How to build e2e test automation for mobile apps

23. MAY 2024
MIKHAIL
MIROSHNICHENKO

Wolt

About myself

Mikhail Miroshnichenko

QA Competence Lead / Mobile Test Automation Engineer at Wolt

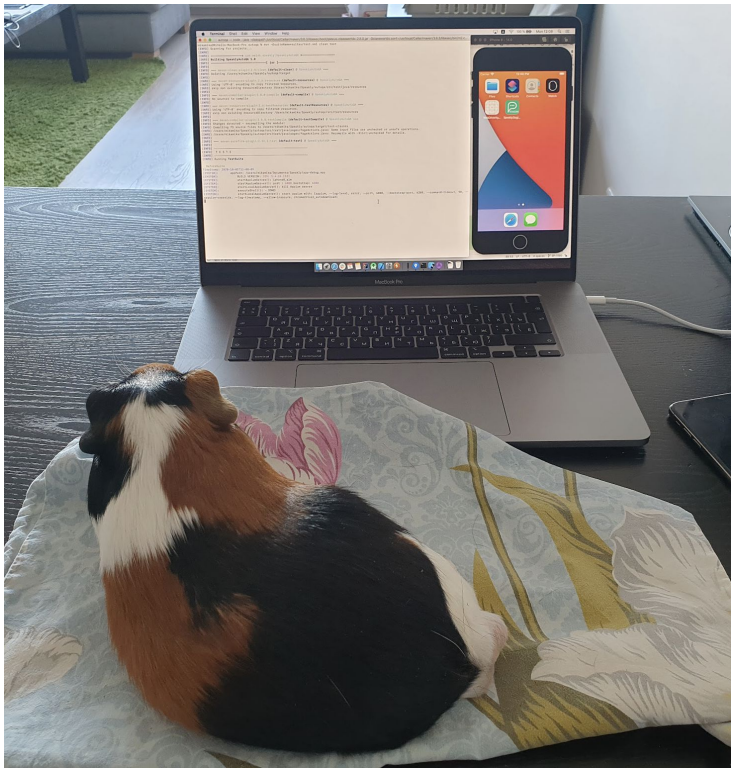
24 years in IT
18 years in QA
11 years in Mobile development
6 years in Test automation

Email (personal): hikari.no.mikem@gmail.com

Email (work): mikhail.miroshnichenko@wolt.com

LinkedIn: linkedin.com/in/miroshnichenkomichael

Dedicated to Murzik the Guinea Pig



Murzik is accompanying me on my way with Test automation for many years.

These days he is fighting with the serious illness.

What will you learn today

How to build e2e test automation for mobile apps: from choosing tools to getting test run reports for a new PR in the Slack channel

... with some examples from the **Wolt Merchant App**

Presentation plan

1. Brief idea of the test automation
2. Setup project in IDE
3. Create automated tests
4. Run test on a local machine
5. Integrate with CI and run in the cloud

Brief idea of the test automation

Wolt

Why to automate test

- Regression testing
- Constant quality control over time
- Check what is difficult or impossible to check manually
- Replace repetitive test
- **Fast feedback** about the quality of a new build

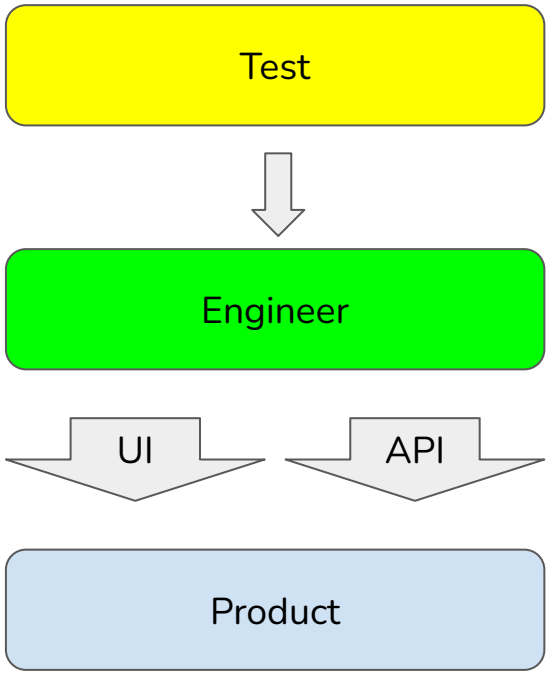
“Automation answers a question which we already know the answer to”

What do we have at the beginning?

- Project source code in GitHub

- Test cases in the Test Management System (used to decide what to automate and how)

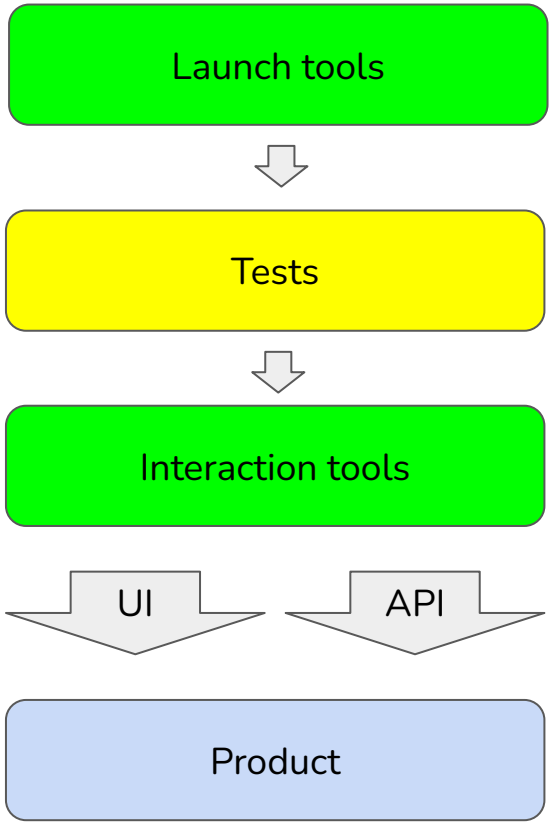
Manual testing scheme



- API (Application Programming Interface): way to interaction between different projects and services

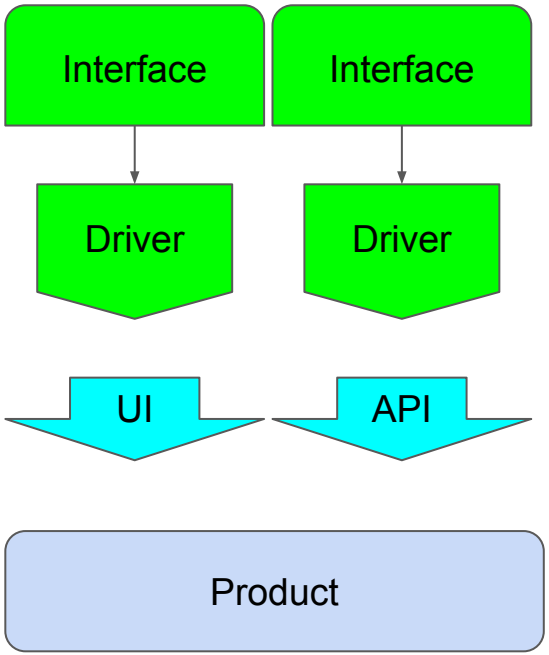
- UI: Graphical User Interface to interact with users and automated tests. *Today we will focus on this one.*

Automated testing scheme



Tools to launch the app and interact with it are called Test automation stack sometimes

Interaction tools

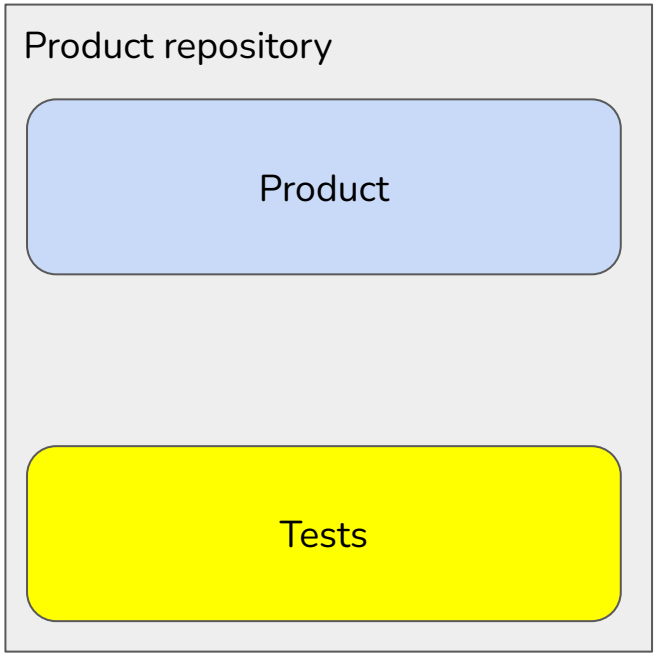


Driver: a low-level tool to interact directly with a specific application interface: UI or API

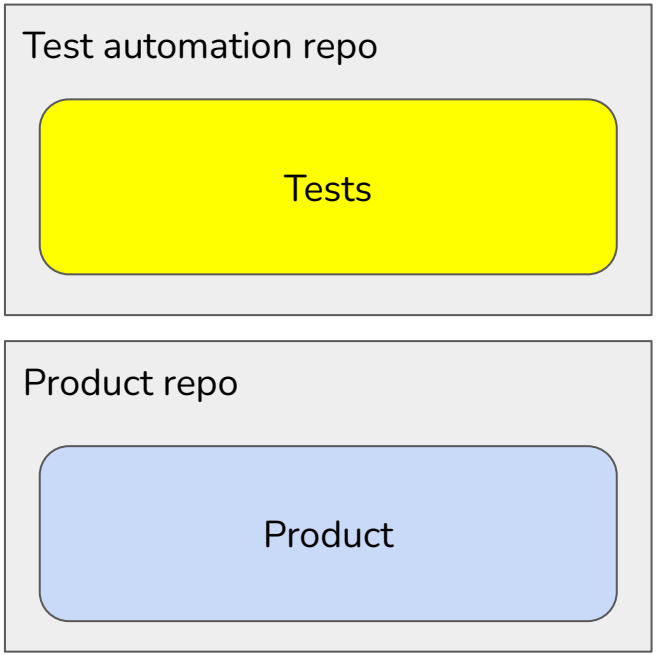
High-level Interface: makes using driver in your tests easier

Automated tests: where to store them?

Inside project



As a standalone project



Automation tools: Two approaches

Native test automation: **one** tool per **one** platform

- Product code and Test project is the same project
- Uses platform built-in drivers directly
- iOS: XCUITest
- Android: UIAutomator, Espresso

Cross-platform test automation: **one** tool for **multiple** platforms

- Tests are in a separate project, which doesn't depend on the Product project
- Using built-in platform drivers through interface
- Mobile: Appium; Web: Selenium
- All-in-one frameworks: Maestro, Robot etc...

Which way to select should be decided by the team: QAs and developers.

Native TA: White box

Pros:

- Can use application components and resources
- Full access to screen elements and their properties
- Much easier to modify app behavior (like mocking backend responses)

Cons:

- Works with the same project it was developed for
- Usually, one should use the same programming language as the application: this is a plus when developers are creating the tests

Cross-platform TA: Black box

Pros:

- One TA project for everything: iOS, Android, Web
- Flexible with choosing programming language
- Project is very small: only code, no heavy resources

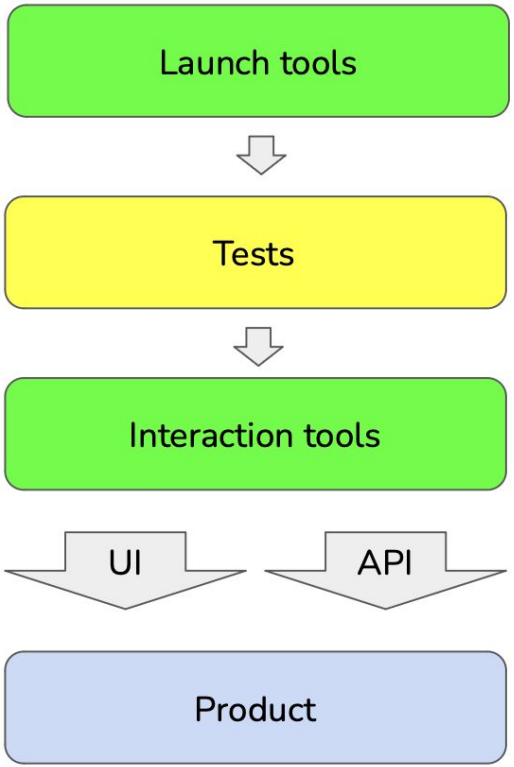
Cons:

- Less flexible access to screen elements and their properties
- More difficult to implement: initial setup is more complex
- Might be more difficult to unboard developers to contribute to another project

Components of the cross-platform TA project

- **Framework**: Launch and Interaction tools are here
  - Managing test runs
  - Configuring environment
  - Collecting test results and composing test reports
  - Interacting with other apps and services
  - Provide basic functions and methods to work with the app and its interface

- **Tests**
  - iOS application
  - Android application
  - ...



Part 2: Project with automated tests

Wolt

Where to start: IDE

IntelliJ Idea: https://www.jetbrains.com/idea/

- Free version for non-commercial development
- Works on Windows and macOS
- Supports various programming languages in the free version: Kotlin, Java
- Has a lot of useful functions to speed up writing the code
- Paid version supports more languages and has more tools

But you can use your favourite IDE as well =)

Maven: project management system

Used for:

- Load all needed components and libraries
- Set up testing parameters
- Execute test suites
- Collect basic data about test results

Configured by: **pom.xml** file

Web-site: https://maven.apache.org

Install (macOS): **brew install maven**

Maven: pom.xml example

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.wolt.merchant.autoqa</groupId>
    <artifactId>WoltAutoQA</artifactId>
    <version>1.0</version>

    <properties>
        <aspectj.version>1.9.2</aspectj.version>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <suiteName>suites/emptySuite.xml</suiteName>
        <env>staging</env> <!-- prod / staging -->
        <timestamp>${maven.build.timestamp}</timestamp>
        <results.directory>target/allure-results</results.directory>
        <kotlin.version>1.5.10</kotlin.version>
        <kotlin.compiler.incremental>true</kotlin.compiler.incremental>
    </properties>
```

Maven: List of used libraries

```
<!-- https://mvnrepository.com/artifact/io.appium/java-client -->
<dependency>
    <groupId>io.appium</groupId>
    <artifactId>java-client</artifactId>
    <version>7.3.0</version>
</dependency>

<!-- https://mvnrepository.com/artifact/org.projectlombok/lombok -->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.10</version>
    <scope>provided</scope>
</dependency>

<!-- https://mvnrepository.com/artifact/org.testng/testng -->
<dependency>
    <groupId>org.testng</groupId>
    <artifactId>testng</artifactId>
    <version>6.14.3</version>
    <scope>test</scope>
</dependency>
```

Maven: Adding library to the project

Web-site: https://mvnrepository.com - use it for searching for libraries

Library import code

```
<!-- https://mvnrepository.com/artifact/io.appium/java-client -->
<dependency>
    <groupId>io.appium</groupId>
    <artifactId>java-client</artifactId>
    <version>7.3.0</version>
</dependency>
```

Maven: How to add tests?

Maven can import all needed libraries to your project and build them.

But how to tell Maven that we have tests and we are going to test the app, not to build it?

Time to introduce **TestNG**!

TestNG: Test management tool

Allows to (by using suite.xml files)

- Configure test parameters
- Split test parameters from build parameters
- Extend Maven tools for configuring test suites
- Store test run results

Defines:

- Actions for setup/teardown of full test suite and individual tests
- Which classes and methods are executable tests

TestNG: Test suite configuration file

```
<!DOCTYPE suite SYSTEM "http://beust.com/testng/testng-1.0.dtd" >
<suite name="Mobile UI Tests">
    <parameter name="product" value="merchantApp"/>
    <parameter name="deviceName" value="ipad_sim"/>
    <parameter name="startType" value="fastReset"/>
    <parameter name="env" value="staging"/>
    <parameter name="runner" value="local"/>
    <parameter name="launchParameters" value="-skipIntros"/>
    <test name="TEST suite">
        <groups>
            <run>
                <include name="ios"/> <!-- android / ios -->
            </run>
        </groups>
        <classes>
            <class name="tests.simpleChecks.SignInTest"/>
        </classes>
    </test>
</suite>
```

TestNG: Test parameters

```
<!DOCTYPE suite SYSTEM "http://beust.com/testng/testng-1.0.dtd" >
<suite name="Mobile UI Tests">
    <parameter name="product" value="merchantApp"/>
    <parameter name="deviceName" value="ipad_sim"/>
    <parameter name="startType" value="fastReset"/>
    <parameter name="env" value="staging"/>
    <parameter name="runner" value="local"/>
    <parameter name="launchParameters" value="-skipIntros"/>
    <test name="TEST suite">
        <groups>
            <run>
                <include name="ios"/> <!-- android / ios -->
            </run>
        </groups>
        <classes>
            <class name="tests.simpleChecks.SignInTest"/>
        </classes>
    </test>
</suite>
```

TestNG: List of sub suites and tests

