UI Автоматизация тестирования мобильных приложений

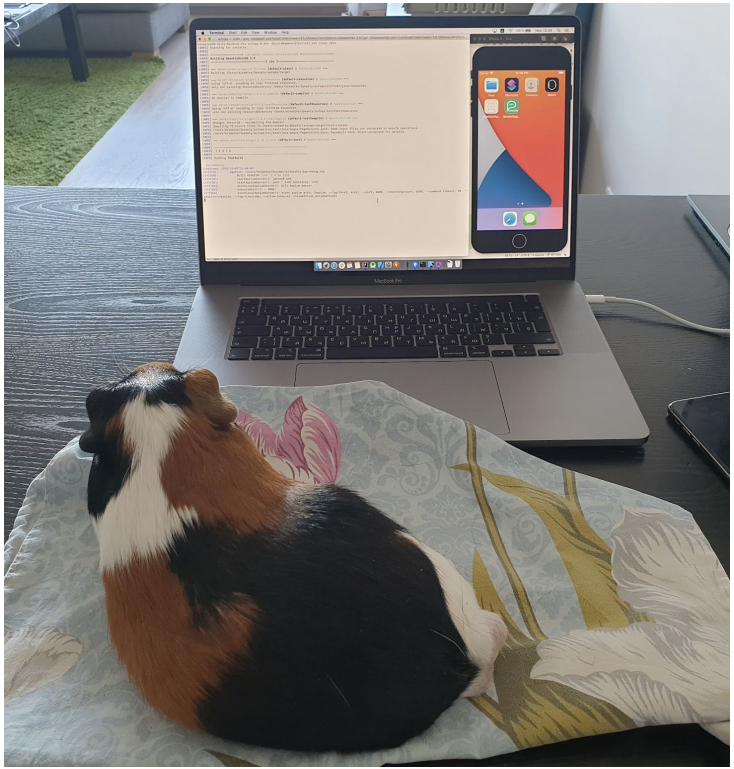
Мирошниченко Михаил

Wolt

hikari.no.mikem@gmail.com

https://www.linkedin.com/in/miroshnichenkomichael/

Обо мне: Мирошниченко Михаил



15 лет в QA

8 лет в Mobile

4 года в Test Automation

3 года в Эстонии

QA Automation в **Wolt** (FI)

Тестировал в:

- Яндекс (RU)
- Доктор Веб (RU)
- Bolt (EE)

Как сделать автоматизацию для мобильных приложений: от выбора инструментов до выполнения тестов в CI

На примере того, как автотесты работают в Wolt Merchant App

Первая серия: Сборка проекта автотестов на GitHub

После добавления тестов в проект с автотестами происходит сборка Docker образа на GitHub

creditornot / autoqa Private

Watch 5 Star 0 Fork 0

Code Issues Pull requests Actions Projects Wiki ...

MA-1159: Modify Merchant App settings with API commands (#58)

main pipeline #129

Cancel workflow ...

Summary

Jobs

- Build and publish d...

Triggered via push now

mikamikaWolt pushed 88ce2d4 main

Status

Queued

Total duration

–

Artifacts

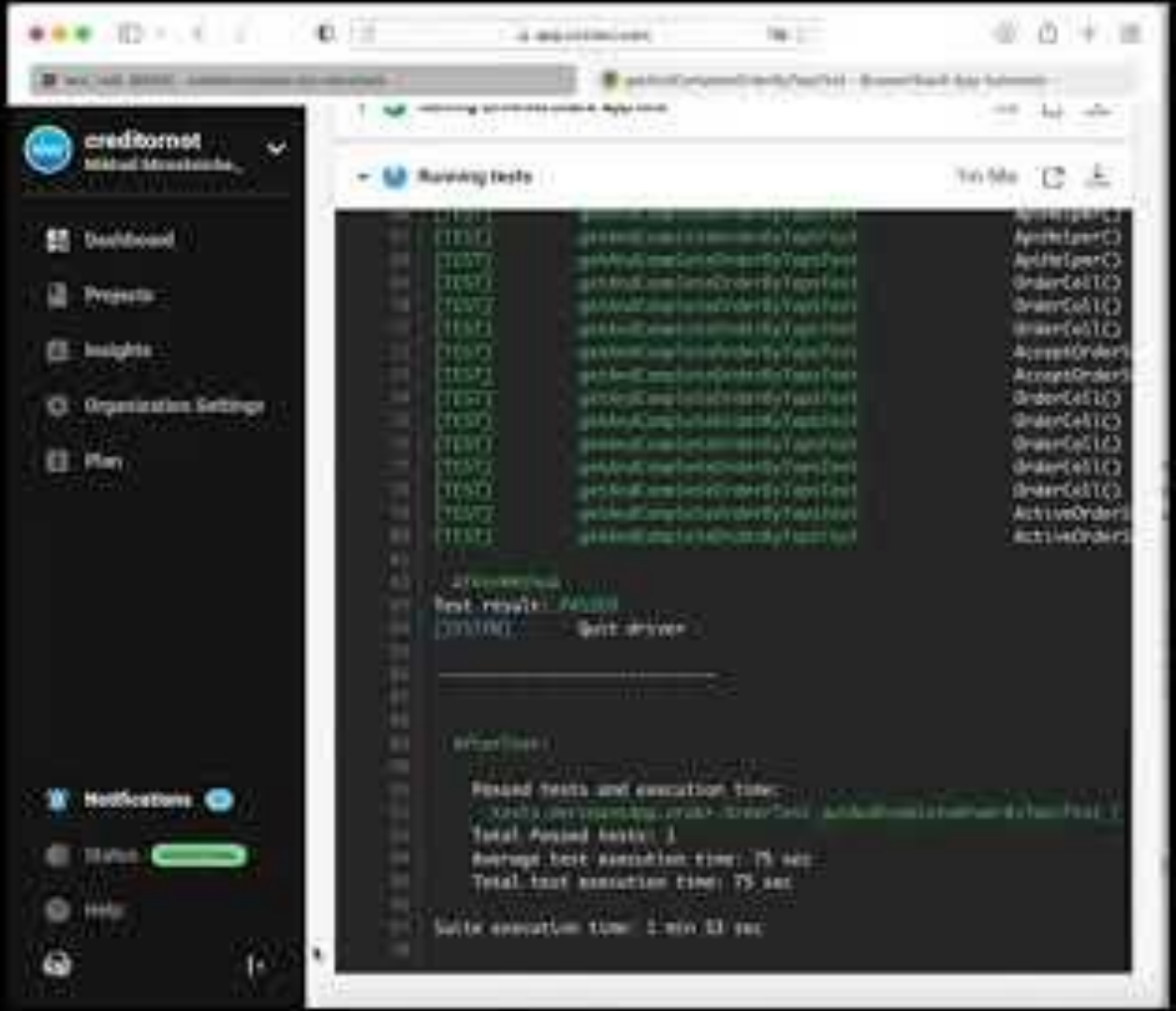
–

main.yml

Вторая серия: Прогон тестов на новом билде

При Push нового кода приложения происходит сборка его дистрибутива и прогон тестов.

Тесты выполняются на ферме с мобильными устройствами


credbornet
Dashboard
Projects
Insights
Organization Settings
Plan
Notifications
Status
Help
Running tests
Test result: PASSED
[DRIVER] Quit driver
Passed tests and execution time:
Total Passed tests: 1
Average test execution time: 75 sec
Total test execution time: 75 sec
Suite execution time: 1 min 13 sec

Сегодня сделаем такую же красоту:

Мобильная ОС: **iOS** и **Android**

Язык разработки: **Kotlin**

Управление проектом: **Maven**

Управление тестированием: **TestNG**

Драйвер для мобильных устройств: **Appium**

Паттерн разработки UI тестов: **PageFactory**

Хранилище кода и сборка проекта: **GitHub**: Actions + Secrets

Среда CI: **CircleCI** + **Docker**

Облачная ферма с устройствами: **BrowserStack**

План сегодняшней встречи

1. Теория автоматизации
2. Создания проекта автотестов в IDE
3. Создание автотестов
4. Исполнение автотестов
5. Интеграция с CI

Для чего автоматизировать тесты?

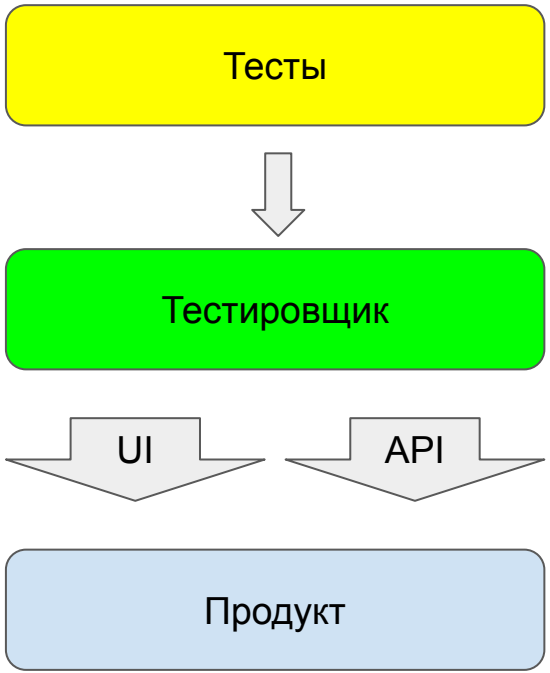
- Регрессионное тестирование
- Постоянный контроль качества во времени
- Проверить то, что долго или не возможно проверить вручную
- Быстрая обратная связь по качеству каждой сборки

Что есть?

- Код проекта в GitHub

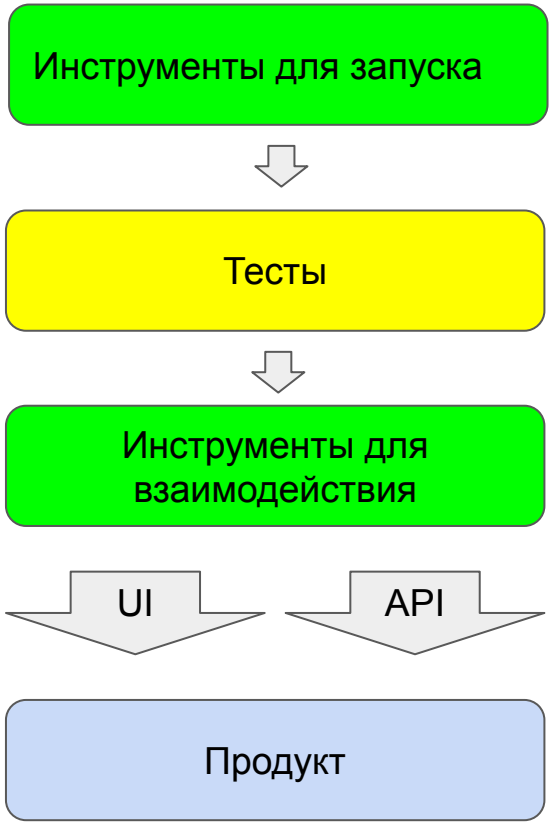
- Есть тест-кейсы

Схема ручного тестирования



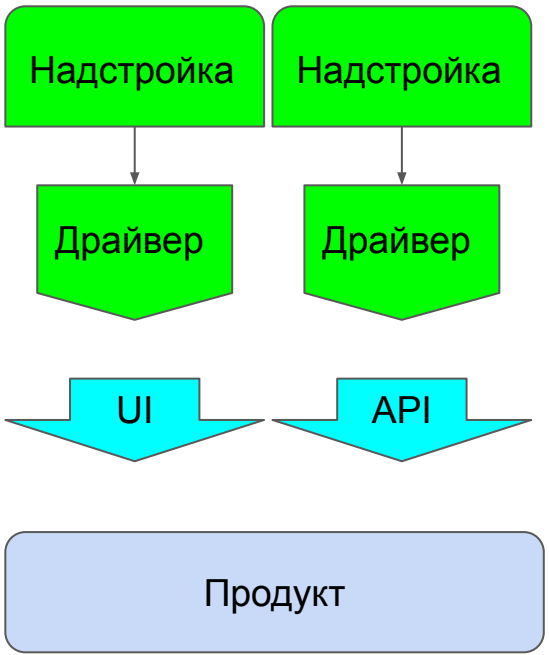
- **API** (Application Programming Interface): основной интерфейс для взаимодействия с другими программами
- **UI**: Графический интерфейс приложения для взаимодействиями с пользователями с UI автотестами

Схема автоматизированного тестирования



Инструменты для запуска и взаимодействия с приложением называются “стеком автотестирования”

Состав инструментов взаимодействия



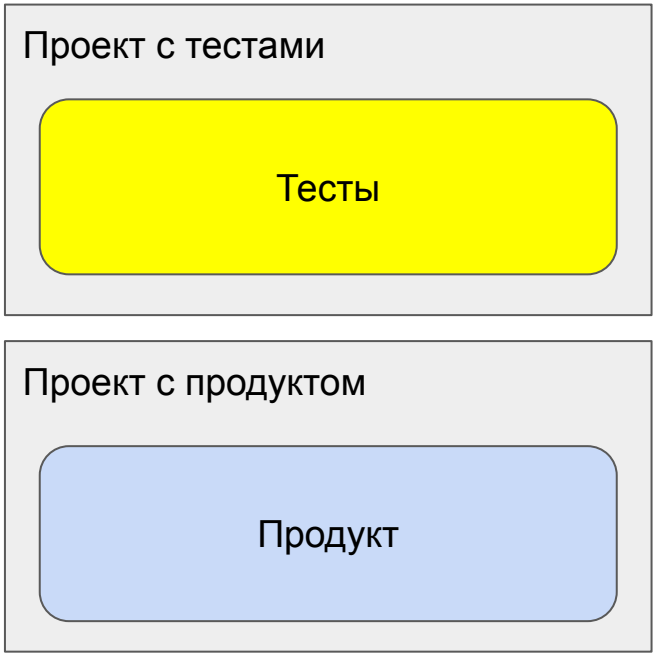
Драйвер: предоставляет программный интерфейс для одного из интерфейсов приложения

Надстройка: делает использование драйвера проще и удобнее

Автотесты: Где хранить тесты?

Внутри проекта

Как отдельный проект



Инструменты: Два подхода к автоматизации

Нативная автоматизация (Native):

- Напрямую используем встроенные драйверы автоматизации
- iOS: XCUITest
- Android: UIAutomator, Espresso
- Код приложения и код тестов - один проект

Не нативная автоматизация (Not native):

- Используем встроенные драйверы через посредника (надстройку)
- Mobile: Appium; Web: Selenium
- Тесты - отдельный проект, не зависящий от проекта приложения

Нативная автоматизация: “Белый ящик”

Плюсы:

- Можно использовать компоненты и ресурсы приложения
- Максимальные возможности доступа к элементам интерфейсы и их свойствам
- Легче модифицировать поведение приложение

Минусы:

- Работает только с приложением, для которого разработаны
- Обычно требуется использовать тот же язык, на котором написано приложение

Не нативная автоматизация: “Черный ящик”

Плюсы:

- Один проект автотестов для iOS, Android, Web
- Свобода в выборе языка и инструментов
- Компактность проекта

Минусы:

- Труднее доступ к определенным свойствам элементов интерфейса
- Немного труднее реализовать: требуется управлять двумя продуктами (приложение и автотесты)

Компоненты проекта not-native автоматизации

- **Framework**:
  - Управление порядком выполнения тестирования
  - Подготовка окружения
  - Подготовка отчетов о ходе тестирования и его результатах
  - Взаимодействие с другими приложениями
  - Предоставление стандартных функций для работы с интерфейсом приложения

- **Тесты**
  - iOS приложение
  - Android приложение
  - ...

Часть 2: Проект автотестов

Перейдем от теории к практике и начнем писать код...

С чего начать: среда разработки

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

- Бесплатная версия для некоммерческой разработки
- Поддерживает разные языки: Kotlin, Java
- Много полезных функций для быстрой разработки

Но вы можете использовать и свою любимую среду разработки! =)

Доклад:

https://www.youtube.com/watch?v=zp0qC6JT0rE

Maven: система управления проектом

Используется для:

- загрузки всех необходимых компонентов
- установки параметров тестирования
- запуска тестов на исполнение
- сбора базовой информации о результатах тестирования

Представляет собой: Файл pom.xml

Сайт: https://maven.apache.org

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

Maven: пример файла pom.xml

```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: добавление библиотек в проект

```
<!-- 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: подключение новых библиотек

Сайт: https://mvnrepository.com - используется для поиска библиотек

Пример кода для подключения новой библиотеки:

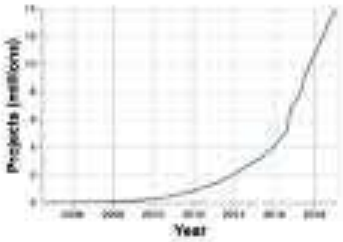
```
<!-- https://mvnrepository.com/artifact/io.appium/java-client -->
<dependency>
    <groupId>io.appium</groupId>
    <artifactId>java-client</artifactId>
    <version>7.3.0</version>
</dependency>
```

MVNREPOSITORY

appiu

Search

Indexed Artifacts (23.4M)



Popular Categories

- Aspect Oriented
- Actor Frameworks
- Application Metrics
- Build Tools
- Bytecode Libraries
- Command Line Parsers
- Cache Implementations
- Cloud Computing
- Code Analyzers
- Collections
- Configuration Libraries

What's New in Maven

SLF4J API Module — **50,756** usages

org.slf4j » slf4j-api » 2.0.0-alpha5 MIT

The slf4j API

Last Release on Aug 30, 2021

Deps Alpha — **64** usages

org.clojure » tools.deps.alpha » 0.12.1036 EPL

Deps Alpha

Last Release on Aug 30, 2021

OSGi LogService Implemented Over SLF4J — **26** usages

org.slf4j » osgi-over-slf4j » 2.0.0-alpha5 MIT

OSGi LogService implementation over SLF4J

Last Release on Aug 30, 2021

Maven: Как добавить тесты?

Maven может собрать ваш проект, но как сказать ему, что ваши классы содержат тестовые методы?

Поможет **TestNG**!

TestNG: инструмент управления тестами

Позволяет с помощью suite.xml-файлов:

- настраивать параметры тестирования
- отделить параметры тестирования от параметров сборки проекта автотестов
- расширять возможности Maven

Задает:

- Действия перед началом и после окончания тестирования, до и после каждого теста
- Какие классы и функции являются запускаемыми тестами

TestNG: Пример suite файла

```
<!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: Параметры тестирования

```
<!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: Список тестовых классов и групп тестов

```
<!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>
```

Драйвер автоматизации: Appium

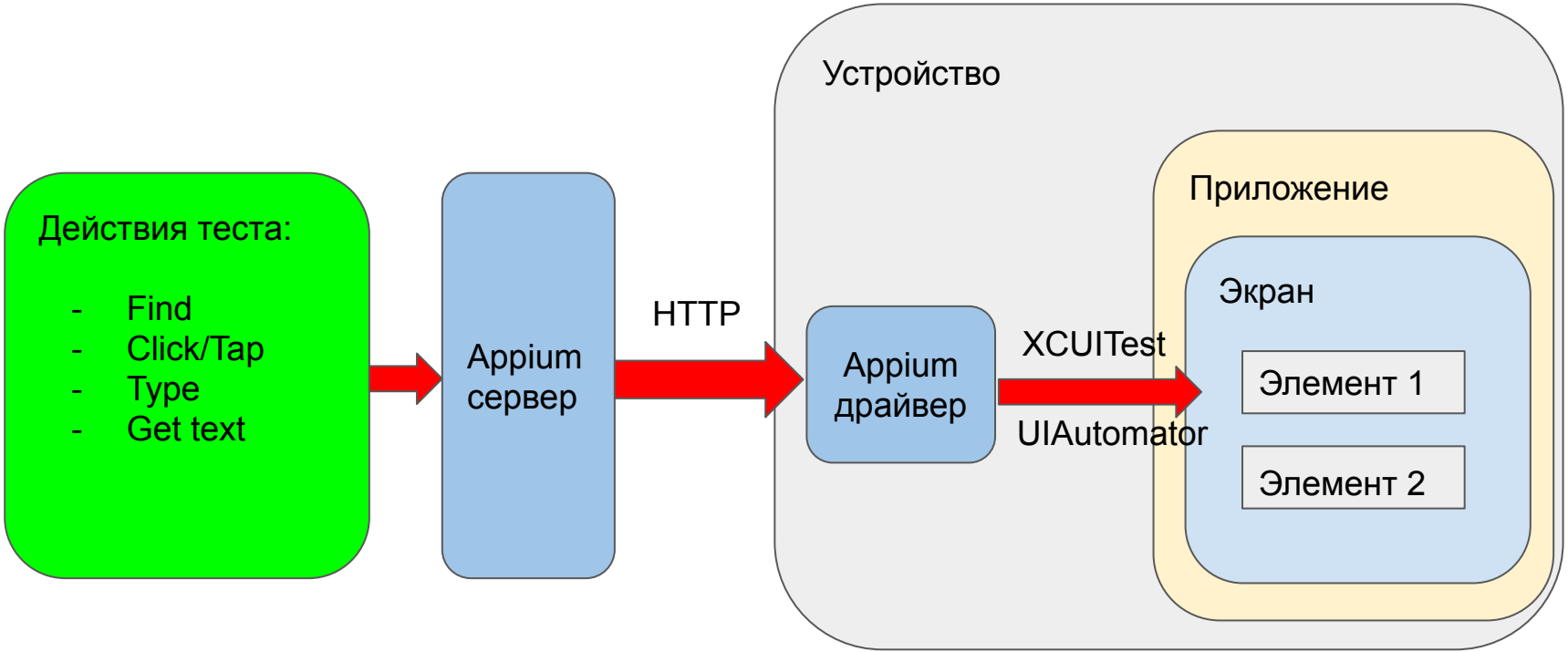
Appium:

- это сервер, к которому мы подключаемся для отправки команд устройству
- это драйверы для каждой отдельной операционной системы
- работает через HTTP

Сайт и установка: https://appium.io

Состоит из: сервер и клиент Appium Inspector

Appium: схема работы сервера



Appium Inspector

Лучший друг разработчика автотестов: позволяет изучить интерфейс приложения

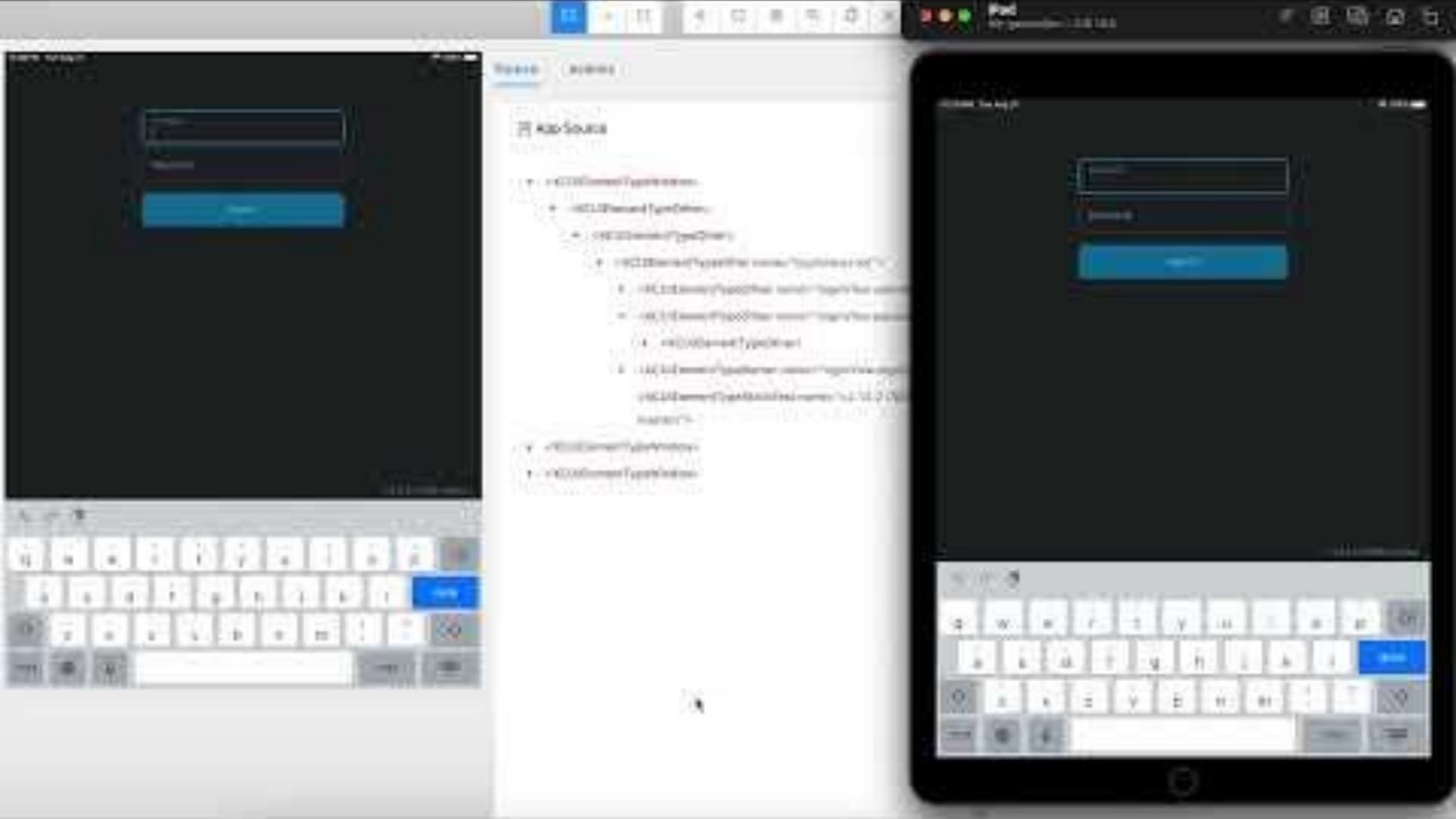
Типовой порядок работы с Appium (для “ручного” использования и в автотестах):

- Запуск сервера
- Задание параметров для подключения: Capabilities
- Подключение к устройству
- Исследование приложения

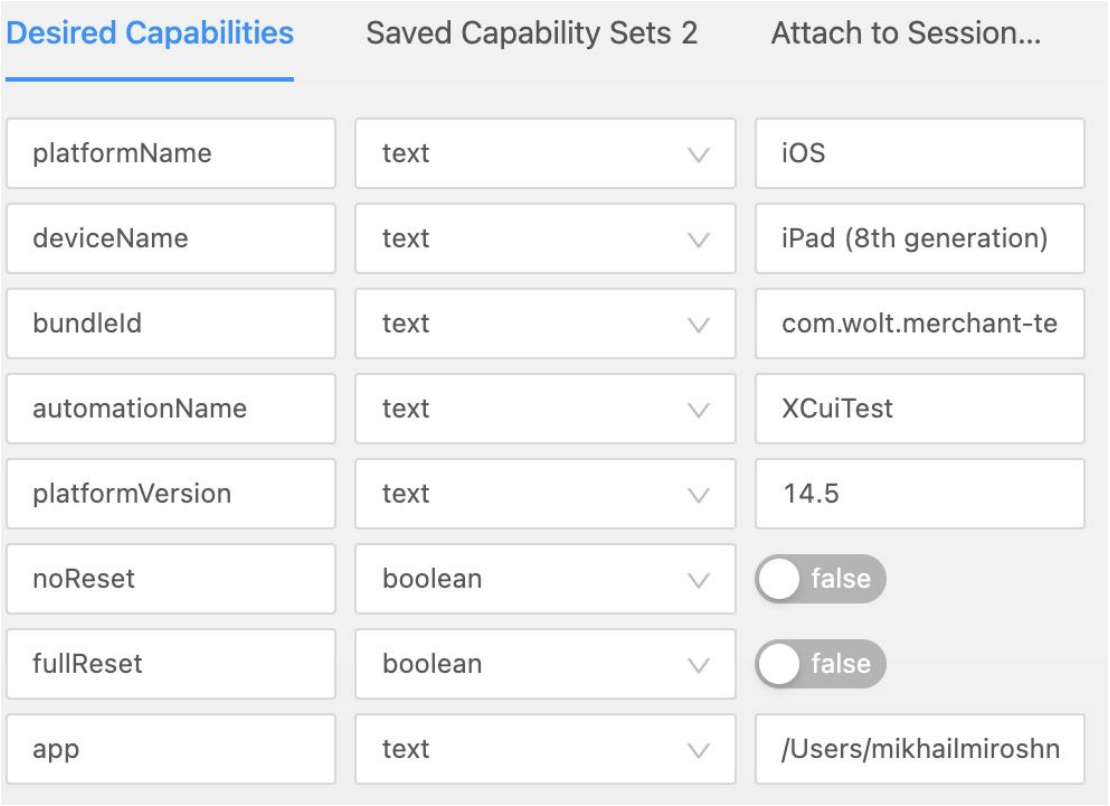
Appium: запуск сервера

Стартовый экран Appium Desktop для запуска сервера.

Возможен запуск и без интерфейса



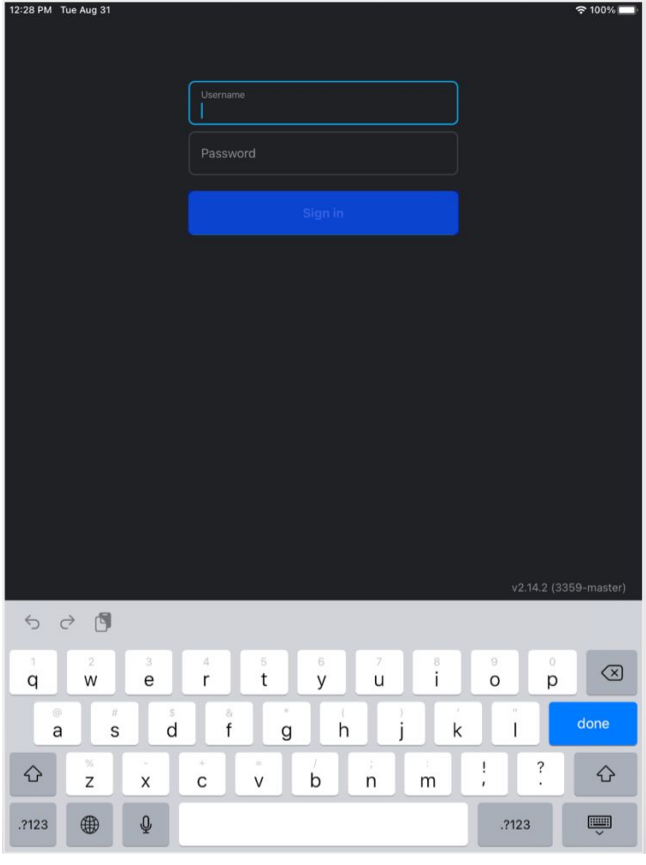
Appium: настройка подключения



Надо указать:
- Платформа
- Устройство
- Приложение
- Драйвер

Дополнительно:
- Настройки очистки данных перед запуском

Appium: Изучение схемы интерфейса



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>
```

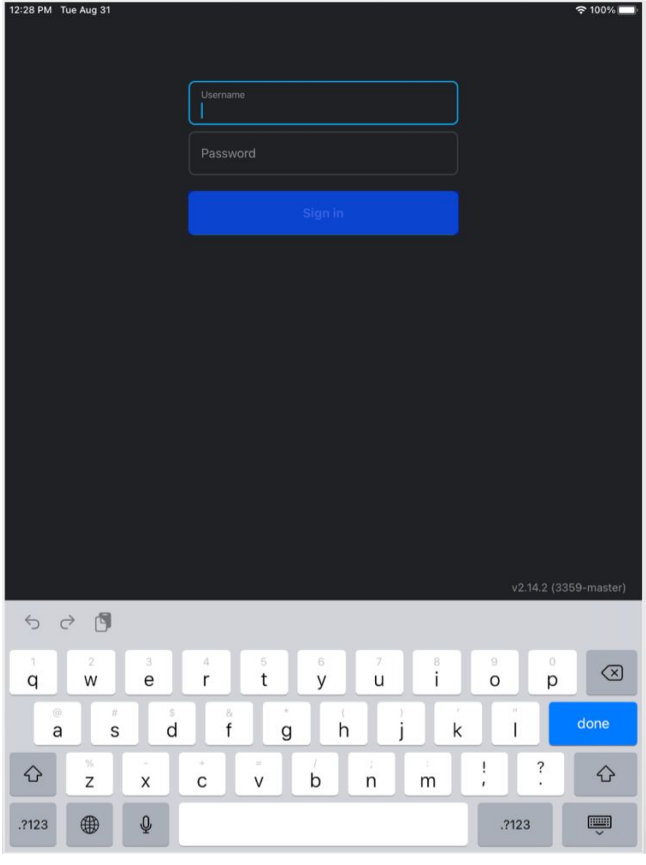
Selected Element

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: Способы нахождения элемента



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: какое приложение можно тестировать?

Самое главное требование к приложения для возможности использовать UI автоматизацию тестирования:

Элементы интерфейса, которые используются в тестах, обязаны иметь уникальные идентификаторы!

Без таких идентификаторов Appium не сможет найти элемент за минимальное время и абсолютно точно.

Начало создания автоматизированного теста

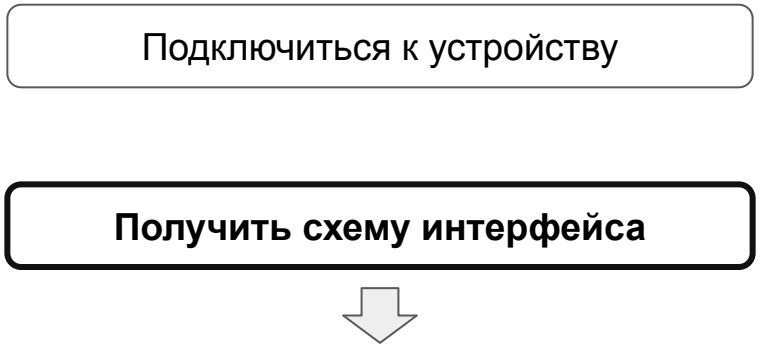
- У нас есть проект с необходимыми модулями
- Настроено подключение к устройству
- Мы знаем структуру интерфейса и как находить элементы на экранах

Часть 3: Создание автотестов

Построение эффективной структуры автоматизированных тестов

Базовая структура автотеста

Базовая структура автотеста



Базовая структура автотеста

Базовая структура автотеста

Базовая структура автотеста

Подключиться к устройству

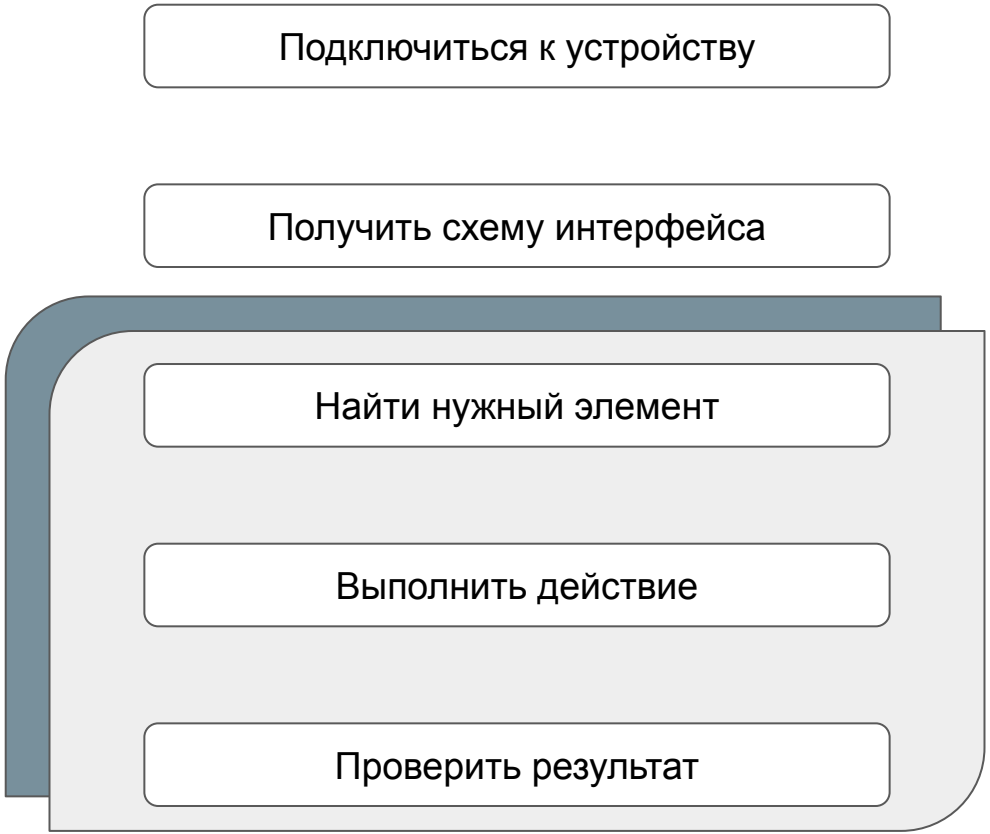
Получить схему интерфейса

Найти нужный элемент

Выполнить действие

Проверить результат

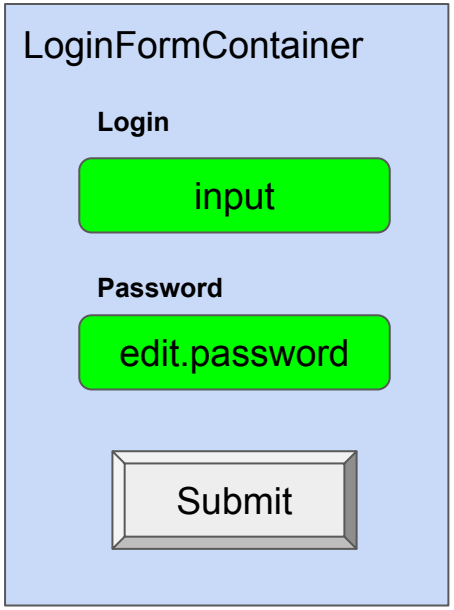
Базовая структура автотеста



Пример команд для работы с элементом

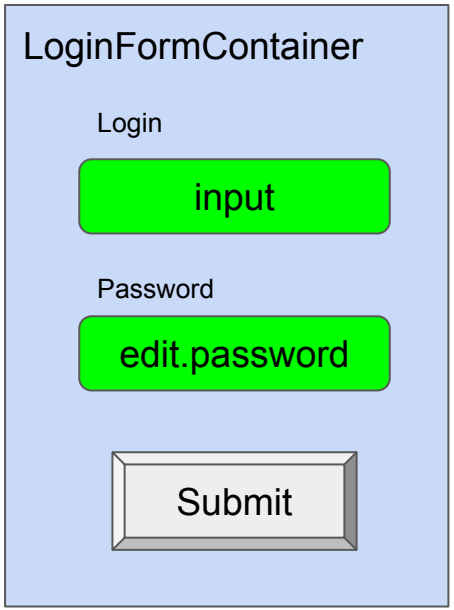
```
// Find edit field element with ID
val editField: MobileElement = driver.findElement(By.id( id: "my_edit_field"))

// 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'"
)
```

Пример автотеста: Форма с логином и паролем




```
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()

}
```

Несколько тестов использующих один элемент



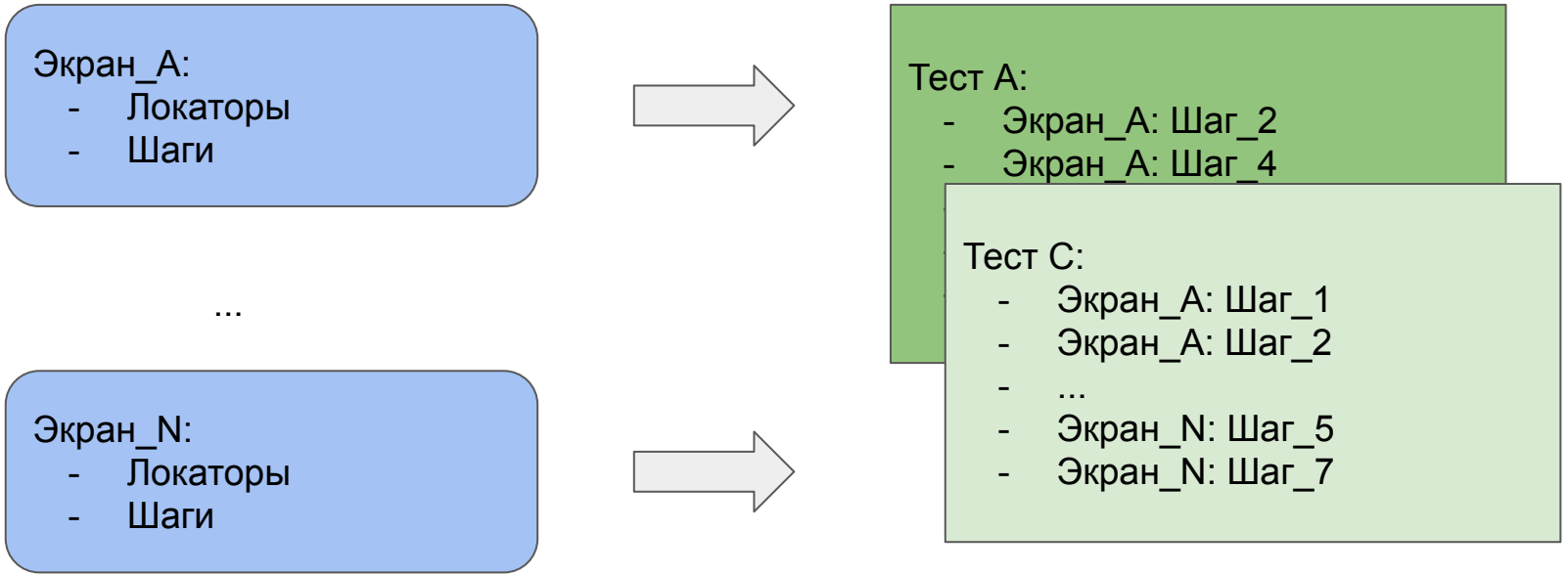
Тест 1: Вход в приложение с парой логин-пароль

- Найти элемент Login
- ...
- **Найти элемент Submit**
- **Нажать Submit**

Тест 2: Попытка входа без логина и пароля

- **Найти элемент Submit**
- **Нажать Submit**

Решение: разделение тестов и их шагов



Page Factory

- Позволяет разнести тесты с локаторами элементов и шагами
- Буквально: Создаем Page Objects
- Дает возможность строить тесты из шагов

Чтобы узнать больше:

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

https://victorz.ru/20171007671

Page Factory: принцип работы

Класс описания экрана (Page Object):

- Локаторы: правила как искать элементы на экране
- Шаги - действия с элементами: поиск, нажатие, ввода текста, получение текста, прокрутка

Использование:

- Инициализация объектов на странице
- Вызов шагов из теста

Page Factory: пример описания экрана - локаторы

