

# Continuous integration for End-to-End testing of mobile applications

Gabriele Fantini

March 14, 2022

# Contents

<b>I</b>	<b>Introduction</b>	<b>4</b>
<b>1</b>	<b>Background</b>	<b>5</b>
1.1	Software testing . . . . .	5
1.1.1	Software verification and validation phase . . . . .	5
1.1.2	Levels of tests . . . . .	7
1.2	Testing mobile applications . . . . .	10
1.2.1	Challenges . . . . .	10
1.2.2	Automation frameworks . . . . .	12
1.2.3	Record and replay tools . . . . .	16
1.2.4	Automated test input generation techniques . . . . .	19
1.2.5	Bug and error reporting/monitoring tools . . . . .	19
1.2.6	Mobile testing services . . . . .	21
1.2.7	Device streaming tools . . . . .	22
1.3	Continuous integration, delivery and deployment . . . . .	22
1.3.1	Continuous Integration . . . . .	22
1.3.2	Continuous Delivery . . . . .	22
1.3.3	Continuous Deployment . . . . .	22
<b>II</b>	<b>Evaluation of tools for mobile testing</b>	<b>24</b>
<b>2</b>	<b>Selected Instruments</b>	<b>26</b>
2.1	Android Espresso . . . . .	26
2.1.1	Architecture Details . . . . .	26
2.1.2	Environment Setup . . . . .	27
2.1.3	Tool Capabilities . . . . .	28
2.2	Appium . . . . .	31
2.2.1	Architecture Details . . . . .	31

2.2.2	Environment Setup . . . . .	32
2.2.3	Tool Capabilities . . . . .	32
2.3	Sikuli . . . . .	35
2.3.1	Architecture Details . . . . .	35
2.3.2	Environment Setup . . . . .	35
2.3.3	Tool Capabilities . . . . .	35
<b>3</b>	<b>Experimental Subject</b>	<b>39</b>
<b>4</b>	<b>Procedure</b>	<b>40</b>
4.1	Test Cases . . . . .	40
4.1.1	Espresso . . . . .	40
4.1.2	Appium . . . . .	40
4.1.3	Sikuli . . . . .	40
<b>5</b>	<b>Tools Evaluation</b>	<b>49</b>
5.1	Final considerations . . . . .	49
5.1.1	Espresso . . . . .	49
5.1.2	Appium . . . . .	50
5.1.3	Sikuli . . . . .	51
<b>III</b>	<b>Tools implementation in a CI/CD pipeline</b>	<b>53</b>
<b>6</b>	<b>Tools for CI/CD</b>	<b>54</b>
6.1	GitHub Actions in depth . . . . .	55
6.1.1	Workflow . . . . .	55
6.1.2	Events . . . . .	55
6.1.3	Jobs . . . . .	56
6.1.4	Steps . . . . .	56
6.1.5	Actions . . . . .	56
6.1.6	Runners . . . . .	56
<b>7</b>	<b>Implementing a CI/CD pipeline</b>	<b>57</b>
7.1	Android Emulator Runner . . . . .	57
7.2	CI/CD pipeline using Espresso . . . . .	58
7.2.1	Repository Workflow . . . . .	58
7.3	CI/CD pipeline using Appium . . . . .	59
7.3.1	OmniNotes Repository . . . . .	59

7.3.2	Appium Repository . . . . .	63
7.4	CI/CD pipeline using Sikuli . . . . .	64
7.4.1	OmniNotes Repository . . . . .	64
7.4.2	Sikuli Repository . . . . .	64
7.4.3	Screenshots Recording . . . . .	69
7.5	The X virtual framebuffer . . . . .	70
<b>8</b>	<b>CI/CD Pipelines Evaluation</b>	<b>72</b>
8.1	Procedure . . . . .	72
8.1.1	Increased Test Set . . . . .	72
8.1.2	Pipelines and Tools Modifications . . . . .	72
8.1.3	Analyzing Fragmentation . . . . .	73
8.1.4	Analyzing Flakiness . . . . .	74
8.2	Results . . . . .	74
8.2.1	Considerations . . . . .	76
<b>IV</b>	<b>Conclusions</b>	<b>78</b>
<b>9</b>	<b>Achievements</b>	<b>79</b>
9.1	Espresso pipeline . . . . .	79
9.2	Appium pipeline . . . . .	79
9.3	Sikuli pipeline . . . . .	80
<b>10</b>	<b>Future developments</b>	<b>81</b>
10.1	Continuous Integration Environment . . . . .	81
10.2	Testing Framework Implementation . . . . .	81
10.2.1	Appium . . . . .	82
10.2.2	Sikuli . . . . .	82
<b>11</b>	<b>Acknowledgements</b>	<b>83</b>

# Part I

## Introduction

# Chapter 1

## Background

### 1.1 Software testing

Software testing is the process of evaluating and verifying that an application or software product is functioning properly against requirements. Benefits of testing include bug prevention, reduced development costs, and improved performance. Software testing is a crucial phase in any software development life cycle, especially in AGILE's ones. An example of a common software life cycle is represented in Figure 1.1. The duration of each iteration depends on the adopted software development practice, varying from one week to several months. Other than the test phase, there is the strategy phase that comprehends a planning stage and an analysis stage. The first helps to define the project scope as well as the problems, the latter helps build the system's requirements. The design stage outlines the details for the overall system, even down to specific aspects. The development phase is where developers actually start writing code and building applications in accordance with the pre-defined design and requirements. The deployment phase is where the final software solution is deployed into the target production environment. And the maintenance phase, in which developers handle possible software bugs or problems.

#### 1.1.1 Software verification and validation phase

The testing phase can be divided into two macro areas: the verification phase and the validation phase.

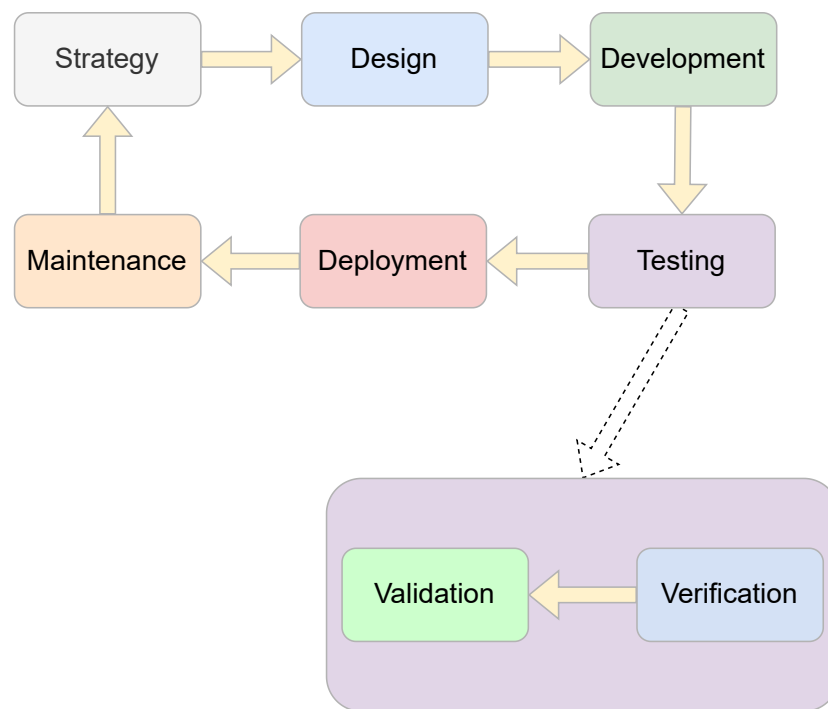


Figure 1.1: Software life cycle with its phases.

## Software verification

Software verification is the process of checking code, design, documents and programs to verify that the software has been built according to the requirements or not. It answers the question of "Are we building the product right?". During this phase no test code is executed, giving it the name of static testing. Methods used in verification are reviews, walkthroughs, inspections and desk-checking. Usually, software verification is done by the quality assurance team.

## Software validation

Software validation is a dynamic process of testing and validating if the software product meets the exact needs of the customer or not. The process helps to ensure that the software fulfills the desired use in an appropriate environment. It answers the question of "Are we building the right product?". This phase includes tests code execution, giving it the name of dynamic testing. Methods used in validation are Black Box Testing, White Box Testing and non-functional testing. Usually, software validation is done by the testing team.

### 1.1.2 Levels of tests

Different levels of tests exist, each one with a specific granularity (how the test is isolated or integrated) and goals. The Figure 1.2 illustrates the test pyramid. As Martin Fowler says:

"The test pyramid is a way of thinking about how different kinds of automated tests should be used to create a balanced portfolio. Its essential point is that you should have many more low-level UnitTests than high level BroadStackTests running through a GUI. [...] In short, tests that run end-to-end through the UI are: brittle, expensive to write, and time consuming to run. So the pyramid argues that you should do much more automated testing through unit tests than you should through traditional GUI based testing." [10].



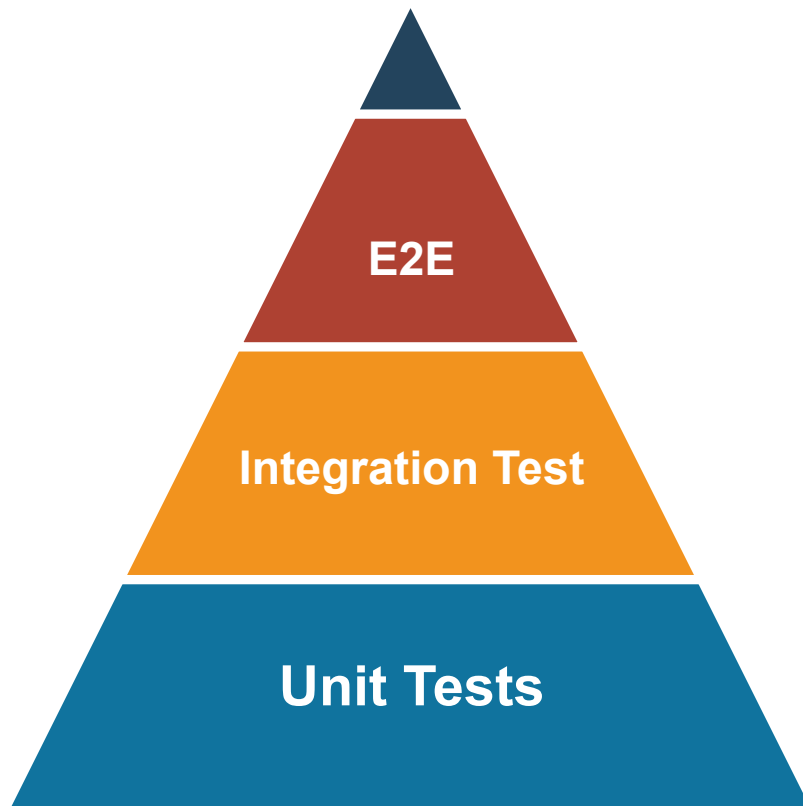


Figure 1.2: Tests pyramid.

## **Unit testing**

In unit testing, the smallest testable parts of an application, called units, are individually and independently scrutinized for proper operation, to ensure that the individual parts of a program work properly on their own, speeding up testing strategies and reducing wasted tests. A unit can be almost anything, like a line of code, a method, or a class. Generally, though a unit is traceable to a single function. This testing methodology is done during the development process by the software developers and sometimes QA staff.

## **Integration testing**

Integration testing usually involves testing a particular unit (usually referred to as a module or functionality) that has dependencies on another unit. The goal of these tests is to check the connectivity and communication between different components of the application. It is performed in an integrated hardware and software environment to ensure that the entire system functions properly.

## **End to end testing**

In end-to-end testing, the goal is to test the product the same way a real user experiences it, testing the application's workflow from beginning to end to make sure everything functions as expected.

## **Others test modalities**

There are also other different modalities in which tests can be made, each one independent from the granularity level and not mutually exclusive from the others. These modalities are:

- Record and Replay testing, a way of testing the application graphical user interfaces. It consists in a tool which captures a user interactive session and then automatically replay it any number of times without human intervention.
- Model Based testing, a technique in which test cases are derived from a model that describes the functional aspects of the system under test.

- Random testing, also known as monkey testing, is a technique where the software is tested by providing random inputs and checking its behavior.
- Black-box testing, where functionalities of software applications are tested without having knowledge of internal code structure, implementation details and internal paths.
- White-box testing, where internal structure, design and coding of software are tested to verify flow of input-output and to improve design, usability and security.
- Regression testing, checking if previously developed and tested software still performs after a change.
- GUI level testing, the process of ensuring proper functionality of the graphical user interface. GUI testing can require a lot of programming and is time consuming whether manual or automatic. Usually GUI tests are also of the type End to End.

## 1.2 Testing mobile applications

In recent years, the mobile industry has grown exponentially, made up of millions of applications and developers, and billions of devices and users. The rapidly evolving hardware and software platforms, in conjunction with their enormous variety, make mobile application testing a difficult task. The subdivision of the tools for mobile testing has been taken from the article *Continuous, Evolutionary and Large-Scale: A New Perspective for Automated Mobile App Testing* [16], among with some of the tools proposed.

### 1.2.1 Challenges

Compared to desktop or web applications, mobile apps are highly event-driven. They accept inputs from users and the surrounding environment, coming from a big set of sensors and hardware components. These diverse input scenarios are difficult to emulate in a controlled testing environment, making near impossible to test apps against a large set of configurations that are representative of “in-the-wild” conditions.

## Fragmentation

The phenomenon of Fragmentation is one of the largest challenges when testing apps: for an app to be successful, developers must assure the correct functioning of it on a very big set of configurations, due to the diversity of existing devices. The set can be represented by a matrix that combines several variations of operating systems, versions and devices. Fragmentation is more prominent in the case of Android, because the open-source nature has led to a large number of devices with different configurations operating during the same time period. A statistic from 2015 says that there were more than 24,000 different Android devices [17] and we can easily assume that this number has gone up a lot since then. In the case of iOS, the fragmentation is lower, as the number of devices is limited and controlled by Apple, and the market share is more biased toward devices with the latest OS [14].

## Test flakiness

A flaky test is a test that both passes and fails periodically without any code changes, in other words, a test with a non-deterministic behavior. As Jason Palmer said in the article *Test Flakiness – Methods for identifying and dealing with flaky tests*, the real cost of test flakiness is a lack of confidence in tests, which places a team with flaky tests in a position similar to a team with zero tests. Important causes of flakiness are [19] [22]:

- **Inconsistent assertion timing**, when the application state is not consistent between test runs.
- **Reliance on test order**, when test are dependent between each other.
- **Unstable test environment**, like failing to allocate enough resources to satisfy test requirements.
- **The application itself**, like race conditions, uninitialized variables, memory leaks.
- **The underlying hardware and operating system**, for example network failures or disk errors.

### 1.2.2 Automation frameworks

A testing framework is a set of guidelines or rules that helps developers or testers to create and design test cases. Usually these are GUI-level tests for mobile apps through hand-written or recorded scripts, that specify a series of action to be performed on GUI-component and then test for some state information via assertion statement. An automation testing framework is an execution engine to perform automated test with the minimal manual interference. Even if the script's code is highly reusable and this family of techniques is quite efficient due to standardization, there are some drawbacks. While these frameworks typically provide cross-device compatibility of scripts in most cases, the fragmentation problem can appear, for example, with different app states or GUI attributes. Also they usually don't support all the complex user actions that the mobile scenario offers, or interfaces to simulate contextual states. The most problematic thing is that scripts are tightly coupled with the application code and as that rapidly evolves, these become very difficult and time consuming to maintain.

- **UIAutomator** is a mobile testing framework offered by Google as a part of SDK Manager, suitable for cross-app functional UI testing across system and installed apps. It consists of a Java library which has APIs to create functional UI tests as well as an execution engine to run the tests. Appium is a newer version of UIAutomator. [8] [20] [13]

**Pros:** UIAutomator was created specifically for android UI testing and is very easy to use for black-box test cases. The Ui Automator Viewer gives a graphical view of UI components which makes it easy to view components on the device and it enables to work directly with UI elements. Also good documentation and tutorials are available.

**Cons:** It supports only Java and Kotlin languages and the web view is not supported. Working with lists using the API is a complicated process.

- **Espresso** is an open source android UI testing framework developed by Google, designed for whit-box testing. It is one of the most popular automation testing frameworks for Android. [8] [20] [13]

**Pros:** Free and open-source. It offers simple API to quickly write UI test cases. It is customizable, easy to use and fast. The de-

pendency on the Hamcrest library allows to use matchers to test complicated scenarios. Also using Espresso Recorder is possible to create UI tests without writing the code.

**Cons:** It works only on the Android platform and support just Kotlin and Java to write test cases. Application source code is required, meaning this framework is mainly for developers. It doesn't support test automation for contact synchronization and push notification. If the test requires working with Android or another app outside the application being tested, additional tools such as UIAutomator might need to be used. Tests use hard coded GUI's Ids, so if any change a tests code refactoring must be done.

- **Appium** is a popular cross-platform mobile testing automation tool, enabling developers to test native, web and hybrid apps on Android and iOS devices. It offers wide coverage of test automation with device settings, gestural inputs, and different environmental conditions. [8] [20] [13]

**Pros:** It support multiple code languages to write test scripts and developer can use the same code to test Android and iOS devices. It is capable of testing outside the target application. Also it is supported by a large community and since it is open-source, it can run on real devices as well as emulators and simulators.

**Cons:** The initial setup is quite complex and the execution performances are relatively slow. Also it has a limited support for gesture. Tester can execute only one test at a time per Mac while using iOS, giving some scalability issues.

- **Robotium** is an open-source test automation framework for native and hybrid mobile apps, developed to effectively perform grey-box testing. Gray-box testing allows to test an application via source code or apk files. [20] [8]

**Pros:** It is easy and quick to write and run tests. It automatically tests multiple devices simultaneously. Robotium integrates with Gradle, Maven, Ant and offers Robotium Recorder that can record test cases quickly.

