

# **AUTOMATED APP TESTING ON REAL MOBILE DEVICES**

Software engineering and quality assurance



Bachelor's thesis

Information and communications technology, HAMK

Autumn, 2020

Patrick Nikanti

---

## TIIVISTELMÄ

Testiautomaatio ja sen monimutkainen sanasto olivat minulle tuoreita käsitteitä vielä ohjelmistokehitysurani alkuvaiheissa. Ne vaikuttavat olevan monelle muullekin alalla olevalle, jos ei tee työkseen ohjelmiston laadunvarmistusta. Tämän takia opinnäytetyössä käydään käytännön esimerkeillä, mitä testiautomaatio on, miten sitä kehitetään ja mitä siltä vaaditaan. Opinnäytetyössä käydään myös läpi, miten taustalla olevat kirjastot, kielet ja ohjelmistokehykset toimivat. Opinnäytetyön toimeksiantona oli kehittää päästä-päähän (end-to-end) testiautomaatio yhtä mobiilisovelluksen pääominaisuutta varten. Yhtenä vaatimuksena oli myös, että testiautomaatiota ajetaan oikeita mobiililaitteita (kuten Android ja iOS älypuhelimia) vasten. Automaatioympäristön pitäisi myös olla skaalautuva kymmeneen tai satoihin mobiililaitteisiin. Itse testiautomaatio on kehitetty Robot Framework automaatiokehityksellä, joka pyörii Python ohjelmointikielen päällä. Opinnäytetyö kuitenkin näyttää, että Python kieleen joudutaan turvautumaan suurimassa osassa ohjelmointilogiikasta. Näyttäen myös kuinka testiautomaation kehitys ei eroa perinteisestä ohjelmoinnista juuri ollenkaan. Työkokemusta testiautomaatiokehityksestä ja ohjelmiston laadunvarmistuksesta minulla oli opinnäytetyön kirjoittamisen aikana noin vuoden verran. Lopulta toimiva päästä-päähän testiautomaatio saatiin rakennettua. Testiautomaatiota ajetaan kymmeniä mobiililaitteita vasten. Kehitystiimimme voi varmistua, että sovelluksen ominaisuus toimii myös uusien ohjelmistoversioiden käyttöönottojen jälkeen. Itse olen tyytyväinen tuloksiin ja oppeihin mitä toimeksiannosta sain.

ABSTRACT

Test automation and its intricate glossary were new things for me, still in the early stages of my software development career. They seem to be for many others in the industry also if not working on software quality assurance. For this reason, the thesis uses practical examples of what test automation is, how it is developed and what is required of it. The thesis will also touch on how the underlying libraries, languages, and software frameworks work. The thesis's commission was to develop end-to-end test automation for a core feature of a mobile application. One requirement was that the test automation is run against real mobile devices (such as Android and iOS smartphones). The automation environment should also be scalable to dozens or hundreds of mobile devices. The test automation itself has been developed with Robot Framework automation framework that runs on top of the Python programming language. However, the thesis shows that resorting to Python language for programming logic is needed for most parts. Also showing that the development of test automation is almost no different from traditional programming. I had about a year of work experience in test automation development and software quality assurance during writing this thesis. Eventually, working end-to-end test automation was built. The test automation is run against dozens of mobile devices. Our development team can ensure that the application feature will continue to function even after new software version deployments. I am happy with the outcome and the learnings I got from the commission.

# Sisällys

1	Introduction.....	1
2	Technologies.....	2
2.1	Robot Framework .....	2
2.1.1	Overview of keywords.....	3
2.1.2	Overview of output files.....	4
2.1.3	Library, resource, and variable files .....	5
2.1.4	Extending Robot Framework.....	6
2.1.5	Gherkin .....	7
2.2	Pabot .....	7
2.3	Appium.....	8
2.3.1	Overview of Appium.....	8
2.3.2	Locator strategies.....	10
2.3.3	Real device automation .....	11
2.4	ADB.....	12
2.4.1	Overview of ADB .....	12
2.4.2	Implementing ADB .....	12
2.5	Libimobiledevice .....	14
2.5.1	Overview of Libimobiledevice.....	14
2.5.2	Problems .....	15
2.5.3	Implementing Libimobiledevice.....	16
2.6	Other components.....	19
2.6.1	Video Calls .....	19
2.6.2	Requests .....	19
3	Development.....	21
3.1	Setup .....	23
3.1.1	Overview .....	23
3.1.2	Intercom.....	28
3.1.3	Mobile devices .....	32
3.1.4	Temporary files .....	35
3.1.5	Problems .....	36
3.2	Given .....	38
3.2.1	Walkthrough.....	38
3.2.2	Problems .....	43

3.3	When.....	44
3.3.1	Walkthrough.....	44
3.3.2	Problems .....	50
3.4	Then .....	52
3.4.1	Walkthrough.....	52
3.4.2	Problems .....	53
3.5	Teardown .....	54
3.5.1	Walkthrough.....	54
3.5.2	Problems .....	61
3.6	Summary .....	62
	References .....	64

## 1 Introduction

The thesis goes through the development of test automation called “video call and door open”. The test automation is run on any new software deployment that is part of the system under test. The system under test is dependent on many software components and services, with some being third-party. This makes it hard to trigger the test automation automatically when there is a new software deployment, especially if the said deployment is a third-party component or service. To tackle this, the test automation is run once a day, and if needed, it can be triggered by hand also. Test automation should not be confused with monitoring because the intention is not to monitor but to conduct regression testing. Regression testing is done to verify that the software changes do not impact the product’s existing functionality. (Software Testing Help, 2021). Usually, a test automation case’s main purpose is to test only one or two things (hence the name “video call and door open”). The “video call and door open” test automation touches and tests lots of different things indirectly, e.g., are the intercom and application background services working as expected?

The solution under test consists of mobile applications for Android and iOS platforms, third-party cloud services (PaaS or Platform as a Service), and third-party hardware (intercoms, access units, keypads). The main purpose of the mobile application is to act as an access controller to apartment buildings. It gives the apartment tenant freedom to grant access to visitors remotely through mobile devices, without a need for traditional hard-installed door phones inside each apartment. This means the visitor can call through the building’s intercom directly to the tenant’s mobile device. The solution supports two ways of calling, video call (requires smartphone or tablet, application, and access to the internet.) Another way is through audio call (requires a phone with GSM access).

The video call provides video and audio feed from the intercom and buttons: open door, mute, speaker, and hang up. The video call and its interface are handled by the application, but the call notification or the push notification is not. The video call initiation mechanism is provided by the mobile device’s push notification service. The push notification in mobile devices is handled at the operating system level. (King, n.d.). The audio call works like a normal GSM call; the door, in this case, is opened through DTMF (Dual-tone multi-

frequency) code. The DTMF signals are sent by pressing the corresponding keys on the mobile device (e.g., 0123#). The DTMF signals are sent by pressing the corresponding keys on the mobile device (e.g., 0123#). (Soluno, 2019). After the code is sent, it is validated on the intercom side. If the received code is correct, the door is opened. In the audio call case, the call from the intercom is converted from the IP network to PSTN (Public switched telephone network) and vice versa via VoIP (Voice over Internet Protocol) gateway. (McCraw, 2020). The solution also enables the tenant to access the apartment building without a traditional key. Either with an RFID (radio frequency identification) tag or with the tenant's mobile device via BLE (Bluetooth low energy) pairing to the building's intercom or the access unit.

The test automation case covers the following case: a visitor calls a video call from the building's intercom to the apartment tenant. When the call is made from the intercom, the tenant receives a push notification call to the mobile device that opens a video call by accepting. When the "open door" button is activated from the video call, the front door opens, paired with an audio cue for the visitor and the tenant.

The "video call and door open" test automation is the first touching point to application test automation via real mobile devices in the project. In the future, expanding to other endeavors, like "audio call and door open" test automation will be relatively easy. This is because the "real mobile device" automation foundation is built with this test automation.

## **2 Technologies**

### **2.1 Robot Framework**

Robot Framework (or Robot for short) is the heart of the technology stack. Robot is an open-source test automation framework built on top of Python programming language. Robot Framework's development started from Pekka Klärck's master's thesis in 2005 but was open-sourced later in 2008. (Klärck, 2021; Laukkanen, 2006)

### 2.1.1 Overview of keywords

Robot uses easy to understand keyword syntax. (Robot Framework, n.d.-a). Keywords can be thought of as functions or method calls in conventional programming languages like Python or JavaScript, as they can have arguments and return values as traditional functions have. Keywords really shine in higher-level abstractions, as the keywords tend to be really descriptive. (Minakov, 2018).

```
*** Keywords ***
Open Browser To Login Page
    Open Browser    ${LOGIN URL}    ${BROWSER}
    Maximize Browser Window
    Set Selenium Speed    ${DELAY}
    Login Page Should Be Open
```

Picture 1. An example of browser test automation setup and its keyword usage. (Robot Framework, n.d.-b)

In the example (Picture 1) “Open Browser” keyword is used with positional arguments “LOGIN URL” and “BROWSER”. Open Browser is a keyword provided by a browser automation keyword library called “SeleniumLibrary”. In reality, “Open Browser” is a method defined in Python but wrapped with a “keyword” decorator to make the method explicitly available as a keyword in Robot Framework. (Robot Framework, 2020a; Robot Framework, 2020b)



```
@keyword
def open_browser(
    self,
    url: Optional[str] = None,
    browser: str = "firefox",
    alias: Optional[str] = None,
    remote_url: Union[str, bool] = False,
    desired_capabilities: Union[str, dict, None] = None,
    ff_profile_dir: Optional[str] = None,
    options: Any = None,
    service_log_path: Optional[str] = None,
    executable_path: Optional[str] = None,
) -> str:
    """Opens a new browser instance to the optional ``url``.
```

Picture 2. Python method “open browser” explicitly made as keyword with the “keyword” decorator. (SeleniumLibrary, n.d.)



Robot does automatic underscore, space, and letter case conversion, when searching for the corresponding keyword, meaning keyword “hello” can be called as “Hello”, “hello” or even “h e l l o” in Robot Framework. (Robot Framework, 2020c)

### 2.1.2 Overview of output files




## H9436: Video Call & Door Open E2E & TA-1004: Video Call & Door Open E2E Log

Generated  
20201127 09:58:53 UTC+02:00  
2 days 6 hours ago




### Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	2	2	0	00:00:39	
All Tests	2	2	0	00:00:39	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
H9436	1	1	0	00:00:14	
TA-1004	1	1	0	00:00:26	
Video Call	2	2	0	00:00:39	

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
H9436: Video Call & Door Open E2E & TA-1004: Video Call & Door Open E2E	2	2	0	00:00:46	
H9436: Video Call & Door Open E2E & TA-1004: Video Call & Door Open E2E: H9436: Video Call & Door Open E2E	1	1	0	00:00:30	
H9436: Video Call & Door Open E2E & TA-1004: Video Call & Door Open E2E: TA-1004: Video Call & Door Open E2E	1	1	0	00:00:43	

Picture 3. Log file overview of video call and door open test automation.

Test results or output files are automatically generated by Robot after the test execution has finished. The generated artifacts are output, log, and report files. The output file contains all the test execution results in machine-readable XML format and acts as a data propagator to log and report files. Log and report files are generated from the output file. The report file gives a higher-level overview of the executed test automation, clearly stating whether the tests have passed or failed. Log files have a more detailed output of the executed test, including information about the executed keywords and their possible outputs. Depending on the log level, the output can be extensive. Available log levels for Robot are: fail, warn, info, debug, and trace. (Robot Framework, 2020d; Robot Framework, 2020f).

### 2.1.3 Library, resource, and variable files

When building software with traditional programming languages, it is considered good to have the code as modular as possible. This increases the reusability of the components and makes the code more maintainable. (Khalfallah, 2020). This is no different in Robot; the keywords should be maintainable and reusable.

```

*** Settings ***
Library      pabot.PabotLib
Library      AppiumLibrary      timeout=10      run_on_failure=Capture Screenshot
Library      Collections
Library      ../resources/loggable.py
Library      ../resources/helper.py
Variables    ../resources/variables.py
Resource     ../resources/helper.robot
Resource     ../resources/given.robot
Resource     ../resources/when.robot
Resource     ../resources/then.robot

```

Picture 4. Settings section in video call and door open main resource file.

The library, variable, and resource files are mostly imported through Robot's "Settings" section. Library files or test libraries contain the lowest-level keywords that often interact with the system under test. Keywords used in test-cases always originate from some test library, but they are often called through a high-level user keyword. These libraries are imported in the Settings section by specifying the name of the file, or the name of the class

or the module of the library. Robot has some built-in libraries, such as “Collections”, that are available out-of-the-box (Picture 4), but these libraries still need to be imported explicitly. (Robot Framework, 2020e). Variable files provide variables for the test-case. Variables can also be set from the command line. However, variable files allow creating variables dynamically and create other variables types than string, e.g., dictionaries or lists that would be impossible through arguments otherwise. Variable files are usually implemented as Python modules (Picture 4). Robot supports all Python base type variables and is not limited to only those, as variables can contain any objects, such as Java hash tables. (Robot Framework, 2020g). Resource files are basically the same as test case files, with some differences, such as the test case tables and some import settings that cannot be added to a resource file. However, the resource file has the same higher-level structure as the test case file. You can set libraries, variables, keywords, and even other resource files in a resource file. (Robot Framework, 2020h). The “video call and door open” test automation uses a separate resource file to keep the suite file clean from all the imports. (Picture 4).

#### **2.1.4 Extending Robot Framework**

Robot makes it easy to extend its functionality by creating keyword libraries. Python library can be imported without having to do any alteration to the original code. Robot does still provide helper libraries for building these keyword libraries. For example, “robot.api.deco” library provides “library” and “keyword” decorators. The “Library” decorator allows configuring the library’s scope, version, documentation format, and listener. It also disables the automatic keyword discovery, making it mandatory to decorate methods with the “keyword” decorator to expose them as keywords. The “keyword” decorator can also be used to name the function as a keyword explicitly. For example, the function “parse cookies” can be explicitly named as a keyword “Validate Cookies”. (Robot Framework, 2020a). Robot comes packed with lots of features, but the power in Robot is the ease to extend its functionality, because of the underlying programming language Python. This is also shown in the sheer number of third-party libraries available. (Awesome Robot Framework, 2021)

### 2.1.5 Gherkin

Robot Framework test cases can be written in many flavors, but the “video call and door open” test automation is written in Gherkin style or in behavior-driven style. Gherkin uses behavior-driven test syntax, meaning the test case is fit into a “Given-When-Then” formula. This is especially useful if the test cases must be understood by non-technical persons, e.g., non-technical project stakeholders.

“Given” describes the initial context of the system, typically something that happened in the past, e.g., “Given user has logged in to the web application”.

“When” step is used to describe an event or an action. This can be a person interacting with the system or an event triggered by another system. For example, “When user inputs bowl of oatmeal into the search field” would be valid for the “When” step.

“Then” step is used to describe an expected outcome or result, e.g., “Then user should see recipe for a bowl of oatmeal”. The “Then” step should use an assertion to compare the actual outcome to the expected outcome.

