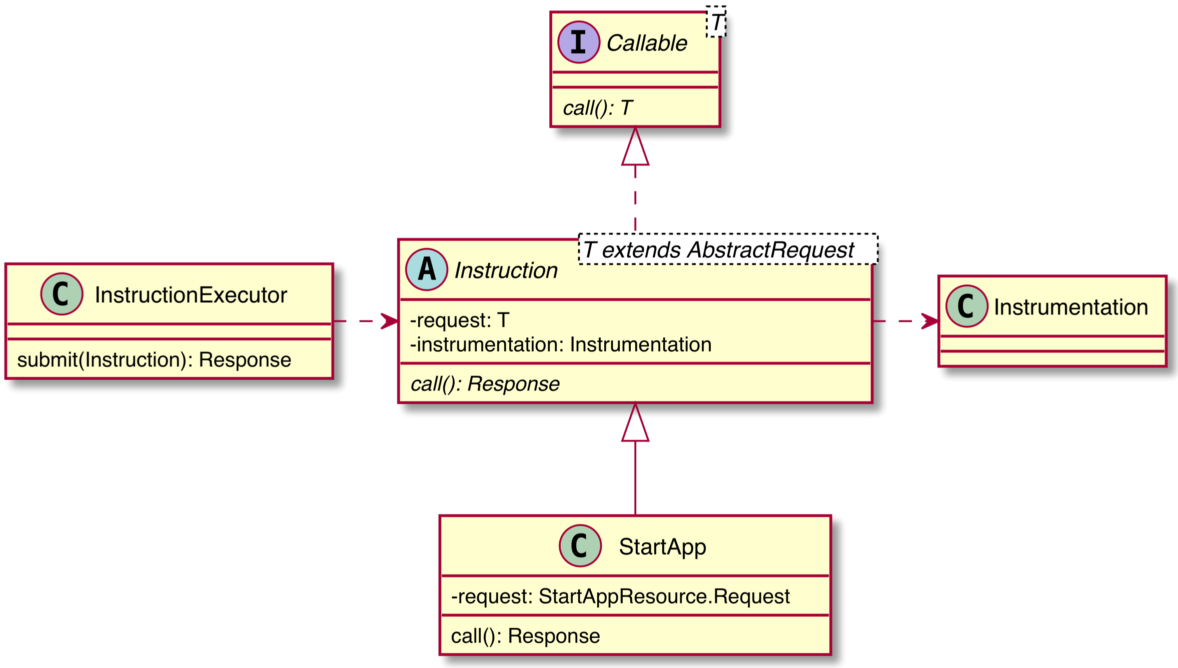
Instrumentation test entry point

Android apps can have two test sets, one standard unit test set and an instrumentation test set[4]. Standard unit tests are run on the developer PC (or CI server), while instrumentation tests are run on the Android device itself. The use case for these tests is testing everything that doesn't touch the Android API with standard unit tests, and testing the Android interactions with instrumentation tests.

The UI Automator library can only be used in the context of an instrumentation test, it cannot be used directly in the main app code. This means that we must provide an entry point for an instrumentation test, that does not test anything, but starts a worker thread with a reference to the instrumentation instead.

UI Automator worker thread

The worker thread is the part that actually executes test instructions. It keeps a reference of the instrumentation and passes it to every instruction class. These instruction classes then actually call the relevant Android API methods to perform actions on the device. See the class diagram in figure 6.7 to see how instructions are designed. The InstructionExecutor contains the actual worker thread.



Callable
T
call(): T
Instruction
T extends AbstractRequest
-request: T
-instrumentation: Instrumentation
call(): Response
InstructionExecutor
submit(Instruction): Response
Instrumentation
StartApp
-request: StartAppResource.Request
call(): Response

Figure 6.7: Instruction class diagram

6.4. Adjustments to initial solution

In this section we describe the major adjustments we did to the initial solution in appendix D. The front-end and back-end are discussed, because of the amount of changes and movement of responsibilities. We also highlight why we implemented the device specific parts for Android only.

6.4.1. Back-end

The biggest part that changed from the initial solution is the design of the main application. This design evolved over time and was adjusted to our needs. Figure D.2 shows the initial design of the main application and 6.2 shows how the back-end looks now (also note the renaming). While modules are still based on their responsibility some things have changes:

- Core is renamed to runner for clarity
- Monitoring is now done in the runner with the use of an observable
- Test creation is moved to the front-end
- The runner does the instruction (executor) instead of the front-end
- The front-end communicates only with the webserver module

6.4.2. Front-end

While the front-end was not defined in depth in the initial solution the front-end gained some responsibilities. Beside test creation the front-end contains almost all code for monitoring the test results.

A lot of business logic takes place in the front-end since it became an SPA. A local copy of the back-end data is kept and updated while the user changes the tests.

6.4.3. Device specific implementation

In our initial solution, we assumed that we were going to implement the device specific parts for both Android and iOS. However, we realised after one week that this would take a lot of time, since we did not have any experience with developing on iOS. Besides, bunq indicated that they would rather have one complete implementation than two half working implementations. In the end we decided to only implement the Android part, and leave a recommendation for the technologies and architecture to use for iOS. We still superficially checked for every feature we implemented whether it was possible to also implement it on iOS.

6.4.4. User flow

In week 6 a meeting was requested by two bunq employees about possible documentation of the apps user flows. These are paths the users have to take in the app to achieve what they want to do. For example: signing up or requesting money. Since these are also actions that need to be tested the question was dropped if it was possible to use the test definitions to document user flows.

Since we already made screenshots on failing tests, it was quite easy to implement screenshots on every step. The test are defined by behaviour and thereby self-documenting. The documentation can also be used to document the user flow illustrated by screenshots. For now tests results output screenshots and give insight into to user flow, but this would be very useful for documentation in the future.

7 Implementation

In this the chapter we discuss some elements that don't have an obvious implementation or might stand out otherwise.

7.1. Step autocompletion

To improve the user experience when writing tests we introduced an autocomplete for test steps. The autocomplete is generated from the original step definition. Since steps are defined by using regular expressions and mappers the autocomplete should be generated from those. We start with the steps defined in figure 7.1.

```
@Step("I type (.+)")
public Type.Request type(String text) {
    return new Type.Request(text);
}
```

```
@Step(value = "I start the (.+) app", mappers = {Mapper.APPS})
public StartApp.Request startApp(String packageName) {
    return new StartApp.Request(packageName);
}
```

Figure 7.1: Step definition in Java

7.1.1. Step detection

To detect if a user is currently typing a step we generate a new regular expression. First we make the matching groups of the regular expression lazy so the given arguments are captures as small as possible. This can be done by adding a '?' behind all quantifiers in the original regular expression, see figure 7.2.

```
I type (.+?)
```

```
I start the (.+?) app
```

Figure 7.2: Result of creating lazy capturing groups

While typing all the characters at the end are optional. Starting with the last character we can wrap every character and the characters wrapped behind that in a non-capturing optional groups. The original groups are treated as characters in this case. This will result in regular expressions that will detect every possible typing position, see figure 7.3.

```
^(?:I(?: (?:t(?:y(?:p(?:e(?: (?:(.+?))?)?)?)?)?)?)?)?$
```

```
^(?:I(?: (?:s(?:t(?:a(?:r(?:t(?: (?:t(?:h(?:e(?: (?:(.+?)(?: (?:a(?:p(?:p)?)?)?)?)?)?)?)?)?)
    ?)?)?)?)?)?)?)?$
```

Figure 7.3: Regular expressions for matching every typing position

The matching regular expressions from figure 7.3 can be created by using the code in figure 7.4.

```
lazyMatchGroups.split(/(\(.*?\)|.)/g)
            .filter(part => part !== ””)
            .reduceRight((acc, item) =>  ”(?:” + item + acc + ”)?”, ””);
```

Figure 7.4: Creating regular expressions for matching every typing position

7.1.2. Step completion

If a step matched smart autocomplete suggestions should be given. Building forward on the regular expressions from figure 7.3 we can retrieve the arguments from the capture groups. And by splitting the original regular expressions from figure 7.1 on the capture groups we can retrieve the static parts. The result of typing ”I start the bu” is shown in figure 7.5.

These results can now be used to show the user the best matches. Also the arguments can be mapped by the mapper specified in the step definition for a more targeted autocomplete. If an argument is undefined we can choose to show a placeholder (for example: three dots) and autocomplete until the first unknown argument.

7.2. Reactive test runner

The test runner is responsible for instructing the test executer in the right order and to provide the client with observable results. For this RxJava[20], an implementation of ReactiveX[18], is used. This enables us to program asynchronous with observable streams. Figure 7.6 shows the design of the runner.

Every entity is responsible for generating it’s template so the client knows what tests results to expect. After that the client can subscribe to the observable step results and place them in the template when

```
arguments = [”bu”]
static = [”I start the ”, ” app”]
```

Figure 7.5: Result of typing a step

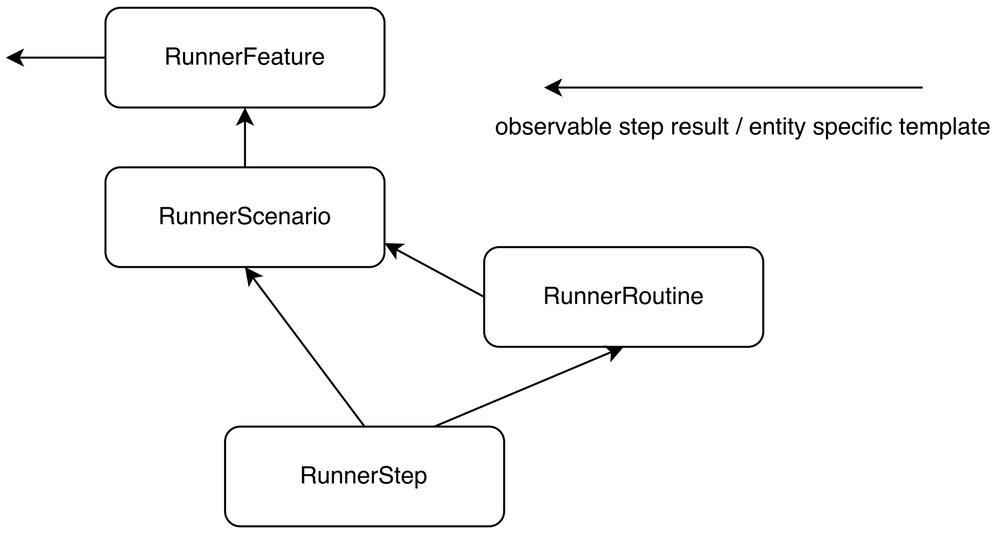


Figure 7.6: Entities of the runner

received. The step result indicates in what template it belongs. A typical `run` method for an entity of the runner now looks like figure 7.7. The method `run` in scenario will combine the observable results of its routines and steps and then sort them by their index so they are executed in the right order.

```
Observable<StepResult> run() {
    return Observable.concat(Observable.fromIterable(runnerSteps), Observable.fromIterable(
        runnerRoutines))
            .sorted(Comparator.comparingInt(RunnerScenarioElement::getIndex))
            .flatMap(RunnerScenarioElement::run);
}
```

Figure 7.7: Run method of the scenario runner

7.3. Android platform connection

The Android platform connection implements the interfaces as described in figure 6.5. These classes then use ADB commands to get the required information. For every ADB command needed, a class exists that defines the ADB command and a parse function to parse its output.

For example: to get a list of devices we use the `adb devices` command. This command will return the list of devices currently connected as shown in figure 7.8.

As stated in section 6.2.4, we used the `adb` command line tool instead of implementing the protocol in Java. For this, we needed to be able to execute external commands from our application. We solved it by starting an external process from the Java application and supplied it with the right param-

```
$ adb devices
List of devices attached
00abf49d65d229e8            device
```

Figure 7.8: `adb devices` output

eters. These parameters were static with the exception of commands targeting a specific device, which needed an identifier for the device. To solve this, we differentiated between platform wide commands (like get all devices), and device specific commands (like get name), and passed the device serial to device specific commands.

Another problem we encountered was that some ADB commands need to be run in the background, while for others the output was relevant so they needed to wait and parse the output. We solved this by adding a boolean that specified whether the executor should wait for the process to terminate or not.

8 Software Development Process

8.1. Methodology

Our software development methodology was agile, as we made a plan at the start of every week. We worked full time in the office, which allowed us to write down everything we wanted to do for the week on a whiteboard, and cross items off when they were finished. Because we only planned for one week in advance, we were able to react to unexpected changes, although nothing major happened. We communicated every few days with our project supervisor about the status and progress of the project. After we had a minimum viable product, we started asking the test owner at bunq for feedback. We clarified and documented his issues and added requested features.