```
<!DOCTYPE suite SYSTEM "http://beust.com/testng/testng-1.0.dtd" >
<suite name="Mobile UI Tests">
    <parameter name="product" value="merchantApp"/>
    <parameter name="deviceName" value="ipad_sim"/>
    <parameter name="startType" value="fastReset"/>
    <parameter name="env" value="staging"/>
    <parameter name="runner" value="local"/>
    <parameter name="launchParameters" value="-skipIntros"/>
    <test name="TEST suite">
        <groups>
            <run>
                <include name="ios"/> <!-- android / ios -->
            </run>
        </groups>
        <classes>
            <class name="tests.simpleChecks.SignInTest"/>
        </classes>
    </test>
</suite>
```

Test automation driver: Appium

Appium:

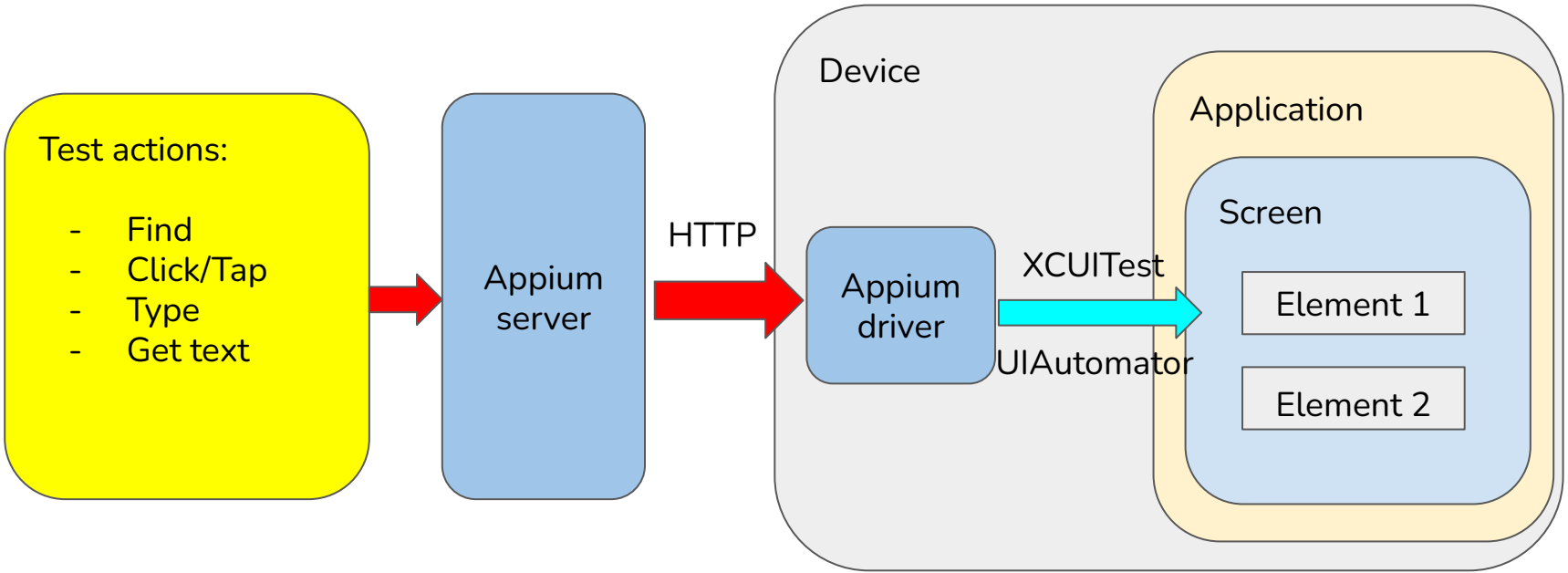
- is a server: we connect to it to send commands to a device
- has drivers for each of supported platforms
- works through HTTPs protocol

Web-site: https://appium.io

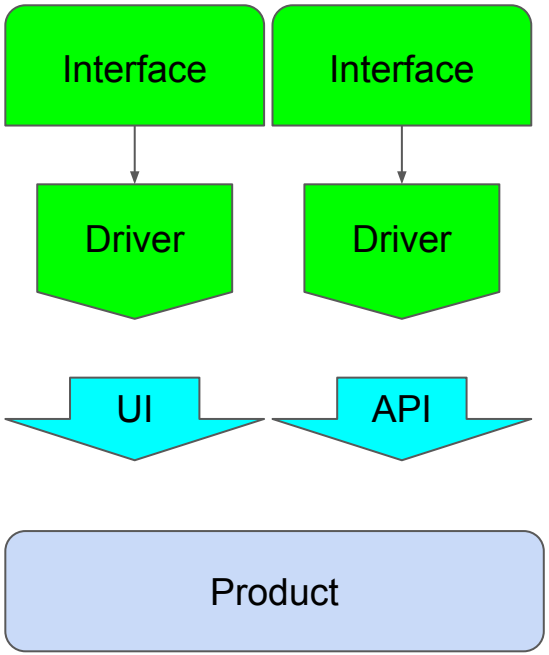
Contains:

- server with drivers
- Appium Inspector client

Appium: Command flow



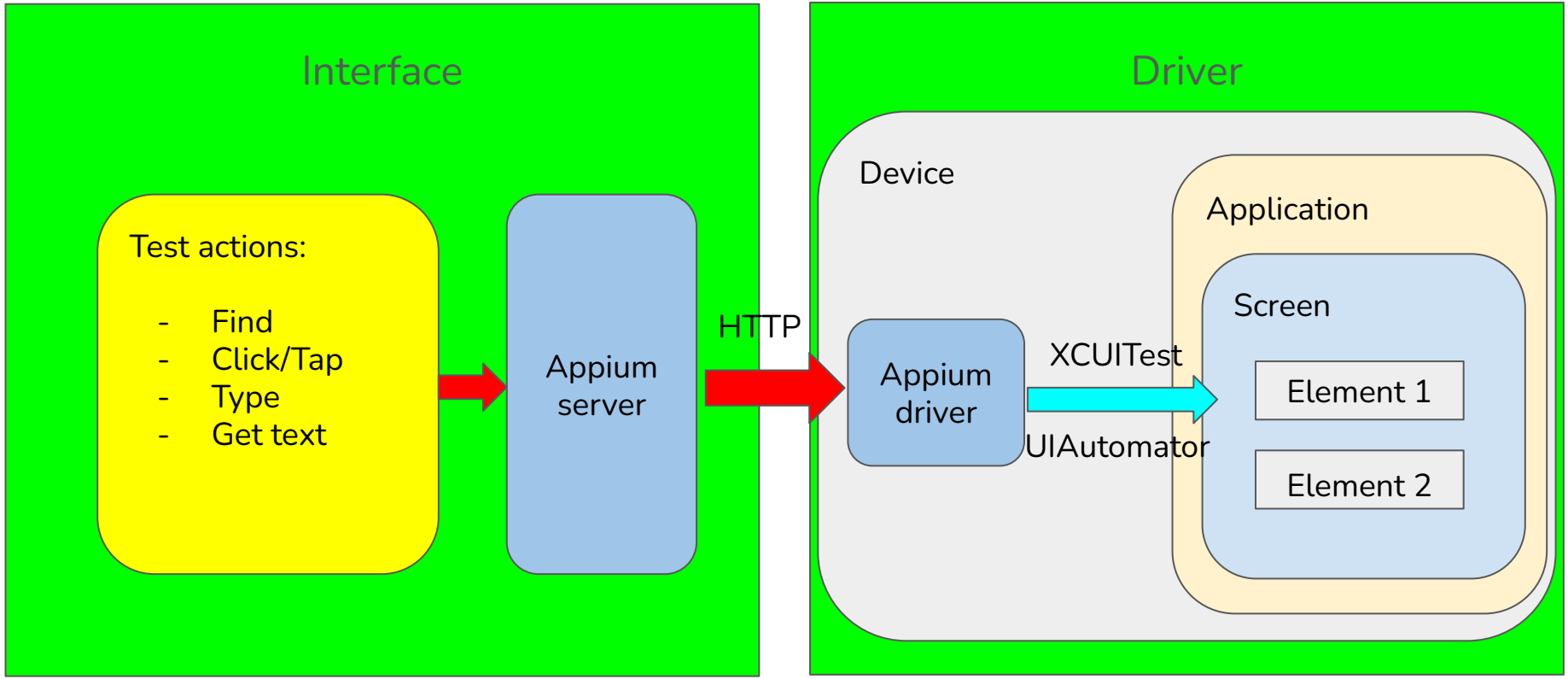
Remember Interaction tools schema?



Driver: a low-level tool to interact directly with a specific application interface: UI or API

High-level Interface: makes using driver in your tests easier

Appium: Command flow



Appium Inspector

The best friend of a person creating the tests: allows you to learn application interface layout and find out which elements to use and how

Usual flow of using Appium:

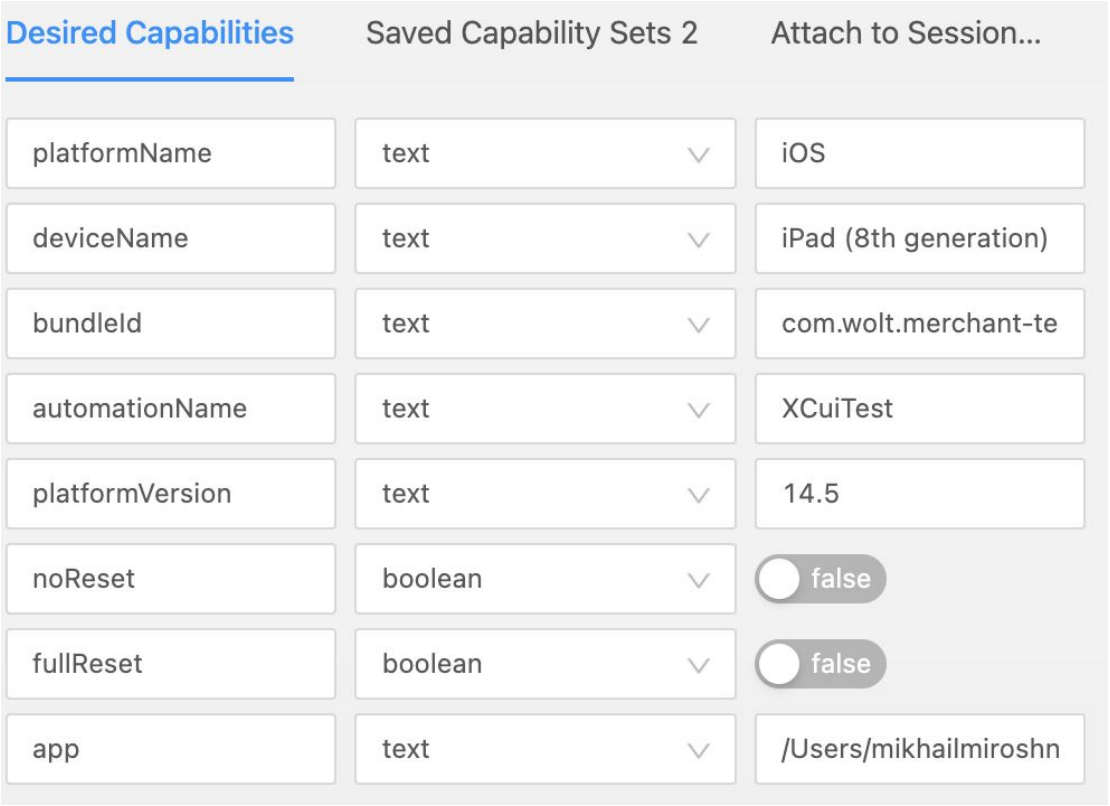
- Startup the server
- Configure connection parameters: Appium capabilities
- Connect to the device
- Learn screen elements layout

Appium: Starting up the server

Home screen of the Appium server.

Also possible to run server UI-less

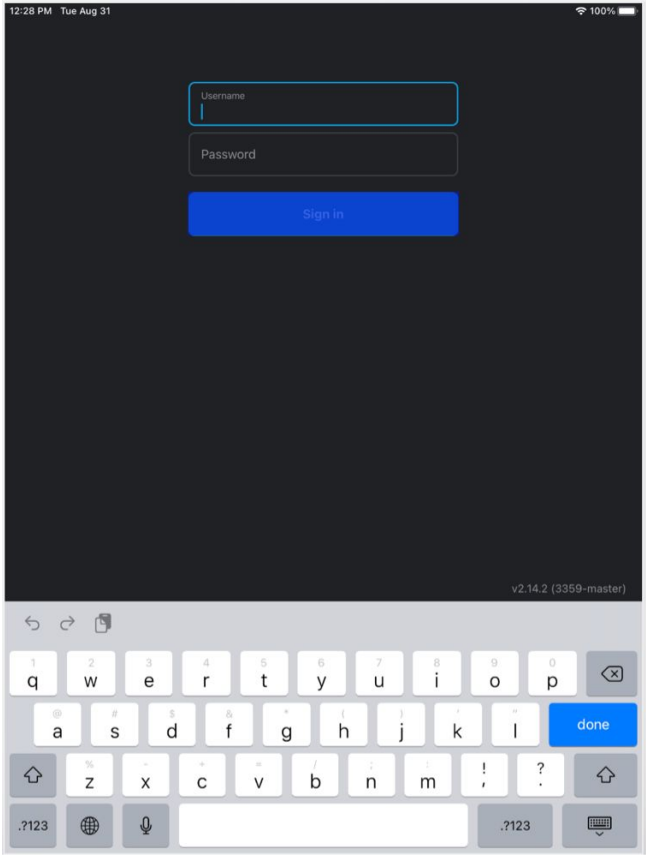
Appium: Connection configuration



Required:
- Platform
- Device
- Application
- Native TA driver

Optionally:
- Clean up setting

Appium: Learning app interface layout



Source Actions

App Source

```
<XCUIElementTypeWindow>
  <XCUIElementTypeOther>
    <XCUIElementTypeOther>
      <XCUIElementTypeOther name="loginView.root
        <XCUIElementTypeOther name="loginView
        <XCUIElementTypeOther name="loginView
          <XCUIElementTypeOther>
        <XCUIElementTypeButton name="loginVie
          <XCUIElementTypeStaticText name="S
          in">
        <XCUIElementTypeStaticText name="v2.14
        (3359-master)">
<XCUIElementTypeWindow>
<XCUIElementTypeWindow>
```

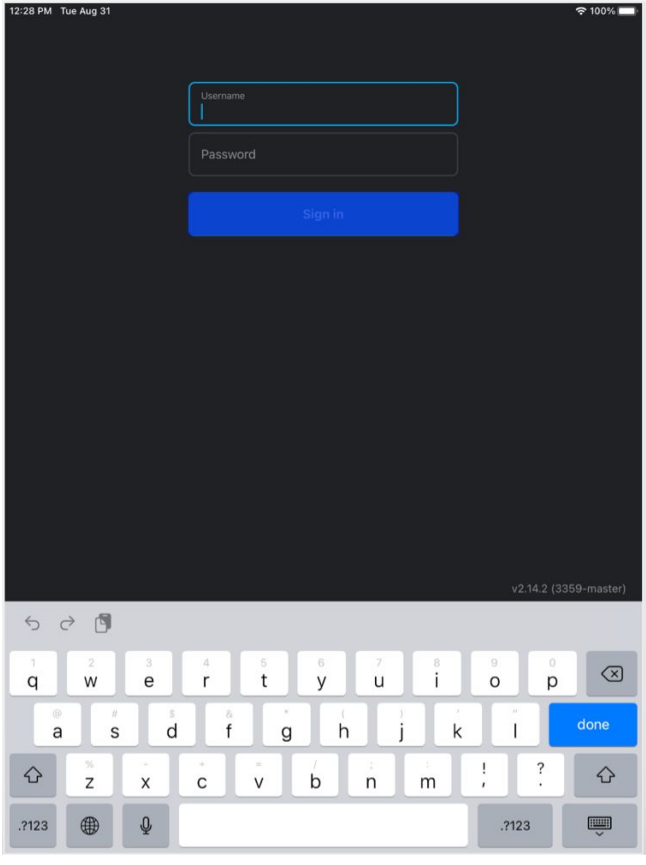
Selected Element

Tap | Send Keys | Clear

Find By	Selector	Time (ms)
accessibility id	loginView.signInButton	Get Timing
-ios class chain(beta)	**/XCUIElementTypeButton[`label == "loginView.signInButton"`]	Get Timing
-ios predicate string(beta)	label == "loginView.signInButton"	Get Timing
xpath	//XCUIElementTypeButton[@name="loginView.signInButton"]	Get Timing

Attribute	Value
elementId	3C000000-0000-0000-0F7D-000000000000
type	XCUIElementTypeButton
name	loginView.signInButton

Appium: Learning ways to locate elements on the screen



Source Actions

App Source

```
<XCUIElementTypeWindow>
  <XCUIElementTypeOther>
    <XCUIElementTypeOther>
      <XCUIElementTypeOther name="loginView.root
        <XCUIElementTypeOther name="loginView
        <XCUIElementTypeOther name="loginView
          <XCUIElementTypeOther>
        <XCUIElementTypeButton name="loginVie
          <XCUIElementTypeStaticText name="S
          in">
        <XCUIElementTypeStaticText name="v2.14
        (3359-master)">
<XCUIElementTypeWindow>
<XCUIElementTypeWindow>
```

Selected Element

Tap | Send Keys | Clear

Find By	Selector	Time (ms)
accessibility id	loginView.signInButton	Get Timing
-ios class chain(beta)	**/XCUIElementTypeButton[`label == "loginView.signInButton"`]	Get Timing
-ios predicate string(beta)	label == "loginView.signInButton"	Get Timing
xpath	//XCUIElementTypeButton[@name="loginView.signInButton"]	Get Timing

Attribute	Value
elementId	3C000000-0000-0000-0F7D-000000000000
type	XCUIElementTypeButton

Appium: The basics

Appium interacts with the app via the Screen Elements Hierarchy Tree

We can access interface structure -> we can create automated UI test

Screen elements, that will be used in tests:

- Must have unique identifiers
- Must be accessible by TA driver

This can be:

- Unique ID
- Accessibility label
- Combination of element type/class with known property

The beginning of TA project development

- We have a project with all required libraries imported

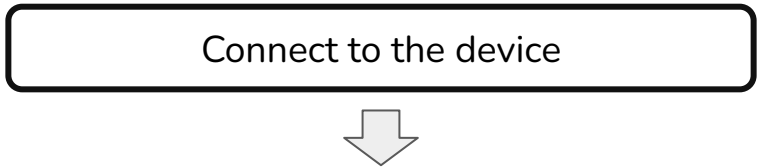
- Connection to the device is configured

- We know screen layouts and how to locate elements used in our tests

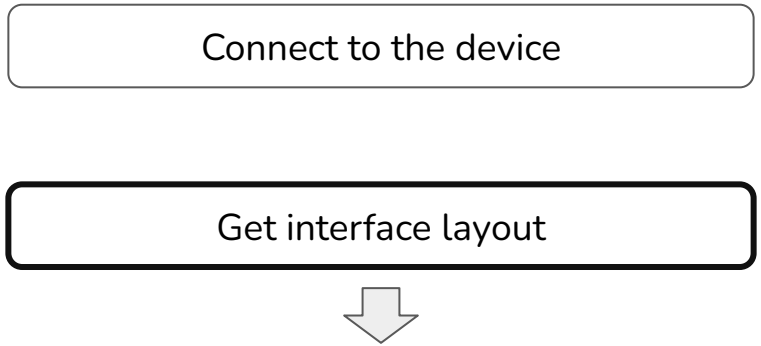
Part 3: Developing automated tests

Wolt

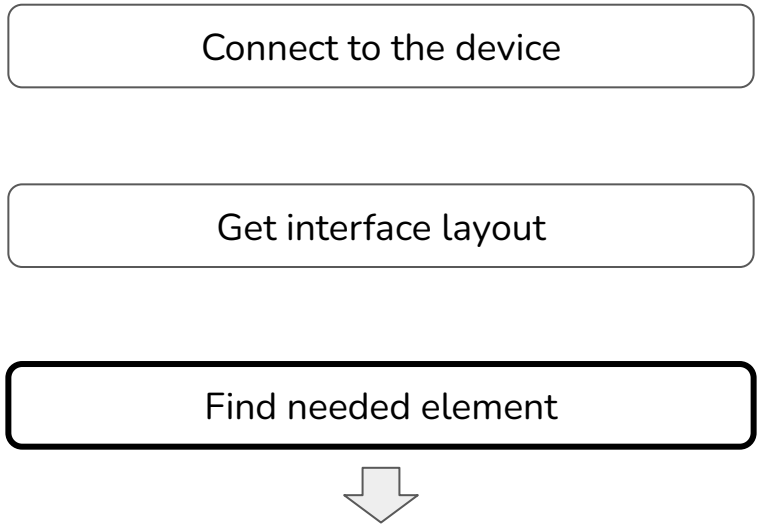
Common automated test structure



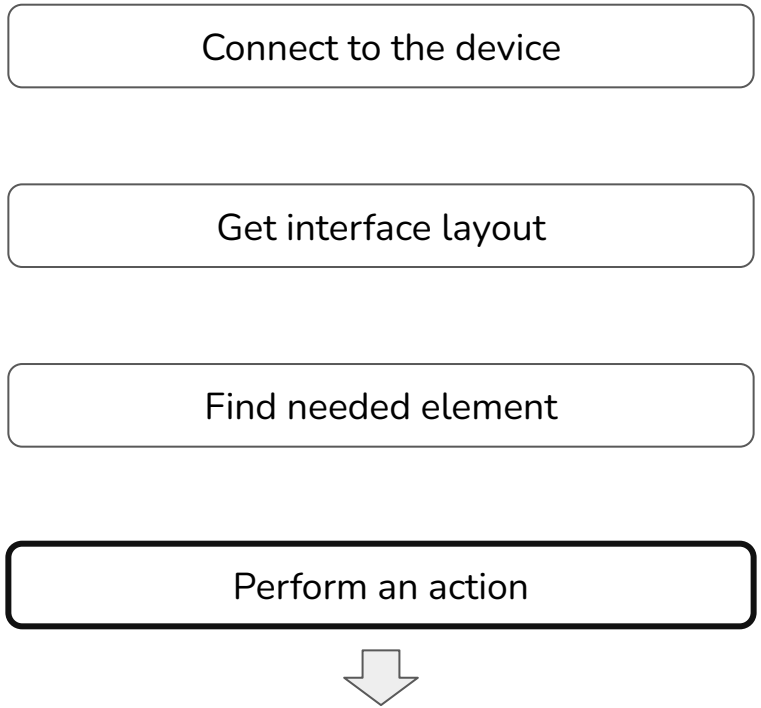
Common automated test structure



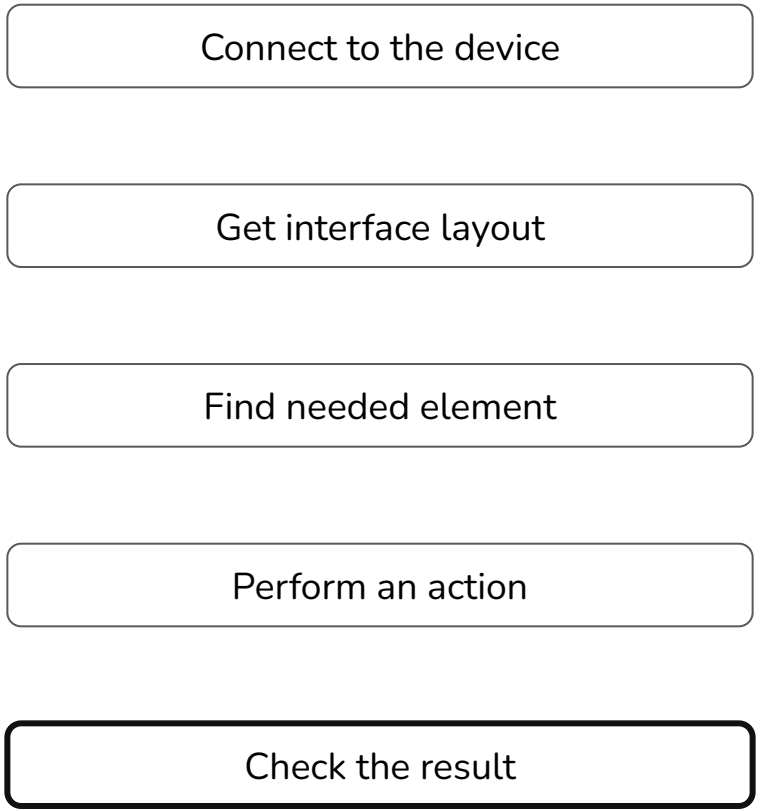
Common automated test structure



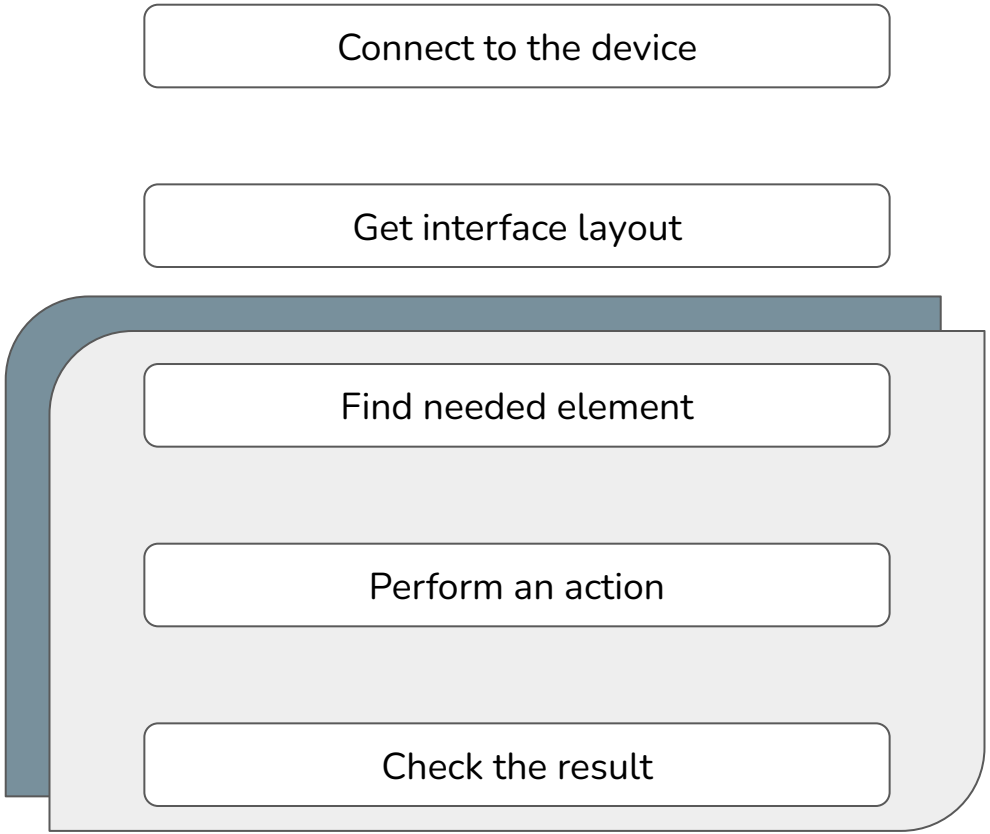
Common automated test structure



Common automated test structure



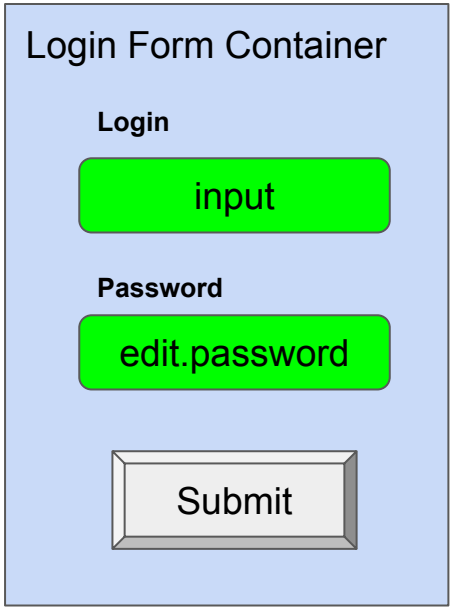
Common automated test structure



Appium commands for the edit field interaction

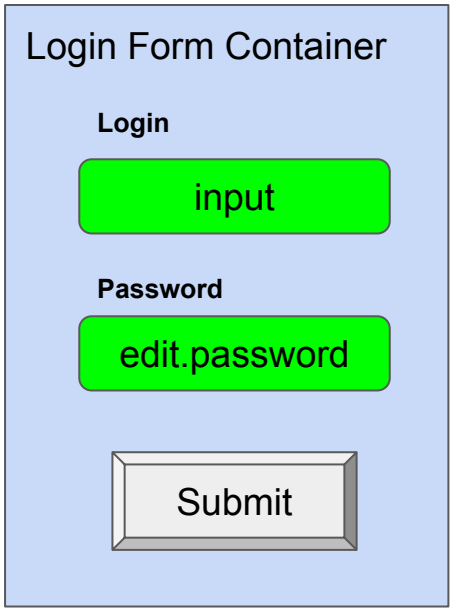
```
// Find edit field element with ID
val editField: MobileElement = driver.findElement(By.id( id: "my_edit_field"))