There are also “And” and “But” steps in the Gherkin syntax. If the test case has successive “Given”, “When” or “Then” steps, the “And” and “But” steps are valid replacements. (Cucumber, 2021a).

Robot Framework, by default, ignores the “Given”, “When”, “Then”, and “But” prefixes from keywords, enabling calling the keywords directly without the Gherkin prefix. This also allows the use of the same keyword with two different prefixes. (Robot Framework, 2020i)

## 2.2 Pabot

Pabot is a parallel executor for Robot, saving execution time when for example, an action has to be repeated multiple times with different test data to provide different outputs. Different executors do not block each other by default, which will save execution time considerably. Pabot is used through command-line with “pabot” command, replacing the Robot’s start command “robot”. (Pabot, 2020a). In addition to supporting all Robot’s

command-line options, Pabot has its own command-line options, such as “processes”, “pabotlib”, and “argumentfile”. (Pabot, 2020b). The example options are the most important for the “video call and door open” test automation. The “Processes” argument caps the number of parallel processes that can run at the same time. The default is a max of 2 and CPU count.

The “Pabotlib” argument starts a PabotLib remote server, but in addition to the argument, PabotLib must also be imported in Robot files in order to be used (Picture 4). PabotLib enables actions like locking and resource distribution within parallel processes through keywords inside Robot. (Pabot, n.d.-a). For example, the locking mechanism is relied on in the “video call and door open” test automation, for some actions can only be used singularly within one process, e.g., calling with the intercom to the mobile device.

The “Argumentfile” arguments are used to control the number of Pabot executors in the “video call and door open” test automation, with an argument file per device under test. The argument file holds command-line options, e.g., “—variable myvariable:hello\_world!”. They are a way to provide lots of arguments to a test with still keeping the start command tidy. Argument files themselves are not Pabot specific, but the argumentfile with index notation is (argumentfile1, argumentfile2, etc.). Argument files support all the options or flags supported by Robot’s command-line arguments. (Robot Framework, 2020j).

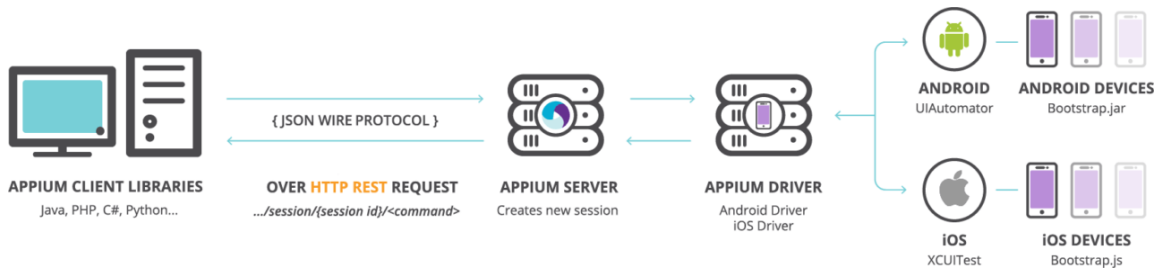
## **2.3 Appium**

### **2.3.1 Overview of Appium**

Appium is an open-source framework for mobile automation. Appium can be used to automate native applications in platforms like iOS, Android, and Windows. Appium wraps vendor-provided frameworks like XCUITest and UiAutomation (iOS platform), UiAutomator and UiAutomator2 (Android platform), and WinAppDriver (Windows platform) to one API, called the WebDriver API. WebDriver in Appium is an extended version of the original Selenium WebDriver with extra API methods like multi-touch gestures and screen orientation, useful for mobile automation. (Appium, n.d.-a; Appium, n.d.-b). Appium in its

heart is a REST API operating over HTTP, making it possible to write a client in any programming language that has an HTTP client API. (Appium, n.d.-c).

There is a multitude of client libraries available for Appium, written in various programming languages and frameworks such as Ruby, Python, Java, Objective C, and Robot Framework. (Appium, n.d.-b). The choice was to use the Robot Framework client, as it is the software stack's main automation framework.



Picture 5. Appium high-level overview diagram. (Arieli, n.d.)

To start an automation session with Appium, a JSON object called desired capabilities must be passed to the Appium server. In Robot Framework's Appium client (AppiumLibrary), the session is created with the "Open Application" keyword, where the desired capabilities are passed native to the client's way (Picture 6).

```

${APPIUM}= Open Application
...   ${PHONE}[appium][url]
...   udid=${PHONE}[udid]
...   systemPort=${PHONE}[appium][system_port]
...   alias=${PHONE}[appium][alias]
...   platformName=${OS}
...   platformVersion=${PHONE}[os][version]
...   appPackage=${PHONE}[application][package_name]
...   appActivity=${PHONE}[application][main_activity]
...   autoGrantPermissions=${true}
...   newCommandTimeout=${180}
...   noReset=${true}

```

Picture 6. Open Application keyword in video call and door open test automation.

Because under the hood Appium is an HTTP server, the session is created using HTTP methods. The HTTP API specification of starting a new session is a request method POST to endpoint “/session”, with the request body containing the desired capabilities JSON object. (Appium, n.d.-d). The passed capabilities (Picture 6) tell the Appium server how the session should be handled, such as which mobile device to connect to, what application will be automated, or if the application state will be reset before starting the session (Appium, n.d.-e). There are also driver-specific capabilities available for fine-tuning the session. These are usually well documented in the driver’s own documentation.

### 2.3.2 Locator strategies

Appium uses different kinds of locator strategies for finding the corresponding element in the application under test. In the AppiumLibrary client, to click a button in the application, the keyword “Click Button” must be called. The keyword takes positional argument containing the locator to the element, e.g., “//UIATableView/UIATableCell/UIAButton” (XPath locator). There are lots of different locator strategies. For example, in the AppiumLibrary client, there are supported locator strategies like ID, Accessibility ID, XPath, Class, Android, Name, iOS, and CSS. Some locator strategies are WebDriver specific. For example, the strategy “Name” is only available for the Selendroid automation engine. The full list of available locator strategies is available in the AppiumLibrary client’s

documentation. (AppiumLibrary, n.d.). The locator strategies originate from Selenium and W3C WebDriver working draft (W3C, 2020; Appium, n.d.-b)

### **2.3.3 Real device automation**

Typically, Appium requires setting-up if performing automation on real mobile devices, like changing OS-specific settings. For example, on Android, the device must be connected via USB, and the USB debugging must be turned on from the developer settings. (See chapter 2.4).

On the iOS side, the real device automation via Appium exclusively works on macOS only, but some workarounds can get this process working with other operating systems. To get an Apple mobile device to communicate with any other platform than the macOS requires the USB multiplexer from Apple, which is not available directly to other operating systems, only through open-source implementation (i.e., `usbmuxd` by `libimobiledevice` project) or by getting iTunes, which provides the Apple's original USB multiplexer. (See chapter 2.5). The iOS side also requires developer settings turned on from the mobile device, and UI automation settings turned on from the developer settings. The developer settings are automatically turned on when the mobile device is connected to an Apple computer with Xcode installed. There is also a workaround for enabling developer settings for other operating systems. The process requires a tool to mount images to Apple devices (i.e., `ideviceimagemounter` by `libimobiledevice` project) and a developer disk image from Xcode. (See chapter 2.5).

Additionally, both Android and iOS devices will prompt to trust the host when the device is connected. These steps are at least required to get the automation to work on real mobile devices.



## 2.4 ADB

### 2.4.1 Overview of ADB

ADB or Android Debug Bridge is a command-line tool to communicate with Android devices. ADB is a client-server program consisting of three components: client, daemon, and server. The client is from which the commands are sent. For example, ADB command-line tool and pure-python-adb (third-party Python client for ADB) are ADB clients. ADB daemon runs the commands on the device, that are sent from the client to the Android device. The daemon runs as a background process in all Android devices. Since the daemon and the client cannot communicate directly, there must be a broker in-between to translate the commands from the client to the daemon, that is the ADB server. ADB server manages the communication between the client and the daemon, and it must be running on the host to communicate with the Android device. (Android Developers, 2021a).

To use ADB with connected devices, a device must be connected over USB, and USB debugging must be enabled from the device. The USB debugging setting is hidden under developer settings in Android devices. To access developer settings, the most common way to unlock it is to tap “Build number” seven times in the “About device” section. The “Build number” may be in different places, depending on the device vendor. (Android Developers, 2021b). ADB enables actions like entering the device shell, installing, and uninstalling applications, getting screenshots, and getting system logs from the device. (Android Developers, 2021a; pure-python-adb, 2020). The “video call and door open” test automation’s technology stack uses pure-python-adb client for ADB because it gives programmatic access to the ADB actions. Python is native to the Robot Framework ecosystem, and it is desired to use homogenous tools.

### 2.4.2 Implementing ADB

ADB in the test automation is implemented in few ways: adbserver, adbbindings modules, and adbkeywords keyword library. The adbserver module provides Android device data retrieving and ADB server handling mechanisms. Adbbindings module, on the other hand,

provides actions such as turning the device screen on and off, opening and closing the status bar, internet connection checking, and application installing and uninstalling.

```
def open_status_bar(self):
    """
    Sends "cmd statusbar expand-notifications" to device.

    The action is currently not validated!

    Returns:
        command status: status of the result as dict
    """
    logger.info(f'{self.deviceModel}: Trying to open status bar')
    self.device.shell('cmd statusbar expand-notifications')
    response = {
        'deviceModel': self.deviceModel,
        'success': True,
        'result': {
            'action': 'open status bar',
            'message': 'Opened status bar successfully',
        },
    }

    logger.info(f'{self.deviceModel}: {response}')
    return response
```

Picture 7. Open status bar method provided by adbbindings module.

In the example (Picture 7), the Android device status bar is opened. This is done by interacting with the device shell through ADB (pure-python-adb client). The command “cmd statusbar expand-notifications” is sent to the device shell. This example, particularly, is not a good example of a robust method because there is no validation to verify that the status bar has actually opened. This is due to its impossible or very hard to get the state of the status bar. All the adbbindings’ method returns are “standardized” to be the same way. (Picture 7). Also, most of the ADB actions are implemented this way, as the underlying ppadb provides an easy-to-use interface to interact with the device. Many actions can be done through the device shell and things that ADB and ppadb provide. This is one reason why there are many more actions on the Android side than on the iOS side. (see chapter 2.5.3).

## 2.5 Libimobiledevice

### 2.5.1 Overview of Libimobiledevice

Libimobiledevice is a cross-platform open-source library to communicate with iDevices (portable devices, such as iPhones, iPads, and iPods manufactured by Apple).

Libimobiledevice library is written in C programming language, with bindings for Python, through a superset of Python called Cython. (Libimobiledevice, 2020a).

Usbmux daemon or `usbmuxd` is the socket daemon used to communicate with Apple devices in the Libimobiledevice library. In fact, `usbmuxd` is the only way to communicate with Apple devices through USB pipe. `usbmuxd` is Apple's USB multiplexer for its own host applications, such as iTunes and Xcode. `usbmuxd` uses a TCP-like protocol but is not directly TCP compliant, one hypothesis to this is that it is more efficient this way. The original `usbmuxd` is available for macOS and Windows (through iTunes) platforms, but not for Linux. This is because no Apple software is available for Linux that requires USB communication to Apple devices (i.e., iTunes or Xcode). To this, a cross-platform open-source implementation of `usbmuxd` has been created and maintained by the libimobiledevice project. The `usbmuxd` project is available as an individual project, and it is hosted in GitHub.

The communication between the iDevices and the host is based on encrypted plist records. (Gabilondo, 2018). Plist or property list file is Apple's way to store serialized objects. Plists are available in XML and binary formats. (Apple Inc., 2018). Libimobiledevice outputs plist records in XML format or "stripped-down" raw string format.

`usbmuxd` is used to handle the USB connections at a lower level. On the higher level, there is a private API called as `MobileDevice` (`MobileDevice.framework` for macOS and `MobileDevice.dll` for Windows) used by Apple's host applications to transfer data between iDevices and the host. There is also a cross-platform open-source implementation of `MobileDevice` API available, called `libusbmuxd`. `libusbmuxd` has been created for the same reason `usbmuxd` was created; no Apple software that requires USB pipe is available for the Linux platform. Thus, there is no infrastructure built to communicate with iDevices on Linux. `libusbmuxd` communicates with a daemon called `lockdownd` operating in the iDevices.

Lockdown provides iOS system information and access to other internal device services. There are many services available for performing actions you would do with ADB on Android side, like install and uninstall applications (`com.apple.mobile.installation_proxy`), requesting screenshots (`com.apple.mobile.screenshotr`), and getting system logs from the device (`com.apple.debugserver`). (Gabilondo, 2018).

Libimobiledevice also enables activating developer settings through the `ideviceimagemounter` tool, but not directly. An Apple developer disk and signature from an Apple computer with Xcode installed needs to be provided to enable the settings. Then the developer image needs to be mounted with the signature to the Apple device. This mounting of the developer disk enables the developer settings on the Apple mobile device. (Chatterjee, 2016).

### **2.5.2 Problems**

As mentioned earlier in the chapter, Python bindings are available for the `libimobiledevice`. However, due to a lack of documentation for building the tool for Windows with the Python bindings, they are not used. (Libimobiledevice, 2020b). There was an effort to build `libimobiledevice` with Python bindings for Windows, but this came with pitfalls, like lacking Python-dev package for MSYS2 (Build platform for Windows) and Windows in general. Python-dev includes Python developer headers needed to build the Python bindings for `libimobiledevice`. There is a workaround to implement the Python developer headers, but other problems arose after that. (Vincent, 2014). Other ways to go about this would have been to cross-compile the library, but this idea was scrapped completely, as others seemed to have problems with cross-compiling. (Geiszl, 2016). The working solution was to use a pre-compiled version of `libimobiledevice` for Windows provided by Quamotion. The pre-compiled version does not come with Python bindings either, presumably for the same reason. It is hard or impossible to set up the system to compile the library with the Python bindings.

### 2.5.3 Implementing Libimobiledevice

Libimobiledevice's command-line tools are wrapped via Python and its subprocess module. The libimobiledevice library consists of three modules: core, setup, and actions. The core module provides helper methods (for example, parsing subprocess output and validating libraries). The setup module contains device UDID (unique device identifier) fetching and device data fetching. The setup module's functionality is the same as the adbserver module's functionality on the Android side. The actions module contains methods like fetching the device name and locale, checking if the application exists on the device, and uninstalling and installing applications - basically, the same functionality as the Android's adbbindings module. Mirroring the Android and iOS modules was the intention when the modules were planned. To make them as close to each other as possible.

When the libimobiledevice library gets instantiated, all the tools or executables provided in the libimobiledevice library are run. Their versions are checked via the "--version" flag with the subprocess module. If the command's return code is something else than "0" (meaning the command executed successfully without errors) or if the executable was not found, an error that the library was not found will be raised.