**Cons:** Can not simultaneously test multiple applications. It does not offer notification handling of mobile devices and managing inconsistent failures is challenging.

- **Robolectric** is a framework to do fast and reliable unit tests on Android. Unlike traditional emulator-based Android tests, Robolectric tests run inside a sandbox which allows the Android environment to be precisely configured to the desired conditions for each test, isolates each test from its neighbors, and extends the Android framework with test APIs which provide minute control over the Android framework's behavior and visibility of state for assertions. [11] [28]

**Pros:** It doesn't require to run an emulator and can be combined with other test tools like Mockito, Espresso etc. .

**Cons:** Excels at aiding Unit testing, but does not cover all the functionality a real device or emulator can offer. Project setup may require a bit of a tinkering. Also as both Gradle and android build tools are shipping out newer build versions at a fast rate, stable Robolectric versions will sometimes start having problems with the changed build tooling.

- **Ranorex** is a GUI test automation framework provided by Ranorex GmbH. It uses standard programming languages such as VB.NET and C# to automate application. Ranorex has a good Record, Replay and Edit User Performed Actions with the Object-Based Capture&Replay Editor, which means that the Ranorex Recorder offers a simple approach to create automated test steps for web, mobile (native and WAP) and desktop applications. Ranorex can be used for regression testing in continuous build environments to find new software bugs much faster. [25]

**Pros:** It is a multi platform application. Ranorex Recorder with its drag-and-drop interface allows for conducting script-free tests applying keyword-driven testing. It offers image-based test automation with smart object identification technique to automatically detect any change to the UI.

**Cons:** It is an expansive licensed tool, it supports only a few languages and it lacks of macOS support. Also it has a small community .

- **Calabash** is a test automation framework that enables creating and executing acceptance tests for Android and iOS apps without coding skills required. It enables automatic UI interactions within an application. Cucumber, a behavior-driven development language, is used to write test scripts. Calabash can test native, web, and hybrid apps simultaneously on hundreds of iOS and Android devices while getting real-time feedback. [7] [20]

**Pros:** The use of Cucumber language allows anyone without programming knowledge to understand and write test codes. It supports simulators, emulators, and real devices. It can be integrated with CI/CD frameworks such as Jenkins.

**Cons:** Xamarin has stopped active development of Calabash in 2017 owing to the rising popularity of native iOS and Android testing frameworks such as XCUITest and Espresso and the increasing growth of Appium as a cross-platform solution. Also the community support is not great. There is no recorder option to record tests which means no code is generated by the tool.

- **Quantum** is Perfecto's open source, cross-platform test automation framework that support behavior-driven development (Perfecto is a cloud platform for web and mobile app testing). It allows users to quickly build a test using a single, JavaScript framework, Jasmine, for testing native and web apps. Quantum provides testNG integration for execution management, ability to write BDD scripts and a wide range of pre-built commands. [24]

**Pros:** It is a free download from GitHub and can be extended as needed.

**Cons:** Very small community and documentation.

- **Qmetry** is a complete package of Selenium-driven automated end-to-end testing with detailed reporting and trending, mobile quality automation and behavior-driven development. Abstracting the technical implementation away from the operational components, it facilitates streamlined and structured approach. It allows reusable test assets. It can run a single test case against multiple test data sets provided through CSV, XML, JSON, Microsoft Excel or custom database. It



supports unified scripting across different digital platforms and advanced reporting with trending and root causes. [23]

**Pros:** Ready made steps available which is designed on the top of selenium native methods which will help the Testers to write their scenarios very easy. It supports Recorder functionality which will automatically store the steps with suitable locators to generate the sample flow of testing scenarios.

**Cons:** Heavy application due to which it goes on hang up if already another heavy application is running.

### 1.2.3 Record and replay tools

As already stated in 1.1.2, Record and Replay is a way to run tests without programming knowledge. This is done by using a tool that allows to manually perform user interactions, also complex ones, on the graphical user interface and save them as a test. This supports fully automatic regression testing of graphical user interfaces and makes application testing a faster and lighter task, that can be easily automated. However, despite the advantages and ease of use these types of tools afford, they exhibit several limitations. Most of these tools suffer from a trade-off between the timing and accuracy of the recorded events and the portability of recorded test scripts. For context, some R&R based approaches leverage the `/dev/input/event` stream situated in the Linux kernel that underlies Android devices. While this allows for extremely accurate R&R, the scripts are usually coupled to screen dimensions and are agnostic to the actual GUI components with which the script interacts. On the other hand, other R&R approaches may use higher-level representations of user actions, such as information regarding the GUI components upon which a user acts. While this type of approach may offer more flexibility in easily recording test cases, it is limited in the accuracy and timing of events. An ideal R&R approach would offer the best of both extremes, with highly accurate and portable scripts, suitable for recording test cases or collecting crowdsourced data. R&R requires oracles that need to be defined by developers, by manually inserting assertions in the recorded scripts or using tool wizards.

- **RERAN** is a record and replay tool for the smartphone Android operating system. At a high level, it captures the input events sent from

the phone to the operating system of a user session, and then allows the sequence of events to be sent into the phone programatically. At low level, RERAN consists of three steps. First events are recorded with the Android SDK's `getevent` tool, then the output is sent into RERAN's Translate program. The output from the Translate program is then sent into RERAN's Replay program. The Replay program sends the events back into the event stream of the phone.[26]

**Pros:** It is able to replay 86 out of the Top-100 free apps on Google Play. Also it can replay complex GUI gestures and some device sensors.

**Cons:** Replay does not always work 100% specially with sessions over 10 minutes.

- **VALERA** stands for Versatile-yet-lightweight Record-and-replay tool for Android. It uses a novel technique named sensor-oriented replay (recording and replaying sensor and network input, event schedules, and inter-app communication via intents) to achieve high accuracy and low overhead. [32]

**Pros:** Versatile and lightweight tool.

**Cons:** Almost inexistent documentation and community. Works only for Android.

- **Mosaic** is a cross-platform (based on Python), timing-accurate record and replay tool for Android-based mobile devices. Mosaic enables cross-platform record and replay through a novel virtual screen abstraction. User interactions are translated from a physical device into a platform-agnostic intermediate representation before translation to a target system. The intermediate representation is human-readable, which allows Mosaic users to modify previously recorded traces or even synthesize their own user interactive sessions from scratch. [18]

**Pros:** Mosaic's unique virtualization scheme abstracts away the hardware and software complexity related to user input to replay user interactions across a variety of different mobile devices.

**Cons:** Almost inexistent documentation and community.

- **Barista** is built on top of Espresso, it provides a simple and discoverable API, removing most of the boilerplate and verbosity of common Espresso tasks.[4]

**Pros:** Improved stability through auto retry and auto scroll. Very good documentation, easy to use.

**Cons:** Idling resources still have to be implemented manually.

- **Sikuli** automates anything that can be seen on the screen of a desktop computer running Windows, Mac or some Linux/Unix. It uses image recognition powered by OpenCV to identify GUI components. This is handy in cases when there is no easy access to a GUI's internals or the source code of the application. Though Sikuli is currently not available on any mobile device, it can be used with the respective emulators on a desktop computer or based on VNC solutions.[30]

**Pros:** Easy and fast black-box testing.

**Cons:** The image recognition is dependant from screenshots resolution, that are used to recognize GUI elements. So taking screenshot in an environment and running tests in another environment cause tests flakiness.

- **Robotium Recorder** is based on the open source Robotium test automation framework. It allows to easily create test cases for android app and re-run them later.[27]

**Pros:** Installation is simple, it works as a plugin for either Eclipse or Android Studio. Easy tests recording.

**Cons:** It only supports a single app at a time and it is for Android only.

- **Espresso Recorder** lets create UI tests without writing any code. By recording a test scenario, it is possible to record interactions with a device and add assertions to verify UI elements in particular snapshots of the app to be tested. Espresso Test Recorder then takes the saved recording and automatically generates a corresponding UI test that can be run to test the app. Espresso Test Recorder writes tests based on the Espresso Testing framework.[5]

**Pros:** It allows to record both interactions and assertions, making possible to verify the state of specific widgets. It also supports multiple assertions.

**Cons:** It supports only basic and simple assertions. Test cases with animations or asynchronous operations cannot be tested.

#### 1.2.4 Automated test input generation techniques

This family of techniques automated the process of input generation to dramatically ease the burden on developers and testers. Such approaches are typically designed with a particular goal, or set of goals in mind, such as achieving high code coverage, uncovering the largest number of bugs, reducing the length of testing scenarios or generating test scenarios that mimic typical use cases of an app. AIG are classified into three categories: random-based input generation, systematic input generation and model-based input generation. Additionally, other input generation approaches have been explored including search-based and symbolic input generation. The specific limitations of these tools fail to address broader challenges, including flaky tests, fragmentation, limited support for diverse testing goals, and inadequate developer feedback mechanisms.

#### 1.2.5 Bug and error reporting/monitoring tools

They are an integral part of many mobile testing workflows. There are two types of tools in this category: tool for bug reporting (also known as issue trackers), and tools for monitoring crashes and resource consumption at runtime. Classic issue trackers only allow reporters to describe the bugs using textual reports and by posting additional files such as screenshots. In the case of tools for monitoring, if developers do not choose to include third-party error monitoring in their application (or employ a crowd-based approach), typically, the only user-feedback or in-field bug reports they receive are from user reviews or limited automated crash reports. Unfortunately, many user reviews or stack traces without context are unhelpful to developers, as they do not adequately describe issues with an application to the point where the problem can be reproduced and fixed.



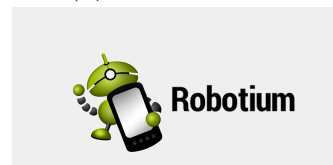
(a) Espresso logo.



(b) Appium logo.



(c) Sikuli logo.



(d) Robotium logo.



(e) Barista logo.

Figure 1.3: Some of the tools logos.

### **1.2.6 Mobile testing services**

Due to the sheer number of different technical challenges associated with automated input generation, and the typically high time-cost of manually writing or recording test scripts for mobile apps, it has become a popular alternative to outsource testing services. Some of the most popular services are: Perfecto Cloud Lab [21], AWS Device Farm [3] and Firebase Test Lab [9].

#### **Crowd-sourced functional testing**

Crowd testing involves a large number of testers based in different locations, both experts and non-experts. With a broader set of people conducting tests in a diverse range of conditions and in a real life scenario, it's more likely that they will spot any bugs. Tester are then compensated for the number of true bugs discovered.

#### **Usability testing**

In a usability-testing session, a researcher (called a “facilitator” or a “moderator”) asks a participant to perform tasks, usually using one or more specific user interfaces. While the participant completes each task, the researcher observes the participant's behavior and listens for feedback, aiming to the measure the UX/UI design of an app with a focus on ease of use and intuitiveness.

#### **Security testing**

Security Testing aims to uncover any design flaws in an app that might compromise security. It can be Grey Box, analyzing application source code to find vulnerabilities, or it can be Black Box, analyzing the application downloaded from a store.

#### **Localization testing**

Localization Testing tries to ensure that an app will function properly in different geographic regions with different languages across the world.

### **1.2.7 Device streaming tools**

Tools for device streaming facilitate developer when testing mobile application by mirroring a connected device to their PC or accessing devices remotely over the internet.

## **1.3 Continuous integration, delivery and deployment**

### **1.3.1 Continuous Integration**

Continuous integration is a practice where developer merge their code changes to the main branch of a repository as often as possible. The changes are then validated by creating a build and running automated tests against it. This avoid integration challenges when comes the moment of merging development branch to the main branch after many commits. Continuous integration puts a great emphasis on testing automation to check that the application is not broken whenever new commits are integrated into the main branch. Another advantage of doing continuous integration is that running a complete test suite for each small change can bring to a better bug detection and resolution.

### **1.3.2 Continuous Delivery**

Continuous delivery automatically deploys all code changes to a testing and/or production environment after the build stage. On top of automated testing there is an automated release process.

### **1.3.3 Continuous Deployment**

In continuous deployment, every change that passes all stages of the production pipeline is released directly to the customers without human intervention: only failed test prevent a new change to be deployed.

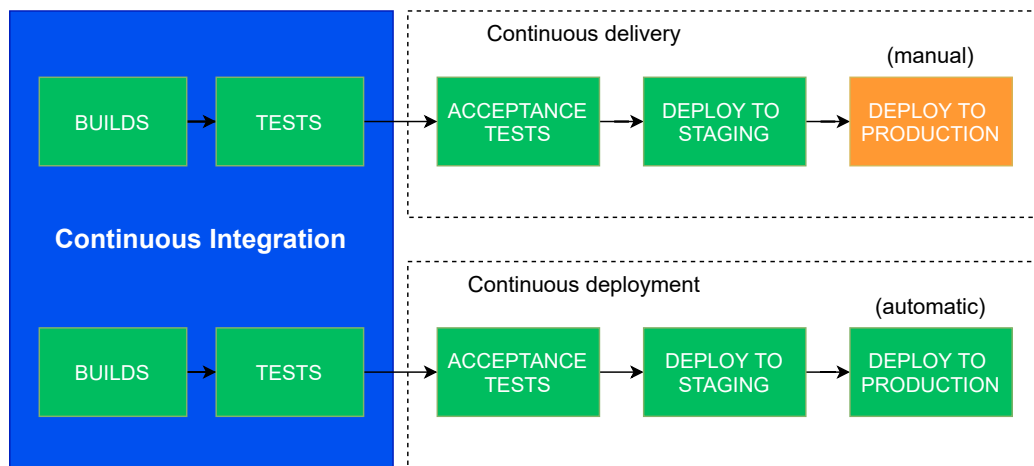


Figure 1.4: The structure of a common CI/CD pipeline and the difference between continuous delivery and continuous deployment.



## Part II

# Evaluation of tools for mobile testing

Studying and evaluating testing frameworks for mobile applications, in a more refined way, is the main goal of this thesis. In particular, the focus is on better describing the modes of operation, the required setup and its difficulties and the issues that may occur during the execution of tests.

# Chapter 2

## Selected Instruments

From the crowded selection of testing tools that the market offers have been chosen three tools. These will be used to conduct GUI level end-to-end tests. The first two tools are from the automation framework group, in particular, the first one is for white-box testing and the second for black-box testing. The last tool is from the record and replay tools and is for black-box testing.

### 2.1 Android Espresso

The choice of Android Espresso is dictated from mainly three reason:

1. Created by Google, it is a native framework for Android automated testing, so it is well maintained, compatible with nearly all Android versions, and it has a big community support.
2. It is stable and it has a simple workflow with fast feedback.
3. Effortless setup and integration with Android Studio, the native Android development environment.

#### 2.1.1 Architecture Details

Espresso provides a large number of classes to test the user interface and the user interaction of an android application. They can be grouped into five categories: [6]

- **JUnit runner**, to run the espresso test cases written in JUnit3 and JUnit4 style test cases. It is specific to android applications and it transparently handles loading the espresso test cases and the application under test both in actual device or emulator, executes the test cases and reports the result of the test cases.
- **JUnit rules**, in particular `ActivityTestRule` to launch an android activity before executing the test cases.
- **View matchers**, to match and find UI elements/views in an android activity screen's view hierarchy.
- **View actions**, to invoke the different actions on the selected/matched view.
- **View assertions**, to assert the matched view is what is expected.

The workflow of the framework can be resumed as:

1. `AndroidJUnit4` will prepare the environment to run all the test cases: it starts the emulator, installs the application and makes sure the application to be tested is in a ready state. It will run the test cases and report the results.
2. The activity/activities to be tested will be started by the Android JUnit runner using the `ActivityTestRule`.
3. Every test case needs a minimum of single `onView` or `onDate` method invocation to match and find the desired view.
4. `onView` and `onDate` return a `ViewInteraction` object, that can either invoke an action or check an assertion.

### 2.1.2 Environment Setup

To avoid test flakiness, it's highly recommended to turn off systems animations on the virtual or physical devices used for testing. Then Espresso dependencies must be added to the project using Gradle: for a complex project and especially for developers that don't know well the application code, this step might require some effort in order to succeed, because incompatibility errors may arise or the required code can be put into the wrong

`build.gradle` file. To run the tests the application code must be available and must compile without error. This is another problem because sometimes the code present on the online repository may be incomplete or with the wrong dependencies.

### 2.1.3 Tool Capabilities

Espresso is well integrated with Android Studio IDE. Tests can be run in sequence or one by one by clicking on the icon that appears at the left of the function annotated with `@Test` as shown in Figure 2.1 and Figure 2.2.

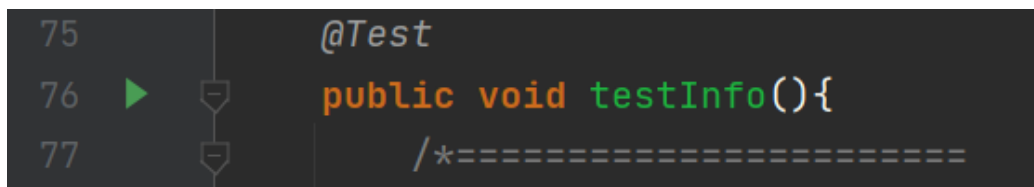


Figure 2.1: Espresso run test button.

