Foundation Level Tester for Appium Syllabus

Release – Version 1.0

Copyright Notice

Revision History

Version	Date	Remark
0.1	13.11.2020	Initial Writing
A0.1 – A0.10	13.11.2021	Rework after review with external customers, Appium® 2.0
A.011RC	20.02.2022	Review RC
1.0A	18.03.2022	Release 1.0A
1.0	28.07.2022	Changed the exam duration Added the student's prerequisites

Table of Contents

Table of Figures

This document was formally released by A4Q on 28th July 2022.

0 Introduction

0.1 Purpose of this Syllabus

This syllabus presents the business outcomes, learning objectives, and concepts underlying the A4Q Foundation Level Tester for Appium certification and training.

This syllabus is aimed at developers and testers or anyone who want to learn mobile automation techniques using Appium®.

0.2 Examinable Learning Objectives and Cognitive Levels of knowledge

Learning objectives support the business outcomes and are used to create the A4Q Foundation Level Tester for Appium exams.

In general, all contents of this syllabus are examinable at K1, K2, K3, K4 levels. That is, the candidate may be asked to recognize, remember, or recall a keyword or concept mentioned in any of the next chapters. The knowledge levels of the specific learning objectives are shown at the beginning of each chapter, and classified as follows:

- K1: remember
- K2: understand
- K3: apply
- K4: analyze

Please check Appendix A – Learning Objectives/Cognitive Level of Knowledge for detailed information.

0.3 The A4Q Foundation Level Tester for Appium Exam

The A4Q Foundation Level Tester for Appium exam will be based on this syllabus. Answers to the exam questions may require the use of material based on more than one section of this syllabus. All sections of the syllabus are examinable, except for the Introduction and Appendices.

Standards, books, and ISTQB® syllabi may be included as references in the A4Q Foundation Level Tester for Appium syllabus, but their content is not examinable, beyond what is summarized in this syllabus itself from such standards, books, and ISTQB® syllabi.

The exam is comprised of 40 multiple-choice questions. Each correct answer has a value of one point in accordance with the “A4Q Foundation Level Tester for Appium Exam Structure and Rules“ document.

A score of at least 65% (26 points) is required to pass the exam and be certified.

The time allowed to take the exam is 60 minutes. If the candidate’s native language is not the same as the examination language, the candidate may be allowed an extra 25% time.

Participation in an A4Q Foundation Level Tester for Appium training course with an accredited training provider is highly recommended but not compulsory. An accredited training provider will add additional information from the official accredited course materials, including in-class exercises, with, among other things, demonstrations of possible errors and solutions. Accredited training providers will also provide the “Student Exercise” document which will support student learning.

The examination can also be taken without having participated in the A4Q Foundation Level Tester for Appium training course where candidates choose to self-study.

0.4 Accreditation

Contact A4Q or A4Q authorized exam providers.

0.5 Level of Detail

The level of detail in this syllabus allows internationally consistent exams. To achieve this goal, the syllabus consists of:

- General instructional objectives describing the intention of the Foundation Level
- A list of terms that students must be able to recall
- Learning objectives for each knowledge area, describing the cognitive learning outcome to be achieved
- A description of the key concepts, including references to sources such as accepted literature or standards

The syllabus content is not a description of the entire knowledge area of Appium® automated testing.

It focuses on test concepts and techniques that can apply to all software projects, including agile projects.

This syllabus does not contain any specific learning objectives related to any software development lifecycle or method, but it does discuss how these concepts may apply in various software development lifecycles.

0.6 How this Syllabus is Organized

There are 5 chapters with examinable content.

The top-level heading for each chapter specifies the time for the chapter; timing is not provided below chapter level. For the A4Q Foundation Level Tester for Appium training course, the syllabus requires a minimum of 12 hours of instruction, distributed across the chapters as follows:

- Chapter 0: Introduction (Information about this Syllabus)
- Chapter 1: Introduction to Mobile Application Testing 220 minutes
- Chapter 2: APPIUM® Introduction 260 minutes
- Chapter 3: APPIUM® - Installation and Setup 110 minutes
- Chapter 4: Using Appium® 335 minutes
- Chapter 5: List of system Capabilities 85 minutes
- Chapter 6: Appium® 2.0 75 minutes
- Chapter 7: Bibliography
- Appendix A – Learning Objectives/Cognitive Level of Knowledge
- Appendix B – Glossary of Appium® Terms
- Appendix C – Sources
- Appendix D – Source Example Editor-APK

given the following average time duration to cover each learning objective (LO):

- K1: 10 minutes
- K2: 25 minutes
- K3: 60 minutes
- K4: 75 minutes

Together with theory explanation of this syllabus, several practical tests or exercises will be shown and asked to the students to actively participate. It will bring the total training time duration to approx. 24-27 hours.

The given duration for each chapter is based on Kn (see Appendix A – Learning Objectives/Cognitive Level of Knowledge for detailed information), however some activities can take longer or can be completed faster.

0.7 Business Context

Organizations are striving to improve their business agility to provide valuable products and services in a world that is changing faster and faster due to digitalization. One important mechanism is to change the culture and mindset of the organization using a variety of principles, frameworks, disciplines and methodologies, such as Agile, Lean, and DevOps, which we have chosen to cover with the term Business Agility.

With Business Agility there is an even greater need for quality and testing as it is a prerequisite to reach and sustain a high level of speed and agility. However, at the same time roles such as test manager, test coordinator, QA Engineer and Tester are changing as organizations focus on having the necessary skills and competencies throughout the organization rather than having dedicated functional roles.

At the same time there is a change away from management roles towards self-empowered teams and supporting leadership. Therefore, classic roles like project managers and test managers struggle to find their place in organization moving towards Business Agility.

However, the need for quality management remains even if companies organize themselves differently, use different methods and increasingly automate processes. Testing professionals in Agile organizations need to understand and support agile principles, methodologies, and processes.

Testers need to learn to use technical aspects such as model-based testing, test automation and test techniques as well as organizational aspects such as value streams, quality management systems, and scaling with multiple agile teams. They also need to gain more soft skills such as training, facilitation, and quality coaching.

0.8 Business Outcomes (BO)

The business outcome describes the benefit that a certified person should be able to deliver using the knowledge and competencies covered in the syllabus. The knowledge and competencies are described in the learning objectives (LO) for each chapter.

- ATF-BO-1 Understand the importance of test automation in Mobile devices
- ATF-BO-2 Correctly apply test automation principles to build maintainable test automation solution
- ATF-BO-3 Be able to implement Appium® scripts that execute functional mobile application tests
- ATF-BO-4 Be able to implement maintainable scripts

Every chapter contains LOs (Learning objectives) e.g., ATF-1.1 and references to the student exercise document, e.g., ATF-H-01 used in accredited training. The references to the training exercise document are not given on the same level of LOs but indicated at the end of the list.

0.9 Prerequisite Knowledge

Candidates for the A4Q Foundation Level Tester for Appium certification should have at least:

- Testing approach knowledge (especially test automation)
- Software programming knowledge. This syllabus uses Java for the examples and therefore Java knowledge is recommended
- Admin knowledge for either Windows or iOS based computers, necessary to install tools and possibly trouble shooting

0.10 System Prerequisite

During the training and specifically exercises, some time will be dedicated to e.g., learn how to install the complete environment and some indications about possible failures and how to solve them. However, to save time it is suggested to the student that they install the environment on his machine before the training session will start. This is not a must, but it will save some time to be invested more in the training and exercises.

0.11 Acronyms

Acronym	Meaning
Adb	Android Debug Bridge
AKA	Also Known As
API	Application Programming Interface
AVD	Android Virtual Device
Bug	"defect" : an Imperfection or deficiency in a work product where it does not meet the requirements or specification (https://Glossary.istqb.org)
cfr	Check For Reference
CI	Continuous Integration
Cmd	Command line (e.g., DOS window)
CSS	Cascading Style Sheets
DDT	Data Driven Testing
DOM	Document Object Model
DRY	Don't Repeat Yourself
FW	Framework
GUI	Graphical User Interface (same as UI)
HTTP	Hyper Text Transfer Protocol
ISTQB®	International Software Testing Qualifications Board
KDT	Keyword Driven Testing
LO	Learning Objective
OSC	Operating Sequence Controller
PD	Person's day / Persons' days
POP	Page Object Pattern
REST	Representational State Transfer
ROI	Return on Investment (win/win situation)
SDLC	Software Development Life Cycle
SOAP	Simple Object Access Protocol
SUT	System Under Test
SRP	Single Responsibility Principle
Session-ID	Channel to exchange data between client and server
SiT	System Integration Test
ST	System Test
TAA	Test Automation Architecture
TCP	Transmission Control Protocol
UAT	User Acceptance Test
UT	Unit Test
UI	User Interface (same as GUI)
W3C	World Wide Web consortium

0.12 Definitions

Definition	Explanation (from ISTQB® glossary)
Unit / Component Testing	A test level that focuses on individual hardware or software components
White box testing	Testing based on an analysis of the internal structure of the component or system
Black box testing	A test technique based on an analysis of the specification of a component or system
System Integration Testing	A test level that focuses on interactions between systems
System Testing	A test level that focuses on verifying that a system as a whole meets specified requirements.
GUI testing	Testing performed by interacting with the software under test via the graphical user interface.
Test automation	The use of software to perform or support test activities.
Test automation solution / plan	A realization/implementation of a test automation architecture, i.e., a combination of components implementing a specific test automation assignment. The components may include commercial off-the-shelf test tools, test automation frameworks, as well as test hardware
Test manual	Testing performed by human (tester) without any support from software or special tool to perform (repetitive) actions fast and without intervention.
Defect	an Imperfection or deficiency in a work product where it does not meet the requirements or specification (https://Glossary.istqb.org)

1 Introduction to Mobile Application Testing

Timing	8*K2, 2*K1 = 220 minutes
Terms	Testing, test automation, manual testing, mobile test automation, roi

Topics & Learning Objectives for this Chapter:

Learning Objectives

ATF-1.1.1	(K1)	Why testing is necessary
ATF-1.1.2	(K1)	Manual Testing
ATF-1.1.3	(K2)	Automated testing
ATF-1.2.1	(K2)	Advantages of Mobile Testing Automation
ATF-1.2.2	(K2)	Some ideas to create good, automated test cases
ATF-1.2.3	(K2)	Differences between Manual and Automated Test
ATF-1.3	(K2)	Manual vs. Automated Tests
ATF-1.4	(K2)	Success Factors of Automated test cases
ATF-1.5	(K2)	Types of Automated test cases (some ideas)
ATF-1.6	(K2)	Test reports by Test automation

1.1 Test Automation Overview

Since the first piece of software was created, some “control” to confirm “it does what it is foreseen to do” has been performed, by the developer themselves and later by the final users.

As time has gone by, testing has become more and more important, and today, there is no application, which can go live, without a series of different tests: Unit Test, System Integration Test, System Test, User Acceptance Test, Load & Performance Test, Stress Test, Penetration Test, Security Test, etc.

The tests are usually performed on a test environment and rarely or only during the last phase of User Acceptance Test (UAT) on a real physical system.

Most of the above-described test types are covered by ISTQB® Syllabi (www.ISTQB.org).

The rest of Chapter 1 will give enough information about test types, as e.g., manual, automation testing or Unit testing, System testing etc., so that you do not need to search information in ISTQB® syllabi or glossary.

Please refer to chapter 0.12 “Definitions”.

1.1.1 Why testing is necessary

Testing became more and more interesting and for several years it has been seen as essential to a successful product launch.

By the different phases of testing, different types of defects can be detected.

As an example, during the Unit test, wrong implementation of requirements or wrong programming would be detected (somehow “white-box” test), whilst in System Test the End-to-end test will be performed (somehow “black-box” test) and not all programming errors will be detected.

A test is necessary to detect as much as possible and as early as possible any defect which would bring the application to an erroneous behavior with consequent e.g., application crash or loss of data.

An early defect detection during integration test phases, will be much cheaper to fix rather than when the application is installed e.g., on hundreds of devices, and the whole process (defect fixing, regression test, System Acceptance Test, application Launch, Release Notes, etc.) needs to be planned.

1.1.2 Manual Testing

Manual testing is a process, where the software tests are performed manually by a human tester. Analysis and evaluation of the applications' functionality, security, usability, performances are done by mean of a user in an explorative process. This ensures that the application responds to user-friendliness. This type of testing is highly time consuming as execution can take sometimes, and defects tend to take time to get recognized. Therefore, as a rule of thumb, a maximum of 15-20% of an application's testing, should be performed manually, whilst the rest should be automated. On a mobile application the automated part can reach up to 95%.

Note: 80-85% of automated test scripts, is generally accepted as a good goal, however the percentage can differ depending on several factors.

The fastest way to perform a test can be synthesized as:

- Definition of test scenario
- Creation of test data
- Perform the test
- Create a report with evidence of performed tests

On a GUI based system

- prepare set of test data: feed the test data in a structured file or file such as spreadsheets
- go through the menus (if the application is GUI oriented)
- enter the data and check the results if they met with the expected one
- create a report

On Embedded environments (e.g., generation of a digital signal through an interface)

- prepare set of test data: fill in predefined test data in a file such as excel or word documents (e.g., Voltage, Current to be entered, measured)
- go through the different circuits (points of measurement)
- given the e.g., input Voltage, check if the measured value is correct
- Enable/disable some switches and perform same measurements
- create a report

On a Mobile Device

- define the test scenario
- prepare set of test data
- download the application to be tested to a physical device on an emulator or simulator
- execute test cases
- create report

As you can see, although different type of environment will be tested, the main tasks are the same

- Identify test scenario
- Prepare set of test data
- Execute test cases
- Create report

In case of small application with few check points to be tested (e.g., few menus on GUI) and assuming a test phase to be performed on monthly basis with an effort of say, 2 days, then a manual test could still be a good and cheap solution.

1.1.3 Automated testing

Automated testing is a very important approach to mobile application testing. In this process, a set of test cases are set up which should generally cover 80-85% and more of the testing process.

The percentage is a general guideline followed in the software industry. Here is a list of test cases that are generally performed through this approach.

- Automate test cases that can be automated.
- Automate test cases for most frequently used functionality.
- Automate test cases to be executed on daily (new build) basis.
- Automate most tedious manual test cases.
- Automate test cases that are highly time consuming or even impossible to perform manually.
- Automate test cases with predictable results.

We have several systems to be tested, following the examples from the previous chapter

In a GUI based system

- Fill in predefined test data in a database or in a flat file such as excel or word documents
- Go through the GUI menus (e.g., 50 different pages)
- Enter the data and check the results if they met with the expected one
- Create a report

On a Mobile Device

- Define the test scenario (application to be tested, environment to be tested, etc.)
- Prepare set of data
- Download the application to be tested on the device (is the test will be executed on real device)
- Download the application on an emulator (usually a virtual device which act as a real one)
- Check for all menu entries, application behaviors under different environmental setup, modified in setup mode outside the application
- Create reports

As you can see, different type of environment will be tested, the main tasks are the same

- Identify test scenario
- Prepare test data
- Execute test cases
- Create report

The main differences between manual and automatic test execution for the above example, can be summarized as

- The number of test cases executed for the same lapse of time is higher using automation tools compared to manual execution.
- Automated tests have a high degree of repeatability. This implies that the same automated test will be done exactly way each time it is executed, and the same exact execution steps can be extended to a range of operating systems and devices. The same is however not true for manual tests as executions steps and speed vary from person to person.

Here it is visible that scenario and test data preparation after the first creation are NOT being changed often, test data could be almost the same, but the test execution of same test cases set on several devices with each different setup (environmental setup), makes the difference between manual and automated test execution.