// Define text to enter
val expectedValue = "Some text to enter"

// Click edit field and enter text
editField.click()
editField.setValue(expectedValue)

// Get actual text in edit field
val actualValue = editField.text

// Compare with expected result
Assert.assertEquals(
    actualValue, expectedValue,
    "Actual text '$actualValue' in edit field is not '$expectedValue'"
)
```

Automated test example: Login form




```
fun testExample() {
    val testUserName = "autoqa"
    val testPassword = "iddqd"
    val formContainer = "LoginFormContainer"
    val loginFieldLocator = "//form[@id='$formContainer//input]"
    val loginField: MobileElement = driver.findElement(By.xpath(loginFieldLocator))
    with(loginField) {  this: MobileElement
        click()
        clear()
        sendKeys(testUserName)
    }
    Assert.assertEquals(loginField.text, testUserName)
    val passwordFieldLocator = "edit.password"
    val passwordField: MobileElement = driver.findElement(By.id(passwordFieldLocator))
    with(passwordField) {  this: MobileElement
        click()
        clear()
        sendKeys(testPassword)
    }
    val ctaButton: MobileElement =
        driver.findElement<MobileElement?>(By.id(formContainer)).findElement(By.linkText( linkText: "Submit"))
    ctaButton.click()

}
```

Multiple tests are using the same element



Test 1: Login with username/password pair

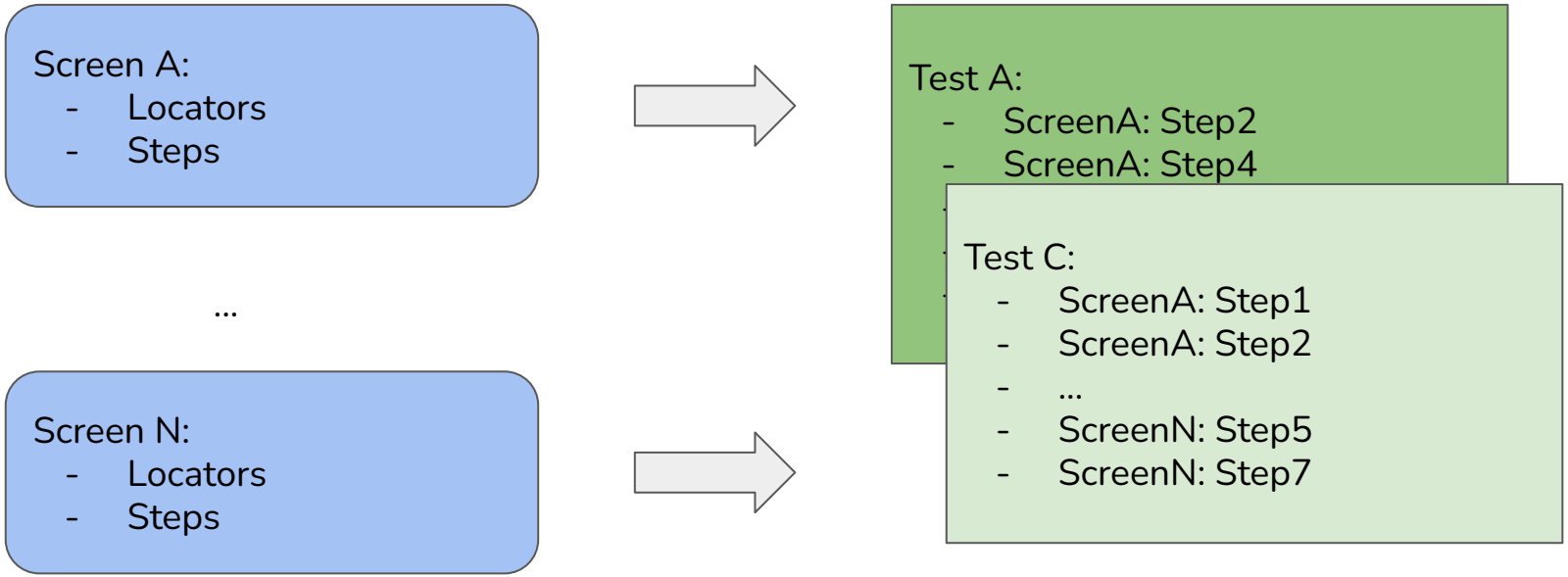
- Find Login element
- ...
- **Find Submit button**
- **Press Submit button**

Test 2: Login without entering credentials

- **Find Submit button**
- **Press Submit button**

Both tests share the same **Step**: *Find and press Login button*

Solution: Split tests and their steps



Page Factory / Page Object

- Allows to separate app tests from screen locators and steps
- Literally: We are creating a Page Objects
- This allows us to compose tests from steps

Nice article:

https://www.guru99.com/page-object-model-pom-page-factory-in-selenium-ultimate-guide.html

Page Factory: Implementation details

Class with the screen definition: Page Object

- Locators: rules how to locate elements for each supported platform
- Steps - actions with elements, located with locators: look for, tap, enter test, get text, scroll etc.

How to use in tests

- Init elements on the screen before calling them in the test
- Call steps from the test

Page Factory: Screen definition - locators

```
@iOSXCUITFindBy(id = "loginView.mainContainer")
@AndroidFindBy(id = "loginScreen.mainView")
private val mainContainer: List<MobileElement>? = null

@iOSXCUITFindBy(id = "loginView.usernameField")
@AndroidFindBy(id = "loginScreen.userEditField")
private val userNameInput: List<MobileElement>? = null

@iOSXCUITFindBy(id = "loginView.passwordField")
@AndroidFindBy(id = "loginScreen.passwordEditField")
private val passwordInput: List<MobileElement>? = null

@iOSXCUITFindBy(id = "loginView.signInButton")
@AndroidFindBy(id = "loginScreen.ctaButton")
private val signInButton: List<MobileElement?>? = null
```

Page Factory: Steps, using located elements

```
fun setPassword(password: String): LogInScreen = also {  it: LogInScreen
    passwordInput?.get(0)!!.sendKeys(password)
}

fun tapSignInButton(): LogInScreen = also {  it: LogInScreen
    signInButton?.get(0)!!.click()
}

fun getSignInButtonText(): String {
    return signInButton?.get(0)!!.text
}
```

Page Factory: Location strategies

FindBy method can locate elements with:

- Tag (Android)
- Accessibility label (iOS)
- Class name
- XPath
- Image (base64 encoded image)
- Android UIAutomator UISelector
- Android Espresso View tag / Data matcher
- iOS UIAutomation classchain, predicate string

https://appium.io/docs/en/commands/element/find-elements/

Page Factory: Variable to store a screen element

val button: List<MobileElement>? = null

- List : Allows to locate all element with the same locator

- ? = null : Initially this variable is NULL

Page Factory: Initialisation

- On creating, screen class is not linked to real elements on the screen

- Before interacting with screen elements, we have to initialise these elements

Page Factory: Where to init Screens - Option 1

In a test that uses steps from these screens (not good: duplicated lines)

- **Init ScreenA**
- **Init ScreenB**
- Use screenA.step1
- Use screenA.step3
- Use screenB.step1
- Use screenB.step4

Page Factory: Where to init Screens - Option 2

When calling the screen from the Screens library: set of Providers

Screen Providers

- **ScreenA**
- **ScreenB**

Test

- On the **ScreenA** { **<-** Init **ScreenA** when calling it for the first time
- Perform **step1** **<-** Use **step1** from initialised **ScreenA**
- Perform **step3**
- }
- On the **ScreenB** { **<-** Init ScreenB
- Perform **step1**
- Perform **step4**
- }

Page Factory: Initialization with the Provider

```
class OnboardingProvider(
    private val driver: AppiumDriver<*>?,
    private val softAssert: SoftAssert
) {

    fun logInScreen(): LogInScreen {
        return LogInScreen(driver, softAssert)
    }
}
```

- **Provider** contains **Screen** definitions
- **One** project can have **multiple** Providers
- **One** provider can have **multiple** screens, united by their kind: Alerts, Modals, App Component

Page Factory: Provider example - App modals

```
class ModalsProvider(
    private val driver: AppiumDriver<*>?,
    private val softAssert: SoftAssert
) {

    fun alertDialog(): AlertDialog {
        return AlertDialog(driver, softAssert)
    }

    fun phoneDialog(): PhoneDialog {
        return PhoneDialog(driver, softAssert)
    }

    fun intercom(): IntercomModal {
        return IntercomModal(driver, softAssert)
    }

    fun moreActionsMenu(): MoreActionsMenu {
        return MoreActionsMenu(driver, softAssert)
    }
}
```

Page Factory: App Root class with Providers

```
class MerchantApp(
    private val driver: AppiumDriver<*>?,
    private val softAssert: SoftAssert
) {

    fun onboarding(): OnboardingProvider {
        return OnboardingProvider(driver, softAssert)
    }

    fun menu(): MenuProvider {
        return MenuProvider(driver, softAssert)
    }

    fun activeOrder(): ActiveOrderProvider {
        return ActiveOrderProvider(driver, softAssert)
    }
}
```

Page Factory: App Root class

- Serves as a single entry point for all application screens

- Aggregates all application providers

- Each product in the project is a separate “application” with its own root class.

Page Factory: Selecting provider for the app

```
merchantApp.
```

onboarding()	OnboardingProvider
settings()	SettingsProvider
activeOrder()	ActiveOrderProvider
menu()	MenuProvider
pastOrder()	PastOrdersProvider
whatsNew()	WhatsNewProvider
modals()	ModalsProvider

Page Factory: Selecting a screen from the Provider

```
merchantApp.modals().
```

↑ m **phoneDialog**()	PhoneDialog
↓ m **intercom**()	IntercomModal
↓ m **alertDialog**()	AlertDialog
↑ m equals(other: Any?)	Boolean
↓ m **moreActionsMenu**()	MoreActionsMenu

Page Factory: Selecting a step for the screen

```
merchantApp.modals().phoneDialog().
```

v	isPhoneDialogShown	PhoneDialog
↑ v	TAG	String
m	getPhoneNumberText()	String
↓ m	getDisclaimerText()	String
↓ m	getTitleText()	String
↓ m	getUserNameText()	String
m	getWarningText()	String
m	tapCloseButton()	OrderCell

Page Factory: Selecting a step for another screen

```
merchantApp.onboarding.logInScreen.
```

v	**isLoginScreenLoaded**	LogInScreen
m	**tapSignInButton**()	LogInScreen
m	**setPassword**(password: String)	LogInScreen
v	TAG	String
m	**setUserName**(userName: String)	LogInScreen

Page Factory: Test with steps

```
@Test
fun exampleTest() {
    merchantApp.onboarding().logInScreen()
        .isLoginScreenLoaded
        .setUserName("autoqa")
        .setPassword("iddqd")
        .tapSignInButton()
}
```

Page Factory: Test using steps from multiple Providers

```
fun invokeIntercomFromSideMenuTest() {

    merchantApp.onboarding
        .signIn()

    merchantApp.menu.header
        .tapMenuButton()

    merchantApp.menu.sideMenu
        .tapMenuItem(MerchantApp.MenuItem.WOLT_SUPPORT)

    merchantApp.modals.intercom
        .isScreenShown()
}
```

- Test contains only steps, no code for locating elements and interacting with them - just like a real Test Case
- One can group steps from the same screen for better readability

Page Factory: Steps, returning a value

... instead of moving to another element or the screen

```
val actualValue = merchantApp.menu().sideMenu().getUserNameLabelText()
val expectedValue = "demo user"

Assert.assertEquals(expectedValue, actualValue, "User name is not correct")
```

Here we got a text from the Text label element and compared it with the expected value.

Page Factory “optimization”

Now we are going to create not a library of pages, but a library for actions on pages

Element actions library

Common element actions might be stored in a separate library

- wait for
- tap
- enter text
- get text
- scroll
- ...

```
@Step( value: "Set user name: '{userName}'")
fun setUserName(userName: String): LogInScreen = also {
    userNameInput?.get(0)!!.sendKeys(userName)
}
```

Get from uncomfortable-to-read code to a short method with arguments

Using actions library in test steps

```
fun setUserName(userName: String): LogInScreen = also {  it: LogInScreen
    sendKeys(userNameInput, userName)
}

fun setPassword(password: String): LogInScreen = also {  it: LogInScreen
    sendKeys(passwordInput, password)
}

fun tapSignInButton(): LogInScreen = also {  it: LogInScreen
    tap(signInButton)
}

fun getSignInButtonText(): String {
    return getText(signInButton)
}
```

“Optimizing” actions on screen elements

Problem: A raw Appium command will throw Unhandled exception on failure

Goal: We’d like to be able to quickly understand why test step failed

Common step failure reasons, leading to runtime exceptions:

- Element doesn’t exist on the screen
- Element has incorrect state: visibility, availability ...
- Action might not be registered or applied
- Text was not entered to the field

Error handling: implementation

Idea:

- Actions on elements should return Boolean value of their result: success or not
- Actions on elements should not throw unhandled exceptions
- Test steps should verify action result

What to do:

- Handle Appium command result in the Actions library
- Add checks to test steps, that use actions on elements

Example: Element action to enter text to the element

```
fun sendKeys(elements: List<MobileElement>?, num: Int, txt: String?): Boolean {
    Start timer when we started trying to enter the text
    do {
        try {
            Trying to enter the text as is. On success, return
            "Everything is OK"
        } catch (e: ElementNotInteractableException) {
            If element can't be interacted ATM, we can try again a bit later
        } catch (e: Exception) {
            e.  If something unexpected happened, return "We have a problem"
        }
        Make a pause between tries
    } while  Until time for tries run out
    return   At the end, return what we got with our tries
}
```

Example: Element action to enter text to the element

```
fun sendKeys(elements: List<MobileElement>?, num: Int, txt: String?): Boolean {
    val startTime = System.currentTimeMillis()
    do {
        try {
            (elements?.get(num) as MobileElement).sendKeys(txt)
            sleep(Timer.TAP_DELAY)
            return true
        } catch (e: ElementNotInteractableException) {
            log( text: TAG + "sendKeys(): retry...")
        } catch (e: Exception) {
            e.printStackTrace()
        }
        sleep(Timer.TAP_DELAY)
    } while (System.currentTimeMillis() < startTime + 10 * 1000) // 10 sec
    return false
}
```

Test steps with action result assertions

```
fun setUserName(userName: String): LogInScreen = also {  it: LogInScreen
    Assert.assertTrue(sendKeys(userNameInput, userName), "${TAG}setUserName(): FAILED")
}

fun setPassword(password: String): LogInScreen = also {  it: LogInScreen
    Assert.assertTrue(sendKeys(passwordInput, password), "${TAG}setPassword(): FAILED")
}

fun tapSignInButton(): LogInScreen = also {  it: LogInScreen
    Assert.assertTrue(tap(signInButton), "${TAG}tapSignInButton(): FAILED")
}
```

TAG - Current class name to add to the error message for better visibility

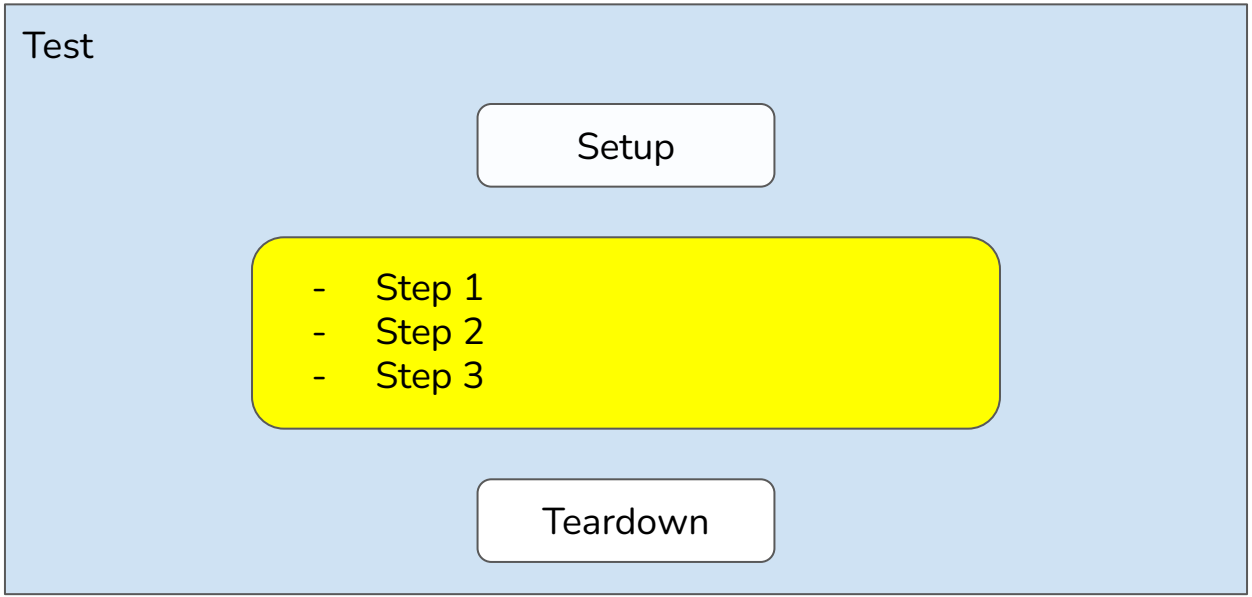
Extending actions that test can do

Now our tests are looking nice and quite stable, without throwing unhandled exceptions

Time to do some preparation work before our tests!

TestNG: Steps before and after the test

Test is working. How to add additional actions for setup and teardown



TestNG: Controlling test flow

Annotation: TestNG **functions** to run actions before and after test methods

- Before testing, once per suite: @BeforeSuite
- Before each test: @BeforeMethod
- After each test: @AfterMethod
- After running all tests, but before end of the suite, once: @AfterTest
- After completing the suite, once: @AfterSuite

Documentation:

https://testng.org/doc/documentation-main.html#annotations

Test management: using TestNG

Examples:

- Start and stop Appium server: **@Before/After Suite**

- Clear cache and create test data before the test, delete or restore test data after the test : **@Before/After Method**

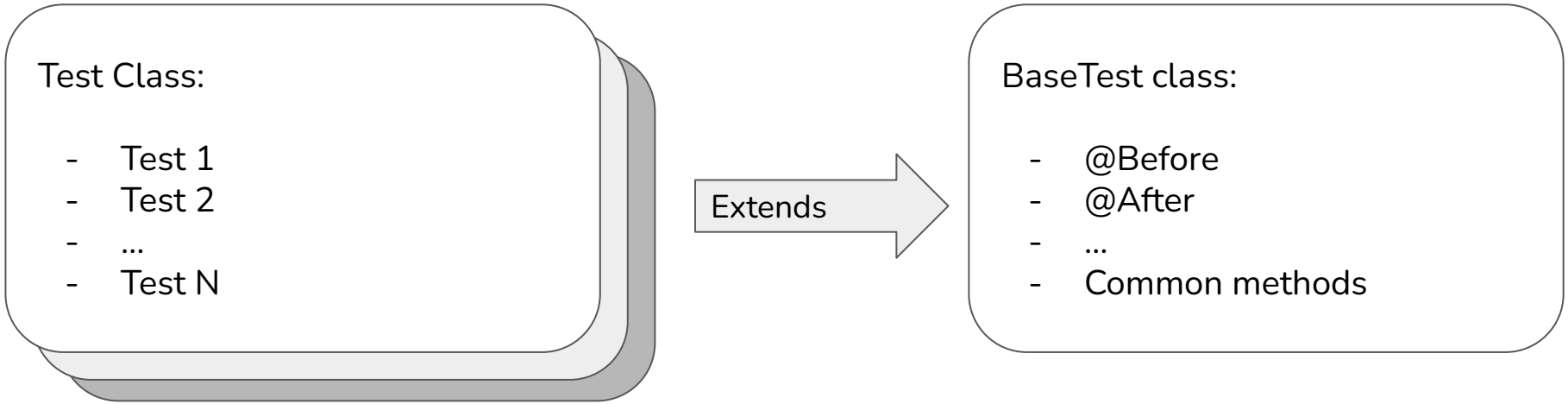
- Collect test statistics and compose reports: **@AfterTest**

Test management: using TestNG

Methods with TestNG annotations