```
@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: Шаги с переменными-элементами

```
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: Способы поиска элемента

FindBy может находить элементы по:

- 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: Переменная для элемента

val button: List<MobileElement>? = null

- **List** : найти все элементы с одинаковым ID одной командой

- **? = null** : изначально переменная пуста (null)

Page Factory: Инициализация

- В начале работе класс экрана не указывает на реальные элементы приложения.

- Прежде чем работать с переменными, описывающими элементы экрана, их надо инициализировать.

Page Factory: Инициализация в провайдере

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

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

Провайдер: объект, объединяющий экраны из одной области приложения

Page Factory: Пример провайдеров

```
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)
    }
}
```

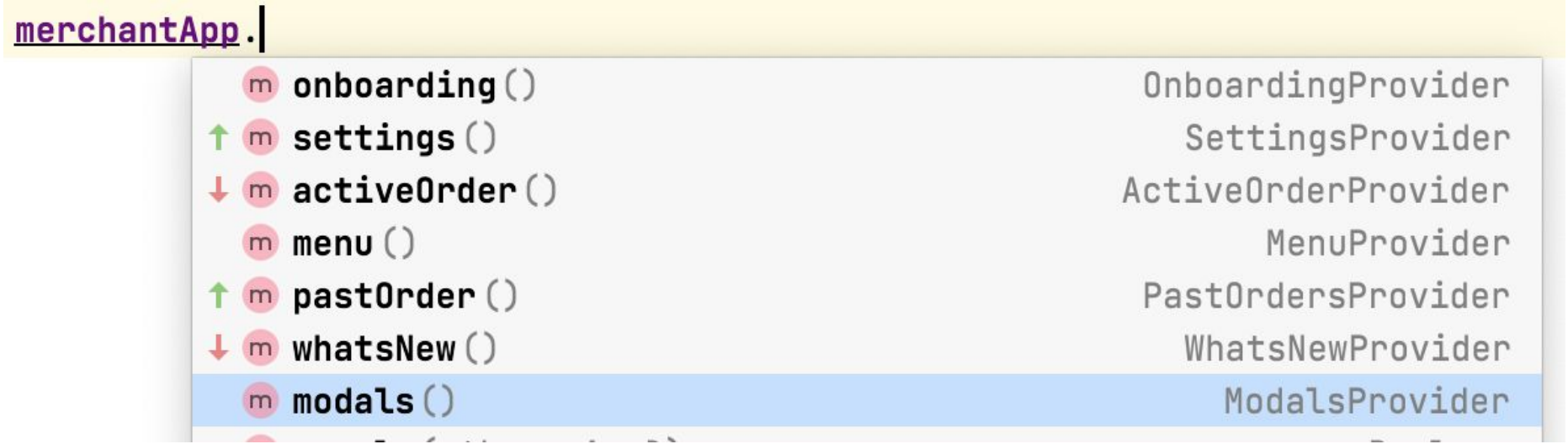
Page Factory: Корневой класс с провайдерами

```
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: Корневой класс

- Служит как единая точка входа для всех экранов приложения
- Объединяет все провайдеры приложения
- Каждый продукт в проекте - отдельное “приложение”

Page Factory: Выбор провайдера в приложении



Page Factory: Выбор экрана в провайдере

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

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

Page Factory: Выбор шага, доступного на экране

```
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: Выбор шага для другого экрана

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

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

Page Factory: Тестовый сценарий

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

Page Factory: Тест

```
merchantApp.onboarding().logInScreen()
    .isLoginScreenLoaded
    .setUserName("user")
    .setPassword("idkfa")
    .tapSignInButton()

with(merchantApp.onboarding().logInScreen()) {  this: LogInScreen
    isLoginScreenLoaded
    setUserName("user")
    setPassword("idkfa")
    tapSignInButton()  ^with
}
```

- Тест содержит только шаги: как тест-кейс
- Можно группировать шаги для наглядности

Page Factory: Шаги, возвращающие значение

… а переводящие на следующий экранный элемент

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

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

Пример: Как мы напишем авто-тест

Пример сценария:

- Запуск приложения
- Ввод логина и пароля
- Ожидание следующего экрана
- Открытие меню
- Выбор пункта
- Проверка что нужный экран показан

```
class SignInTest : BaseTest() {

    @Test(description = "my test")
    fun myTest() {

        merchantApp.onboarding().logInScreen()
            .isLoginScreenLoaded
            .setUserName("user")
            .setPassword("idkfa")
            .tapSignInButton()

        merchantApp.activeOrder().activeOrder()
            .isActiveOrderScreenShown

        merchantApp.menu().
    }
}
```

```
header()
sideMenu()
equals(other: Any?)
toString()
hashCode()
to(that: B) for A in kotlin
javaClass for T in kotlin.jvm
also {...} (block: (MenuProvider) -> Unit) fo…
apply {...} (block: MenuProvider.() -> Unit) …
let {...} (block: (MenuProvider) -> R) for T in kotlin
run {...} (block: MenuProvider.() -> R) for T in kotlin
```

“Оптимизация” Page Factory

Создадим объекты не страничек, а действий на страницах

Библиотека действий с элементами

Основные действия над элементами также можно вынести в отдельную библиотеку:

- ожидание
- нажатие
- ввод текста
- получение текста
- прокрутка
- ...

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

Использование библиотеки действий в шагах теста

```
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)
}
```

“Оптимизация” действий над элементами

Цель: Мы хотим иметь возможность быстро понять где что-то пошло не так.

Возможные причины падения теста:

- Элемент отсутствует на экране
- Элемент имеет некорректное состояние: видимость, доступность…
- Нажатие не может быть выполнено
- Текст не был введен в поле
- …

Appium падает так же как Selenium, только *мобильно* =)

Обработка ошибок: реализация

Идея:

- Действия над элементами должны возвращать Boolean значение своего результат.
- Шаги тестов проверяют результат действий.

Что делаем:

- Добавляем: обработку результатов вызова Appium команд в библиотеку действий
- Добавляем: проверки в шаги теста, использующие действия

Пример: ввод текста

```
fun sendKeys(elements: List<MobileElement>?, num: Int, txt: String?): Boolean {
    Засечем время начала попыток
    do {
        try {
            Попытаемся просто ввести текст в элемент. Если получилось, вернем “Все ок”
        } catch (e: ElementNotInteractableException) {
            Если не нажимается, можно попробовать еще раз
        } catch (e: Exception) {
            Если все упало, вернем “Не удача”
        }
        Сделаем паузу между попытками
    } while Пока время на попытки не вышло
    return Вернем результат что получилось
}
```

Пример: ввод текста

```
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
}
```

Шаги теста с проверкой действий с элементами

```
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 - Метка с именем текущего класса

Расширение возможностей теста

Тест красив и стабилен =)

Пора провести небольшую подготовительную работу перед тестами =)

TestNG: Шаги “вокруг” теста

Тест работает. Как добавить дополнительные действия до и после тестирования?

TestNG: Управление тестом

Функции TestNG для дополнительных действий до и после теста

- До прогона тестов: @BeforeSuite
- Перед каждым тестом: @BeforeMethod
- После каждого теста: @AfterMethod
- После прогона всех тестов: @AfterTest
- После окончания тестирования: @AfterSuite

Документация:

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

Управление тестом: применение TestNG

Примеры:

- Запуск и остановка Appium сервера: @Before/After Suite

- Сброс кеша приложения, очистка “мусора” после теста: @Before/After Method

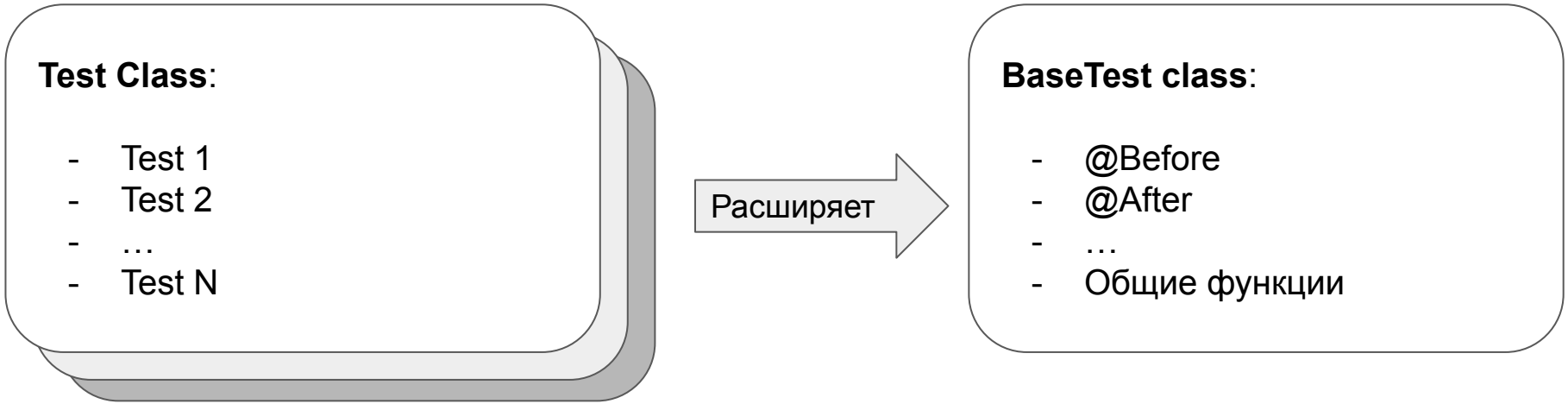
- Сбор статистики о тестировании: @AfterTest

Управление тестом: применение TestNG

Функции с аннотациями TestNG

```
@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) {...}
```

Иерархия тестовых классов



Отчеты

Наш тест работает, умеет готовить окружение и даже подчищает за собой временные данные. Как представить результат работы теста?

```
[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] ------------------------------------------------------------------------
```

Текстовый лог о ходе теста в консоли

Во время разработки и при тестирование продукта:

- Показывает что тест делает в данный момент

- Сохраняет полную историю выполнения теста для анализа

- Показывает ход выполнения, когда не видно экрана устройства

Текстовый лог: реализация

```
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")
}
```

Текстовый лог в работе

```
  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
```

“Оптимизация” отчетов

От статичного текстового лога, привязанного в исполнению теста, к подробному отчету с картинками и на человеческом языке. Где-то на публичном сервере.

Создание отчетов с помощью Allure

Для чего: Мы хотим иметь возможность легко понять и увидеть статус протестированного приложения.

Как начать использовать:

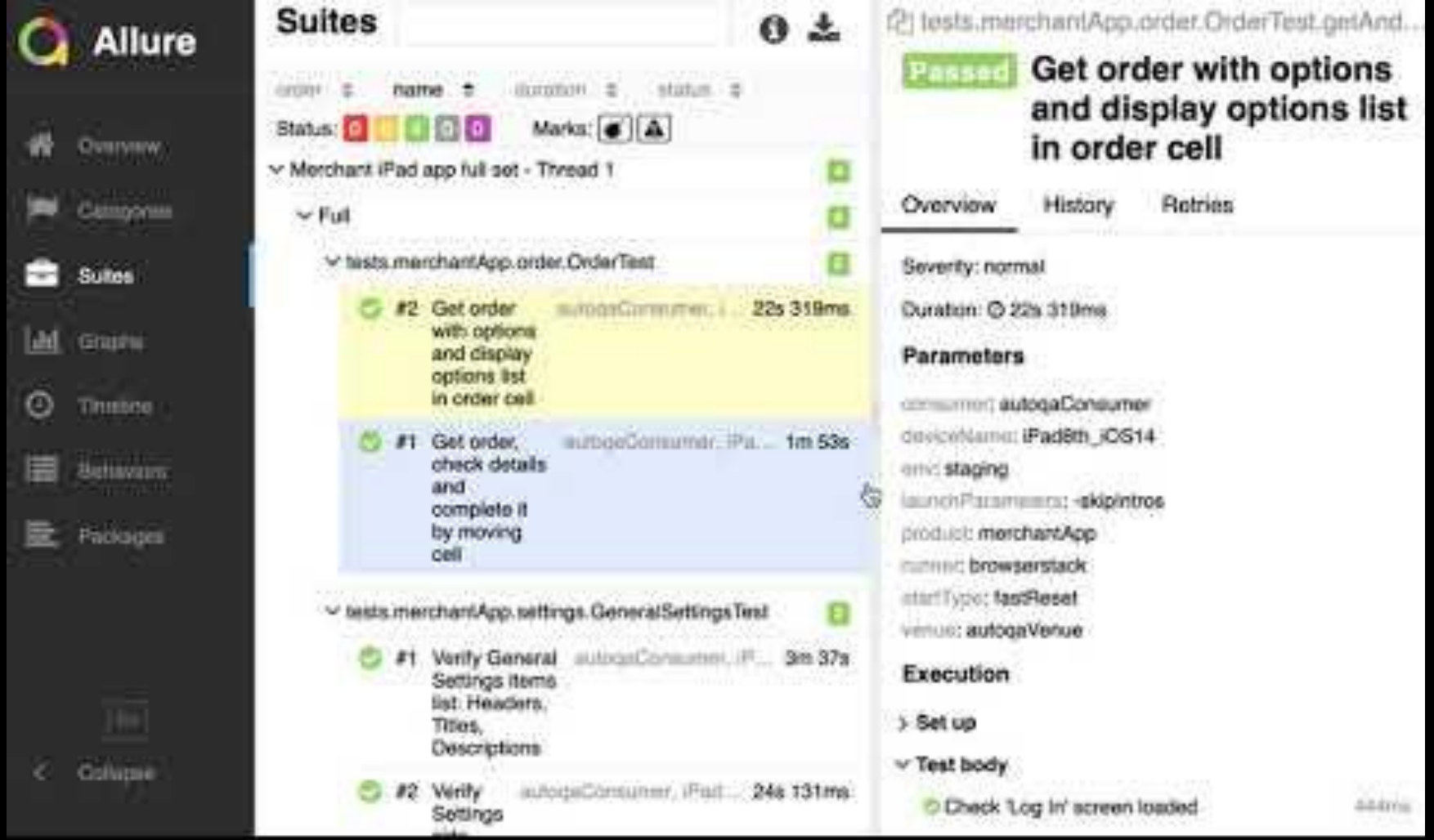
- Подключить библиотеку к проекту с Maven
- Добавить аннотации к классам, тестам и шагам
- Сгенерировать отчет

Важно: Allure - это веб-сервер.

Доклад А. Ерошенко на Heisenbug: https://www.youtube.com/watch?v=tTV00kSutQI

Allure: Пример

Представим, что мы прогнали тесты. Посмотрим, как выглядит веб страничка с отчетом.


Allure
Overview
Categories
Suites
Graphs
Timeline
Behaviors
Packages
Collapse
Suites
order
name
duration
status
Status:
Marks:
Merchant iPad app full set - Thread 1
Full
tests.merchantApp.order.OrderTest
#2 Get order with options and display options list in order cell
22s 319ms
#1 Get order, check details and complete it by moving cell
1m 53s
tests.merchantApp.settings.GeneralSettingsTest
#1 Verify General Settings items list: Headers, Titles, Descriptions
3m 37s
#2 Verify Settings
24s 131ms
tests.merchantApp.order.OrderTest.getAnd...
Passed
Get order with options and display options list in order cell
Overview
History
Retries
Severity: normal
Duration: 22s 319ms
Parameters
consumer: autoqaConsumer
deviceName: iPad8th_iOS14
env: staging
launchParameters: -skipIntros
product: merchantApp
runner: browserstack
startType: fastReset
venue: autoqaVenue
Execution
Set up
Test body
Check 'Log In' screen loaded

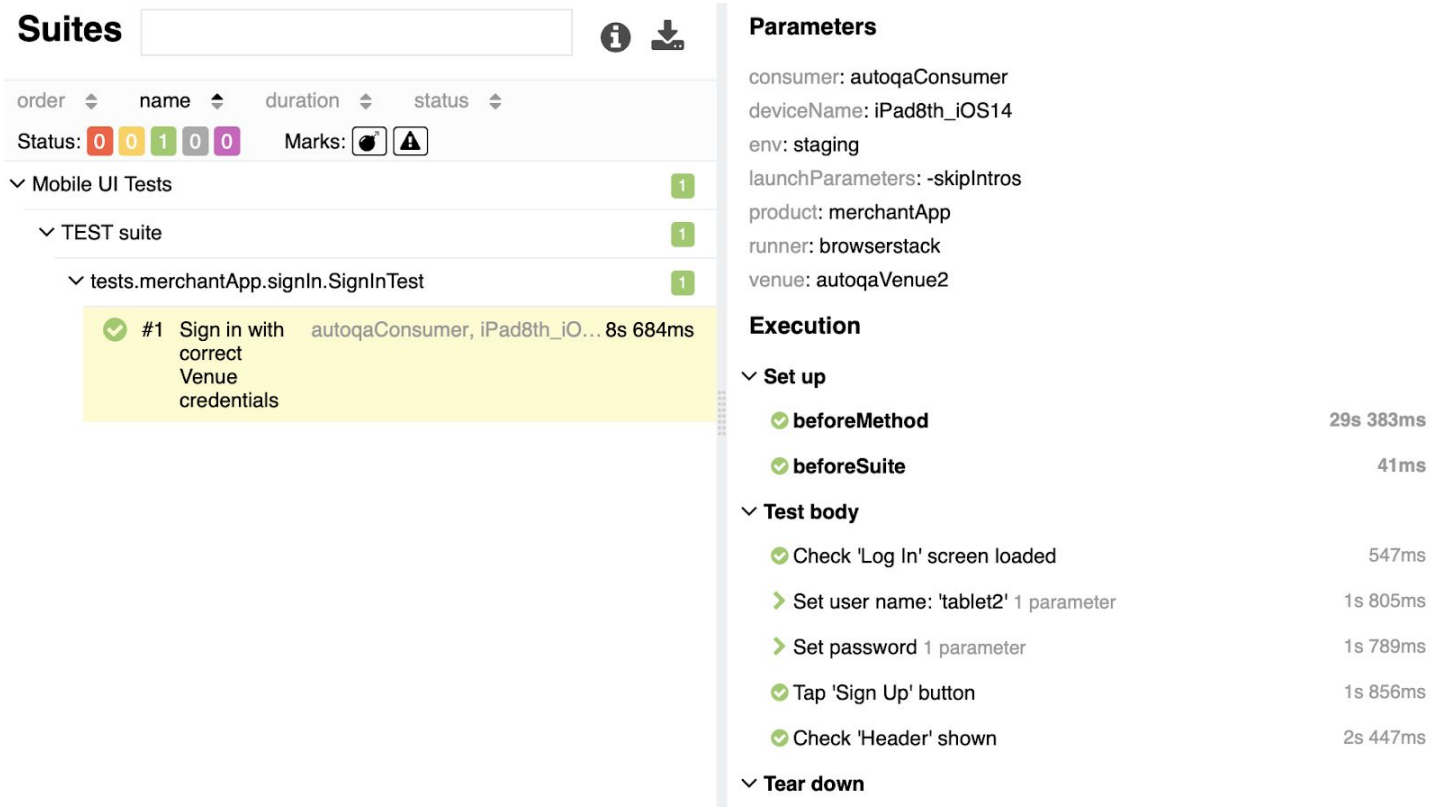
Allure: Добавление аннотаций к классу с тестами

```
@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: Добавление аннотаций к классу с экраном

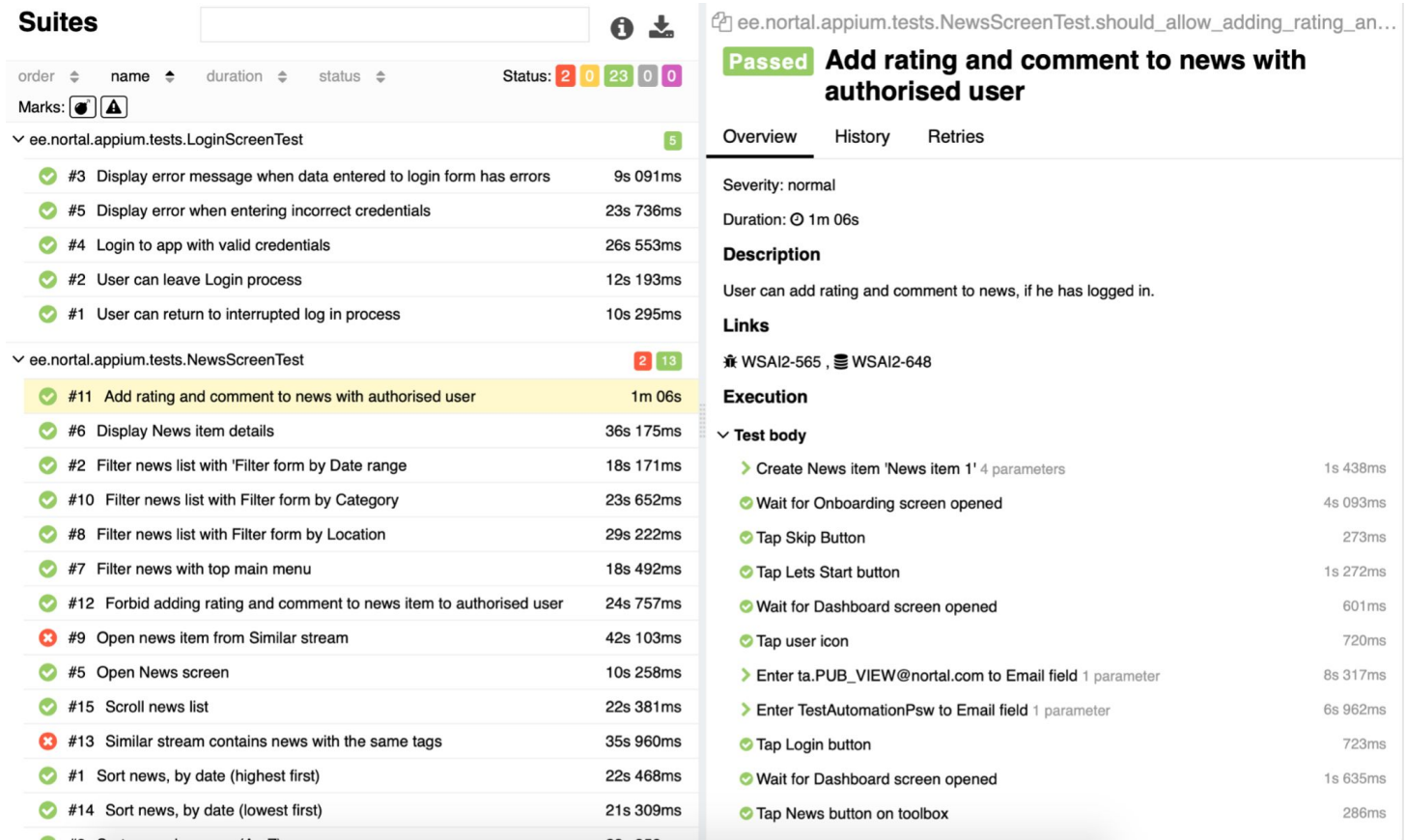
```
@Step("Set user name: '{userName}'")
fun setUserName(userName: String): LogInScreen = also { it: LogInScreen
    log( text: TAG + "setUserName(): " + userName)
    Assert.assertTrue(sendKeys(userNameInput, userName), "${TAG}setUserName(): FAILED")
}