8.2. Testing

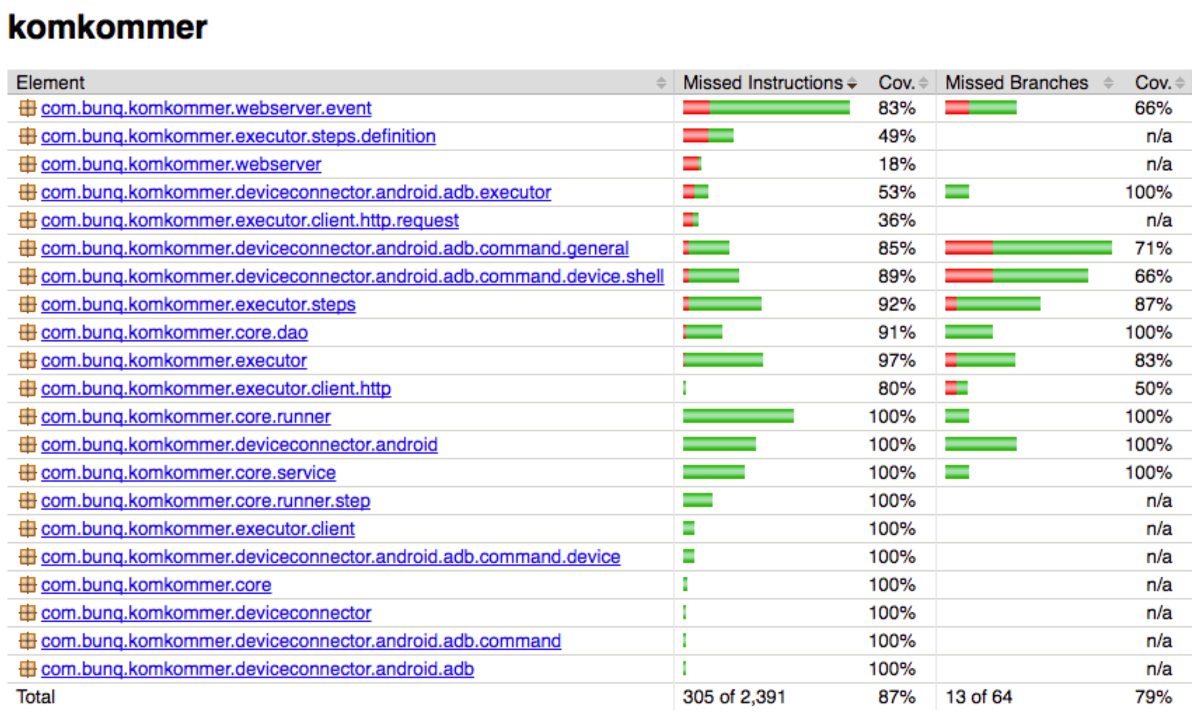
8.2.1. Back-end

The back-end is tested with unit tests using JUnit. See figure 8.1 for the coverage report. The `com.bunq.komkommer.executor.client.rest.instruction` package coverage is low because it contains a lot of holder classes which all have a private constructor to prevent initialisation. The coverage report does not take into account that these do not need to be tested.

We used Mockito to inject mocks of dependencies to test specific classes. Because we used Spring for dependency injection, it was relatively easy to inject different dependencies during testing.

8.2.2. Android

Testing the Android app posed more of a challenge, because how do you test something that tests something? The Android application does nothing else but transforming HTTP requests in UI Automator calls, so unit testing this did not seem useful. In addition, since the app touches a lot of Android APIs,



komkommer

Element	Missed Instructions	Cov.	Missed Branches	Cov.
com.bunq.komkommer.webserver.event		83%		66%
com.bunq.komkommer.executor.steps.definition		49%		n/a
com.bunq.komkommer.webserver		18%		n/a
com.bunq.komkommer.deviceconnector.android.adb.executor		53%		100%
com.bunq.komkommer.executor.client.http.request		36%		n/a
com.bunq.komkommer.deviceconnector.android.adb.command.general		85%		71%
com.bunq.komkommer.deviceconnector.android.adb.command.device.shell		89%		66%
com.bunq.komkommer.executor.steps		92%		87%
com.bunq.komkommer.core.dao		91%		100%
com.bunq.komkommer.executor		97%		83%
com.bunq.komkommer.executor.client.http		80%		50%
com.bunq.komkommer.core.runner		100%		100%
com.bunq.komkommer.deviceconnector.android		100%		100%
com.bunq.komkommer.core.service		100%		100%
com.bunq.komkommer.core.runner.step		100%		n/a
com.bunq.komkommer.executor.client		100%		n/a
com.bunq.komkommer.deviceconnector.android.adb.command.device		100%		n/a
com.bunq.komkommer.core		100%		n/a
com.bunq.komkommer.deviceconnector		100%		n/a
com.bunq.komkommer.deviceconnector.android.adb.command		100%		n/a
com.bunq.komkommer.deviceconnector.android.adb		100%		n/a
Total	305 of 2,391	87%	13 of 64	79%

Figure 8.1: Coverage report for the back-end

testing was only possible using instrumented tests (see section 6.3.2).

Ultimately, we decided to test the Android app with a sample app. This sample app is a simple app that has reproducible actions and reactions; for example: display some text if a certain button is pressed. In our tests, which are run on the device, we start this sample app and make an HTTP request to localhost. Since the actions and reactions were rather specific, we could test for failing behaviour as well. You can still think of these tests as unit tests, because they test a functional unit of the application, even though the unit consists of an HTTP request and an executed UI Automator procedure, which is more than one unit with regards to the code. See figure 8.2 for the coverage report.

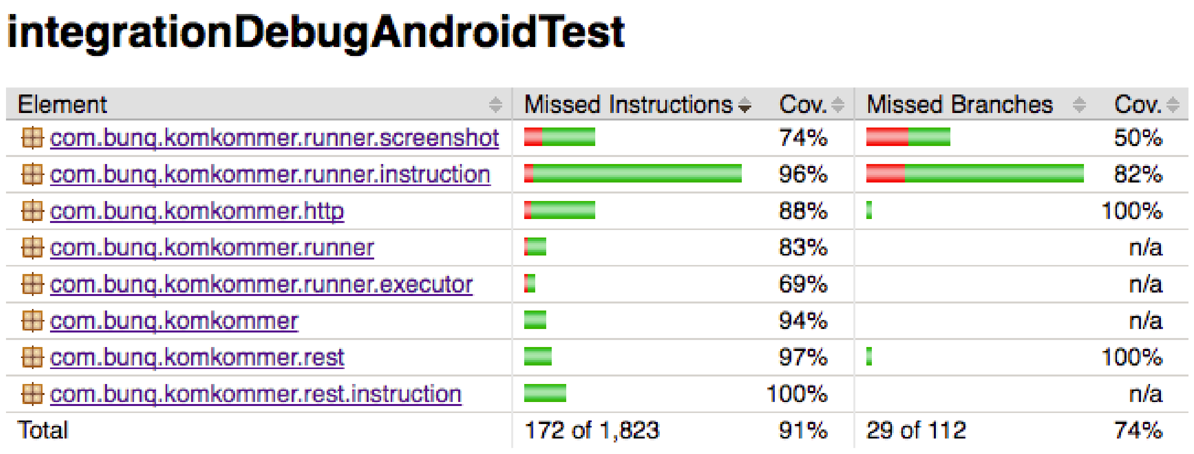
8.3. Tools

8.3.1. Build tool

During the research phase, we decided to use Maven as our build tool. However, seeing that we had to use Gradle anyway for Android, we realised using one build tool for both projects would be easier to maintain, so we decided to use Gradle for the back-end as well.

8.3.2. Version control

Following the non-functional requirements, we used Git to manage changes to the product. We were provided with a GitLab instance by bunq. For every feature, we created a branch and made a merge



integrationDebugAndroidTest

Element	Missed Instructions	Cov.	Missed Branches	Cov.
com.bunq.komkommer.runner.screenshot		74%		50%
com.bunq.komkommer.runner.instruction		96%		82%
com.bunq.komkommer.http		88%		100%
com.bunq.komkommer.runner		83%		n/a
com.bunq.komkommer.runner.executor		69%		n/a
com.bunq.komkommer		94%		n/a
com.bunq.komkommer.rest		97%		100%
com.bunq.komkommer.rest.instruction		100%		n/a
Total	172 of 1,823	91%	29 of 112	74%

Figure 8.2: Coverage report for the Android app

request when it was ready to be merged. Because we were working full time in the same office, and we are only a group of two, most communication about those merge requests was verbal. If there was a problem with a merge request, one of us would just ask the other for clarification. However any unclarity worth documenting was still put in writing. This worked pretty well, because this way we were both able to understand the whole code base.

8.3.3. Continuous integration

bunq gave us access to a Jenkins[1] server to use for continuous integration (CI). Since we used Gradle as build tool, it was quite easy to set up CI. The CI server would just compile the project and run the tests on every merge request. This saved us from some simple mistakes that could have slipped through during code review otherwise.

8.3.4. Static analysis

We used a local instance of SonarQube[22] to ensure code quality. It showed us potential vulnerabilities, bugs and code smells and gave us an indication of the quality of the code. We also used it to verify test coverage. During the research phase, static analysis was not really considered, but we decided to use it during the project after experimenting with it, because we realised using it would improve the quality of the code.

8.4. Software Improvement Group feedback

During the project, the code was submitted to the Software Improvement Group (SIG). SIG evaluated the code and provided us with feedback, which can be found in appendix C.

Since we value maintainability for this code base, especially because it is not going to be maintained by us, but by someone else, we were not disappointed with the score of 4 out of 5. However, we of course wanted to achieve 5 out of 5, we refactored the code according to the feedback.

For the Unit Size part, those problems only really existed in the Android application, so we split up the long methods there. This was not hard and did not take much time, maybe an hour including testing and verifying.

Fixing the Unit Complexity was a little bit harder, but it still only took a day. When we inspected the part of the code ourselves, we realised that in some places, we connected the data persistence layer directly to the controller layer, without services in-between, like figure 6.3. To fix this we introduced a service layer that bundled the necessary parameters, which reduced unit complexity significantly.

9 Conclusion and Recommendation

In this chapter we conclude if we reached the specified goals or not. Recommendations for extending the product are given.

9.1. Conclusion

This section shows the reflection on the goals. Mainly the functional requirements are highlighted, but also the non-functional requirements are reflected upon.

9.1.1. Functional requirements

All of the must haves and should haves are implemented in the final product. In chapter 5, 6, and 7, details about the actual implementation can be found.

Automatically testing the mobile app has been achieved. After the tests have been specified by creating a sequence of simple steps, the user can run the tests by connecting a phone and pressing a button. Most app functionality can be tested, with the exception of scrolling and some non-deterministic behaviour (for the non-deterministic behaviour in the bunq apps a temporary solution was added), for example: dialogs sometimes, depending on external factors, interrupting a step.

If there are any instructions that the framework is not supporting it is very easy to add new instructions to the existing implementation. To add an instruction to the framework, one class and one method have to be implemented in the back-end, and two classes in the Android implementation.

The framework is universal, every app can be tested using it and does not require changes to existing applications.

Tests are defined by either specifying what kind of action should be performed, or verifying what should be displayed on the screen.

The testing framework is easily usable by non-developers. Available steps are autocompleted, which makes it easier to learn how to use the framework and what kind of instructions are supported. Mapping the identifiers as described in section 6.2.3 will probably require reading some (layout) code, or at least some input from the app developers. However, these identifiers won't change often and will for the most part only need to be configured once.

Currently the testing framework only supports Android, but it is written with extensibility in mind. Other platforms like iOS and Windows Phone can be easily integrated by implementing the interfaces in the platform connector and by adding extra columns in the configuration tables.

Implementation is not tested, only identifiers of elements in the app are used. The tests are purely defined by behaviour and consist of atomic steps that do not carry state.

The existing apps are not modified. Testing only requires an extra app on the phone that provides the endpoint for test instructions.

Regarding the could haves, they can all be implemented without requiring changes to existing parts, except for supporting multi-device tests, because that would require some small changes in the test executor. For simultaneously editing tests, a multi-user module should be added to prevent multiple users from overriding the saved tests.

The command line runner is not implemented, see recommendations.

9.1.2. Improvements

If we have to start the project again with what we learned, we would have made the platform connection module separate from the back-end, and let them communicate with a standardised API. This way, the back-end could have been run in the cloud, while the platform connection module could have been run on so-called 'slaves', which would be separate computers. This would have allowed easier support for running the tests from a continuous integration server as part of a deploy routine. Due to the limited time frame and the fact that we did not realise this during the research phase, it was not implemented.

In general, the project timeframe was rather short. We would have wanted to implement more features, but as time was limited, we had to prioritise. While we did not implement many test instructions (as can be seen in appendix F), we tried to make it as easy as possible to add more instructions.