Given the following example:

- 50 test cases to be executed for a given application on a given device
- test cases execution duration for manual test 2 Days (PD)
- test cases execution for automated test 0.3 Days (PD)
- 8 different Test environment (devices) (we can assume also same device with different setup)

Test Env	#TC	Dur Man	Dur Aut
1	50	2	0.3
2	100	4	0.6
3	150	6	0.9
4	200	8	1.2
5	250	10	1.5
6	300	12	1.8
7	350	14	2.1
8	400	16	2.4

We see that test automation will save approx. 15 days of execution for a set of tests for a release

Getting a new application to be tested on monthly basis, the Return of Investment (ROI) is immediately visible

Note: The time spent to automat and maintenance is not taken in consideration in the example.

1.2 Introduction to Mobile Application Testing

For mobile device we intend all devices which can be connected via cable to a server or other system, or be able to work alone, i.e., connected to a server via WiFi (wireless).

A mobile application needs to be tested and the Mobile Testing is the process to go through all checks.

The goals are to ensure that the expected functionalities work as defined in the Acceptance criteria.

Mobile applications have an extremely short time to deliver therefore it is extremely important to have a core of test cases (scripts) to be performed as much as possible, automatically and for a minimal percent manually.

1.2.1 Advantages of Mobile Testing Automation

Automation of mobile testing has proven to be helpful. The list below is a non-complete but exhaustive list of main advantages about automation of mobile application testing:

- Enable the automatic re-execution of a set of test cases, after every code increment or change
- Increases efficiency of testing and reduces human errors.
- Helps enhancement of regression test execution
- Saves a lot of time in comparison to manual test execution
- Test scripts can be executed in parallel on multiple devices (real devices, emulators, simulators)
- Same test suites can be performed repeatedly

Mobile applications are tested against:

- Quality
  - Does the application exactly perform what is foreseen?
- Security
  - Is the mobile application secure to stop/block any attack by external (e.g., hack) or internal (e.g.. malware Software)?

- Usability
  - Is the application user friendly (ease of use)? Is the application self-explanatory? Is there availability of help?
- Adaptability and interoperability (Different platforms)
  - Is the Mobile application correctly resized on different devices (e.g., screen 5, 5.1 or 7 inches) with all menus correctly shown?
- Synchronization, Connectivity
  - Is the mobile application able to connect to the server (e.g., Telecom, Swisscom), over 2G, 3G, 4G, 5G, edge?
- Rollout
  - for mobile application, time to market is a vital factor. Defect Fixes, new functionalities, new services need to be delivered almost on daily basis,
- Installation / download
  - How easily can the application be installed, removed, and reinstalled, updated. Does the procedure work as expected?

We can easily understand that the Quality, Security, Usability can be “somehow” automatically tested, but most of the checks need to be performed manually.

Other is the adaptability test; the functional test as well the application installation, should be tested automatically, because of the need to deliver the application on daily basis. The execution of above points, need to be automated and be executed and repeated on different devices with different processors, memory, screen size, etc.

1.2.2 Some ideas to create good, automated test cases

To create good and predictive working automated test cases, to be executed automatically, it is necessary to have a good understanding of the application to be tested, all functional and non-functional requirements need to be analyzed.

Any clarification and agreement should be taken together with the product owner and the developer. As outcome, the "should" and "must" are clarified. The team then knows what should be (recommended) and what must (mandatory) be tested together with the corresponding priorities of the tests.

Test Automation does not come for free, and in most cases, it requires continuous maintenance. The assumption that an initial investment would be enough to run automation forever, is unfortunately not true. Only when the application under test is not changing, then the automation maintenance requires in general small or ideally no effort at all.

The testing team or the responsible people for testing (manual and/or automation) needs to consider several action points and get ready well before they get the application to test and make sure those are respected and healthy during the testing time:

- Test environment (Real device, emulator, simulator)
- Test Data
- Test Framework (ideally adapted and integrated to the adopted software development practices)
- Reporting and Monitor/Logging tools (status, evidences, execution time, etc.)
- Test Supporting Tools (create/maintain/execute test cases, defect reporting tools, etc.)

And from planning perspectives

- Plan approx. 10-15% of testing budget for creation, maintenance, execution and reporting of automated test cases
- Calculate a ROI to be shown to the management
- Plan the time for execution, such as daily, after a new build and full or reduced test execution

Automating test cases is becoming a more and more an exciting activity and many developers decide to change from “only” software development to test automation.

The next list (not complete) shows some advantages and disadvantages in automatic or NOT automatic.

Main advantages of automation:

- Running automated tests case is usually more efficient and less time consuming than running same test cases manually.
- Some test cases can be performed only automatically (e.g., fast reaction time of a given component).
- A simulation of simultaneous access by several users (not Load & Performance test) can be easily simulated by test automation and not possible by manual testers.
- Perform full regression test or health check test following a new build.
- Perform automatic test in parallel to other activities (testers can work to e.g., new test cases creation).
- Test cases can be performed during the night or non-working days.

Main disadvantages of automation:

- Not all manual test cases are clearly defined and therefore the automated could not execute correctly.
- The efficiency of automation tests is highly dependent on the requirements. If the requirements are incomplete or continually changing, then the automation tests will not bring much gain.
- Continuous change in requirements.
- A Set up for testware, test infrastructure can be very expensive or complex (over- or under- dimensioned for the real needs).
- Tool and test environment set up / maintenance costs can be high.
- Outsourcing or contracting of automation tasks may not be cost effective on the long term. Developing in house competence may be a better option.
- Writing ad-hoc test application, and not adopting e.g., the DDT technique.

Some ideas to reduce disadvantages and increase advantages

- Define clear expectation of test automation execution, in terms of
  - Execution time
  - Criticality of performed test cases: automatic vs manual execution
  - Amount of test cases to be executed
  - Duration of manual test execution vs. same automated
- Get motivated and well-trained testers (knowledge of testware, SDK, application)
- Keep the testers (continuity and motivation)
- Writing reusable test application vs. ad-hoc application
- Adopt whenever possible DDT technique. It will reduce the software maintenance costs
- A critical Proof of Concept before starting the new journey to automate at any cost is highly recommended.
- DO NOT AUTOMATE at any cost.

1.2.3 Differences between Manual and Automated Test

What is the main difference between a manual test and an automated test?

A manual test always requires a human intervention. The tester is the active and critical path for all testing activities. He has to start application, enter any type of data, confirm, check for results and compare them vs expected results.

A test automation is an automated engine (e.g., an ad-hoc application), capable to perform several preprogrammed instructions in each sequence or randomly, without any human intervention and repeated for an indefinite number of times.

We try to give a definition of test automation in a compact way.

Test automation englobes both test scripting and test execution. Execution happens in a predefined sequence using predefined or dynamic set of test data, without or with limited human intervention and completes with a human friendly report (HTML, flat file, etc.) with all performed tests and results.

A well-defined test automation framework should be able to execute all planned test cases, regardless the test result itself, i.e. a positive result as well a negative or discrepancies between expected and real result, should not bring the test application to crash. At the end of the executed test cases set, all executed scripts will generate a human readable report. Below an example of minimum reported information:

ID	test case name	Execution Result	Expected Result	Evidence	Screenshot
001	Test_0001-Case1	Passed	Passed	Positive test	ScreenShot1
002	Test_0001-Case2	Failed	Passed	Positive Test	ScreenShot2
003	Test_0005-Case1	Failed	Failed	Negative Test	ScreenShot3

Figure 1 : Test execution report (Example)

As you can see, in the table above, the Test_0001-Case2 is failed, whilst the expected result is to pass, but the execution was not interrupted.

What is the main difference between test automation and mobile test automation?

The difference between a test automation and mobile test automation is mainly given by testing on a fix wired systems like an application on a desktop (e.g., a GUI to interact with some data) and mobile devices like mobile phone or any devices usually connected via Wireless connection.

In this syllabus we will talk mainly on mobile test automation.

1.3 Manual vs. Automated Tests

In the early days of pioneering test automation, a lot of tool-robots have been made available on the market.

The chimera to perform long test activities sometimes on critical devices reducing human risks and moving the test activities from human to a machine (Robot) was the enormous success.

Unfortunately, the technology was too young, and it was clear within a short time that automating was not bringing any payback and the excitements was degreasing.

As result more and more test cases were created, executed, and maintained for manual execution.

Over the time and after every release the number of test cases were increased, and a maintenance was more and more difficult. In absence of real working automation tools and extremely high cost did not help to start moving towards automation.

A shift-left to a new era of test automation was given by some tools “easy to program” by using visual basic programming language, predefined library such as “wait for a given time” or “wait until the enter key is pressed”. The framework was very heavy in terms of system resources needs and with low performances, nevertheless the way to restart test automation was given.

The user library and framework became more and more lean, reliable, and faster.

Some new features like record the user activities and play back were helping in test automation, in fact the user had to perform once the test execution and the recording feature was able to reproduce one-to-one the user activities by changing data (yes DDT was starting)

After the first visible successes, departments within companies as well as new companies were investing in test automation tools, framework, techniques, etc. The new era of test automation started.

At the beginning there was not a clear separation between software developer and automation tester. The first test automation engineer was defined as “poor developer”, over the time they became more important and now the test automation engineer is same if not better than a software developer in terms of product knowledge.

A test automation engineer must know the application in its completeness, how it is developed and how it works resp. behaves to create test cases to detect as much as possible failures.

1.4 Success Factors of Automated test cases

As described in the previous chapters, to automate test cases it is to have a good preparation and knowledge for both tools and application to be tested:

- Management helping and supporting the team
- Motivated and stable team
- Enthusiastic perform test to find failure and make the product more robust)
- Clear goals
- Good and stable automation tools
- Stable and reliable test environment, including simulators and /or emulators
- Supporting process, well established SDLC within company
- Clear and stable requirements

The clear definition of what can and what cannot be tested and / or automated

- Units are usually tested within unit testing
- GUI: are user interfaces working as expected?
- API
- Data access (store and retrieve data)

All above test areas can be easily tested by the developer or by tester by mean of Unit test, using Mock data and standard unit testing tools.

Same unit test can be automated and performed as part of regression test during every new software build.

Starting from System Integration Test till System Test and possibly User Acceptance Test, most of the foreseen test cases can be automated. Simple example

1. Start the application
2. Login into
3. Enter some data and store it
4. Retrieve some other data and check for quality
5. Logout

All the above test phases can be automated, if the application plans to use test data read from any file (e.g., Excel) and not being hard coded, same test cases can be repeated with different data.

As an example, the login name “Albert Einstein” which would allow the user to login, could be duplicated in “Albert Einst**ie**n” or “Albert **1**stein” and check the reaction of the application, such as “user unknown”, or “illegal Character in the username”, etc.

Given this example we can also create a simple Proof of Concept if test cases automation would help.

Assuming the following test cases resp. steps needs following time to execute

Step ID	Case	Manual exec (Secs.)	Automatic exec (Secs)
00	Starting script and load data	N/A	15
01	Start the application	10	3
02	Login onto	20	3
03	Enter some data, store them	30	5
04	Retrieve some data, check quality	30	5
05	Logout	5	1

For just one test execution we need 95 secs for manual test and 17 secs for automatic test + once 15 sec during the initialization Phase for test data load.

In this example we see the automated test execution completes 5 time faster than a manual test, and if we perform the same cycles by changing the e.g., username 10 times, we see immediately the ROI. At other times it is not easy to see, but such evaluation is crucial to decide whether to automate or NOT to automate and achieve success or not.

1.5 Types of Automated test cases (some ideas)

As discussed in the previous chapters the automation of test case at any cost is not a good practice.

Moreover, having a huge set of test case automated, it is not always good to execute them after every e.g., new build due to possible performance issues.

To optimize test case execution, reduce unnecessary maintenance, creation of huge report, data analysis, the automated test case should be prioritized.

Given the test execution requirements, the test case should be grouped in test suites to execute them separately or as a whole.

Given following requirements (example)

- We have 6 builds per day, so a Smoke test needs to be completed within 7 minutes
- We need a fast regression test suite to complete within max 60 minutes
- From 23:00 till 6:00 no new builds will be created
- We have an official build every Friday from 16:00

The test automation team should prioritize the test case. They would be grouped in test sets to be executed in the given sequence (of time). As a result, the following table could be created

ID	Test Type	Max duration	Used Test Set	# of TCs	DDT (Test Data)
01	Smoke test	7 min.	SMOKETestSet	15	1
02	Fast Regression	60min	FASTRegrTestSet	149	10
03	Full regression	3 hours	FULLRegrTestSet	894	50
04	Official build test	48 hours	FULLRegrTestSet	894	1500

The number of test cases are decided based on type of test cases itself (MUST, SHOULD, COULD) and the time available to execute the test:

- MUST (Priority High)
- SHOULD (Priority Medium)
- COULD (Priority Low)

An Initial “Dry run” execution could help indicating the test cases execution duration.

Depending on test environment the test cases execution can considerably vary.

We can easily add or remove the test cases in the given Test suites, the only to perform all the test cases in the maximum given time, as per test cases automation we can perform the Test Set giving more, or less input data, to cover more scenarios.

Please see the ID-03 and ID-04 in the above table, the same test set are used, however, given the time for the execution, for the ID-04 (running over a weekend) we can perform check with huge number of test data.

Note: This is an example and not necessary cover all real examples, please use it only as a clue.

1.6 Test reports by Test automation

Once the test cases are automated (see above manual vs. automate), it is necessary to have an easy-to-understand report. In the chapter “Differences between Manual and Automated Test” (Chapter 1.2.3 above) there is a definition of a minimum report. The most recent tools provide a deep information with all test steps executed, the execution time and duration, link to the test step, evidence on data, screenshot, link to build number, number of execution cycle, etc.

All outputs are generated automatically by the framework and available to the user (usually test automation engineer), most of the reporting information can be selected or deselected to be part of the final report for stakeholders.

The analysis of report helps to understand if

- All planned test cases have been executed (possibly human mistake in list definition)
- Which test cases have been correctly performed (regardless the expected result).
- The execution time is in the given expected range (e.g., DB Deadlock could prolong execution)
- The overall system behavior reflects the expectation (duration time, etc.)

An expert test analyst can analyze the report in a short time by checking the overall result or detailed information and propose the Go for further tests activities (manual of automated) or NoGo reject the new build and revert back to developer to be fixed (e.g., MUST test cases execution did not pass).

The test report should be available to all project’s members including business stakeholders.

2 APPIUM® Introduction

Timing	4*K2, 2*K3, 1*K1 = 100+150+10 = 260 minutes
Terms	Mobile App, native app, emulator, simulator, hybrid app

Topics & Learning Objectives for this Chapter:

Learning Objectives

ATF-2.1	(K2)	What is Appium®
ATF-2.2	(K2)	Appium® Philosophy
ATF-2.2.1	(K2)	What Appium® is?
ATF-2.3	(K2)	Why using Appium® is a good choice
ATF-2.4	(K3)	Appium® Architecture
ATF-2.5	(K3)	Emulator vs. Simulator vs. Real Devices
ATF-2.6	(K1)	Appium® standard Repository

ATF-2H-01	Why to use Appium®
ATF-2H-02	When to use Appium®
ATF-2H-03	Appium® client-server architecture
ATF-3H-04	emulators, simulators

2.1 What is Appium®

Appium® is an open source, cross-platform automation testing tool developed as an addition to the already well-established Selenium framework. It has been since 2012 and officially presented during a Selenium conference. Finally, the project was donated to the JS Foundation in 2016 [cfr. JSF-01].

Appium® has been developed with the main goal of running scripts automatically to test native and hybrid mobile applications on iOS, Android and Windows platforms.