```
@Parameters( ...value: "product", "deviceName", "runner")
@BeforeSuite(alwaysRun = true)
fun beforeSuite(
    @Optional( value: "") product: String,
    @Optional( value: "") deviceName: String,
    @Optional( value: "") runner: String
) {...}

@AfterSuite(alwaysRun = true)
fun tearDown(iTestContext: ITestContext) {...}
```

Test classes hierarchy



Reports

Out test is running, it can prepare test environment, and it cleans up temporary data. How to present test result?

Here is a basic test run report, composed by Maven

```
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 300.554 s - in TestSuite
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  05:19 min
[INFO] Finished at: 2021-08-13T14:40:10+03:00
[INFO] ------------------------------------------------------------------------
```

Text log in the terminal when running the test

Useful when developing and testing a product:

- Displays what test does at any moment of time
- Preserves history of all test steps: nice for debugging and analysis
- Shows execution flow, when devices screen is not visible (running a test on the remote device)

Text log: Implement by extending test steps

```
fun setUserName(userName: String): LogInScreen = also {  it: LogInScreen
    log( text: TAG + "setUserName(): " + userName)
    Assert.assertTrue(sendKeys(userNameInput, userName), "${TAG}setUserName(): FAILED")
}

fun setPassword(password: String): LogInScreen = also {  it: LogInScreen
    log( text: TAG + "setPassword(): ")
    Assert.assertTrue(sendKeys(passwordInput, password), "${TAG}setPassword(): FAILED")
}

fun tapSignInButton(): LogInScreen = also {  it: LogInScreen
    log( text: TAG + "tapSignInButton():")
    Assert.assertTrue(tap(signInButton), "${TAG}tapSignInButton(): FAILED")
}
```

Text log in action

```
  BeforeMethod
[SYSTEM]        Test parameters: checkSideMenuItemsListTest, product: merchantApp, env: staging
[SYSTEM]        startDriver(): LOCAL
[SYSTEM]        startDriver(): appFile: /Users/mikhailmiroshnichenko/Documents/Wolt/MerchantTest.app
[SYSTEM]        startDriver(): Driver DO FAST RESET
[SYSTEM]        startDriver(): Launch parameters: -skipIntros
[SYSTEM]        startDriver(): deviceName: iPad (8th generation), OS version: 14.5
[SYSTEM]        startDriver(): Local IOSDriver
Jul 07, 2021 5:28:16 PM io.appium.java_client.remote.AppiumCommandExecutor$1 lambda$0
INFO: Detected dialect: W3C
[SYSTEM]        startDriver(): completed in 13 sec
[TEST]          checkSideMenuItemsListTest                  LogInScreen()               | isLoginScreenLoaded():
[TEST]          checkSideMenuItemsListTest                  LogInScreen()               | setUserName(): tablet
[TEST]          checkSideMenuItemsListTest                  LogInScreen()               | setPassword():
[TEST]          checkSideMenuItemsListTest                  LogInScreen()               | tapSignInButton():
[TEST]          checkSideMenuItemsListTest                  Header()                    | isHeaderLoaded():
[TEST]          checkSideMenuItemsListTest                  Header()                    | tapMenuButton():
[TEST]          checkSideMenuItemsListTest                  SideMenu()                  | isSideMenuShown():
[TEST]          checkSideMenuItemsListTest                  SideMenu()                  | getActiveOrdersItemText():
[TEST]          checkSideMenuItemsListTest                  SideMenu()                  | getPastOrdersItemText():
[TEST]          checkSideMenuItemsListTest                  SideMenu()                  | getEditMenuItemText():
[TEST]          checkSideMenuItemsListTest                  SideMenu()                  | getTutorialVideosItemText():
[TEST]          checkSideMenuItemsListTest                  SideMenu()                  | getWoltSupportItemText():
[TEST]          checkSideMenuItemsListTest                  SideMenu()                  | getSettingsItemText():
[TEST]          checkSideMenuItemsListTest                  SideMenu()                  | getWhatsNewItemText():

  AfterMethod
Test result: PASSED
[SYSTEM]        quit driver
```

“Optimizing” reports

From a static text log, linked to a test execution on a specific test runner, to a complex test report with images and on a human language. Somewhere on a publicly available web-server.

Creating reports with Allure

Why: We'd like to be able to understand what was tested, with what test and which steps, and what was the result - in an easily accessible manner

How to start using it:

- Add library to the TA project with Maven
- Add special annotations to classes, tests and steps
- Generate a report

Important: Allure is a web-server, running locally or in the cloud.

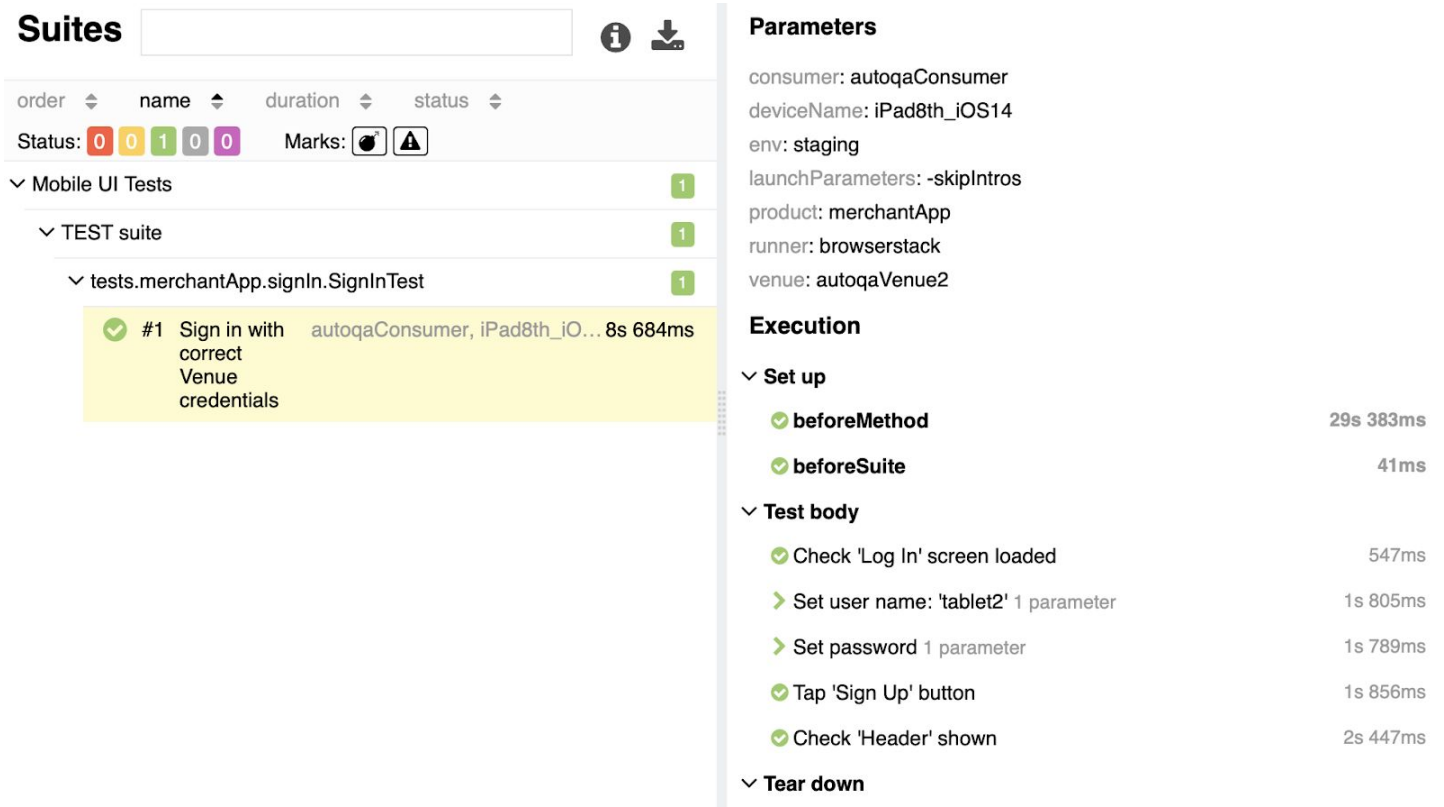
Allure: Adding annotations to a test class

```
@Epic( value: "autoqa demo")
@Feature( value: "Using Allure for reporting")
class SignInTest : BaseTest() {

    @Test(description = "Sign in with correct Venue credentials", groups = ["ios"])
    fun signInWithCorrectCredentialsTest() {

        merchantApp.onboarding().signIn()
    }
}
```

Allure: Adding annotation to a test class

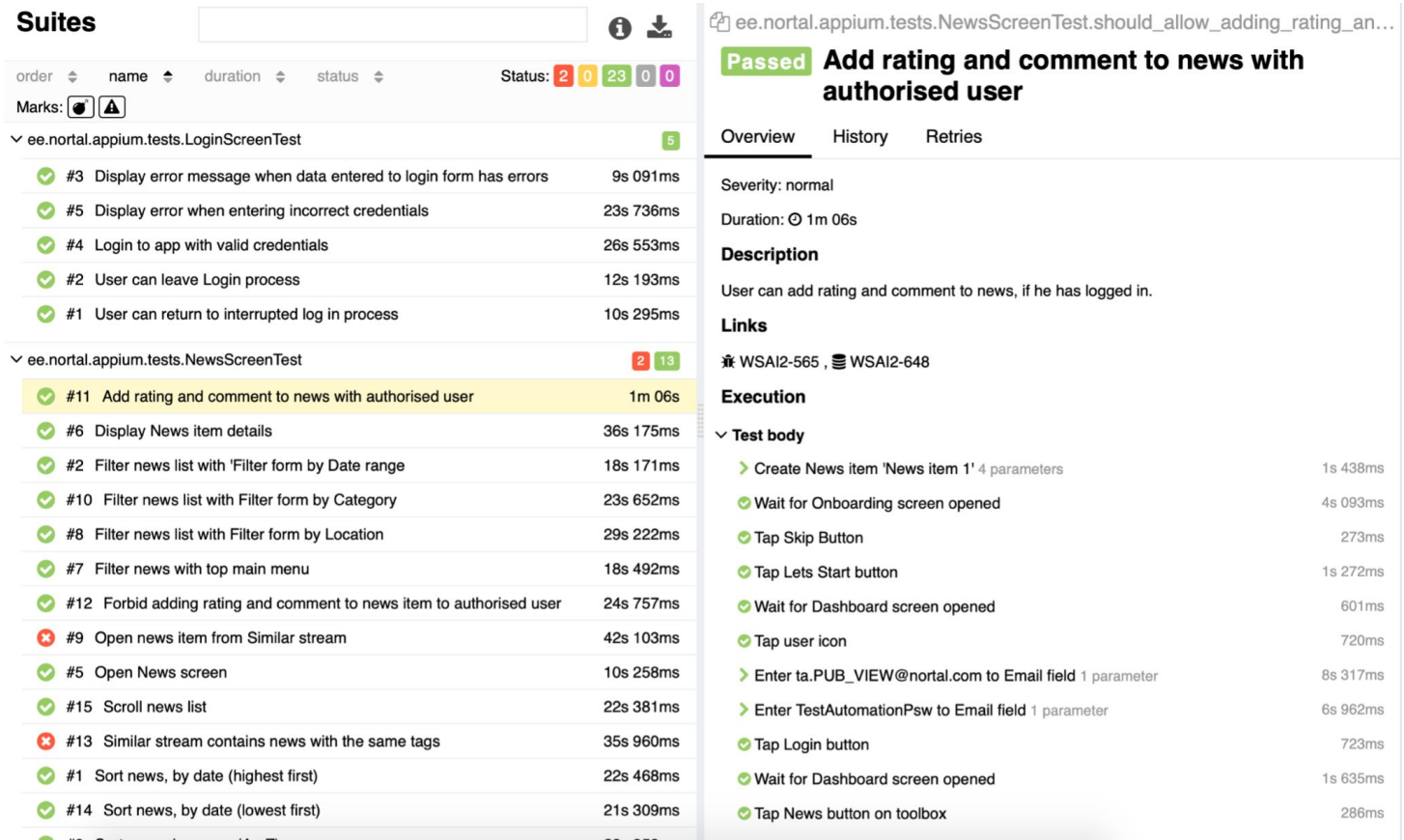
```
@Step("Set user name: '{userName}'")
fun setUserName(userName: String): LogInScreen = also { it: LogInScreen
    log( text: TAG + "setUserName(): " + userName)
    Assert.assertTrue(sendKeys(userNameInput, userName), "${TAG}setUserName(): FAILED")
}

@Step("Set password")
fun setPassword(password: String): LogInScreen = also { it: LogInScreen
    log( text: TAG + "setPassword(): ")
    Assert.assertTrue(sendKeys(passwordInput, password), "${TAG}setPassword(): FAILED")
}

@Step("Tap 'Sign Up' button")
fun tapSignInButton(): LogInScreen = also { it: LogInScreen
    log( text: TAG + "tapSignInButton():")
    Assert.assertTrue(tap(signInButton), "${TAG}tapSignInButton(): FAILED")
}
```

Allure: Web page with the test report

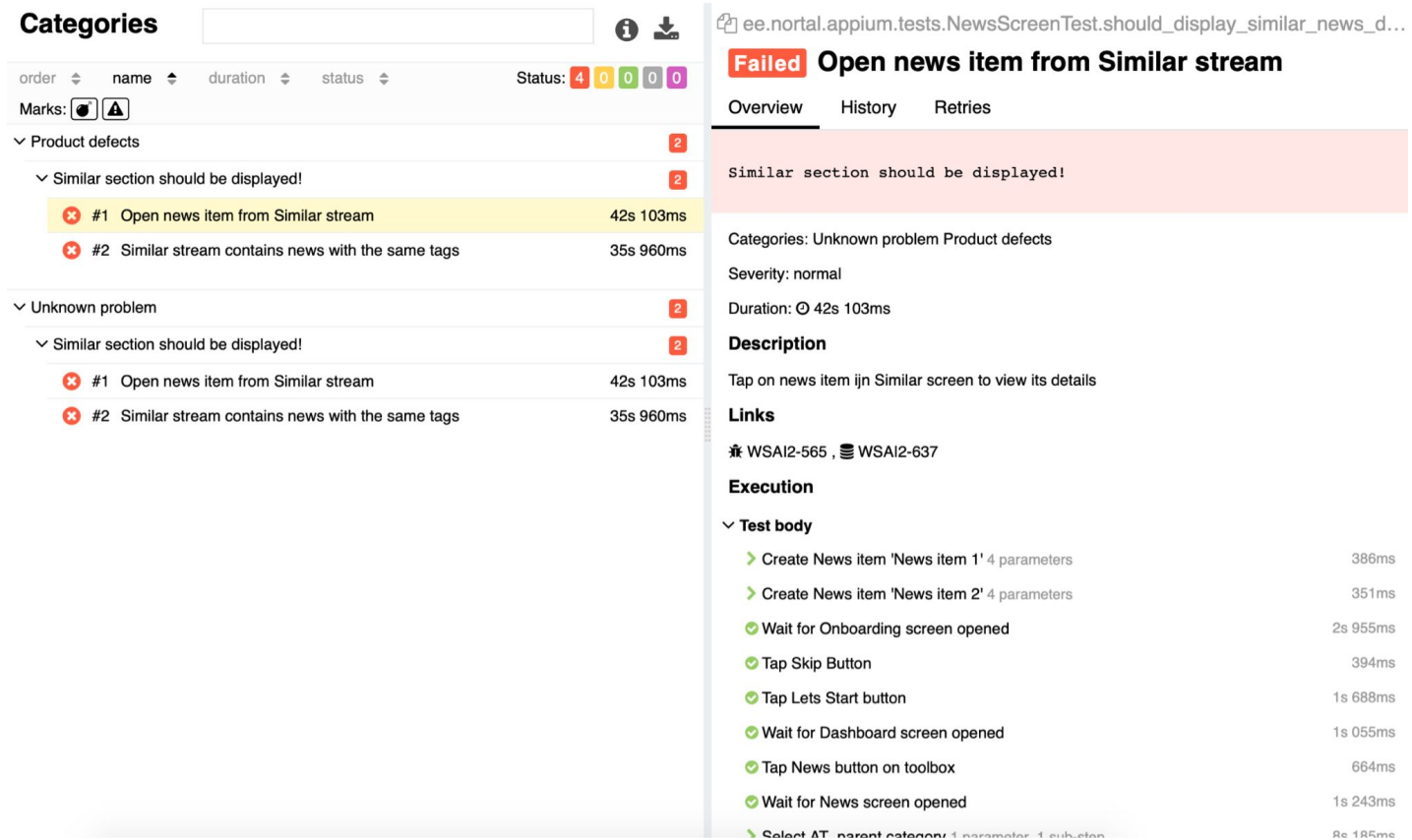


Allure: Test runs history

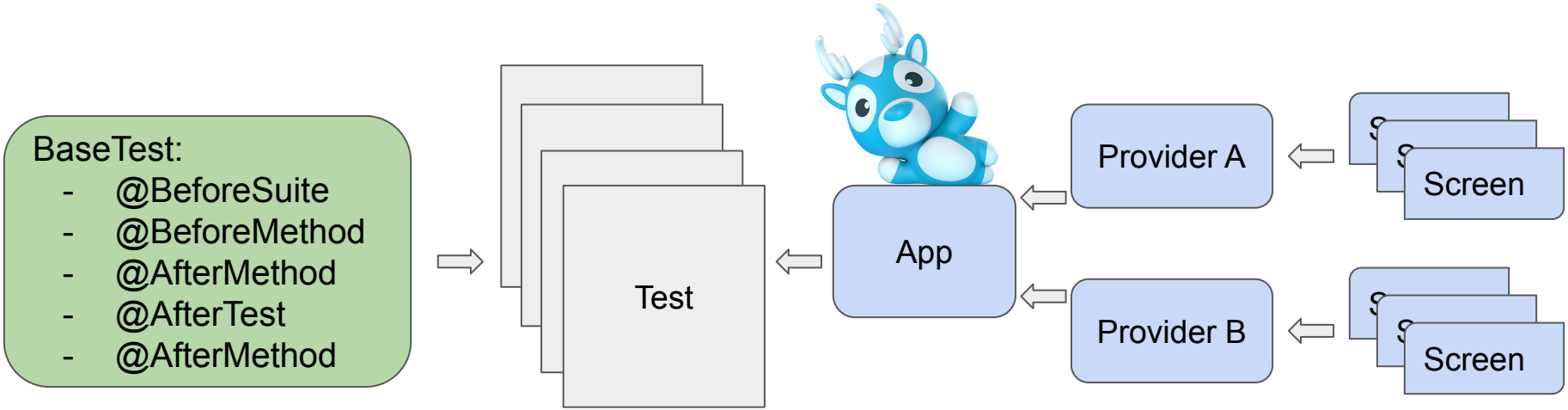
Allure: Test list with their steps



Allure: Display test failure reason in the report



Final TA project structure

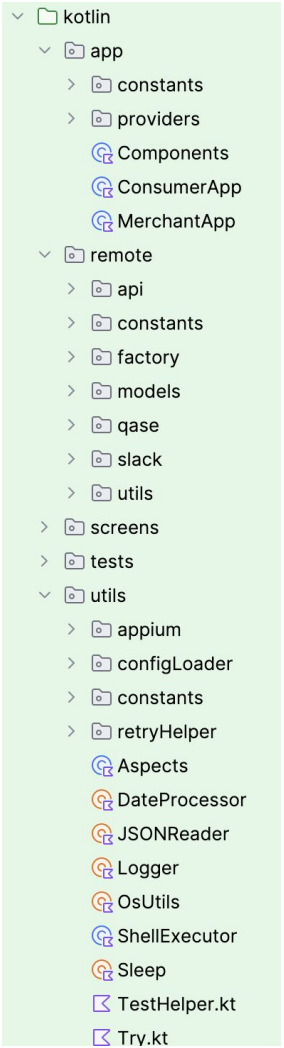


Where to run automated tests?

Now we have **what** to run and a **tool** to run. But **where** and **when** to run?

- **By yourself**: developer or QA engineer executes a test by manual commands

- **On Continuous Integration servers**:
  - Push to a development branch
  - Open Pull request for review
  - Merge to Master



Part 4: Running automated tests

Wolt

Executing tests on a local machine

Requires:

- Real device
- Emulator / Simulator
- All needed tools and libraries

\+ Works as fast as possible: device under the test and test runner are located on the same machine

\- Requires quite complex initial setup
\- Requires powerful machine to host virtual devices

Real device

Pros:

+ Test runner performance is not important
+ Same environment as real user has
+ All popular services are available: Google, Apple etc.

Cons:

- Limited access to some device / OS functions
- Device should be bought - might be expensive
- You have to maintain devices yourself - might be even more expensive

Virtual device

Pros:

+ Any configuration without extra expenses
+ You can do anything with the device

Cons:

- No popular services: Markets, vendor-specific and 3rd party apps
- High hardware requirements for the test runner

You need to consider the need and differences between virtual and real devices from testing point of view.

iOS devices

Includes:

- iOS simulator, as a part of XCode IDE for macOS

- iPhone / iPad

iOS devices

Specifications:

- Requires macOS
- Requires a developer account to build for a real app
- Some functions are blocked on a real device
- Interface composition might differ between OS versions
- App for a virtual device: **.app** x86 (ARM on a Apple Silicon device)
- App for a real device: **.ipa** ARM

Android devices

Includes:

- Emulator by Android Studio: AVD - Android Virtual Device, Win / macOS
- Real devices

Android devices

Specifications:

- Can build an app file, that will work on both virtual and real devices
- Easy to install to almost any device

Demo: Running the test on an iPad Simulator

Using **Wolt Merchant App for iPad**: Wolt application for restaurant and stores to process orders and manage their menu.

Test: Reject an order

- Sign in to the application
- Send order to the app
- Reject received order

Where is was executed:

- Locally, on my machine
- On an iPad 8th Simulator

Execute on another machine: same setup as local one

- **With building the TA project and on an attached device.**
  - Java
  - Maven
  - Appium
  - iOS: XCode
  - Android: Android debug tools
  - Allure (for reports only, not needed for running)

Execute on another device: device farms

- **With building the TA project and on an attached device.**
  - Java
  - Maven
  - Appium
  - iOS: XCode
  - Android: Android debug tools
  - Allure (for reports only, not needed for running)
- **With building the TA project and on a remote device in the cloud device farm**
  - Java
  - maven
  - Appium is not needed: it is deployed on the device farm

Execute on another machine: inside Docker image

- **With building the TA project and on an attached device.**
  - Java
  - Maven
  - Appium
  - iOS: XCode
  - Android: Android debug tools
  - Allure (for reports only, not needed for running)
- **With building the TA project and on a remote device in the cloud device farm**
  - Java
  - maven
  - Appium is not needed: it is deployed on the device farm
- **In Docker** with project and with all imported libraries
  - Perfectly suited for CI

Using Docker: Dockerfile