9.1.3. Ethical implications

There are very few ethical implications as result of this project, however since it is always good to think about the ethical results of a project we list the ethical implications.

The main implication is that the testing framework is automating a lot of manual work and can result in the loss of jobs on the market. Which may result in the need of alternate income models or an other system that results in social stability. This is of course a standard consequence of any kind of automating, and it is not really specific to this product. Bugs in the framework can also cause a false confidence in the quality of mobile app releases with no one directly responsible. Untested app releases will in most cases only lead to a loss of user satisfaction because 'something doesn't work', but can also lead to for example data loss or data leakage. One of the more unexpected results is that it is possible to take over control of someone phone when USB debugging is enabled and the owner of the phone accepts an incoming connection, however these chances are very low (usually, only developers have the USB debugging option enabled, and they probably know about the risks accepting random incoming connections).

9.2. Recommendations

Since the product will be used and extended by bunq, we will list some recommendations for the product in this section.

9.2.1. Unexpected popups

For the bunq app, unexpected popups posed a problem. We thought of the possible solutions to this problem and came up with two solutions. Both solutions work with optional routines, which will be bound to a step. The optional routines can be specified just like the normal routines and should only run if necessary. It is necessary for the user to specify the conditions in the optional routine to keep it from starting by default (for example: `Given I should see an element with Confirm`), which will increase the probability of the user making mistakes.

The first solution would be to implement these optional routines with the platform specific asynchronous facilities, for example UiWatchers in the UI Automator library for Android. The advantage of this is that it's relatively easy to get the optional routines by running instructions asynchronously while waiting for confirmation. More advantages are that it's not hard to get a confirmation that an asynchronous routine has run and an asynchronous routine can even run during another instruction. However, the disadvantage is that it requires calling the HTTP methods on the device itself, which will add quite some code to the device specific implementation, which we kept as small as possible. Another disadvantage is that it requires asynchronous facilities to be available on every future platform for which the product might be used.

In the second solution the back-end is responsible for executing the step and the optional routine simultaneously. If the step fails and optional routine succeeds the step can be ran after the optional routine. The biggest advantage is that the code would only need to be written once, and can then be used on every possible platform. The disadvantage is that it might be more work to implement properly because it would require some rewriting in the test executor code. We suggest this solution.

9.2.2. iOS implementation

For the iOS implementation, we recommend mostly the same structure as the Android implementation. XCUITest can be used instead of UI Automator, and Xcode[27] will be needed to communicate with the devices. As for the HTTP server, RestKit[19] seems a great fit, because it explicitly supports iOS.

9.2.3. Independent device connector

We suggest to create a individual application of device connector module, so the user can run the device connector on his local machine instead of the whole back-end.

9.2.4. Command line runner

To instruct the tests without using the front-end or socket.io, it would be useful to have a command line runner. The way to do this would be to create a module similar to the webserver that makes use of the the Spring Boot Command Line Runner. The Spring application properties can now be reused and passed through the command line. Collecting the test results is done by subscribing on the test observer.

9.2.5. Test reporter

Since tests are done on a weekly basis before the release of a new version of the app we suggest to build a test reporter that uses the command line runner or the socket.io web interface. This test report should run automatically against the latest staging build or triggered manually. The main purpose of the test runner would be to show tests results.

9.2.6. User flows

To fully utilise the user flow functionality we recommend extending the test reporter with automatic user flow documentation. When the tests have ran the tests results can be used to document the user flow. The existing structures of categories, features and scenarios can be reused to make the user flows browsable.

By detecting changes in tests, in case the app changes, it would be very nice to show the difference between different user flows between app versions. User flow documentation can be kept up-to-date with the app and create insights for the employees. This will also resolve the problem of employees having to make lots of manual taken screenshots, because they can just look up the screenshots from the existing work flows.

Bibliography

[1] Jenkins. `https://jenkins.io/`. (Accessed on 06/22/2017).

[2] Android Debug Bridge. `https://developer.android.com/studio/command-line/adb.html`. (Accessed on 06/21/2017).

[3] Processes and Application Life Cycle | Android Developers. `https://developer.android.com/guide/topics/processes/process-lifecycle.html`, . (Accessed on 06/21/2017).

[4] Getting Started with Testing | Android Developers. `https://developer.android.com/training/testing/start/index.html#test-types`, . (Accessed on 06/21/2017).

[5] ANTLR. `http://www.antlr.org/`. (Accessed on 05/02/2017).

[6] Appium: Mobile App Automation Made Awesome. `http://appium.io/`. (Accessed on 05/02/2017).

[7] Calaba.sh - Automated Acceptance Testing for iOS and Android Apps. `http://calaba.sh/`, . (Accessed on 05/02/2017).

[8] Introduction to Calabash. `https://developer.xamarin.com/guides/testcloud/calabash/introduction-to-calabash/`, . (Accessed on 05/02/2017).

[9] Cucumber. `https://cucumber.io/`. (Accessed on 05/02/2017).

[10] EarlGrey. `http://google.github.io/EarlGrey/`. (Accessed on 05/02/2017).

[11] Espresso. `https://google.github.io/android-testing-support-library/docs/espresso/`. (Accessed on 05/01/2017).

[12] Flux | Application Architecture for Building User Interfaces. `https://facebook.github.io/flux/docs/in-depth-overview.html`. (Accessed on 06/15/2017).

[13] Gherkin · cucumber/cucumber Wiki. `https://github.com/cucumber/cucumber/wiki/Gherkin`. (Accessed on 05/02/2017).

[14] Gradle Build Tool. `https://gradle.org/`. (Accessed on 06/22/2017).

[15] Service Name and Transport Protocol Port Number Registry. `https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml`. (Accessed on 06/21/2017).

[16] Maven – Welcome to Apache Maven. `https://maven.apache.org/`. (Accessed on 06/22/2017).

[17] React - a JavaScript library for building user interfaces. `https://facebook.github.io/react/`, . (Accessed on 06/15/2017).

[18] ReactiveX. http://reactivex.io/, . (Accessed on 06/15/2017).

[19] Github - RestKit/RestKit: RestKit is a framework for consuming and modeling RESTful web resources on iOS and OS X. https://github.com/RestKit/RestKit. (Accessed on 06/21/2017).

[20] Home · ReactiveX/RxJava Wiki. https://github.com/ReactiveX/RxJava/wiki. (Accessed on 06/15/2017).

[21] Socket.IO. https://socket.io/. (Accessed on 06/15/2017).

[22] Continuous Code Quality | SonarQube. https://www.sonarqube.org/. (Accessed on 06/22/2017).

[23] Spring. https://spring.io/, . (Accessed on 06/19/2017).

[24] Spring Boot. https://projects.spring.io/spring-boot/, . (Accessed on 06/19/2017).

[25] Testing Support Library. https://developer.android.com/topic/libraries/testing-support-library/index.html#UIAutomator. (Accessed on 05/01/2017).

[26] WebDriver. https://www.w3.org/TR/webdriver/. (Accessed on 05/02/2017).

[27] Xcode - Apple Developer. https://developer.apple.com/xcode/. (Accessed on 06/21/2017).

[28] User Interface Testing. https://developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/testing_with_xcode/chapters/09-ui_testing.html. (Accessed on 05/01/2017).

[29] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 435–445, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568271. URL http://doi.acm.org/10.1145/2568225.2568271.

A

Original Project Description

bunq might be a bank, but we're an IT-company first, founded and run by coders. That means automation is our game! Our systems are subject to regular deploys. We'd like to automate parts of our (code) testing on a larger scale than we have done until now. This setup needs to keep our extreme security demands in mind, since we are a bank. That's where you come in! You'll find a way to automatically test parts of our setup.

We would like to see a complete system of scripts to automate (a large part of) our testing. More importantly, the solution needs to be scalable to make it future-proof. We aim for a working prototype at the end of the project.

B

Post Update Basic Features Check

What is this part of the testrun about?

The checks below are all used to ensure basic functionality is working correctly, these checks are always ran before we start company wide testing and before we start TestFlight (iOS) or Alpha (Android) releases.

Signup

- Signup UserSuperLight in Slice
- Upgrade UserSuperLight to UserLight in bunq
- Upgrade UserLight to UserPerson in bunq
- Signup UserCompany in bunq

Payments

- Send a normal bunq-to-bunq Payment picking someone from suggestions
- Send a normal bunq-to-bunq Request picking someone from the contact picker
- Make an iDEAL deposit
- Make a SOFORT deposit
- Trigger a card payment card payment
- Make an online iDEAL payment
- Create and pay bunq.me requests through SMS
- Create and claim a bunq.to payment through email

Slice

- Create a group
- Send a chat message and a chat attachment in a group
- Add a custom expense
- Add an expense from a payment
- Settle the group

Other

- Setup a Connect
- Order a Card
- Event overview shows up correctly
- Check push messages
- Send a chat message and a chat attachment in Payment or Request
- Support chat
- Open admin

C Software Improvement Group Evaluation

The code scores 4 out of 5 stars on our maintainability model, which means that the maintainability of the code is above average. The highest score wasn't achieved due to lower scores for Unit Size and Unit Complexity.

Note: because your score is already quite high, all the recommendations are small points of improvement.

For Unit Size, the percentage of methods of which the length is above average is considered. Splitting these methods up in smaller pieces causes every part to be easier to understand and test, and therefore easier to maintain.

In your project, the method `Fling.call` is a good candidate to split up further. The part between line 31 and 43 is actually a separate method, `getDirection()`. At this moment, it's not yet a problem to do everything in one method, but when the functionality keeps growing, these kinds of refactors benefit the readability greatly. Similar problems occur in other subclasses of Instruction.

For Unit Facing, the percentage of units with an above average amount of parameters is considered. Usually, an above average amount of parameters indicates a lack of abstraction. Furthermore, a large number of parameters can lead to confusion in calling the method, and usually also mean longer methods.

In your case, the constructor of Runner has remarkably many parameters. Besides, four of these parameters are lists. If you look at what happens with these parameters, it's not too bad: most of them are only used to create a RunnerContext object. It would be better to pass the RunnerContext object to the Runner directly, that way you prevent the Runner depending on large amounts of data that aren't its direct responsibility.

The presence of test code is promising, hopefully the amount will increase when new functionality is added.

Generally, the code scores above average, hopefully this level will be maintained during the rest of the development phase.

D Initial Solution

We propose the following modular design, see figure D.1. First we introduce a server app that can be installed on the mobile device that can do actions and assertions through the UI automation libraries for Android and iOS: UI Automator[25] for Android and XCUITest[28] for iOS. This server app gets its instructions from the main application.

We also introduce an application that will be run on a continuous integration host. This application will serve a web interface for creating and editing tests, and showing test run results. It also connects to the server app to send instructions and receive results.

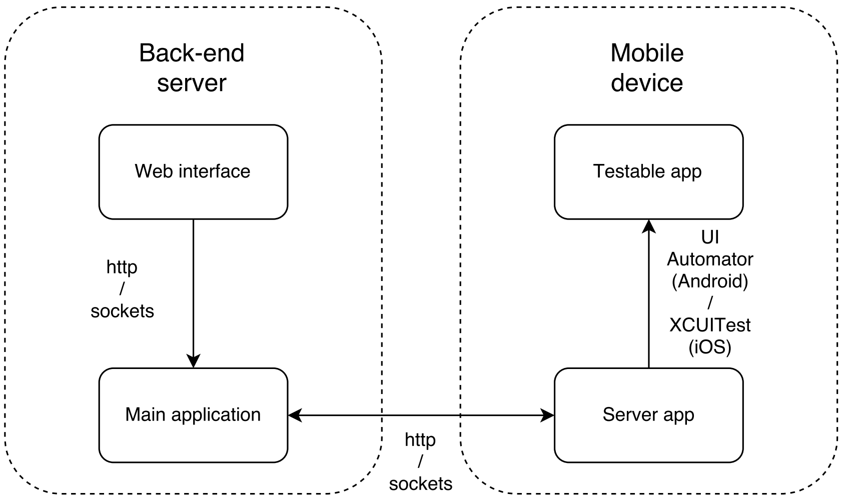


Figure D.1: Proposed design of the system

D.1. Back-end server

The main responsibility of the back-end server is to run the created test against the staging environment. It is also responsible for handling user instructions, monitoring and hosting the test creation toolkit.

D.1.1. Web interface

The web interface is what the testers will interact with. The core system can be instructed, tests can be monitored and new tests can be created or adjusted.

D.1.2. Main application

The main application is the core functionality of the framework. The core will interpret tests and send instructions to the server apps that are installed on the mobile devices. These instructions are part of the core protocol. It also checks whether actions and assertions on the testable app succeeded or failed through the server app.

It also has the possibility to handle user instructions like redoing certain tests or pausing the system. The core is continuously monitored and reports to a web interface. To help non-developers creating tests a test creation toolkit will optionally be offered to the user.