In the example (Picture 8), Apple's mobile device UDIDs are fetched. A command is issued through the subprocess's run method. It is given the "idevice\_id" executable as a parameter, and other parameters, such as where the stdout (standard output) and stderr (standard

error) of the command are captured. The outputs are captured to a variable called “udids”. After the command has run, the output exit code is checked that it is “0”.

```
def get_udids(self) -> list:
    """
    Get unique device identifiers of connected iOS devices.
    """
    ret = []
    logger.info('Trying to get udids from connected devices.')
    try:
        udids = run(
            ['idevice_id'],
            stdout=PIPE,
            stderr=PIPE,
            cwd=self.settings['cwd'],
            timeout=self.settings['timeout'],
        )
        assert udids.returncode == 0, f'idevice_id: Command did not return "0". Output: {udids}'

    except FileNotFoundError as err:
        raise AssertionError(
            'Could not find idevice_id tool. Please check and verify that the tool is accessible from CLI.'
        ) from err

    unfiltered_udids = self.parse_io(udids.stdout)

    # Get the first value from splitting a string with whitespace
    for udid in unfiltered_udids:
        ret.append(udid.split()[0])

    logger.debug(f'Return value: {ret}')
    return ret
```

Picture 8. Get UDIDs method from libimobiledevice library.

Next, in the method, the command’s output is parsed via the “parse io” method, which decodes the output to UTF-8 format and possibly splits the string as an array by its line

endings.

```
def parse_io(self, io: bytes, to_list: bool = True) -> (list, str):
    """
    Decode bytes to string. Usable when parsing subprocess.run() method output.
    Uses encoding given to the class (defaults to UTF-8).

    Arguments:
        io: Stdout or stderr bytes from subprocess.run() method
        to_list: Convert io to list. Expects Boolean type. Defaults to True.
    """
    if to_list:
        ret = io.decode(self.settings['encoding']).splitlines()
    else:
        ret = io.decode(self.settings['encoding'])

    return ret
```

Picture 9. Parse IO method from libimobiledevice library.

To the decoded and parsed output, there is one last parsing. (Picture 8). The array is split by whitespace, and the left side of the string is retrieved. The left side has the wanted identifier, and the right side has optional information that is not needed. Lastly, the parsed UDIDs are returned.

Most methods provided by the libimobiledevice library are built the same way as the “get udids” method. First, the information is obtained from the libimobiledevice tool via subprocess. Then the output is parsed and validated. A keyword library has been built specially to wrap these libimobiledevice methods for the test automation. Currently, it only provides only one keyword, the “Install Latest Application Version” keyword. The keyword checks from the device if the application has been installed and its version. If the version mismatches or the application is not installed, the host provided application will be installed, and the possible mismatch version will be uninstalled.

There are not many actions (actions like waking and putting the device screen to sleep or accessing the device shell) that can be done with libimobiledevice or through any iOS communication tool that can be done on the Android side with ADB, or not without hacking the system. One reason for this is the closed-source nature of Apple products. However, this

is not a big setback; the most important thing that these tools (ADB and libimobiledevice) can provide for the test automation is data gathering, which is possible via libimobiledevice.

## 2.6 Other components

### 2.6.1 Video Calls

The video call feature uses the SIP (Session Initiation Protocol) signaling protocol for controlling (initiating, maintaining, and terminating) video and audio call sessions. SIP devices can establish connections directly with each other through direct SIP call or, more typically, through SIP Proxy and SIP Registrar.

SIP Registrar is a network server responsible for user registration in a certain network section. SIP device registration is necessary for a user to be accessible to the others via SIP number (example of SIP number URI, [sip:example-user@voip-provider.com](mailto:sip:example-user@voip-provider.com)). SIP proxy is a network server responsible for call routing. There can be one or more SIP proxies between the users. (2N, n.d.; Ramella, 2021).

SIP in the internet protocol suite is in the application layer, where protocols like HTTP, HTTPS, and TLS/SSL reside. SIP is designed to be independent of the underlying transport layer protocol. In the system under test's video calls, SIP is paired with the transport layer's TCP (Transmission Control Protocol). However, the user can switch to UDP (User Datagram Protocol) in the mobile application if some connectivity problems are present. The SIP also supports both IPv4 and IPv6. For the video and audio streams, RTP (Real-Time Transport Protocol) is used. The port numbers, protocols, and codecs in the streams (audio and video) are defined and negotiated via the SDP (Session Description Protocol). (IDG Communications, Inc., 2004)

### 2.6.2 Requests

Requests is an HTTP library for Python, allowing to send HTTP requests, for example, to communicate with HTTP APIs. Requests library abstracts all the complexities of making HTTP



requests into a simple library. With Requests, it is possible to consume the intercom’s built-in HTTP API, making actions like calling possible. (Picture 10).

```

def call_contact(self, number: str):
    """
    HTTP GET api/call/dial?number=arg(number)

    Arguments:
        number: number can be retrieved from request get_contacts

    Returns:
        response: response as json
    """
    endpoint = urljoin(self.api, f'api/call/dial?number={number}')
    response = requests.get(
        endpoint,
        headers=self.headers,
        auth=(self.username, self.password),
        verify=False,
    )
    self._check_header(response)
    self._check_auth(response)
    logger.debug(f'Received from endpoint {endpoint}\n{response.json()}')
    return response.json()

```

Picture 10. Method “call contact” in intercom client library.

A GET or POST request must be done to the endpoint “api/call/dial” and supply it with an argument “number” to perform a call. The argument “number” in the test automation is fetched from another intercom API endpoint. If the request is made with the POST method, the request arguments or body would be supplied in a Python dictionary and given to the POST method as “data” or “json” argument. In the GET method, the arguments are supplied as query string directly to the request URL. (Picture 10). The header and the authentication are also supplied to the request in the intercom client library. The header contains content-type, which is “application/json” because JSON data is expected from the endpoint. The authentication argument is supplied by the arguments username and password as a tuple type. If the “auth” argument is not supplied specifically with a type, e.g., “auth=HTTPDigestAuth(username, password)”, the authentication is a type of HTTP basic authentication. This is also used in the “video call and door open” test automation. As a general note, the basic authentication should only be used over an encrypted protocol like

HTTPS. While basic authentication encodes the credentials in base64, the credentials are not encrypted, making them easy to decode and vulnerable in plain HTTP. (Mozilla Developer Network, 2021).

```
def _check_header(self, response):
    assert (
        response.headers.get('content-type') == self.headers['Content-Type']
    ), 'HTTP request response returned unsupported header content-type.'

def _check_auth(self, response):
    try:
        assert (
            response['error']['description'] != 'authorization required'
        ), 'Authorization failed! '
    except (KeyError, TypeError):
        pass
```

Picture 11. Response validations in intercom client library.

Because the Requests library makes the response objects highly readable and easy to parse, it is easy to validate that the response header's content-type is in the "application/json" type format. Furthermore, the response object does not contain an authorization error. (Picture 11). Another possible validation would be to check that the response object does not contain any errors. In this case, the key "error" would be validated, that it does not exist in the response object, and if it does, an error will be raised. Requests also supply JSON methods to encode and decode Python dictionaries. This is especially useful when dealing with JSON content that, for example, the intercom HTTP API provides. To decode a JSON object to a Python dictionary, the "json" method call must be used to the response object. (Picture 10).

### 3 Development

The development of video call and door open test automation started from a need to monitor the feature in the production environment because this feature has not always worked reliably. The system under test depends on third-party hardware (intercom) and software (cloud services). The system is used with dozens of different phone and tablet variants, be it Android or iOS operating system, with different OS flavors and vendor-specific

settings on the Android side. Also, not to exclude different ISPs (internet service providers) in different countries and their settings. Is the network operating in IPv4 or IPv6? How does the call behave when the connectivity is poor or the client's internet is in a roaming setting? There are tons of different variables in this equation that may cause the video call feature not to work as intended. Therefore, some certainty is needed that the feature is working as expected. That is why the test automation was developed initially.

The creation of test automation should start with writing the high-level keywords without going too much into implementation. (Picture 12).

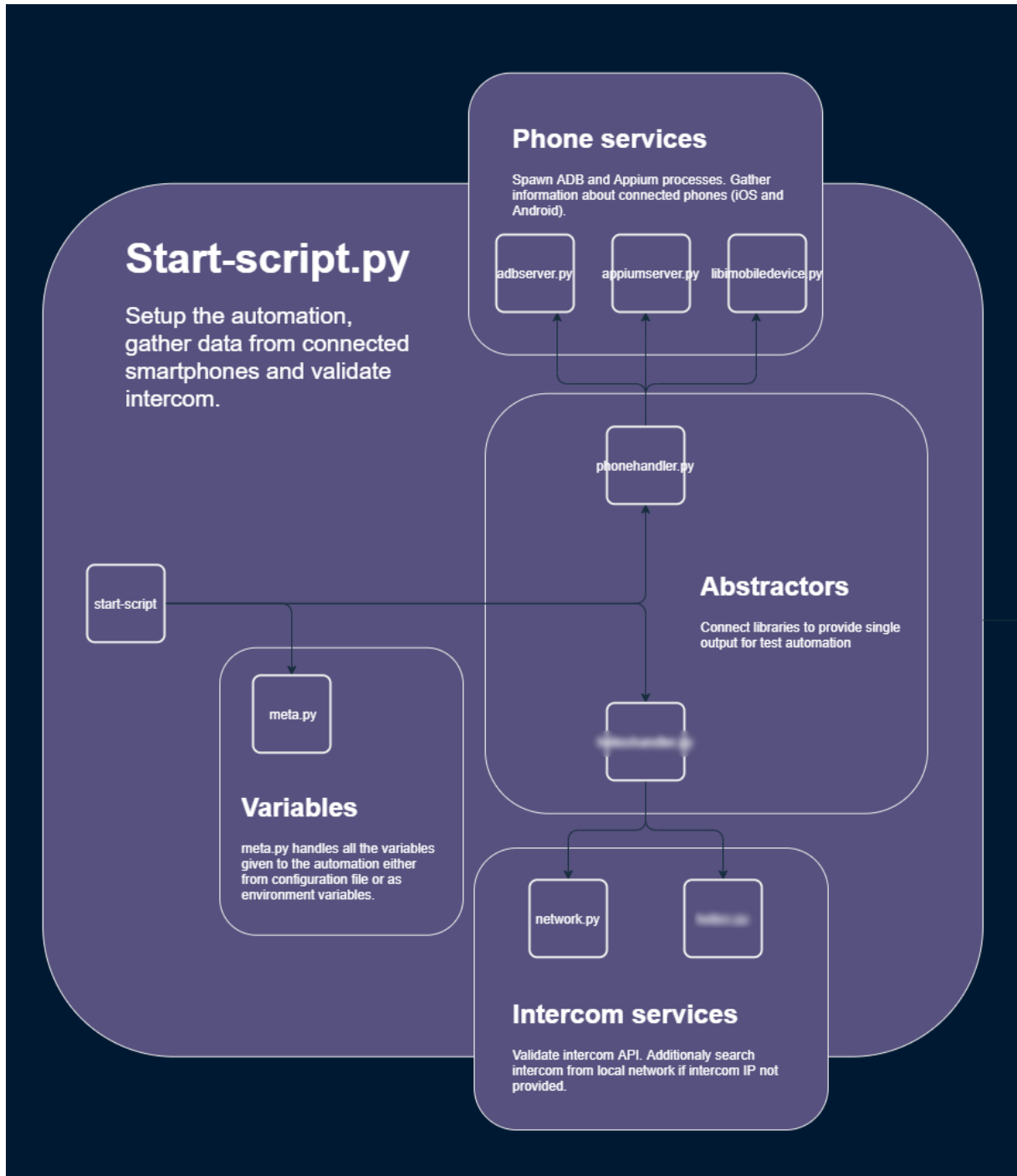
```
*** Test Cases ***
Door Open From Intercom Video Call
  [Setup]   Acquire Lock   videocall
  Loggable Given Visitor Called From Intercom
  Loggable When Tenant Answers Video Call
  Loggable And Tenant Opens Door From Video Call
  Loggable Then Door Will Open
  [Teardown] Release Lock   videocall
```

Picture 12. Video call and door open test case flow.

Because the test automation is written in Gherkin style, each step and the expected outcome of the test is easy to understand. (see chapter 2.1.5). The aesthetics of the high-level keywords suffer a bit from the wrapper keyword “Loggable”, which was developed to time and gather measurements from the keyword it wraps.

## 3.1 Setup

### 3.1.1 Overview



Picture 13. Video call test automation start-script diagram.

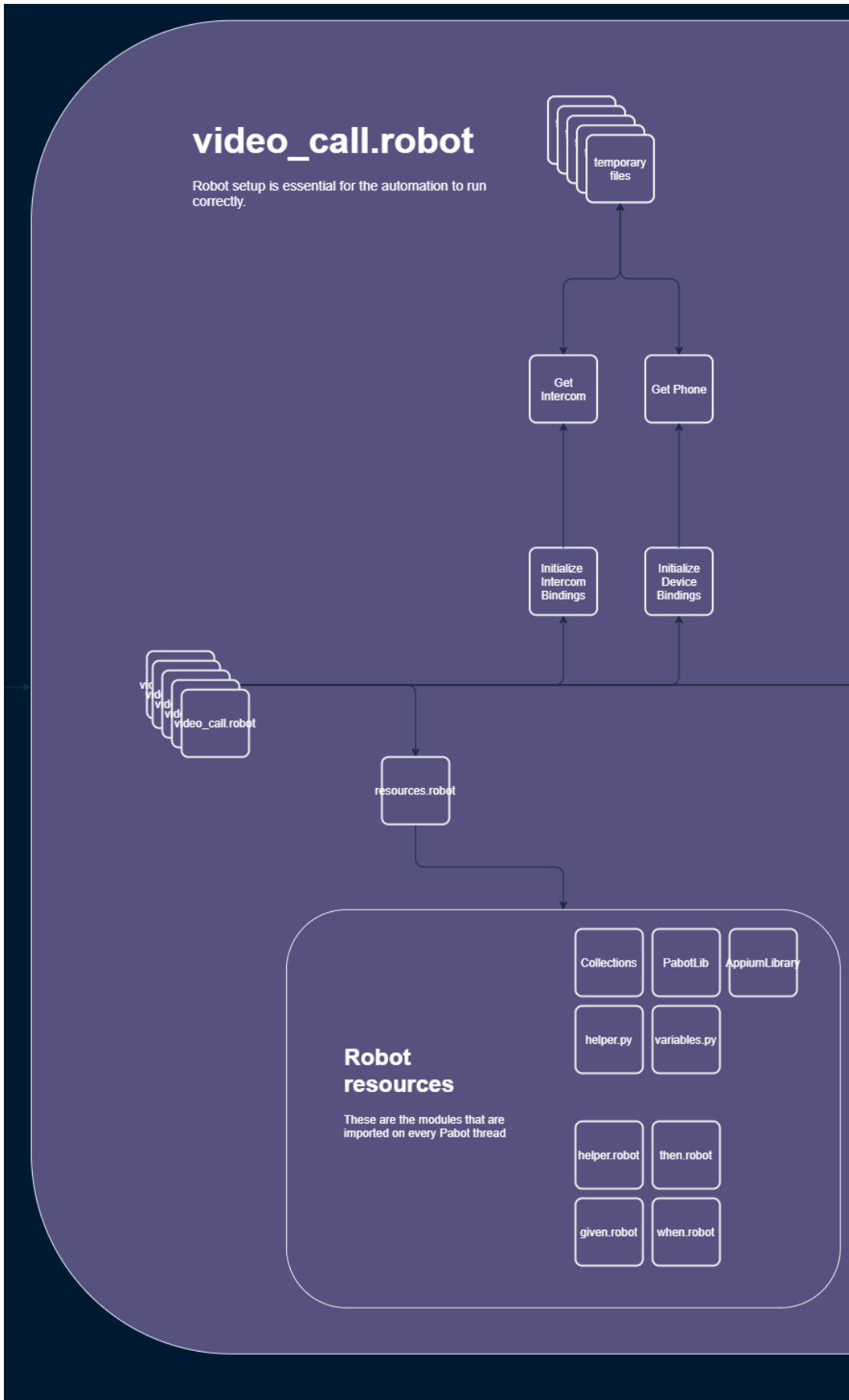
There are multiple layers of setup needed. The first touching point is the Python start-script that forms device and intercom files and a Pabot start command. As seen from the start-

script diagram (Picture 13), the configuration or environment variables are acquired from a single module, the “meta” module. The meta module acquires the configuration variables from a configuration file or from environment variables, depending on what is passed as arguments to the module. Another notable thing is that the functionalities of the “service modules” are abstracted to provide the test automation a simple input and output.