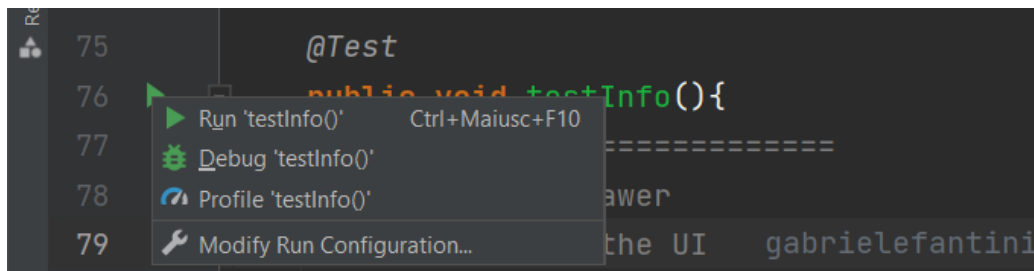


Figure 2.2: Espresso run test button expanded.

Each time a test needs to be run a Gradle build executes, which takes between 2 to 4 minutes. An emulator or a real device must be available to run the tests. It's possible to execute only 1 test at a time on each device. Tests results are displayed in a dedicated section of the Android Studio IDE ( Figure 2.3 ). If a test failed the cause of the error is quite comprehensive ( Figure 2.4 ).

Tests can be executed also via `./gradlew connectedAndroidTest` command. This allows to automate test execution and produce a Gradle tests report.



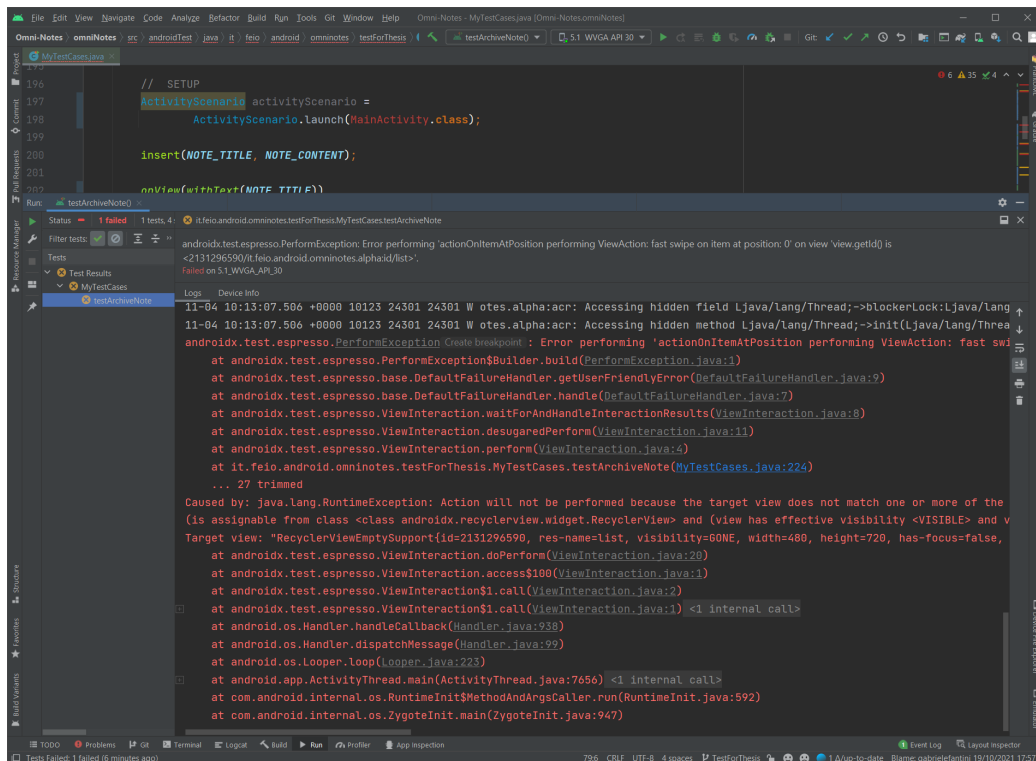


Figure 2.4: Espresso failed test.

## 2.2 Appium

Appium has been chosen because of:

1. It is well documented and widely spread across the developers community.
2. It supports cross platform test cases.
3. Because Appium uses JSON Wire Protocol for client/server communication, it allows writing clients (and so test scripts) in many different languages.

### 2.2.1 Architecture Details

Appium uses vendor-provided automation frameworks (Apple Instruments, XCTest, UIAutomation for iOS and UiAutomator/UiAutomator2 for Android). Those are wrapped in one API called **WebDriver API**, which specifies

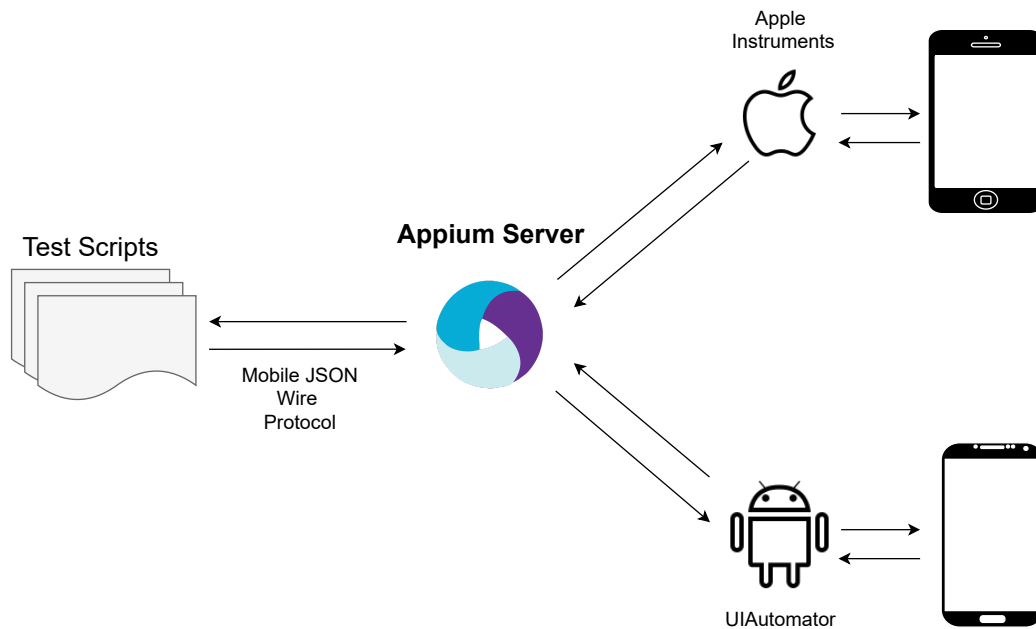


Figure 2.5: Appium architecture details.

a client-server protocol. With this client-server architecture, a client (with



test scripts) written in any language can be used to send the appropriate HTTP request to the server. [2]

## 2.2.2 Environment Setup

Appium setup is very tricky. First of all Appium server must be installed, that runs on Node.js, second it must be configured to connect to the android device (simulated or real). Then Appium Inspector must be installed, a tool to rapidly inspect the GUI and record tests, and configured to connect to the Appium server. Last but not least one of the available Appium client libraries must be chosen, depending on the programming language wanted to be used.

## 2.2.3 Tool Capabilities

### Appium Server GUI

Is a graphical interface for the Appium Server. It allows to set options,

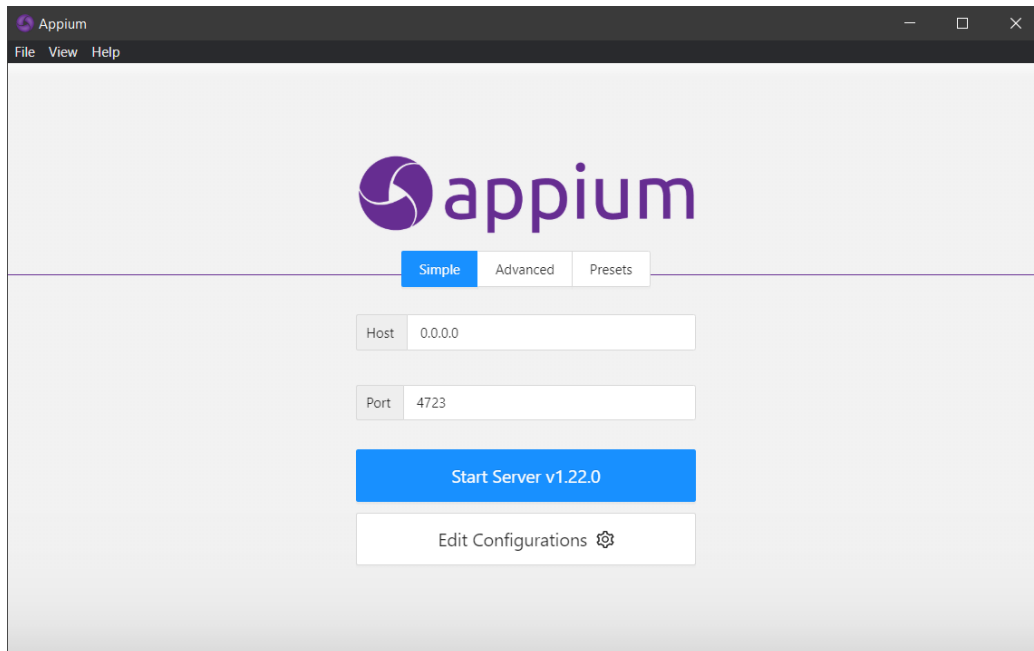


Figure 2.6: Appium Server GUI.

start/stop the server, see logs, etc... Also, it is not needed to use Node/NPM to install Appium, as the Node runtime comes bundled with Appium Desktop (Figure 2.6)

## Appium Inspector

Is basically just an Appium client (like WebdriverIO, Appium's Java client, Appium's Python client, etc...) with a user interface. There's an interface for specifying which Appium server to use, which capabilities to set (Figure 2.7),

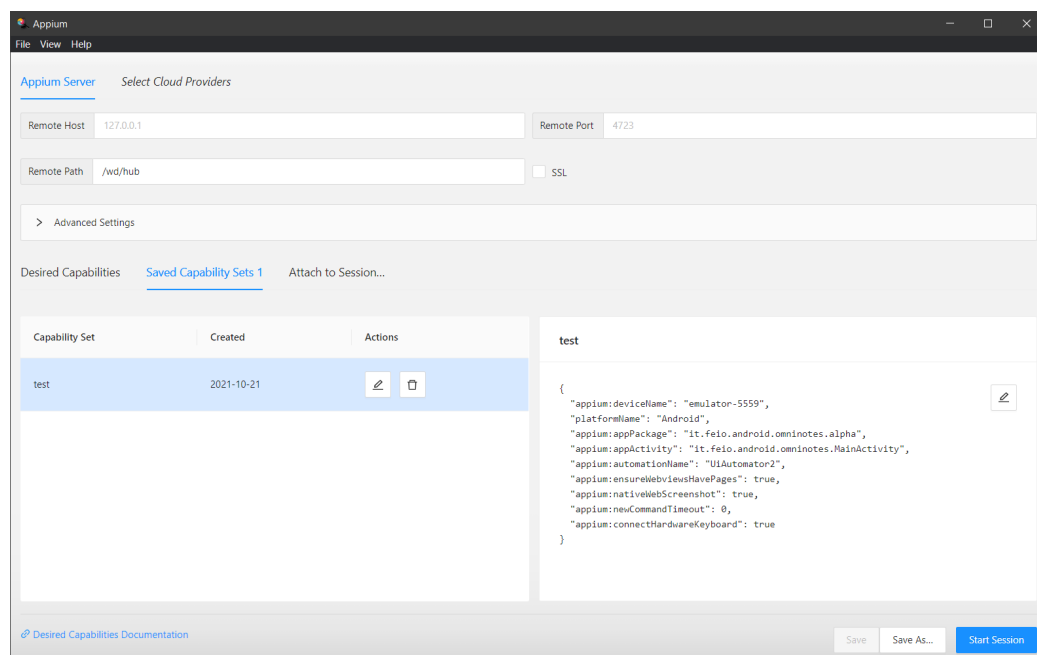


Figure 2.7: Start a session in Appium Inspector.

and then interacting with elements and other Appium commands once started a session. This allows to easily inspect the GUI of the application (Figure 2.8) and record a test case using the record functionality (Figure 2.9) which automatically generates the test script based on the client library selected.

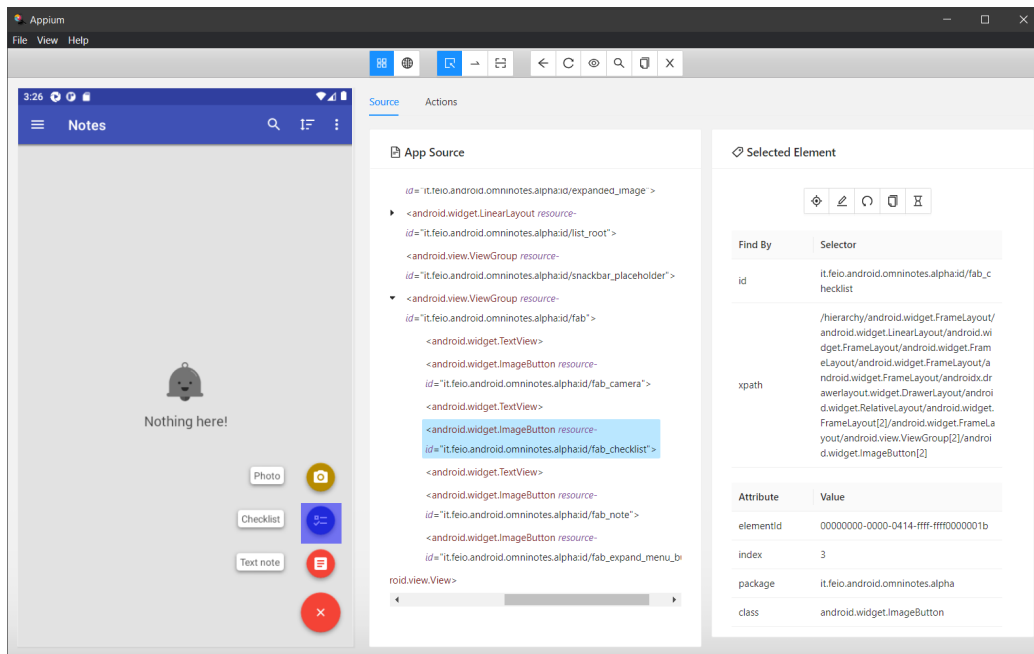


Figure 2.8: Appium Inspector connected.

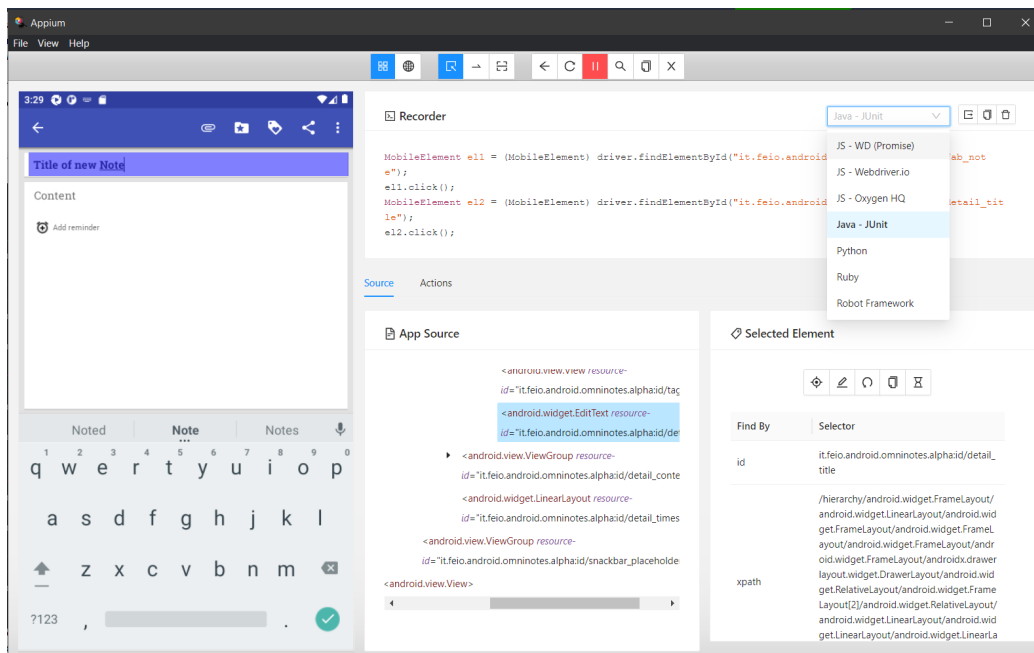


Figure 2.9: Appium Inspector test recording functionality.

## **Appium Client Library**

Once the tests have been recorded using Appium Inspector, they can be saved and run using one of the available client libraries for Appium. In this case, it has been used the Java Client Library, IntelliJ Idea Community IDE, and the TestNG Framework to make assertions and organize tests.

## **2.3 Sikuli**

Sikuli is a good pick from the record and replay tools because:

1. Very fast set up with nearly zero configuration.
2. Fast and easy test recording, with little coding knowledge required.
3. Cross platform tests.

### **2.3.1 Architecture Details**

Sikuli is written in Java and is used as a library that exposes some APIs.

### **2.3.2 Environment Setup**

Sikuli is the fastest option. It is as easy as downloading the Sikuli portable jar file and launching it. When running test scripts an Android Emulator with the app to be tested must be available and running. To save and run tests in a more organized way, it has been used the Sikuli Java Library, the TestNG framework and IntelliJ Idea Community for the IDE.

### **2.3.3 Tool Capabilities**

The Sikuli portable jar offers a graphical interface that allows to capture a portion of the screen and save them as .png images (Figure 2.10).

Then it shows them as thumbnails on the scripts section, which gives the possibility to run tests as Python scripts on the go (Figure 2.11), performing an action on items captured and making assertions. Sadly Sikuli does not support well (or at all) complex user interaction such as swipe, pinch, scroll down, etc... This is a huge limiting factor to test case complexity and completeness.