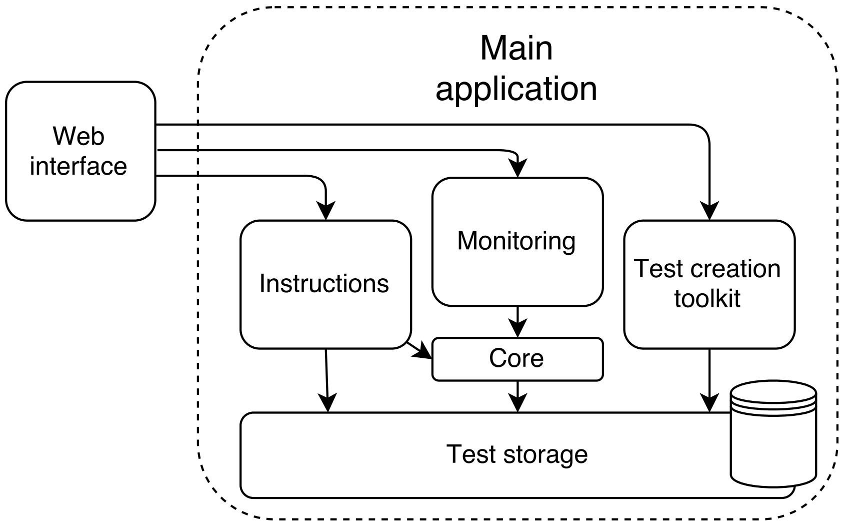


Figure D.2: Proposed design of the main appliction

D.2. Mobile device

The mobile device can be any testing device, emulator (Android) or simulator (iOS).

D.2.1. Testable app

The testable app remains exactly the same, so the testing framework can run without adding any code or dependencies.

D.2.2. Server app

The server app functions as an adapter. It is the talking point for the back-end server and receives instructions from the main application, the client. After receiving instructions it will pass trough those instructions to the UI Automator or XCUITest and thereby leaving the testable app unchanged.

This part of the testing framework is the only platform dependent part and therefore the codebase and complexity need to be minimised. When functionality is shared between Android and iOS in terms of testing the client should be responsible for handling the platform specific differences. The final result is shown in figure D.3.

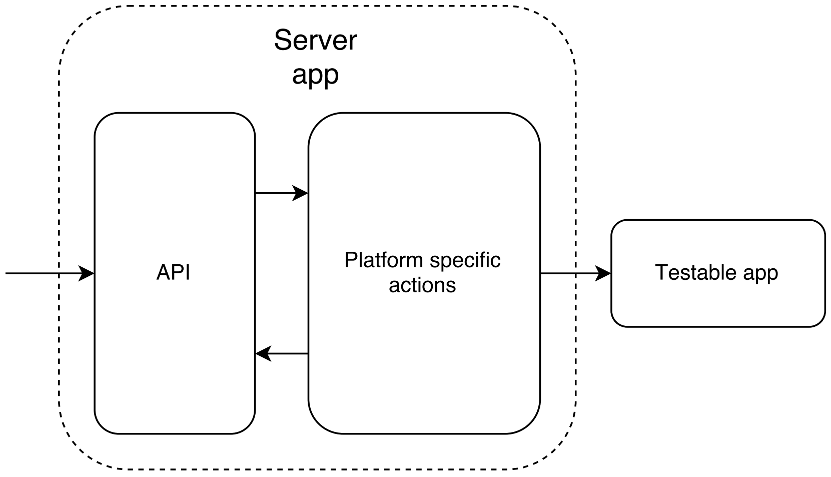


Figure D.3: Proposed design of the server app

D.3. Tests

Tests are going to be implemented as simple steps with blanks to fill in. This also makes it possible to define the tests for both Android and iOS with the same code.

Converting the current test definitions is relatively easy. The first test from appendix B can now be converted to a test like figure D.4.

If there are substantial architectural differences for certain scenarios between the Android and iOS apps, we will support writing separate tests for both platforms.

```
Feature: Account creation
    As a user I want to have an amazing experience when creating an account. I should be able
         to create and upgrade Slice and bunq accounts.

    Scenario: Successful signup UserSuperLight in Slice
        Given I start the slice app
        When I press the button with text JOIN US
        Then I see the slice phone number screen
        When I type 612345678
        And I click on next
        Then I see the slice verification screen
        And I receive an SMS
        When I type 123456
        Then I see the slice profile creation screen
        When I type Firstname Lastname
        And I press the checkbox with I accept all terms and conditions
        And I click on next
        Then I see the slice permission screen
        When I press the button with text skip
        And I press the button with text skip
        Then I see the slice main screen

    Scenario: ...
```

Figure D.4: Example of a successful signup test

E

Device API Reference

E.1. General

E.1.1. Request

A request must be a POST request with MIME-type `application/json`.

It must contain the following:

name	type	description	when
screenshot	boolean	whether a screenshot must be created for every step or only on failure	always

E.1.2. Response

A response must have MIME-type `application/json`.

It must contain the following:

name	type	description	when
failed	boolean	indicates whether the instruction was successful or not	always
message	string	a message describing the success or failure	always
screenshot	string	a base64-encoded screenshot	on failure

E.2. Instructions

E.2.1. /instruction/press_element_by_id

Press an element by its ID.

Request

name	type	description	required
viewId	string	the id of the element to press	yes

E.2.2. /instruction/press_element_by_text

Press an element by its text. The specified text may also be a substring. It's case sensitive.

Request

name	type	description	required
text	string	the text of the element to press	yes

E.2.3. /instruction/press_back

Press the back button.

Request

No specific request.

E.2.4. /instruction/register_dialog_watcher

Register a watcher that automatically accepts all dialogs conform a specified format. Only one watcher can be registered at the same time.

Request

name	type	description	required
layoutElements	list[string]	if and only if all of these IDs are visible, the dialog should be accepted	yes
okButton	string	the ID of the button to press to accept the dialog	yes

E.2.5. /instruction/start_app

Start a specific application.

Android

It will start the activity with the `android.intent.category.LAUNCHER` category.

Request

name	type	description	required
appName	string	the package name of the application	yes

E.2.6. /instruction/type

Type a string.

Android

The available characters are limited to what is typable with a standard ANSI or ISO US International keyboard.

Request

name	type	description	required
text	string	the text to type	yes

E.2.7. /instruction/unregister_dialog_watcher

Deregister the watcher that is registered with `/instruction/register_dialog_watcher`. If no dialog watcher is registered, nothing will happen.

Request

No specific request.

E.2.8. /instruction/verify_screen

Verify if a layout ID is currently visible on the screen.

Request

name	type	description	required
screenName	string	the ID of the layout to verify	yes

E.2.9. /instruction/verify_text

Verify if the specified text is currently visible on the screen. May also be a substring.

Request

name	type	description	required
text	string	the text to verify	yes

E.2.10. /instruction/fling

Swipe in the specified direction.

Request

name	type	description	required
direction	string	the direction to swipe to (can only be left or right)	yes

F

Implemented Instructions

The parts in-between `<>` are dynamic parameters, and the *italicised* parts are optional parameters.

I type `<text>`: Type `<text>` on the device.

I *possibly* press the element with `<text>`: Press an element containing `<text>`. Will not fail if no such element is found when *possibly* is specified.

I *possibly* press the `<id>` element: Press an element with that can be identified with `<id>`. Will not fail if no such element is found when *possibly* is specified.

I press the back button: Press the back button on the device. **I start the `<app>` app:** Start the `<app>` application.

I should see the `<id>` screen: Verify whether a layout element exists that can be identified with `<id>`.

I should see an element with `<text>`: Verify whether `<text>` is visible.

I fling `<direction>`: Swipe in the `<direction>` direction. `<direction>` should be one of `left` or `right`.

I wait for `<seconds>` seconds: Do nothing for `<seconds>` seconds.

G

Project Infosheet

See next page.

General Information

Title of the project: Automated Testing
Name of the client organisation: bunq
Date of the final presentation: July 3rd, 2017

Client

Name: Ir. W. Van
Affiliation: bunq

Coach

Name: Dr. ir. F. Palomba
Affiliation: Software Engineering Research Group

Description

Every week bunq loses a lot of time with manual testing the weekly releases of their Android and iOS apps. This work is also done by non-developers. Ideally bunq wants a solution that makes automated testing of their apps possible by non-developers. Defining tests once for iOS and Android is considered a big bonus. Since bunq has more than one app the solution should be app-independent.

Challenge
The main challenges were handling all the asynchronous data actions that had to be performed combined with I/O operations. We resolved these problems by using RxJava and by abstracting the I/O-layer. The workload was heavy for two persons.

Research
During the research phase we mainly learned about the already available tooling for testing Android and iOS apps. We learned that the available tools were not sufficient for what we were trying to do, but that some aspects of tools could be reused. Also some frameworks could be used as partial solution for subproblems.

Process
Since we both had our area of expertise one of us started working on the back-end and the other one on the android app. By defining the API early we could work towards a working product. Later the person that was working on the back-end started building the front-end and attached it to the back-end. The person that created the android app integrated it even further by creating a device connector. In the end we worked on all the parts together. All code has gone trough pull requests and in the end we both understand all components.

Product
The product resulted in a behaviour driven testing application for Android with extensibility for iOS in mind (interfaces left to implement for iOS). Non-developers can define the behaviour they want to see from their application and run the tests. Tests can be done on multiple devices simultaneously. The testing framework is also app-independent. The code is tested by both unit tests and implementation tests.

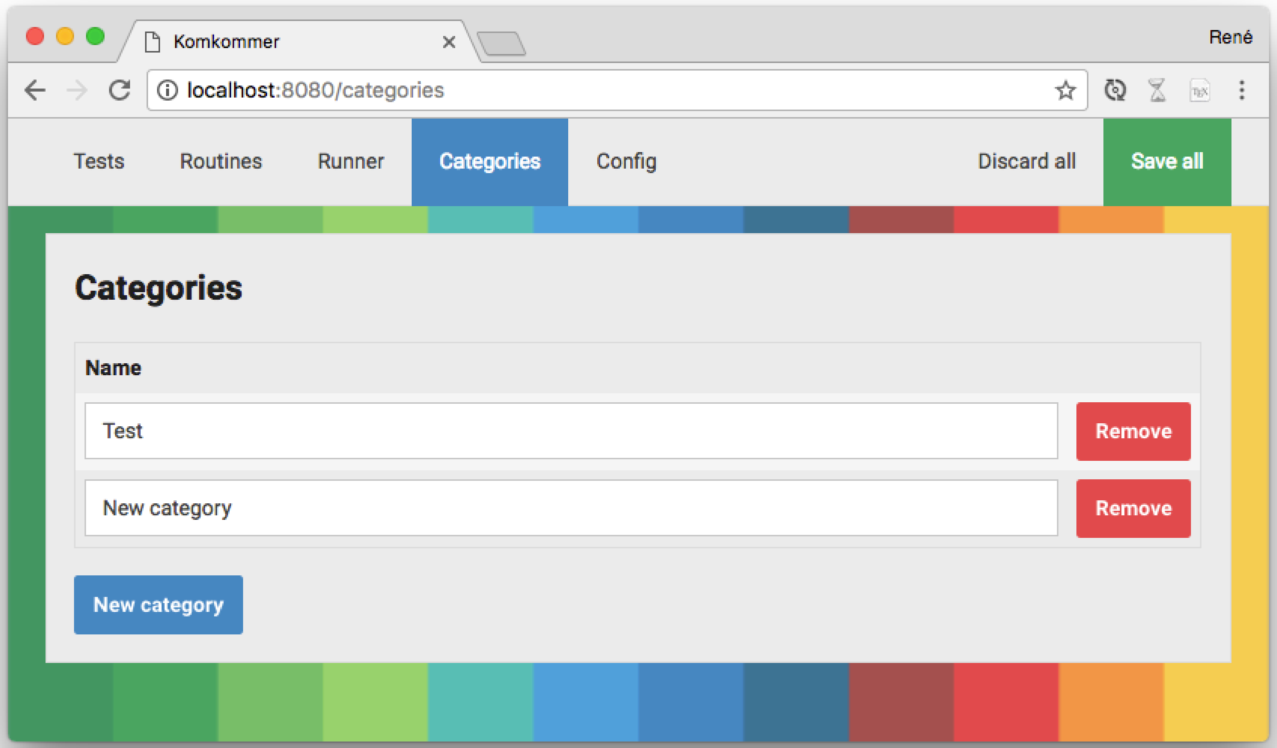
Outlook
The framework is already in use and will be further extended by the client. Recommendations in terms of how to proceed were made.