A native application is that one developed to run on iOS or Android systems using the SDK development platforms. Refer to chapter 0.11 "Acronyms" oben for detailed info about apps types.

- Mobile web apps
  - Websites accessed using a mobile browser, e.g., Safari on iOS, Chrome on Android
- Hybrid apps
  - apps with a native container and one or more Web Views embedded in that container. The Web Views are little frameless browser windows that can show content from the web or from locally stored HTML files. Hybrid apps allow the use of web technologies within a native-like user experience
- Native app
  - apps built using the native mobile SDKs and APIs

Appium® can now also support testing of desktop applications on Windows Operating System using an automation interface software called "WinAppDriver".

2.2 Appium® Philosophy

The Appium® philosophy is based on the idea to avoid any changes in the native application, leading to recompilation of the app to perform any test.

Automate mobile app testing by using any supported programming language (see below some of them) and test framework with full access to interfaces.

What you test and validate, will be delivered to the final user as it is, no changes necessary.

Most of the popular programming languages are supported by both Selenium and Appium® WebDriver API client libraries, C++, C#, Java, JavaScript, Python as an example.

The next list summarizes the main points about Appium®'s philosophy

- No need to recompile the original application for test purposes, what you test will go live.
- No limitation imposed on programming languages. Most popular programming languages are supported.
- Test are performed on test environment which are production alike.

2.2.1 What Appium® is?

Appium® is an open-source automation tool managed by JS-Foundation [JSF-01]. It is free to be downloaded and modified.

Appium® client is not developed for a single programming language, it supports all languages supported by selenium libraries, such as C++, C#, Java, etc.

Appium® does not need any special tool to be recompiled. The server can be installed as is.

Appium® is not limited to a specific framework.

2.3 Why using Appium® is a good choice

Appium® consists of 2 main components

- Appium® Server
  - It is responsible to convert the user commands to the real automation steps.
- Appium® client(s)
  - There are different Appium® clients, one for each supported programming language.
  - All clients offer same features, regardless the used language.
  - It offers API interfaces to perform a predefined set of actions.

The client part is created by the user as per their needs. The created test cases once checked for quality and working as expected, can be reused for other frameworks, mobile devices, without any need to adapt or recompile.

Note: due to some differences in Android and iOS capabilities, some adaptation may be necessary.

Apart from iOS, Android's adapters, specialized for the different architectures, the application does not need to be recompiled or configured to be executed on test environment or real device.

The use and reuse of mobile automated test cases running on Appium® for different architectures, devices or applications does not generate any cost.

There are several tools like Appium®, such as Calabash, MonkeyTalk, Earlgrey, however, why we should choose Appium®?

The next list is a non-exhaustive list to support the decision to use Appium®

- Appium® is open source and free of cost of use.
- No need to change or rebuild the application (what you test, goes live)
- Great support from Appium® community
- Lot of examples available and ready to use
- Huge and detailed online documentation
- Multiplatform Support: Appium® supports all mobile applications running on Android, iOS, Windows, running either on physical device, emulator or simulator.

2.4 Appium® Architecture

Appium® is a Node.js server communicating over HTTP that creates and handles WebDriver sessions.

The Appium® is like Selenium®, it receives HTTP requests from client with JSON payload and handles them.

The next picture (Figure 1 below) shows in an all-in-one picture a client-server connection.

Appium® is based on client-server architecture.

A client connects to a server via HTTP protocol request. The Server creates a session assigning an unique session-id, all the next test activities will be executed as part of that session. So multiple clients can be tested in parallel by creating different session.

➔ Appium® server is at its heart, a server that exposes REST APIs. It receives requests from client(s), listens for commands, executes them on a mobile device, responds with HTTP response representing the result of the executed command.

A native application can run on a physical device or on a emulator/simulator.

A session is a medium to send commands to the specific test application; a command is always performed in the context of a session.

In the next picture, both emulators and physical devices are connected to the Appium® server, as a full scenario.

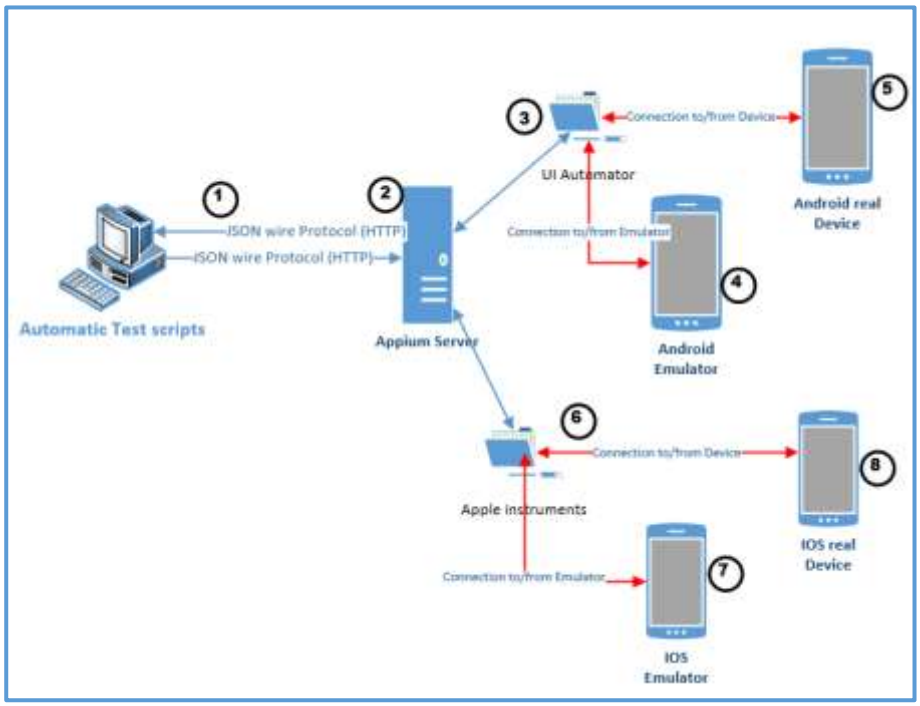


Figure 2 : Test environment (full test environment)

Explanation for Figure 2:

1

Automatic tests scripts

1. User created test cases

2

Appium® Node.js server

2. The transport mechanism mobile JSONWP, “JSON Wire Protocol” is a specific set of predefined endpoints exposed by REST API.
3. The Appium® server engine, receives commands from user, dispatches them to the correct sessionId (in the example above to the connections (3) for Android and (6) for iOS devices. Any answer coming back from the connection’s points will be sent back to the test script by mean of sessionId.

3

UI Automator Framework

4. For Android V.17 or above the UI Automator FW
5. For Android Version below V.17, Selendroid FW should be used.

4

Android emulator

6. The Android emulator for a given device (e.g., HUAWEI Pro 20) where the mobile application to be tested is download

5

Android Real device

7. The End final device where the mobile application to be tested is download

6

Apple Instruments

8. Message dispatcher to emulator, real devices

7

iOS emulator

9. An iOS emulator for a given device (e.g., iPhone 12) where the mobile application to be tested will be download

8

iOS Real device

10. The End final real device where the mobile application to be tested is download

2.5 Emulator vs. Simulator vs. Real Devices

To reduce a need for testing only on real devices, Appium® supports both real devices, emulators and simulator.

What is an emulator?

An emulator is a mimic of hardware and software of a given real device.

- It uses the same software as installed on a real device.
- Most of the peripherals and sensors are emulated as well.
- The user "should not distinguish the behavior on an emulator vs a real device, apart reaction time and other performances.
- The main advantage of emulator is to behave as it would be a physical device.

What is a simulator?

- It is not a replication (mimic) of a real hardware and software.
- Usually uses predefined patterns (e.g., setup a sensor will return "setup completed" but no behavioral changes in test execution (light on will return "light is on" and light off command will return "light is off" any further check f the light is on, could report "light is off".
- The main advantage of a simulator is to see if the application is working, regardless of the real peripheral behavior.
- In Appium® emulator are used to reflect the real behavior.

Pros and Cons of emulator / simulator

- The usage of a simulator is encouraged during the first phases of development, something like System Integration Test (SiT) or Unit Test (UT). This because during the first phase of development (or test), the main flow (behavior) is tested, regardless the real functionalities.
- However, the maintenance of device simulator is too expensive and used if no emulator is available.
-
- An emulator should be used during System Test (ST) / Functional Test (FT), Non-Functional Test (NFT) and User Acceptance Test (UAT) for a given application.
- Due to the low maintenance cost and high scalability, the emulator is preferred during all the testing activities for a mobile application.
- An emulator can be set up to run with different Operating System versions.

Pros and cons of using physical devices for application testing:

- High initial cost to buy physical devices
- A physical device will be obsolete in a short time and the updates will be discontinued after a relative short time.
- Setup of a physical devices, following a high initial cost, need to perform regular OS updates or in special cases keep the same OS version (disable any update).
- Final test on physical device will be encouraged, however to avoid huge investment, it would be good to perform a survey or market check, to understand which devices are in use at most and reduce the physical test on the few devices (e.g., HUAWEI P30 with OS Version xx.yy and iPHONE 12 with OS Version nn).

2.6 Appium® standard Repository

The Appium® kit can be downloaded from the GitHub Appium® repository https://github.com/appium/appium-desktop. The download of desktop version which has a nice graphical interface, should be preferred.

Alternatively you can download the tool from http://appium.io/, JS-foundation [JSF-001] repository.

You can download Appium® for Mac, Android, Windows as image (installer) or source code.

Check https://github.com/appium/appium for Main Appium® repository and https://github.com/appium/appium/tree/master/sample-code for examples on several supported languages.

The actual stable version of Appium® as of January 2022 is the **1.22.0**.

Please check in the above pages for new version in either stable or development stage.

3 APPIUM® - Installation and Setup

Timing	4*K2 + 1*K1 = 110 minutes
Terms	Android, Sdk, Setup, System variables

Topics & Learning Objectives for this Chapter:

Learning Objectives

ATF-3.1 (K1) Installing Appium®
ATF-3.2 (K2) Install Appium® on Windows system
ATF-3.2.1 (K2) Installation of all necessary components and tools
ATF-3.3 (K2) Install Appium® on MAC system
ATF-3.3.1 (K2) Installation of all necessary components and tools

ATF-3H-01 (P38) Appium® installation

3.1 Installing Appium®

Appium® is not just a single binary; it requires following environments such as Android_SDK and Java Development environment (assuming we are using Java Appium® client). System Environmental Variables need to be created or modified to accommodate the new tools' location (e.g. Javapath, Path, etc.).

The next steps are shown as an example and do not replace the way how to install the tools as explained in the official Appium® website https://appium.io/ .

The screenshot in the following sections describe a typical Appium® installation.

Several packages need to be installed before using Appium®, some of them, e.g., Java, could be already installed on your system, in this case check the minimum version to be installed. If necessary, you need to update to a new version of Java.

The next list shows the tools, libraries to be downloaded and installed to a local system (e.g., your PC)

- Java JDK
- Node.js (if make use Installation via NPM)
- Appium® Server
- Appium® Client Libraries (downloaded using Maven or as Zip)
- Android SDK is needed
- Xcode (Mac)

3.1.1 Prerequisites

Before you start using Appium®, you need to install several packages and tools. The installation of the Appium® framework is a simple process, if all works as expected it will be ok, otherwise some administration experience such as changing some environmental system variables would be needed. In the next pages in this chapter, you will see an installation example with some screenshots to show what is expected to see or necessary to check following an installation.

You can download and install Appium® via NPM or by downloading and installing Appium® Desktop.

In this syllabus we follow the installation of Appium® Desktop, which is a comfortable Desktop-based installation procedure to launch the Appium® server.

The following assumptions are made in order to reduce the description to a single environment, although the installation to other environments is almost the same.

- Android Environment
- Windows 10 based Host machine, 64 bits.
- Used programming language Java
- Programming knowledge of Java [and Selenium]

3.2 Install Appium® on Windows system

The next lists will show what needs to be downloaded and the order of installation. Please note some tools or libraries can be already installed or present on your system:

- Download
  1. Android Studio
  2. Appium® Jar files for Java
  3. Appium® Desktop
  4. Java
- Install
  1. See next chapter 3.2.1 unterhalb

3.2.1 Installation of all necessary components and tools

We need to install Java environment, Android Studio, and setup some system variables.

The next chapters will give an overview of a typical installation procedures. Of course, this is an example and different ways to install all the tools can be used. For a detailed installation check, please refer to the student exercise document (should be shared during the training).

3.2.1.1 Part-1: Installation and Java Setup

Download and install the latest version of Java sdk (*as of 1.1.2021*, jdk 15.0.2 which will be used in this syllabus)

- Compressed Archive → jdk-15.0.2_windows-x64_bin.zip
- Image → jdk-15.0.2_windows-x64_bin.exe

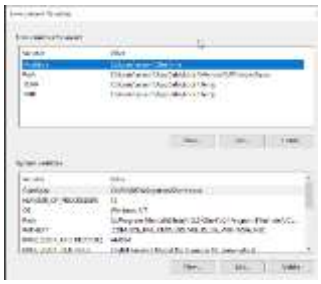
Figure 3 : Java jdk Installation and setup

Follow the installation instructions, after a short time the installation will be completed.

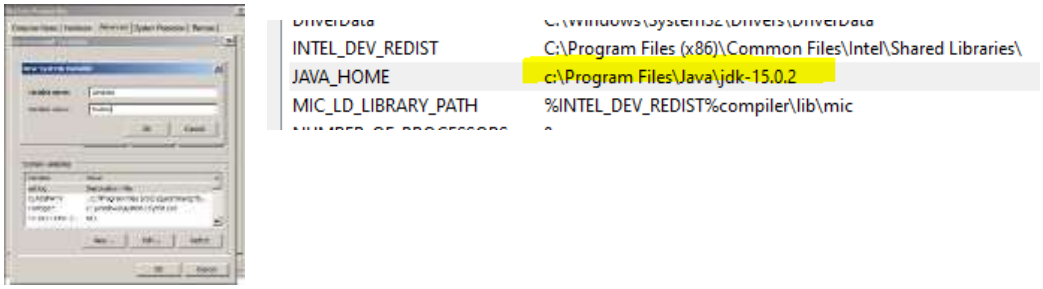
Please refer also to https://docs.oracle.com/javase/tutorial/tutorialLearningPaths.html for documentation.

Setup (create or change) Java the environmental variables : JAVA_HOME and PATH

- The Java_HOME points to the installation directory, default **"C:\Program Files\Java**" resp. e.g.: "**C:\Program Files\Java\jdk11.0_10**"
- In windows Start (⊞) search for system variables

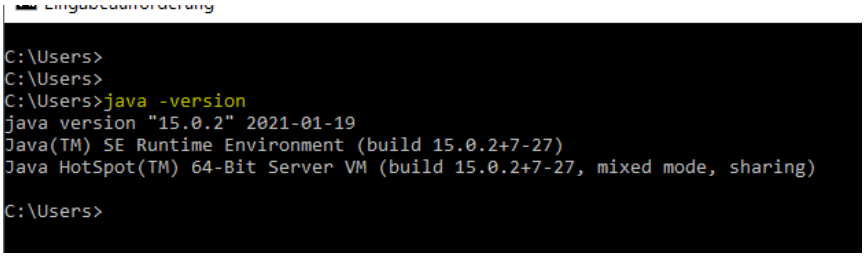


- If necessary, create the variable JAVA_HOME



- And add the new variable (JAVA_HOME) to the PATH path.

Note: Once the installation is completed, it is possible to check the java version by entering the command "java -version" in the command file "Dos command line".