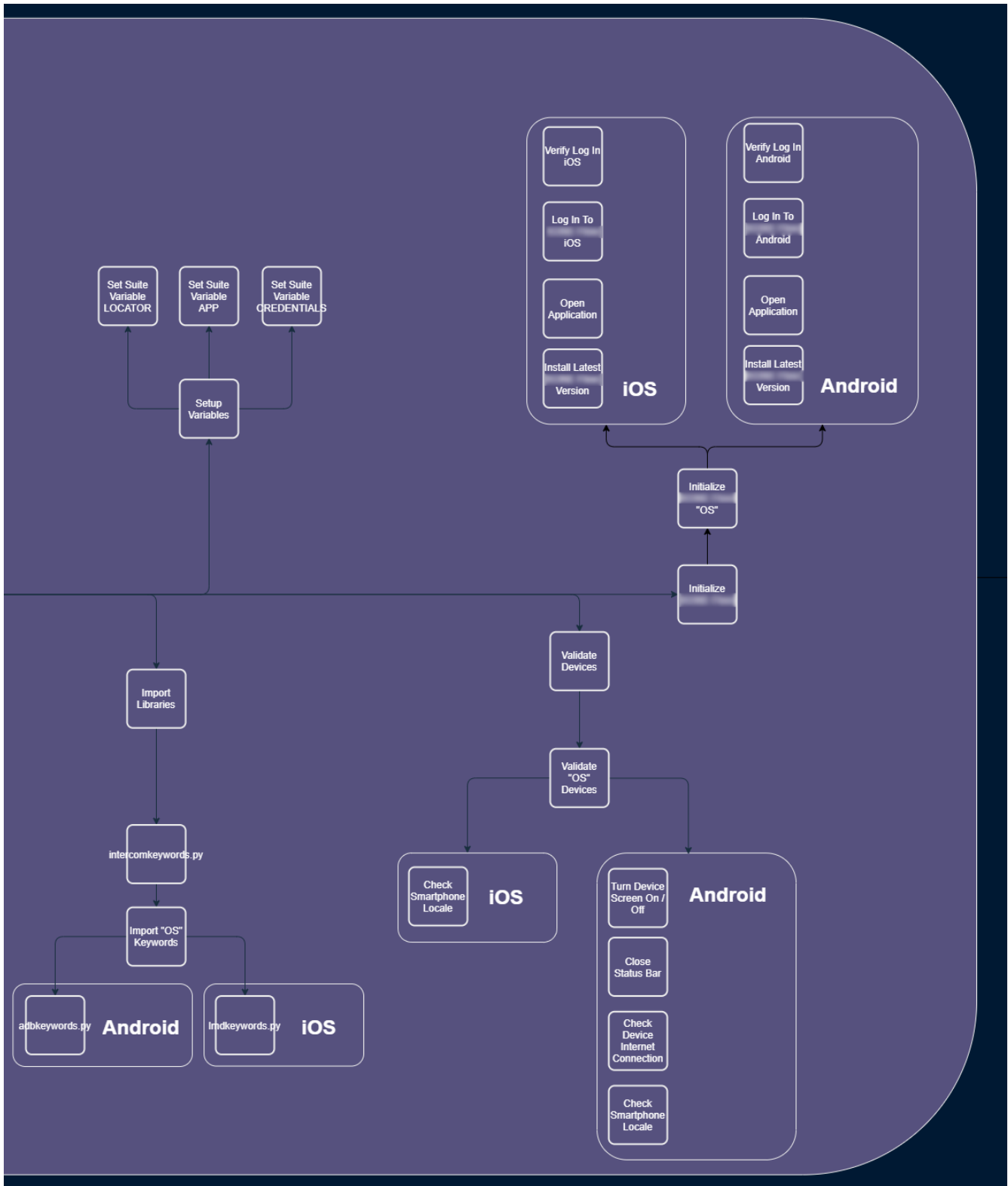
There are two abstractors, the phonehandler and the intercomhandler modules. (Picture 13). The abstractors provide a simple interface specifically tailored for this automation. The service modules are also, in their way, wrappers for the original libraries. For example, the adbserver wrapper provides ADB-related stuff and directly uses the ppadb (pure-python-adb) module or client as a base for its actions. The intention for these service libraries is to act as a more general-purpose so that they could be used in other projects and in a completely different scope, not test automation only. These service libraries gather useful information to the Robot and Pabot processes. The intercom and mobile devices’ data is gathered in these temporary files. (See chapter 3.1.4). The data is used in the Pabot start command (Picture 26) to verify certain things in the automation and for measurements that can be later visualized.

The setting-up continues on the Robot side also, as the created artifacts need to be imported. The setup starts by importing all the needed libraries and resources. (Picture 14, Picture 15). Then by importing the intercom and mobile device JSON files and with them instantiating intercomkeywords and adbkeywords (Android) or lmdkeywords (iOS) libraries (depending on the OS of the mobile device). After this, the automation variables are set (locators, app, and credential variables). They are used for various purposes in the automation. For example, locators are the application locators that Appium uses to interact with the application elements. App variables contain useful information about the application (e.g., on the Android side, this holds app activities). Credential variables are mainly if the automation has to log in to the application. After these steps, the devices are validated, which is more thorough on the Android side. (Picture 15). Because on the iOS side, these actions are either hard to implement or straight impossible to do. (Wyatt8740, 2018). The actions are performed by ADB (adbkeywords library) and libimobiledevice (lmdkeywords library). Last in the setup phase, the application is initialized. This means the application is checked from the mobile device, and possibly the host-provided version is installed if the

application does not exist. (Picture 15). After this, the Appium session is initiated with the “Open Application” keyword. The information that has been gathered from the mobile devices is supplied as arguments. (Picture 16). For the last part of the setup, the test automation logs into the application. Because the argument “noReset” is set to true, the application’s state will not be reset. Meaning the application will stay logged in if the application version has not been updated. For this case, at the start of the “Log In To Application” keyword, it will be checked that if the application is already in the main menu. If the application is in the main menu, the keyword will exit immediately. The main menu validation is by checking that a locator exists, typically a prominent locator in the context. The context, in this case, is the main menu, and the locator element is the navigation bar. If this element is not accessible or visible, it can be assumed that the user has not been logged in, and the automation will proceed with the “Log in to application” keyword normally.



Picture 14. Video call test automation test suite setup diagram part 1.



Picture 15. Video call test automation test suite setup diagram part 2.



```

Initialize ios
  Install Latest ios Version
  ${APPIUM}= Open Application
    ... ${PHONE}[appium][url]
    ... automationName=XCUITest
    ... deviceName=${PHONE}[devicename]
    ... udid=${PHONE}[udid]
    ... wdaLocalPort=${PHONE}[appium][wda_local_port]
    ... alias=${PHONE}[appium][alias]
    ... platformName=${PHONE}[os][name]
    ... platformVersion=${PHONE}[os][version]
    ... app=${PHONE}[application][package_name]
    ... autoGrantPermissions=${true}
    ... newCommandTimeout=${180}
    ... noReset=${true}
    ... usePrebuiltWDA=${HOST}[IOS][USEPREBUILTWDA]
    ... derivedDataPath=${HOST}[IOS][DERIVEDDATAPATH]
  Log In To ios
  Verify Log In ios

```

Picture 16. Video call test automation's Initialize Application "iOS" keyword.

### 3.1.2 Intercom

The intercom handler is an abstractor or wrapper that gives methods like acquiring and validating intercoms and forming a dictionary containing useful information about the intercom. In a case where intercom IP is not provided, the underlying network module will scan the host's private network for devices with the intercom mac prefix. The mac prefix determines the device's vendor. An issue that arises here is that the private network address cannot be assumed because for some router vendors, the private network might be 192.168.1.0, and for some, it might be 192.168.8.0. Another issue is that there can be multiple network interfaces connected to the host machine, meaning there might be multiple private networks that need to be searched. The libraries that do the actual magic are netifaces and Scapy, which give an intuitive but technical interface to work. First, with netifaces, the OS-dependent method to get the connected network interfaces is obtained. On macOS (Darwin) and Linux, this is the same underlying method. (Picture 17).

```

@classmethod
def get_private_networks(cls, addressPool: str = "192.168."):
    """
    Get all private networks found from host, parse returned networks with address pool
    Arguments:
        addressPool: address pool to filter found networks. Defaults to '192.168.' (private network pool).
    Returns:
        networks : list of networks found and parsed
    """
    ifaces = interfaces()
    networks = []

    if platform == "win32":
        interfaceNames = get_windows_if_list()
    elif platform == "linux" or platform == "darwin":
        interfaceNames = get_if_list()
    else:
        raise AssertionError("Operating system is not supported!")

```

Picture 17. Network module - get private networks method part 1.

Then the found network interfaces are iterated over and checked if the network is a type of "AF\_INET", meaning that the address family (AF) is INET or IPv4 family. (Picture 18). This is because the intercoms only provide IPv4 addresses. After validating the address family, a class of the IPv4 network is instantiated. The address pool can be verified from the class that it is in the "16-bit block" private network range, i.e., in the 192.168.0.0 – 192.168.255.255 range. (Rekhter, et al., 1996). When the network is verified as an IPv4 network operating in the "16-bit block" private network territory (192.168.0.0 – 192.168.255.255 range), the data can be gathered from the network. (Picture 18).

```

for iface in ifaces:
    addresses = ifaddresses(iface)
    addresses_inet = ifaddresses(iface)[AF_INET]
    # if ipv4 address family in interfaces
    if AF_INET in addresses:
        for address in ifaddresses(iface)[AF_INET]:
            ipaddr = ip_interface(address["addr"] + "/" + address["netmask"])
            if ipaddr.network.compressed.startswith(addressPool):
                interfaceName = list(
                    filter(lambda x: iface in x["guid"], interfaceNames)
                )[0]["description"]
                networks.append(
                    {
                        "id": iface,
                        "network": ipaddr.network.compressed,
                        "name": interfaceName,
                    }
                )
    logger.debug(networks)
return networks

```

Picture 18. Network module - get private networks method part 2.

A mac broadcast can be sent to the private networks after the data is gathered from them. This is done via the Scapy library, as it allows packet sending at network layer 2 or data link layer. First, the Ethernet frame is formed with a destination of "FF:FF:FF:FF:FF", i.e., mac broadcast address. Then an ARP frame with packet destination of the private network (the network that the "get private networks" method supplied) is formed. After the frames are formed, Scapy's "srp" (send and receive packets at layer 2) method is called. The method is provided with the frames (Ether and ARP) as arguments. (Picture 19). This method does an ARP ping to the network. The ARP ping response is filtered with the mac vendor prefix (the intercom mac vendor) that the method was given. (Picture 20). Lastly, the responses with the correct mac prefix are gathered and returned.

```

@classmethod
def get_devices_by_vendor(
    cls,
    networks: list,
    macvendor: str = "":
):
    """
    Sends mac broadcast to given networks and parses the received packets with given mac vendor prefix.
    Use together with get_private_networks method!
    Arguments:
        networks: List of networks to send mac broadcast to.
        macvendor: mac vendor prefix to parse. Defaults to ""
    Returns:
        network devices: List of device dictionaries, containing useful information about the device.
    """
    broadcast = "FF:FF:FF:FF:FF:FF"
    conf.verb = 0
    for network in networks:
        network_pool = network["network"]
        network_name = network["name"]

        logger.info(
            f"Searching devices from {network} with vendor prefix {macvendor}"
        )
        ether = Ether(dst=broadcast)
        arp = ARP(pdst=network_pool)
        answered, _ = srp(ether / arp, timeout=2, iface=network_name, inter=0.1)
        network_devices = []

```

Picture 19. Network module – get devices by vendor method part 1.

```

    for sent, received in answered:
        ip = received.psrc
        mac = received.hwsrc.upper()
        if mac.startswith(macvendor):
            logger.info(
                f"{ip}: found device with mac address {mac} that matches mac vendor prefix {macvendor}"
            )
            network_devices.append(
                {
                    "ipaddress": ip,
                    "macaddress": mac,
                    "network": network,
                }
            )

    if network_devices:
        logger.debug(
            f"Network devices with correct mac prefix found! {network_devices}"
        )
    else:
        logger.warning(
            f"Did not find any network devices with mac prefix: {macvendor}"
        )

    return network_devices

```

Picture 20. Network module – get devices by vendor method part 2.

The automation will try to interact with the matched network devices' HTTP API to determine the correct device. The intercom's endpoint that provides basic information

about the device is requested. The information provided by the endpoint includes hardware version, software version, serial number, and device name, to name a few. By performing this API call, the API credentials and the device information given are validated. If the API call returns expected information, the automation will continue. Otherwise, the process is repeated n number of times.

### 3.1.3 Mobile devices

The phonehandler module (Picture 13) abstracts the handling of ADB and Appium servers (spawning and killing processes). The processes are handled through the service libraries (adbserver and appiumserver libraries), which are more general-purpose libraries.

Therefore, specific abstractors or wrappers have been built (the same applies to the intercom abstractor library) to serve this particular test automation. First, the ADB and LMD classes are instantiated (from the modules adbserver and libimobiledevice) to get the connected mobile devices from both sides. The device data is gathered from methods provided by the ADB and LMD libraries.