@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: Веб страничка с отчетом о тестировании

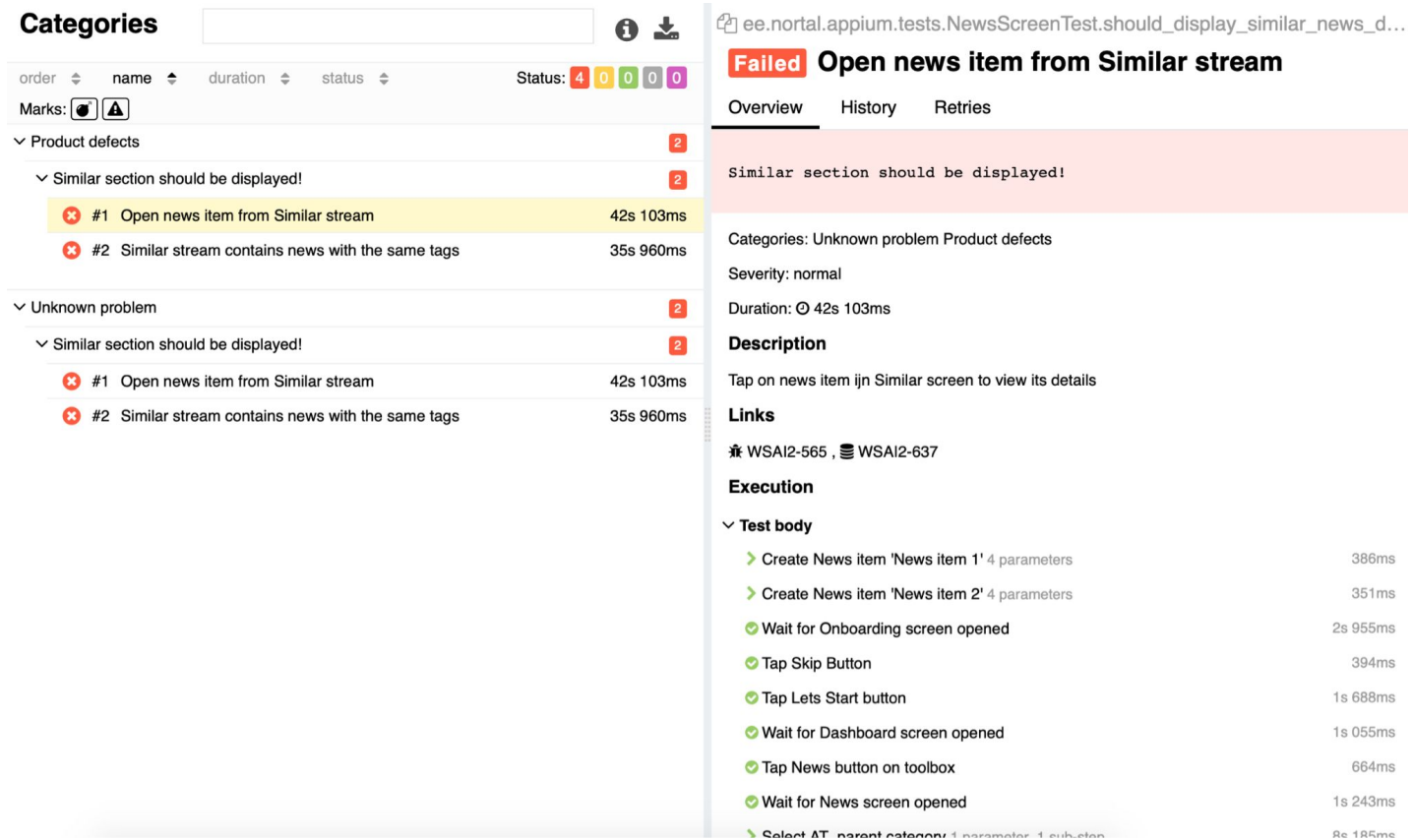


Allure: Обзор истории прогонов

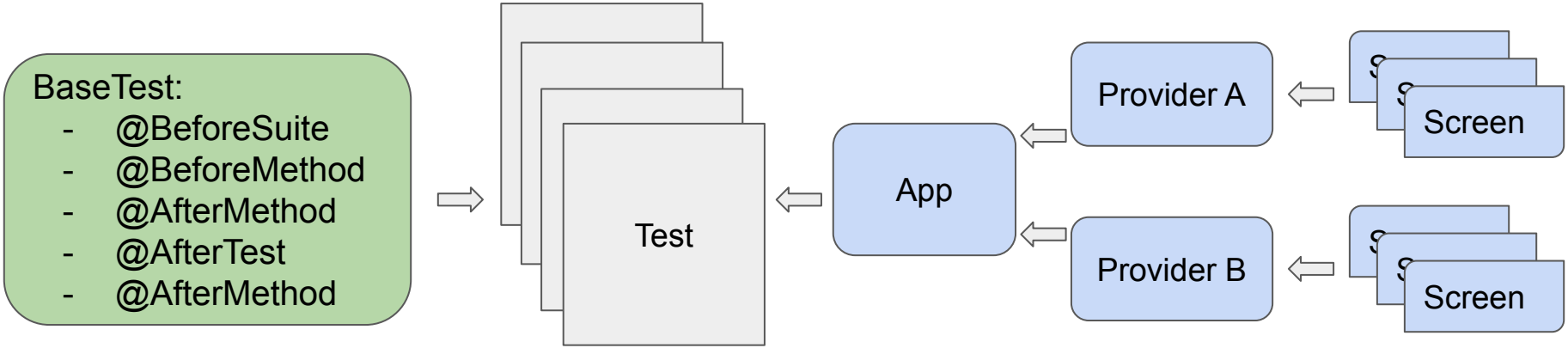
Allure: Список текстов и их шаги



Allure: Вывод причины падения теста



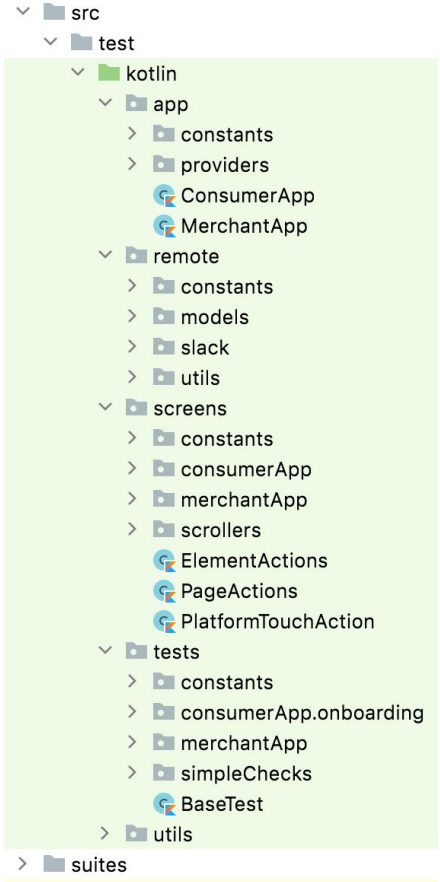
Финальная архитектура проекта



Где запускать автотесты?

Теперь у нас есть что запускать и чем запускать. Но где это запустить, чтобы получить быстрый отчет о состоянии приложения?

- **Самостоятельно**: разработчик и тестировщик вручную
- **На серверах Continuous Integration**:
  - Push в ветку новой задачи
  - Создание Pull request
  - Merge to Master



Часть 4: Запуск автотестов

Запускаем созданные автотесты у себя и у соседа

Запуск теста на локальной машине

- Реальное устройство
- Эмулятор / Симулятор
- Все необходимые драйверы и инструменты

\+ Работает максимально быстро (поскольку рядом с агентом, исполняющим тест)

\- Требует соответствующее аппаратное обеспечение

Реальное устройство

+ Не важна производительность агента
+ Реальное программное окружение, как у пользователя
+ Доступны все популярные сервисы Google, Apple и т.д.

- Ограничения по доступу к определенным функциям
- Надо покупать отдельно
- Необходимо самостоятельно обслуживать устройства

Эмулятор

+ Любая конфигурация без дополнительных затрат
+ Возможно все

- Нет популярных сервисов
- Высокие требования к производительности агента

iOS устройства

Включает:

- Симулятор iOS, идущий в составе IDE XCode для macOS

- iPhone / iPad

iOS устройства

Особенности:

- Нужен macOS
- Требуется аккаунт разработчика для сборки приложения на реальное устройство
- Многие функции заблокированы на реальном устройстве.
- Интерфейс строится по разному в зависимости от версии ОС
- Приложение для симулятора: **.app** x86 (ARM на устройстве с M1)
- Приложение для устройства: **.ipa** ARM

Android устройства

Включает:

- Эмулятор Android Studio: AVD - Android Virtual Device, Win / macOS
- Реальное устройство

Android устройства

Особенности:

- Можно собрать дистрибутив приложения apk, который будет работать и в эмуляторе, и на любом реальном устройстве
- Просто установить приложение на любое устройство

Пример: Выполнение теста на симуляторе iPad

На примере **Wolt Merchant App**: приложения для ресторанов для приема и обработки заказов

Тест: Отмена полученного заказа

- Войти в приложение
- Отправить заказ в приложение
- Отменить принятый заказ

Как работает:

- На локальной машине
- На iPad симуляторе

Запуск теста на другой машине: все и сразу

- **С компиляцией проекта и на подключенном устройстве**
  - Java
  - maven
  - Appium
  - iOS: XCode
  - Android: Android debug tools
  - Allure (для отчетов)

Запуск теста на другой машине: на ферме устройств

- С компиляцией проекта и на подключенном устройстве
  - Java
  - maven
  - Appium
  - iOS: XCode
  - Android: Android debug tools
  - Allure (для отчетов)
- **С компиляцией и на удаленном устройстве**
  - Java
  - maven
  - Appium не нужен: он настроен на ферме устройств

Запуск теста на другой машине: в образе системы

- С компиляцией проекта и на подключенном устройстве
  - Java
  - maven
  - Appium
  - iOS: XCode
  - Android: Android debug tools
  - Allure (для отчетов)
- С компиляцией и на удаленном устройстве
  - Java
  - maven
  - Appium не нужен: он настроен на исполнителе тестов
- **В Docker** с уже собранным проектом автотестов - отлично для CI

Использование 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
```

Использование Docker: Порядок сборки

Создание файла с описанием образа на основе Maven образа:
https://hub.docker.com/_/maven

Сборка образа из файла:

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

Запуск тестов в контейнере:

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

Использование Docker: Работа в контейнере

На чем запускать тесты при работе в контейнере?

Запуск тестов в контейнере:

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

На ферме устройств

Облачные фермы

Что делать, когда надо запускать приложения и тесты на реальных устройствах, но их у вас нет (или есть, но недостаточно).

Или девайсы купить можно, но вот поддерживать их не хочется.

Решение: Облачные сервисы с реальными устройствами (Mobile / Web)

- BrowserStack
- SauceLabs
- LambdaTest (beta)

Облачная ферма устройств

- Предоставляется по подписке

- Имеет парк распространенных мобильных устройств с разными версиями ОС
- Имеет парк web-окружений с популярными браузерами и их версиями.

- Позволяет запускать автотесты
- Дает Live-доступ к устройства через веб-браузер

Облачная ферма устройств

Представим, что мы запустили наши тесты на устройстве сервиса BrowserStack...

FILTERS

Build > shouldDisplayWhatsNewPopUpAtSignIn

Delete Session | Report a bug | Contact Support

Local Testing: Off | Session ID: | User: | Protocol: MJSONWP

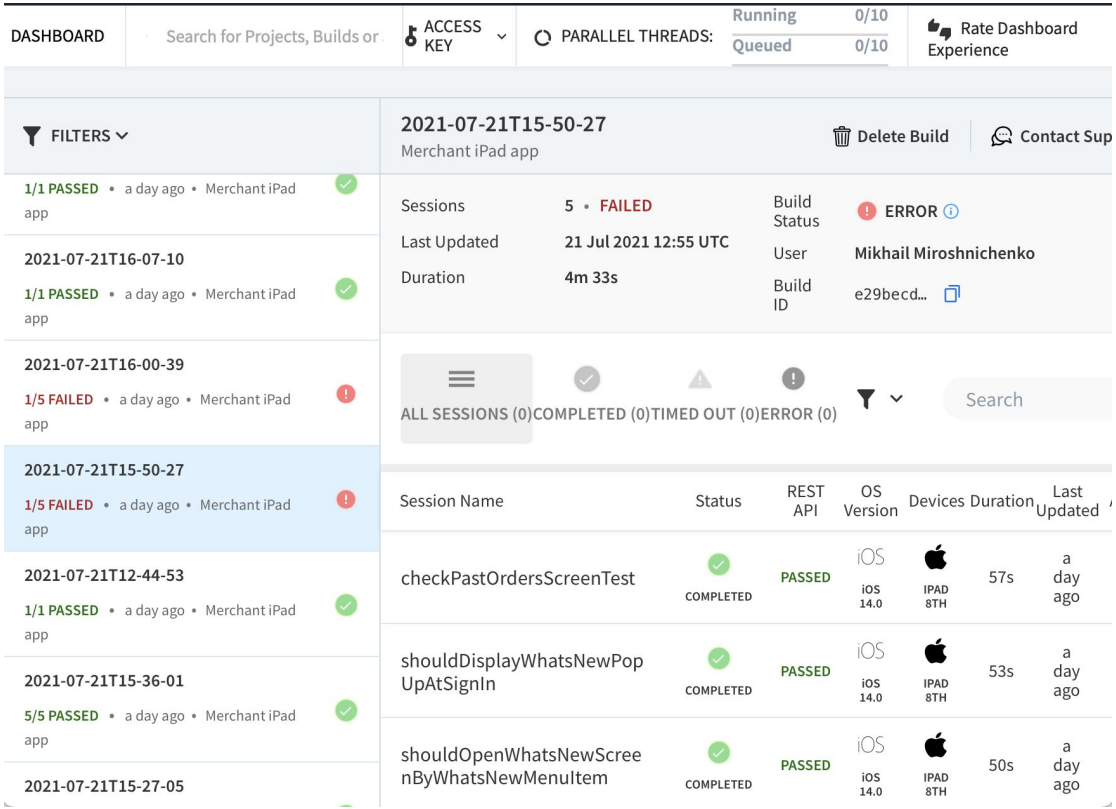
REST API Session: Test passed

Caps | Input Capabilities | Device Capabilities

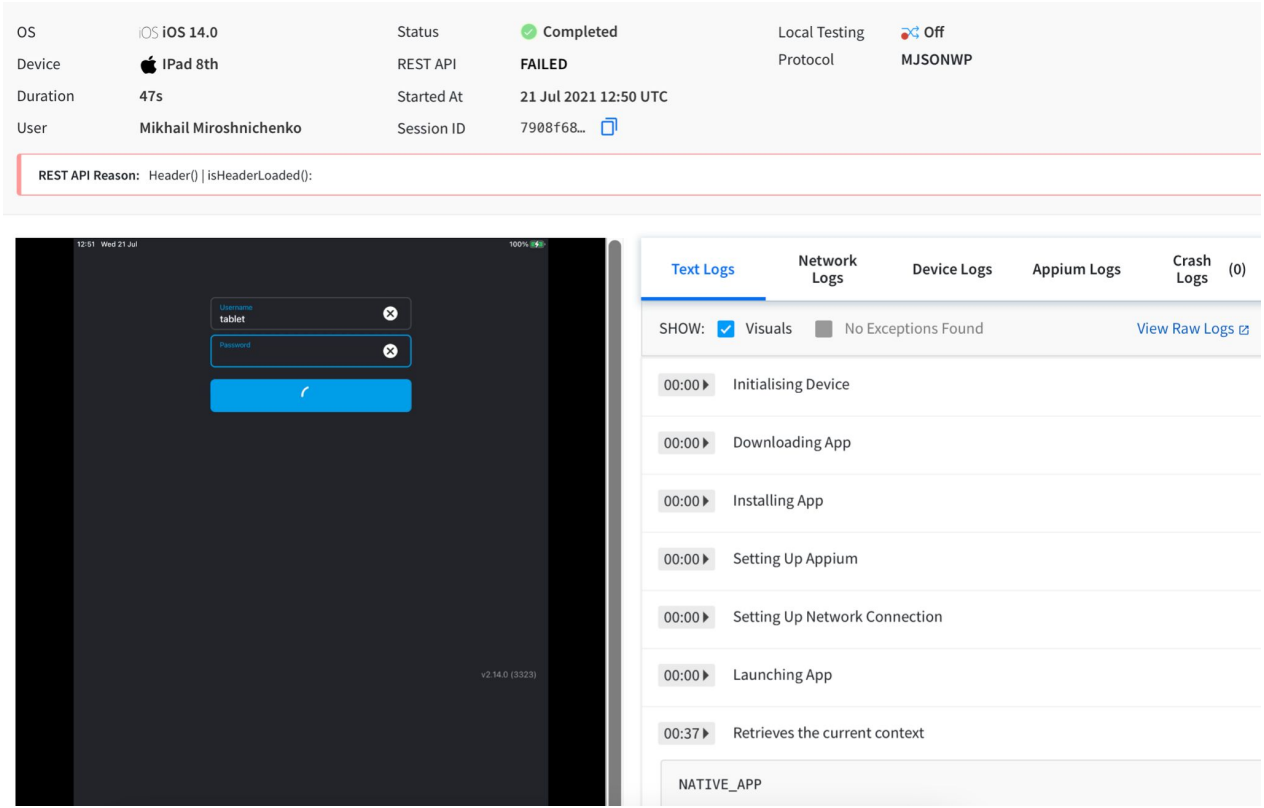
appName	
bundleId	
version	3.14.3

Test Logs | Network Logs | Device Logs | Appium Logs | Crash Logs

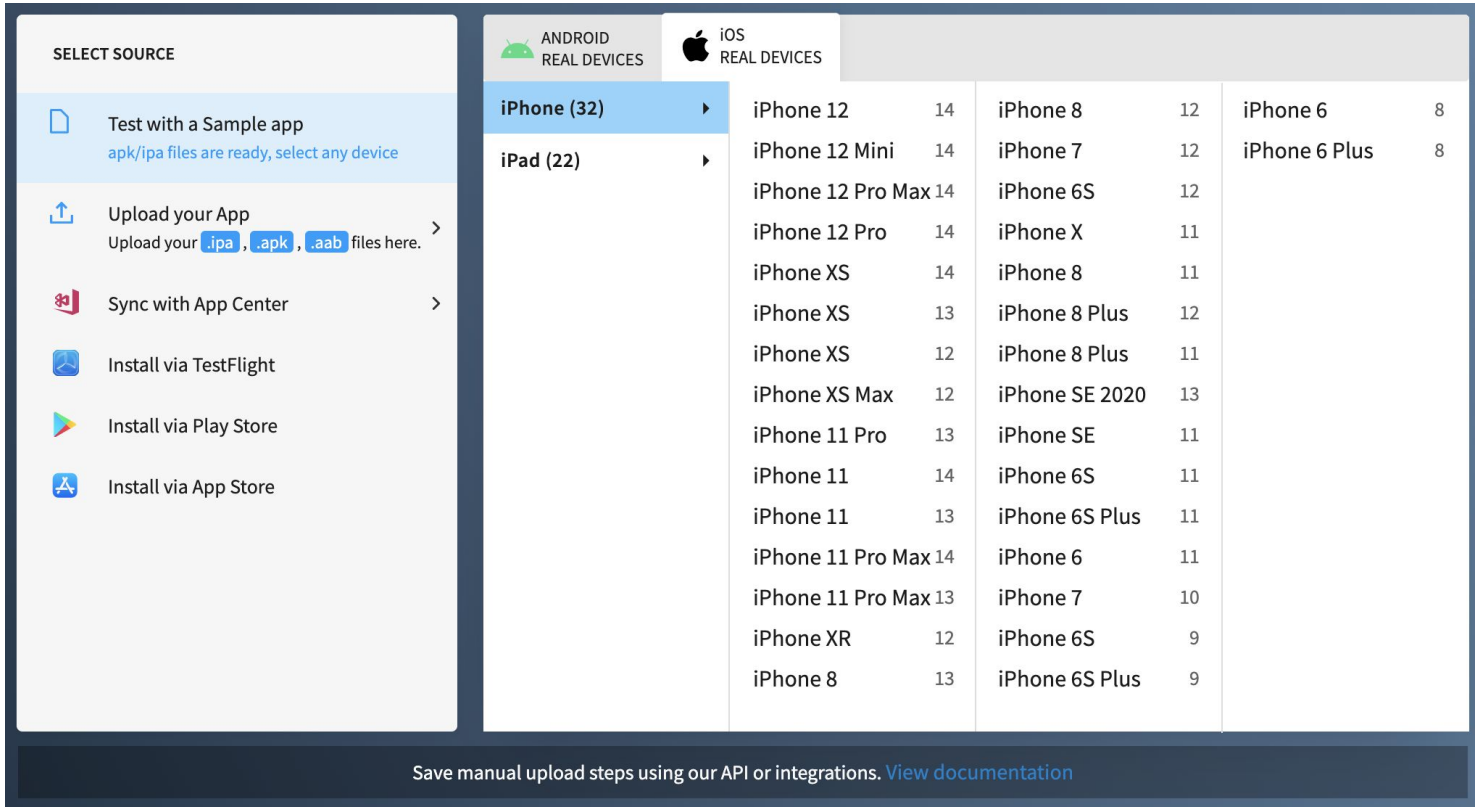
BrowserStack: Список прогонов тестов



BrowserStack: Просмотр информации о сессии

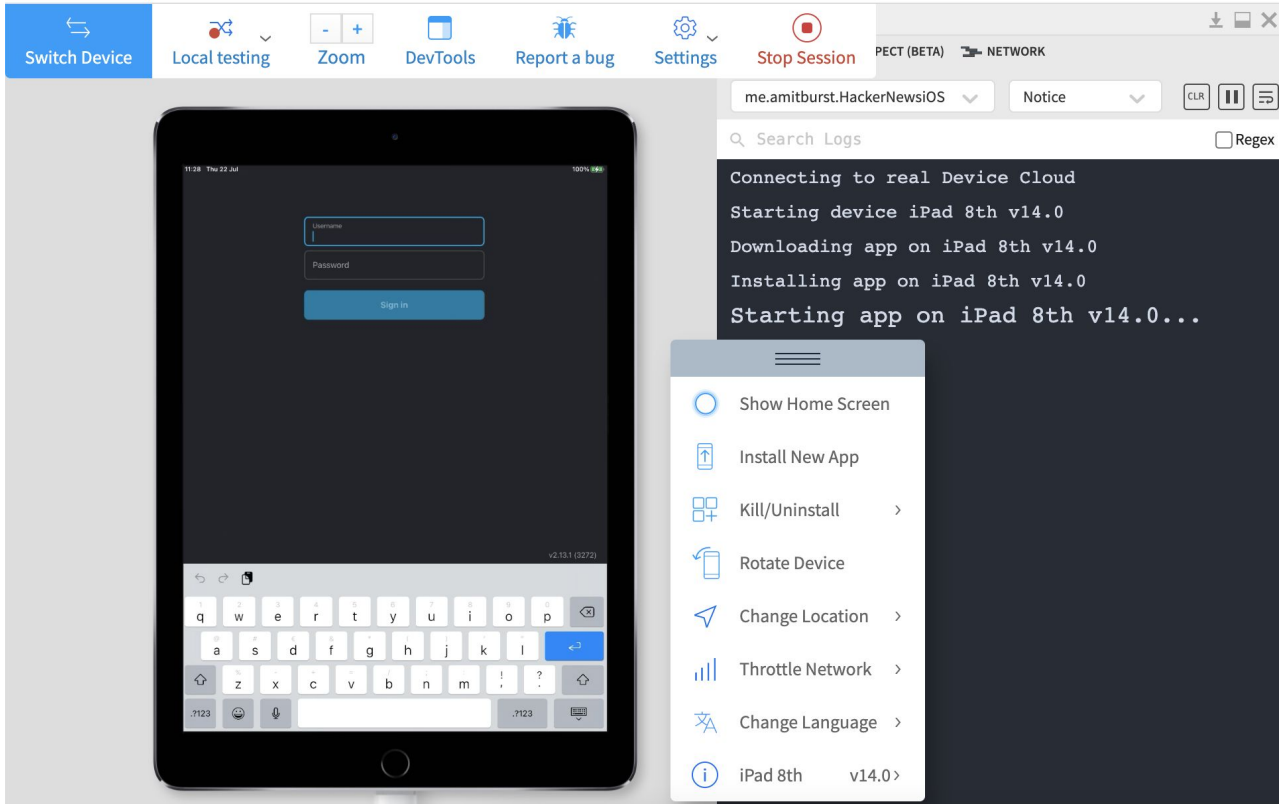


BrowserStack: Live сессия - настройка




Switch Device
Local testing
Zoom
DevTools
Report a bug
Settings
Stop Session
Connecting to real Device Cloud
Starting device iPad 8th v14.0
Downloading app on iPad 8th v14.0
Installing app on iPad 8th v14.0.

BrowserStack: Live сессия - использование



Запуск теста в облаке

Шаги для реализации:

- **Получить** логин-пароль для доступа к BrowserStack
- **Загрузить** приложения в облако
- **Подключиться** к устройству из теста

Документация с примерами кода:

https://www.browserstack.com/docs/app-automate/appium/getting-started/java

Настройка подключения к BrowserStack