Members of the project team

Name: René Vennik
Interests: High performance scalable systems, problem solving, programming concepts
Contribution and role: Architecture Design; Front-end; Back-end runner, executor and webserver

Name: Wim de With
Interests: System administration, low level architecture
Contribution and role: DevOps; Back-end device connector; Android implementation

H
Screenshots



Categories

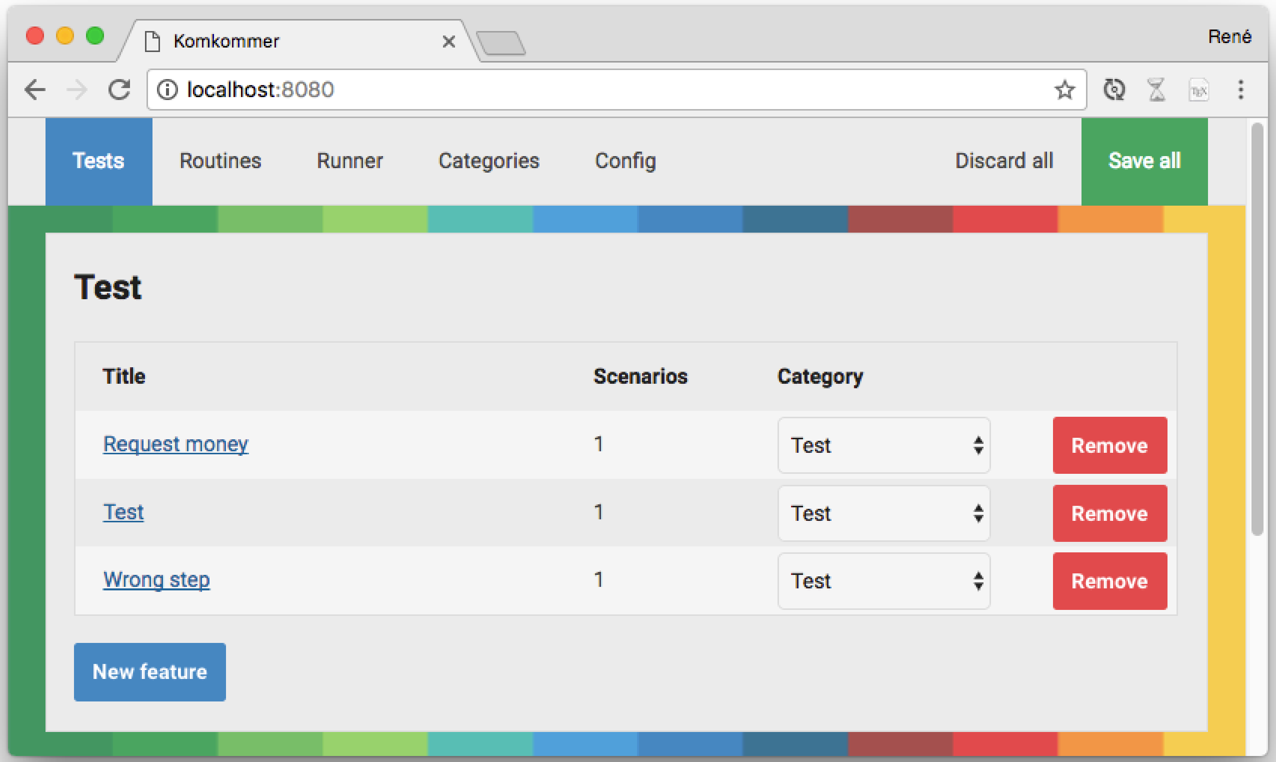
Name

Test
Remove

New category
Remove

New category

Figure H.1: Create and modify categories

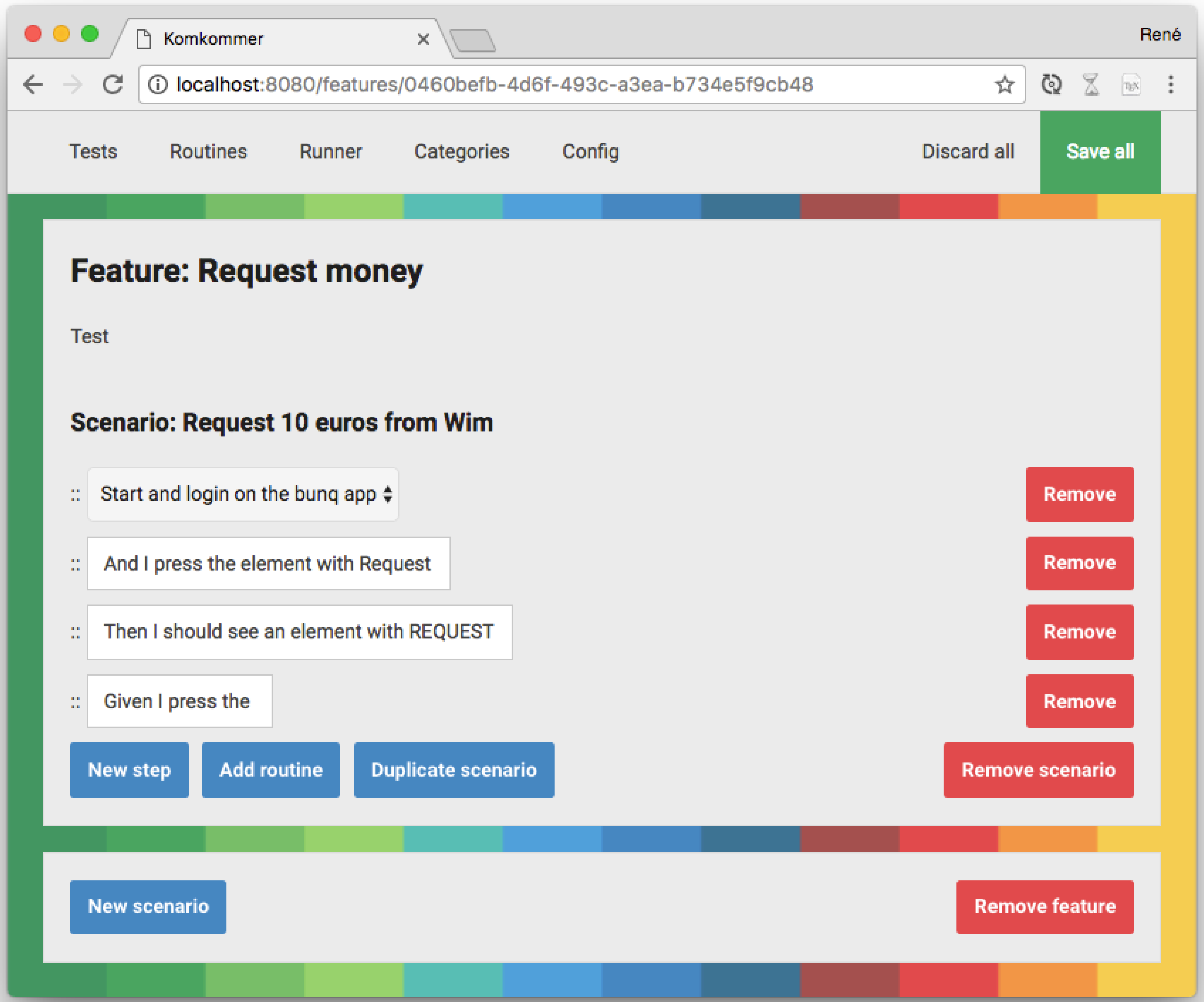


Test

Title	Scenarios	Category	
Request money	1	Test	Remove
Test	1	Test	Remove
Wrong step	1	Test	Remove

New feature

Figure H.2: View, move and navigate to tests (features)



Feature: Request money

Test

Scenario: Request 10 euros from Wim

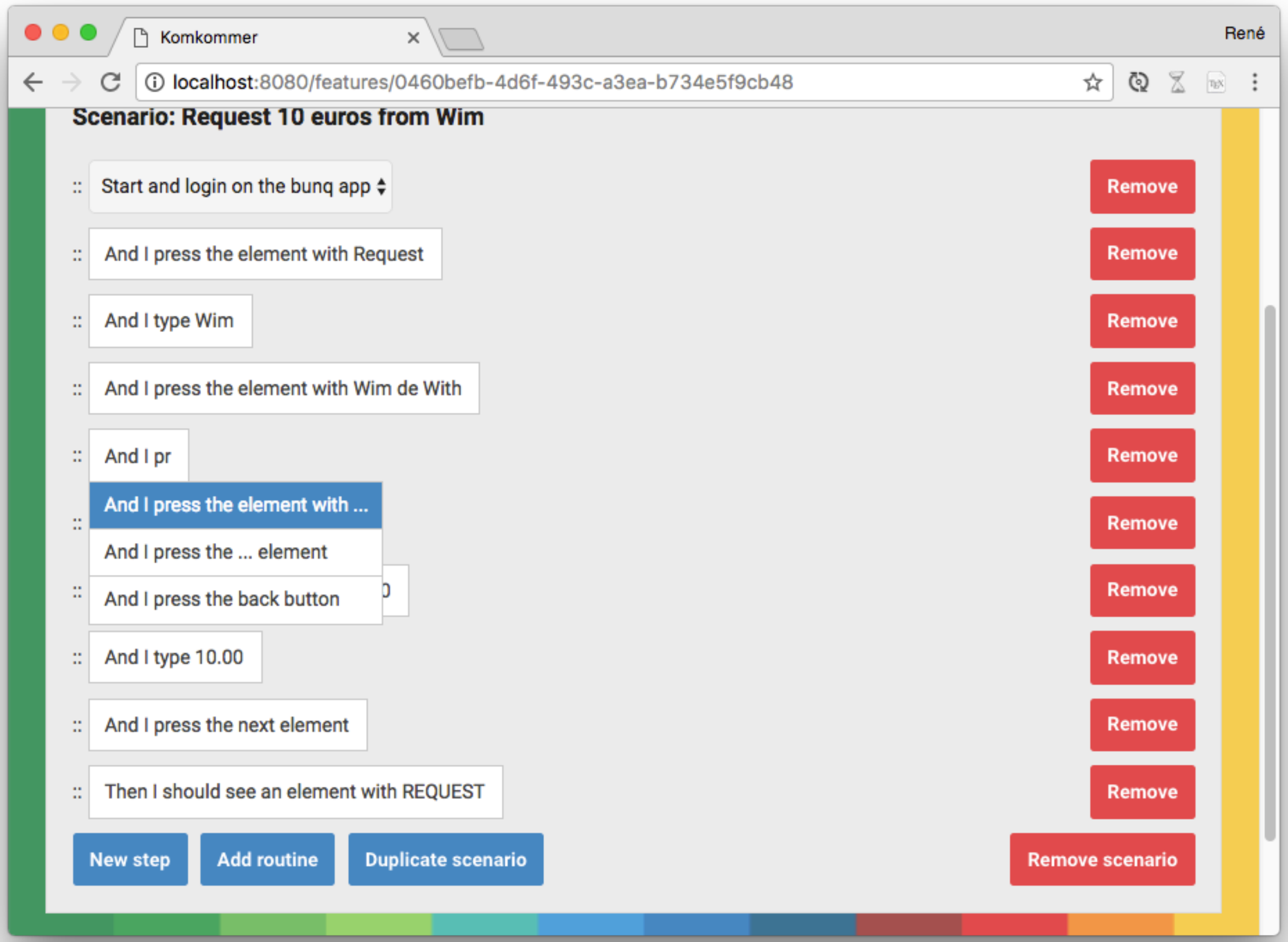
:: Start and login on the bunq app

:: And I press the element with Request

:: Then I should see an element with REQUEST

:: Given I press the

Figure H.3: Change a feature


Komkommer

localhost:8080/features/0460befb-4d6f-493c-a3ea-b734e5f9cb48

René

Scenario: Request 10 euros from Wim

Start and login on the bunq app

And I press the element with Request

And I type Wim

And I press the element with Wim de With

And I pr

And I press the element with ...