Picture 21, Picture 22). The data is gathered from various services and sources provided by the ADB and LMD clients. Some of the data is directly accessible through the client methods. Some need a bit more digging, e.g., on the iOS side, the correct domain name is needed (com.apple.mobile.battery) to get information about the battery.

```
def get_devices(self):
    ret = []
    android_devices = self.adbserver.get_device_data()
    ios_devices = self.lmd.get_device_data()
    ret.extend(android_devices + ios_devices)
    return ret
```

Picture 21. The get devices method from the phonehandler module.

```

ret.append(
    {
        'devicemodel': info['ProductType'],
        'devicename': info['DeviceName'],
        'udid': info['UniqueDeviceID'],
        'ecid': str(info['UniqueChipID']),
        'os': {
            'name': 'iOS',
            'version': info['ProductVersion'],
        },
        'locale': locale['Language'].split('-')[0],
        'batterylevel': int(battery['BatteryCurrentCapacity']),
        'application': {
            'file': self.ipa,
            'name': self.ipa_data['CFBundleDisplayName'],
            'version': self.ipa_data['CFBundleShortVersionString'],
            'package_name': self.ipa_data['CFBundleIdentifier'],
        },
    },
)

```

Picture 22. Return dictionary from the get device data method in the libimobiledevice module.T

```

ret.append(
    {
        'deviceModel': device.get_properties().get('ro.product.model'),
        'udid': device.serial,
        'os': {
            'name': 'Android',
            'version': device.get_properties().get('ro.build.version.release'),
        },
        'locale': device.get_properties().get('persist.sys.locale').split('-')[0],
        'batterylevel': device.get_battery_level(),
        'application': {
            'file': self._apk.filename,
            'name': self._apk.application,
            'version': self._apk.version_name,
            'package_name': self._apk.package,
            'main_activity': self._appActivity
            or self._apk.get_main_activity(),
        },
    },
)

```

Picture 23. Return dictionary from the get device data method in the adbserver module.

With the gathered data, the Appium servers with one server per mobile device can be spawned. The Appium servers are started and stopped via the Appium client's AppiumService module. (Picture 25). After the server starts, the device dictionary will be extended with the Appium server information. (Picture 24).

```
def start_appium_servers(self):
    ret = []
    for index, device in enumerate(self.devices):
        appium = AppiumServer(
            deviceIndex=index,
            appiumHost=self.appiumHost,
            appiumPort=self.appiumPort,
            systemPort=self.systemPort,
            wdaLocalPort=self.wdaLocalPort,
        )
        appium.start_server()
        ret.append({"devicemodel": device["devicemodel"], "instance": appium})
        device["appium"] = {
            "url": appium.url,
            "appium_port": appium.appiumPort,
            "system_port": appium.systemPort,
            "wda_local_port": appium.wdaLocalPort,
            "alias": str(appium.appiumPort),
        }
    return ret
```

Picture 24. Method start Appium servers from the phonehandler module.

```
def start_server(self):
    """
    Wrapper around AppiumService.start to start Appium server
    """
    self._client.start(
        args=['--address', str(self.appiumHost), '--port', str(self.appiumPort)]
    )
    logger.info(f'Started Appium server on {self.url}')

def stop_server(self):
    """
    Wrapper around AppiumService.stop to stop Appium server
    """
    self._client.stop()
    logger.info(f'Stopped Appium server on {self.url}')
```

Picture 25. Start and stop server methods from the AppiumServer module

### 3.1.4 Temporary files



Picture 26. Video call test automation temporary files overview.

The temporary JSON and argument files are created from the intercom and the mobile devices' gathered data. The JSON files are created with help from the built-in module "json" coupled with Python's built-in I/O handling. The json module enables to encode and decode JSON with ease. The data gathered from the mobile devices and the intercom are Python



dictionaries. This means encoding the data directly to JSON without any alteration is easy. The encoding to a file is via the “json.dump” method, which accepts a file-like object as an output argument. The method can be provided a file-like object with built-in IO handling of Python. (Picture 27).

```
with open(f'resources/temp/{deviceModel}.json', 'w') as deviceFile:
    json.dump(device, deviceFile)
```

Picture 27. Using Python’s built-in IO handling to dump JSON from memory to the file.

```
with open(f'resources/temp/{deviceModel}.args', 'w') as argumentFile:
    argumentFile.write(f'--name {deviceModel}: {self.suitename}')
    argumentFile.write(f'\n--variable TA_NAME:{self.suitename}')
    argumentFile.write(f'\n--variable PHONE:{deviceFilePath}')
    argumentFile.write(f'\n--variable INTERCOM:{intercomFilePath}')
```

Picture 28. Using Python’s built-in IO and writing line by line to the file.

For argument files, the data needs to be inputted line-by-line. (Picture 28).

The argument file, by its name, is a way to provide arguments to Pabot or Robot processes. Complex variables like Python dictionaries cannot be imported directly as arguments or environment variables to the process. (Robot Framework, 2020k). Therefore, the dictionary is converted to a JSON file. Also, strings can only be provided as arguments to the process. Therefore, the absolute path to the JSON file is passed as an argument and not its contents. Another specialty provided in the argument file is that the test suite is renamed with a “— name” flag. The name of the mobile device can be added to the test suite dynamically this way. The argument and JSON files are created iteratively, with one JSON and argument file per device. The argument files also determine the number of Pabot executors because Pabot handles the provided argument files as individual test suites to execute. (See chapter 2.2).

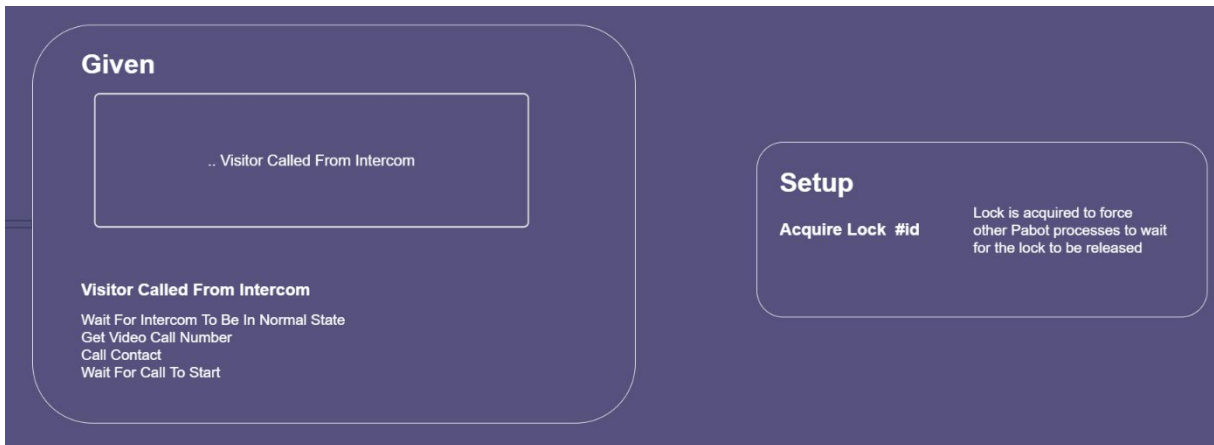
### 3.1.5 Problems

The Python start-script or the Python setup phase was added later in development in an ad hoc manner because the gathering of the mobile device data and the parallel process’ needed to be separated. During the first drafts of the automation, the data gathering process was done on every device (on every Pabot process). This did not only bring

unnecessary overhead to the process, but it was also unstable. The problem was discovered when proper logging was added to the automation. The logs showed that the data gathering action was fired on every process when it should have been fired only once. The start-script was developed suddenly, without any actual planning or specifications. Therefore, it is still a little bit rough around the edges. For example, the name is off, because technically it is not a start-script but a handler or a completely new layer to the automation process. The start-script or the additional Python layer helped solve other problems also. For example, the Pabot start command can be built dynamically now, which was a problem without an answer. The problem was that the number of Pabot processes was fixed, and before, it was fixed to ten processes. Meaning, if there were only three processes or phones under test, the rest of the processes would have been force failed, and on the other side, the maximum was only ten processes.

The implementation of the dynamic start command is still a little messy, as the creation of the JSON and the argument files must be dealt with during the building of the start command. This is because the JSON file paths need to be linked to the argument file. The most robust way to do this is when the instance of the IO stream is available. This way, it is possible to refer to the IO instance file path attribute directly. (Picture 27). The fetching of the intercom provided problems also. The underlying library Scapy is a monolith of a library, allowing communication with a wide range of protocols, such as OBD, CAN, Bluetooth, HTTP, TCP, and TUN/TAP, to name a few. (Biondi & Scapy, 2021). While this seems good on paper, it brings problems like finding good documentation, and examples are simply non-existent or hard to find.

## 3.2 Given



Picture 29. From right to left (reversed), test case setup and “Given” high-level user keywords.

### 3.2.1 Walkthrough

Starting with the “Given Visitor Called From Intercom” keyword, where the test case’s initial context is set: a visitor calls from the intercom to the apartment tenant he is visiting. For this keyword, the intercom and its API must be interacted.

```

*** Keywords ***
Visitor Called From Intercom
    Wait For Intercom To Be In Normal State

    ${number}=    Get Video Call Number
    ...           name=${HOST}[CREDENTIALS][ANDROID][${PHONEMODEL}][name]
    ${SESSION}=   Call Contact
    ...           ${number}
    Set Suite Variable    ${SESSION}
    Wait For Call To Start    ${SESSION}

```

Picture 30. User high-level keyword “Visitor Called From Intercom” from robot file “given.robot”.

Because the test automation is repeated multiple times, the first keyword (the keyword “Wait For Intercom To Be In Normal State”) should be a check to make sure that the intercom is in its so-called normal state, meaning that there is no on-going call.

```

Wait For Intercom To Be In Normal State
Sleep 2s reason=Give time for intercom to catch up
Wait For Call Line To Be Empty timeout=30
Wait For Door To Close timeout=10

```

Picture 31. Keyword “Wait For Intercom To Be In Normal State” from robot file “helper.robot”.

The keyword to check the intercom state wraps two keywords and a forced sleep, which in normal cases is not recommended. If there is a workaround for sleep, that should always be applied instead of sleeping. In this case, the sleep forces the executor to wait a bit before interacting with the API. Without the sleep, the API rate limit will be hit because the requests are made too quickly. Uncertainty is the last thing wanted from an automated test. The keywords are checking that the call line is empty (the keyword “Wait For Call Line to Be Empty”) and that the door state is closed (the keyword “Wait For Door To Close”). These keywords directly communicate with the intercom API.

```

@keyword('Wait For Call Line To Be Empty')
def wait_for_call_line_to_be_empty(self, timeout: str = '10', cycle_sleep: str = '1'):
    timeout = int(timeout)
    cycle_sleep = int(cycle_sleep)
    start_time = time.time()

    while (time.time() - start_time) < timeout:
        response = self.api.get_call_status()
        if not response['result']['sessions']:
            return True
        time.sleep(cycle_sleep)

    raise AssertionError(
        f'Timeout while waiting for call line to be empty! Last status received: {response}'
    )

```

Picture 32. Keyword “Wait For Call Line To Be Empty” from keyword library “intercomkeywords”.

All the “Wait for” keywords in the test automation are built around a timeout and cycle sleep mechanism. While “current time - start time < timeout value” (e.g. in the UNIX Epoch time “1616332413” - “1616332404” < “10”) is true, the loop will continue. If the loop manages to expire before the condition (key-value pair “sessions” is empty) is fulfilled, an assertion error will be raised. Another thing to note is that by default, all Robot scalar

variables (e.g., “\${my scalar variable}”) are strings. This means with all the keyword implementations written in Python, it is a good practice to expect arguments as string type and cast them to integers or floats when needed, or so is done in the “video call and door open” test automation. Inside the loop (Picture 32), the intercom client library’s “get call status” method is called until the time exceeds. The method handles the direct API call and returns the response in Python readable format. (Picture 33).

```
def get_call_status(self, session: str = ''):
    """
    HTTP GET api/call/status?session=arg(session)

    Get call status of given session

    Arguments:
        session: call session ID returned from call_contact request

    Returns:
        response: response as json
    """
    if session:
        endpoint = urljoin(self.api, f'api/call/status?session={session}')
    else:
        endpoint = urljoin(self.api, f'api/call/status')

    response = requests.get(
        endpoint,
        headers=self.headers,
        auth=(self.username, self.password),
        verify=False,
    )
    self._check_header(response)
    self._check_auth(response)
    logger.debug(f'Received from endpoint {endpoint}\n{response.json()}')
    return response.json()
```

Picture 33. Method “get call status” from intercom client library.

A GET request is performed in the intercom API client to the endpoint “api/call/status”. Depending on the argument “session”, the method will retrieve all the call channels or a specific one from the intercom. After getting the response, it is validated, parsed, and returned to the caller. The “Wait For Door To Close” keyword (Picture 34) works on the same premise; an API endpoint is polled until the timeout or the condition matches. In this case,

the return object is mapped and made sure that all the switch states are Boolean “False”, i.e., none of the switches are open. (Picture 34).

```

@keyword('Wait For Door To Close')
def wait_for_door_to_close(self, timeout: str = '10', cycle_sleep: str = '1'):
    timeout = int(timeout)
    cycle_sleep = int(cycle_sleep)
    start_time = time.time()

    while (time.time() - start_time) < timeout:
        response = self.api.get_switch_status()
        if not response['success']:
            raise AssertionError(
                f'Did not receive success response. Response {response}'
            )

        io_status = response['result']
        state_closed = list(map(lambda x: x['active'] == False, io_status['switches']))

        if all(state_closed):
            logger.info('Verified that all IO ports are in closed state!')
            return True

        time.sleep(cycle_sleep)

    raise AssertionError(
        f'Timeout while waiting for door to close! Last status received: {io_status}'
    )

```

Picture 34. Keyword “Wait For Door To Close” from intercom keyword library.

The door or the relay stays open by default from 5 to 10 seconds, meaning the timeout can be safely set to 10 seconds for the keyword. However, the keyword “Wait For Call Line To Be Empty” requires a much longer timeout. If the last executor’s video call does not hang up, or the intercom takes a bit longer to return to the so-called “normal” state, it must be given enough time not to fail the following executors. The length limit of a single video call from the intercom is 5 minutes. This timeout of 5 minutes could be matched to the timeout given to the “Wait For Call Line To Be Empty” keyword so that if a call does not hang up, the automation can safely wait for 5 minutes and not fail the new executor, just because the last call did not hang up correctly. Another thing in the future that can be done is, if the call does not hang up, is that it is closed from the intercom API directly. This has not been yet implemented. When it has been made sure that the intercom is in the normal state, the video call number can be acquired from the intercom directory. The tenant video call

number is fetched with the intercom name shown in the catalog.

```
@keyword('Get Video Call Number')
def get_video_call_number(self, name: str):
    response = self.api.get_contacts()
    assert response['success'], f'Did not receive success from the API request! Response {response}'
    contacts = response['result']

    try:
        contact = list(filter(lambda x: x['name'].startswith(name), contacts['users']))[0]
        video_call_number = list(
            filter(lambda x: not x['peer'].startswith('sip'), contact['callPos'])
        )

        if len(video_call_number) > 1:
            logger.warn('Found multiple video call numbers! Returning the first one!')

        return video_call_number[0]['peer']
    except IndexError as err:
        raise AssertionError(
            f'Could not get a video call number from intercom! Searched for name "{name}" from intercom directory'
        ) from err
```

Picture 35. Keyword “Get Video Call Number” from intercom keyword library.