3.2.1.2 Part-2: Android Studio Installation and Setup

Open in browser the page https://developer.android.com/studio, download the Android studio (as of 1.1.2021 the version is 4.1.2 for Windows 64-bit) and install it.

Figure 4 : Appium® Installation and Setup

Create or change the address to ANDROID_HOME system variable ANDROID_PATH = (*usually is the following path*) "C:\Users**\name**\AppData\Local\Android\sdk" where "**name**" is your user name, and add to the PATH the following paths:

%ANDROID_HOME%\platform-tools
%ANDROID_HOME%\tools
%ANDROID_HOME%\tools\bin

To check that the installation is correct, start the Windows console command and enter the command "sdkmanager –list"

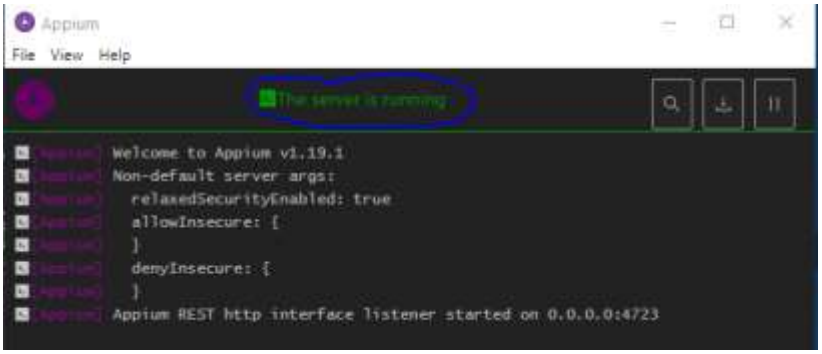
If you see a list similar to above picture, then the Android installation is successfully completed.

3.2.1.3 Part-3: Appium® tool setup

Download from „https://github.com/appium/appium-desktop/releases" the latest Appium® tool Appium-Server-GUI-windows-1.22.3.exe

Follow the installation instructions, once the Appium® is being installed and the app is start, you should see the following window

To check that the installation is successfully completed, start the server and check if the next window is shown.



3.3 Install Appium® on MAC system

Installing Appium® on a MAC based system is not quite same as for Android for Windows, the next chapters will show and explain the main steps.

3.3.1 Installation of all necessary components and tools

We need to install Java environment, setup some system variables, and install Appium® desktop application. The next chapters will give an overview of the installation procedures.

Note: For a detailed installation check, please refer to the student document extension to this syllabus.

3.3.1.1 Part-1: Installation and Java Setup

The Tools installation on a MAC system is like the installation on Android

➔ Download, install and setup java system environment Variables
- Download Java “dmg” from java official download page
- Install Java, following the instruction given by installation package
- Once the installation is completed, without any error messages, enter on a terminal windows the command “**java -version**”, if You get a java version number, then the installation was successful
- Setup environment variables, on a terminal window enter the following commands
  - vim ~/.bash_profile
  - In insert mode, enter the following elements:
    a. export JAVA_HOME = $(/usr/libexec/java_home)
    b. export PATH = $JAVA_HOME/bin=$PATH
    c. export PATH = /usr/local/bin:PATH
  - Save the changes, close the file (**:wq!**” command)
  - Enable the new PATH: source ~/.bash_profile

3.3.1.2 Part-2: Installation of XCode and Homebrew

The installation of Xcode will be the next step.

- Search for Xcode on app store and install it
- Install homebrew [cfr Appendix C (6a)], by entering the code shown below on a terminal window
  - /usr/bin/ruby -e "$(curl -fsSL - https://raw.githubusercontent.com/Homebrew/install/master/install)"
- Execute the following commands
  - Brew update
  - Brew doctor
  - xcode-select –install (to complete the installation of XCode)

Depending on using Windows or MAC you should have now installed the Appium® framework on your machine.

4 Using Appium®

Timing	2*K1, 7"K2, 3*K3 = 20+90+225 = 335 minutes
Terms	Android, iOS, Sdk, Setup, System variables, Android Studio, Appium® Server, Xcode, sync & wait mechanisms, maintainability (of test scripts)

Topics & Learning Objectives for this Chapter:

Learning Objectives

ATF-4.1	(K1)	Main pre-requisites
ATF-4.1.1	(K2)	Typical test script structure
ATF-4.2	(K1)	Writing a test procedure on Appium® 1.x
ATF-4.2.1	(K3)	Writing a test procedure for app (explanation of Body)
ATF-4.2.2	(K2)	Writing a test procedure for Android app (Example-1)
ATF-4.2.3	(K2)	Writing a test procedure for iOS app (Example-2)
ATF-4.2.4	(K2)	How to create a test script - step by step
ATF-4.2.4.1	(K3)	Start the Appium® application
ATF-4.3	(K2)	Executing a test procedure on real device
ATF-4.4	(K2)	Sync (Wait Mechanisms)
ATF-4.5	(K2)	Logging Mechanism
ATF-4.6	(K3)	Maintainability of test scripts

ATF-4H-01	Creation of a test procedure, several exercises
ATF-4H-02	Wait API (Explicit, Implicit)

4.1 Main pre-requisites

Developing test automation scripts with Appium® framework could be done by using any of the main programming language as a client (Ruby, Python, Java C# (.NET) and JavaScript).

In this document, will be used the Java implementation of the Appium® client libraries to show some test examples.

Similar procedures, as described here would be done, if, for any reason, is chosen a different client language to implement the test scripts.

To start writing a script with Appium® a set of software tools must be installed as pre-requisite:

- **Java JDK**: to compile and run Java code.
- **Android Studio**: IDE to develop test scripts and manage SDK and emulators for Android app.
- **Appium® Server**: the core of Appium®, it can be installed local or in a cloud and it serves to handle the communication between the chosen client language and the mobile app.
- **XCode**: IDE to develop test scripts and manage SDK and emulators for iOS app.
- **Maven**: to manage the project's build, reporting and documentation.
- **Junit or TestNG**: framework to support test development, their execution and reporting.

In the next chapter a list of common, Android and iOS capabilities are listed, together with some examples. Snippets of code will be used to demonstrate working examples.

4.1.1 Typical test script structure

The test automation framework where Appium® is integrated can be complex or less complex based on project needs.

In general, in a layered architecture framework the Appium® libraries needs to be integrated with several other components for reporting, exceptions handling, database connection handling, logging handling, authentication, localization, integration with third party tools and others.

Here we focus only on the Appium® test script and what is the minimum needed to run it.

The next sub-chapters show an example of creating a test script using e.g. JUnit.

A test procedure follows the programming language syntax and guidelines, in the example for this syllabus and student exercise document, the Java programming language is used.

Start with some directives to force the execution of a piece of code at a given phase, such as before the test start, or just before the test ends.

Following section will serve to better understand the code examples snippets described in 4.2.2, 4.2.3, 4.2.4. Let's got through the main blocks to construct a test script.

Dependencies are loaded by the maven pom.xml file and here below are the main to include in it:

```
<dependencies>
  <!-- https://mvnrepository.com/artifact/io.appium/java-client -->
  <dependency>
    <groupId>io.appium</groupId>
    <artifactId>java-client</artifactId>
    <version>LATEST</version>
  </dependency>

  <!-- https://mvnrepository.com/artifact/org.seleniumhq.selenium/selenium-java -->
  <dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-java</artifactId>
    <version>LATEST</version>
  </dependency>

<!-- https://mvnrepository.com/artifact/junit/junit -->
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>LATEST</version>
  <scope>test</scope>
</dependency>

</dependencies>
```

A typical test script is structured in several parts: Import of standard libraries, @Before, @Test, @After

- **Imports**
  Import of standard libraries and Appium® specific libraries need to be done.
  Example of needed object to import: MobileElement, AndroidDriver, DesiredCapabilities
- **@Before**
  The directive will ensure that methods and anything defined following the declaration itself, will be executed before the execution of any test.
  The @Before contains for example all setup and initialization tasks to be execute prior any other command sent to the device (emulator or real device).

- **@Test**
  The heart of test is defined in method body annotated with @Test annotation.
  It is the main part of execution. Here all interactions with device and the application under test will be performed, e.g., send data or commands to the application under test, wait for a user-input (e.g.,

  by an explicit Wait), etc.

- **@After**
  The directive ensures that all activities defined after it, will be executed as last, before the application will exit. The @After is usually understood as clean-up of any data, reset the device to initial state or quit the communication driver. Most of the commands are not necessary but they can be explicitly executed as part of good programming rule.

Although not mandatory, it is a good practice to instruct explicitly the server (in @Before) and therefore the device or emulator how to set up the device to execute the tests, as an example

- Orientation (Page)
- Setting a timeout to a specified values if the device would not answer

will instruct the server how to set up the (page) orientation, a timeout (to wait) before answering back in case of no actions, etc.

4.2 Writing a test procedure on Appium® 1.x

The next section will show 2 examples written using Appium®, one for an Android app and one for an iOS app. In general, the test should look like the same but what changes is the device (real or emulator) and app setup which is handled by the DesiredCapability of the Android or iOS driver.

Scenario of the test:

- *Loading a mobile app AppiumBank;*
- *Log in with a user JohnB and password "testAppiumBank";*
- *Click Make Payment button;*
- *Enter Payment details (Phone, Name, Amount, Country);*
- *Click Send Payment button;*
- *Confirm Payment with yes;*

The two examples below do contain the sections described previously (Imports, @Before, @Test, @After)

In the Before section is set, which device to use with its properties (deviceName, deviceId), mobile OS to use (platformName and platformVersion), app specific properties (absolutePathToExecutable, appPackage).

The test-method section is the core part of the test which implements the scenario above.

The After-method section in this case is only responsible to quit the driver and graciously close the test.

4.2.1 Writing a test procedure for app (explanation of Body)

Here below you find a breakdown of the sections of a simple test script for an Android app and a detailed explanation line by line of the code.

Adopted standard for explanation, code example

- Explanation test

```
Code example
```

Imported packages section:

- AndroidDriver is the driver that allows to interact with the device
```
import io.appium.java_client.android.AndroidDriver;
```

- AndroidElement is the Mobile App Element on which we are interacting
```
import io.appium.java_client.android.AndroidElement;
```

- Capabilities are used to set attributes to tell Appium® how you want your tests to be executed. The main capabilities are set at Selenium level, but more specific can be set at the device level, hence Android in this case.
```
import org.openqa.selenium.remote.DesiredCapabilities;
import io.appium.java_client.remote.MobileCapabilityType;
import io.appium.java_client.remote.AndroidMobileCapabilityType;
```

- To handle execution parameters and assertion. Here it serves to be able to use the Before and After code sections
```
import org.junit.*;
```

- To identify elements on the mobile app using the selenium API
```
import org.openqa.selenium.By;
```

- Import the Object ScreenOrientation to instruct selenium in which orientation to test the app
```
import org.openqa.selenium.ScreenOrientation;
```

- Used to set the URL where the Appium® server is running
```
import java.net.MalformedURLException;
import java.net.URL;
```

@Before script section:

- Set capability “testName” with a descriptive name
```
dc.setCapability("testName", "Quick Start Android Native Demo");
```

- Set capability “deviceName” to instruct Appium® which device to use
```
dc.setCapability("deviceName",”<device_name>”);
```

- Set capability “udid” to set the deviceId
```
dc.setCapability("udid","<device_id>");
```

- Set capability “platformName” to set the OS to Android
```
dc.setCapability("platformName", MobilePlatform.Android)
```

- Set capability “platformVersion” to set the Android version
```
dc.setCapability("platformVersion", "7.0");
```

- Set capability “skipUnlock” to instruct Appium® to unlock the device or not before the script execution
```
dc.setCapability("skipUnlock","true");
```

- Set capability “noReset” to instruct Appium® to reset the app state or not before the script execution
```
dc.setCapability("noReset","false");
```

- Set capability “autoAcceptAlerts” it instructs Appium® to auto accept platform alerts automatically. This includes privacy access permission alerts. Default is false and is valid only for iOS platform.
```
dc.setCapability("autoAcceptAlerts","false");
```

- Set capability “app” to instruct Appium® the path where to find the app executable
```
dc.setCapability(MobileCapabilityType.APP, <absolute_path_to_apk_file>);
```

- Set capability “appPackage” for the Java package of the Android app you want to run. By default this capability is received from the package manifest. Valid only for Android platform
```
dc.setCapability(AndroidMobileCapabilityType.APP_PACKAGE, "com.appium.Appium®Bank");
```

- Set capability “appActivity” for the Android activity you want to launch from your package. This often needs to be preceded by a “.” (dot). By default, this capability is received from the package manifest
```
dc.setCapability(AndroidMobileCapabilityType.APP_ACTIVITY, ".LoginActivity");
```

- Set capability “bundled” for the app under test. Useful for starting an app on a real device or for using other caps which require the bundleID during test startup. To run a test on a real device using the bundle ID, you may omit the ‘app’ capability, but you must provide the ‘udid’ (appId)
```
dc.setCapability(IOSMobileCapabilityType.BUNDLE_ID, "com.appium.Appium®Bank");
```

- Create the AndroidDriver object with the URL to the running Appium® server
```
driver = new AndroidDriver<>(new URL("https://0.0.0.0:4723/wd/hub"), dc);
```

- Set the selenium implicit wait to 15 seconds. The maximum time which the driver will wait before throwing a not found element exception
```
driver.manage().timeouts().implicitlyWait(15, TimeUnit.SECONDS;
```

@Test script body section:

- Instruct the driver to use the portrait orientation
```
driver.rotate(ScreenOrientation.PORTRAIT);
```

- Identify the app elements (username and password) by id and fill them with required values
```
driver.findElement(By.xpath("//*[@id='usernameTextField']")).sendKeys("JohnB");
driver.hideKeyboard();
driver.findElement(By.xpath("//*[@id='passwordTextField']")).sendKeys("testAppium®Bank ");
```

- Identify the login button element by id and click it
```
driver.findElement(By.xpath("//*[@id='loginButton']")).click();
```

- Identify the Make Payment button element by id and click it
```
driver.findElement(By.xpath("//*[@id='makePaymentButton']")).click();
```

- Identify the app elements (phone, name, amount, country) by id or text and fill them with required values
```
driver.findElement(By.xpath("//*[@id='phoneTextField']")).sendKeys("0541234567");
driver.findElement(By.xpath("//*[@id='nameTextField']")).sendKeys("Jon Baker");
driver.findElement(By.xpath("//*[@id='amountTextField']")).sendKeys("50");
driver.findElement(By.xpath("//*[@id='countryButton']")).click();
driver.findElement(By.xpath("//*[@text='Switzerland']")).click();
```

- Identify the Send Payment button element by id and click it
```
driver.findElement(By.xpath("//*[@id='sendPaymentButton']")).click();
```

- Identify the Confirmation button element by text and click it
```
driver.findElement(By.xpath("//*[@text='Yes']")).click();
```

@After script section:

- Instruct the Appium® Driver to quit the execution
```
driver.quit();
```

4.2.2 Writing a test procedure for Android app (Example-1)

Here below find please the complete script with all sections combined:

```
@Before
public void setUp() throws MalformedURLException {
   dc.setCapability("testName", "Quick Start Android Native Demo");
   dc.setCapability("deviceName",”<device_name>”);
   dc.setCapability("udid","<device_id>"); //DeviceId from "adb devices" command
   dc.setCapability("platformName", MobilePlatform.Android)
   dc.setCapability("platformVersion", "7.0");
   dc.setCapability("skipUnlock","true");
   dc.setCapability("noReset","false");
   dc.setCapability(MobileCapabilityType.APP, <absolute_path_to_apk_file>);
   dc.setCapability(AndroidMobileCapabilityType.APP_PACKAGE, "com.appium.AppiumBank");
   dc.setCapability(AndroidMobileCapabilityType.APP_ACTIVITY, ".LoginActivity");
   driver = new AndroidDriver<>(new URL("https://0.0.0.0:4723/wd/hub"), dc);
   driver.manage().timeouts().implicitlyWait(15, TimeUnit.SECONDS);
}
```

```
@Test
public void quickStartAndroidNativeDemo() {
   driver.rotate(ScreenOrientation.PORTRAIT);
   driver.findElement(By.xpath("//*[@id='usernameTextField']")).sendKeys("JohnB");
   driver.hideKeyboard();
   driver.findElement(By.xpath("//*[@id='passwordTextField']")).sendKeys("testAppiumBank ");
   driver.findElement(By.xpath("//*[@id='loginButton']")).click();
   driver.findElement(By.xpath("//*[@id='makePaymentButton']")).click();
   driver.findElement(By.xpath("//*[@id='phoneTextField']")).sendKeys("0541234567");
   driver.findElement(By.xpath("//*[@id='nameTextField']")).sendKeys("Jon Baker");
   driver.findElement(By.xpath("//*[@id='amountTextField']")).sendKeys("50");
   driver.findElement(By.xpath("//*[@id='countryButton']")).click();
   driver.findElement(By.xpath("//*[@text='Switzerland']")).click();
   driver.findElement(By.xpath("//*[@id='sendPaymentButton']")).click();
   driver.findElement(By.xpath("//*[@text='Yes']")).click();
}
```

```
@After
public void tearDown() {
    driver.quit();
}
}
```

4.2.3 Writing a test procedure for iOS app (Example-2)

The example below implements the same scenario described for the Android environment (Example-1 above) but adapted for an iOS platform.