```
FROM maven:latest

# Create folders
RUN mkdir -p /usr/tests && \
    mkdir -p /usr/tests/src && \
    mkdir -p /usr/tests/suites \
    mkdir -p /usr/external_files

# Define working dir with test files
WORKDIR /usr/tests

# Copy source files
COPY src /usr/tests/src

# Copy suites
COPY suites /usr/tests/suites

# Copy main MVN file
COPY pom.xml /usr/tests

# Copy device settings file
COPY capabilities.json /usr/tests

# Pre install maven dependencies
RUN mvn install
```

Using Docker: Build order

Create a dockerfile based on a Maven image:
https://hub.docker.com/_/maven

Build image from the dockerfile:

```
docker build -t autoqa -f DockerfileTest .
```

Run tests inside container:

```
docker run --rm autoqa /bin/bash -c "mvn
-DsuiteName=mySuite.xml test"
```

Using Docker: Execute in the Container

Where to run tests: container has no virtual devices inside and GitHub runner is not connected to any real device?

Run tests inside container:

```
docker run - - rm autoqa /bin/bash -c "mvn
-DsuiteName=mySuite.xml test"
```

Answer: On a device farm

Cloud device farms

What to do when you have to run your app and tests on real device, but you have no devices (or you have some, but not enough)?

Or you can get needed device, but don’t want to perform maintenance.

Solution: Cloud device farms with real devices for Mobile and Web apps

- BrowserStack
- SauceLabs

Cloud device farm

- Available by subscription plan

- Has popular mobile devices with various OS versions
- Has Web-environments with popular browsers and different versions

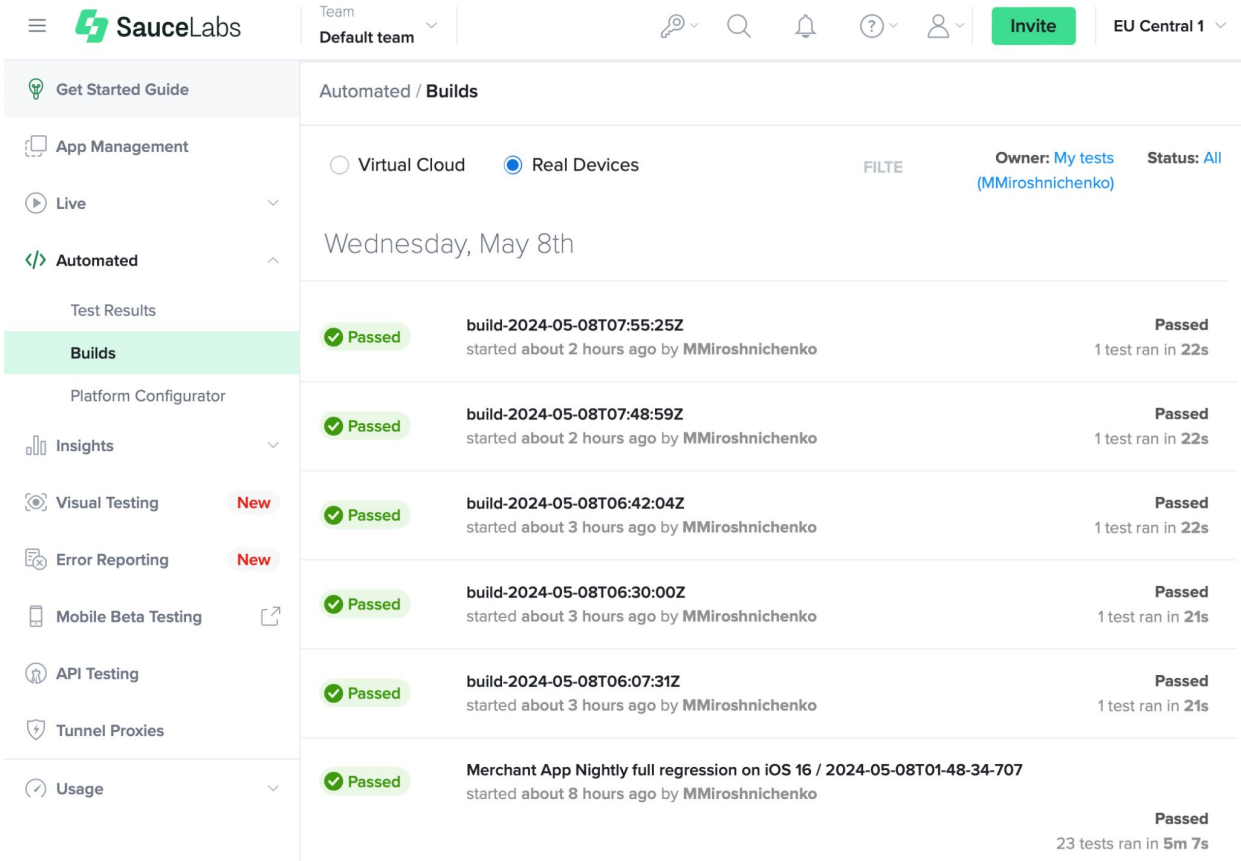
- Allows to run automated tests
- Allows manual testing on available devices/browsers through web-interface

Cloud device farm

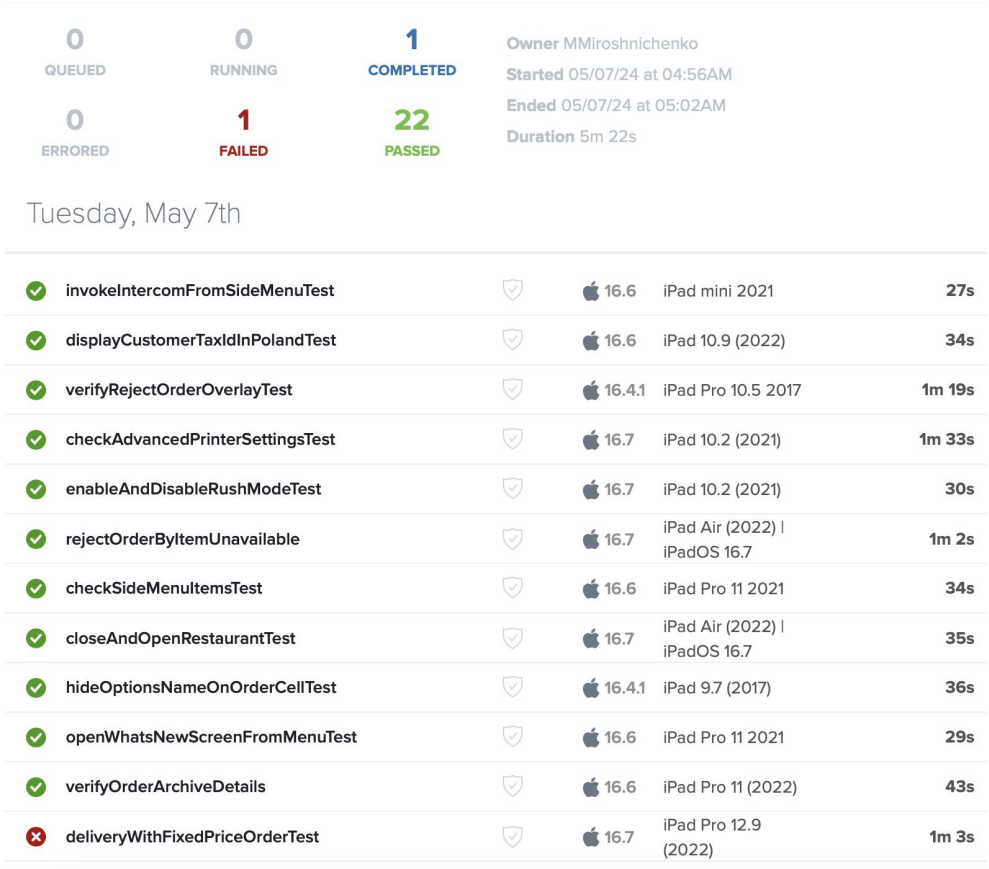
Let’s take a look on a SauceLabs dashboard with the list of executed tests

- List of test suites
- List of test in the suite
- Test details

Test suites list



Test list in the suite

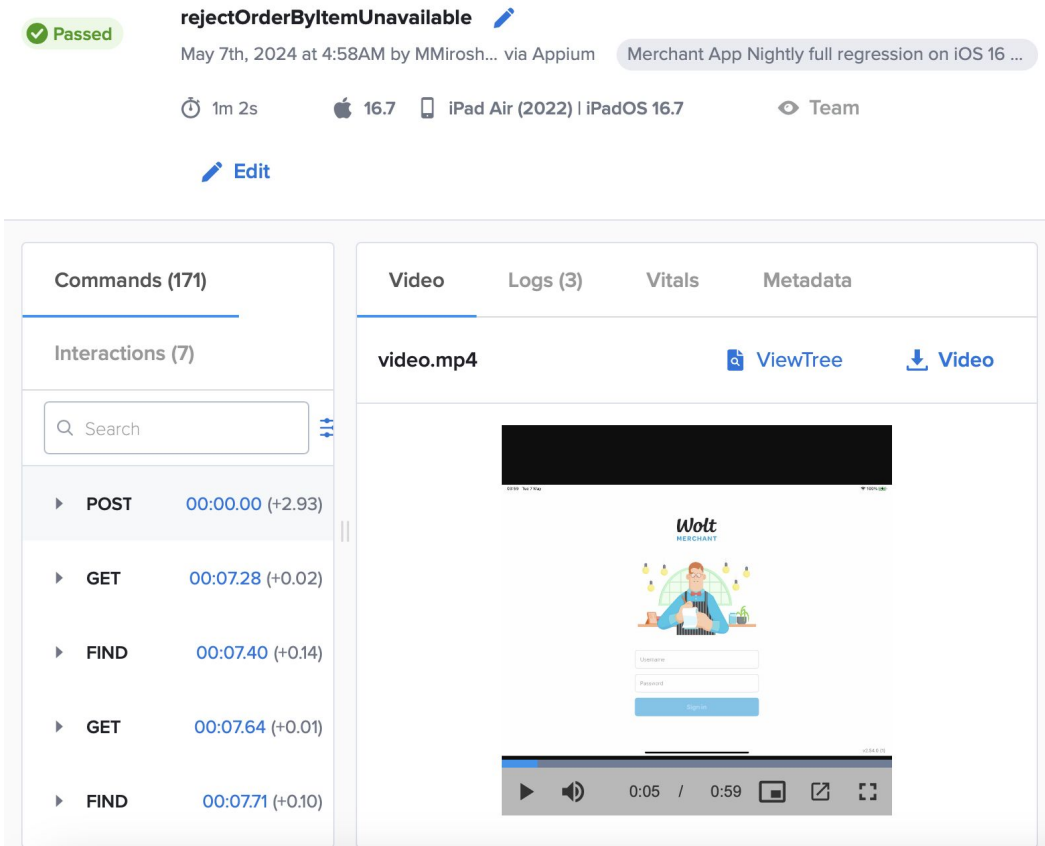


0 QUEUED	0 RUNNING	1 COMPLETED	Owner MMiroshnichenko	
0 ERRORED	1 FAILED	22 PASSED	Started 05/07/24 at 04:56AM	
			Ended 05/07/24 at 05:02AM	
			Duration 5m 22s	

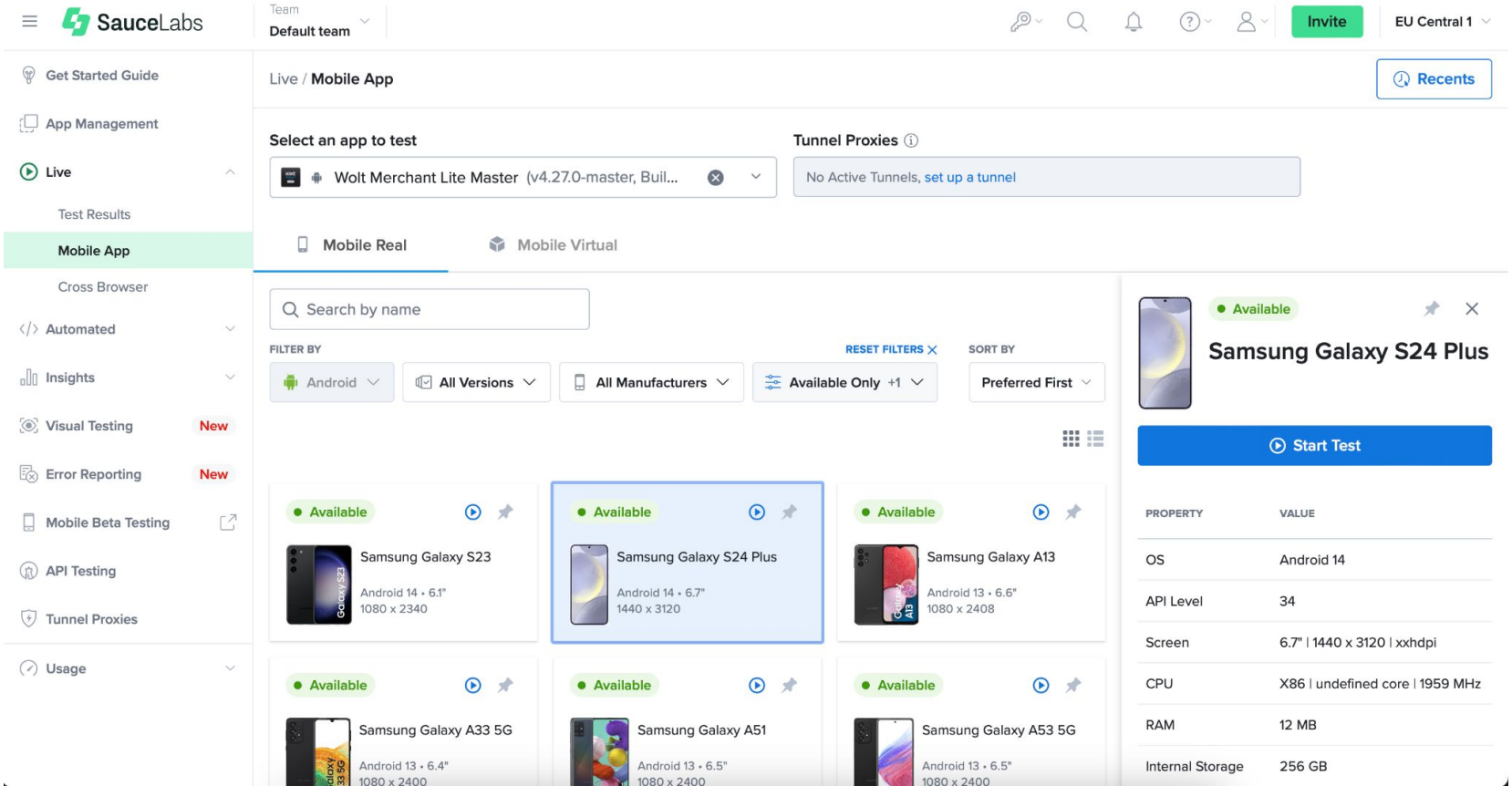
Tuesday, May 7th

invokeIntercomFromSideMenuTest	16.6	iPad mini 2021	27s
displayCustomerTaxIdInPolandTest	16.6	iPad 10.9 (2022)	34s
verifyRejectOrderOverlayTest	16.4.1	iPad Pro 10.5 2017	1m 19s
checkAdvancedPrinterSettingsTest	16.7	iPad 10.2 (2021)	1m 33s
enableAndDisableRushModeTest	16.7	iPad 10.2 (2021)	30s
rejectOrderByItemUnavailable	16.7	iPad Air (2022) | iPadOS 16.7	1m 2s
checkSideMenuItemsTest	16.6	iPad Pro 11 2021	34s
closeAndOpenRestaurantTest	16.7	iPad Air (2022) | iPadOS 16.7	35s
hideOptionsNameOnOrderCellTest	16.4.1	iPad 9.7 (2017)	36s
openWhatsNewScreenFromMenuTest	16.6	iPad Pro 11 2021	29s
verifyOrderArchiveDetails	16.6	iPad Pro 11 (2022)	43s
deliveryWithFixedPriceOrderTest	16.7	iPad Pro 12.9 (2022)	1m 3s

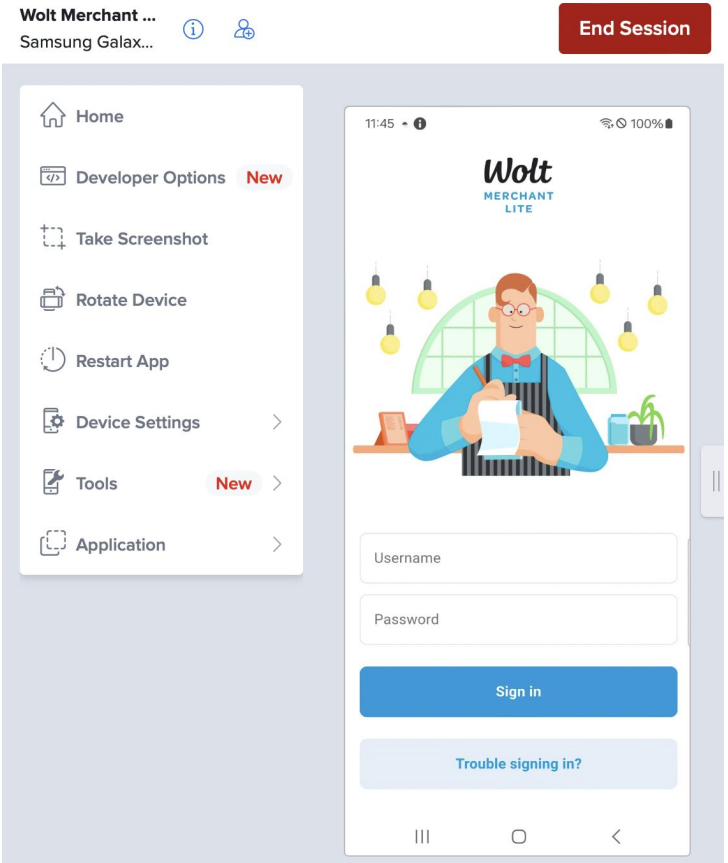
Test details



Live session - pick up a device



Run an app on a remote device



How to run a test in the cloud

Steps to implement running an automated test in the cloud:

- **Get** an access token for your account
- **Upload** an app to the Device farm storage
- **Connect** to a device with a test
- **Run** test as usual
- ....
- **Receive** the test result and the video recording

Setup connection to SauceLabs

```
MutableCapabilities caps = new MutableCapabilities();
caps.setCapability("platformName", "Android");
caps.setCapability("appium:platformVersion", "11");
caps.setCapability("appium:deviceName", "Samsung.*Galaxy.*");
caps.setCapability("appium:orientation", "portrait");
caps.setCapability("appium:app", "storage:filename=<file-name>");
MutableCapabilities sauceOptions = new MutableCapabilities();
sauceOptions.setCapability("username", "SAUCE_USERNAME");
sauceOptions.setCapability("accessKey", "SAUCE_ACCESS_KEY");
caps.setCapability("sauce:options", sauceOptions);
```

Then add URL to the SauceLabs servers:

```
"https://${cloudUser.user}:${cloudUser.key}@ondemand.${serverName.url}.saucelabs.com$port/wd/hub"
```

Now we have setup for execution in the cloud

What to run: product tests

With what to run: our framework

Where to run: inside Docker container

On what to run: in the cloud device farm

Time to configure CI

Part 5: CI configuration

Wolt

TA integration to the product CI: plan

- Project is in GitHub
- Docker image built with GitHub Actions
- Store important data in GitHub Secrets
- Store Docker image in AWS ECR - Amazon Elastic Container Registry
- Build and run TA with GitHub runners
- Send test reports to Slack and TMS Qase

TA integration to the product CI: Docker image

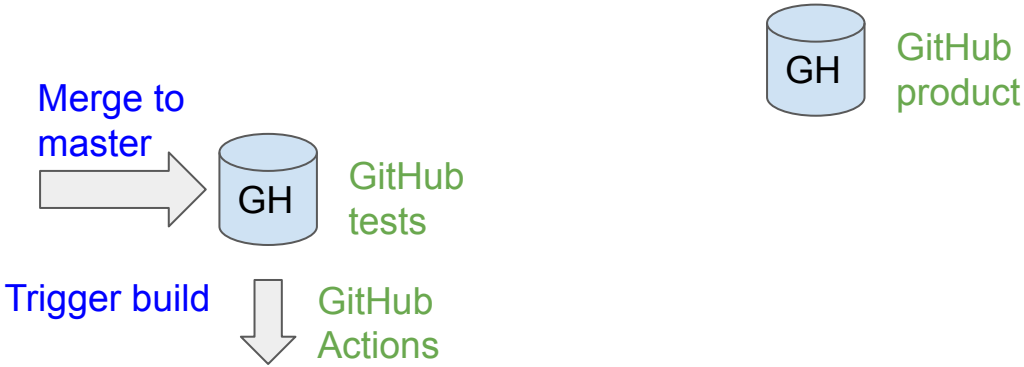
- Project is in GitHub
- Docker image built with GitHub Actions
- Store important data in GitHub Secrets

Why to use Docker:

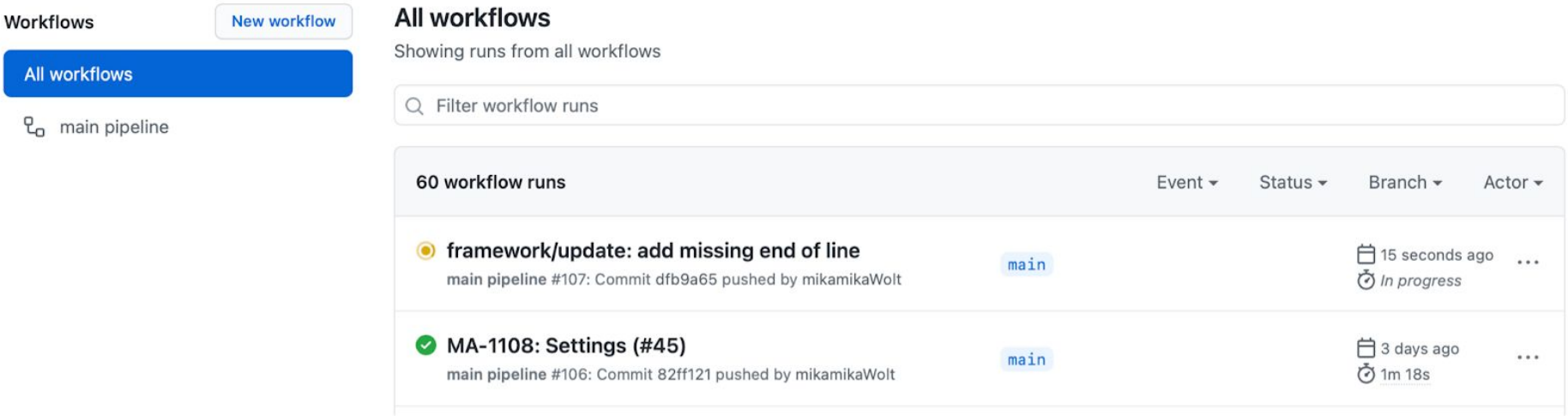
- GH runner has no Java and Maven to run Test Automation project
- We can pack everything needed to run the tests: Java, Maven, project dependencies once and don’t wait for import when building and running the suite for the first time
- Docker image means that we have a stable and reliable environment to run test suites.

CI: Repos with the source code

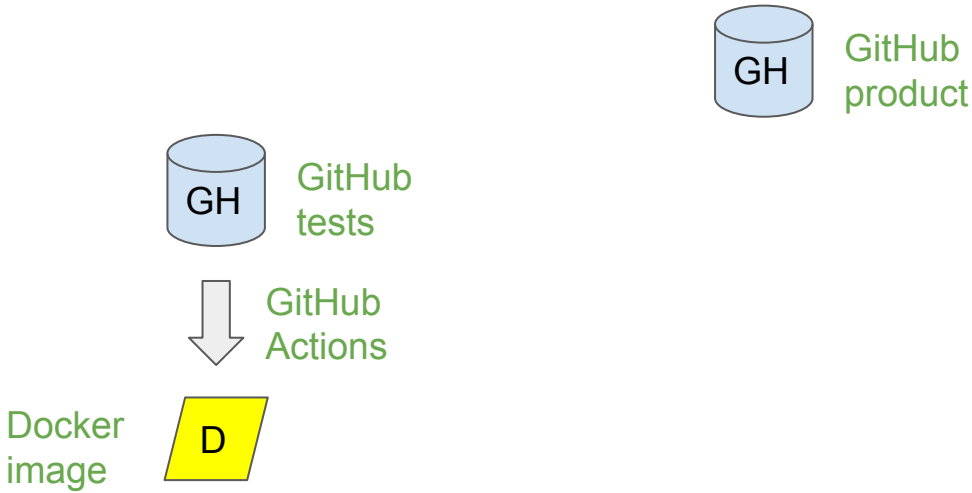
CI: Merging new tests triggers GitHub Actions



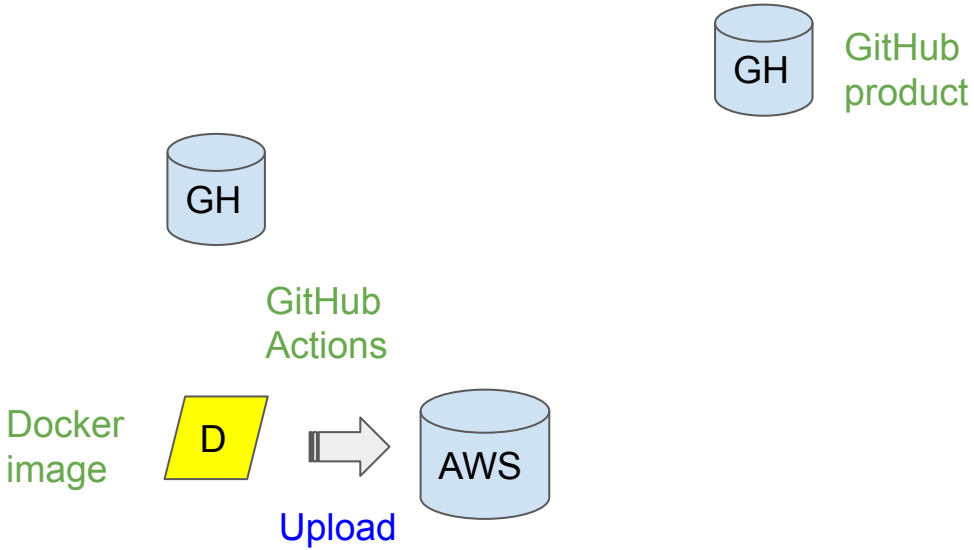
GitHub Actions: Building TA project has begun



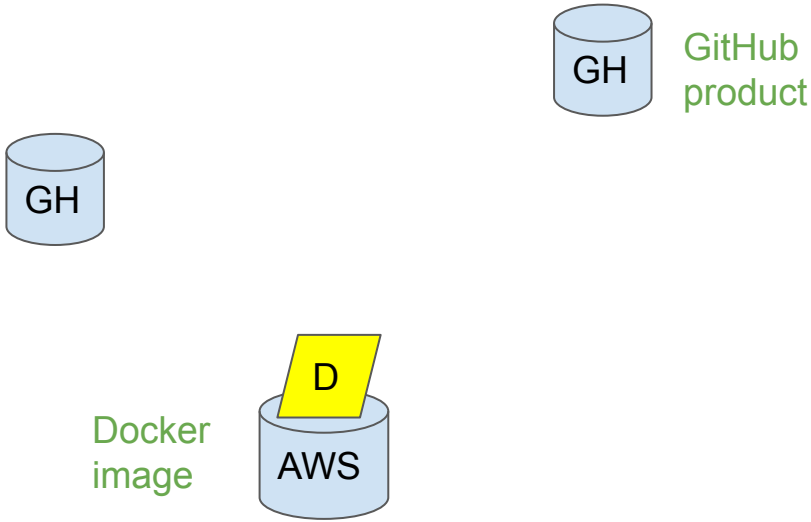
CI: GH Actions builds Docker image



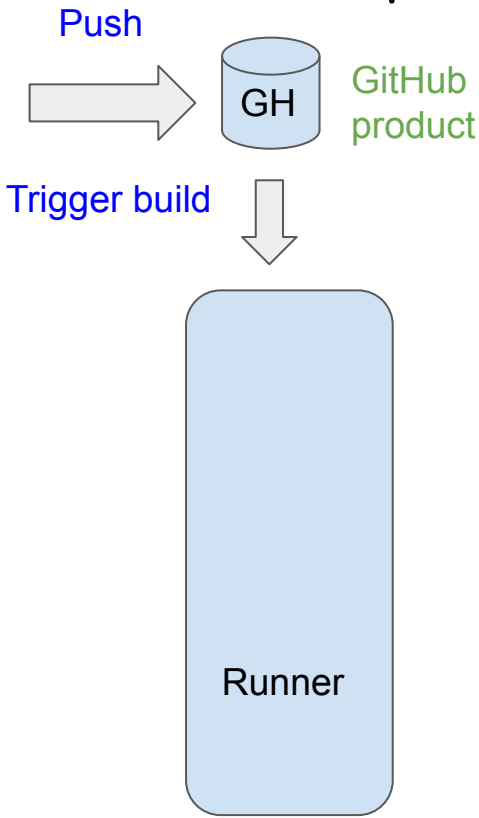
CI: GH Actions uploads image to AWS



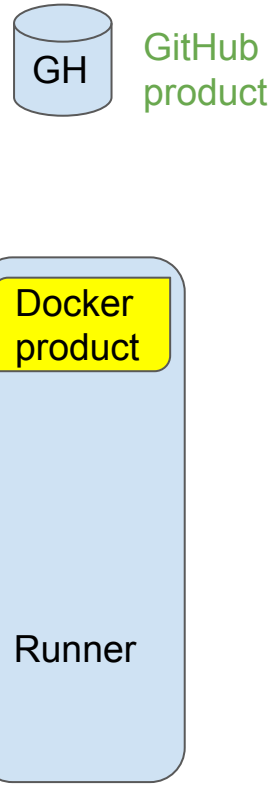
CI: AWS has latest version of TA project



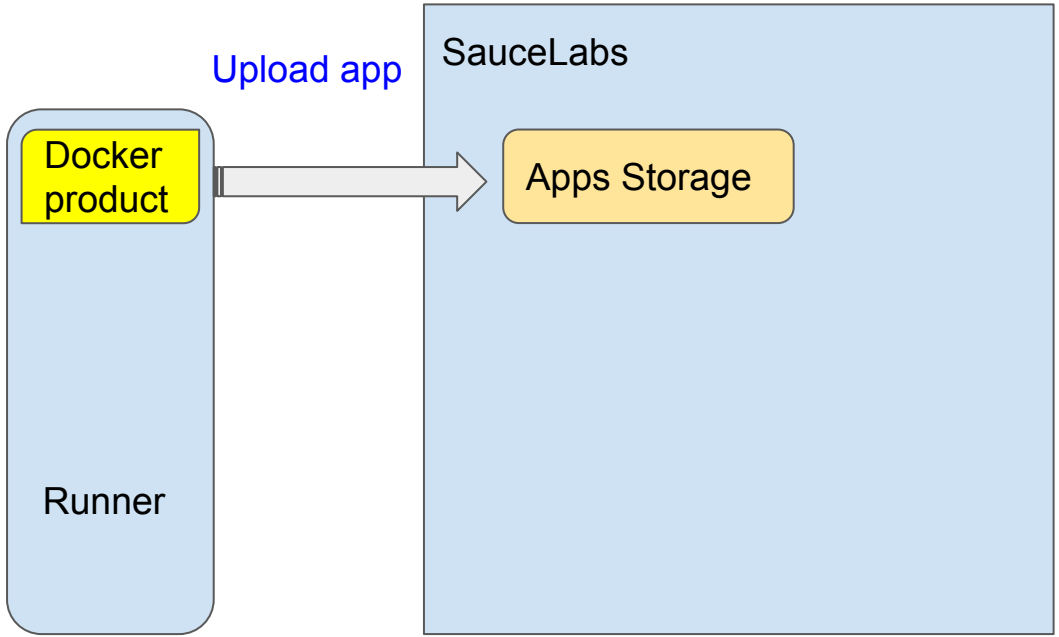
CI: Open PR in the Product repo spins up GH runner



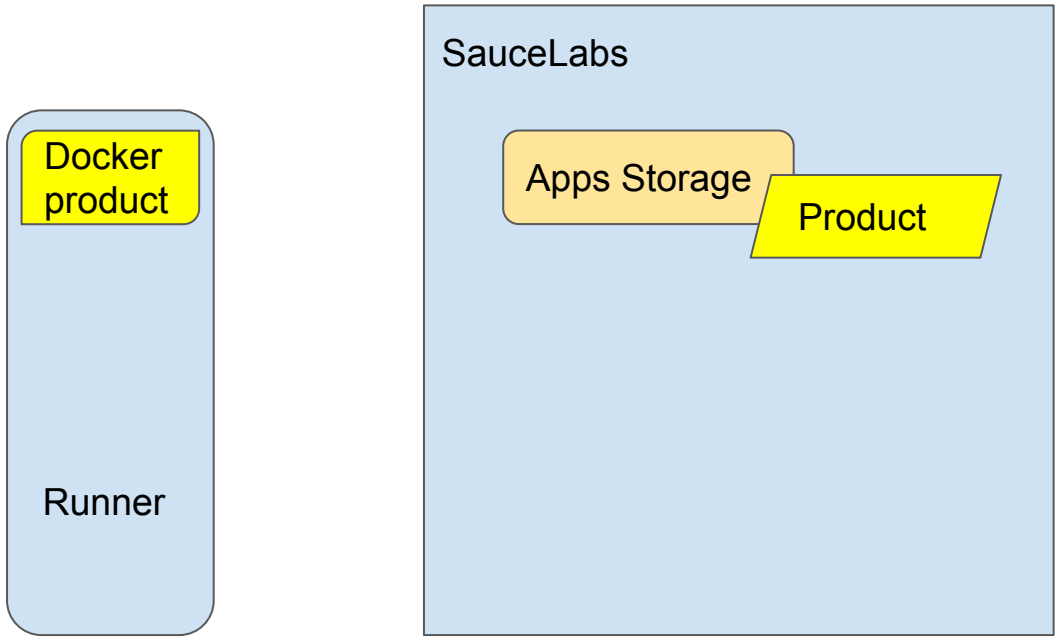
CI: Build product with macOS runner



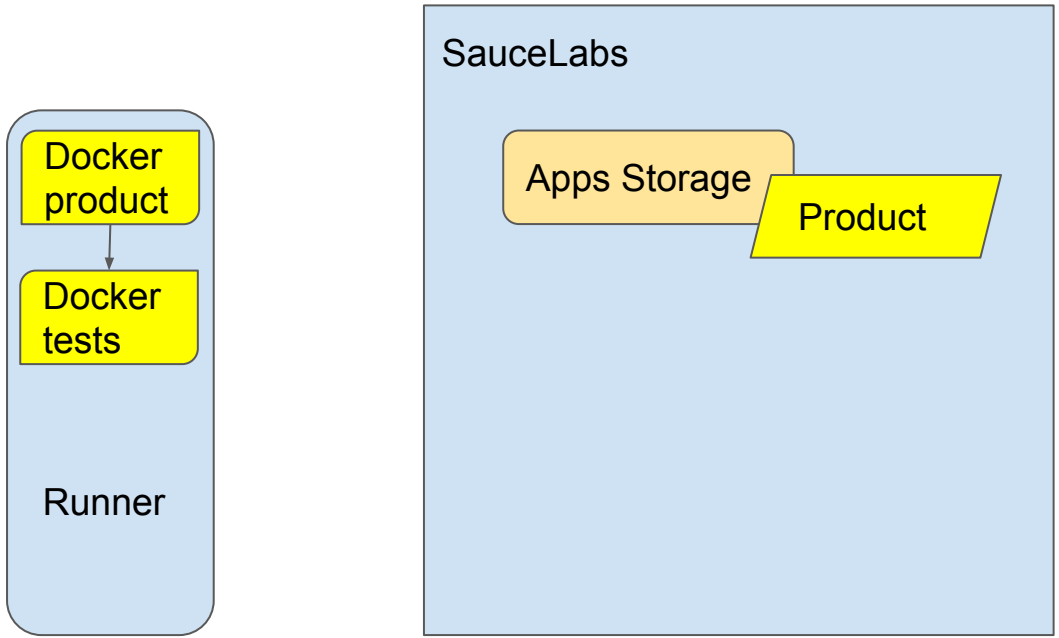
CI: Upload product to the Cloud storage



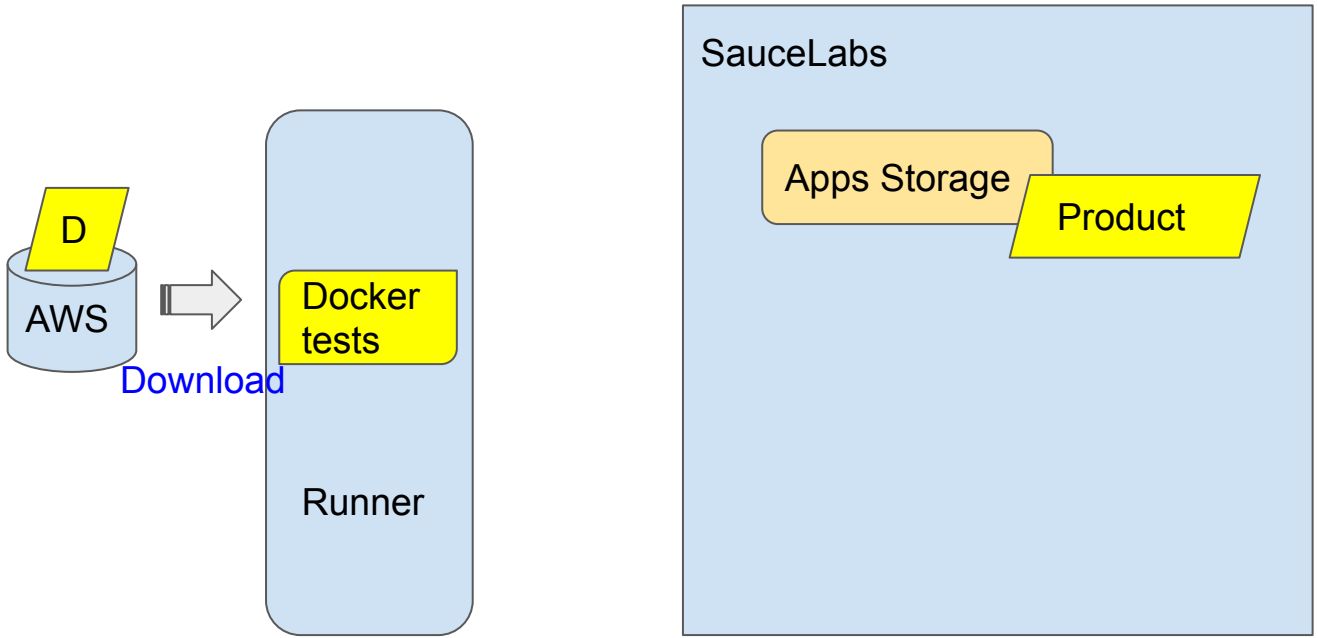
CI: Product is ready to be tested on a remote device



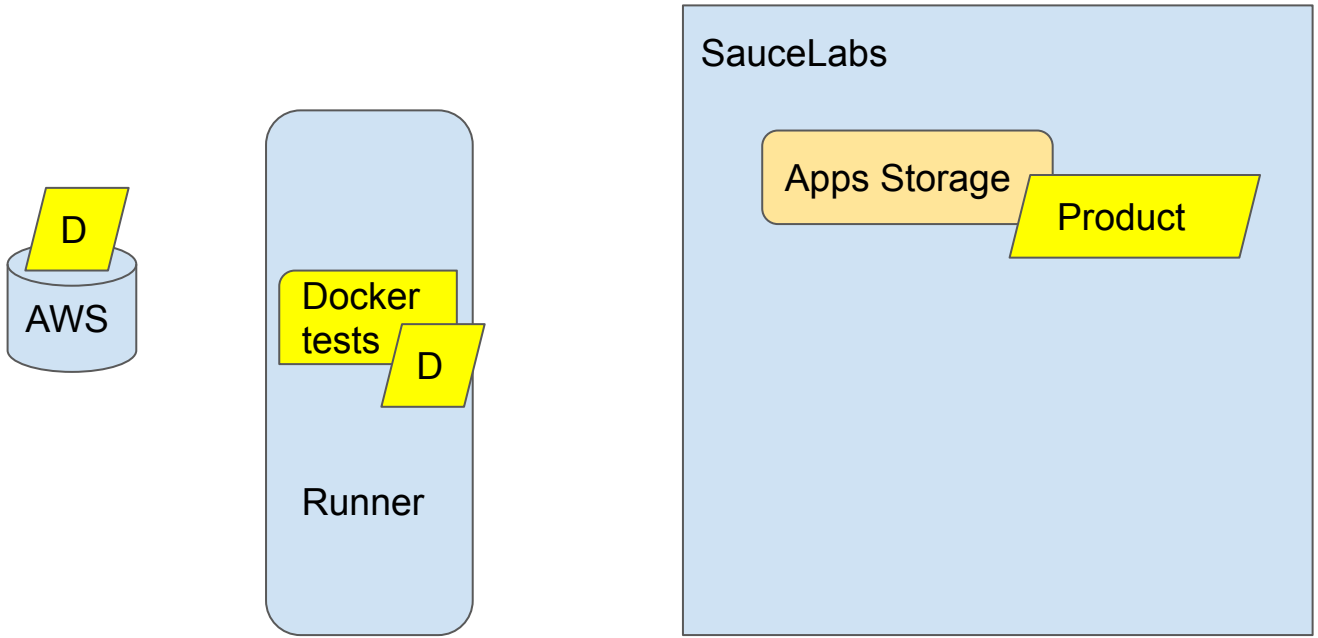
CI: Spin up tests



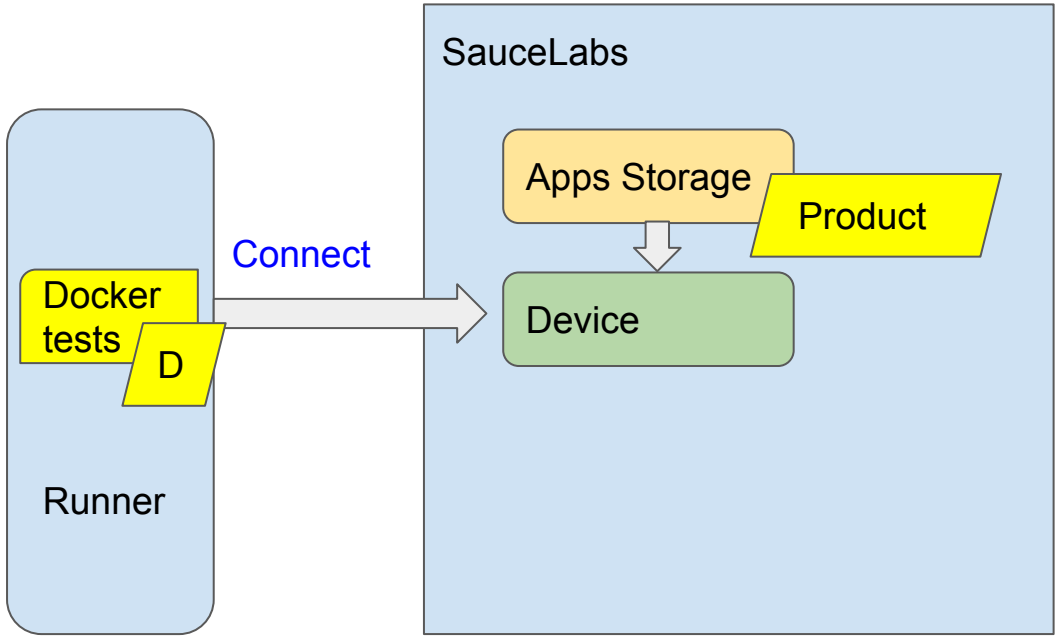
CI: Download latest tests from AWS



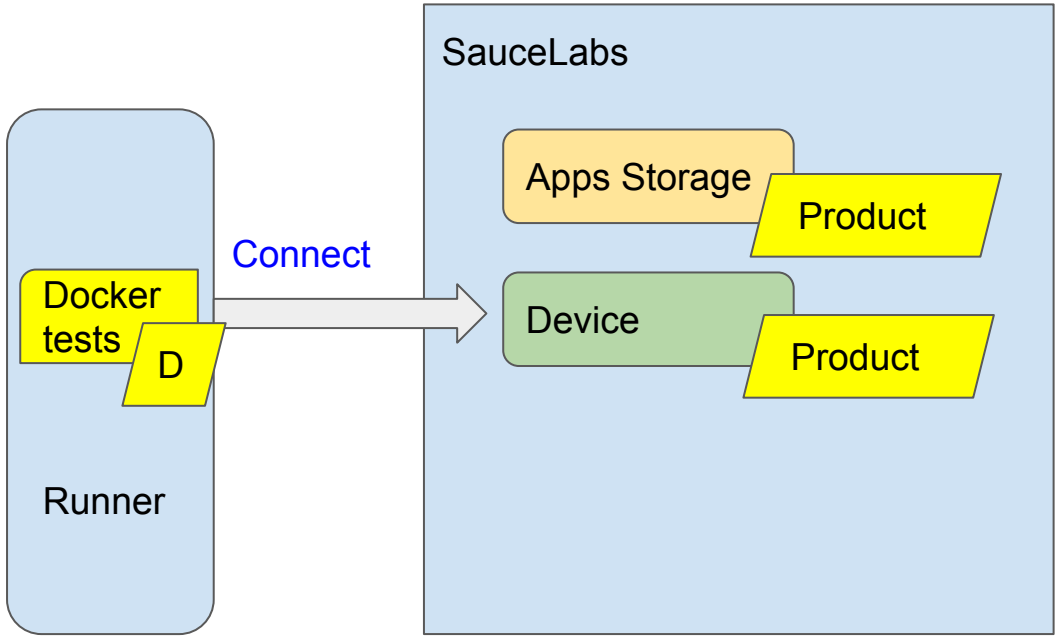
CI: Tests are ready to run in the Docker container



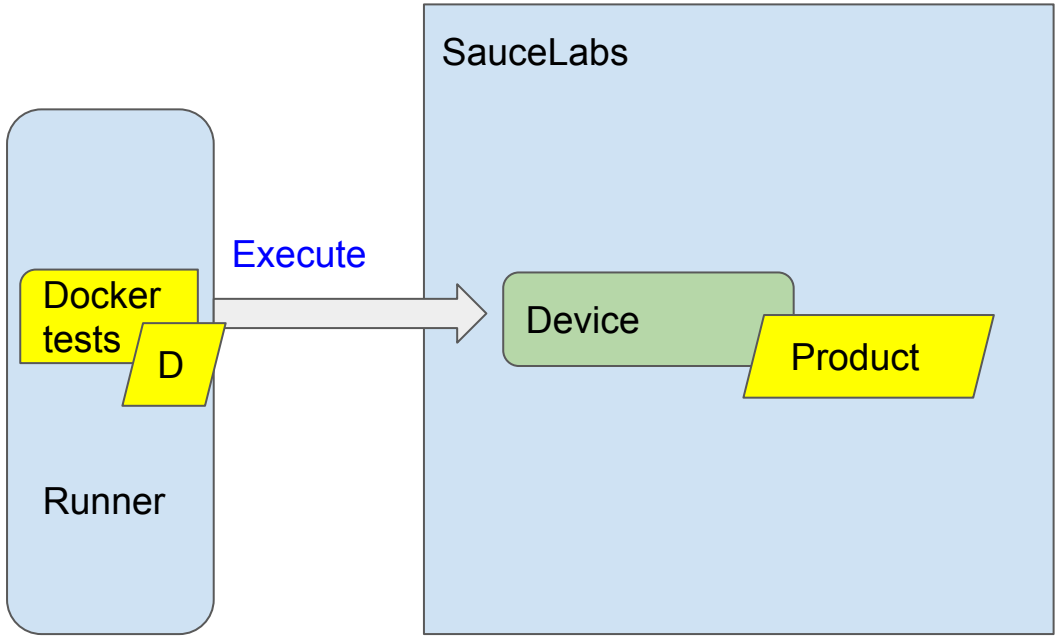
CI: Connect to SL and install the product to a device



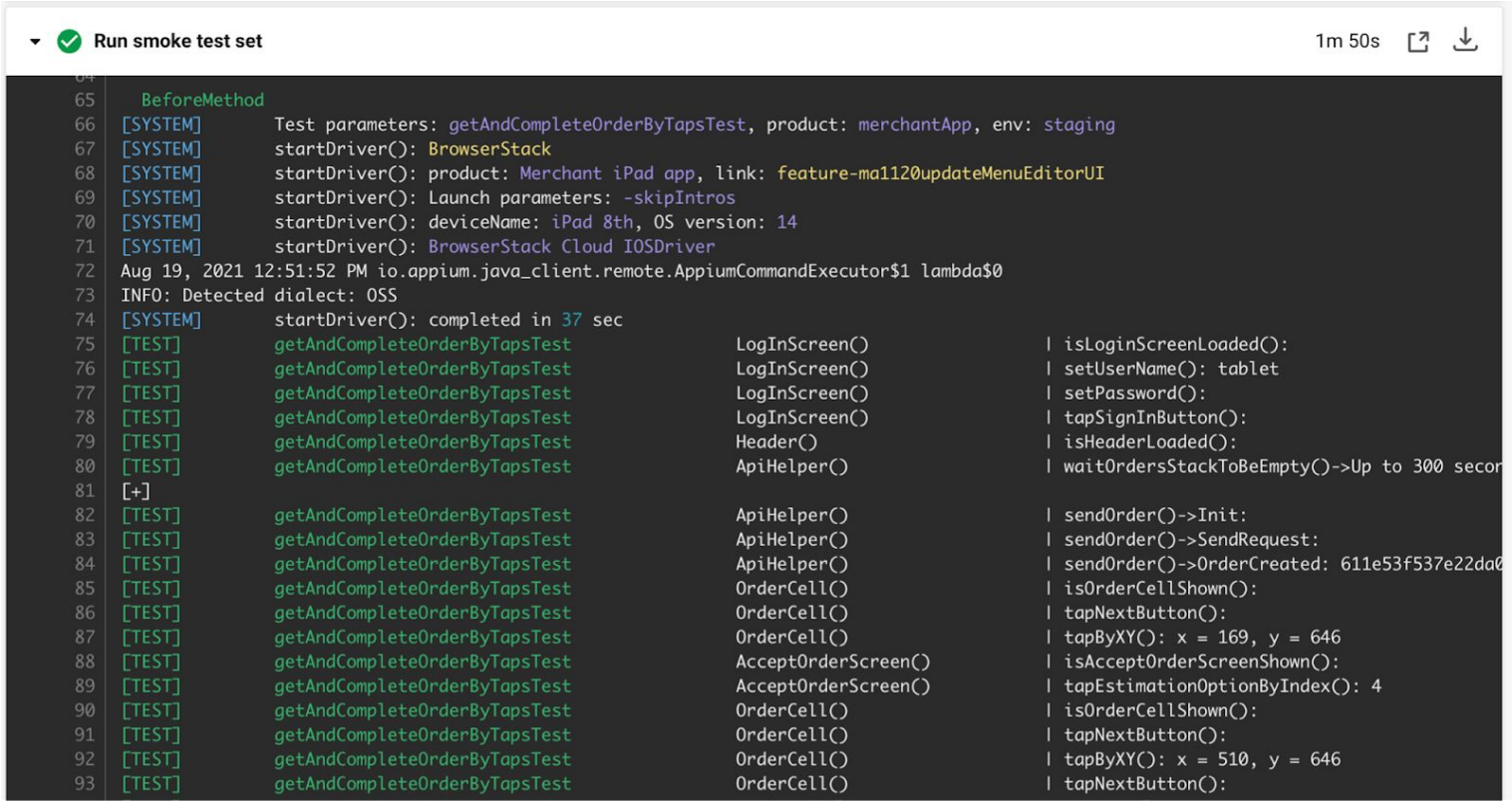
CI: Cloud device is ready to run the test



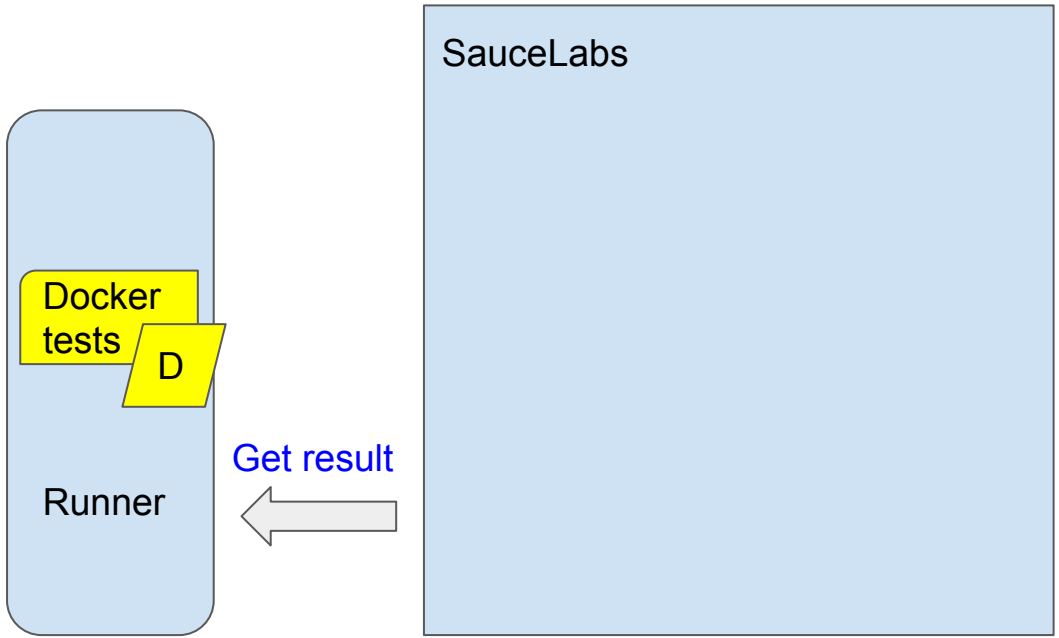
CI: Running tests



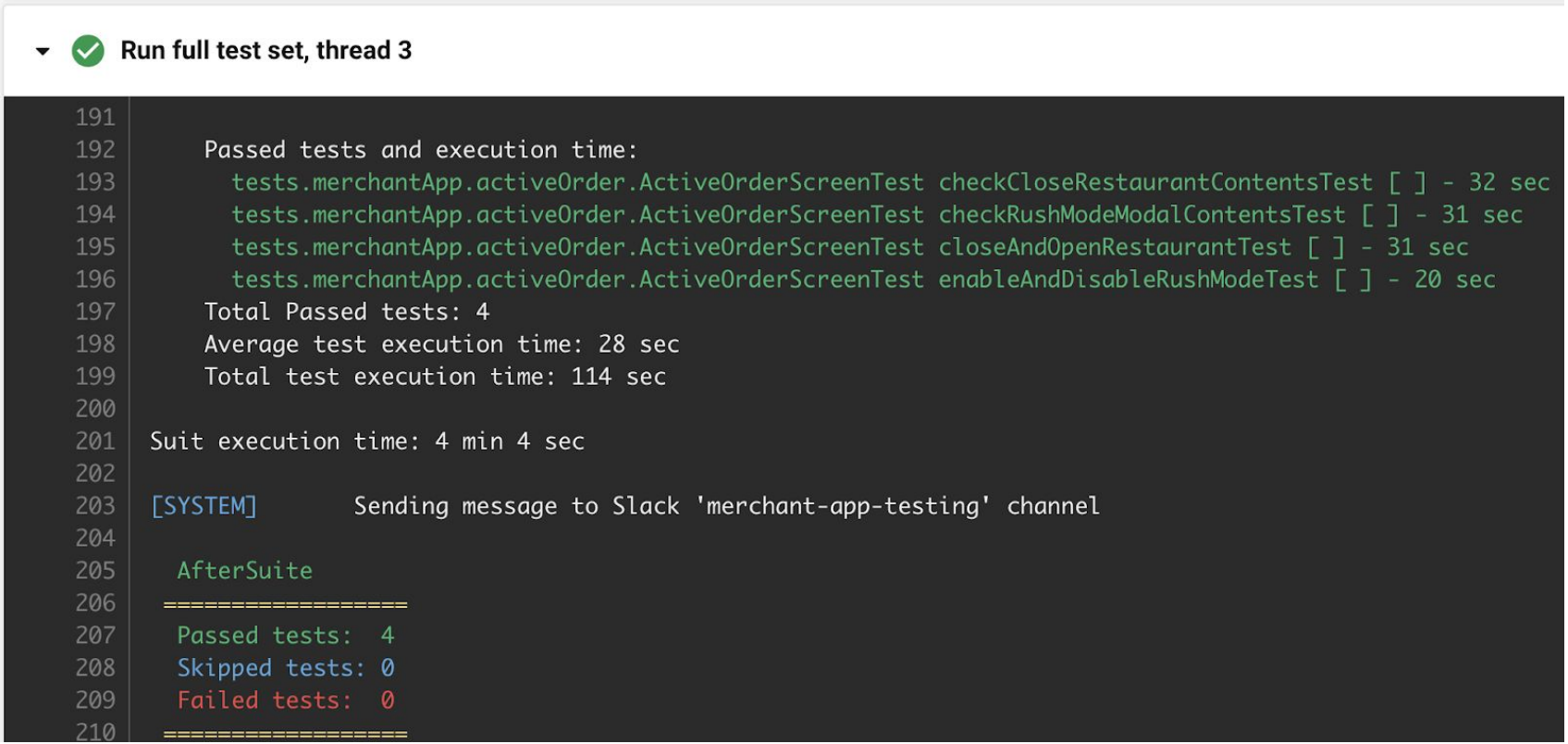
Gh Workflow: View test logs



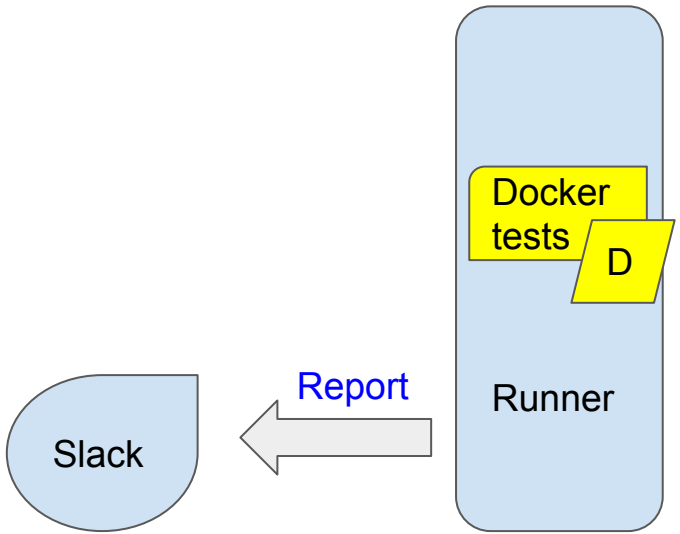
CI: Get test results



GH Workflow: Job result



CI: Send a message to the Slack channel



CI: General plan

CI: Going deeper

How to configure all this stuff in practice

GitHub Actions for the Repo with tests

Idea:

Goal: create Docker image with everything needed to run the test in the container

What is needed:

- Base image with Java and Maven
- TA project files from repo
- Test data (logins, passwords): not stored in the repo, get from the GH Secrets
- All libraries in the project (Appium, TestNG etc.)
- Pack everything to the image

Result:

- Ready-to-use image
- Guaranteed stable and reliable

GitHub Actions

Implementation:

- Set secrets with the sensitive test data

- Create GitHub Actions configuration file

- Check results on GitHub: Action tab

Docs: https://docs.github.com/en/actions

GH Actions: Secrets

- Options
- Manage access
- Security & analysis
- Branches
- Webhooks
- Notifications
- Integrations
- Deploy keys
- Autolink references
- Actions
- Secrets

Actions
Dependabot

Actions secrets

New repository secret

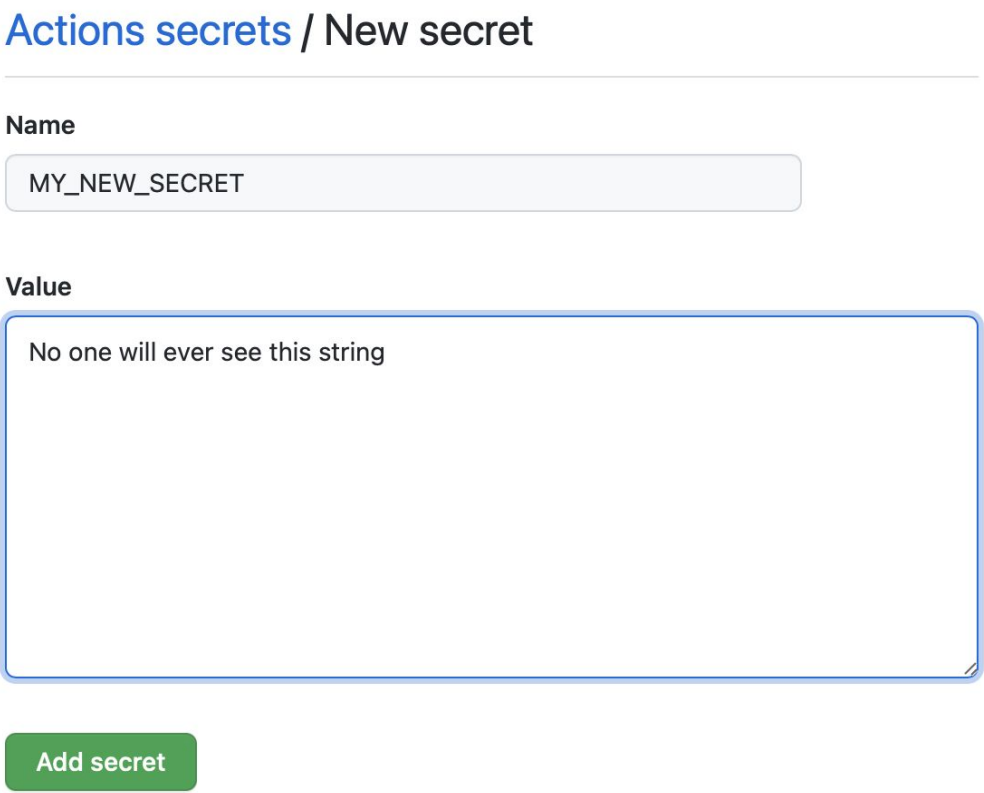
Secrets are environment variables that are **encrypted**. Anyone with **collaborator** access to this repository can use these secrets for Actions.

Secrets are not passed to workflows that are triggered by a pull request from a fork. Learn more.

Repository secrets

CLOUD_USERS	Updated on 7 Jul	Update	Remove
ECR_AWS_ACCESS_KEY_ID	Updated on 20 Apr	Update	Remove
ECR_AWS_SECRET_ACCESS_KEY	Updated on 20 Apr	Update	Remove
JENKINS_CREDENTIALS	Updated on 20 Apr	Update	Remove
SLACK_ACCESS_KEY	Updated on 23 Jul	Update	Remove
USERS	Updated on 28 Jul	Update	Remove

GH Actions: Creating a new secret



GH Actions: Configuration file