And I press the ... element

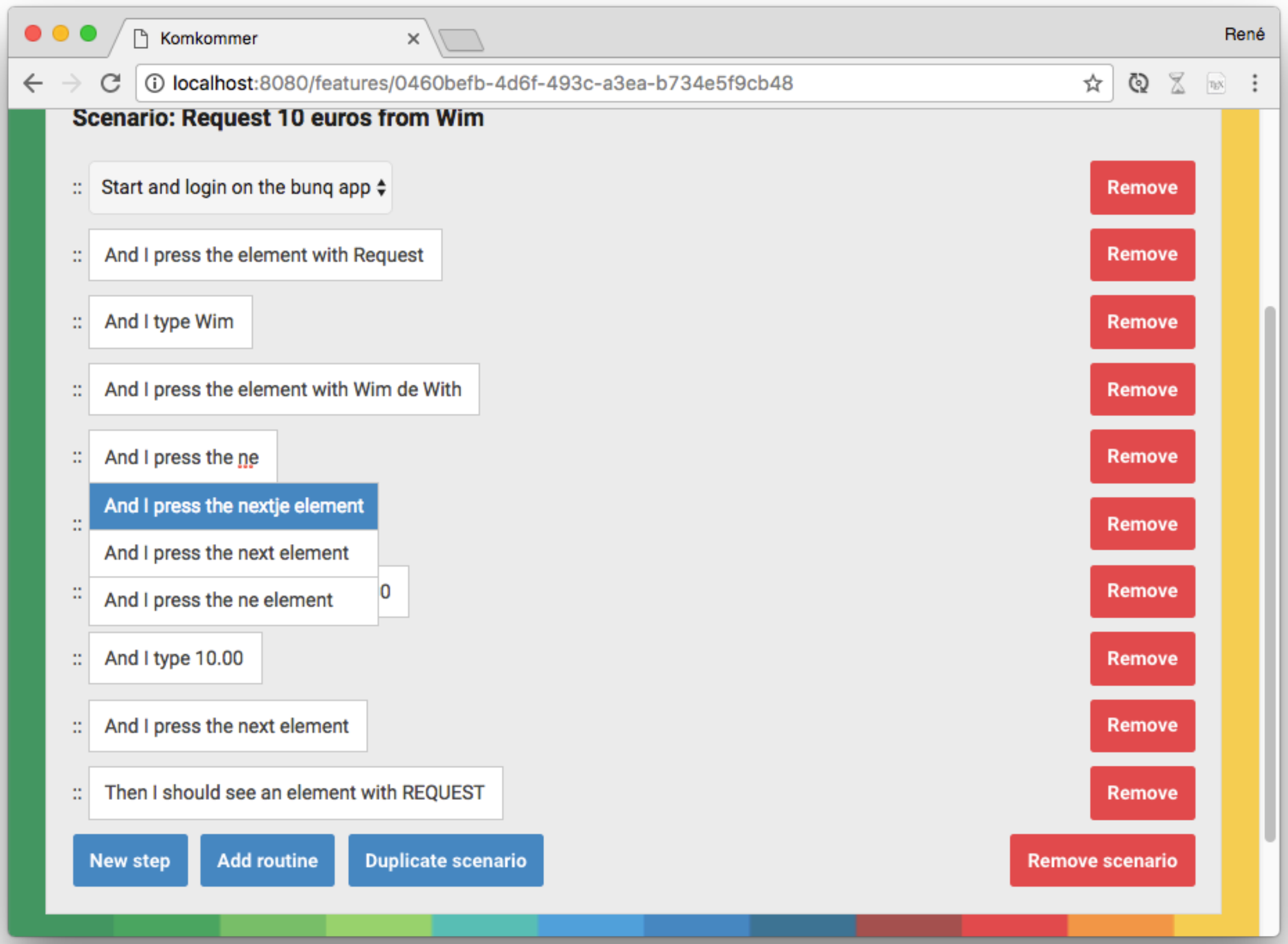
And I press the back button

And I type 10.00

And I press the next element

Then I should see an element with REQUEST

Figure H.4: Autocomplete of step


Komkommer
René
localhost:8080/features/0460befb-4d6f-493c-a3ea-b734e5f9cb48

Scenario: Request 10 euros from Wim

Start and login on the bunq app

And I press the element with Request

And I type Wim

And I press the element with Wim de With

And I press the ne

And I press the nextje element

And I press the next element

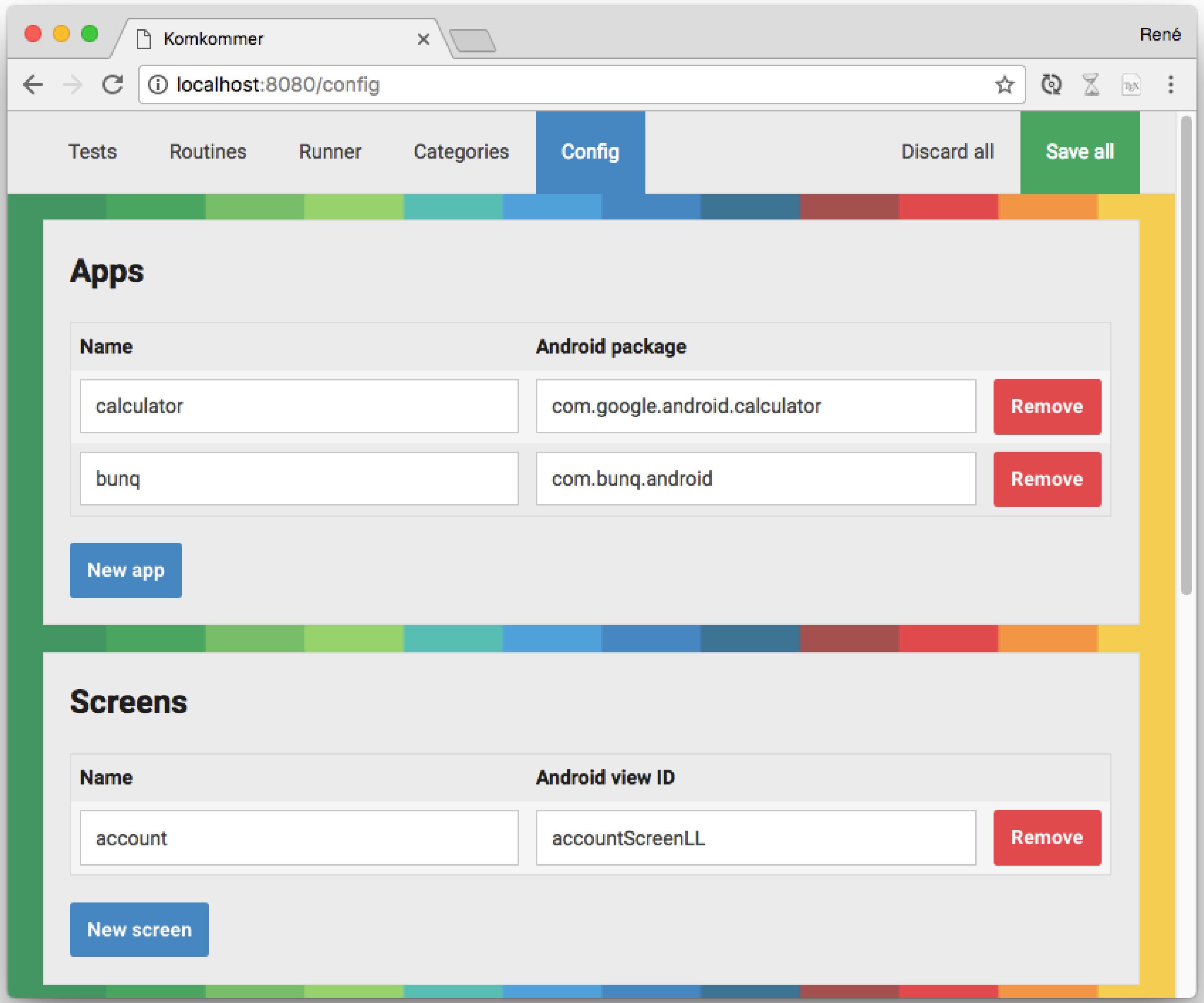
And I press the ne element

And I type 10.00

And I press the next element

Then I should see an element with REQUEST

Figure H.5: Autocomplete of arguments



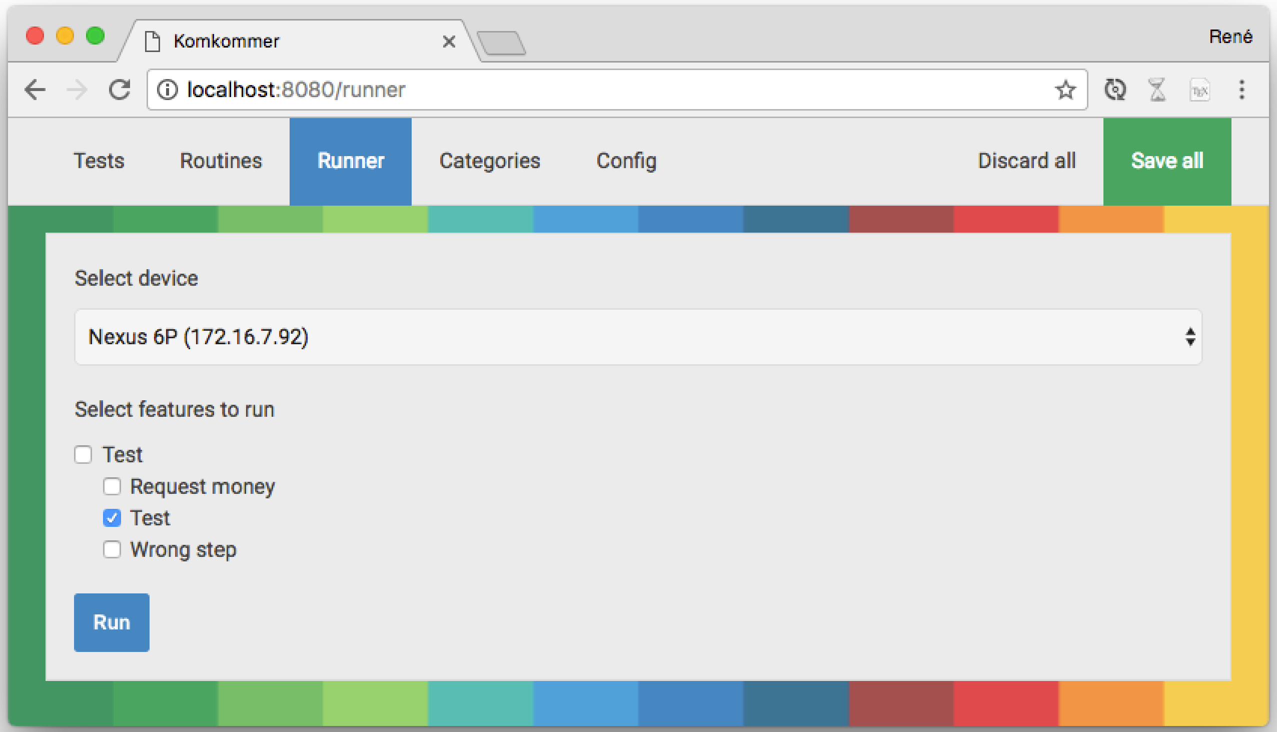
Apps

Name	Android package	
calculator	com.google.android.calculator	Remove
bunq	com.bunq.android	Remove

Screens

Name	Android view ID	
account	accountScreenLL	Remove

Figure H.6: Configure mappings



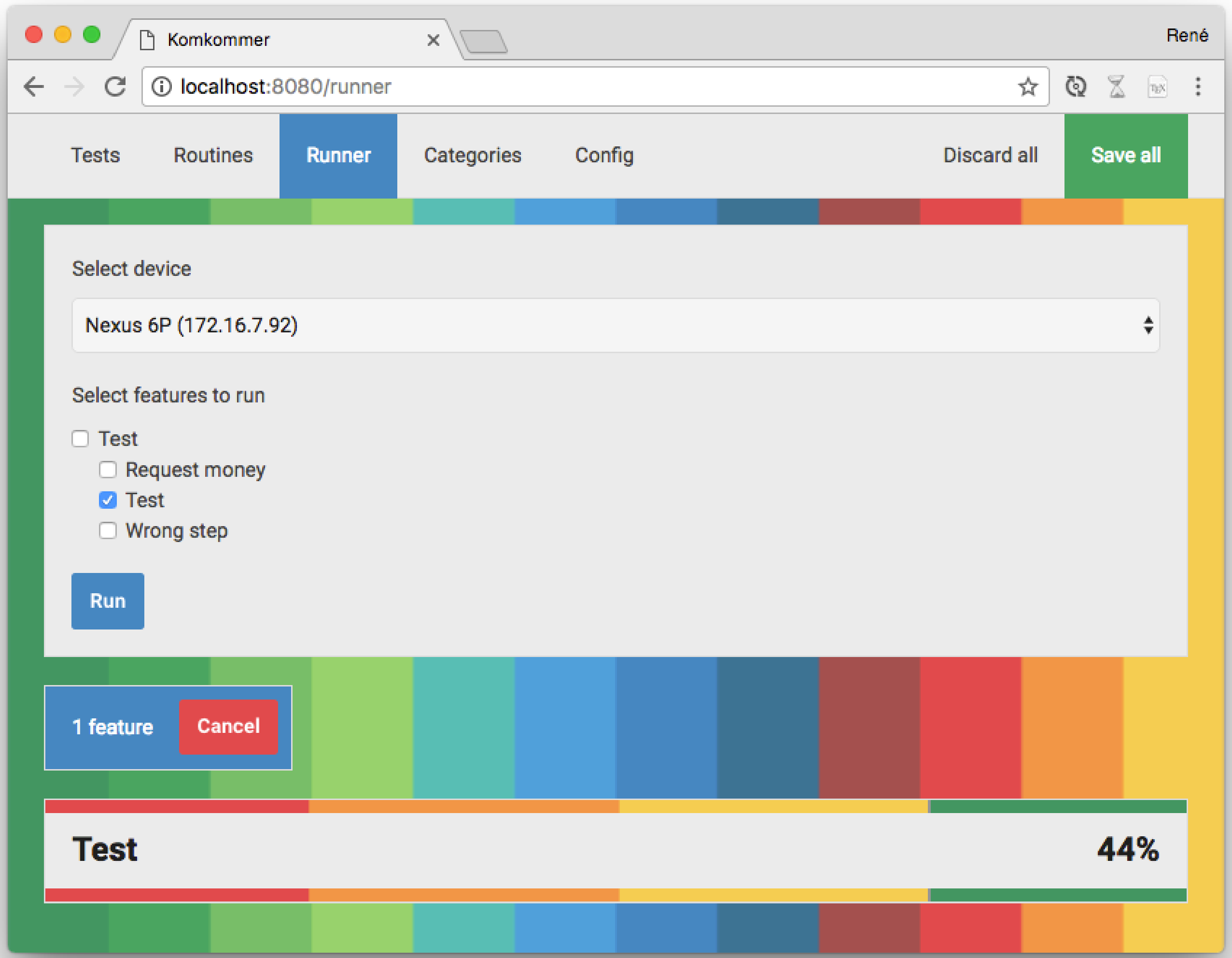
Select device

Nexus 6P (172.16.7.92)