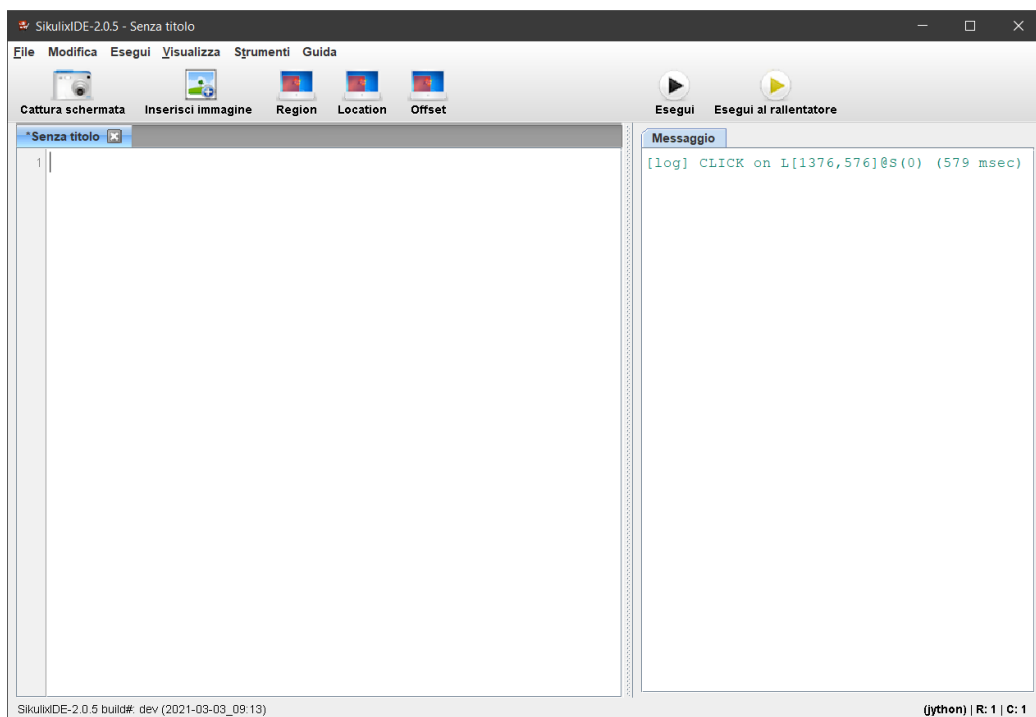


Figure 2.10: Sikuli IDE Home.

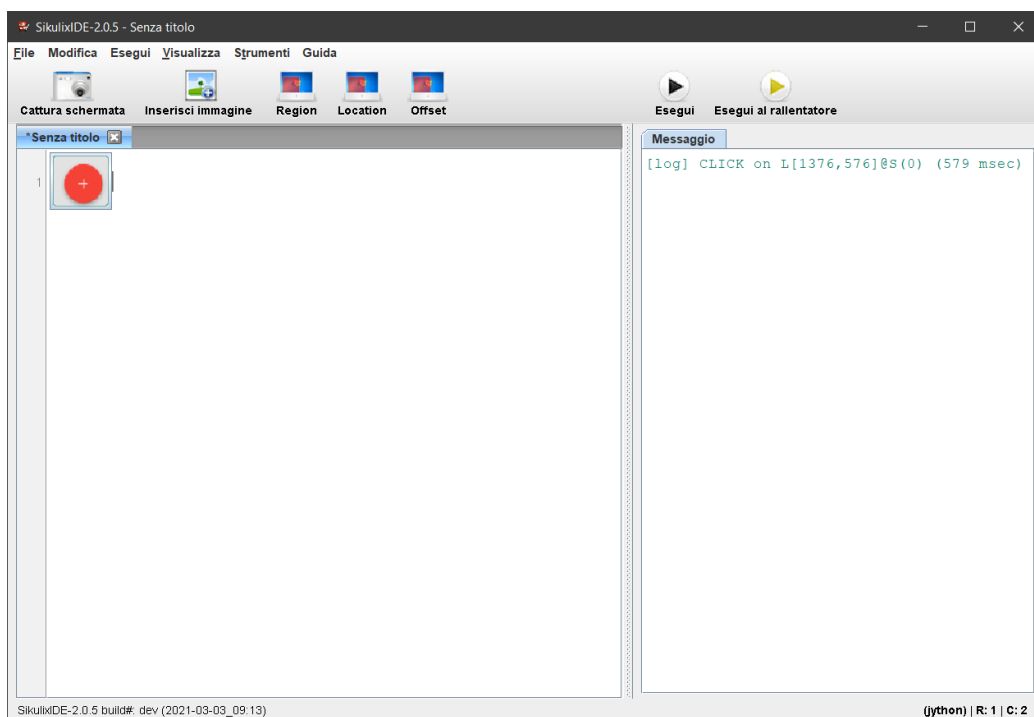


Figure 2.11: Sikuli IDE with captured element.

The thumbnails can be shown also as file names, so it is possible to copy and use them in the separate Java code (Figure 2.12).

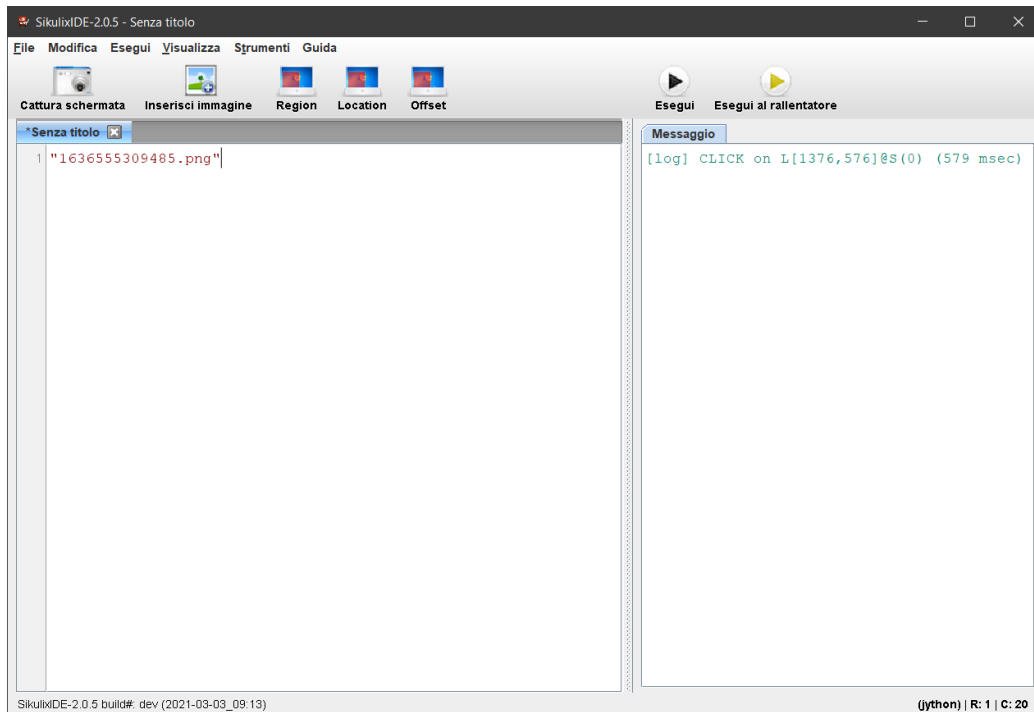


Figure 2.12: Sikuli IDE with captured element as file name.

## Chapter 3

# Experimental Subject

The application that has been chosen to run the tests is OmniNotes. It is an open-source application, with code available on the GitHub Repository. The project structure is simple enough to be understood in a short times and the code compiles with all the dependencies without errors, making it a very good candidate.

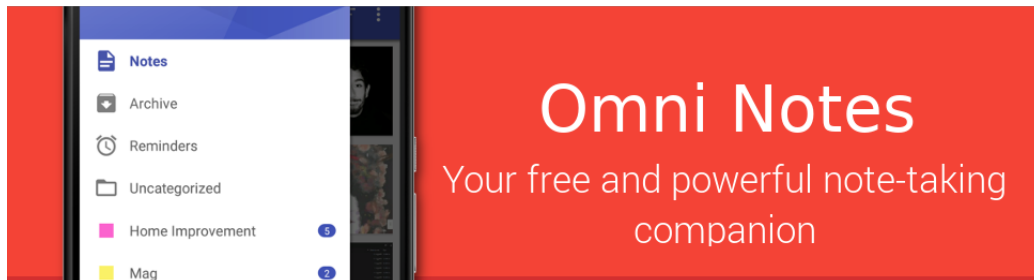


Figure 3.1: OmniNotes.



# Chapter 4

## Procedure

### 4.1 Test Cases

Five test cases, that represent some of the main user actions, have been chosen.

1. Navigate through the app to the info section (Figure 4.1).
2. Insert one new note (Figure 4.2).
3. Archive an existing note (Figure 4.3).
4. Search for a note (Figure 4.4).
5. Delete a note and empty the trash (Figure 4.5).

The code of the tests is available at **Test Repository**.

#### 4.1.1 Espresso

A comprehensive tests summary is shown in Figure 4.6.

#### 4.1.2 Appium

A comprehensive tests summary is shown in Figure 4.7.

#### 4.1.3 Sikuli

A comprehensive tests summary is shown in Figure 4.8.

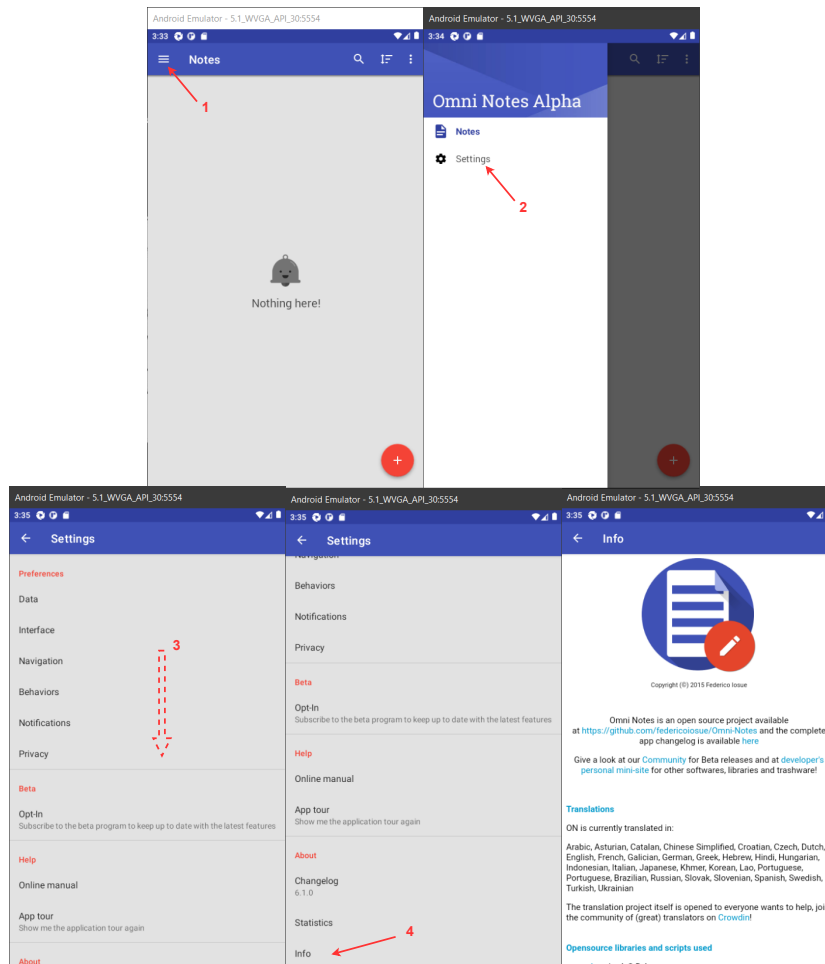


Figure 4.1: Navigate through the app to the info section.

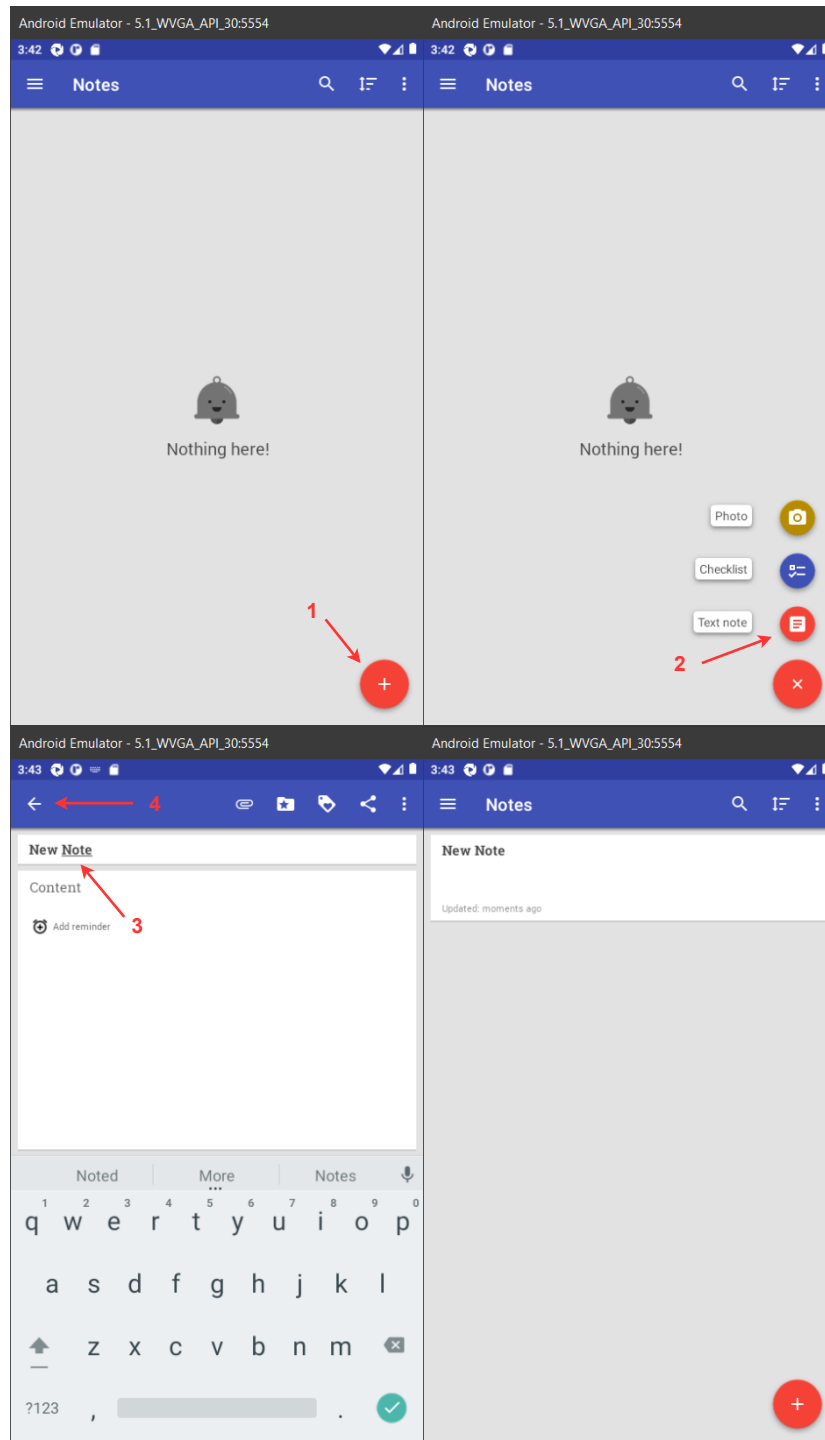


Figure 4.2: Insert new note.

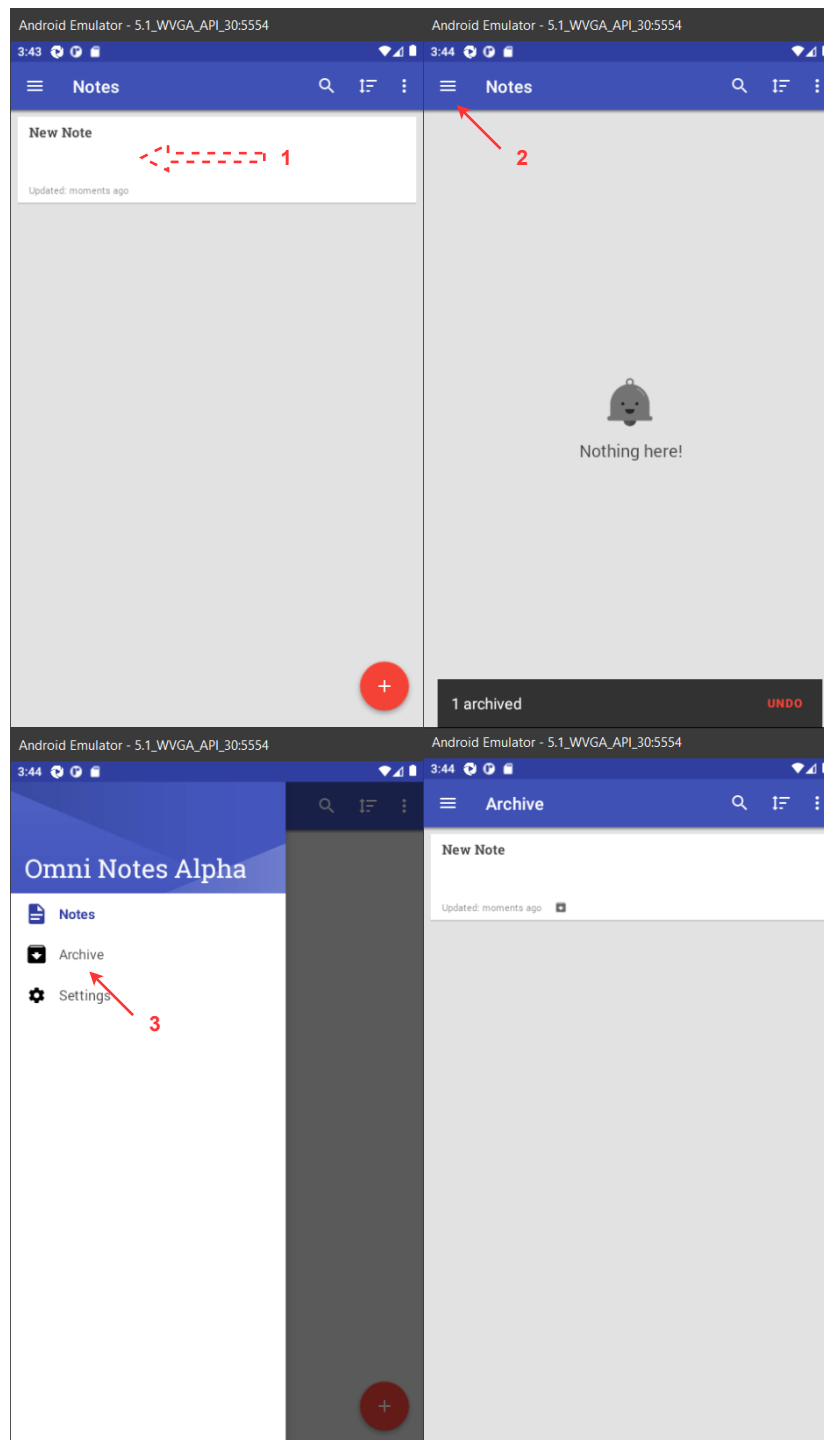


Figure 4.3: Archive a note.