The script is like the previous one, only few capabilities are changed, as they are platform specific (e.g., “autoAcceptAlerts” or “bundled”).

```
import io.appium.java_client.ios.iOSDriver;
import io.appium.java_client.ios.iOSElement;
import io.appium.java_client.remote.iOSMobileCapabilityType;
import io.appium.java_client.remote.MobileCapabilityType;
import org.junit.*;
import org.openqa.selenium.By;
import org.openqa.selenium.ScreenOrientation;
import org.openqa.selenium.remote.DesiredCapabilities;

import java.net.MalformedURLException;
import java.net.URL;

public class LocaliOSTest {

  protected iOSDriver<iOSElement> driver = null;
  DesiredCapabilities dc = new DesiredCapabilities();

  @Before
  public void setUp() throws MalformedURLException {
    dc.setCapability("testName", "Quick Start iOS Native Demo");
    dc.setCapability("deviceName", "<device_name");
    dc.setCapability("udid", "<device_id>");
    dc.setCapability("platformName", MobilePlatform.iOS);
    dc.setCapability("autoAcceptAlerts","false");
    dc.setCapability("platformVersion", "10.2");
    dc.setCapability("skipUnlock","true");
    dc.setCapability("noReset","false");
    dc.setCapability(MobileCapabilityType.APP, < absolute_path_to_ipa_file >);
    dc.setCapability(iOSMobileCapabilityType.BUNDLE_ID, "com.appium.AppiumBank");

    driver = new iOSDriver<>(new URL(("https://0.0.0.0:4723/wd/hub"), dc);
   driver.manage().timeouts().implicitlyWait(15, TimeUnit.SECONDS)
  }

  @Test
  public void quickStartiOSNativeDemo() {
    driver.rotate(ScreenOrientation.PORTRAIT);
    driver.findElement(By.xpath("//*[@id='usernameTextField']")).sendKeys("JohnB");
    driver.hideKeyboard();
    driver.findElement(By.xpath("//*[@id='passwordTextField']")).sendKeys("testAppiumBank ");
    driver.findElement(By.xpath("//*[@id='loginButton']")).click();
    driver.findElement(By.xpath("//*[@id='makePaymentButton']")).click();
    driver.findElement(By.xpath("//*[@id='phoneTextField']")).sendKeys("0541234567");
    driver.findElement(By.xpath("//*[@id='nameTextField']")).sendKeys("Jon Baker");
    driver.findElement(By.xpath("//*[@id='amountTextField']")).sendKeys("50");
    driver.findElement(By.xpath("//*[@id='countryButton']")).click();
    driver.findElement(By.xpath("//*[@text='Switzerland']")).click();
    driver.findElement(By.xpath("//*[@id='sendPaymentButton']")).click();
    driver.findElement(By.xpath("//*[@text='Yes']")).click();
  }

  @After
  public void tearDown() {
      driver.quit();
  }
```

4.2.4 How to create a test script - step by step : Example 3

In this section we will go through the main steps to create an automated script.

All supporting tools are already installed and ready to use.

The next screenshot is shown as an example, in fact they can differ between different OS and version of Appium®.

The next example has been recorded on a macOS.

The next script has the goal to check the behavior of an editor APK, such as

- Start application
- Create a new text file
- Enter some text
- Save it
- Close it
- Check the existence of created text document

4.2.4.1 Start the Appium® application

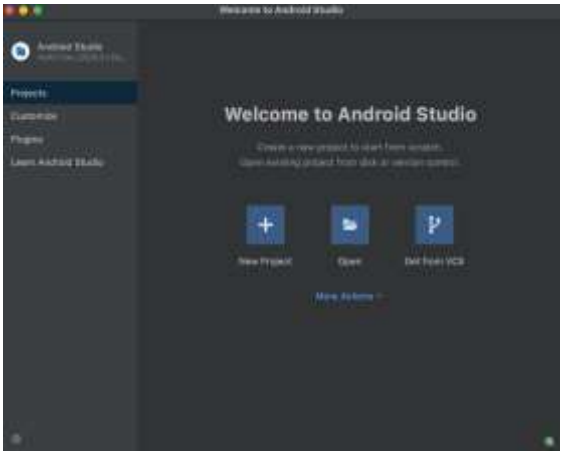
Start the Appium® application and click the “Start Server” button.

The next picture will be shown once the Server

- has been started
- ready to receive commands from the Client API language
- send them to the target device over HTTP

4.2.4.2 Start Android Studio.

The SDK and at least a device (emulator or real) should be ready to use.



4.2.4.3 Start the device using the “green run” action button.

The selected and started device should now be ready to receive commands from the Appium® server.

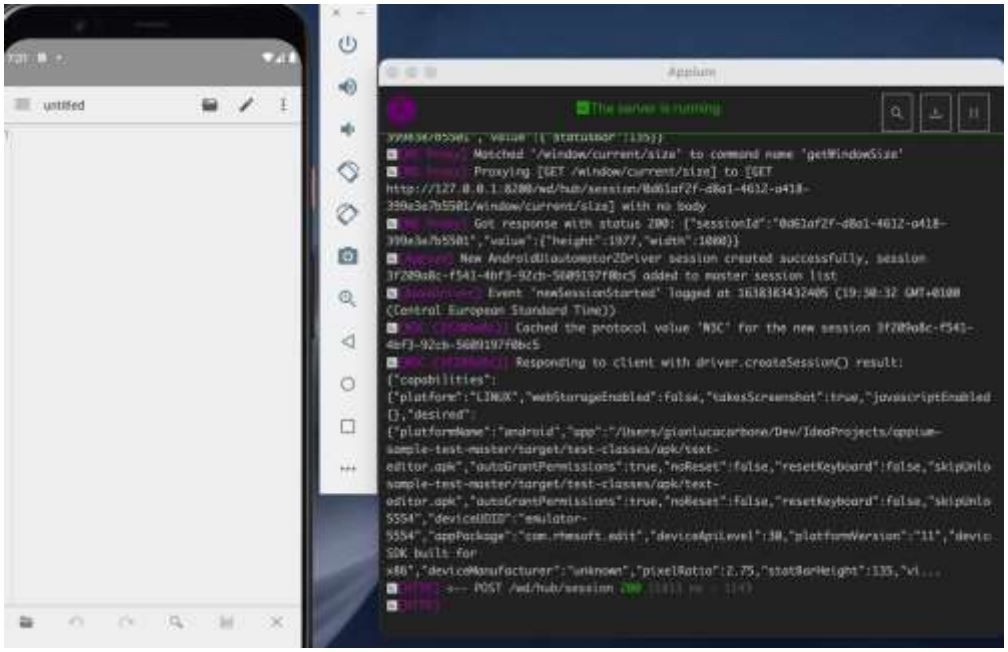
(For testing purpose, we have used a text editor APK and implemented a general scenario to edit, save and verify the existence of a text file.)

We assume we do not have all info about "actions" implemented in the APK, so we need to find them.

- Identify elements on the APK
  Here after is explained how to identify elements on a mobile app using the Appium® client and trigger actions or get status on them. We are going to describe the Java implementation of the methods below:

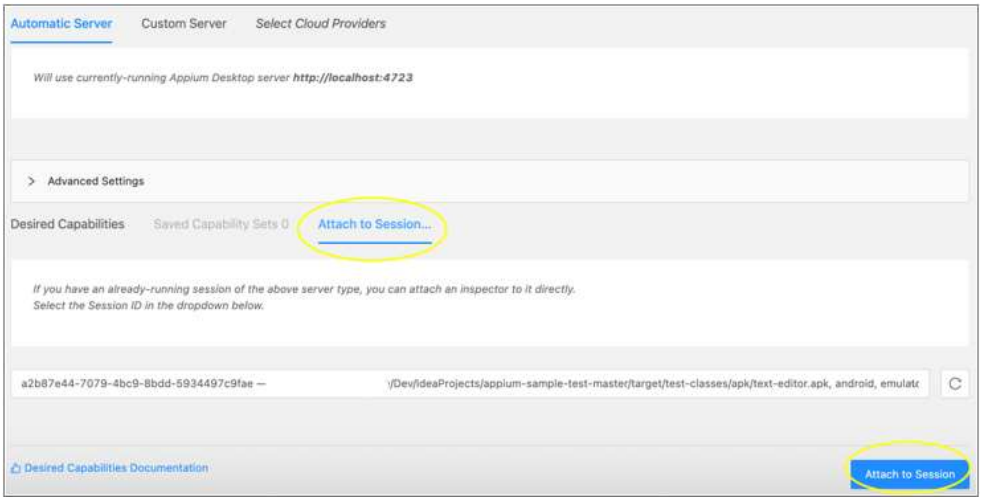
```
writeContentInFile();
clickSaveButtonAndEditFileName();
goToInternalStorageAndVerifyFileIsPresent();
```

- Implementation and execution of script
  - Using a debug mode, the developer can set a break point soon after the application has started and then attach the session to the Appium® server as described below.
  - Set a break point before the first test method is executed and after the Appium® capabilities are loaded
  - Start the execution of APK on Appium® side
  - The emulator device should show the view below.

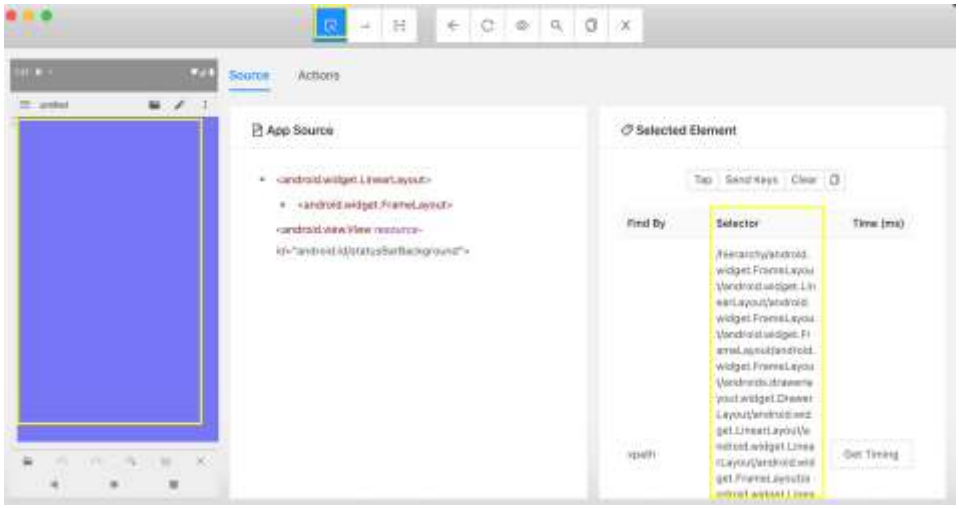


- Click the magnifying glass highlighted in the screenshot below.

- Once the Appium® window below is open, the user can set his own Desired Capabilities in a JSON format or go to "Attach to Session" and leave the Appium® server to automatically get capabilities and attach the session to the available device. See next screenshot



- The current view in the device is represented in the Appium® window below and from here we can identify app elements and interact with them.



- On the left side we find the current view of the app under test that got captured by the Appium® server and on the right side are listed the ways to identify that area (in the forms of mobile element) in the test scripts.
- In this case the highlighted Xpath can be used to identify the area where to write content in the text app and this will be used in the test script to send the commands which will serve to send the text to write into the file.

Now that we know how to navigate to the mobile elements and get the identifier, we can implement the methods for our first method and following the others.

```
writeContentInFile
```

In this method the text file area where to write our content is identified and a pre-defined text is written.

```
driver.findElement(By.xpath(TEXT_FIELD)).click();
Actions action = new Actions(driver);
action.sendKeys(MY_TEXT).perform();
```

*TEXT_FIELD is the **xpath** and has the value below which we retrieved in the way described above*

"/hierarchy/android.widget.FrameLayout/android.widget.LinearLayout/android.widget.FrameLayout/android.widget.FrameLayout/android.widget.FrameLayout/androidx.drawerlayout.widget.DrawerLayout/android.widget.LinearLayout";

Create an Actions object which will take care to send the text MY_TEXT. It contains the text to be written into the file

```
clickSaveButtonAndEditFileName
```

Identify the “Save button” and click on it. User is redirected in the edit file name section. Here the file name is edited and the user confirms to save it.

```
driver.findElementsByAccessibilityId(SAVE_BUTTON).get(0).click();
wait.until(ExpectedConditions.visibilityOfElementLocated(By.xpath(EDIT_FILE_NAME))).clear();
wait.until(ExpectedConditions.visibilityOfElementLocated(By.xpath(EDIT_FILE_NAME))).sendKeys(fileNme + FILE_EXTENSION);
wait.until(ExpectedConditions.visibilityOfElementLocated(By.id(CONFIRM_FILE_NAME))).click();
wait.until(ExpectedConditions.visibilityOfElementLocated(By.id(DONE_FILE_NAME))).click();
```

This method implements a Selenium explicit “Wait” on each step, in order to be sure the target element is available and can be interacted with.

The identification of the elements *SAVE_BUTTON, EDIT_FILE_NAME, CONFIRM_FILE_NAME and DONE_FILE_NAME* is done in the same way as in the previous method, hence we skip further explanations.

```
goToInternalStorageAndVerifyFileIsPresent
```

User goes to internal storage and verifies that the new created file is present.

```
driver.findElementsByAccessibilityId(OPEN_BURGER_BUTTON).get(0).click();
wait.until(ExpectedConditions.visibilityOfElementLocated(By.xpath(GO_TO_STORAGE_MANAGER))).click();
wait.until(ExpectedConditions.visibilityOfElementLocated(By.xpath(GO_TO_INTERNAL_STORAGE))).click();
```

```
getFilesFromInternalStorage();
Assert.assertTrue(INTERNAL_STORAGE_CONTENT.contains(fileName + FILE_EXTENSION), "Verify Internal Storage contains" +
fileName + FILE_EXTENSION);
```

As in the previous methods, here we also use explicit waits.