Select features to run

- [ ] Test
  - [ ] Request money
  - [✓] Test
  - [ ] Wrong step

Figure H.7: Run selected features



Select device

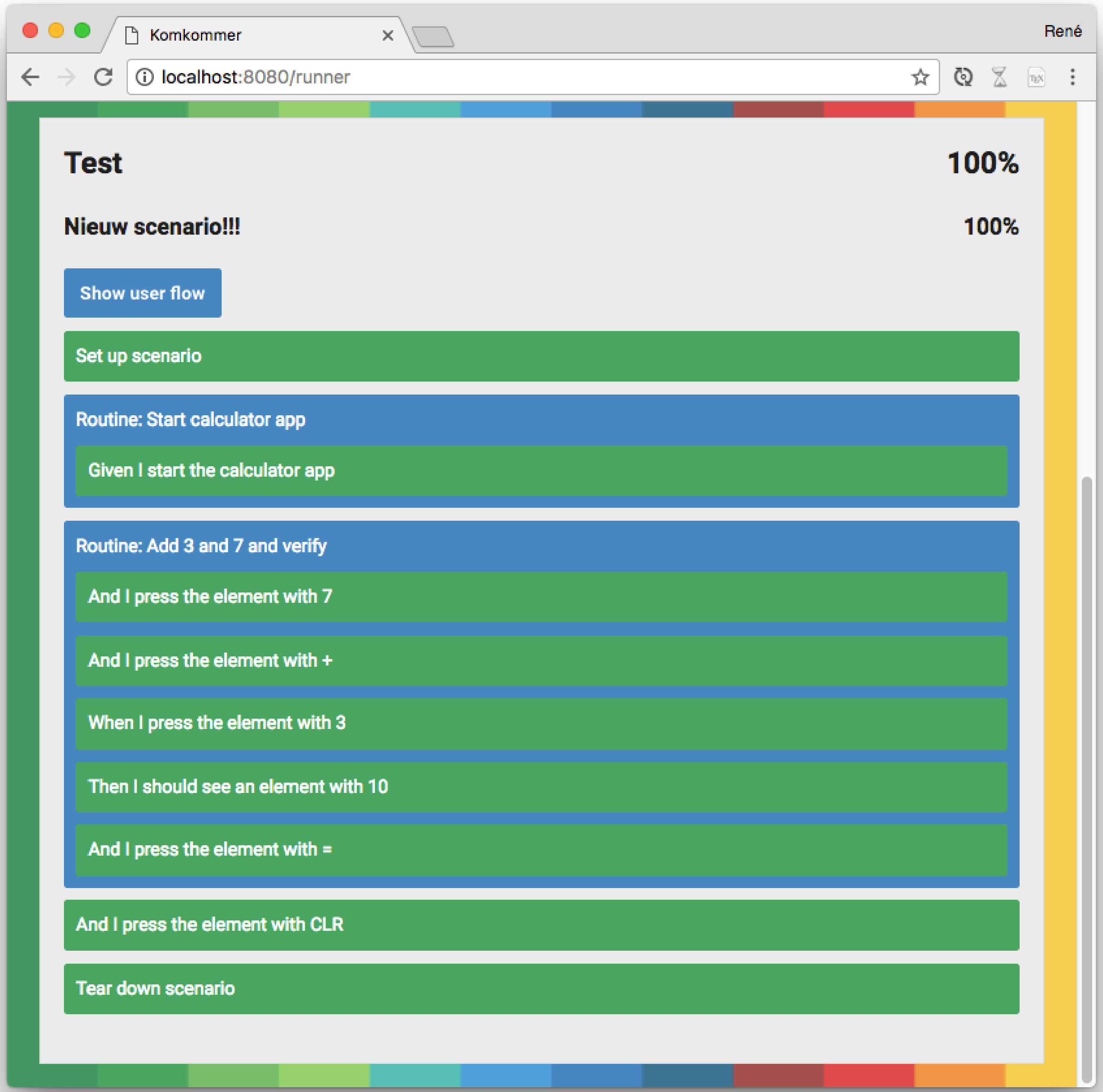
Nexus 6P (172.16.7.92)

Select features to run

- [ ] Test
  - [ ] Request money
  - [✓] Test
  - [ ] Wrong step

Test 44%

Figure H.8: Running runner



Test 100%

Nieuw scenario!!! 100%

Set up scenario

Routine: Start calculator app

Given I start the calculator app

Routine: Add 3 and 7 and verify

And I press the element with 7

And I press the element with +

When I press the element with 3

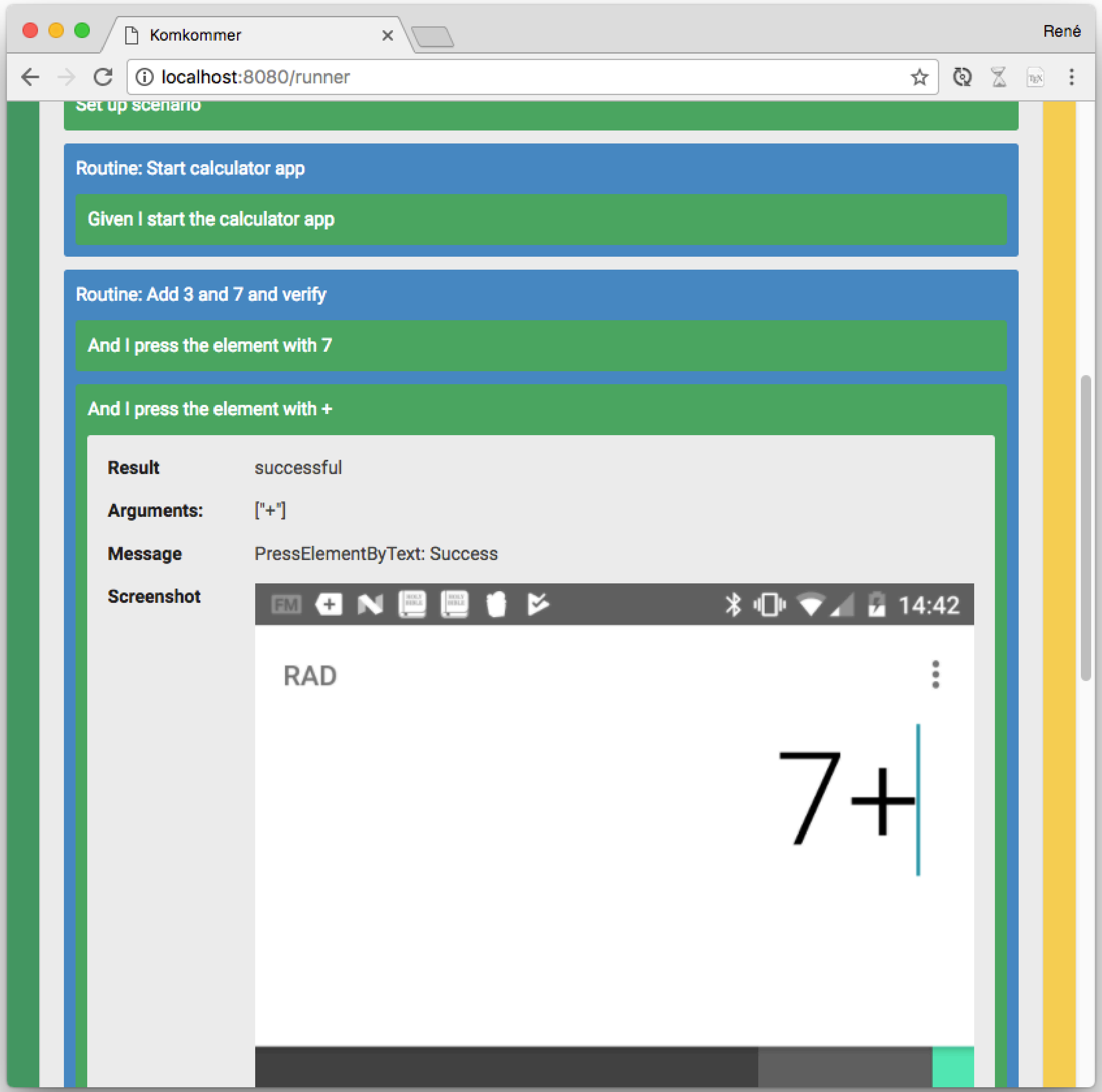
Then I should see an element with 10

And I press the element with =

And I press the element with CLR

Tear down scenario

Figure H.9: Result of runner



Set up scenario

Routine: Start calculator app

Given I start the calculator app

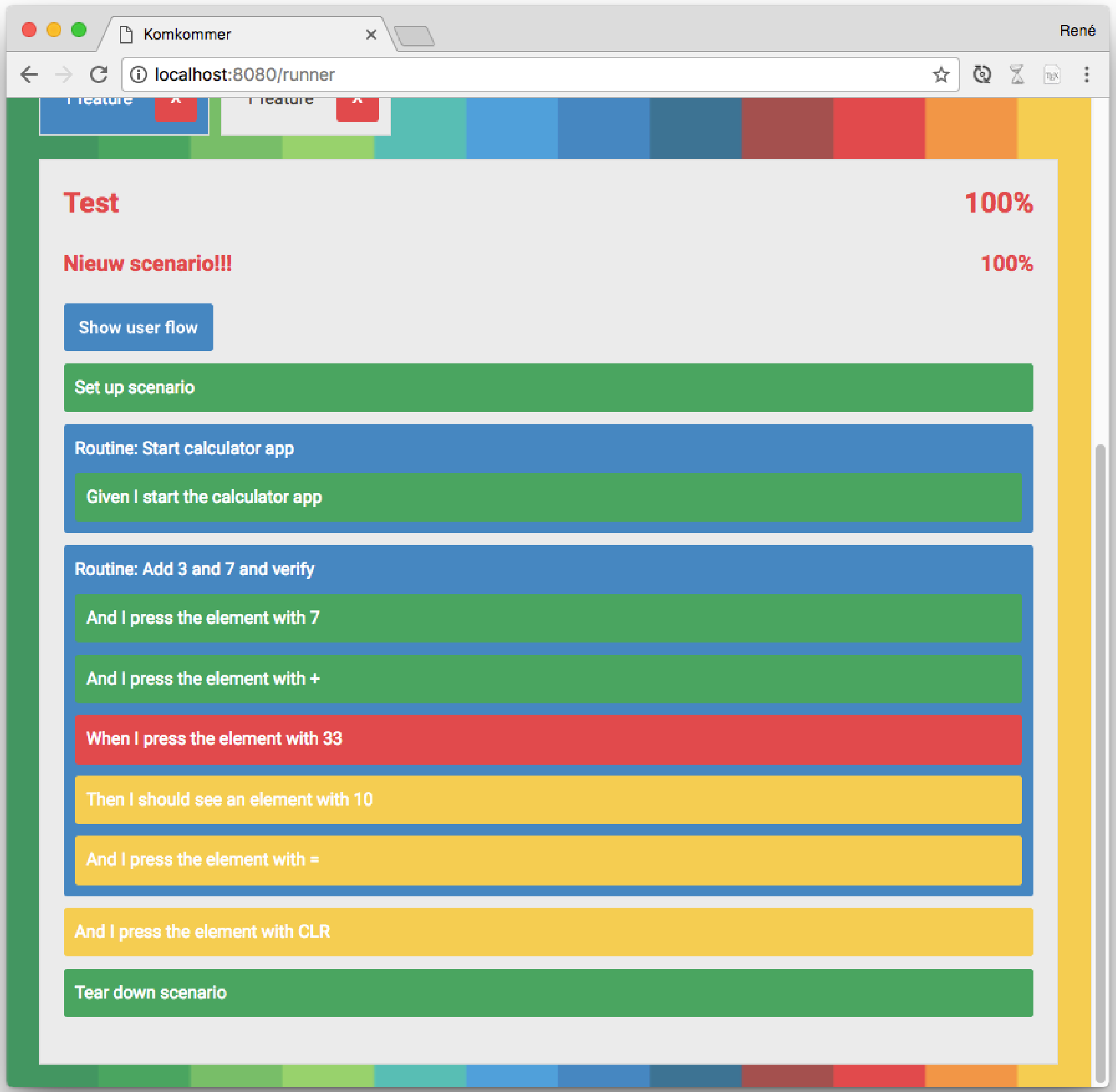
Routine: Add 3 and 7 and verify

And I press the element with 7

And I press the element with +

Result	successful
Arguments:	["+"]
Message	PressElementByText: Success
Screenshot	