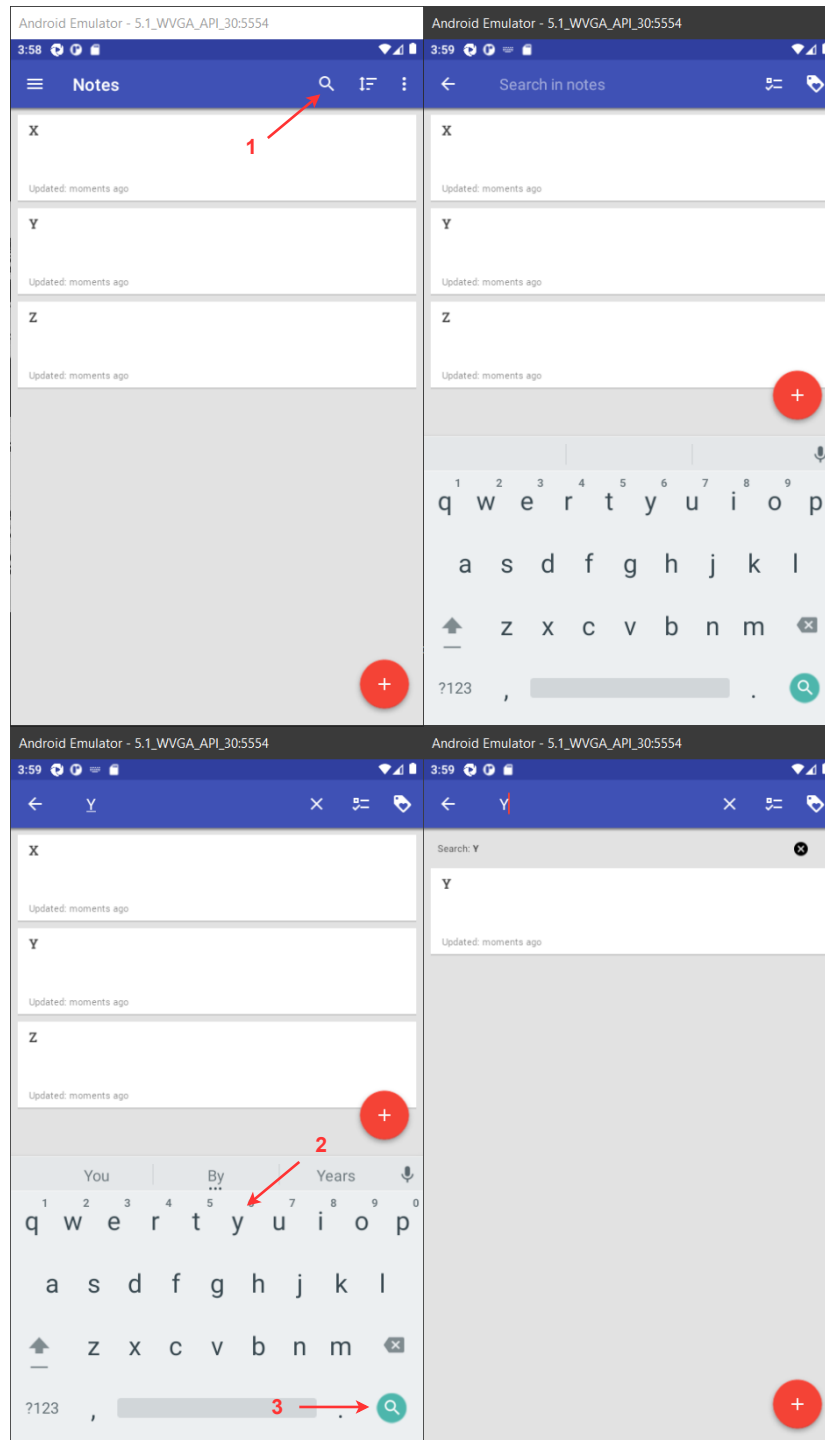


Figure 4.4: Search a note.

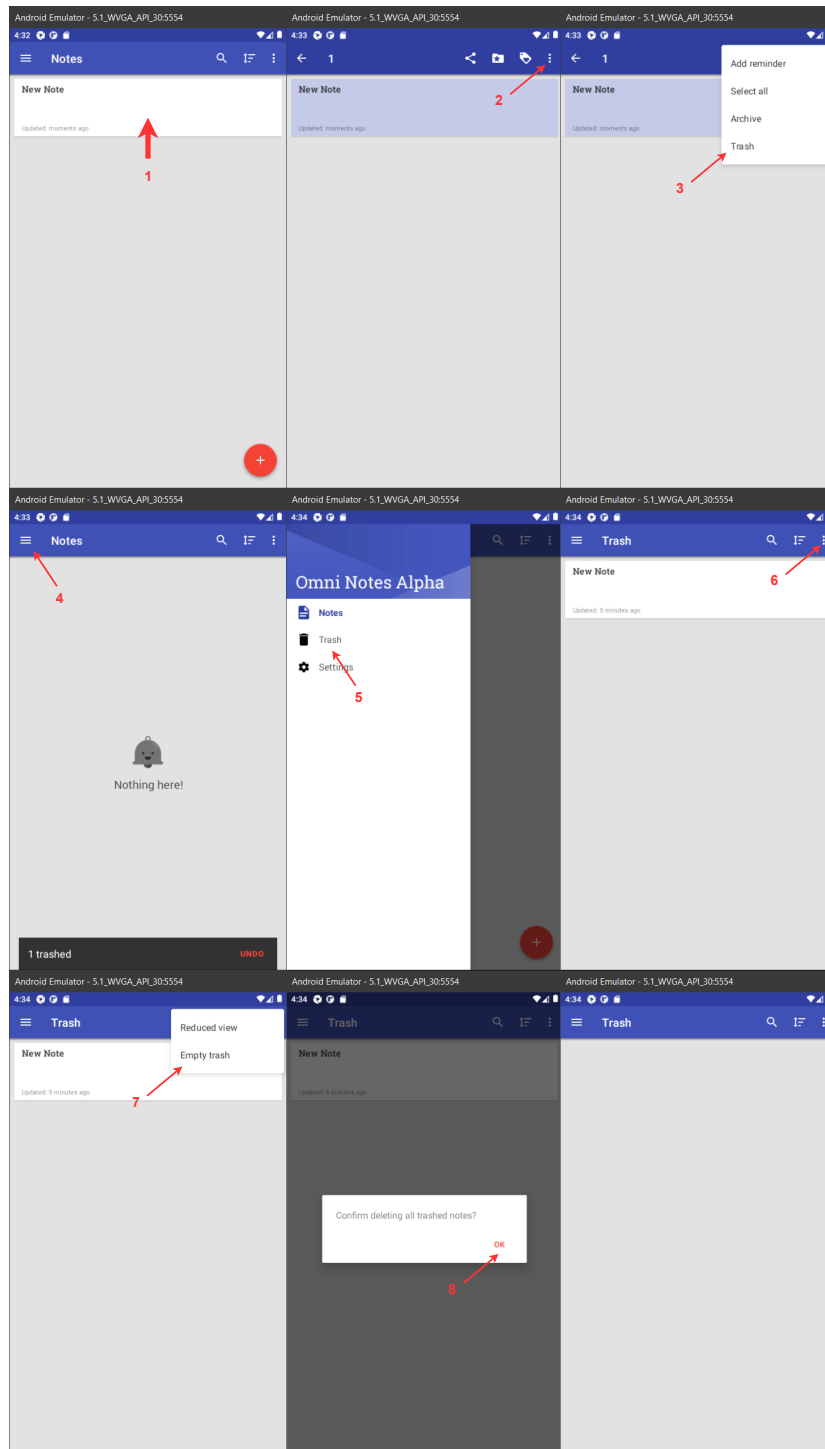


Figure 4.5: Delete a note and empty the trash.


Espresso		
Test	Test Result	Problems Encountered
1. Navigate through the app to the info section.		
2. Insert one new note.		
3. Archive an existing note.		This test include a swipe action. Complex user action like this one are replicated without problems by Espresso, that offers a rich set of API. The swipe action introduces a implicit delay in the test flow and must be handled with a "waitForText()" to avoid test flakiness
4. Search for a note.		
5. Delete a note and empty the trash.		Espresso handles well also the lists, offering a set of API for the Re-cyclerView. In this case it has been used "actionOnItemAtPosition()" that allows to easily select an item from a list, avoiding ambiguities that otherwise may arise if searching an item from layout ID or textcontent.

Figure 4.6: Espresso tests results.





Appium		
Test	Test Result	Problems Encountered
1. Navigate through the app to the info section.	Ok, but longer lists may cause problems	The scroll down action is handled manually with a TouchAction that performs a scroll down, moving from the bottom of the screen to the top. This is not so precise and can cause test flakiness.
2. Insert one new note.		
3. Archive an existing note.	Ok, but can cause test flakiness with different screen resolutions.	The swipe action is handled with a manual swipe from the right to the left of the screen. This is quite verbose and both the swipe action and the scroll down of the previous test have been manually generated, because the Appium Layout Inspector does not generate working code
4. Search for a note.		
5. Delete a note and empty the trash.	Possible ambiguities can cause test flakiness.	Lists are handled not so well: using the "findElement(By.xpath())" it is possible to encounter ambiguities that will lead to test failure.

Figure 4.7: Appium tests results.










Sikuli		
Test	Test Result	Problems Encountered
1. Navigate through the app to the info section.		This test case can not be reproduced due to the absence of scrolling down interaction support.
2. Insert one new note.		
3. Archive an existing note.	Possible test flakiness caused by replacing swipe left with drag and drop.	The swipe left action has been simulated with a "dragDrop" from right to left. Currently complex mobile user interaction are not supported and must be simulated with the available API.
4. Search for a note.		
5. Delete a note and empty the trash.		

Figure 4.8: Sikuli tests results.

# Chapter 5

## Tools Evaluation

### 5.1 Final considerations

#### 5.1.1 Espresso

- **The Ease of Use** is good. Espresso APIs are simple to learn and well documented. The layout IDs can be easily discovered using the Layout Inspector functionality of Android Studio.
- **Test Flakiness** is mainly caused by asynchronous events that can be present during a test, such as loading data from an external source, or by animation embedded into the application and that can't be removed via animation settings on Android. After all, tests with Espresso have good reliability.
- **Test Execution Time** is quite alarming. Even a small modification of the test code requires some minutes to rebuild, making debugging of test scripts an annoying task and possibly affecting scalability.
- **Test Scalability** is not good if Espresso is taken as a standalone solution: after each test the application state can't be reset, so it is difficult to make each test independent from the others. Having the layout IDs coupled with the test scripts is not a big problem if both the test and the application code are in the same repository because, when a modification occurs in the code, a correct refactor is run across the entire project using the IDE functionality.

## Conclusion

Espresso is a great tool for developers that know well the application code and want to write E2E tests simply and rapidly. The integration on a pipeline of CI/CD can be a viable solution with the addition of other tools that can manage application state between tests (like Android Test Orchestrator), and the ability to run multiple tests at once, which means having the capability of run multiple Gradle builds simultaneously and then having multiple emulated devices available.

### 5.1.2 Appium

- Appium is **easy to use**. With Appium Inspector tests are recorded quickly and easily. In the situation examined the last version of Appium Inspector generates an obsolete code of the java client library, so extra steps were needed to adapt the code. In an ideal scenario, tests are recorded and then run with near-zero effort. In this specific case test results are handled using the IntelliJ IDE with the TestNG framework, and displayed in a specific section of the IDE. If a test failed the cause of the error is not very comprehensive and needs a detailed investigation.
- **Test Flakiness** is caused mainly by animation delays and can be fixed by setting explicit waits on the elements that give problems or an implicit wait on the driver. Running tests when Appium Inspector is connected can cause some unexpected errors. Sometimes running one test multiple times does not generate the same results. In conclusion, Appium is less reliable than Espresso, but it remains a solution with good reliability.
- **Test Execution Time** is less than Espresso, due to the ability to run tests without recompiling the application code each time: it takes just the compilation time of the Appium client code.
- **Test Scalability** is very good: after each test the application state is reset, avoiding conflicts and promoting tests isolation. The fact that the application does not need to recompile each time gives the possibility to run a lot of tests in much less time. Tests can also be parallelized using one of four strategies:

1. Running multiple Appium servers, and sending one session to each server.
2. Running one Appium server, and sending multiple sessions to it.
3. Running one or more Appium servers behind the Selenium Grid Hub, and sending all sessions to the Grid Hub.
4. Leveraging a cloud provider (which itself is running many Appium servers, most likely behind some single gateway).

Tests with complex user interaction require writing some code manually and may break just with minor application changes. Also, long lists with similar elements can be difficult to handle.

## Conclusion

Appium is a brilliant solution to run E2E tests of an application, also without knowing its code. It is a good candidate to build a test automation suite due to its abilities of scalability and tests isolation. The main problem that may limit the effectiveness of the testing tool is random test flakiness and the difficulties to replicate complex user interactions. The first can be alleviated by running the Appium solution in a stable and controlled environment, the second standardizing the code.

### 5.1.3 Sikuli

- **Ease of Use** Sikuli is very easy to use, due to its fast setup and intuitiveness of test cases recording.
- **Test Flakiness:** Sikuli uses image recognition powered by OpenCV to identify GUI components, so it is robust to test flakiness caused by animations or GUI random delays but it requires the complete control of the current OS screen and input peripherals such as mouse and keyboard.
- **Test Scalability** is not good at all. First of all application state can't be reset between tests, making tests dependent on each other. Second, Sikuli requires a full desktop environment to run. This yields to running multiple tests in parallel a heavy computational task.

## **Conclusion**

Sikuli is a good tool for running E2E testing without the need to access application code or to a GUI's internals. It's an easy tool with a fast setup and good versatility, but the main drawbacks are the inability to reproduce complex user interaction, the absence of test isolation and the computational cost of running tests on parallel with multiple OS instances.

## Part III

# Tools implementation in a CI/CD pipeline

# Chapter 6

## Tools for CI/CD

Currently, there is a vast amount of tools for CI/CD in the market. From this crowded place have been examined only two tools:

1. Jenkins, an open-source automation server in which the central build and continuous integration process take place. It is a self-contained Java-based program with packages for Windows, macOS, and other Unix-like operating systems. With hundreds of plugins available, Jenkins supports building, deploying, and automating software development projects. The license is free and it has an active community behind it. [12]
2. GitHub Action, that enables the creation of custom software development lifecycle workflows directly in a GitHub repository. These workflows are made out of different tasks so-called "actions" that can be run automatically on certain events, allowing the creation of a complete CI/CD pipeline. Actions are free for every open-source repository and include 2000 free build minutes per month for all private repositories, which is comparable with most CI/CD free plans. [31]

Jenkins supports declarative syntax for the pipeline, which is very similar to the GitHub Actions syntax. It follows that, once a CI/CD pipeline has been defined in one of the two tools, it's nearly painless to switch to the other tool. For this reason, it has been implemented only a solution with the GitHub Actions.