The “Get Video Call Number” keyword interacts with the intercom API through the “get contacts” method provided in the API client library. The client library performs a POST request to an endpoint “api/dir/query”. The request can be provided a body with different filters, but these are not yet implemented in the test automation. The response is filtered with the argument given to the method. Because the contact can hold multiple audio and video numbers, the numbers are filtered, and the first match is returned. Because the libraries in context are not general-purpose libraries, this kind of functionality is fine (returning the first match only). However, still unexpected behavior like this should be at least logged explicitly. When the video call number has been acquired, the actual call can be initiated with the keyword “Call Contact”.

```
@keyword('Call Contact')
def call_contact(self, number):
    response = self.api.call_contact(number=number)
    assert response['success'], f'Calling contact failed! Intercom did not return a call session.'
    return response['result']['session']
```

Picture 36. Keyword “Call Contact” from intercom keyword library.

The “Call Contact” keyword itself is very simple. The API client library method “call contact” is called with the number gotten earlier from the intercom directory as an argument. The only validation done is that the API response is successful. The keyword returns the session number, which will be used for a further assertion. The session will be stored as a suite variable, meaning it will be available for the suite scope. (See Picture 30). The last step in the

Given keyword is a check that the call started from the intercom side. The keyword “Wait For Call To Start” interacts with the API client’s “get call status” method, which was used in the “Wait For Call Line To Be Empty” keyword also. However, this time the output is validated from the session. The session key-value pair has to be either “connecting” or “ringing”. This way, it can be validated that the call started.

```
@keyword('Wait For Call To Start')
def wait_for_call_to_start(self, session, timeout: str = '10', cycle_sleep: str = '1'):
    accepted_states = ['connecting', 'ringing']

    timeout = int(timeout)
    cycle_sleep = int(cycle_sleep)
    start_time = time.time()

    while (time.time() - start_time) < timeout:
        response = self.api.get_call_status(session=session)
        try:
            if (
                response['result']['sessions'][0]['session'] == session
                and response['result']['sessions'][0]['state'] in accepted_states
            ):
                return True
        except KeyError:
            logger.debug(f'Correct key was not found from response. {response}')

        time.sleep(cycle_sleep)

    raise AssertionError(
        f'Timeout while waiting for call to start! Last status received: {response}'
    )
```

Picture 37. Keyword “Wait For Call To Start” from intercom keyword library.

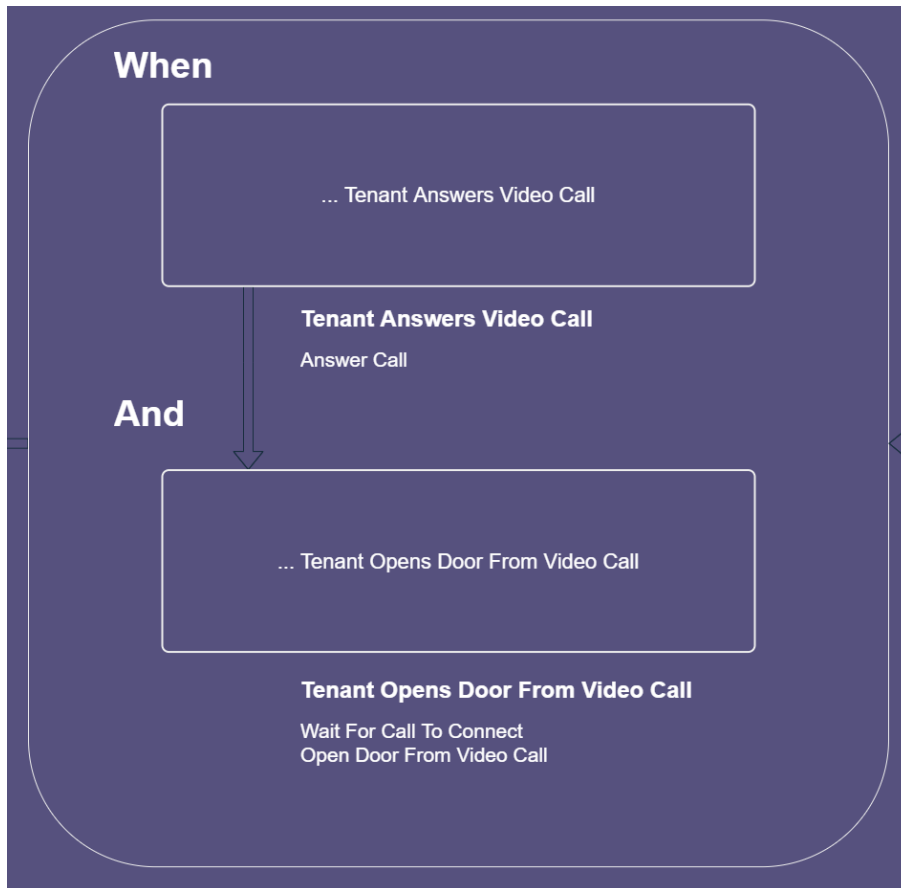
### 3.2.2 Problems

In the first keyword, when the intercom status is validated as “normal” (Picture 31), the use of sleep should be replaced. Without the forced sleep, the tests will occasionally fail because the intercom API rate limit is hit. This makes the API unreachable and causes the test execution to fail. One solution to this would be that after acquiring a lock, sleep will be forced at the very beginning. This does not entirely solve the problem, but the forced sleep placement would be more logical this way.



One pet peeve also is that the high-level keywords should be cleaner, i.e., the “Visitor Called From Intercom” keyword (Picture 30) should not contain variable placements and multi-line keywords.

### 3.3 When



Picture 38. “When” and “And” step’s high-level user keywords.

#### 3.3.1 Walkthrough

The purpose of the “When” step is to describe an action or event. (Cucumber, 2021b). The first keyword is the “Given Tenant Answers Video Call” keyword, which is just a wrapper for another keyword, the “Answer Video Call” keyword provided by the Python library `helper.py`.

```

*** Keywords ***
Tenant Answers Video Call
    Answer Video Call

Tenant Opens Door From Video Call
    Wait For Call To Connect    ${SESSION}
    Open Door From Video Call

```

Picture 39. High-level user keywords from robot file “when.robot”.

The helper python keyword library comes especially handy because it is needed to perform actions depending on the operating system of the mobile device. The “If” statements and all comparison logic in the test automation are done on Python to keep the readability as good as possible.

```

@keyword('Answer Video Call')
def answer_video_call(timeout: str = '2'):
    video_call_status = BuiltIn().run_keyword('Wait For Video Call').lower()
    # give time for UI to update
    time.sleep(int(timeout))

```

Picture 40. Keyword “Answer Video Call” from keyword library “helper.py” part 1.

The keyword “Answer Video Call” calls the “Wait For Video Call” keyword. The “Wait For Video Call” keyword checks the operating system and, on the Android side, the call location

because there are two different locations where the call can land.

```
@keyword("Wait For Video Call")
def wait_for_video_call(timeout: str = "10", cycle_sleep: str = "1"):
    phone_os = BuiltIn().get_variable_value(ROBOT_VARIABLES["operating_system"]).lower()

    if phone_os == "android":
        BuiltIn().run_keyword("Open Status Bar")

        timeout = float(timeout)
        cycle_sleep = float(cycle_sleep)
        start_time = time.time()

        while (time.time() - start_time) < timeout:
            call_in_app = BuiltIn().run_keyword_and_return_status(
                "Should Be In Call"
            )
            if call_in_app:
                BuiltIn().run_keyword("Close Status Bar")
                return "Android: Application"

            call_in_statusbar = BuiltIn().run_keyword_and_return_status(
                "Page Should Contain Element",
                GLOBALVARS.LOCATORS["ANDROID"]["XPATH"]["STATUSBAR"]["ACCEPT_CALL"],
            )
            if call_in_statusbar:
                return "Android: Status Bar"

            time.sleep(cycle_sleep)

        raise AssertionError(f"Timeout while waiting for video call!")

    elif phone_os == "ios":
        # verify that element from the callkit is visible.
        BuiltIn().run_keyword(
            "Wait Until Page Contains Element",
            GLOBALVARS.LOCATORS["IOS"]["XPATH"]["CALLKIT"]["HANG_UP"],
            str(timeout),
        )
        return "iOS: Video Call"
```

Picture 41. Keyword “Wait For Video Call” from keyword library “helper.py”.

The device’s status bar is opened on the Android side, and also, a “Wait For” loop is initiated. In the loop, two keywords are polled: “Application Should Be In Call” and “Page Should Contain Element”. The first keyword checks if the push notification (video call notification) launched a full-screen activity, i.e., video call dialog (Android 9 and lower). Because Android applications are built around activities, this enables to check through Appium the application activity, from which it can be validated that the call activity has started. (Android Developers, 2019). The second keyword, “Page Should Contain Element”, checks that the “accept call” locator is visible in the status bar. Because on Android 10 or higher, the full-

screen activity launch from the push notification requires additional “full-screen intent” permission. (Android Developers, 2021). The application under test is not currently requesting this permission.

Because Appium cannot interact with push notifications if the status bar is not visible, the status bar must be opened at the beginning of the keyword. One hypothesis is that Appium cannot interact with elements, which host application is not in view, this is technically true, but the root cause is unknown. On the iOS side, there is only one way the system will interact with the video call push notification. For iOS, the interaction problem is also present; in Appium, it is impossible to interact with elements, which host application is not in view (or so it is assumed). To this, a swipe action from top to middle needs to be performed. This is because there is no action to activate the status bar directly in iOS (no such action is found yet from the internal iOS API, as the API is not publicly documented). The next step in the automation depends on the state of the application and the push notification state. (Picture 42). All of the paths should lead to the call being answered and with the status bar closed.

```

if video_call_status == "android: application":
    BuiltIn().run_keyword(
        "Page Should Contain Element",
        GLOBALVARS.LOCATORS["ANDROID"]["ID"]["CALLKIT"]["ANSWER_CALL"],
    )
    BuiltIn().run_keyword(
        "Click Element",
        GLOBALVARS.LOCATORS["ANDROID"]["ID"]["CALLKIT"]["ANSWER_CALL"],
    )

elif video_call_status == "android: status bar":
    BuiltIn().run_keyword("Open Status Bar")
    BuiltIn().run_keyword(
        "Page Should Contain Element",
        GLOBALVARS.LOCATORS["ANDROID"]["XPATH"]["STATUSBAR"]["ACCEPT_CALL"],
    )
    BuiltIn().run_keyword(
        "Click Element",
        GLOBALVARS.LOCATORS["ANDROID"]["XPATH"]["STATUSBAR"]["ACCEPT_CALL"],
    )
    BuiltIn().run_keyword("Close Status Bar")

elif video_call_status == "ios: video call":
    # swipe from top to middle to enable fullscreen call.
    BuiltIn().run_keyword("Swipe By Percent", 80, 10, 80, 40)
    BuiltIn().run_keyword(
        "Wait Until Page Contains Element",
        GLOBALVARS.LOCATORS["IOS"]["XPATH"]["CALLKIT"]["ACCEPT_CALL"],
    )
    BuiltIn().run_keyword(
        "Click Element",
        GLOBALVARS.LOCATORS["IOS"]["XPATH"]["CALLKIT"]["ACCEPT_CALL"],
    )
else:
    raise AssertionError(
        f"Video call status not supported! Received: {video_call_status}"
    )

```

Picture 42. Keyword “Answer Video Call” from keyword library “helper.py” part 2.

After the video call has been answered, the door can be opened from the video call dialog, but before this, one more validation needs to be done from the intercom side. (Picture 39). The session state needs to be checked that it has changed to “connected”. The “get call status” method from the “intercomkeywords” library needs to be called with the session

gotten earlier from the “Call Contact” keyword.

```
@keyword('Wait For Call To Connect')
def wait_for_call_to_connect(self, session, timeout: str = '10', cycle_sleep: str = '1'):
    timeout = float(timeout)
    cycle_sleep = float(cycle_sleep)
    start_time = time.time()

    while (time.time() - start_time) < timeout:
        response = self.api.get_call_status(session=session)
        try:
            if (
                response['result']['sessions'][0]['session'] == session
                and response['result']['sessions'][0]['state'] == 'connected'
            ):
                return True
        except KeyError:
            logger.debug(f'Correct key was not found from response. {response}')

    raise AssertionError(f'Call was not connected! Status received: {response}')
```

Picture 43. Keyword “Wait For Call To Connect” from intercom keyword library.

Now the door can be opened from the video call. (Picture 44). The correct locator for the correct operating system is needed, and it needs to be verified that Appium can find the element so that it can be interacted. Because the application under test is a tad bit different on iOS and Android, the locator strategies are different also. In Android, the element ID is at disposal, which is a robust way of locating locators. However, unfortunately for iOS, not all elements have IDs available; thus, the Xpath locator strategy needs to be used instead. Xpaths can sometimes be very flaky, especially if an absolute Xpath is provided, which includes the whole DOM (document object model) structure to the corresponding element. (Guru99, n.d.).

For example, if an open door is opened, it needs to be verified that the door opened, but because the keyword is explicitly doing that in the “then” step, it does not have to be

validated here.

```

@keyword('Open Door From Video Call')
def open_door_from_video_call(timeout: str = '2'):
    open_door_element = None
    phone_os = BuiltIn().get_variable_value(ROBOT_VARIABLES['operating_system']).lower()
    timeout = float(timeout)
    time.sleep(timeout)

    if phone_os == 'android':
        open_door_element = GLOBALVARS.LOCATORS['ANDROID']['ID']['CALLKIT']['OPEN_DOOR']

    elif phone_os == 'ios':
        open_door_element = GLOBALVARS.LOCATORS['IOS']['XPATH']['CALLKIT']['OPEN_DOOR']

    assert (
        open_door_element
    ), 'Operating system of device is not supported! Could not find correct element!'

    BuiltIn().run_keyword(
        'Wait Until Page Contains Element',
        open_door_element,
    )
    BuiltIn().run_keyword('Click Element', open_door_element)

```

Picture 44. Keyword “Open Door From Video Call” from helper keyword library.