```
capabilities.setCapability("device", "Samsung Galaxy S21 Ultra")
    capabilities.setCapability("os_version", "10.0")
    capabilities.setCapability("project", "My First Project")
    capabilities.setCapability("build", "My First Build")
    capabilities.setCapability("name", "Bstack-[Java] Sample Test")
    capabilities.setCapability("app", "my_awesome_android_app")


driver = AndroidIOSDriver<AndroidElement>(
URL("https://"+userName+":"+accessKey+"@hub-cloud.browserstack.com/wd/hub"), capabilities);
```

И это все!

Тесты готовы.

Есть что запускать: тесты для продукта

Есть чем запускать: наш фреймворк

Есть где запускать: в контейнере Docker

Есть на чем запускать: на облачной ферме

Поехали в CI!

Часть 5: Интеграция с CI

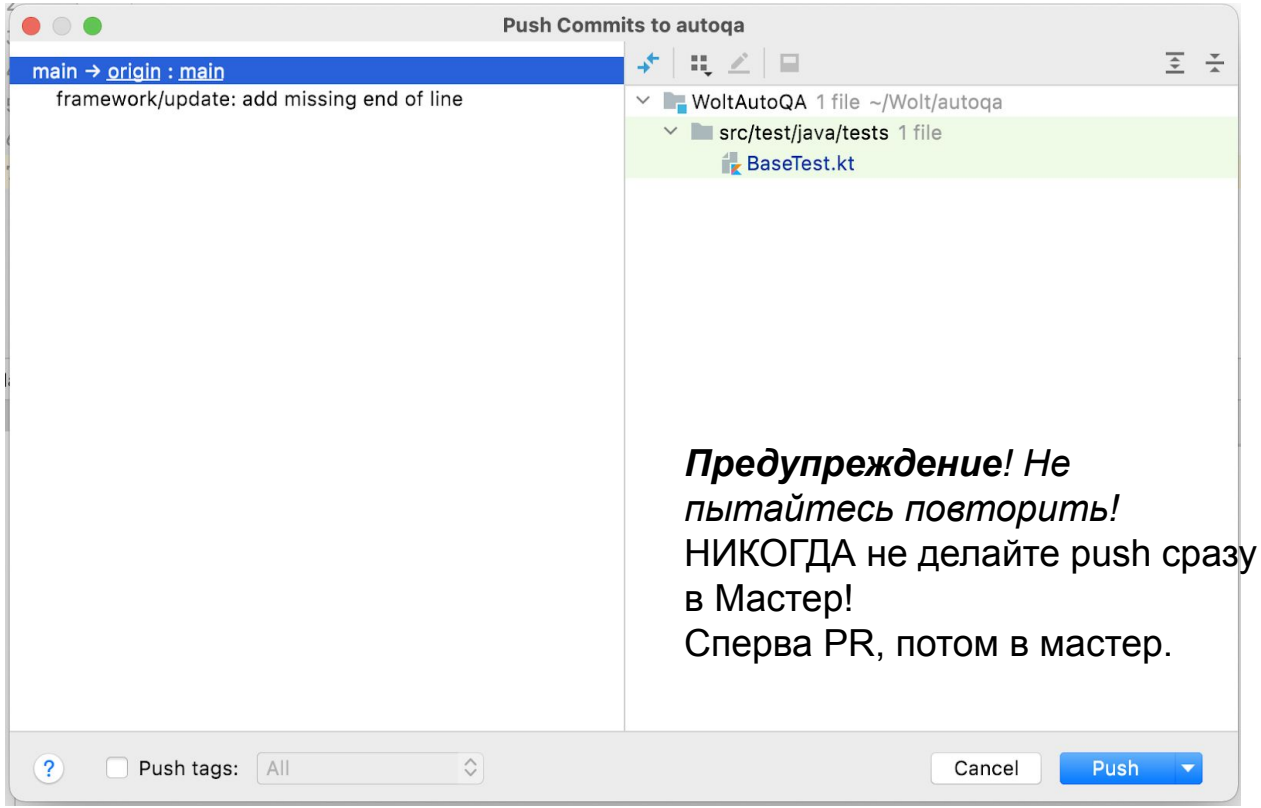
Дополним сборку продукта на CI прогоном автотестов

Интеграция тестов в CI: план

- Хранение проекта в GitHub
- Сборка Docker образа с GitHub Actions
- Хранение ценной информации в GitHub Secrets
- Размещение образа в облаке Amazon Web Storage
- Сборка продукта и прогон тестов на CircleCI
- Отправка результатов тестирования в Slack

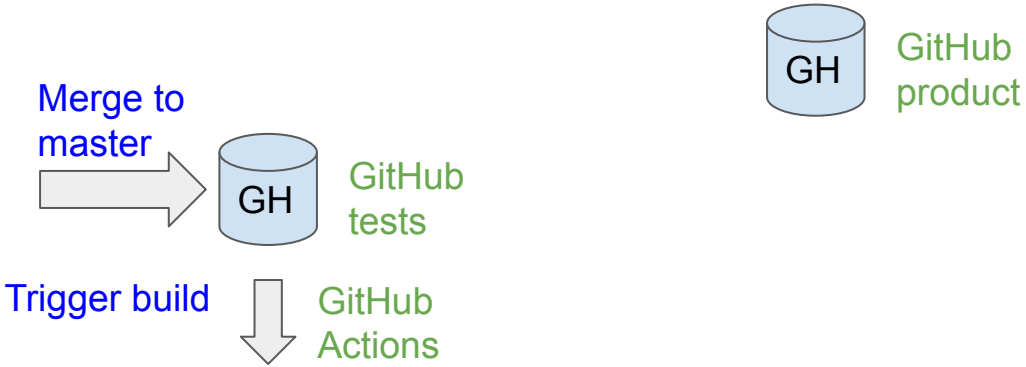
CI: Репозитории с исходным кодом

autoqa репозиторий: Push to Master

