Что такое Appium?

Appium — это свободно распространяемый фреймворк с открытым исходным кодом, предназначенный для тестирования пользовательского интерфейса мобильных приложений. Он помогает тестировать нативные, гибридные и веб-приложения и проводить автоматизированное тестирование на физических устройствах, а также с помощью эмулятора и симулятора. Он предлагает кросс-платформенное тестирование приложений — единый API работает для сценариев тестирования платформы Android и iOS.

Он не отягощен зависимостью от операционной системы мобильного устройства — у Appium есть фреймворк или оболочка, которые переводят команды Selenium Webdriver в команды UIAutomation (iOS) или UIAutomator (Android) в зависимости от типа устройства, а не любого типа ОС.

Appium поддерживает все языки с клиентскими библиотеками Selenium, такие как: Java, Objective-C, JavaScript с node.js, PHP, Ruby, Python, C# и т. д.

#### **Как работает Appium?**

* Appium — это «HTTP-сервер», написанный с использованием платформы Node.js и управляющий сессиями iOS и Android с использованием проводного протокола Webdriver JSON. Следовательно, перед инициализацией Appium Server в системе необходимо предварительно установить Node.js.
* Когда Appium скачан и установлен, на компьютере настраивается сервер с REST API.
* Он получает запрос на подключение и команду от клиента и выполняет эту команду на мобильных устройствах (Android/iOS).
* Он отвечает ответами HTTP. Опять же, чтобы выполнить этот запрос, он использует платформы автоматизации мобильного тестирования для управления пользовательским интерфейсом приложений. Фреймворк вроде:

— Apple Instruments для iOS (Instruments доступен только в Xcode 3.0 или более поздних версиях с OS X v10.5 и более поздних версиях).

— Google UIAutomator для Android API уровня 16 или выше.

— Selendroid для Android API уровня 15 или ниже.

## Описание

Кроссплатформенный фреймворк с открытым кодом для автоматизации тест-кейсов мобильных приложений — нативных, гибридных и веб-приложений. Создавался изначально для тестирования на Android и iOS, как на основных мобильных платформах, со временем удобство и удачность фреймворка привели к созданию WAD-драйвера (WinAppDriver) для тестирования десктопных Windows-приложений. Проект начинался как инструмент командной строки, устанавливаемый через node,js, со временем появился удобный интерфейс, инспектор элементов, и прочие удобства.

### Три вида мобильных приложений

### Нативные

Традиционно требовательные к «железу» и ориентированные на производительность, написанные на «языках семейства С» и с помощью соответствующих SDK. Разработка обходится дорого, но такие приложения:

* Быстрые и отзывчивые
* Выглядят приятно
* Хорошо интегрируются с аппаратной частью
* Интуитивный UI/UX

Стоит посмотреть на Spotify, Snapshot, Pinterest и Skype на смартфоне, чтобы понять, что нативное приложение при прочих равных — лучше.

### Веб-приложения

Вместе с тем, у нативных приложений есть некоторые недостатки; бывают ситуации, когда лучше веб-приложение; оно запускается в мобильном браузере, поэтому не требуется его (иногда усложненная) установка, а разработка намного дешевле и быстрее. Разрабатываются такие приложения на HTML/CSS/JavaScript, отсюда преимущества:

* Малая стоимость разработки
* Проще поддержка
* Не обязательна официальная «приемка» в аппстор
* Не нужна инсталляция

Например, мобильный Aliexpress это веб-приложение.

### Гибридные

Сочетают преимущества нативных и веб-приложений. Скачиваются из аппстора и имеют доступ ко всем нужным функциям смартфона. Представляют собой «веб-приложение в контейнере». Тоже разрабатываются на HTML/CSS/JS, и могут запускаться на любой платформе. Поэтому имеют заметные преимущества:

* Одна кодовая база на всех платформах
* Быстрая разработка
* Удобное масштабирование и перенос между платформами
* Доступ ко всем аппаратным функциям смартфона

Гибридные приложения: Basecamp, Instagram.