## 6.1 GitHub Actions in depth

GitHub Actions help automate tasks within the software development cycle. They are made of multiple components that work together to run jobs. (Figure 6.1)

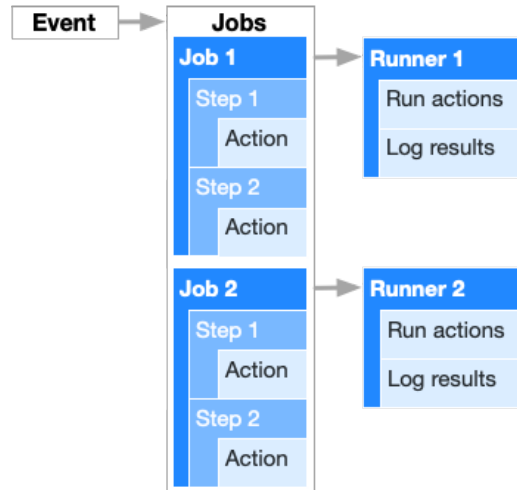


Figure 6.1: GitHub Actions components.[31]

### 6.1.1 Workflow

The workflow is an automated procedure added into a GitHub repository. It is made up of one or more jobs and can be scheduled or triggered by an event. Workflows can be made reusable, avoiding duplication and making them easy to maintain. [31]

### 6.1.2 Events

An event is a specific activity that triggers a workflow. For example, it can originate from GitHub after a push of a new commit to a repository, when an issue or pull request is created, etc... Also, it is possible to trigger a workflow using the repository dispatch webhook when an external event occurs.[31]



### **6.1.3 Jobs**

A job is a set of steps that execute on the same runner. By default, a workflow with multiple jobs will run those jobs in parallel but can be configured to run sequentially.[31]

### **6.1.4 Steps**

A step is an individual task that can run commands in a job. A step can be either an action or a shell command. Each step in a job executes on the same runner, allowing the actions in that job to share data with each other.[31]

### **6.1.5 Actions**

Actions are standalone commands that are combined into steps to create a job. Actions are the smallest portable building block of a workflow. They can be custom made, or created by the GitHub community. [31]

### **6.1.6 Runners**

A runner is a server that has the GitHub Actions runner application installed. It can be hosted by GitHub, or can be hosted in a dedicated server. A runner listens for available jobs, runs one job at a time, and reports the progress, logs, and results back to GitHub. GitHub-hosted runners are based on Ubuntu Linux, Microsoft Windows, and macOS, and each job in a workflow runs in a fresh virtual environment. [31]

# Chapter 7

## Implementing a CI/CD pipeline

Three different pipelines have been implemented, each one using one of the tools evaluated before. All of the three pipelines can be combined together to reach different testing objectives.

### 7.1 Android Emulator Runner

All the pipelines use the GitHub Action named `reactivecircus/android-emulator-runner`, that installs, configures and runs hardware-accelerated Android Emulators on macOS virtual machines. The old ARM-based emulators were slow and are no longer supported by Google. The modern Intel Atom (x86 and x86\_64) emulators require hardware acceleration (HAXM on Mac & Windows, QEMU on Linux) from the host to run fast. This presents a challenge on CI as to be able to run hardware accelerated emulators within a docker container, KVM must be supported by the host VM which isn't the case for cloud-based CI providers due to infrastructural limits. Fortunately, the macOS VM provided by GitHub Actions has HAXM installed so it is possible to create a new AVD instance, launch an emulator with hardware acceleration, and run Android tests directly on the VM. This is also possible to achieve on a self-hosted Linux runner, but it will need to be on a compatible instance that allows enabling KVM (for example AWS EC2 Bare Metal instances). In conclusion, to run fast and smoothly, the action requires a GitHub MacOS runner. [1] The action does the following steps:

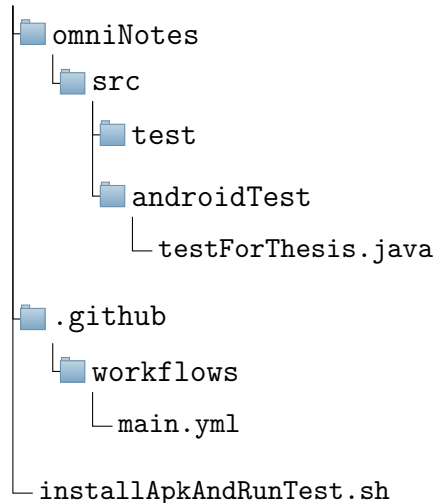
1. Installs and updates the required Android SDK components.

2. Creates a new instance of AVD (Android Virtual Device) with the provided configurations.
3. Launches a new Emulator with the provided configurations.
4. Waits until the Emulator is booted and ready to use.
5. Runs a custom script provided by the user once the Emulator is ready.
6. Kills the Emulator and terminates the action once the script is finished.

## 7.2 CI/CD pipeline using Espresso

All the code is contained in one repository: the code is available at the [GitHub Repository](#). The repository structure is illustrated in the figure. The overall system structure is represented in Figure 7.1.

EspressoContinuousIntegration



### 7.2.1 Repository Workflow

The repository workflow is defined in the `main.yml` file. After each push event a job is started. The job is executed on a runner with the latest version of the Ubuntu operating system and it is made of 5 steps:

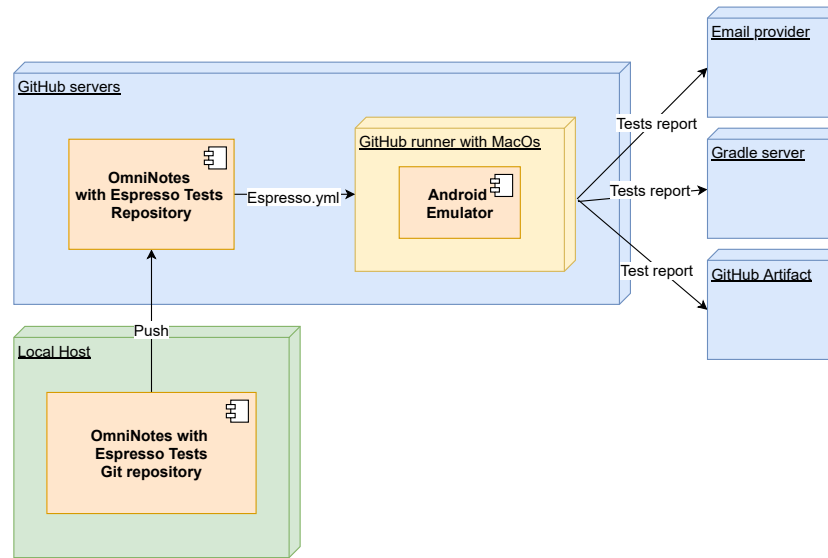


Figure 7.1: UML deployment diagram of Espresso CI/CD pipeline.

## 7.3 CI/CD pipeline using Appium

All the code has been split into two repositories. One repository contains all the application source code and the other contains tests code. The overall system structure is represented in Figure 7.3.

### 7.3.1 OmniNotes Repository

The code is available at the GitHub Repository. The repository structure is illustrated below.

```
OmniNotesContinuousIntegration
├── omninotes
├── .github
│   └── workflows
│       └── main.yml
```

The `omninotes` directory contains the application source code and the `.github/workflow` contains the `main.yml` files, which defines a workflow shown in Figure 7.4.

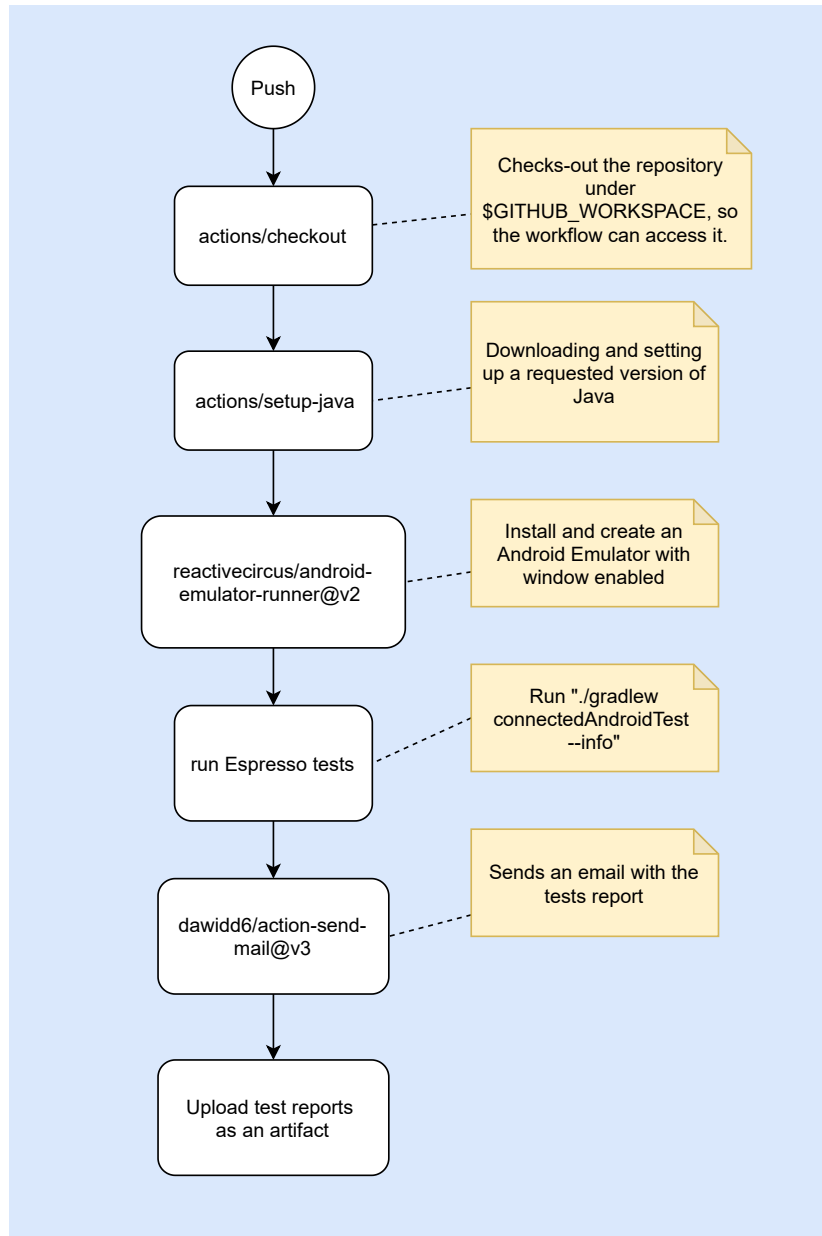


Figure 7.2: Workflow of the Espresso GitHub repository.

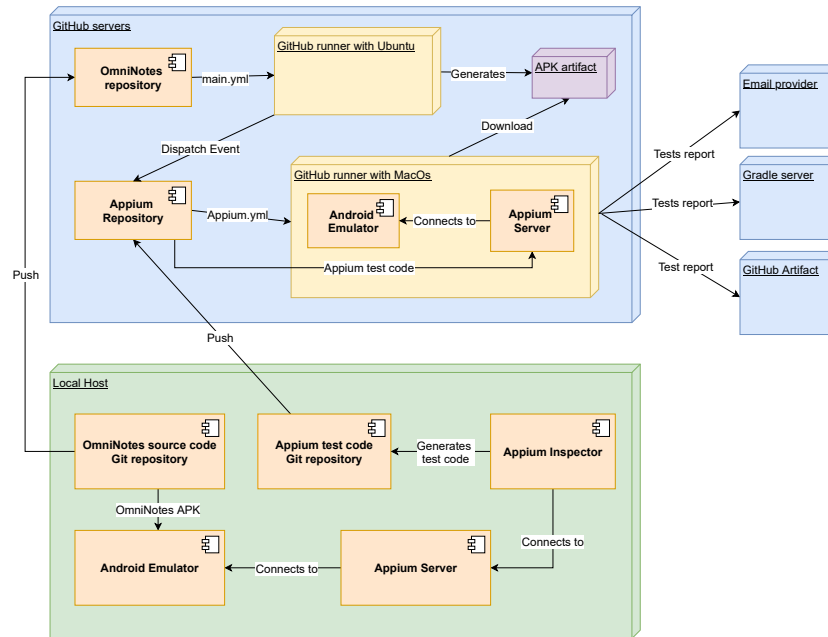


Figure 7.3: UML deployment diagram of Appium CI/CD pipeline.

## OmniNotes Repository workflow

The repository workflow is defined in the `main.yml` file. After each push event a job is started. The job is executed on a runner with the latest version of the Ubuntu operating system and it is made of 5 steps:

1. Checkout. Checks out the repository under the GitHub Workspace , so the workflow can access it and generate the APK from the source code.
2. Setup JDK. Downloads and sets up a requested version of Java.
3. Build APK. Runs the bash command `bash ./gradlew assembleDebug --stacktrace` which builds the APK.
4. Upload APK. Uploads the APK as an artifact from the workflow, allowing to share data between jobs.
5. Send event to another repository. Executes a Curl POST to AppiumContinuousIntegration and to

`SikuliContinuousIntegration`, sending a dispatch event. This requires GitHub's username and the access token of the recipient repository. Both values must be stored as secrets on this repository, respectively named `PAT_USERNAME` and `PAT_TOKEN`.

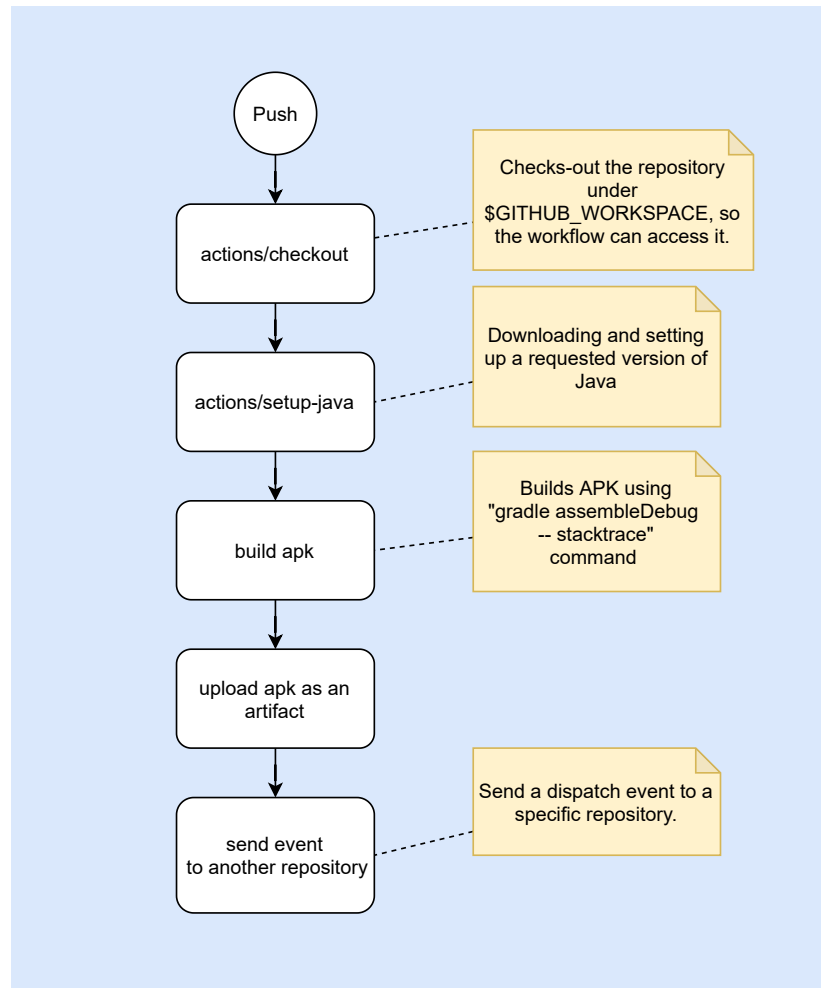
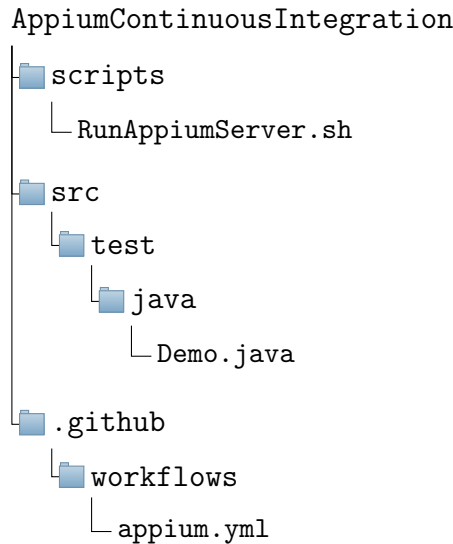


Figure 7.4: Workflow of the GitHub repository containing the application source code.

### 7.3.2 Appium Repository

The code is available at the [GitHub Repository](#). The repository structure is illustrated in the figure.



The `scripts` directory contains the `RunAppiumServer.sh`, a bash script to install Appium server on a GitHub runner, the Appium client test code in the `src` folder and the `.github/workflow` contains the `appium.yml` files, which defines a workflow shown in Figure 7.5.

#### Appium Repository workflow

The repository workflow is defined in the `appium.yml` file. After each push event or after a dispatch event, a job is started. It is possible to run tests in parallel with different emulated devices and versions using a matrix, that allows you to create multiple jobs by performing variable substitution in a single job definition. The job is executed on a runner with the latest version of the Mac OS operating system and it is made of 8 steps:

1. Checkout. Checks out the repository under the GitHub Workspace, so the workflow can access it.
2. Download APK artifact. Download an artifact from another repository. In order to do this, a Personal Access Token of the other repository is needed. It must have a "repo" scope and must be stored as a secret of



this repository, with the name of `ACCESS_TOKEN`. The artifact is stored under `./apps` directory.

3. Set Up JDK. Downloads and sets up a requested version of Java.
4. Install and Run Appium Server. Elevate privileges of `RunAppiumServer.sh` and execute the script. This installs Appium Server and runs it.
5. Give Gradlew privileges.
6. Run Appium Test. It's executed the `reactivecircus/android-emulator-runner` action with the script `./gradlew test --info`, which runs Appium tests.
7. Send Email. After all the tests are completed, a test report is generated. Using the `dawidd6/action-send-mail` it is possible to send the report via email.
8. Upload Test Report as a GitHub artifact, so the test results are directly accessible from the workflow results in the Action section on the GitHub repository.

## 7.4 CI/CD pipeline using Sikuli

All the code has been split into two repositories. One repository contains all the application source code and the other contains tests code. The overall system structure is represented in Figure 7.6.

### 7.4.1 OmniNotes Repository

It is the one described in the previous solution, at subsection 7.3.1 on page 59.

### 7.4.2 Sikuli Repository

The code is available at the `GitHub Repository`. The repository structure is illustrated below.