```
name: GitHub Actions Demo

on: [push]

jobs:

  Explore-GitHub-Actions:

    runs-on: ubuntu-latest

    steps:

      - name: List files in the repository

        run: |

          ls ${{ github.workspace }}
```

GitHub Actions: Import test data from Secrets to the image

```yaml
on:
  push:
    branches:
      - main

env:
  AWS_REGION: eu-west-1
  IMAGE_VERSION: 1.0
  PROJECT: ${{ github.event.repository.name }}

jobs:
  build:
    name: Build and publish docker image
    runs-on: self-hosted
    steps:

      - name: Checkout code
        uses: actions/checkout@v2

      - name: Create file users.json
        run: |
          touch users.json

      - name: Fill users.json
        run: |
          echo ${{ secrets.USERS }} | base64 -d > users.json
```

GitHub Actions: Build the image

```yaml
- name: Configure AWS credentials
  uses: aws-actions/configure-aws-credentials@v1
  with:
    aws-access-key-id: ${{ secrets.ECR_AWS_ACCESS_KEY_ID }}
    aws-secret-access-key: ${{ secrets.ECR_AWS_SECRET_ACCESS_KEY }}
    aws-region: ${{ env.AWS_REGION }}

- name: Login to Amazon ECR
  id: ecr-login
  uses: aws-actions/amazon-ecr-login@v1

- name: Build Docker image
  env:
    ECR_REGISTRY: ${{ steps.ecr-login.outputs.registry }}
  run: docker build -t $ECR_REGISTRY/$PROJECT:$IMAGE_VERSION .