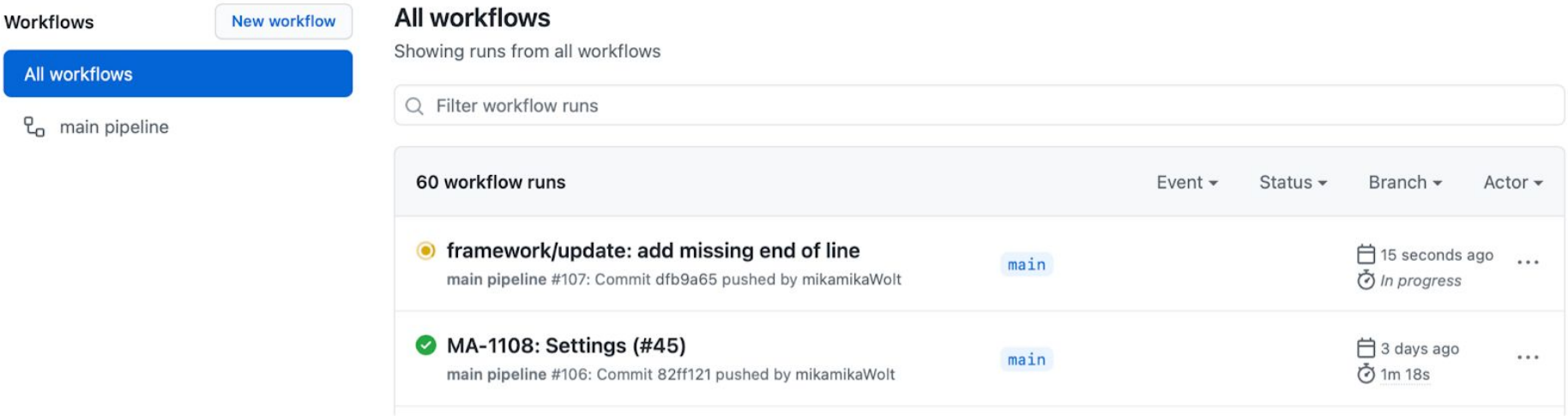


***Предупреждение**! Не пытайтесь повторить!*
НИКОГДА не делайте push сразу в Мастер!
Сперва PR, потом в мастер.

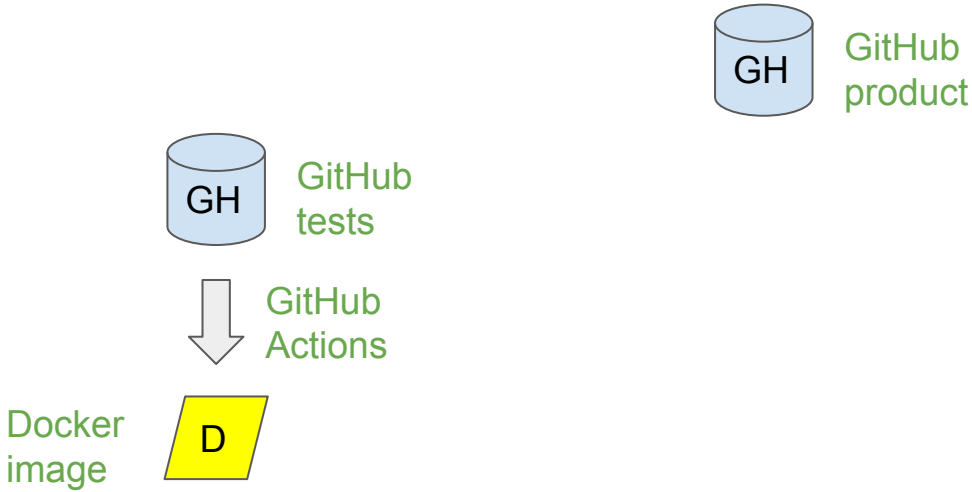
CI: Добавление нового кода запустит GH Actions



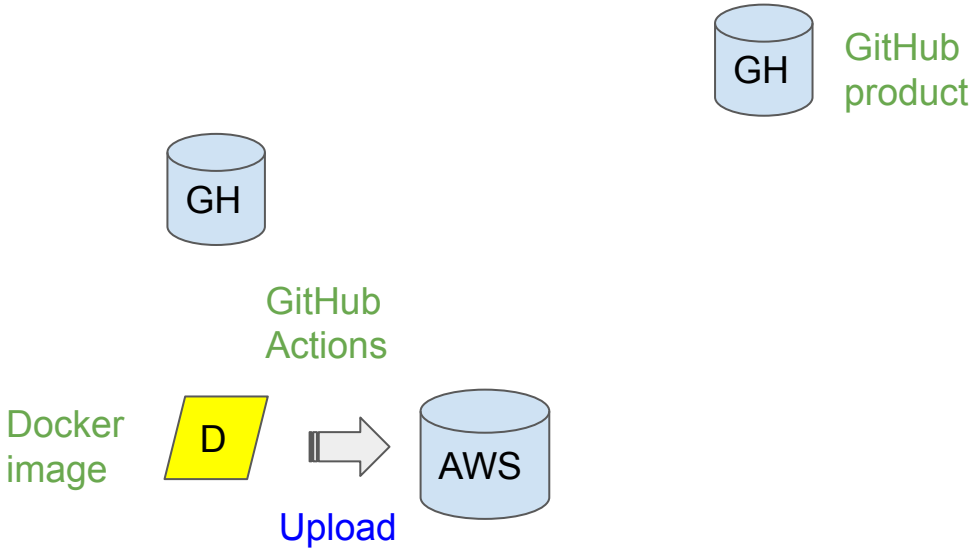
GitHub Actions: Сборка проекта автотестов запущена



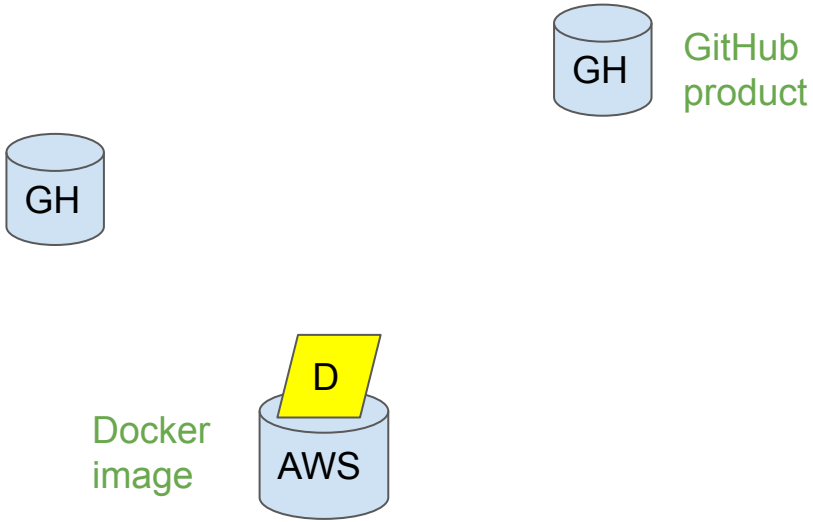
CI: GH Actions собирает Docker image



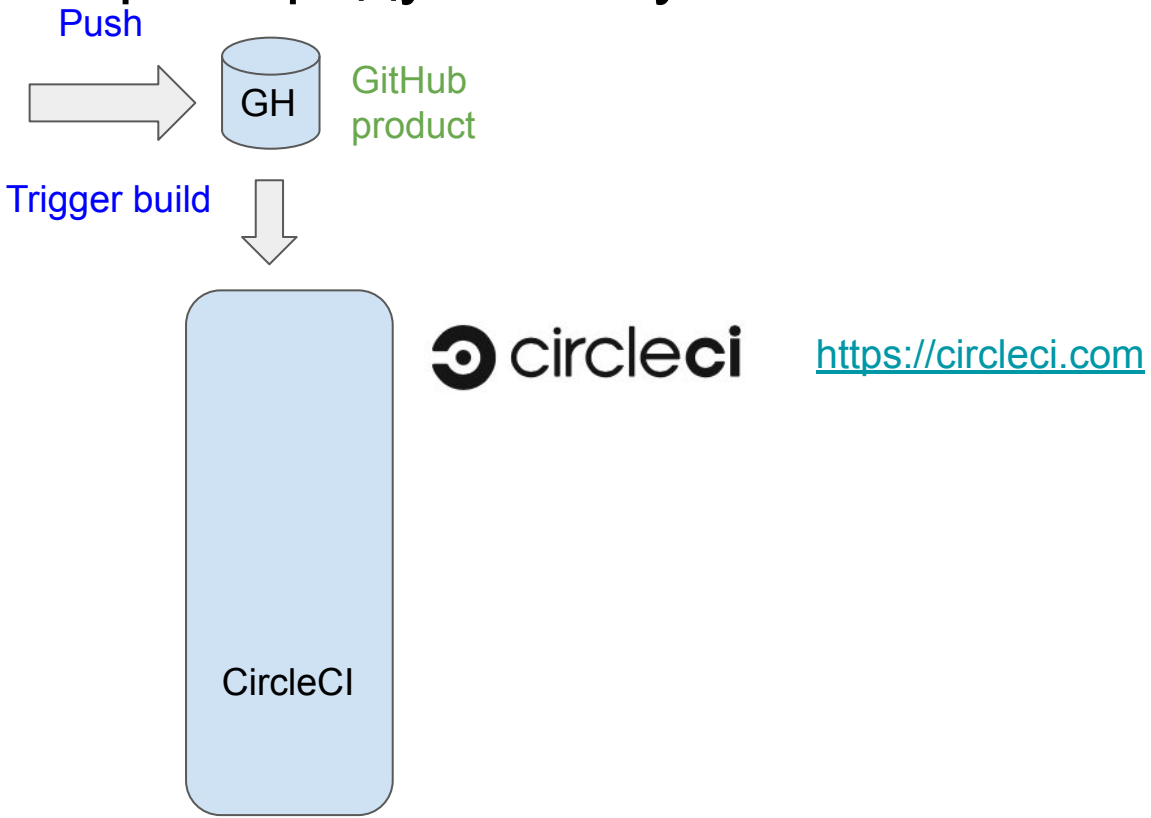
CI: GH Actions загружает образ на AWS



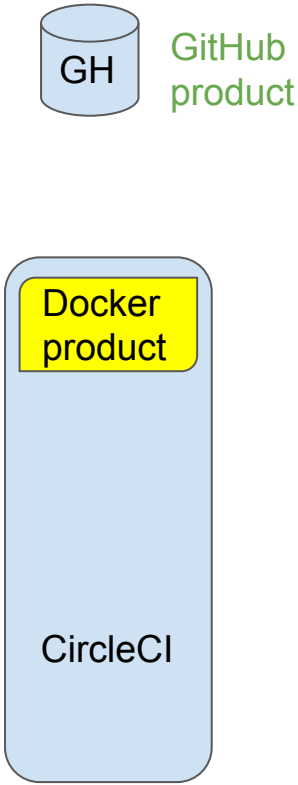
CI: AWS содержит свежую версию autoqa



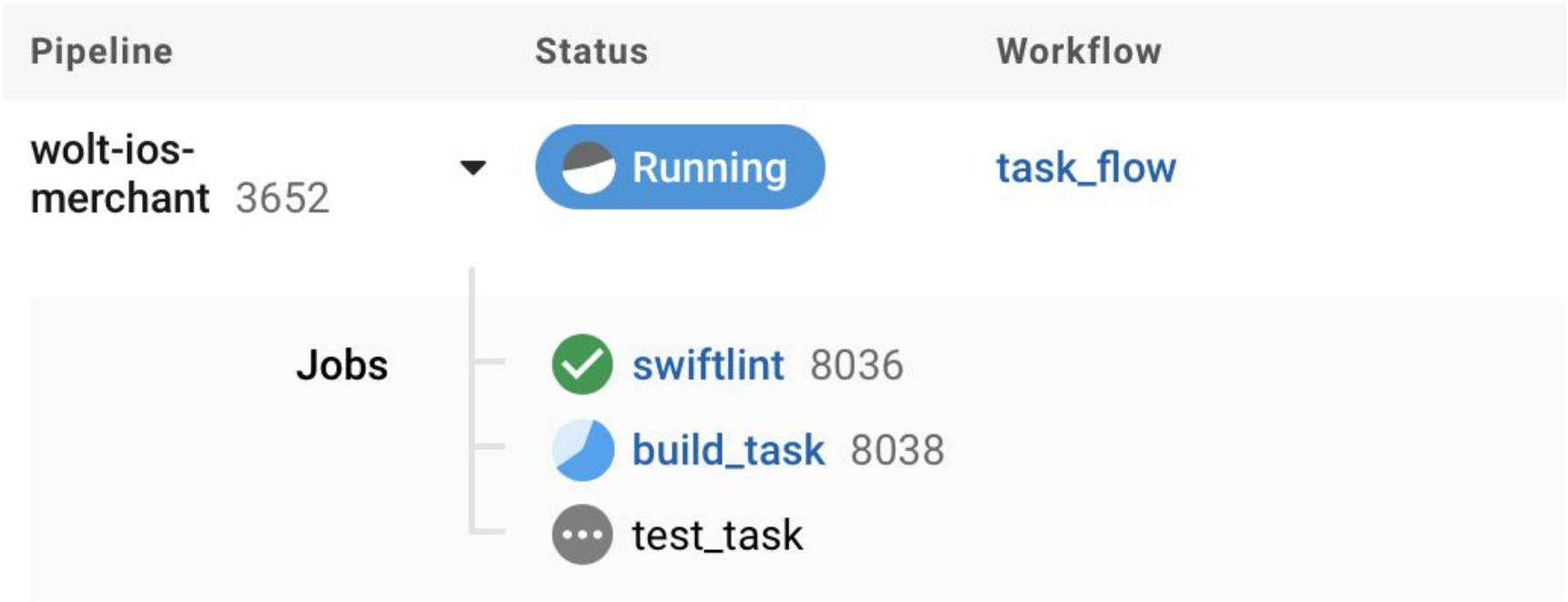
CI: Push в репозиторий продукта запускает CircleCI



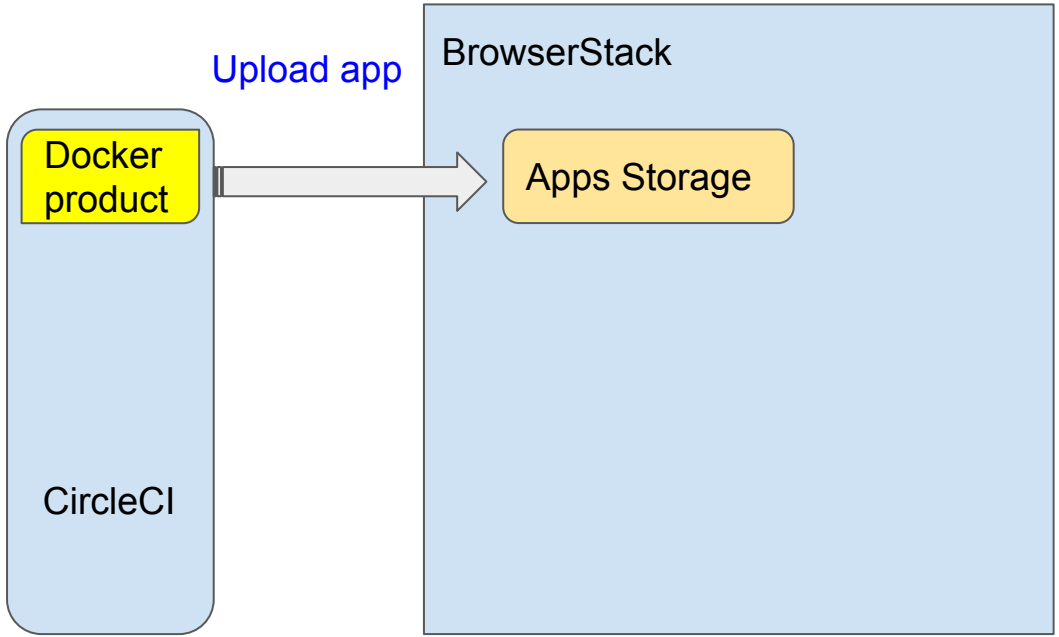
CI: Сборка продукта в контейнере на базе macOS



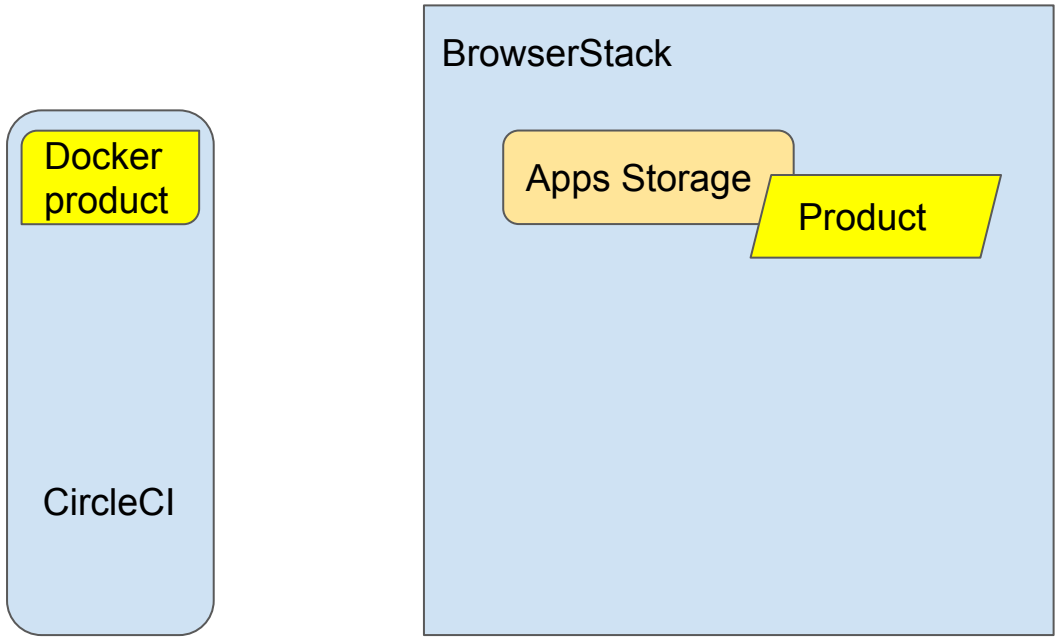
CircleCI: Начало сборки продукта



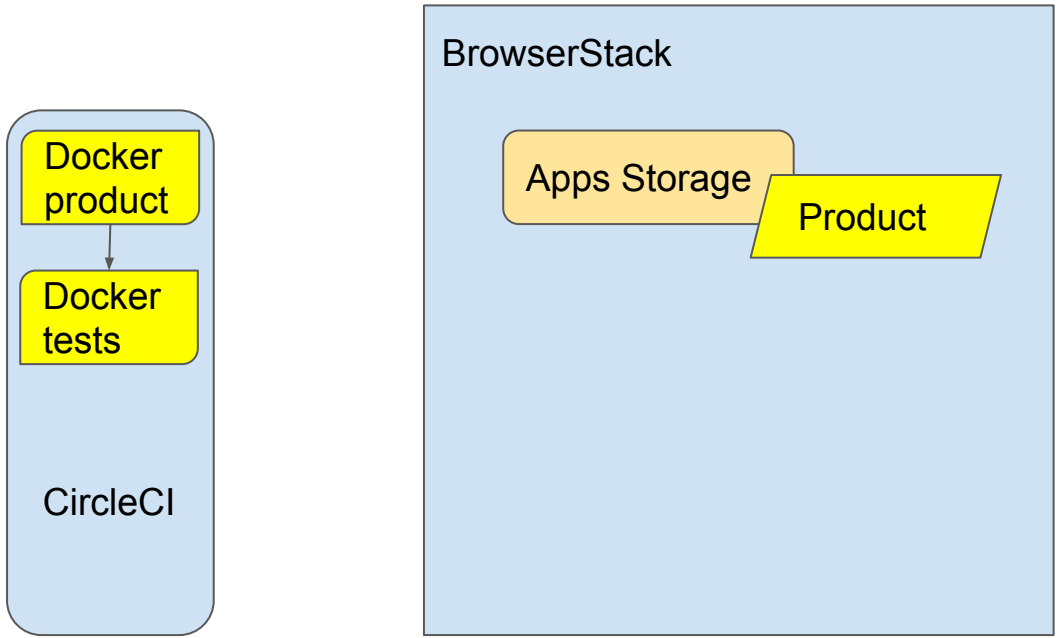
CI: Загрузка продукта в облачное хранилище



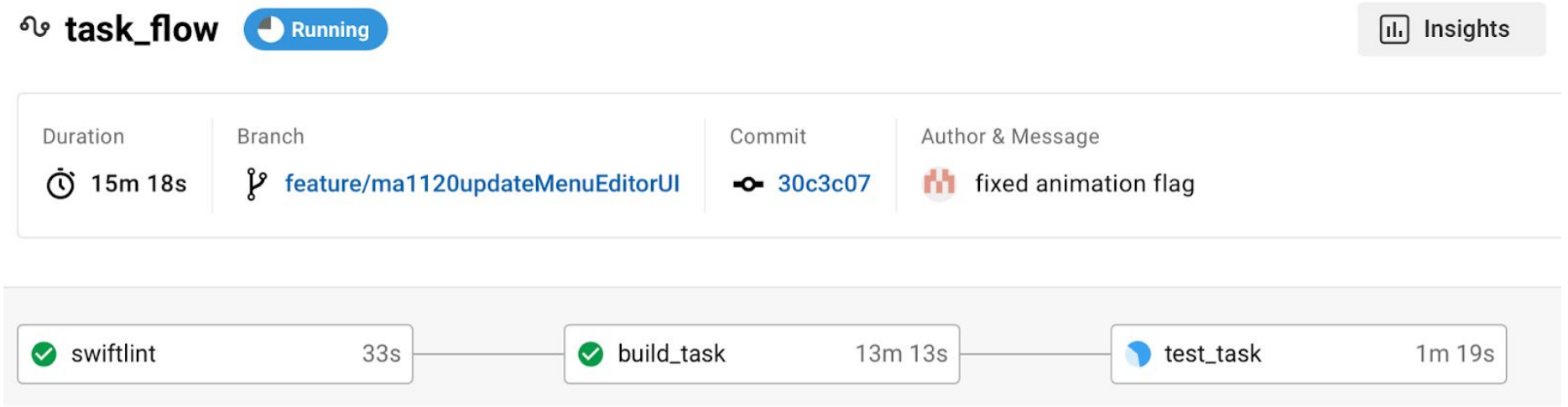
CI: Продукт готов к использованию на BS



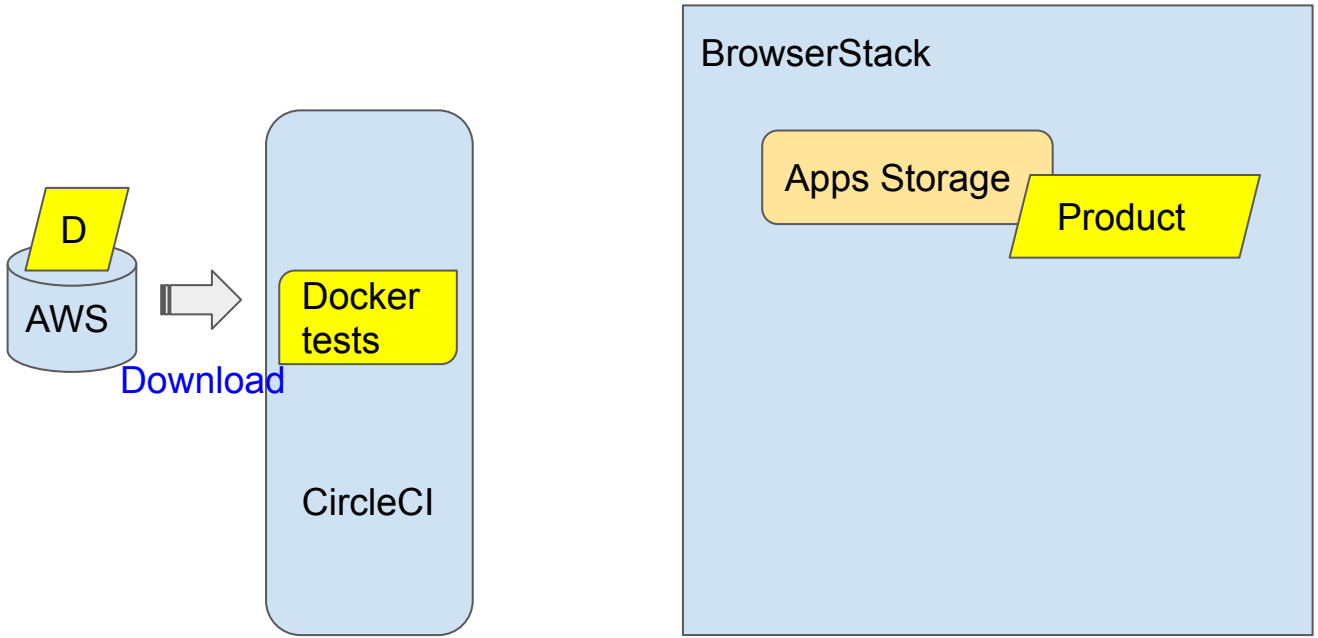
CI: Начало прогона тестов после сборки продукта



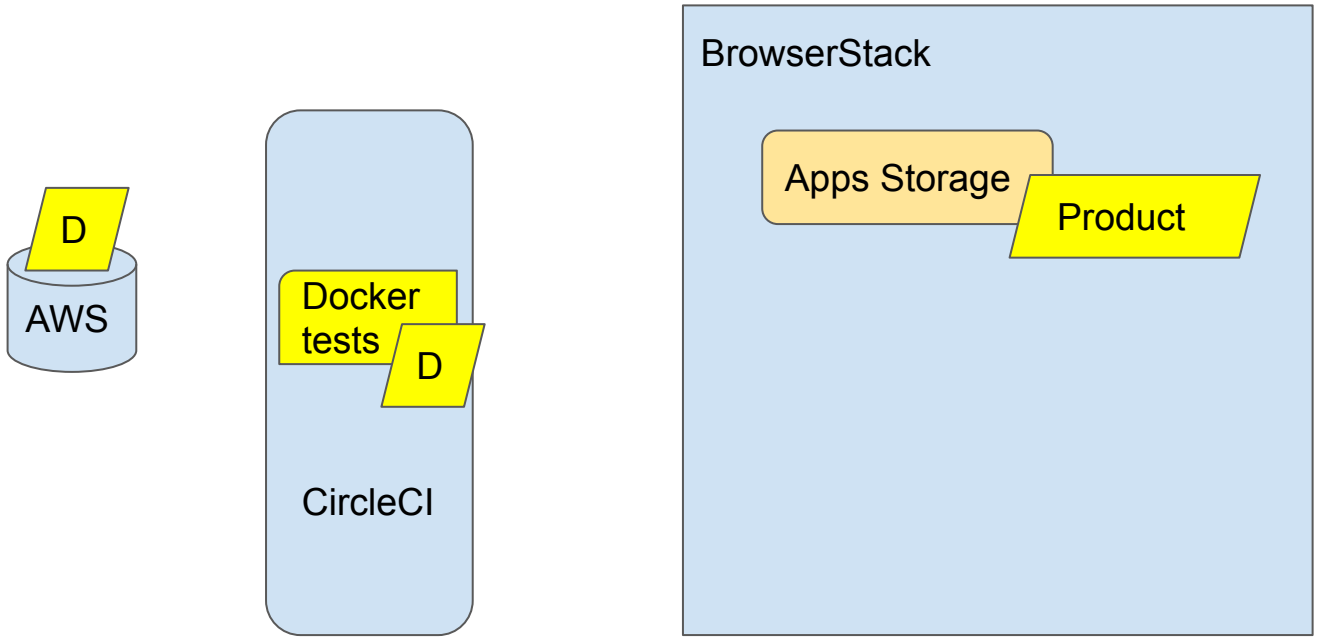
CircleCI: Job с тестами запущен



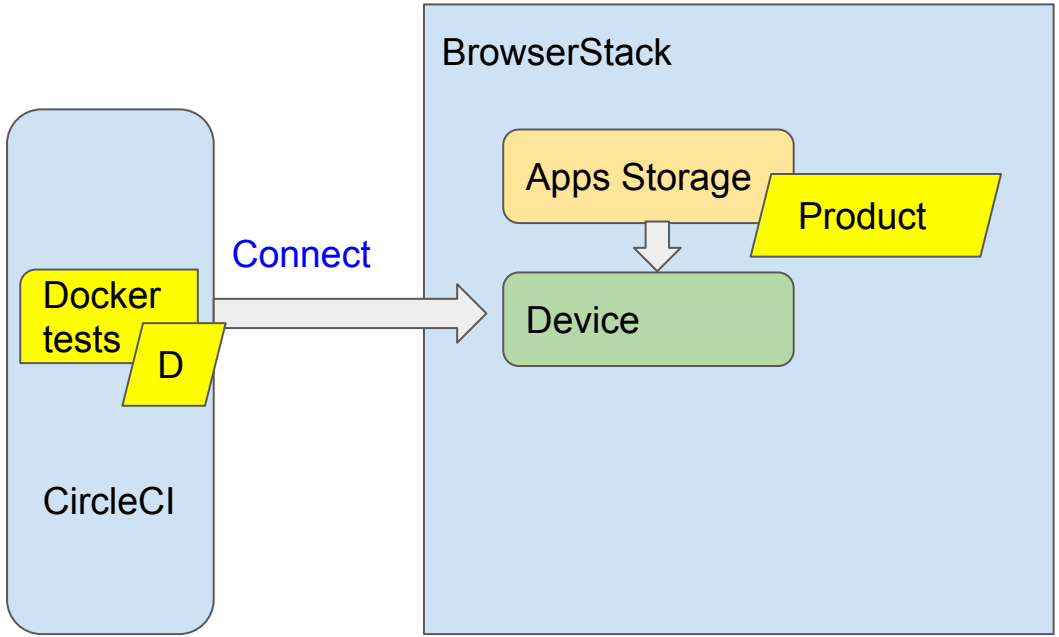
CI: Загрузка последней версии autoqa