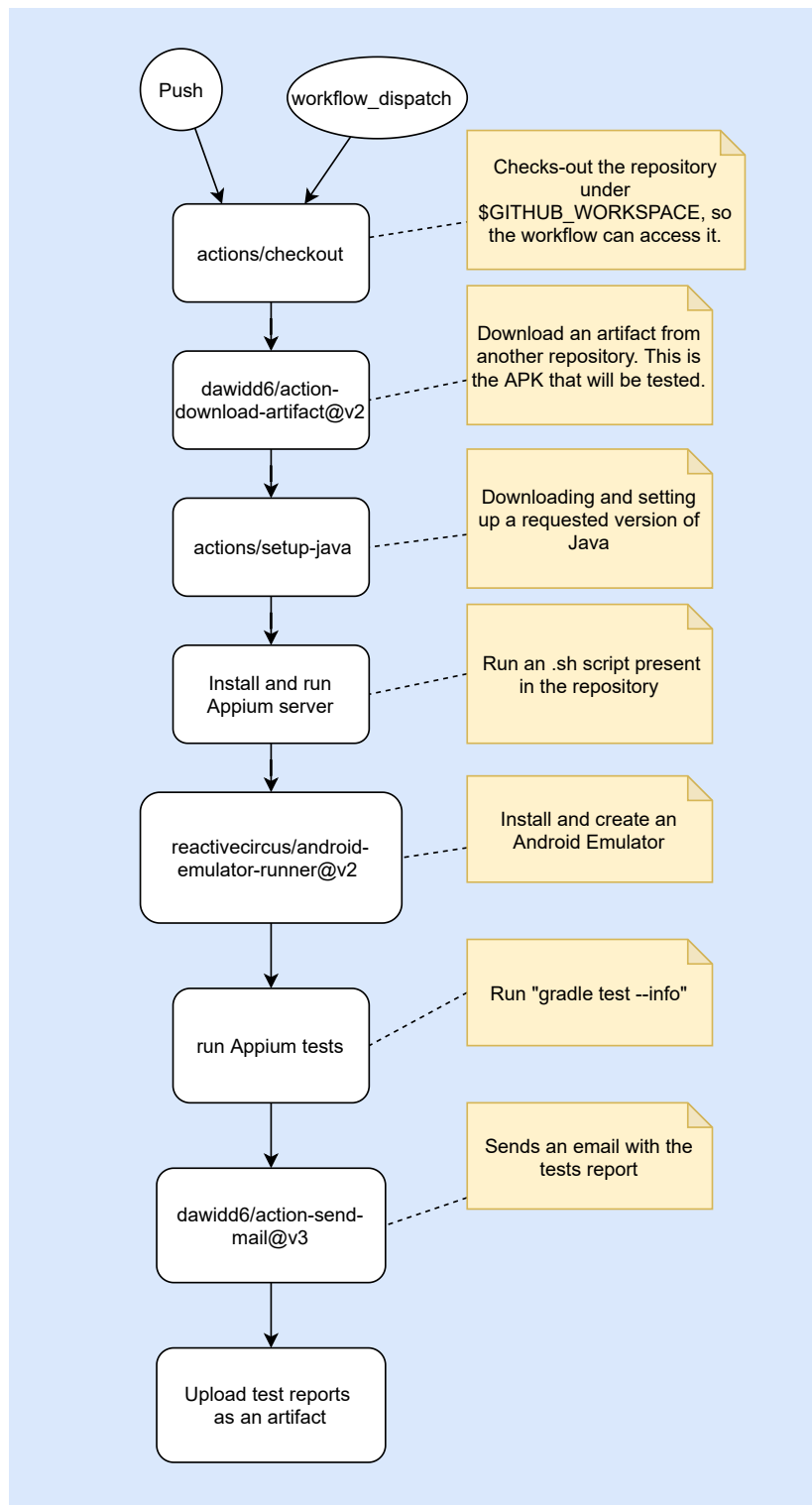


Figure 7.5: Workflow of the GitHub repository containing the Appium tests.

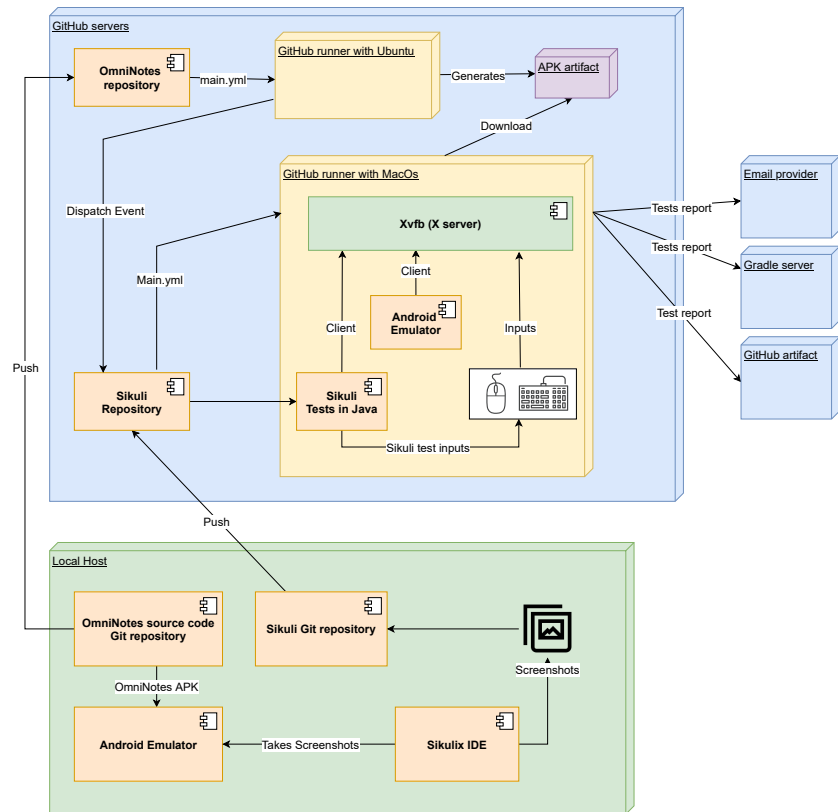


Figure 7.6: UML deployment diagram of Sikuli CI/CD pipeline.

```

SikuliContinuousIntegration
├── ArchiveNote.sikuli
├── DeleteNoteAndEmptyTrash.sikuli
├── GoToInfo.sikuli
├── InsertNewNote.sikuli
├── SearchNote.sikuli
├── LongTestRun.sikuli
├── InstallAPKandRunTests.sh
├── src
│   ├── test
│   │   └── java
│   │       └── Tests.java
├── .github
│   └── workflows
│       └── main.yml

```

All the `.sikuli` folders contain screenshots used for Sikuli tests. In the `src` folder is `Tests.java`, with java code to run Sikuli tests. The `.github` folder contains the `main.yml` that defines the workflow described in the Figure 7.7.

## Sikuli Repository Workflow

The repository workflow is defined in the `main.yml` file. It consists of a job with 9 steps executed in a runner with MacOS :

1. Checkout. Checks out the repository under the GitHub workspace, so the workflow can access it.
2. Download APK artifact. Download an artifact from another repository. In order to do this, a Personal Access Token of the other repository is needed. It must have a "repo" scope and must be stored as a secret of

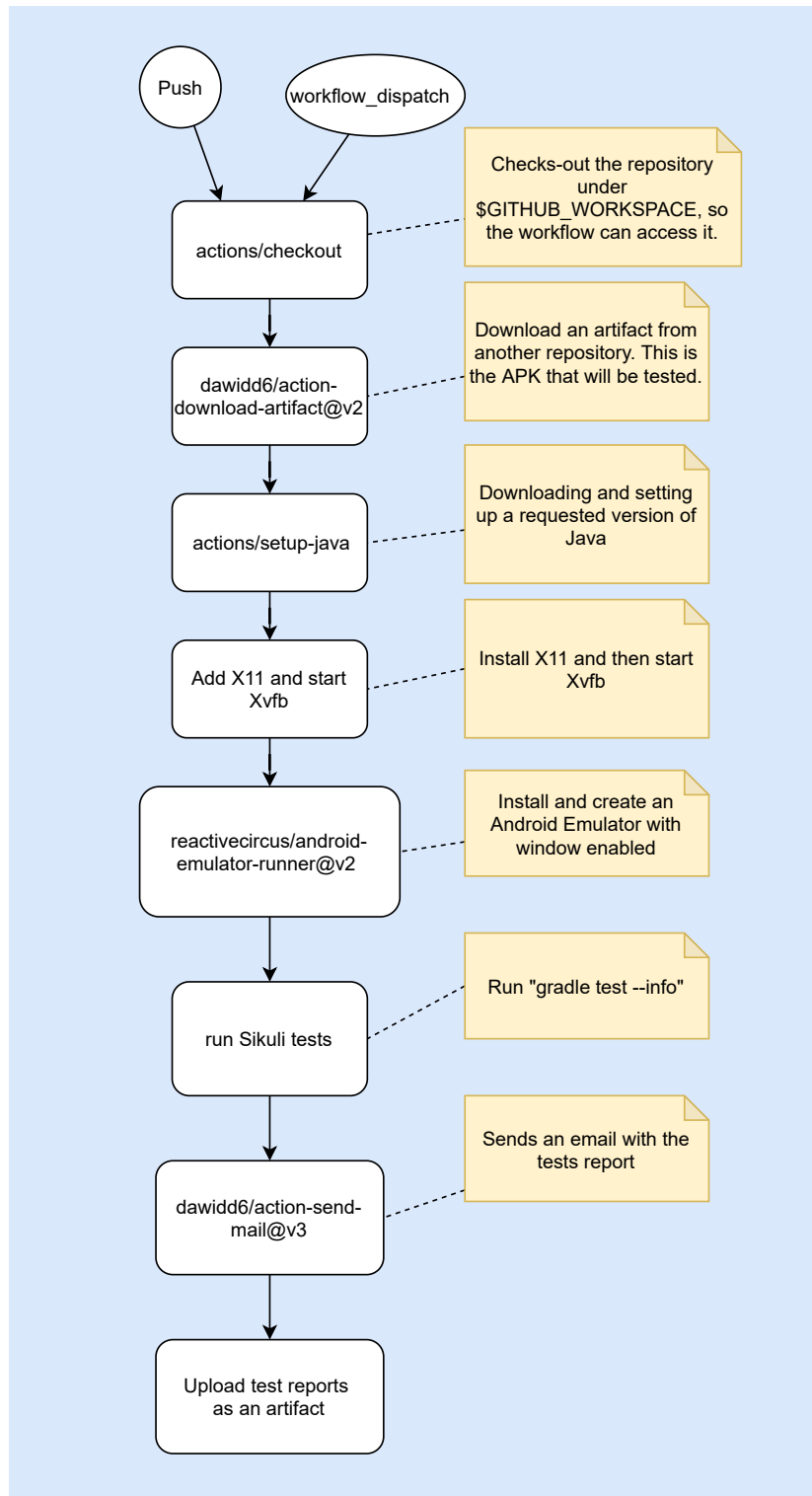


Figure 7.7: Workflow of the GitHub repository containing the Sikuli tests.

this repository, with the name of `ACCESS_TOKEN`. The artifact is stored under `./apps` directory.

3. Set Up JDK. Downloads and sets up a requested version of Java.
4. Give Gradlew privileges.
5. Add X11. Install the X Window System, also called X11. It provides a basic framework for a GUI environment.
6. Start Xvfb on DISPLAY number 1. Start the X virtual framebuffer, a display server implementing the X11 display server protocol.
7. Run Sikuli Tests. It's executed the `reactivecircus/android-emulator-runner` with `no-window` options disabled, which enables the display of the emulator, and with `enable-hw-keyboard`, which enables keyboard input to the emulator. This option modifies the file `config.ini` of the Android Virtual Device. In particular, the last one has been added after an issue opened on the GitHub Action's repository and without it enabled, Sikuli can't send key input to the emulator. After the AVD is created and the Android Emulator starts, it is executed the script `InstallAPKandRunTest.sh`. It connects the terminal to the DISPLAY number 1. Then using ADB it installs the apk on the emulator and launches the application. When the application starts it executes `./gradlew test --info` that launches the Sikuli tests written in Java.
8. Send Email. After all the tests are completed, a test report is generated. Using the `dawidd6/action-send-mail` it is possible to send the report via email.
9. Upload Test Report as a GitHub artifact, so the test results are directly accessible from the workflow results in the Action section on the GitHub repository.

### 7.4.3 Screenshots Recording

The ideal solution represented in Figure 7.6 assumes that it is possible to replicate exactly the environment present on the GitHub runner, allowing to record the tests locally. In reality, this is not possible with the runner hosted directly by GitHub and setting up a self-hosted runner requires a real

dedicated server. To obviate this, Appium's solution has been used to take screenshots directly on the GitHub runner instance.

## 7.5 The X virtual framebuffer

The X virtual framebuffer (Xvfb) is a display server implementing the X11 display server protocol. The X11 protocol, also called X Window System, is a network-transparent windowing system for bitmap displays. It uses a client-server model: X server program runs on a computer with a graphical display and communicates with various client programs. Any application that requires GUI and interacts with the X server is called X client. X server requires a graphical display and without it, it will not start and all the X clients would fail. In the GitHub runner scenario there isn't any display, thus a standard X server can't be executed and the Android Emulator, which aims to output the emulated device screen, and Sikuli, that needs a display to recognize display areas similar to its test screenshots, can't be executed. So a virtual X server has been used, which instead of outputting signals to screen, outputs signals to memory. An implementation of a virtual X server is Xvfb. It does not require any kind of graphics adapter, screen or input device. A framebuffer is a memory buffer of a GPU, where are memorized all the data required to represent a frame on the screen. In Xvfb the framebuffer instead of being on the GPU memory, is saved on the RAM. A representation of Xvfb is on Figure 7.8. So, as shown in Figure 7.6, on the GitHub runner it's installed and executed an X virtual framebuffer, with Android Emulator and Sikuli being both X clients. The first outputs the emulated screen and the second search on the virtual display for screenshots match and then sends virtual inputs.[15]

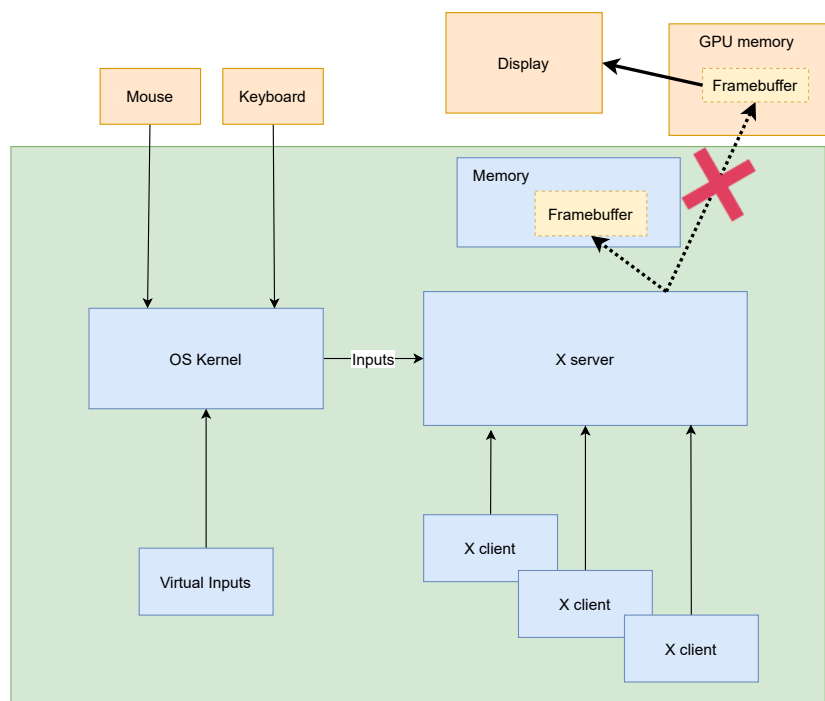


Figure 7.8: Xvfb structure.



# Chapter 8

## CI/CD Pipelines Evaluation

The proposed solutions in Chapter 7 have been analyzed in this Chapter, testing their performances in different environments and situations.

### 8.1 Procedure

#### 8.1.1 Increased Test Set

In addition to the test cases introduced in 4.1 have been added another five:

1. Insert a new checklist.
2. Insert a new note with reminder.
3. Sort notes.
4. Add a new category.
5. Remove an existing category.

#### 8.1.2 Pipelines and Tools Modifications

The pipelines introduced in Chapter 7 have been modified to execute multiple tests in the same workflow. In addition to that, a build matrix has been used to allow the workflows to run tests across multiple Android devices. The build matrix is created using the `strategy` keyword on the `.yaml` file.

## Espresso

The Android Test Orchestrator has been used in order to achieve tests isolation on the same workflow. This required only some changes to the `build.gradle` file.

## Appium

An implicit wait of five seconds has been added to the "AndroidDriver" to improve test's stability. When searching for a single element, the driver should poll the page until an element is found or the timeout expires, whichever occurs first. When searching for multiple elements, the driver should poll the page until at least one element is found or the timeout expires, at which point it should return an empty list.[29]

## Sikuli

To achieve tests isolation, before each test, it is called a function named `cleanAppState` which sequentially executes three ADB commands:

1. `adb shell am force-stop`
2. `adb shell pm clear`
3. `adb shell am start`

This solution is a stretch and can cause test flakiness: for example, the `adb shell am start` command restarts the application, but the application's startup time depends on the execution environment and can vary a lot.

### 8.1.3 Analyzing Fragmentation

To analyze the impact of the device's fragmentation problem, for each pipeline the 10 test cases have been executed on 10 different devices. The devices chosen are shown in Table 8.1.