To verify that file is present, we search in the device internal storage, get all files and verify its presence.

NOTE: For the complete source please refer to “Appendix D – *Source Example Editor-APK*”.

4.3 Executing a test procedure on real device

As discussed in the previous chapter a test can be executed indifferently on an emulator or real device.

To work on a real device a few setup things need to be done, apart from that all others is managed by Appium® Agent (check also the picture in chapter **2.4**).

- Connect your mobile to the computer via USB
- Enable USB-Debugging on Mobile (in Developer option)
- In Cmd line on your windows computer, enter the command “**Adb devices**”

The next screenshot indicated the most possible answers

- No device connected to computer (or no USB-Debugging enabled)
  List of device(s) is empty
- Device is connected, USB-Debugging is enabled but non confirmed
  The device is recognized, but not authorized to be contacted
- Device connected and USB-Debugging is approved
  Device connected and USB-Debugging is approved too

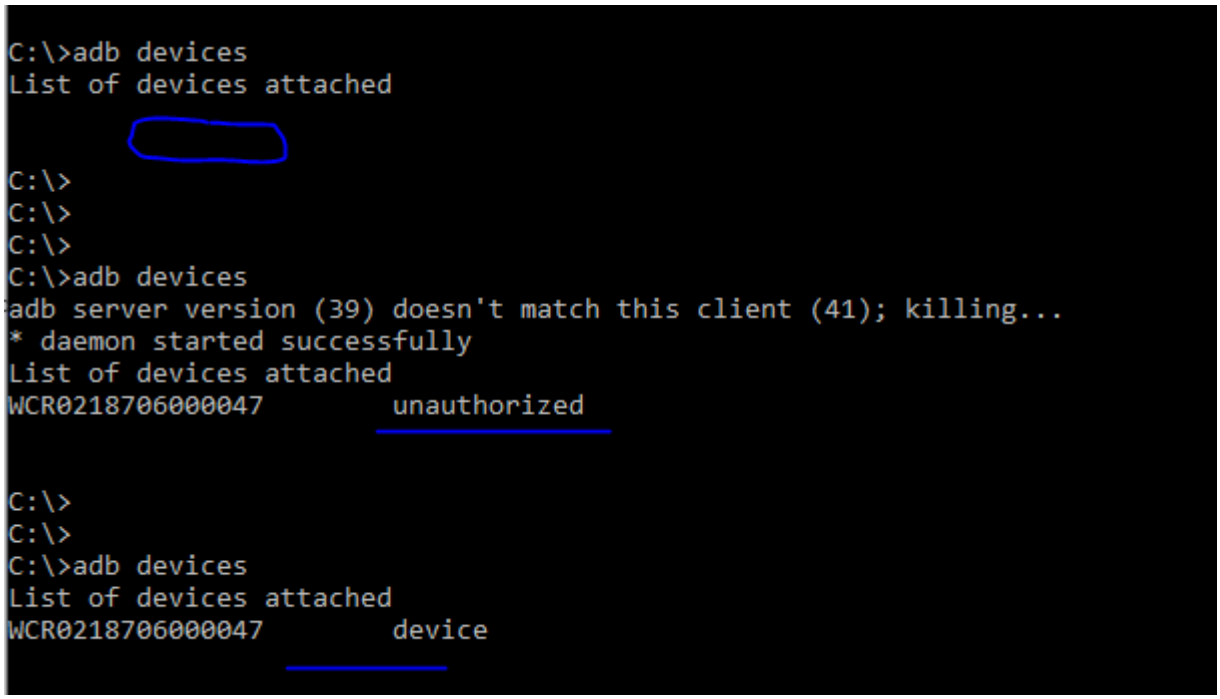


Figure 5 : Check for real device connection (adb command)

Start the test script in same manner as you have executed using an emulator. The application to be tested, will be downloaded on mobile device and the test execution will start.

4.4 Sync (Wait Mechanisms)

The handling of the interaction and communication between the client and the devices (physical or emulator), is the core part of Appium® framework. However, the one of the most critical part of an automated script is the time to complete a tasks **or** the capability to take the control after a specified action is performed **or** a timeout is elapsed.

1. The application (APK) reacts to a command in a short time
2. The application (APK) does not react (within a given time or not at all)
3. The application (Script) takes the control back after a successfully completed command or a timeout

To make the point 3 happening, Appium® same as in Selenium uses two main strategies of wait, implicit and explicit. In addition, the Java Thread sleep recommended to use in very rare cases, and we will see below the reason:

- **Implicit Wait**

  An Implicit Wait informs the Appium® web driver to wait for specific amount of time before returning back to the script with an exception as e.g., "NoSuchElementFound". It is set before the script begins and is valid for all elements the script interacts with during the full execution unless a different wait (e.g., explicit) is set for a specific element.

  An implicit wait can be set to wait for different units of time, such as hours, minutes, seconds, etc. The method accepts 2 parameters, the TimeUnit and a numeric value. In the example below, we instruct to wait for 45 Sec. before getting back a result:

```
driver.manage().timeouts().implicitlyWait (45, TimeUnit.SECONDS);
```

Example of implicit wait after loading an Apk

```
// capabilities has been loaded before
driver = new RemoteWebDriver (new URL ("http://127.0.0.1:4723/wd/hub"), capabilities);
driver.manage().timeouts().implicitlyWait(7, TimeUnit.SECONDS);
```

Example of implicit wait to find an element “Start”

```
driver.findElement(By.name("Start")).click();
driver.manage().timeouts().implicitlyWait(12,TimeUnit.SECONDS);
```

- **Explicit Wait**

  The explicit wait is a dynamically “set wait command” for a specified condition to be completed. Instruct the WebDriver by using Appium® Explicit wait.

  The next will be “a Wait until” the condition happens, i.e., the 10 seconds is elapsed.

```
...
WebDriver wait = new WebDriverWait (driver, 10);
wait.until (ExpectedConditions.visibilityOfElementLocated(By.id("An_Element")));
```

Another way to wait is given by Thread.sleep

- Thread.sleep

  The thread sleep, basically tells the java process to wait statically for a certain given amount of time. It will always wait for the time even when it would not be necessary (for example the element is already available, or the page as already shown a failure).

```
...
Thread.sleep (10)
 ...
```

  Note: It is highly recommended to avoid the **Thread.sleep** wait, which could exponentially slow down the entire test execution.

4.5 Logging Mechanism

The native programming languages like Java offer a variety of log methods. Do exist several Logging frameworks and/or libraries options in the market that can be used to expand the logging features offered with more organized logging templates and capabilities.

Depending on language and methods, the log methods can store text or performance information, such the time spent e.g., to reload a page.

As described in the previous chapters, it will be a good rule to log information in different way, at beginning to check the flow of scrips itself, would be necessary to store all executed steps.

Once the test scripts have been checked and they are stable, then the reported information can be reduced to less relevant info. It is a good practice to implement through a flag the possibility to enable or disable on demand the use of logged information.

Logged details should be used for reporting purpose of the executed flow and to help the test automation developer to track the flow and correct in a short time any discrepancy.

Not to forget, Logs should not record any critical or personal information as per security definition.

So, all logs of **sensitive** data MUST be deactivated or disabled once the test will be completed.

4.6 Maintainability of test scripts

The automated scripts should be created in a way that can be easily understood by any stakeholder. For technical stakeholders, code must be easily readable and self-documented while for non-technical stakeholders, there are additional libraries that enable to write tests in a business language that make it easy to follow and comprehend.

In general, what defines a high-quality test is first of all, the ease to identify a problem in the System under test (SUT), when the test fails. The intent of the test must be simple to read from the code. Finally, the test runs fast and is reliable.

A code has a good maintainability when changes are not frequently needed and are easy to apply. In addition, when a change is a made to the SUT and it does not affect the use case scenario it should not break the test scenario. When a change to the SUT breaks a test, it has the minimum possible butterfly effect on test suites.

To minimize the impact of a change, use stable control locator strategy with accessibility id (for iOS the accessibility identifier, for Android the content description and for Windows the AutomationId).

XPath, if used as per best practices, can be very robust and less susceptible to application changes. However, when IDs or accessibility IDs are available in the app, the first choice for the locator should be on those, for the simple reason that they will allow to pinpoint the exact element in the app we care about, even if the UI changes appearance.

XPath is the most expressive and commonly accepted locator strategy, but automation engineers have long warned of its drawbacks, and the low performance is the first reason which should be considered.

Time and expertise to find the right or optimal XPath has its cost as, unfortunately, very often is preferred the first found XPath solution which accomplishes the goals over the best solution.

The presence of IDs and Accessibility IDs is a luxury which automation engineers cannot always expect,

Hence working with XPath and mastering its best practice is still a must and its usage is still a well-accepted industry standard.

In addition, we should minimize repetition of code, minimize the things tested by one test case and create abstraction between the test and the SUT.

Below we are introducing several patterns and practices for good script maintainability:

- Don’t Repeat Yourself (DRY)
- Single Responsibility Principle (SRP)
- Page Object Pattern (POP)
- Descriptive and Meaningful Phrases (DAMP)
- The SOLID Principle

4.6.1 Don’t Repeat Yourself (DRY)

The **DRY** is simply an approach, or we can say a different perceptive to programmers. DRY stands for Don’t Repeat Yourself. In Java, it means don’t write the same code repeatedly. Suppose you are having the same code at many places in your program, then It is known as DRY, You are repeating the same code repeatedly at different places. Hence, the solution is obtained using the DRY concept by placing the methods in place of all repeated codes and define the code in one method. So, by calling methods, we will reach the principle DRY.

4.6.2 Single Responsibility Principle (SRP)

The **Single Responsibility Principle** as the name suggests, this principle states that each class should have one responsibility, one single purpose. This means that a class will do only one job, which leads us to conclude it should have only one reason to change.

We don't want objects that know too much and have unrelated behaviour. These classes are harder to maintain. For example, if we have a class that we change a lot, and for different reasons, then this class should be broken down into more classes, each handling a single concern. Surely, if an error occurs, it will be easier to find.

4.6.3 Page Object Pattern (POP)

A combination of both DRY and SRP, is represented by the **Page Object Pattern.** Page Object is a Design Pattern which has become popular in test automation for enhancing test maintenance and reducing code duplication. A page object is an object-oriented class that serves as an interface to a page of your SUT. The tests then use the methods of this page object class whenever they need to interact with the UI of that page. The benefit is that if the UI changes for the page, the tests themselves don't need to change, only the code within the page object needs to change. Subsequently all changes to support that new UI are in one place.

The Page Object Design Pattern provides the following advantages:

- There is a clean separation between test code and page specific code such as locators (or their use if you're using a UI Map) and layout.
- There is a single repository for the services or operations offered by the page rather than having these services scattered throughout the tests.

In both cases this allows any modifications required due to UI changes to all be made in one place.

4.6.4 Descriptive and Meaningful Phrases (DAMP)

DAMP promotes the readability of the code. To maintain code, you first need to understand the code. To understand it, you must read it. Consider for a moment how much time you spend reading code. It's a lot. DAMP increases maintainability by reducing the time necessary to read and understand the code.

4.6.5 SOLID principle

Follow the SOLID Principle (cfr. Appendix C – Sources [13])

The SOLID principles are guidelines that can help you create maintainable and extendable classes and systems. It can be shortly recalled as

a. **S**: Single responsibility → Every class is responsible for exactly one thing, changing anything in a class will change only one thing

b. **O**: Open-closed → The classes should be open for extension but closed for modification. This can be achieved with inheritance.

c. **L**: Liskov substitution → Derived classes must be substitutable for their base classes.

d. **I**: interface segregation → Make fine grained interfaces that are client specific, Classes should be as much as possible specialized

e. **D**: Dependency inversion → Depend on abstractions, not on concretions, i.e., class(es) should not depend on concrete details defined in other classes

5 List of system Capabilities

Timing	1*K1, 3*K2 = 10+75 = 85 minutes
Terms	Capabilities

Topics & Learning Objectives for this Chapter:

Learning Objectives

ATF-5.1 (K1) Server Capabilities
ATF-5.2 (K2) Common Capabilities to all OS
ATF-5.3 (K2) Android specific Capabilities
ATF-5.4 (K2) iOS specific Capabilities

5.1 Server Capabilities

The next list shows the “Desired capabilities” used to communicate with the server.

A Desired capabilities is a JSON object sent by the client to the server, i.e., a command with parameters such as keys or values.

The client issues the command with the necessary parameter(s) and send the information (capabilities) to the server to be executed.

To use the capabilities the class “import org.openqa.selenium.remote.DesiredCapabilities” must be imported in a Java application.

The next chapters show the common capabilities for the environments and the specific one for Android and iOS.

By the capabilities (or setup commands), it is possible to change the standard behavior on the device. As an example, if the device works in portrait mode setting capability to landscape, will change the display mode for the given device from Portrait to Landscape, this could e.g., help to check if the displayed text is readable in both ways (portrait, landscape) and no text or UI graphic are hidden to the user.

The next tables will show the main capabilities ordered by

- Common Server Capabilities
- Android Capabilities
- iOS Capabilities

Note: the list below may not be complete

5.2 Common Capabilities to all OS

ID	Capability	Definition / Examples
01	App	**Definition:** Absolute local path or remote HTTP URL of the .ipa,.apk, or .zip file. **Description**: Appium® will install the app binary on the appropriate device. **Example**: caps.setCapability ("app","/apps/demo/MyExample.apk" caps.setCapability ("app","http://app.com/app.ipa"); or **Note**: In case of Android OS, specifying the appPackage and appActivity capabilities (see table capabilities for Android below), then this capability is not required:

02	automationName	**Definition:** It defines the automation engine. If Android SDK version is lower than 17, then the value Selendroid has to be entered, otherwise, the capability takes the default value
		
		Example:
		DesiredCapabilities caps = new DesiredCapabilities(); // object is created
		caps.setCapability ("automationName", "**Selendroid**"); // to set capability value for versions // lower than 17
		Note: On iOS the capability is not necessary

03	browserName	**Definition:** Define the browser
		Example:
		Android→ caps.setCapability("browserName", "**chrome**"); // browser Chrome
		caps.setCapability(MobileCapabilityType.BROWSER_NAME, "Chrome");
		
		iOS → caps.setCapability("browserName", "**safari**"); // Browser Safari
		caps.setCapability(MobileCapabilityType.BROWSER_NAME, "safari");

04	deviceName	**Definition** Define the type of mobile device or emulator to be used
		Example:
		caps.setCapability("deviceName", "Galaxy-**1**");
		caps.setCapability(MobileCapabilityType.DEVICE_NAME,"Nexus-1");

05	newCommandTime , newCommandTimeout	**Definition**: Wait for the given time to end a session
		Description: Appium® will wait for the defined time in seconds for a command from the client before the client is assumed to been quit. By Default Appium® defines a timeout of 60 seconds.
		Example:
		caps.setCapability ("newCommandTimeout", "**15**"); // Set timeout to 15 secs.
		Caps.setCapability (MobileCapabilityType.NEW_COMMAND_TIMEOUT, "15");

06	platformName	**Definition:** Set the mobile OS platform.
		Description: Values as iOS, Android are supported
		Example:
		caps.setCapability ("platformName","Android"); // Platform is Android
		caps.setCapability (MobileCapabilityType.PLATFORM_NAME, "Android");

07	platformVersion	**Definition: S**et the mobile OS version
		Description: Mobile OS can be entered
		Example:
		caps.setCapability("platformVersion","**6.4**"); // Version 6.4
		caps.setCapability(MobileCapabilityType.PLATFORM_VERSION, "6.4")