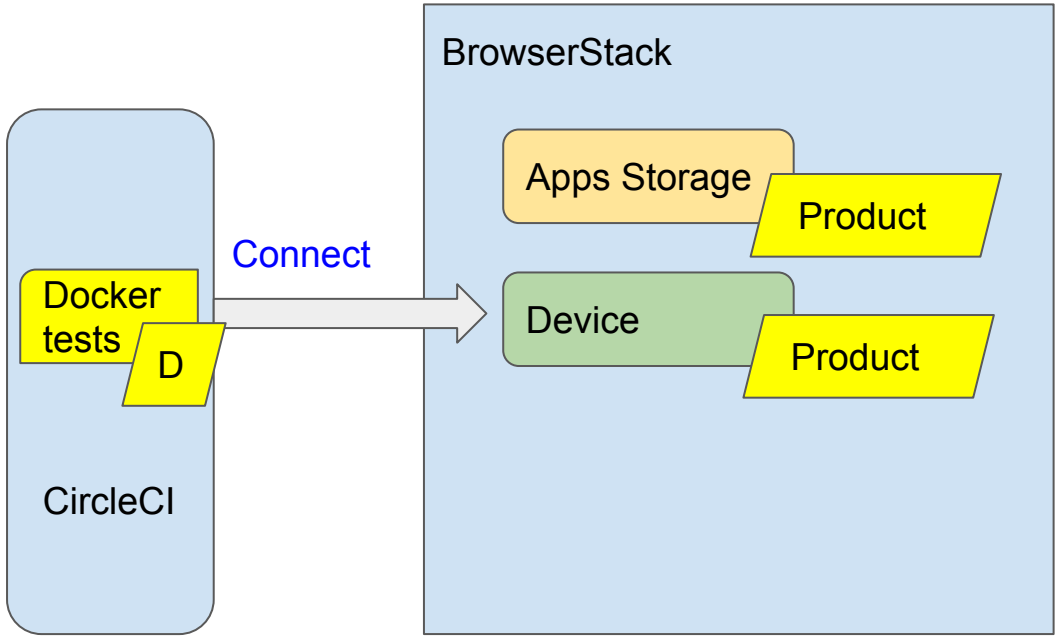
CI: Тесты autoqa запущены внутри контейнера



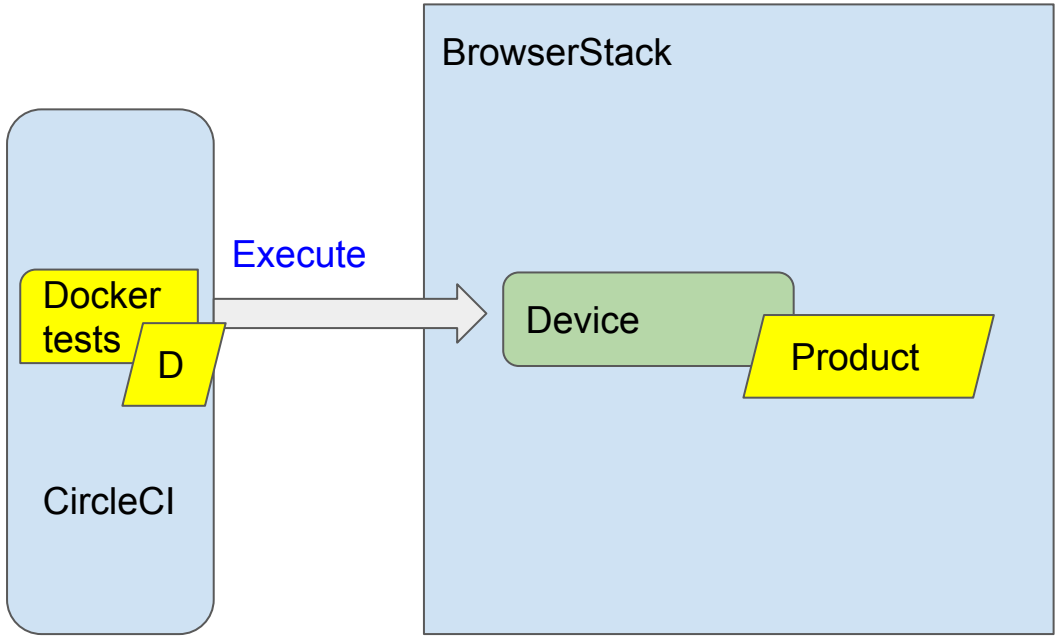
CI: Подключение к устройству и установка продукта



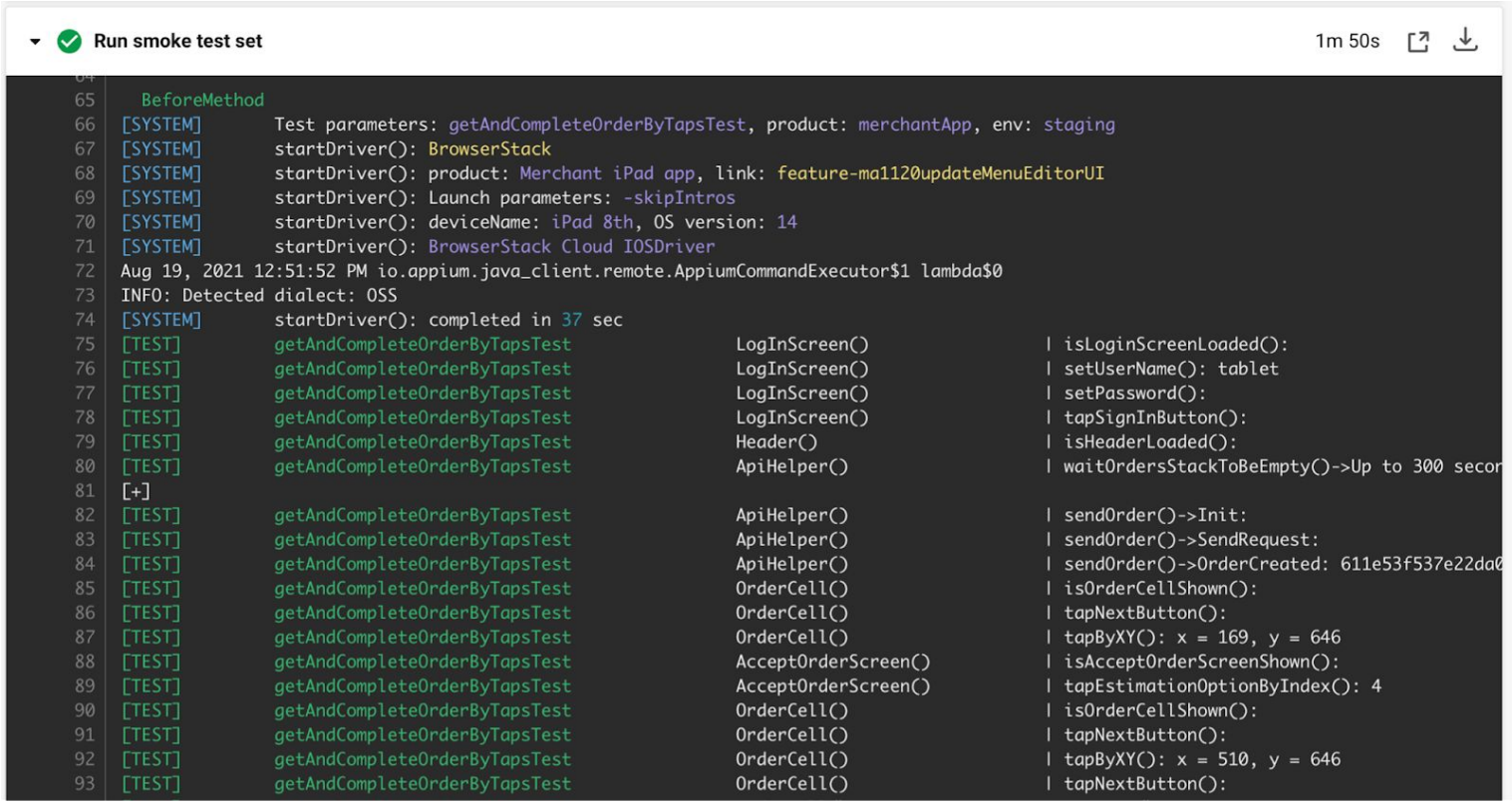
CI: Устройство с продуктов готово к использованию



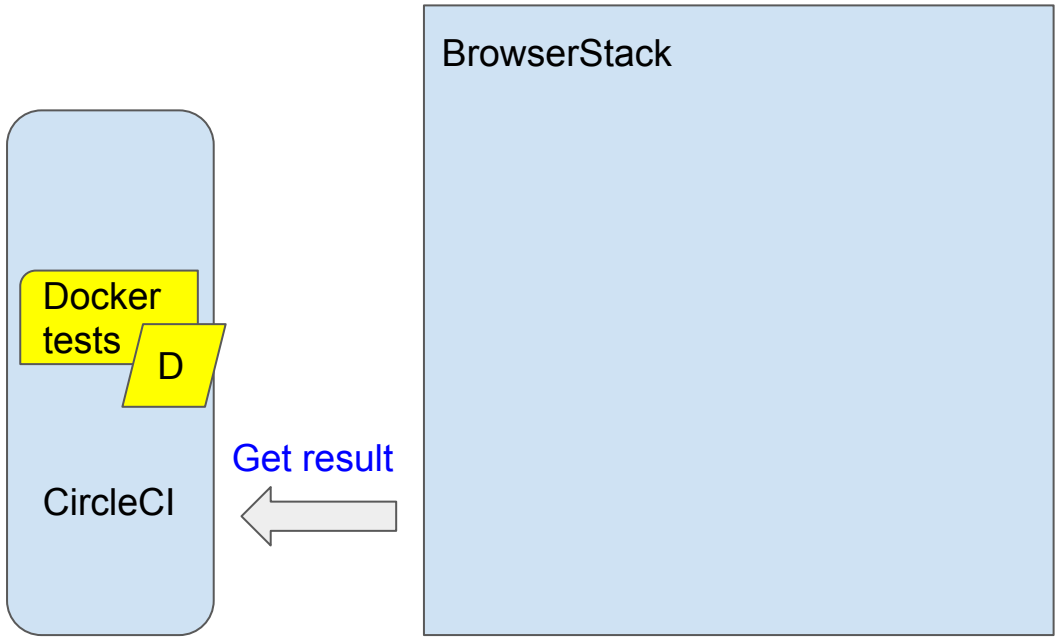
CI: Выполнение UI тестов



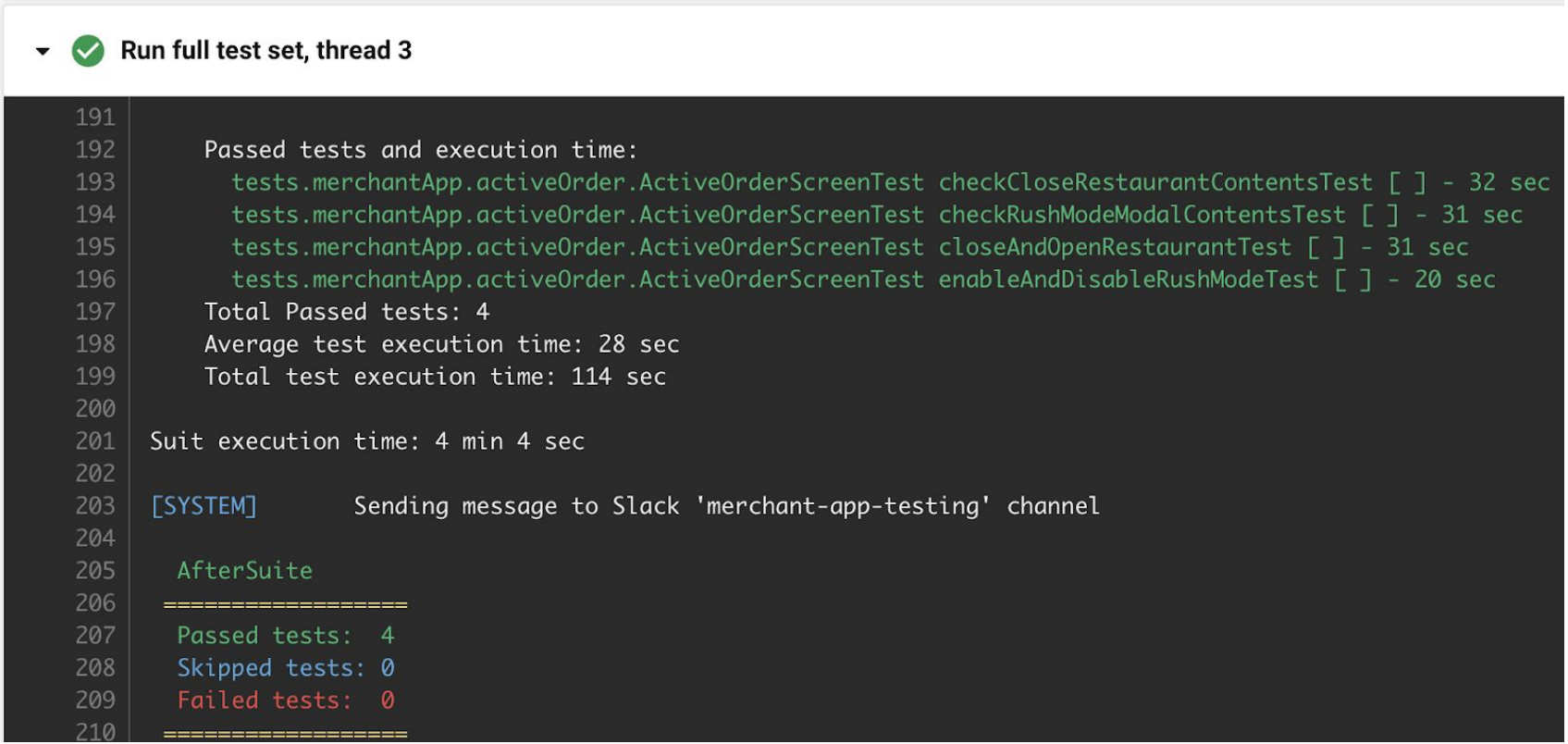
CircleCI: Просмотр лога autoqa в свойствах Job



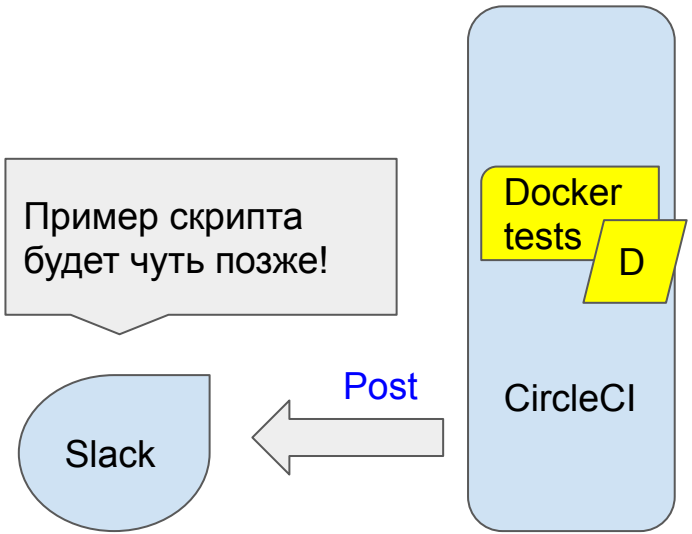
CI: Получение результатов тестирования



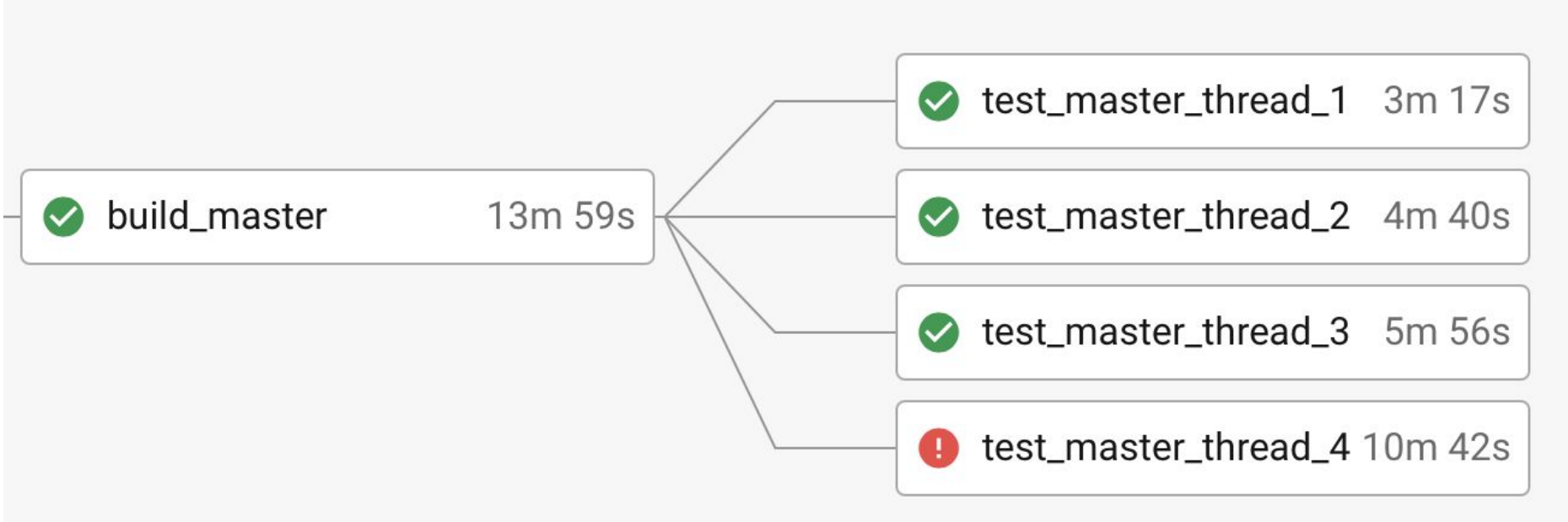
CircleCI: Сводный отчет о тестировании в Job



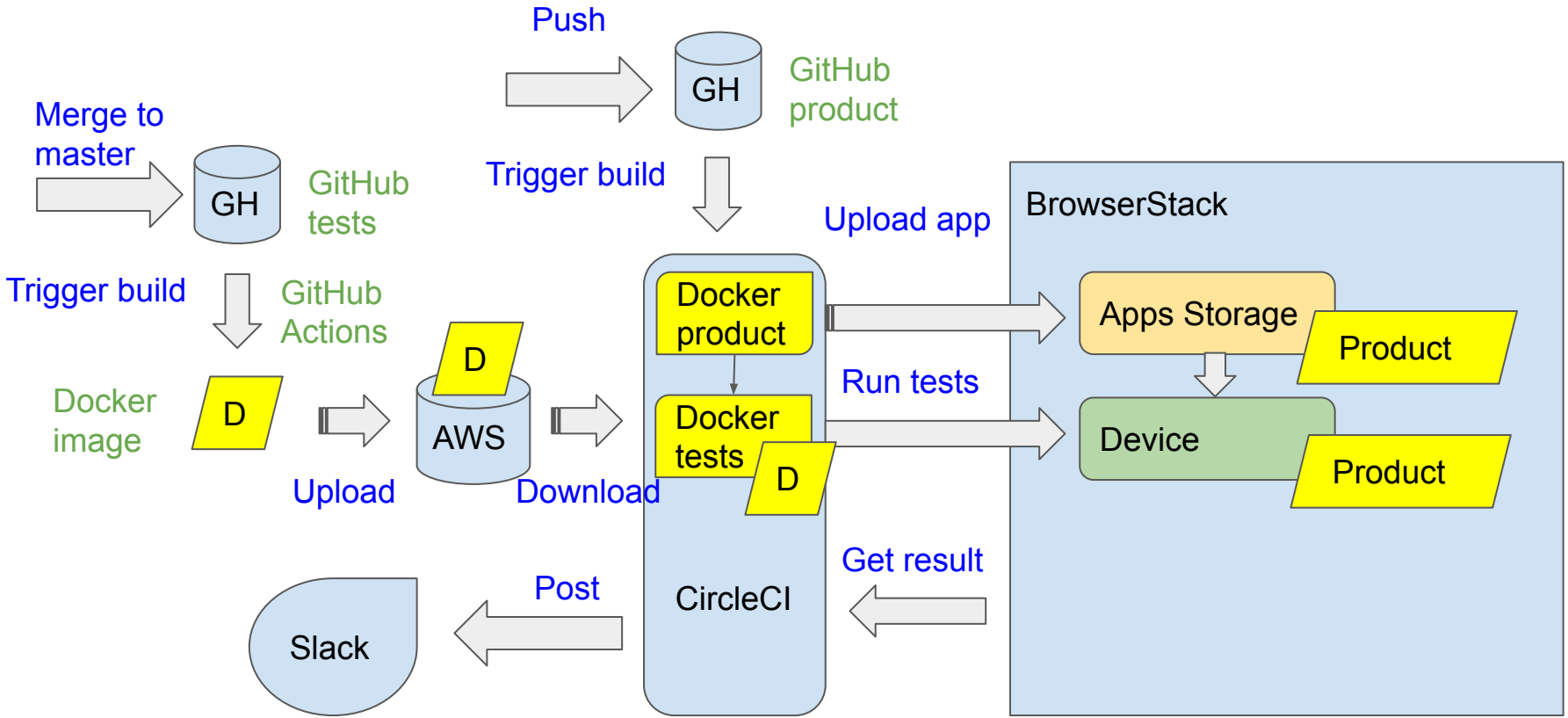
CI: Отправка сообщения в Slack с результатами



CircleCI: Сборка и тестирование завершены



CI: Общая схема



CI: Погружаемся в детали

Разберем детально, как работает этот план. И как, собственно, его реализовать.

GitHub Actions

Имплементация:

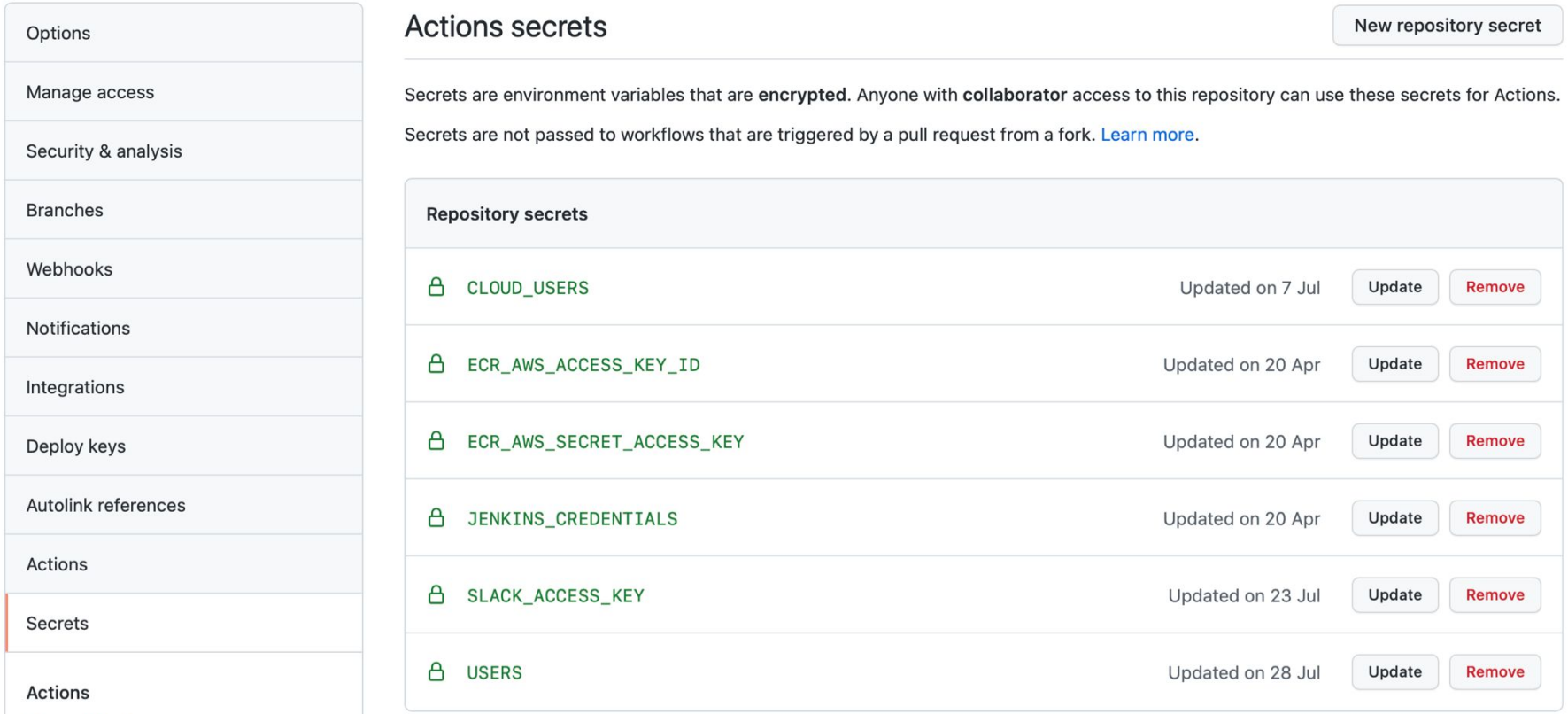
- Задание переменных

- Создание Конфигурационного файла

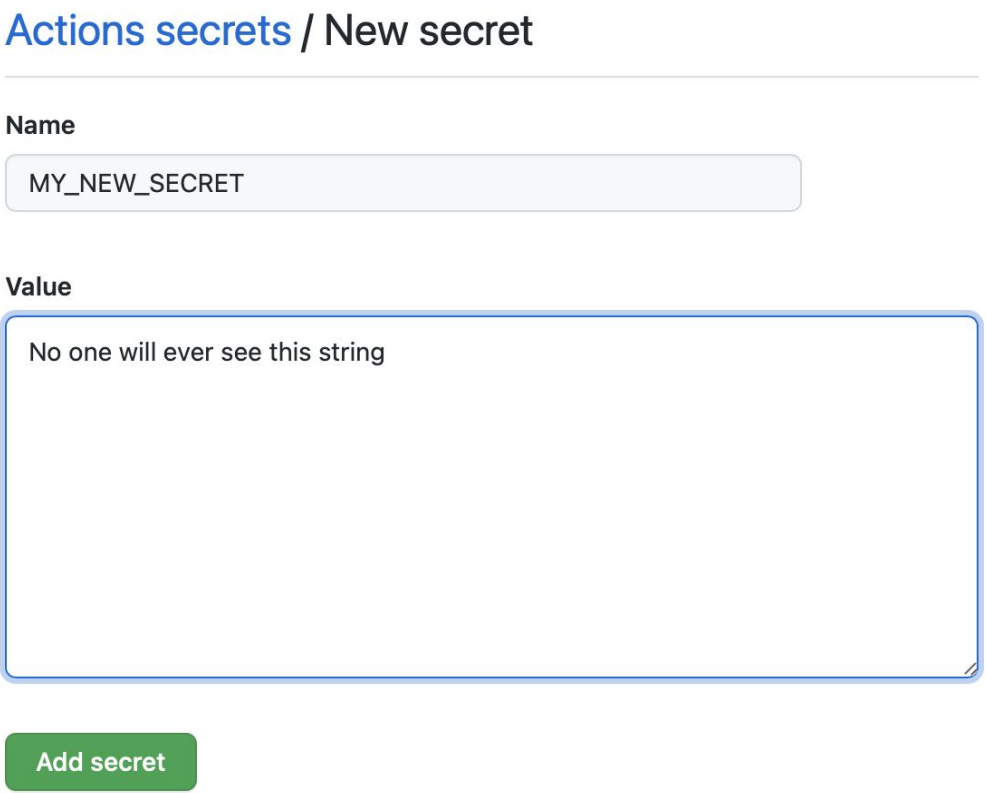
- Сборка на GitHub: раздел Action

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

GH Actions: Secrets



GH Actions: Создание нового секрета



GH Actions: Конфигурационный файл