В Appium можно тестировать все три типа мобильных приложений, на обеих платформах, и этим он удобен.

## Философия Appium

В процессе разработки создатели фреймворка старались учесть недостатки существовавших на то время тестовых фреймворков «первого поколения» и создали **4 принципа Appium**:

1. Для автоматизации приложения не нужно рекомпилировать его и вообще как-то модифицировать.
2. Не должно быть привязки к какому-то языку или фреймворку.
3. При автоматизации API не нужно изобретать колесо — лучше воспользоваться имеющимися лучшими решениями.
4. Фреймворк должен быть с открытым кодом.

## Компоненты Appium

### Клиент-серверная архитектура

В «сердце» Appium — веб-сервер с REST API. Он получает запросы от клиента, слушает его команды, выполняет их на девайсе, и отправляет HTTP-ответ с результатом выполнения. Клиент-серверная архитектура имеет преимущество: можно писать тесты на любом ЯПе, поддерживающем клиентский API по HTTP (а проще воспользоваться одной из библиотек Appium). Можно поставить сервер на другой машине, не той на которой выполняются тесты. Можно хранить код тестов на облачном сервисе типа Sauce Labs.

### Сессии

Клиенты открывают сессию с сервером, каждая библиотека имеет особенности, общее то что все они отправляют запрос Post/session на сервер, с JSON-объектом, имеющим название “desiredcapabilities”. Сервер откроет сессию автоматизации и отправит ID сессии, который будет использован при отправке будущих команд.

### DesiredCapabilities

Это набор ключей и значений (например в форме **map**или **hash**), отправляемых Appium-серверу, и описывающих, какую открыть сессию автоматизации, или как изменить поведение сервера во время сессии. Например, отправляется набор platformName в iOS, чтобы открыть iOS-сессию (а не Android). Или свойство safariAllowPopups ставится в положение true, и тогда во время сессии в Safari JS будет разрешено открывать всплывающие окна.

### Appium Server

Написан на node.js. Может быть собран и установлен из исходников или прямо из NPM.

### Appium Clients

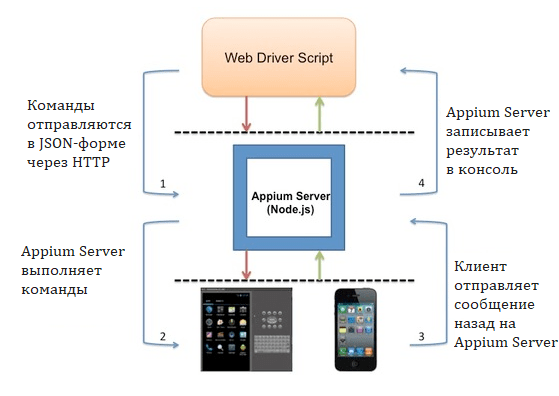
Клиентские библиотеки (на Java, Ruby, Python, PHP, JavaScript, C#), поддерживающие расширения WebDriver-протокола для Appium. Они работают вместо стандартного WebDriver-клиента.

### Appium.app и Appium.exe

GUI-обертки вокруг Appium-сервера, поставляемые вместе со всем необходимым ему, включая Инспектор, контролирующий иерархию приложения.

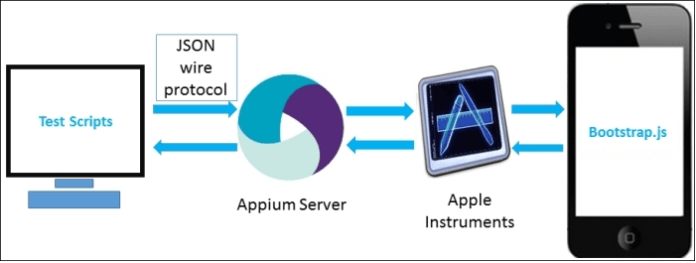
## Архитектура Appium

Appium это HTTP-сервер, написанный на node.js, который создает WebDriver-сессии для iOS и Android. Appium запускает тест-кейс на девайсе, то есть запускает там сервер и слушает через прокси команды от основного Appium-сервера. Тестировщик пишет сценарии, выполняемые на девайсе/эмуляторе путем отправки запросов/ответов на Appium-сервер.