08	autoLaunch	**Definition: I**nstall and launch the app automatically. **Description**: Enable the automatic installation of application (default value is **true**). To disable the automatic installation and lauch for the application the value **false** has to be entered **Example**: caps.setCapability("autoLaunch","**false**"); // Disable the automatic installation and launch for app.
09	Language	**Definition**: Set the language on the simulator/emulator. **Example**: caps.setCapability("language","**It**"); // It works only on the simulator/emulator

10	Udid	**Definition:** A unique device identifier to identify iOS physical device. **Description**: String up to 40 characters long (e.g., 1234544356…). This capability is used when automating apps on iOS physical device. UDID can retrieved iTunes, by looking on Serial Number: **Example**: caps.setCapability("udid", "**01234567890123456789012345678901234567890123456789**");
11	orientation	**Definition**: Start the mobile app on a given orientation on emulator **Example**: caps.setCapability("orientation", "**PORTRAIT**"); // or Landscape **Note**: It works on simulator/ emulator only
12	noReset	**Definition:** Reset the application's state before starting the session **Example**: Default value is **False**, i.e. no reset performed caps.setCapability("noReset"-," **true**"); // request the Application to perform a reset
13	fullReset	**Definition**: reset the application **d. On Android** , it resets the app's state by uninstalling the app instead of clearing the app data, i.e. will remove the app after the session is complete. **e. On iOS** delete the entire simulator folder Defaul value is **False** for both OS **Example**: caps.setCapability("fullReset", "**true**");

5.3 Android specific Capabilities

ID	Capability	Definition / Examples
01	appPackage	**Definition**: This capability is for the Java package of the app to run on Android **Example**: location.android.calculator_2 caps.setCapability("appPackage", "location.android.calculator_2");
02	appActivity	**Definition**: Specify the Android activity to launch from a given package (see appPackage capability) **Example**: to specifiy the activity Calculator from package calculator_2 caps.setCapability("appActivity", "location.android.calculator_2.Calculator");
03	appWaitActivity	**Definition**: Android activity for which the user wants to wait **Example**: caps.setCapability("appWaitActivity","com.android.calculator2.Calculator");
04	appWaitPackage	**Definition**: Package of the Android app to wait **Example**: given the package on the location: location.example.android.myTestApp caps.setCapability ("appWaitPackage","location.example.android.myTestApp");
05	deviceReadyTimeout	**Definition**: Set the timeout (in seconds) while waiting for the device to be ready **Example**: caps.setCapability("deviceReadyTimeout","10");
06	enablePerformanceLogging	**Definition**:Enable the Chrome driver's performance logging by the use of this capability. **Description**: It will enable logging only for Chrome and web view; the default value is **false**: **Example**: caps.setCapability("enablePerformanceLogging", "true");
07	androidDeviceReadyTimeout	**Definition**:To set the timeout in seconds for a device to become ready after booting **Example**: caps.setCapability ("androidDeviceReadyTimeout", "20"); // Timeout set to 20 seconds
08	androidDeviceSocket	**Definition**: Set DevTool socket name. **Description**: Capability used to set DevTools socket name. Needed when an app is a Chromium-embedding browser. The socket is opened by the browser and the ChromeDriver connects to it as a DevTools Client (e.g., chrome_DevTools_remote) **Example**: caps.setCapability ("androidDeviceSocket", "chrome_DevTools_remote");
09	Avd	**Definition**: Name of avd to launch **Description**: Using this capability, the name of avd to launch canbe defined **Example**: caps.setCapability ("avd", "AVD_NEXUS_5");
10	avdLaunchTimeout	**Definition**:Time to wait until the avd will be lauched **Description**: This capability helps to define how long to wait for an **avd** to launch and connect to the Android Debug Bridge (ADB). Value is defined in milliseconds; default value is 120000 ms **Example**: caps.setCapability ("avdLaunchTimeout", "240000");
11	avdReadyTimeout	**Definition**: Time to wait until the avd finish its boot animation **Description**: You can specify the wait time (in milliseconds) for an **avd** to finish its boot animations Value is defined in milliseconds; default value is 120000 ms **Example**: caps.setCapability ("avdReadyTimeout", "240000");
12	avdArgs	**Definition**: To pass the additional emulator arguments when launching an avd **Example**: to pass the parameter "netfast" enter caps.setCapability ("avdArgs","**netfast**");
13	chromedriverExecutable	**Definition**: Give the absolute local path to the WebDriver executable (if the Chromium embedder provides its own WebDriver, it should be used instead of the original ChromeDriver bundled with Appium®) **Example**: caps.setCapability("chromedriverExecutable","/abs/path/to/webdriver");

14	autoWebviewTimeout	**Definition**: The following capability allows to set the time (in milliseconds) for which you need to wait for the Webview context to become active; the default value is 2000ms: To change the time to 3 secs (3000ms) enter **Example**: caps.setCapability ("autoWebviewTimeout", "3000");
15	intentAction	**Definition**: the capability "Intent action" is used to start an activity, as shown in the example below. The default value is android.intent.action.MAIN. For example, to start android.intent.action.MAIN or android.intent.action.VIEW enter **Example**: caps.setCapability("intentAction", "android.intent.action.VIEW");
16	intentCategory	**Definition**: The capability "intent category" is used to start the activity **Default value**: android.intent.category.LAUNCHER, android.intent.category.APP_CONTACTS: **Example**: caps.setCapability ("intentCategory", "android.intent.category.APP_CONTACTS");
17	intentFlags	**Definition**: Flags are used to start an activity (the default is 0x10200000), for example, 0x10200000: **Example**: caps.setCapability ("intentFlags", "0x10200000");
18	unicodeKeyboard	**Definition**: Enable Unicode input by using the following code the default value is **false**: **Example**: caps.setCapability ("unicodeKeyboard","**true**");
19	resetKeyboard	**Definition**:Set the keyboard to its original state The default value is **false**, No reset will be perfomed **Example**: caps.setCapability ("resetKeyboard","**true**");

5.4 iOS specific Capabilities

ID	Capability (iOS)	Explanation
01	calendarFormat	**Definition**: Set the calendar format for the iOS simulator. It applies only to a simulator **Example**: caps.setCapability("calendarFormat"," Gregorian"); \\ format to Gregorian
02	bundled	**Definition**: Start an app on a real device **Description**: BundleId is used to start an app on a real device or to use other apps that require the bundled during the test startup, for example, io.appium.TestApp: **Example**: caps.setCapability("bundleId"," io.appium.TestApp");
03	launchTimeout	**Definition**: Specify the amount of time needed to wait for Instruments **Description**: Specify the amount of time to wait for Instruments before assuming that it hung and the session can be considered failed. The Value is in milliseconds **Example**: caps.setCapability("launchTimeout","30000");
04	locationServicesEnabled	**Definition**: Enable location services. Applicable to simulator only **Description**: This capability is used to enable location services. **Example**: caps.setCapability("locationServicesEnabled","**false**");
05	locationServicesAuthorized	**Definition**: Location services alert does (not) pop up. This capability works together with the capability "bundled" **Description**: This capability can be used together with bundled by using the bundled capability. This capability applies only on simulator. After setting this, the location services alert doesn't pop up. The default value is **false** **Example**: caps.setCapability("locationServicesAuthorized","**true**");
06	autoAcceptAlerts	**Definition**: Accept privacy permission alerts automatically **Description**: Accept privacy permission alerts automatically, such as contacts, photos, location, once they would arise By default, the automatic alert is deactivated **Example**: caps.setCapability("autoAcceptAlerts","true");
07	nativeInstrumentsLib	**Definition**:You can use the native instruments library by setting up this capability: **Description**: **Example**: caps.setCapability("nativeInstrumentsLib","true");
08	nativeWebTap	**Definition**: Enable/disable real web taps in Safari, which are non-JavaScript based. **Description**: Used to enable/ disable real web taps in Safari, which are non-JavaScript based. The default is **false** **Example**: caps.setCapability("nativeWebTap","**true**");
09	safariAllowPopups	**Definition**: It allows JavaScript to open new windows in Safari. The default is the current simulator setting. **Description**: You can use this capability on a simulator only. **Example**: caps.setCapability("safariAllowPopups","**false**");
10	safariIgnoreFraudWarning	**Definition**: Prohibits Safari from displaying a fraudulent website warning. **Description**: This capability can be used only on a simulator **Example**: caps.setCapability("safariIgnoreFraudWarning","**false**");
11	safariOpenLinksInBackground	**Definition**:This capability instructs / enables Safari to open links in new windows. **Description**: the default keeps the current simulator settings: **Example**: caps.setCapability("safariOpenLinksInBackground","true");
12	interKeyDelay	**Definition**: Delay the keystrokes sent to an element when typing **Description**: Delay the keystrokes sent to an element when typing the key. The paramenter (time) is in milliseconds **Example**: caps.setCapability("interKeyDelay","**100**");

6 Appium® 2.0

Timing	3*K2 = 75 minutes
Terms	N/A

Topics & Learning Objectives for this Chapter:

Learning Objectives

ATF-6.1 (K2) Overview on Appium® 2.0
ATF-6.2 (K2) Starting Appium® 2.0: what has changed
ATF-6.3 (K2) Starting Appium® 2.0: An example

6.1 Overview on Appium® 2.0

The Appium® Consortium released first version almost 7 years ago (cfr. Chapter 2.1). Since then, Appium® has rolled out a lot of new features and automation backend architecture has also changed a lot.

At the time of writing this document Appium® 2.0 is ready to use but still in its beta version.

New / Improved features and some of the core goals for this major revision of Appium® 2.0 are shown below:

- **Decouple the drivers**

  Appium®'s platform drivers (the XCUITest driver, UiAutomator2 driver, Espresso driver, etc...) have very little in common with one another. They really ought to be developed as independent projects that implement the same interface and can be used equivalently with the Appium® server. Appium® 2.0, the code for these drivers will no longer be bundled with the main Appium® server. This drastically decreases the size of an Appium® install. It also supports to install only the drivers you need to use. It also makes it possible to freely update drivers independently of Appium® and of one another, so that you can get the latest changes for one driver while sticking with a known stable version of another driver. In short this is a very powerful optimization.

- **Create a driver ecosystem**

  By using any existing Appium® drivers as a template, anyone can create their own custom drivers with a minimum of extra code. All of these custom drivers can then be installed by any Appium® user.

- **Create a plugin ecosystem**

  In addition to drivers, it's become clear that there are a huge variety of use cases for Appium®, which involve the use of special commands or special ways of altering the Appium®'s behavior for specific commands. Some examples are e.g., Find Element by Image API or the Appium + Test AI Classifier.

  Appium® plugins will help with various use cases that require a change to Appium®'s default behavior. These changes provide users an opportunity to build new plugins in the future.

  Plugins have been designed to add arbitrary functionality before or after actual commands. Also, the plugin can modify the Appium® Server itself to introduce new commands and distribute them. For example, the current "image" plugin from Appium® helps find an element by an image and helps in image comparison. It is possible to install those feature as independent plugins and anyone in the world will be able to easily create Appium® plugins to implement new commands or alter the behavior of existing commands.

- **Miscellaneous standardization**

  There have been a lot of deferred work on Appium® that kept getting pushed off because it could introduce a breaking change into the API. In Appium® 2.0 is taken the opportunity to make those changes and keep bringing Appium® into the future.

6.2 Starting Appium® 2.0: what has changed

Appium® team has released Version 2.0.0-beta end of 2021, with the official Appium® image plugin and some minor defect fixes and improvements.

The next part will indicate in some details the main changes

- Install Appium® 2.0 beta version can be installed using the below command
  npm install –g appium@next
- *-ah : to* specify a directory to install Appium® drivers
  Appium server -ah /path/to/install/drivers driver install uiautomator2. No default path.
- -ka *or* –keep-alive-timeout *to specify the number of seconds the Appium® server should apply as both the keep-alive timeout and the connection timeout for all requests. Defaults to 600 seconds*. ***Appium server -ka 800***

Appium® drivers (UIAutomator2 driver, XCUITest driver, Espresso Driver, etc) are tightly coupled with Appium® Server but from Appium® 2.0, these drivers are separated from Appium® Server and can be installed separately based on users' needs.

Creating custom drivers for any new platform or special use cases are going to be easy from Appium® 2.0.

- Command to list all available drivers in Appium® 2.0
  appium driver list, an output similar to the next below, should be displayed
  Listing of available drivers
  - uiautomator2 [not installed]
  - xcuitest [not installed]
  - youiengine [not installed]
  - windows [not installed]
  - mac [not installed]
  - mac2 [not installed]
  - espresso [not installed]
  - tizen [not installed]
  - flutter [not installed]
  - safari [not installed]
  - gecko [not installed]
- Command to install a driver (e.g., uiautomator2 driver
  appium driver install uiautomator2, An output similar to the next below, should be displayed
  Attempting to find and install driver 'uiautomator2'
  Installing 'uiautomator2' using NPM install spec
  'appium-uiautomator2-driver'
  Driver uiautomator2@1.61.2 successfully installed
  - automationName: UiAutomator2
  - platformNames: ["Android"]
- Command to all installed Appium® drivers
  appium driver list –installed, An output similar to the next below, should be displayed

Result:
Listing installed drivers
- uiautomator2@1.61.2 [installed (NPM)]
There are many other options offered by the Drivers CLI to install drivers from a specific source, update all the installed drivers, etc. This simplifies the overall installation process and users have the flexibility to choose to upgrade drivers without updating the server itself

- Command to list all available **Appium® Plugins**
  appium plugin list, an output similar to the next below, should be displayed
  Result
  Listing available plugins
  - images [not installed]
- Command to install an Appium® plugin
  appium plugin install "images", where "images" it the plugin to be installed
- Command to activate a plugin when starting the Appium® server
  appium server -ka 800 --plugins=images -pa /wd/hub, where "images" it the plugin to be installed
  Result
  [Appium] Welcome to Appium v2.0.0-beta.10
  [Appium] Non-default server args:
  [Appium] plugins: { 0: images}
  [Appium] basePath: /wd/hub
  [Appium] keepAliveTimeout: 800
  [Appium] tmpDir: /var/folders/f5/fwh8w_ms6q377gn_fb2bmjp40000gp/T
  [Appium] Appium REST http interface listener started on 0.0.0.0:4723
  [Appium] Available drivers:
  [Appium] - uiautomator2@1.61.2 (automationName 'UiAutomator2')
  [Appium] - xcuitest@3.36.0 (automationName 'XCUITest')
  [Appium] Available plugins:
  [Appium] - images@1.1.2 (ACTIVE)

No code changes are required in the client script by using the latest Appium® server.

The next list is a subset of all server arguments (both «new» and «deprecated»).

Appium® server can take-up several optional arguments. Several parameters are deprecated in Version 2.0, whilst some new have been added. Check "Appendix C – Sources [18]" for a complete list of server's arguments, default values and examples.

Flag	Default	Description
--shell	null	Enter REPL mode
--allow-cors	false	Turn on CORS compatibility mode, which will allow connections to the Appium® server from within websites hosted on any domain. Be careful when enabling this feature, since there is a potential security risk if you visit a website that uses a cross-domain request to initiate or introspect sessions on your running Appium® server.
--ipa	null	(iOS-only) abs path to compiled .ipa file
-a, --address	0.0.0.0	IP Address to listen on
-p, --port	4723	port to listen on
-pa, --base-path	/wd/hub	Initial path segment where the Appium®/WebDriver API will be hosted. Every endpoint will be behind this segment.