```
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: настройка - импорт данных

```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: настройка - сборка образа

```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 }}
```

Загрузка приложения в облако

В облаке у нас уже есть тесты, но еще нет продукта на котором их запускать.

Загрузка файла приложения в облако

```
curl -u "USERNAME:ACCESSKEY"

-X POST "https://api-cloud.browserstack.com/app-automate/upload"

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

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

custom_id - это уникальный идентификатор приложения в облачном хранилище. Так вы можете хранить сколь угодно много разных версий вашего приложения одновременно.

Fastlane

Fastlane - это инструмент для автоматизации процессов сборки и выкладки приложений

Сайт: 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
```

Использования Fastlane для загрузки файла

```
desc "Upload build to BrowserStack"

lane :uploadToBSfromCI do |options|

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

    browserstack_app_name = options[:cloudLink]

    curl = "curl -u \"" + browserstack_login + "\" -X POST \"" + browserstack_url + "\" -F \"file=@" + ipa_path + "\" -F
\'data={\"custom_id\": \"" + browserstack_app_name + "\"}\'"

    UI.message(curl)

    app_upload_response = sh(curl)

    UI.message(app_upload_response)

    end
```

Запуск тестов на загруженном приложении

Теперь у нас есть команда для загрузки приложения.

Расширим пайплайн, собирающий наше приложение на CI.

CircleCI

Для чего мы его используем:

- Имеет готовые образы с разными версиями XCode
- Имеет встроенные функции для сборки iOS приложения и работы с сертификатами

CircleCI

- Расширим этап сборки приложения, добавив его загрузку в BrowserStack

- Добавим новый job с исполнением автотеста:
  - Загрузка Docker - образа из Amazon
  - Запуск теста в образе

CircleCI: добавление этапа загрузки в облако

```
# Building the app
  build_master:
    macos:
      xcode: 12.5.0
    environment:
      FL_OUTPUT_DIR: output
    shell: /bin/bash --login -o pipefail
    steps:
      - checkout
      - fix-xcode-checkouts
      - run:
          name: Install components
          command: bundle install
      - run:
          name: Build ipa file, upload to AppCenter and BS:MerchantAppMaster
          command: |
            bundle exec fastlane buildAndUpload
            bundle exec fastlane uploadToBSfromCI cloudLink:MerchantAppMaster
```

CircleCI: прогон тестов на загруженном приложении

```
test_master:
  docker:
    - image: eu-west-1.amazonaws.com/autoqa:1
  steps:
    - run:
        name: Run full test set
        command: mvn -DsuiteName=suites/App/full.xml test
```

CircleCI: добавление тестов в pipeline

```
workflows:
# When merging to master branch
  master_flow:
    when:
        and:
          - equal: [ master, << pipeline.git.branch >> ]
    jobs:
      - swiftlint
      - build_master:
          requires:
            - swiftlint
      - test_master:
          requires:
            - build_master
```

CIrcleCI: Нельзя просто так взять и запустить...

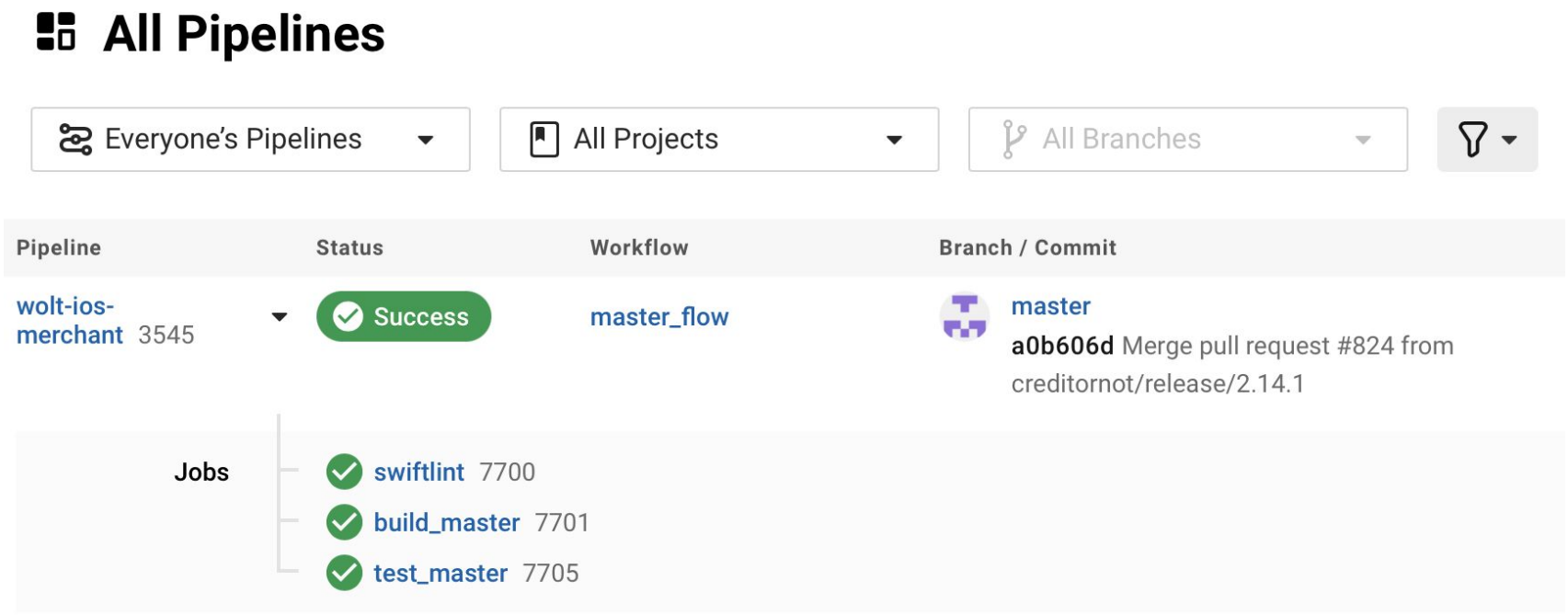
```
test_master:
  docker:
    - image: eu-west-1.amazonaws.com/autoqa:1
  steps:
    - run:
        name: Run full test set
        command: mvn -DsuiteName=suites/App/full.xml test
```

- CircleCI всегда запускает код в корне системы
- … а тесты лежат в подкаталоге

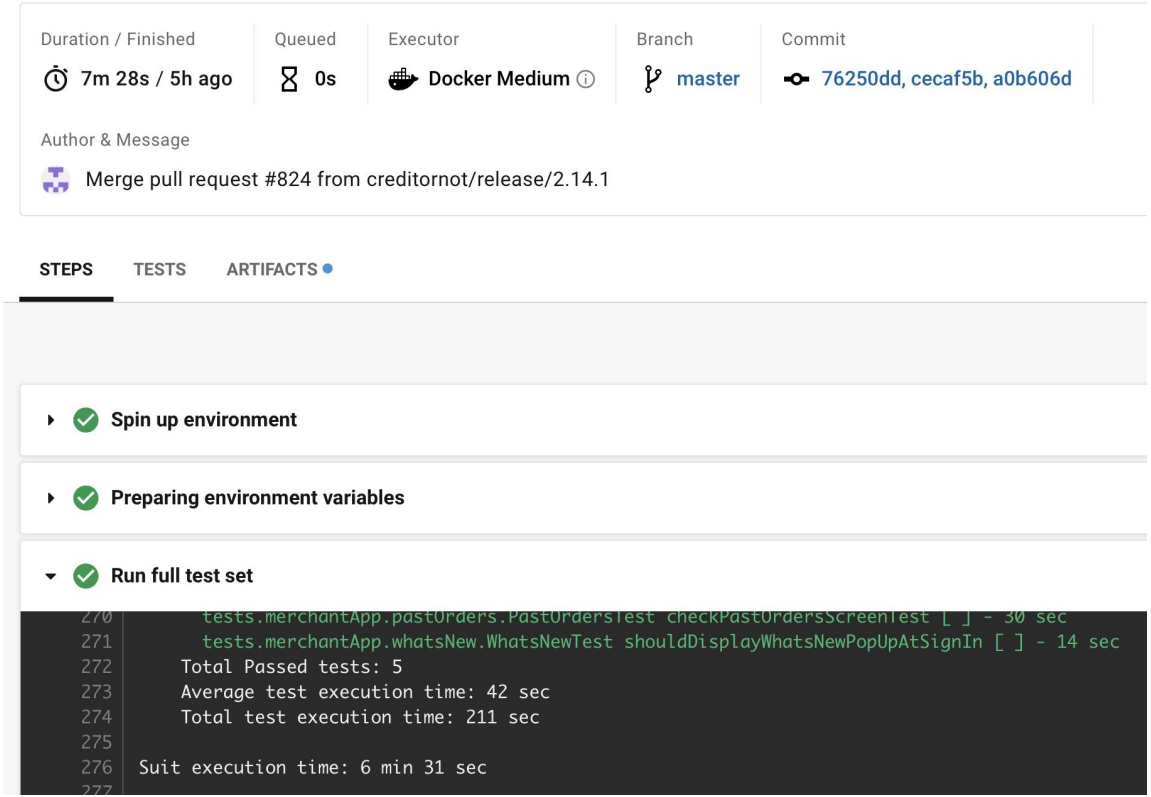
CircleCI: Прогон тестов - исправление =)

```
test_master:
  docker:
    - image: eu-west-1.amazonaws.com/autoqa:1
  steps:
    - run:
        name: Run full test set
        command: cd /usr/tests && mvn -DsuiteName=suites/App/full.xml test
```

CircleCI: Список выполненных job в пайплайне



CircleCI: Информация о пайплайне

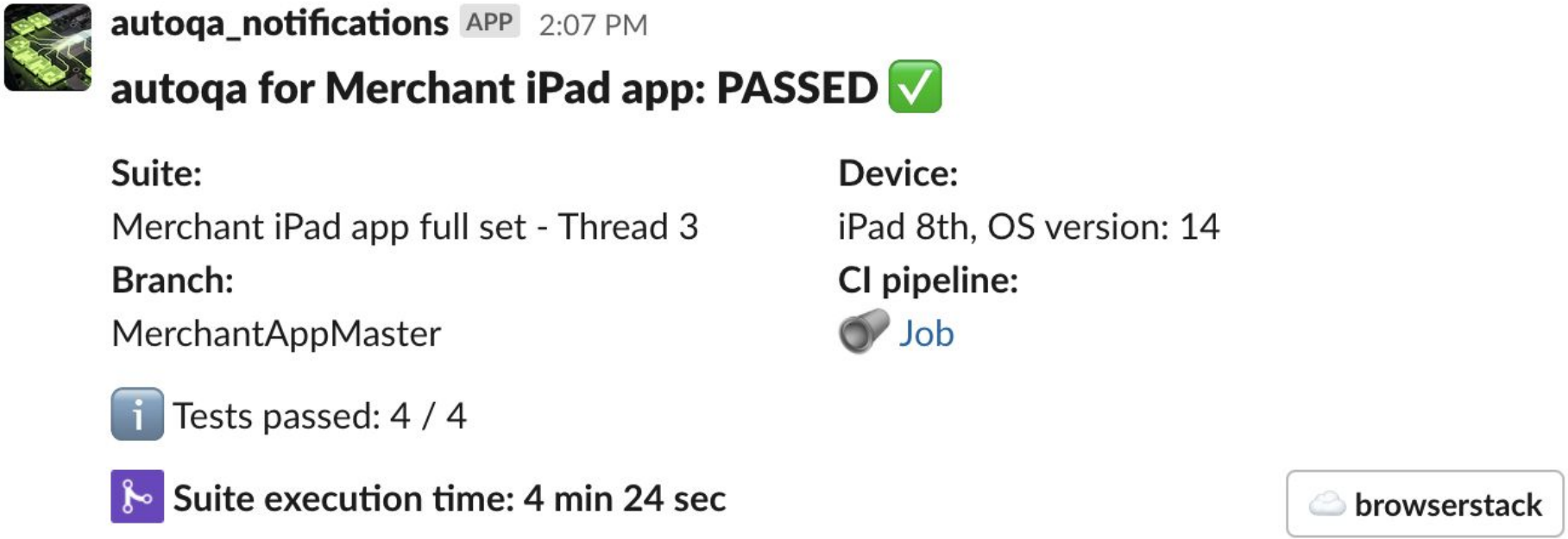


Отправка сообщений в Slack

Что полезно добавить в сообщение:

- Статус (Все прошло или есть ошибки)
- Сведения о прогоне: ветка GitHub, окружение, suite
- Ссылка на пайплайн CI
- Ссылка на прогон на BrowserStack

Пример сообщения в Slack с результатами теста



Slack: Порядок настройки

- Создание Slack-бота для отправки сообщение и настройка его прав доступа

- Создание канала (channel)

- Имплементация кода, который будет формировать сообщение и отправлять его в соответствующий канал в определенной форме (JSON)

Slack - полезные ссылки

- BrowserStack API: получение информации и прогонах и отдельных сессиях:

https://www.browserstack.com/docs/app-automate/api-reference/appium/sessions#get-session-details

- Slack API: составления и отправка сообщений:

https://api.slack.com/authentication/basics

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

- Конструктор сообщений Slack:

https://app.slack.com/block-kit-builder/

Slack: пример кода для отправки сообщения

```
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}'"
        )
    }
}
```

Slack: пример кода для отправки сообщения

Используемые инструменты:

- **JSON**: формирование тела запроса
- **okhttp**: создание, отправка и обработка HTTP запросов
- **Jackson**: мощный инструмент работы с JSON объектами

+ **enum** класс SlackChannel: для удобного хранения констант

CI: итоговый порядок исполнения

Проект автотестов:

- Сборка образа тестов с GitHub Actions при merge to master
- Публикация образа на сетевое хранилище (Amazon)

CI: итоговый порядок исполнения

Проект приложения:

- Сборка приложения
- Загрузка дистрибутива в BrowserStack
- Скачивание последнего образа с тестами
- Запуск тестов внутри контейнере на загруженном приложении в облаке
- Публикация результатов

Список использованной литературы

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 Action: https://github.com/features/actions

CircleCI: https://circleci.com

Browserstack: https://www.browserstack.com

Репозиторий с примером проекта

Тестовый проект с реализованной структурой Page Factory, использующий управление тестами TestNG и умеющий отправлять сообщения в Slack.

GitHub: https://github.com/heavy-razzer/heisenbug-autoqa-demo

autoqa: Mobile UI test automation framework

Время удивительных вопросов и неожиданных ответов. Или наоборот.