Figure H.10: Step result of runner



Komkommer

René

localhost:8080/runner

Feature

Feature

Test 100%

Nieuw scenario!!! 100%

Show user flow

Set up scenario

Routine: Start calculator app

Given I start the calculator app

Routine: Add 3 and 7 and verify

And I press the element with 7

And I press the element with +

When I press the element with 33

Then I should see an element with 10

And I press the element with =

And I press the element with CLR

Tear down scenario

Figure H.11: Failed step result of runner

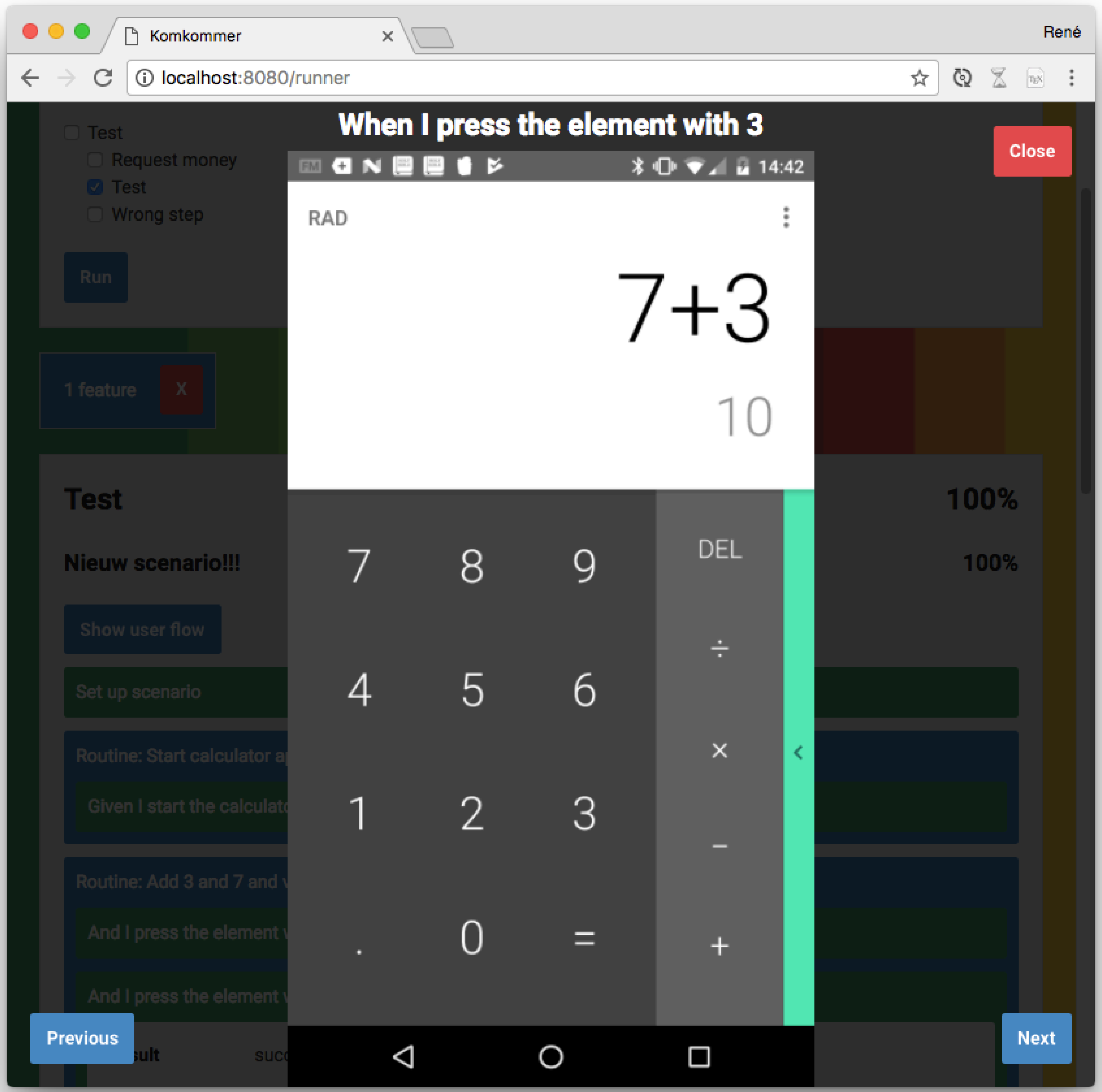
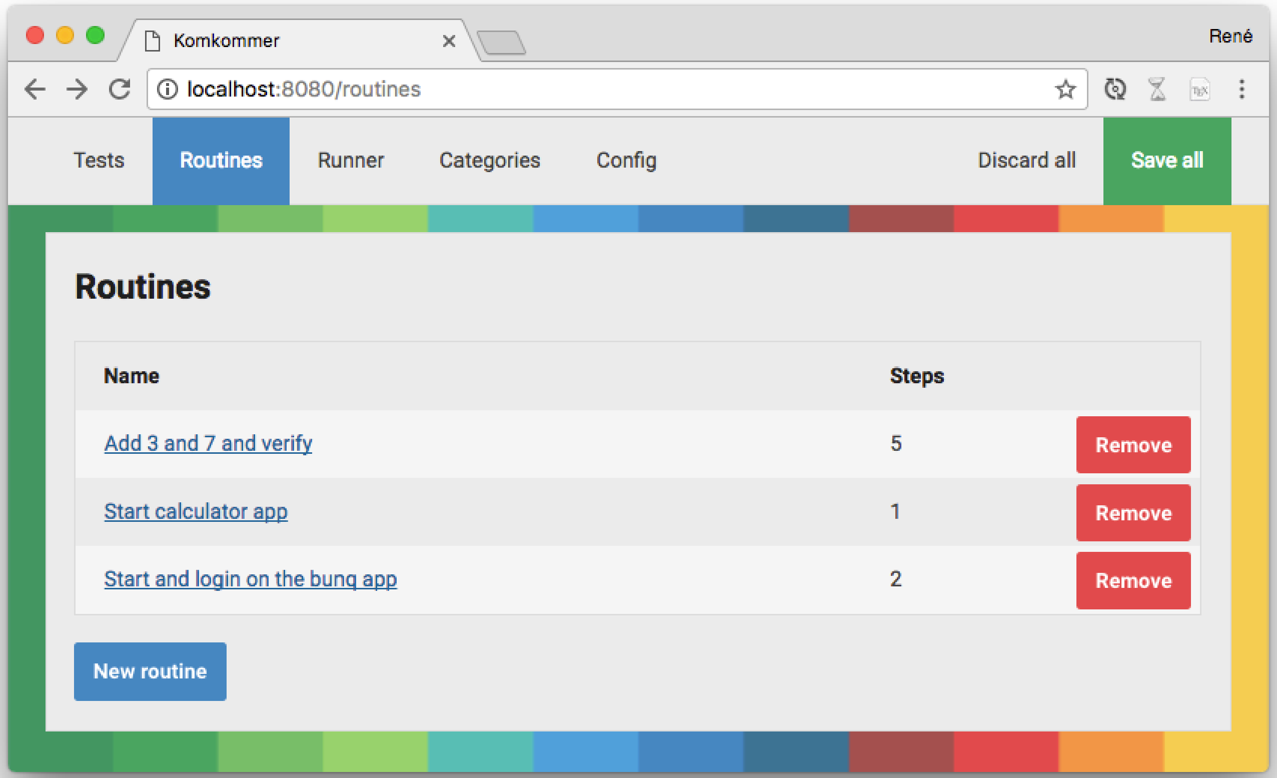


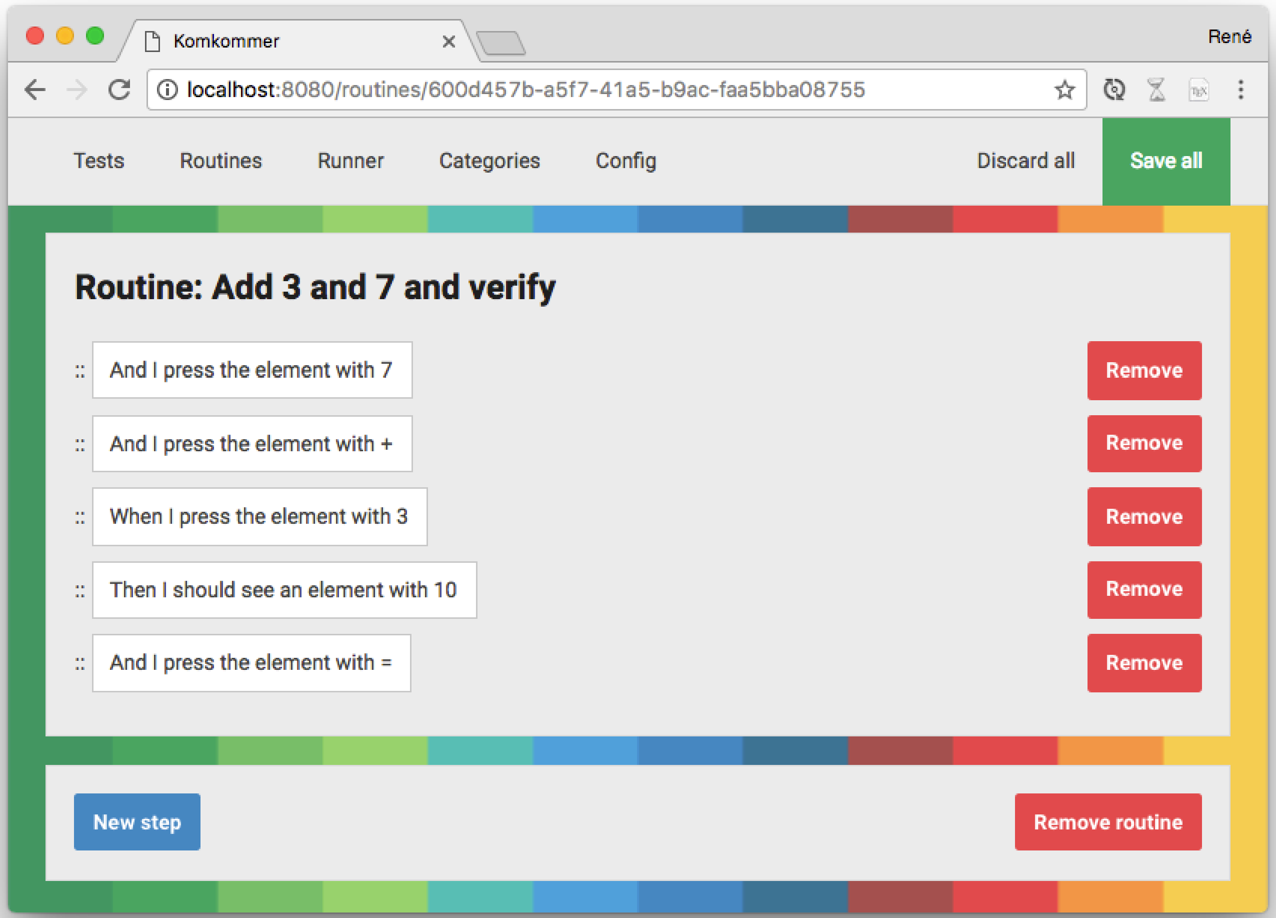
Figure H.12: User flow of feature



Routines

Name	Steps
Add 3 and 7 and verify	5
Start calculator app	1
Start and login on the bunq app	2

Figure H.13: Overview of routines



Routine: Add 3 and 7 and verify

And I press the element with 7

And I press the element with +

When I press the element with 3

Then I should see an element with 10

And I press the element with =

Figure H.14: Change a routine