### 3.3.2 Problems

Keyword “Answer Video Call” wraps the keyword “Wait For Video Call”, which is built like an internal method or keyword, that should only be used in the context of the “Answer Video Call” keyword. There are few reasons why the keyword should be kept as an “internal” keyword or method, one being that the device is not being returned to its original state in all the cases. On the Android side, if the call is in the status bar, the status bar that was opened at the beginning is not closed. This makes the automation better flowing, not repeatedly opening and closing the status bar. Another reason is that the return values are a bit unorthodox for this keyword particularly. The return values can be fixed to be cleaner, e.g., if the return were a dictionary with the operating system and the video call location key-value pairs, instead of a string containing the information. However, the changing of the return value will not fix the root problem, as one would not expect this kind of return value from a keyword called “Wait For Video Call”; it should be something completely different. The reason why the “Wait For Video Call” keyword should be kept separate (not internal method) is that if it were called as a method, it would not be shown as a keyword in the Robot logs. There would not be any note in the logs that the method was accessed or even

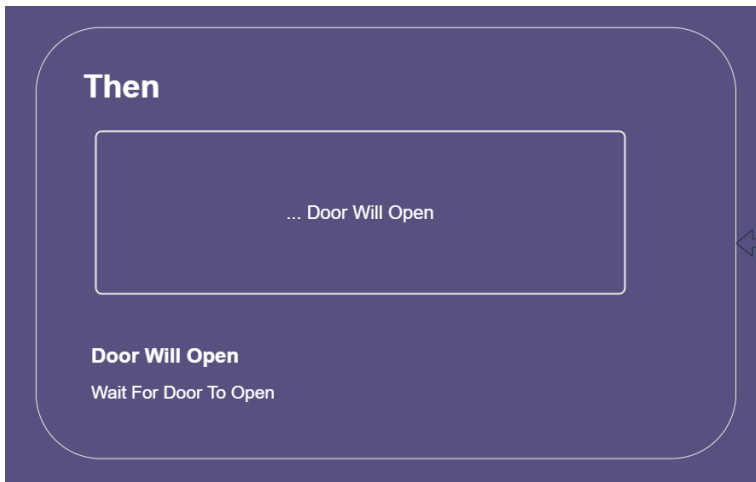
called. Only the things that are logged explicitly inside the method would be logged, but these logs would be shown that they are just inside the nearest keyword, which would be “Answer Video Call” in this case. Another thing to consider is that if the keyword “Wait For Video Call” were merged to the “Answer Video Call”, but should it be merged? It is technically a different entity, so one would argue that it is good to keep the logic modular.

Problems arose with interacting with the video call push notification because the test automation was first used only on Android devices pre-Android 10. It was assumed that the full-screen activity was launched every time. Upon testing with Android 10 devices, the difference caused the automation to fail, causing a need to rework the automation and interact directly with the video call push notification. The problem was brought up while debugging and fetching the new locators for the automation in the Appium desktop inspector. The elements could be seen on the inspector, but they were not interactable. Also, at the time (August 2020) of development, there were no guides on how to interact with the push notification elements. This trivial-sounding problem turned out to be a bigger hassle. However, with testing different combinations, it was found that when the status bar is active, the elements were also interactable in Appium.

One rule the automation is not following with the Gherkin implementation is that it is recommended to have a singular “When” step in a test scenario. (Cucumber, 2021b). The automation has two steps in the “When” step, keywords “When Tenant Answers Video Call” and “And Tenant Opens Door”. Workaround for this would be to switch the “Tenant Answers Video Call” to “Given” step, i.e., “Given Tenant Answers Video Call”, and then to perform the actual action “When Tenant Opens Door” and assert with the keyword “Then Door Will Open” in the “Then” step. This structure will be changed in the future to the proposal above or to something entirely else.



### 3.4 Then



Picture 45. “Then” step’s one high-level user keyword.

#### 3.4.1 Walkthrough

The “Then” step should describe the expected outcome and assert it. (Cucumber, 2021c). This is done in the “video call and door open”, as the high-level keyword tells what is expected (“Door Will Open”) and it is validated.

```
*** Keywords ***
Door Will Open
  Wait For Door To Open
```

Picture 46. High-level user keyword from robot file “then.robot”.

“Then Door Will Open” keyword does one thing, asserts that the intercom has at least one relay in an “ON” state. The check is done through the intercom API, and because the intercom has multiple relays, the automation asserts that there is at least one relay in the “ON” state. This behavior can be relied on because even if the system has multiple triggers to act on the relays, they do still act on the same relay. They act on the same relay because the system is designed this way. The assertion checks that one of the relays is in the “ON” state by mapping the “switches” key in the response gotten from the intercom (switches is a dictionary). The mapping enables to iterate over every switch and assert their key “active” state, which is represented in a Boolean value. If any of these switches are active, the

keyword and the whole test automation will succeed. Note that the video call is still active, and it is not going to be closed in the “Then” step. Returning the system to its normal state happens in the teardown section.

```
@keyword('Wait For Door To Open')
def wait_for_door_to_open(self, timeout: str = '10', cycle_sleep: str = '1'):
    timeout = float(timeout)
    cycle_sleep = float(cycle_sleep)
    start_time = time.time()

    while (time.time() - start_time) < timeout:
        response = self.api.get_switch_status()
        if not response['success']:
            raise AssertionError(
                f'Did not receive success from {self.api} request! Response {response}'
            )

        io_status = response['result']

        state_open = list(map(lambda x: x['active'] == True, io_status['switches']))
        if any(state_open):
            logger.info('Verified that one IO port is in open state!')
            return True

        time.sleep(cycle_sleep)

    raise AssertionError(
        f'Timeout while waiting for door to open! Last status received: {io_status}'
    )
```

Picture 47. Keyword “Wait For Door To Open” from intercom keyword library.

### 3.4.2 Problems

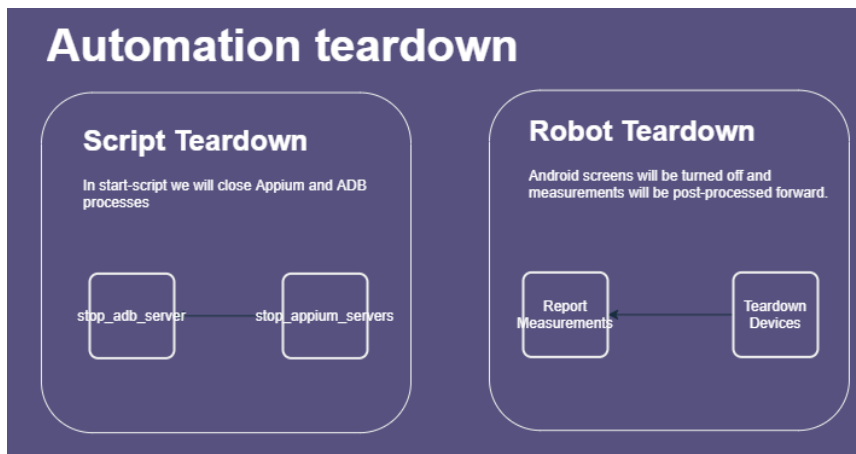
The “Then” step was the least problematic part of the automation, but some problems still arose. In previous drafts of the automation, the video call was closed at the end of the “Then” step, not in the teardown. When a keyword fails after a video call connection was made meant that the automation does not reach the “Then” step. Keyword failing in Robot means that the rest of the test case keywords are not run, except the suite teardown. This caused the video call never to be closed by the automation. Instead, when something failed, the Pabot lock was freed in the suite teardown so that the next executor and the mobile device can be tested. The video call’s maximum time before automatic disconnect is 5 minutes. The automation had to wait 5 minutes but was only waiting for 30 seconds in the first keyword of the “Given” step, where the automation waits for the call line to be empty.

One failure caused most or all of the executors to fail. The problem was caught quickly, and the video call closing keywords were put to the suite teardown.

### 3.5 Teardown



Picture 48. Test case teardown frees Pabot lock.



Picture 49. Robot and start-script teardowns.

#### 3.5.1 Walkthrough

```
Teardown For Suite
Teardown Devices
Report Measurements
```

Picture 50. Teardown keywords from video call and door open suite file.

In the suite teardown, two keywords are called: “Teardown Devices” and “Report Measurements”, the first keyword still interacts with the system under test as the video call

is still presumably open. The call needs to be closed from the mobile device to free the call line for other executors and mobile devices. In the “Teardown Devices” keyword, the “Hang Up Call” keyword is called. It is called before the “OS check” because the keyword does the same check itself.

```
@keyword('Teardown Devices')
def teardown_devices():
    phone_os = BuiltIn().get_variable_value(ROBOT_VARIABLES['operating_system']).lower()
    BuiltIn().run_keyword('Hang Up Call')
    if phone_os == 'android':
        BuiltIn().run_keyword('Turn Device Screen Off')
```

Picture 51. Keyword “Teardown Devices” from helper keyword library.

The keyword is straight forwarded; the “hang up” button element is fetched depending on the mobile device OS. Then the element is verified and interacted. After the interaction, necessary validation is done from the intercom and the mobile device.

```
@keyword('Hang Up Call')
def hang_up_call():
    hang_up_element = None
    phone_os = BuiltIn().get_variable_value(ROBOT_VARIABLES['operating_system']).lower()
    session = BuiltIn().get_variable_value(ROBOT_VARIABLES['session'])

    if phone_os == 'android':
        hang_up_element = GLOBALVARS.LOCATORS['ANDROID']['ID']['CALLKIT']['HANG_UP']

    elif phone_os == 'ios':
        hang_up_element = GLOBALVARS.LOCATORS['IOS']['XPATH']['CALLKIT']['HANG_UP']

    assert (
        hang_up_element
    ), 'Operating system of device is not supported! Could not find correct element!'

    BuiltIn().run_keyword('Wait Until Page Contains Element', hang_up_element)
    BuiltIn().run_keyword('Click Element', hang_up_element)
    BuiltIn().run_keyword('Wait For Call To End', session)
    BuiltIn().run_keyword('Wait For Video Call State', 'disconnected')
```

Picture 52. Keyword “Hang Up Call” from helper keyword library.

The first validation checks from the intercom API that there is no call linked to the session gotten from the “Given” step’s “Call Contact” keyword. This keyword highly resembles the

“Wait For Call Line To Be Empty”, the only difference being that this keyword requires a “session” argument and does the response validation differently. The validation is done by checking that the request-response returned an error with a description of “session not found”. An error is expected because a session argument to the API is explicitly provided. When there are no sessions active (meaning no calls active), the API returns an error.

```
@keyword("Wait For Call To End")
def wait_for_call_to_end(
    self, session, timeout: str = "10", cycle_sleep: str = "1"
):
    timeout = float(timeout)
    cycle_sleep = float(cycle_sleep)
    start_time = time.time()

    while (time.time() - start_time) < timeout:
        response = self.api.get_call_status(session=session)
        try:
            if response["error"]["description"] == "session not found":
                return True
        except KeyError:
            logger.debug(f"Correct key was not found from response. {response}")

        time.sleep(cycle_sleep)

    raise AssertionError(
        f"Timeout while waiting for call to end! Last status received: {response}"
    )
```

Picture 53. Keyword “Wait For Call To End” from intercom keyword library.

The following keyword also resembles the “Wait For Call To End” keyword, but this time, the validation is done via Appium from the mobile device side. The keyword accepts two states: connected and disconnected, as the validation can be done differently depending on the case. The validation is also different for iOS and Android. In Android, if the call state is validated as “disconnected”, the application’s current activity is fetched and checked that it does not match the call activity set in the test automation’s variables. If the status argument were set to “connected”, this would be validated the other way around, i.e., the activity would be expected to be call activity.

On the iOS side, because there are no application activities like in Android that could be retrieved to determine the window or view the mobile device is in, the validation has to be done another way. One good validation to tackle this is to check if an element known to a

particular view is visible. The “open door” element can be used as a validation, whether the mobile device is in the call or not.

```

@keyword('Wait For Application Video Call State')
def wait_for_application_video_call_to_be(status: str, timeout: str = '10', cycle_sleep: str = '1'):
    phone_os = BuiltIn().get_variable_value(ROBOT_VARIABLES['operating_system']).lower()
    status = status.lower()
    timeout = float(timeout)
    cycle_sleep = float(timeout)

    assert (
        status == 'connected' or status == 'disconnected'
    ), 'Keyword only accepts "Connected" or "Disconnected" as status.'

    if phone_os == 'android':
        call_activity = GLOBALVARS.HOST['ANDROID']['APP']['CALL_ACTIVITY']
        start_time = time.time()

        while (time.time() - start_time) < timeout:
            activity = BuiltIn().run_keyword('Get Activity')
            if status == 'connected' and activity in call_activity:
                return True
            elif status == 'disconnected' and activity not in call_activity:
                return True

            time.sleep(cycle_sleep)

        raise AssertionError(
            f'Timed out while waiting for Application to enter call activity {call_activity}\n'
        )

```

Picture 54. Keyword “Wait For Application Video Call State” from helper keyword library part 1.

```

elif phone_os == 'ios':
    if status == 'connected':
        BuiltIn().run_keyword(
            'Wait Until Page Contains Element',
            GLOBALVARS.LOCATORS['IOS']['XPATH']['CALLKIT']['OPEN_DOOR'],
            str(timeout),
        )
    elif status == 'disconnected':
        BuiltIn().run_keyword(
            'Wait Until Page Does Not Contain Element',
            GLOBALVARS.LOCATORS['IOS']['XPATH']['CALLKIT']['OPEN_DOOR'],
            str(timeout),
        )
    return True

```

Picture 55. Keyword “Wait For Application Video Call State” from helper keyword library part 2.

That is the last action done on the iOS side, but for Android, there is one additional step: close the screens. This step would be done with iOS devices also if it were possible. The

“Turn Device Screen Off” keyword wraps the “turn screen off” method from the general-purpose ADB wrapper. The “turn screen off” method sends “input keyevent KEYCODE\_SLEEP” to the device’s shell. The closing of the screen is validated by sending a shell command “dumpsys power | grep mWakefulness” to the device. From the command’s output, the states “asleep” and “dozing” can be mapped. If either of them is present, it can be assumed that the screen was turned off.



```

@keyword('Turn Device Screen Off')
def turn_device_screen_off(self):
    state = self.controller.turn_screen_off()
    if state['success']:
        return True
    else:
        raise AssertionError('Screen was not closed')

```

Picture 56. Keyword “Turn Device Screen Off” from ADB keyword library.

```

def turn_screen_off(self):
    """
    Sends "input keyevent KEYCODE_SLEEP" to device.
    Validate with parsing "device shell dumpsys mWakefulness" status.

    Returns:
        command status: status of the result as dict
    """
    logger.info(f'{self.deviceModel}: Trying to turn screen off')

    accepted_states = ['asleep', 'dozing']
    self.device.shell('input keyevent KEYCODE_SLEEP')
    mWakefulness = self.device.shell('dumpsys power | grep mWakefulness').lower()
    sleeping = list(map(lambda x: x in mWakefulness, accepted_states))

    if any(sleeping):
        response = {
            'deviceModel': self.deviceModel,
            'success': True,
            'result': {
                'action': 'turn screen off',
                'message': 'Turned screen off successfully',
            },
        }
    else:
        response = {
            'deviceModel': self.deviceModel,
            'success': False,
            'result': {
                'action': 'turn screen off',
                'message': 'Could not verify that screen is turned off',
            },
        }

    logger.info(f'{self.deviceModel}: {response}')
    return response

```

Picture 57. Method “Turn screen off” from library “adbbindings”.

After closing the screens on the Android side, the gathered measurements can be posted to the test results database, IBM's Cloudant database. Cloudant is a non-relational database, supporting JSON notation. (IBM Corp., 2020)

```
@keyword('Report Measurements')
def report_measurements():
    send_cloudant = GLOBALVARS.HOST["SEND_CLOUDANT"]
    ta_results = BuiltIn().get_variable_value(ROBOT_VARIABLES['results'], default={})
    ta_name = BuiltIn().get_variable_value(ROBOT_VARIABLES['ta_name'], default='undefined')
    phone_data = BuiltIn().get_variable_value(ROBOT_VARIABLES['phone'])

    phone_data['results'] = ta_results
    phone_data['suiteName'] = ta_name

    if send_cloudant.lower() == 'true':
        logger.info('Posting data to Cloudant!')
        cloudant = CloudantLibrary()
        cloudant.post_data(phone_data)
    else:
        logger.info(phone_data)
```

Picture 58. Keyword “Report Measurements” from helper keyword library.