Flag	Default	Description
--reboot	false	- (Android-only) reboot emulator after each session and kill it at the end
--command-timeout	60	[DEPRECATED] No effect. This used to be the default command timeout for the server to use for all sessions (in seconds and should be less than 2147483). Use newCommandTimeout cap instead
--no-reset	false	[DEPRECATED] - Do not reset app state between sessions (iOS: do not delete app plist files; Android: do not uninstall app before new session)
--full-reset	false	[DEPRECATED] - (iOS) Delete the entire simulator folder. (Android) Reset app state by uninstalling app instead of clearing app data. On Android, this will also remove the app after the session is complete.
--app-pkg	null	[DEPRECATED] - (Android-only) Java package of the Android app you want to run (e.g., com.example.android.myApp)
--app-activity	null	[DEPRECATED] - (Android-only) Activity name for the Android activity you want to launch from your package (e.g., MainActivity)

6.3 Starting Appium® 2.0: An example

In this section we try to answer the question below:

What was changed to run successfully the automated script described in section 4.2.4 with Appium® 2.0?

- As prerequisite, Appium® 2.0 is installed on the machine with: ***npm install –g appium@next***
- At list one driver is installed: **appium driver install uiautomator2**
- In a terminal window we run the Appium® server with the command "**appium**"

All elements we need in the script have been already inspected previously, but the inspector for Appium® 2.0 is available at https://github.com/appium/appium-inspector/releases

The following changes were required in order to execute the same script with Appium® 2.0.

- Added capability "automationName" to tell Appium® which driver we want to use:
  caps.setCapability("automationName", "UiAutomator2");
- The URL of the Appium® server does not need the routing to "/wd/hub/" and it is now just http://0.0.0.0:4723:
  driver = new AndroidDriver<>(new URL("http://0.0.0.0:4723"), caps);

After applied the code changes, and given the Appium® server is running with an emulator device ready to receive commands, the script should run successfully without further changes.

In case we want to inspect more elements, we are going to use the new inspector downloaded previously for Appium® 2.0. The inspector should look like the view below.

Again, given the Appium® server is still running with an emulator device ready we need to add following two capabilities in order to start a session in the inspector:

- platformName, that we set for our needs to "Android"
- automationName, that we set to "UiAutomator2" for the driver to use

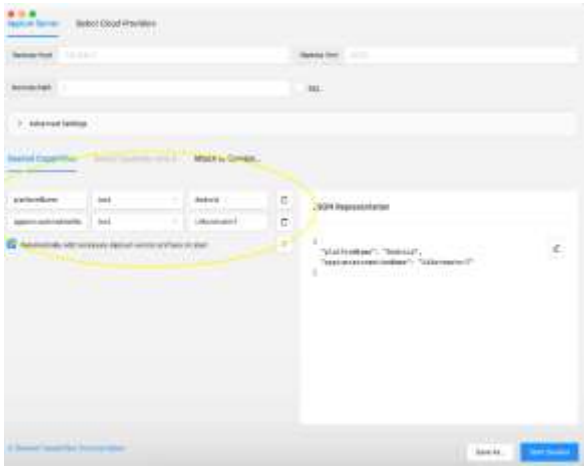
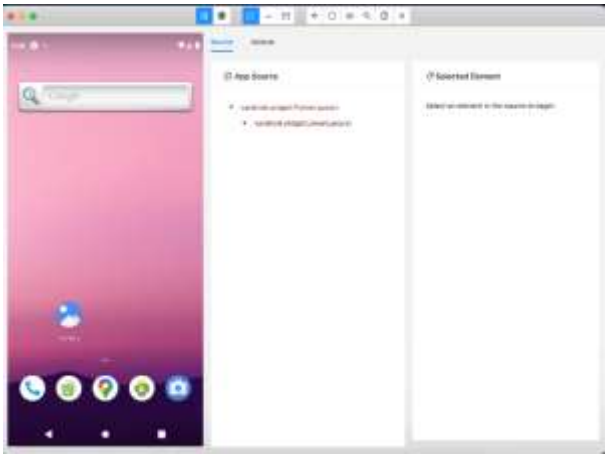


Figure 6 : UIAutomator2

Click Start session, the current view of the emulator is reproduced in the inspector and ready for the inspection analysis.



7 Bibliography

ID	Edition	Title	Edition / Version
01	Manoj Hans	Appium® Essentials	2015
02	Shankar Garg	Appium® Recipes	2016
03	ISTQB	CTAL Test Automation Engineer	2016
04	ISTQB	CTFL Mobile Application Testing	2019

8 Appendix A – Learning Objectives/Cognitive Level of Knowledge

The following learning objectives are defined as applying to this syllabus. Each topic in the syllabus will be examined according to the learning objective for it. The level of Knowledge is the same as per ISTQB®.

Level 1: Remember (K1)

The candidate will recognize, remember, and recall a term or concept.

Keywords: Identify, Remember, retrieve, recall, recognize, know

Level 2: Understand (K2)

The candidate can select the reasons or explanations for statements related to the topic, and can summarize, compare, classify, categorize, and give examples for the testing concept.

Keywords: Summarize, generalize, abstract, classify, compare, map, contrast, exemplify, interpret, translate, represent, infer, conclude, categorize, construct models

Level 3: Apply (K3)

The candidate can select the correct application of a concept or technique and apply it to a given context.

Keywords: Implement, execute, use, follow a procedure, apply a procedure

Level 4: Analyze (K4)

The candidate can separate information related to a procedure or technique into its constituent parts for better understanding and can distinguish between facts and inferences. Typical application is to analyze a document, software or project situation and propose appropriate actions to solve a problem or task.

Keywords: Analyze, organize, find coherence, integrate, outline, parse, structure, attribute, deconstruct, differentiate, discriminate, distinguish, focus, select

9 Appendix B – Glossary of Appium® Terms

Ref	Name	Meaning
JSF-01	JS Foundation	The Open JS Foundation is made up of 32 open-source JavaScript projects including Appium®, Dojo, Electron, jQuery, Node.js, and webpack. The mission is to support the healthy growth of JavaScript and web technologies by providing a neutral organization to host and sustain projects, as well as collaboratively fund activities that benefit the ecosystem.

10 Appendix C – Sources

ID	Link	Remark
01	www.Istqb.org	
02	https://github.com/appium/appium-desktop	
03	http://appium.io/	
04	https://docs.oracle.com/javase/tutorial/tutorialLearningPaths.html	
05	https://developer.android.com/studio	
06	https://raw.githubusercontent.com/Homebrew/install/master/install)	
6a	How to Install Homebrew on Mac {Step-by-Step} + How To Use It (phoenixnap.com)	
07	www.appium.org	
08	https://www.selenium.dev/documentation/en/webdriver/	Used in Exercise Doc
09	https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction	Used in Exercise Doc
10	https://en.wikipedia.org/wiki/XPath	Used in Exercise Doc
11	https://en.wikipedia.org/wiki/Session_(computer_science)	Used in Exercise Doc
12	http://appium.io/docs/en/writing-running-appium/caps/	Used in Exercise Doc
13	https://en.wikipedia.org/wiki/SOLID	
14	https://github.com/appium/appium	Main Appium® repository
15	https://github.com/appium/appium/tree/master/sample-code	Appium® Example's repository
16	https://appiumpro.com/editions/32-finding-elements-by-image-part-1	Link for Appium® 2.0
17	https://appiumpro.com/editions/101-ai-for-appium--and-selenium	Link for Appium® 2.0
18	http://appium.io/docs/en/writing-running-appium/server-args/index.html#server-flags	Appium® 2.0 server flags
19	https://Glossary.istqb.org	

11 Appendix D – Source Example Editor-APK

Here you can find the complete Java source code for the Editor exercise described in chapter “How to create a test script - step by step” above.

The complete project is stored in the zip below

APKEditor.Exercise.zip

Source : ioSampleTestEditor.java

```
import io.appium.java_client.android.AndroidDriver;
import io.appium.java_client.remote.AndroidMobileCapabilityType;
import io.appium.java_client.remote.MobileCapabilityType;
import org.apache.commons.lang3.RandomStringUtils;
import org.openqa.selenium.By;
import org.openqa.selenium.interactions.Actions;
import org.openqa.selenium.remote.DesiredCapabilities;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;
import org.testng.Assert;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;

import java.net.MalformedURLException;
import java.net.URL;
import java.util.Objects;

public class ioSampleTestEditor extends  Utils {

   final String fileName = RandomStringUtils.randomAlphanumeric(FILE_NAME_SIZE);

   @BeforeMethod
   public void setup() throws MalformedURLException {
      DesiredCapabilities caps = new DesiredCapabilities();
      String editorApkPath = Objects.requireNonNull(getClass().getResource("apk/text-editor.apk")).getPath();
      caps.setCapability(AndroidMobileCapabilityType.AUTO_GRANT_PERMISSIONS, true);
      caps.setCapability("skipUnlock", "true");
      caps.setCapability("noReset", "false");
      caps.setCapability("unicodeKeyboard", false);
      caps.setCapability("resetKeyboard", false);
      caps.setCapability(MobileCapabilityType.APP, editorApkPath);
      driver = new AndroidDriver<>(new URL("http://0.0.0.0:4723/wd/hub"), caps);
      wait = new WebDriverWait(driver, 10);
   }

   @Test
   public void fileCreationTest() {

      writeContentInFile();
      clickSaveButtonAndEditFileName(fileName);
      goToInternalStorageAndVerifyFileIsPresent(fileName);
   }

   public void writeContentInFile() {
      driver.findElement(By.xpath(TEXT_FIELD)).click();
      Actions action = new Actions(driver);
      action.sendKeys(MY_TEXT).perform();
   }
```

```
  public void clickSaveButtonAndEditFileName(String fileNme) {

    driver.findElementsByAccessibilityId(SAVE_BUTTON).get(0).click();
    wait.until(ExpectedConditions.visibilityOfElementLocated(By.xpath(EDIT_FILE_NAME))).clear();
    wait.until(ExpectedConditions.visibilityOfElementLocated(By.xpath(EDIT_FILE_NAME))).sendKeys(fileNme + FILE_EXTENSION);
    wait.until(ExpectedConditions.visibilityOfElementLocated(By.id(CONFIRM_FILE_NAME))).click();
    wait.until(ExpectedConditions.visibilityOfElementLocated(By.id(DONE_FILE_NAME))).click();
  }

  public void goToInternalStorageAndVerifyFileIsPresent(String fileName) {
    driver.findElementsByAccessibilityId(OPEN_BURGER_BUTTON).get(0).click();
    wait.until(ExpectedConditions.visibilityOfElementLocated(By.xpath(GO_TO_STORAGE_MANAGER))).click();
    wait.until(ExpectedConditions.visibilityOfElementLocated(By.xpath(GO_TO_INTERNAL_STORAGE))).click();
    getFilesFromInternalStorage();
    Assert.assertTrue(INTERNAL_STORAGE_CONTENT.contains(fileName + FILE_EXTENSION), "Verify Internal Storage contains" + fileName +
FILE_EXTENSION);
  }

  @AfterMethod
  public void teardown() {
    driver.quit();
  }
}
```

Source : Utils.java

```
import io.appium.java_client.MobileElement;
import io.appium.java_client.TouchAction;
import io.appium.java_client.android.AndroidDriver;
import io.appium.java_client.touch.offset.PointOption;
import org.openqa.selenium.By;
import org.openqa.selenium.NoSuchElementException;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;

import java.util.HashSet;
import java.util.List;
import java.util.Set;

public class Utils {

  public static final String TEXT_FIELD =
"/hierarchy/android.widget.FrameLayout/android.widget.LinearLayout/android.widget.FrameLayout/android.widget.FrameLayout/android.wi
dget.FrameLayout/androidx.drawerlayout.widget.DrawerLayout/android.widget.LinearLayout";
  public static final String SAVE_BUTTON = "Save";
  public static final String EDIT_FILE_NAME =
"/hierarchy/android.widget.FrameLayout/android.widget.LinearLayout/android.widget.FrameLayout/android.widget.FrameLayout/android.wi
dget.FrameLayout/androidx.appcompat.widget.LinearLayoutCompat/android.widget.FrameLayout/android.widget.FrameLayout/android.widg
et.LinearLayout/android.widget.LinearLayout/android.widget.EditText";
  public static final String CONFIRM_FILE_NAME = "android:id/button1";
  public static final String DONE_FILE_NAME = "com.rhmsoft.edit:id/done";
  public static final String MY_TEXT = "This is some text to write on a text editor for testing purpose.";
  public static final String OPEN_BURGER_BUTTON = "Open navigation drawer";
  public static final String GO_TO_STORAGE_MANAGER =
"/hierarchy/android.widget.FrameLayout/android.widget.LinearLayout/android.widget.FrameLayout/android.widget.FrameLayout/android.wi
dget.FrameLayout/androidx.drawerlayout.widget.DrawerLayout/android.widget.LinearLayout/android.widget." +

"LinearLayout/android.widget.FrameLayout/android.widget.LinearLayout/android.widget.FrameLayout/android.widget.ScrollView/android.wi
dget.LinearLayout/android.widget.HorizontalScrollView/android.widget.FrameLayout/android.view.View[3]";
  public static final String GO_TO_INTERNAL_STORAGE =
"/hierarchy/android.widget.FrameLayout/android.widget.LinearLayout/android.widget.FrameLayout/android.widget.FrameLayout/android.wi
```

```
dget.FrameLayout/android.widget.LinearLayout/android.widget.RelativeLayout/android.widget.FrameLayout/androidx.recyclerview.widget.RecyclerView/android.widget." +
        "RelativeLayout[1]/android.widget.LinearLayout/android.widget.TextView[1]";
  public static final String GET_FILES =
"/hierarchy/android.widget.FrameLayout/android.widget.LinearLayout/android.widget.FrameLayout/android.widget.FrameLayout/android.widget.FrameLayout/androidx.drawerlayout.widget.DrawerLayout/android.widget.LinearLayout/android.widget.RelativeLayout/android.widget.FrameLayout/androidx.recyclerview.widget.RecyclerView/android.widget.FrameLayout";
  public static final String GET_FILES_NAME = "//android.widget.RelativeLayout/android.widget.RelativeLayout/android.widget.TextView[1]";

  public static final String FILE_EXTENSION = ".txt";
  public static final int FILE_NAME_SIZE = 5;
  public static final int SCROLLS = 4;
  public static final Set<String> INTERNAL_STORAGE_CONTENT = new HashSet<>();

  public WebDriverWait wait;
  public AndroidDriver<MobileElement> driver;

  public void getFilesFromInternalStorage() {

    for (int i = 0; i <= SCROLLS; i++) {
      getInternalStorageContent();
    }
  }

  public void getInternalStorageContent() {
    wait.until(ExpectedConditions.visibilityOfElementLocated
        (By.xpath(GET_FILES)));

    List<MobileElement> listOf = driver.findElements(By.xpath(GET_FILES));

    for (MobileElement list : listOf) {

      String element = null;
      try {
        element = list.findElement(By.xpath(GET_FILES_NAME)).getText();
      } catch (NoSuchElementException e) {
        e.printStackTrace();
      }
      INTERNAL_STORAGE_CONTENT.add(element);
    }
    scrollDown();
  }

  private void scrollDown() {
    //if pressX was zero it didn't work for me
    int pressX = driver.manage().window().getSize().width / 2;
    // 4/5 of the screen as the bottom finger-press point
    int bottomY = driver.manage().window().getSize().height * 4 / 5;
    // just non zero point, as it didn't scroll to zero normally
    int topY = driver.manage().window().getSize().height / 8;
    //scroll with TouchAction by itself
    scroll(pressX, bottomY, pressX, topY);
  }

  private void scroll(int fromX, int fromY, int toX, int toY) {
    TouchAction = new TouchAction(driver);
    touchAction.longPress(PointOption.point(fromX, fromY)).moveTo(PointOption.point(toX, toY)).release().perform();
  }
}
```