- name: Push Docker image
  env:
    ECR_REGISTRY: ${{ steps.ecr-login.outputs.registry }}
  run: docker push $ECR_REGISTRY/$PROJECT:$IMAGE_VERSION

- name: Logout from Amazon ECR
  if: always()
  run: docker logout ${{ steps.ecr-login.outputs.registry }}
```

Upload app to the cloud storage

We have tests already in the cloud, but we have nothing to run yet.

Upload app file to the Cloud device farm storage

```
curl -u "USERNAME:ACCESSKEY"

-X POST "https://api.eu-central-1.saucelabs.com/v1/storage/upload"

-F "file=@/Users/MyUser/Documents/apps/product.apk"

-F 'data={"custom_id": "my_awesome_android_app"}'
```

custom_id - Unique name of your app in the cloud storage. With it one can have multiple versions of the same app.

Fastlane

Fastlane is a tool to simplify and automate app deployments, especially mobile ones

Web site: https://fastlane.tools

```
platform :ios do
  desc "Builds, achieves and uploads ipa to Test server"
  lane :upload_lane do
    # Download repo
    # Build release app
    # Upload to Test server
  end
end
```

Using Fastlane to upload a file to the SauceLabs

```
desc "Upload build to SauceLabs"

lane :uploadToLfromCI do |options|

    ipa_path = "/Users/distiller/project/output/gym/myProduct.ipa"

    saucelabs_app_name = options[:cloudLink]

    curl = "curl -u \"" + saucelabs_login + "\" -X POST \"" + saucelabs_url + "\" -F \"file=@" + ipa_path + "\" -F
\'data={\"custom_id\": \"" + bsaucelabs_app_name + "\"}\'"

    UI.message(curl)

    app_upload_response = sh(curl)

    UI.message(app_upload_response)

    end
```

Checkpoint

We have:

- Product repo with build workflows
- Image with tests

Next:

- Extend product workflow to upload a new build to the cloud storage
- Extend product workflow to run tests in the image over new build

CI configuration: initial state

Product workflow

when: Development event: Merge to Master, Create release, Open PR to review

run:

- Build app
- Run unit test
- Deploy to App distribution platform (TestFlight, FireBase)

CI configuration: Add workflow to run tests

Product workflow

when: Development event: Merge to Master, Create release, Open PR to review

run:

- Build app
- Deploy to App distribution platform (TestFlight, FireBase)
- Upload app with **Fastlane** to the Device farm cloud storage with a specific *app name*

Test workflow

when: triggered by a product workflow or manually

using: test suite, app name

run: test suite on the product with *app name*

CI configuration: Final product workflow

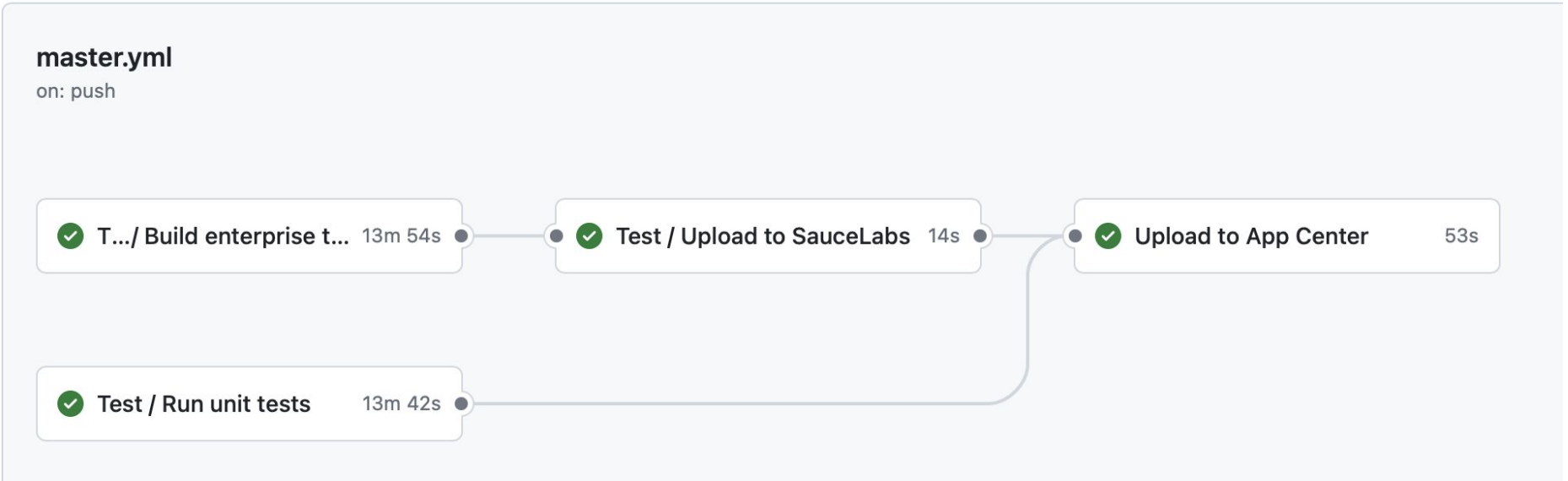
Product workflow

when: Development event: Merge to Master, Create release, Open PR to review

run:

- Build app
- Deploy to App distribution platform (TestFlight, FireBase)
- Upload app to the Device farm cloud storage with a specific **app name**
- Run “Test workflow” with event specific **suite** over **app name**

GitHub workflow



CI configuration: extra

- Build app
- Deploy to App distribution platform (TestFlight, FireBase)
- Upload app to the Device farm cloud storage with a specific **app name**
- Run “Test workflow” with event specific **suite** over **app name**

In any moment of time we have:

- Latest and intermediate products build in the cloud storage
- Everyone can try any build though SauceLabs Live session
- We can re-run any test suite on any previous build

Reporting to Slack

What might be useful to add to the report:

- Status (everything is ok or there are issues)
- Run details: GitHub branch, environment, test suite
- Link to the GitHub workflow
- Link to the SauceLabs sessions recording
- Link to the TMS run
- In case of test failures: exception details

Test run details (short)

autoqa_notifications APP 2:07 PM

autoqa for Merchant iPad app: PASSED ✅

Suite:
Merchant iPad app full set - Thread 3

Device:
iPad 8th, OS version: 14

Branch:
MerchantAppMaster

CI pipeline:
Job

Tests passed: 4 / 4

Suite execution time: 4 min 24 sec

browserstack

Test run details (Extended)

AutoQA APP 5:02 AM — Tuesday, May 7th

Merchant App Nightly full regression on iOS 16: FAILED ❌

Device:	Source:
iPad, OS version: 16	merchant-app-master.ipa

self delivery order	**Passed: 2/2 ✅**
weight based orders	**Passed: 1/1 ✅**
orders	**Passed: 4/5 ❌**
deliveryWithFixedPriceOrderTest	*Failed*
deliveryWithFixedPriceOrderTest	*Skipped*
purchases archive	**Passed: 2/2 ✅**
active order screen	**Passed: 4/4 ✅**
settings	**Passed: 3/3 ✅**
side menu	**Passed: 2/2 ✅**
intercom	**Passed: 1/1 ✅**
order context menu	**Passed: 2/2 ✅**
authorisation	**Passed: 1/1 ✅**

Execution on device farm

Workflow in GitHub

Test run on Qase

1 reply 3 days ago

Test run details: List of failed tests

1 reply

AutoQA APP 2 months ago

weight based orders

❌ *getAndCompleteWeightedOrderTest*

```
OrderModal not loaded expected [true] but found
[false]
```

⏩ *getAndCompleteWeightedOrderTest*

```
OrderModal not loaded expected [true] but found
[false]
```

order context menu

❌ *verifyRejectOrderOverlayTest*

```
MoreActionsMenu not loaded expected [true] but found
[false]
```

Slack: How to set up Slack reporting

- Create Slack-bot to send messages and configure its access right

- Create channels for notifications

- Implement code to compose JSON message with test details and send it to specific Slack channel with Slack API

Slack: Method in the TA repo to send a message

```
fun sendMessage(channel: SlackChannels, message: String, isMarkDown: Boolean = true) {
    sysLog( text: "Sending message to Slack '${channel.description}' channel")

    val payload = JSONObject()
    payload["channel"] = channel.value
    if (isMarkDown) {
        payload["text"] = "autoqa notification"
        payload["blocks"] = message
    } else {
        payload["text"] = message
    }

    val request = Request.Builder()
        .url(OutsourcedURLs.SLACK_POST_MESSAGE.url)
        .header( name: "Authorization",  value: "Bearer $token")
        .addHeader( name: "content-type",  value: "application/json")
        .post(payload.toString().toRequestBody())
        .build()

    client.newCall(request).execute().use { response ->
        val responseJson: JsonNode = ObjectMapper().readTree(response.body?.string()?.trim())
        Assert.assertTrue(
            responseJson.get("ok").asBoolean(),
            "${TAG}sendMessage(): Failed sending message to channel '${channel.name}'"
        )
    }
}
```

CI: Final flow

Repo with automated tests project:

- Build image with tests when merging to master
- Publish image to AWS ECR to make it available for other repos in the Company

CI: Final flow

Product repo:

- Build the app
- Upload app to the Cloud device farm storage
- Download latest image with tests
- Run tests over current build
- Publish results

Useful links

Idea: https://www.jetbrains.com/idea/

Maven: https://maven.apache.org

Appium: https://appium.io

TestNG: https://testng.org/doc/index.html

Docker: https://www.docker.com

Fastlane: https://fastlane.tools

GitHub Actions: https://github.com/features/actions

SauceLabs: https://saucelabs.com/

Slack API: https://api.slack.com/messaging/sending

What tools was used today

Mobile OS: iOS and Android

Programming language: Kotlin

Project management: Maven

Test management: TestNG

Test automation driver: Appium

Test project development pattern: PageFactory

Code storage: GitHub with GitHub Actions

CI: GitHub + Docker

Device cloud: SauceLabs

Part 6: Bonus topics

Wolt

Bonus topic 1: Text logs

Text logs with explicit log message

```
fun setUserName(userName: String): LogInScreen = also { it: LogInScreen
    log( text: TAG + "setUserName(): " + userName)
    Assert.assertTrue(sendKeys(userNameInput, userName), "${TAG}setUserName(): FAILED")
}
```

Use Reflection with AspectJ to print method detain in runtime

```
@Step("Set user name: '{userName}'")
fun setUserName(userName: String): LogInScreen = also { it: LogInScreen
    Assert.assertTrue(sendKeys(userNameInput, userName), "setUserName(): FAILED")
}
```

Bonus topic 1: Text logs

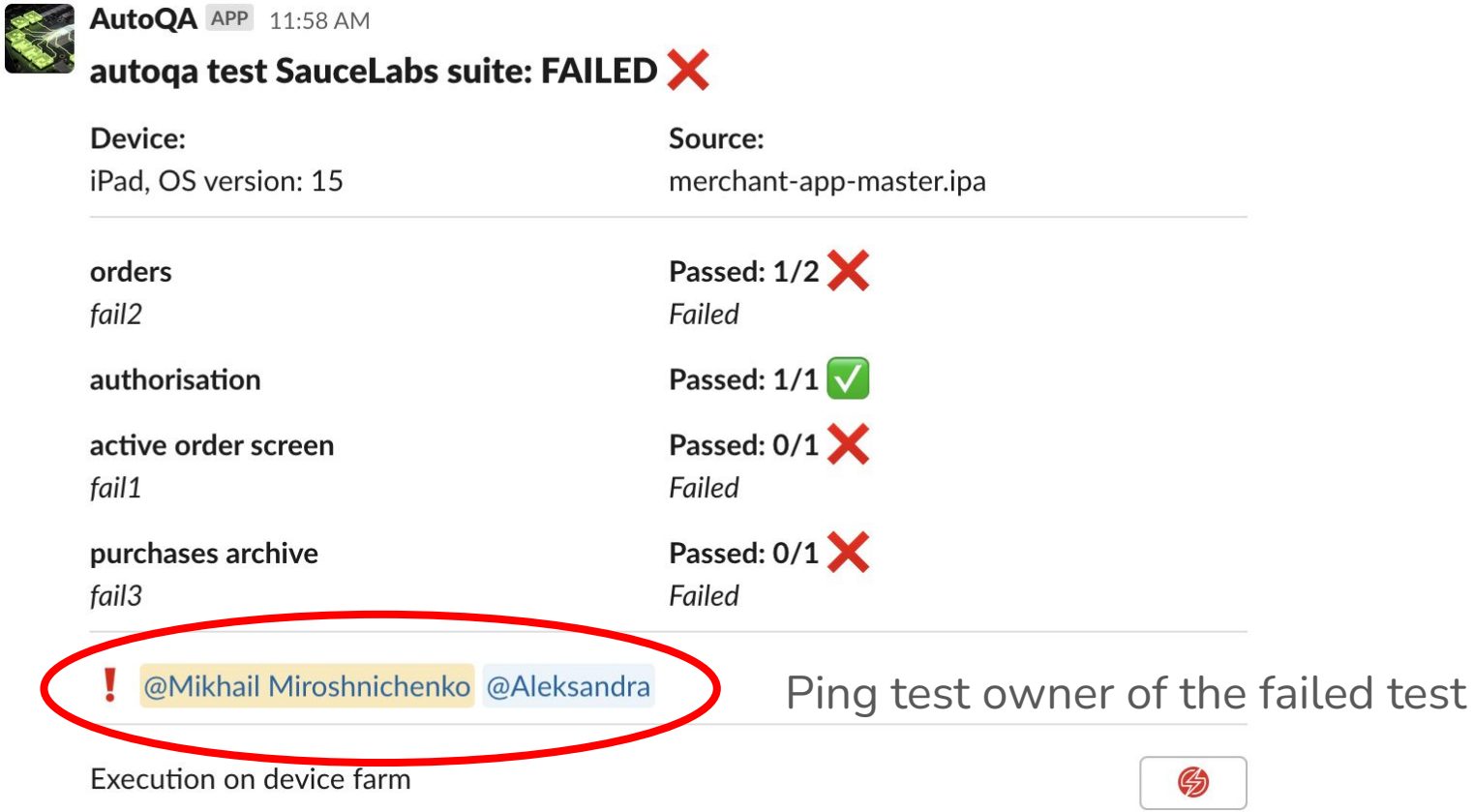
```
@Aspect
class Aspects {

    /** If test method has TestOwner annotation, add this method to the lost of tests with test owners ...*/
    @Before("@annotation(app.constants.TestOwner)")
    fun logOwner(joinPoint: JoinPoint) {...}

    /**
     * For methods with Step annotation from screens package.
     * Print testName-ScreenClass-MethodName-ParametersList in the text log
     */
    @Before("@annotation(io.qase.api.annotation.Step) && execution(* screens.*.*.*.*(..))")
    fun stepMethod(joinPoint: JoinPoint) {
        val methodName = getMethodName(joinPoint)
        if (methodName != "isScreenLoaded" && methodName != "isScreenShown") {
            val parametersList = getMethodParameters(joinPoint)
            val passwordFieldIndex = getMethodArguments(joinPoint).indexOf("password")
            if (passwordFieldIndex != -1)
                parametersList[passwordFieldIndex] = "***"
            log( text: "${setTag(getParentClassName(joinPoint))}$methodName $parametersList")
        }
    }
```

Bonus topic 2: Test owners and test groups

1. Add a test owner to notify them by Slack ping if this test failed
2. Add Group property: to group tests in test reports or compose suites

```
@TestOwner(TestOwners.MERCHANTAPP)
@CaseTitle("Get order, check details and complete it by moving cell")
@Test(
    description = "Get order, check details and complete it by moving cell",
    groups = ["ios", Components.MerchantApp.ORDERS]
)
fun getCheckAndCompleteOrderTest() {
```

Bonus topic 2: Test owners and test groups



Ping test owner of the failed test

Bonus topic 3: Reporting to Qase TMS

Link automated tests with test-cases in the Qase TMS:

- Link existing test-cases with automated ones
- Create test-cases based on automated tests and their steps
- Create and complete test runs in the TMS when running automated tests

Bonus topic 3: Qase configuration

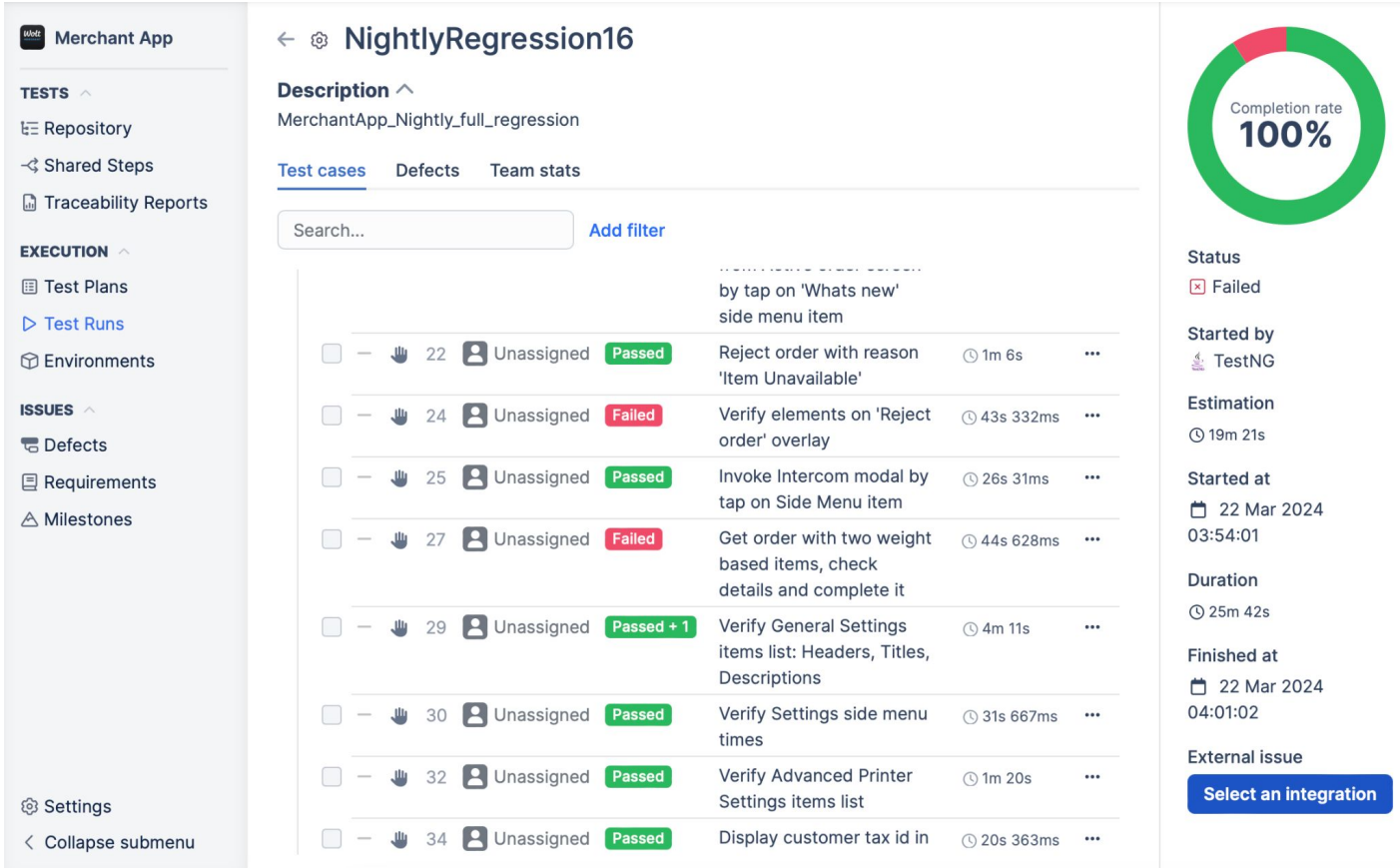
Configuration:

1. Add library to the project dependencies
2. Define tests with @Test, @CaseTitle or @CaseID annotations

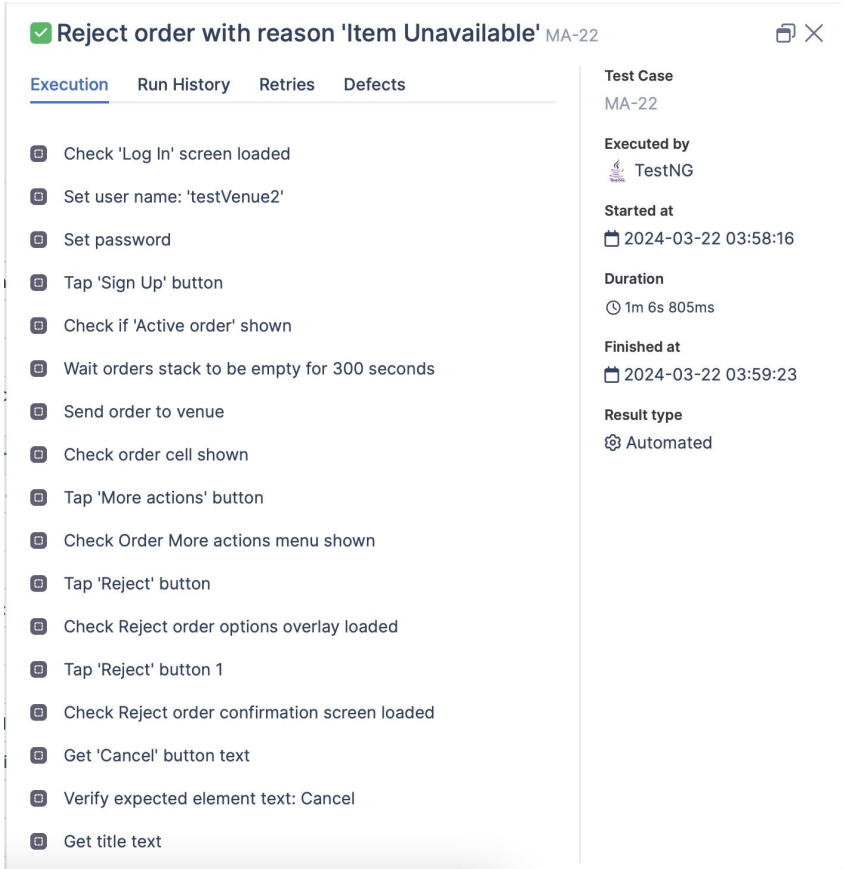
Technical details:

1. Work as a library, available to import with Maven
2. Uses TestNG or JUnit annotation to gather test result
3. Manages test runs automatically
4. Has Qase API to control everything

Bonus topic 3: Qase Test list in the Test run



Bonus topic 3: Qase Test steps



Want to know more about mobile testing?

Building great mobile apps: Tips and considerations

https://careers.wolt.com/en/blog/tech/building-great-mobile-apps-tips-and-considerations

Thank you!

Wolt