The choice was dictated by the already available devices offered by the `reactivecircus/android-emulator-runner` GitHub action. Nevertheless, it's possible to notice that the screen resolution and aspect-ratio vary a lot, for example from Pixel 3a to a Pixel or a Pixel c (as shown in Table 8.1), offering a good benchmark, especially for the Sikuli solution. For this one, as

	screen resolution	screen aspect ratio
Nexus 6	2560x1440 px	16:9
Nexus 6P	2560x1440 px	16:9
Nexus 9	2048x1536 px	4:3
Nexus S	800x480 px	16:9
Nexus 5X	1920x1080 px	16:9
Pixel 2	1920x1080 px	16:9
Pixel c	2560x1800 px	1: $\sqrt{2}$
Pixel xl	2560x1440 px	16:9
Pixel 3a	2220x1080 px	18.5:9
Pixel 4 xl	3040x1440 px	19:9

Table 8.1: Devices utilized for tests with their screen resolution and aspect ratio.

already said in 7.4.3, the screenshots have been taken from Appium’s solution on Nexus 6.

#### 8.1.4 Analyzing Flakiness

To analyze test flakiness, the 10 test cases have been executed 10 times on the Nexus 6. Each workflow execution has the same environment but, given the fact that GitHub hosts runners as a free service, the performances vary a lot depending on the availability of resources. This increases test flakiness due to unexpected lags of the Android emulator.

## 8.2 Results

To evaluate the solutions, only the success or the failure of each test has been taken into account. This is because as already said in 8.1.4 the service in use is free and the performances vary depending on the availability of the resources, so values like the execution time can not be taken into account. The results of the tests are reported in Figure 4.6, Figure 4.7 and Figure 4.8.

As shown in Table 8.2, Espresso has the highest success rates, followed by Appium and then Sikuli.

	fragmentation	flakiness
Espresso	89%	97%
Appium	83%	88%
Sikuli	17%	34%

Table 8.2: Success rate of tests of the different solutions.

<b>Espresso Fragmentation</b>	Nexus 6	Nexus 6P	Nexus 9	Nexus S	Nexus 5X	pixel_2	pixel_c	pixel_xl	pixel_3a	pixel_4_xl
insertNoteWithReminder	ok	ok	failed	ok	ok	ok	failed	ok	ok	ok
deleteCategory	ok	ok	failed	ok	ok	ok	failed	ok	ok	failed
searchNote	ok	ok	ok	ok	ok	ok	failed	failed	ok	ok
insertNewCategory	ok	ok	ok	ok	ok	ok	failed	failed	ok	ok
insertNewCheckList	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok
deleteNoteAndEmptyTrash	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok
sortNotes	ok	ok	ok	failed	ok	ok	failed	ok	ok	ok
infoMenu	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok
archiveNote	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok
insertNewNote	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok
<b>Espresso Flakiness (Nexus 6 x 10)</b>	1	2	3	4	5	6	7	8	9	10
insertNoteWithReminder	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok
deleteCategory	ok	failed	ok	ok	ok	ok	ok	ok	ok	ok
searchNote	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok
insertNewCategory	ok	ok	ok	ok	ok	failed	ok	failed	ok	ok
insertNewCheckList	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok
deleteNoteAndEmptyTrash	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok
sortNotes	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok
infoMenu	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok
archiveNote	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok
insertNewNote	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok

Figure 8.1: Results of Espresso's solution.

<b>Appium Fragmentation</b>	Nexus 6	Nexus 6P	Nexus 9	Nexus S	Nexus 5X	pixel_2	pixel_c	pixel_xl	pixel_3a	pixel_4_xl
insertNoteWithReminder	ok	ok	failed	ok	ok	ok	failed	ok	ok	ok
deleteCategory	ok	ok	ok	ok	ok	ok	failed	ok	ok	ok
searchNote	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok
insertNewCategory	ok	ok	ok	ok	ok	ok	failed	ok	ok	ok
insertNewCheckList	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok
deleteNoteAndEmptyTrash	ok	ok	ok	failed	ok	ok	failed	failed	ok	ok
sortNotes	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok
infoMenu	failed	failed	failed	failed	failed	failed	failed	failed	failed	failed
archiveNote	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok
insertNewNote	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok
<b>Appium Flakiness (Nexus 6 x 10)</b>	1	2	3	4	5	6	7	8	9	10
insertNoteWithReminder	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok
deleteCategory	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok
searchNote	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok
insertNewCategory	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok
insertNewCheckList	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok
deleteNoteAndEmptyTrash	failed	ok	ok	ok	ok	ok	ok	ok	ok	failed
sortNotes	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok
infoMenu	failed	failed	failed	failed	failed	failed	failed	failed	failed	failed
archiveNote	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok
insertNewNote	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok

Figure 8.2: Results of Appium's solution.

<b>Sikuli Fragmentation</b>	Nexus 6	Nexus 6P	Nexus 9	Nexus S	Nexus 5X	pixel_2	pixel_c	pixel_xl	pixel_3a	pixel_4_xl
insertNoteWithReminder	ok	ok	failed	failed	ok	ok	failed	ok	failed	failed
deleteCategory	failed	failed	failed	failed	failed	failed	failed	failed	failed	failed
searchNote	ok	ok	failed	failed	ok	ok	failed	ok	failed	failed
insertNewCategory	failed	failed	failed	failed	failed	failed	failed	failed	failed	failed
insertNewCheckList	failed	failed	failed	failed	failed	failed	failed	failed	failed	failed
deleteNoteAndEmptyTrash	failed	ok	failed	failed	ok	failed	failed	failed	failed	failed
sortNotes	failed	failed	failed	failed	failed	failed	failed	failed	failed	failed
infoMenu	undefined	undefined	undefined	undefined	undefined	undefined	undefined	undefined	undefined	undefined
archiveNote	failed	failed	failed	failed	failed	failed	failed	failed	failed	failed
insertNewNote	ok	ok	failed	failed	ok	ok	failed	ok	failed	failed
<b>Sikuli Flakiness (Nexus 6 x 10)</b>	1	2	3	4	5	6	7	8	9	10
insertNoteWithReminder	ok	ok	failed	ok	ok	ok	ok	ok	ok	ok
deleteCategory	failed	failed	failed	failed	failed	failed	failed	failed	failed	failed
searchNote	ok	ok	failed	ok	ok	ok	ok	ok	ok	ok
insertNewCategory	failed	failed	failed	failed	failed	failed	failed	failed	failed	failed
insertNewCheckList	failed	ok	failed	ok	ok	failed	failed	failed	ok	ok
deleteNoteAndEmptyTrash	ok	failed	failed	failed	failed	ok	failed	failed	failed	failed
sortNotes	failed	failed	failed	failed	failed	failed	failed	failed	failed	failed
infoMenu	undefined	undefined	undefined	undefined	undefined	undefined	undefined	undefined	undefined	undefined
archiveNote	failed	failed	failed	failed	failed	failed	failed	failed	failed	failed
insertNewNote	ok	ok	failed	ok	ok	ok	ok	ok	ok	ok

Figure 8.3: Results of Sikuli’s solution.

## 8.2.1 Considerations

### Espresso

Regarding fragmentation (Figure 8.1), Espresso’s solution handles well nearly every device except for the Pixel c, which is a tablet with an unusual aspect ratio of  $1 : \sqrt{2}$ , which can cause some unexpected GUI behaviours. Ignoring that, the success rate rises up to 93% which is a good result. Espresso handles well also the flakiness (Figure 8.1), with a success rate of 97%: the tests that involve the deletion and insertion of a category are the only ones with some failures. This can be attributed to the fact that they require handling some complex dialog. Ignoring that the success rate raises to 100%.

### Appium

When the test of navigation through info section is performed remotely on a GitHub runner the ”TouchDown action” fails every time, giving the test’s success rate of 0% for both the fragmentation and the flakiness. If the ”infoMenu” test is excluded the success rate in flakiness reaches 97% which is comparable to the Espresso one. Regarding fragmentation, if also the Pixel c’s results are ignored for the reasons explained in 8.2.1, the success rate raises up to 93%, the same of Espresso’s solution. (Figure 8.2)

## Sikuli

Looking at Figure 8.3 seems that Sikuli's success rates are near 0%, but further considerations are needed to better understand the phenomena.

1. The "infoMenu" test, which expects to navigate through the app to the info section, can not be executed as explained in Figure 4.6.
2. The "archiveNote" test fails because the swipe left action (Figure 4.3) is executed with Sikuli `dragDrop` function, which is sensible to the position of the Android emulator on the host screen.
3. The "deleteCategory" and "insertNewCategory" tests fail because the dialog for the addition of a new category (Figure 8.4) has the category's colour, which can be selected with a click on the dot in the center, that is picked randomly each time. This makes impossible for Sikuli to recognize that GUI element.

So considering only the tests "insertNoteWithReminder", "searchNote", "insertNewNote" and ignoring Pixel c results for the reasons explained in 8.2.1, the success rate in fragmentation is near 52% and in flakiness is 90%.

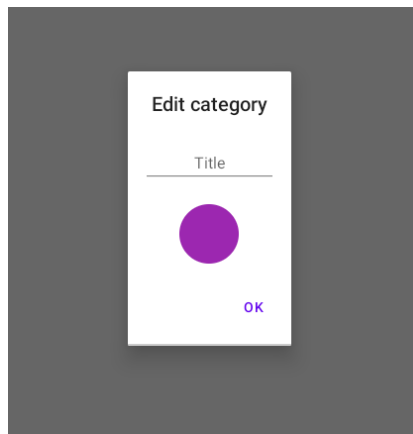


Figure 8.4: Dialog for the addition of a new category in Omninotes application.

# **Part IV**

## **Conclusions**

# Chapter 9

## Achievements

### 9.1 Espresso pipeline

The solution that utilizes Android Espresso necessitates nearly zero setups: it is required only to have a GitHub repository and to define the workflow of the pipeline in a `.yaml` file. Although it is limited to the Android ecosystem, this solution is the most reliable of the three, but it requires more time to write tests. In conclusion, it is the most suitable solution for developers that want to write white box, end-to-end, GUI levels tests in an Android environment and are inclined to spend more time writing tests in favor of reliability and robustness to flakiness and fragmentation (8.2).

### 9.2 Appium pipeline

The solution based on Appium has similar performances to the Espresso one. To take advantage of fast test recording, it necessitates also a local setup that includes Appium Server, Appium Inspector and an Android emulator, requiring more time to set up than the Espresso solution. The application and the tests code are split into two GitHub repositories: this can be both a benefit, which offers a separation between the development and testing phase (Figure 1.1), and a drawback, because of increased maintenance costs. In conclusion, this solution is more suitable for developers that work with both Android and iOS, that want to have the ability to record tests in a fast way and write them in the language they prefer, sacrificing some reliability and robustness that the Espresso solution offers.



## 9.3 Sikuli pipeline

The proposed solution records the tests by taking screenshots of the Android emulator on the GitHub runner using the Appium pipeline. This removes all the benefits that Sikuli offers, which are fast test recordings with nearly zero setups. In a scenario where it is possible to have a self-hosted GitHub runner with a reproducible environment in a local host where screenshots for tests can be taken, developers may benefit from this solution to make simple black-box, end-to-end test in an extremely fast way. For example, it can be very useful for making simple end-to-end tests of a legacy application, like compiling some forms. As can be seen in 8.2, in simple test cases on a single device, this solution has good reliability, especially in relationship with the velocity and simplicity of test recording.

# Chapter 10

## Future developments

In this chapter will be suggested some of the improvements to the proposed solutions, that could be done in the future.

### 10.1 Continuous Integration Environment

The following improvements can be brought to all the pipelines, independently of the testing framework used.

- Using **self-hosted GitHub runners or Jenkins** in order to have more control of the testing environment and make it more stable. This can significantly reduce test flakiness, as an unstable testing environment is one of the major causes (1.2.1). The controlled environment will also allow to test performances others than the simple success rates.
- Generating **better and more detailed test reports** or aggregating the reports of more pipelines in a single solution.
- Using a **custom action for the Android emulator** instead of `ReactiveCircus/android-emulator-runner`.

### 10.2 Testing Framework Implementation

The implementation of the testing framework in both the local and the pipeline environments plays a primary role not only in the performances but also in the usability and the speed with which developers record tests.

### **10.2.1 Appium**

Appium can be improved in the local environment with an update of the Appium Inspector, so the tests can be directly recorded with the necessity to add only a little code to complete test scripts. This will drastically speed up test recording.

### **10.2.2 Sikuli**

Having the same environment both local and on the pipeline could improve a lot the solution based on Sikuli, allowing to easily take screenshots locally instead of taking them with the tricky solution adopted with the use of Appium (7.4.3). This would also reduce test flakiness and improve the test's success rates. The mechanism adopted to isolate tests necessitates being improved with the ability to reset the application state consistently. Finally, the screenshots taken for tests must be handled in a better way to support the scalability of the solution.

# Chapter 11

## Acknowledgements

I thank Prof. Luca Ardito, Prof. Morisio Maurizio, Prof. Torchiano Marco and Riccardo Coppola to gave me the possibility to make this thesis. I thank my loved ones, that have always supported me throughout my studies.

# Bibliography

- [1] *android-emulator-runner GitHub Repository*. URL: <https://github.com/ReactiveCircus/android-emulator-runner>.
- [2] *Appium Official Website*. URL: <https://appium.io/docs/en/about-appium/intro/?lang=en>.
- [3] *AWS Device Farm website*. URL: <https://aws.amazon.com/it/device-farm/>.
- [4] *Barista GitHub repository*. URL: <https://github.com/AdevintaSpain/Barista>.
- [5] *Create UI tests with Espresso Test Recorder*. URL: <https://developer.android.com/studio/test/other-testing-tools/espresso-test-recorder>.
- [6] *Espresso Testing Framework - Architecture*. URL: [https://www.tutorialspoint.com/espresso\\_testing/espresso\\_testing\\_architecture.htm](https://www.tutorialspoint.com/espresso_testing/espresso_testing_architecture.htm).
- [7] Continuous Testing Expert. *A Comparison Report on the Top 4 iOS Testing Tools*. URL: <https://digital.ai/catalyst-blog/a-comparison-report-on-the-top-4-ios-testing-tools>.
- [8] Continuous Testing Expert. *Comparing the Top 4 Android Testing Tools*. URL: <https://digital.ai/catalyst-blog/comparing-the-top-4-android-testing-tools>.
- [9] *Firebase Test Lab website*. URL: <https://firebase.google.com/docs/test-lab>.
- [10] Martin Fowler. *TestPyramid*. URL: <https://martinfowler.com/bliki/TestPyramid.html>.
- [11] *Get Started Page*. URL: <http://roboelectric.org/>.
- [12] *Jenkins's official website*. URL: <https://www.jenkins.io/doc/>.

- [13] Tomer Kalmovich. *Fastest Mobile Automation Testing Tool – Appium vs. XCUITest vs. TestProject vs. UiAutomator vs. Espresso*. URL: <https://blog.testproject.io/2020/04/08/fastest-mobile-automation-testing-tool-appium-vs-xcuitest-vs-testproject-vs-uiautomator-vs-espresso/>.
- [14] Federica Laricchia. *iOS version share of Apple devices worldwide 2016-2022*. URL: <https://www.statista.com/statistics/565270/apple-devices-ios-version-share-worldwide/>.
- [15] Lei Mao. *Running X Client Using Virtual X Server Xvfb*. URL: <https://leimao.github.io/blog/Running-X-Client-Using-Virtual-X-Server-Xvfb/>.
- [16] Kevin Moran Mario Linares-Vásquez and Denys Poshyvanyk. *Continuous, Evolutionary and Large-Scale: A New Perspective for Automated Mobile App Testing*. URL: <https://arxiv.org/pdf/1801.06267.pdf>.
- [17] Leo Mirani. *There are now more than 24,000 different Android devices*. URL: <https://qz.com/472767/there-are-now-more-than-24000-different-android-devices/>.
- [18] *Mosaic’s GitHub Repository*. URL: <https://github.com/Matthalp/mosaic>.
- [19] Jason Palmer. *Test Flakiness – Methods for identifying and dealing with flaky tests*. URL: <https://firebase.google.com/docs/test-lab>.
- [20] Yash Patel. *Which Are The Top And Trending Mobile Testing Frameworks?* URL: <https://www.thesunflowerlab.com/blog/which-are-the-top-and-trending-mobile-testing-frameworks/>.
- [21] *Perfecto Cloud Lab website*. URL: <https://www.perfecto.io/supported-devices/monthly-subscription-plans>.
- [22] George Pirocanac. *Test Flakiness - One of the main challenges of automated testing*. URL: <https://testing.googleblog.com/2020/12/test-flakiness-one-of-main-challenges.html>.
- [23] *QMetry Automation Framework*. URL: <https://qmetry.github.io/qaf/latest/docs.html>.
- [24] *Quantum Perfecto Integration*. URL: <https://www.perfecto.io/integrations/quantum>.
- [25] *Ranorex home page*. URL: <https://www.ranorex.com/>.

- [26] *RERAN - Record and Replay for Android*. URL: <https://www.androidreran.com/>.
- [27] *Robotium Recorder*. URL: <https://plugins.jetbrains.com/plugin/7513-robotium-recorder>.
- [28] S.D. *Stack Overflow answer*. URL: <https://stackoverflow.com/questions/18271474/roboelectric-vs-android-test-framework>.
- [29] *Set Implicit Wait Timeout*. URL: <https://appium.io/docs/en/commands/session/timeouts/implicit-wait/>.
- [30] *Sikulix official website*. URL: <http://sikulix.com/>.
- [31] *Understanding GitHub Actions*. URL: <https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions>.
- [32] Iulian Neamtiu Yongjian Hu. *VALERA: An Effective and Efficient Record-and-Replay Tool for Android*. URL: <https://ieeexplore.ieee.org/document/7833000>.