In the Cloudant wrapper method “post\_data”, the payload or “phone\_data” dictionary is sent through the requests library with a “POST” (HTTP POST) method to the corresponding Cloudant database URL. A scheduled script checks the Cloudant database for new records, retrieves any new records, and posts them to InfluxDB. The data is posted to InfluxDB because InfluxDB's data can be visualized directly in the Grafana graphing system. InfluxDB and Grafana are behind a business network that cannot be accessed from the automation server Jenkins that the “video call and door open” test automation runs on.

### 3.5.2 Problems

Lots of keywords resemble each other by name and logic. The resemblance is not particularly a problem in the teardown, but it does culminate here. The problem is in the general implementation of the keywords. Keywords “Wait For Video Call”, “Wait For Application Video Call State”, “Application Should Not Be In Call”, “Application Should Be In Call”, “Wait For Call Line To Be Busy”, “Wait For Call Line To Be Empty”, “Wait For Call To

End”, “Wait For Call To Start”, “Wait For Call To Connect”, “Call Should Be Connected”, “Call Should Not Be Connected” are most problematic ones if you do not know the context, these keywords are hard to distinguish from each other. Fortunately, these keywords can be modified.

The most uncomplicated keywords are the ones that are validating something by its state, for example, the keywords “Application Should Be In Call” and “Application Should Not Be In Call”. An acceptable rework for this example would be to refactor them as one and name it as “Application Call State Should Be”. The keyword would accept the state it expects as an argument, and possible states would be “Connected” and “Disconnected”. This same rework was already done to the “Wait For Application Video Call State” keyword as it used to be two separate keywords. The initial problem here lies in the planning and execution, as it was agreed to make explicit keywords that provide exact action. Another possible modification to these keywords would be to merge the “Wait For” keywords with the “Should Be”. The only difference between these keywords is that the “Wait For” keyword polls the action for a given amount of time, and the “Should Be” keywords do the action once. This “Should Be” and “Wait For” practice was taken from Selenium and Appium libraries, but the “video call and door open” toolkit will work better with all-around keywords. It is not sure what these keywords will change, but these keywords certainly need a change.

### **3.6 Summary**

The development of video call test automation proves that building test automation requires lots of knowledge in different areas. Knowing the tricks needed to get the device to communicate with the host and having the knowledge to deal with a pile of protocols and libraries is needed. Most of the things just require lots of reading and a bit of perseverance. The development did not come without issues, but the learnings gotten from the test automation will help in the future, and the groundwork for real mobile device automation has been done. This will significantly help when creating other test automation cases that require testing against mobile devices.

The test automation’s infrastructure is durable to scaling to some extent, mainly thanks to Pabot. The automation will remain stable, even if the automation runs + 10 real mobile

devices. The automation will perform the test cases readily. The cap currently comes from the number of mobile devices available at hand, which is not many for this purpose. The host machine's USB controllers will have their limits, but it is better to expand the automation to something entirely else. The most prominent way to scale the system is to take the automation to a cloud service like BrowserStack or Sauce Labs, which can provide the mobile devices to test with, so the automation can truly be scaled.

The automation development gave lots of new learnings and new perspectives for working with Robot Framework and Python. Things like integrating Robot and Python to provide highly readable automation flows via Gherkin, but still being highly flexible and integrable with other systems thanks to Python. This development gave all the ingredients for creating truly powerful test automation. The development process went well, even when considering the hiccups because those are to learn from. Now our team has a new test automation that can be run on software deployments to make sure one of our system's main functionalities keeps working, and also, there is a new great foundation to build more test automation.

## References

- 2N. (n.d.). *2N Wiki - SIP Proxy - How to register 2N® IP Intercom to SIP Proxy server*. Retrieved from 2N Wiki: <https://2nwiki.2n.cz/pages/viewpage.action?pageId=71075280>
- Android Developers. (2019, December 27). *Android Developers - Documentation - Introduction to Activities*. Retrieved from Android Developers: <https://developer.android.com/guide/components/activities/intro-activities>
- Android Developers. (2021, February 24). *Android Developers - Documentation - Create a Notification*. Retrieved from Android Developers: <https://developer.android.com/training/notify-user/build-notification#urgent-message>
- Android Developers. (2021a, February 18). *Android Debug Bridge*. Retrieved from Android Developers: <https://developer.android.com/studio/command-line/adb>
- Android Developers. (2021b, February 18). *Android Debug Bridge - Enable adb debugging on your device*. Retrieved from Android Developers: <https://developer.android.com/studio/command-line/adb#Enabling>
- Appium. (n.d.-a). *Introduction to Appium*. Retrieved from Appium: <http://appium.io/docs/en/about-appium/intro/?lang=fi#appium-design>
- Appium. (n.d.-b). *List of client libraries with Appium server support*. Retrieved from Appium: <http://appium.io/docs/en/about-appium/appium-clients/index.html>
- Appium. (n.d.-c). *Introduction to Appium - Appium concepts*. Retrieved from Appium: <http://appium.io/docs/en/about-appium/intro/?lang=fi#appium-concepts>
- Appium. (n.d.-d). *Create New Session - HTTP API specifications*. Retrieved from Appium: <http://appium.io/docs/en/commands/session/create/#http-api-specifications>
- Appium. (n.d.-e). *Appium desired capabilities*. Retrieved from Appium: <http://appium.io/docs/en/writing-running-appium/caps/>
- AppiumLibrary. (n.d.). *AppiumLibrary keyword documentation (version 1.5.0.7)*. Retrieved from GitHub: <https://serhatbolsu.github.io/robotframework-appiumlibrary/AppiumLibrary.html>
- Apple Inc. (2018, April 6). *Apple Developer - Property list documentation*. Retrieved from Apple Developer: [https://developer.apple.com/library/archive/documentation/General/Conceptual/D](https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/PropertyList.html)  
[evPedia-CocoaCore/PropertyList.html](https://developer.apple.com/library/archive/documentation/General/Conceptual/DDevPedia-CocoaCore/PropertyList.html)

- Arieli, G. (n.d.). *What is Appium Server?* Retrieved from Experitest:  
<https://experitest.com/appium-blog/what-is-appium-server/>
- Awesome Robot Framework. (2021, February 22). *Read me*. Retrieved from GitHub:  
<https://github.com/fkromer/awesome-robotframework/blob/master/README.md>
- Biondi, P., & Scapy. (2021, February 26). *Scapy documentation (version 2.4.4.dev0)*. Retrieved from Scapy: <https://scapy.readthedocs.io/en/latest/>
- Chatterjee, A. (2016, June 28). *Suggest any workaround to Enable Developer Options on iOS (preferably 9 and higher) without use of MacOS or Xcode?* Retrieved from Stack Overflow: <https://stackoverflow.com/a/38078768/14762853>
- Cucumber. (2021a, February 15). *Gherkin Reference - Steps*. Retrieved from Cucumber: <https://cucumber.io/docs/gherkin/reference/#steps>
- Cucumber. (2021b, February 15). *Gherkin Reference - When*. Retrieved from Cucumber: <https://cucumber.io/docs/gherkin/reference/#when>
- Cucumber. (2021c, February 15). *Gherkin Reference - Then*. Retrieved from Cucumber: <https://cucumber.io/docs/gherkin/reference/#then>
- Gabilondo, J. (2018, December 10). *Understanding usbmux and the iOS lockdown service*. Retrieved from Medium:  
[https://medium.com/@jon.gabilondo.angulo\\_7635/understanding-usbmux-and-the-ios-lockdown-service-7f2a1dfd07ae](https://medium.com/@jon.gabilondo.angulo_7635/understanding-usbmux-and-the-ios-lockdown-service-7f2a1dfd07ae)
- Geiszl, A. (2016, August 30). *Stack Overflow - Cross-compiling for Windows Python error*. Retrieved from Stack Overflow: <https://stackoverflow.com/q/39225184>
- Guru99. (n.d.). *Guru99 - XPath in Selenium WebDriver Tutorial: How to Find XPath?* Retrieved from Guru99: <https://www.guru99.com/xpath-selenium.html#3>
- IBM Corp. (2020, December 22). *IBM Cloud documentation - JavaScript Object Notation (JSON)*. Retrieved from IBM Cloud:  
<https://cloud.ibm.com/docs/Cloudant?topic=Cloudant-json>
- IDG Communications, Inc. (2004, May 11). *Network World - What is SIP?* Retrieved from Network World: <https://www.networkworld.com/article/2332980/lan-wan-what-is-sip.html>
- Khalfallah, H. B. (2020, December 24). *How Can We Measure Our Software's Modularity and Dependencies?* Retrieved from Better Programming:  
<https://betterprogramming.pub/inside-software-modularity-and-related-metrics-2e5af2b447dc>

- King, K. (n.d.). *Twilio - What Is Push Notification?* Retrieved from Twilio:  
<https://www.twilio.com/docs/glossary/what-is-push-notification#how-are-push-notifications-added-to-an-application>
- Klärck, P. (2021, March 11). *PyPi - Robot Framework - History*. Retrieved from PyPi:  
<https://pypi.org/project/robotframework/>
- Laukkanen, P. (2006, February 24). *Data-Driven and Keyword-Driven Test Automation Frameworks*. Retrieved from Eliga: <http://eliga.fi/writings.html>
- Libimobiledevice. (2020a, September 27). *Libimobiledevice - Features*. Retrieved from Libimobiledevice: <https://libimobiledevice.org/#features>
- Libimobiledevice. (2020b, September 27). *Libimobiledevice - Get started*. Retrieved from Libimobiledevice: <https://libimobiledevice.org/#get-started>
- McCraw, C. (2020, July 23). *What is a VoIP Gateway?* Retrieved from GetVoIP:  
<https://getvoip.com/library/what-is-a-voip-gateway/>
- Minakov, T. (2018, November 17). *What is a keyword in Robot Framework?* Retrieved from Stack Overflow: <https://stackoverflow.com/a/53350101>
- Mozilla Developer Network. (2021, February 19). *Mozilla MDN - HTTP authentication - Security of basic authentication*. Retrieved from Mozilla Developers:  
[https://developer.mozilla.org/en-US/docs/Web/HTTP/Authentication#security\\_of\\_basic\\_authentication](https://developer.mozilla.org/en-US/docs/Web/HTTP/Authentication#security_of_basic_authentication)
- Pabot. (2020a, August 31). *Pabot readme file (version 1.10.1)*. Retrieved from GitHub:  
<https://github.com/mkorpela/pabot#basic-use>
- Pabot. (2020b, August 31). *Pabot readme file (version 1.10.1)*. Retrieved from GitHub:  
<https://github.com/mkorpela/pabot#command-line-options>
- Pabot. (n.d.-a). *PabotLib documentation (version 0.67)*. Retrieved from Pabot:  
<https://pabot.org/PabotLib.html>
- pure-python-adb. (2020, August 5). *pure-python-adb readme file - examples (version 0.30-dev)*. Retrieved from GitHub: <https://github.com/Swind/pure-python-adb/blob/master/README.rst#examples>
- Ramella, B. (2021, January 5). *What is a SIP URI?* Retrieved from GetVoIP:  
<https://getvoip.com/library/what-is-a-sip-uri/>
- Rekhter, Y., Cisco Systems, I., Moskowitz, B., Corp., C., Karrenberg, D., NCC, R., . . . Silicon Graphics, I. (1996, February). *RFC1918, chapter 3. Private Address Space*. Retrieved from Internet Engineering Task Force: <https://tools.ietf.org/html/rfc1918>

Robot Framework. (2020a, September 1). *Source code for robot.api.deco (version 3.2.2)*.

Retrieved from Robot Framework API documentation: [https://robot-framework.readthedocs.io/en/v3.2.2/\\_modules/robot/api/deco.html](https://robot-framework.readthedocs.io/en/v3.2.2/_modules/robot/api/deco.html)

Robot Framework. (2020b, September 1). *Robot Framework user guide (version 3.2.2)*.

Retrieved from Robot Framework:

<https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#limiting-public-methods-becoming-keywords>

Robot Framework. (2020c, September 1). *Robot Framework user guide (version 3.2.2)*.

Retrieved from Robot Framework:

<https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#keyword-names>

Robot Framework. (2020d, December 23). *Robot Framework output files (version 3.2.2)*.

Retrieved from GitHub:

<https://github.com/robotframework/robotframework/blob/master/doc/userguide/src/ExecutingTestCases/OutputFiles.rst#output-file>

Robot Framework. (2020e, September 1). *Robot Framework user guide (version 3.2.2)*.

Retrieved from Robot Framework:

<https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#using-test-libraries>

Robot Framework. (2020f, September 1). *Robot Framework user guide (version 3.2.2)*.

Retrieved from Robot Framework:

<https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#log-levels>

Robot Framework. (2020g, September 1). *Robot Framework user guide (version 3.2.2)*.

Retrieved from Robot Framework:

<https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variable-files>

Robot Framework. (2020h, September 1). *Robot Framework user guide (version 3.2.2)*.

Retrieved from Robot Framework:

<https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#resource-files>

Robot Framework. (2020i, September 1). *Robot Framework user guide (version 3.2.2)*.

Retrieved from Robot Framework:



<http://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#behavior-driven-style>

Robot Framework. (2020j, September 1). *Robot Framework user guide (version 3.2.2)*.

Retrieved from Robot Framework:

<https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#argument-files>

Robot Framework. (2020k, September 1). *Robot Framework user guide - Setting variables in command line*. Retrieved from Robot Framework:

<http://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#setting-variables-in-command-line>

Robot Framework. (n.d.-a). *Introduction*. Retrieved from Robot Framework:

<https://robotframework.org/#introduction>

Robot Framework. (n.d.-b). *Examples*. Retrieved from Robot Framework:

<https://robotframework.org/#examples>

SeleniumLibrary. (n.d.). *Source code for BrowserManagementKeywords (version 5.1.0)*.

Retrieved from GitHub:

<https://github.com/robotframework/SeleniumLibrary/blob/master/src/SeleniumLibrary/keywords/browsermanagement.py>

Software Testing Help. (2021, February 18). *Software Testing Help - What Is Regression Testing? Definition, Tools, Method, And Example*. Retrieved from Software Testing Help:

<https://www.softwaretestinghelp.com/regression-testing-tools-and-methods/>

Soluno. (2019, December 10). *DTMF*. Retrieved from Soluno:

<https://www.soluno.com/glossary/dtmf/>

W3C. (2020, August 24). *W3C Working Draft - WebDriver - Locator strategies*. Retrieved from

<https://www.w3.org/TR/webdriver/#locator-strategies>

Vincent. (2014, June 11). *Stack Overflow - How to get python-dev for windows?* Retrieved from Stack Overflow: <https://stackoverflow.com/a/24163212>

Wyatt8740. (2018, February 10). *Stack Overflow - debug bridge for iPhone / shell command prompt*. Retrieved from Stack Overflow: <https://stackoverflow.com/a/48716848>