### iOS

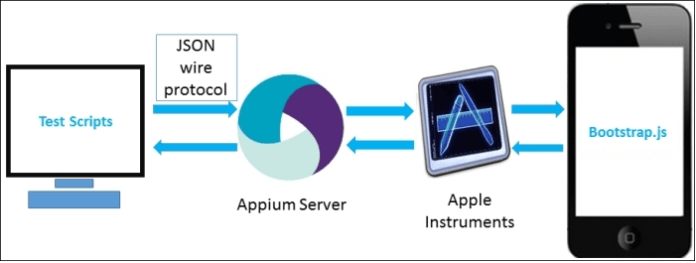
Appium «проксирует» команду в скрипт UIAutomation, запускаемый в окружении Mac Instruments. Оно предназначено для профилирования, контроля и билда приложений, а также имеет компонент для автоматизации. В нем можно писать команды на JavaScript, и через UIAutomation API команды взаимодействуют с интерфейсом. Автоматизация Appium работает через эти библиотеки.



Выше изображена архитектура Appium в контексте автоматизации на iOS. Жизненный цикл команды выглядит так: веб-драйвер Selenium принимает команду в коде и отправляет ее, в JSON-форме, HTTP-запросом на Appium-сервер. Этот сервер обрабатывает контекст (iOS или Android) и направляет команду на сервер команд Instruments (работающий на node.js), тот ее «подхватывает» и выполняет через bootstrap.js в iOS-окружении Instruments. После выполнения команды клиент отправляет Appium-серверу ответ, который с деталями записывается в логи.

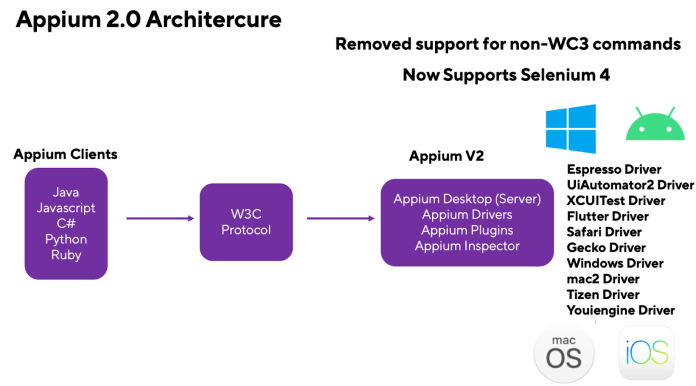
### Android

Примерно так же, Appium направляет через прокси команду в UIAutomator на девайсе. UIAutomator —  нативный фреймворк UI-автоматизации, поддерживающий запуск тест-кейсов на JUnit непосредственно на девайсе из командной строки. Основной язык Java, и возможны все другие ЯПы, поддерживаемые WebDriver.



На диаграмме имеем bootstrap.jar (вместо bootstrap.js), здесь тест-кейс, скомпилированный из Java-кода. При его выполнении запускается TCP-сервер на девайсе и клиент в Appium-процессе.

В прошлом году вышел Appium 2.0, его архитектура изображена ниже:



## Преимущества и недостатки

### Плюсы

* Бесплатный и открытый
* Кроссплатформенность, гибридные и нативные приложения
* Совместим с JSON-веб-драйвером и Selenium Grid
* «Облачное тестирование» через [testdroid](https://www.wikiwand.com/en/Testdroid)
* Поддержка C#, Java, PHP, Python, Ruby
* Удобная автоматизация
* Не нужна рекомпиляция
* Поддерживает симуляторы, эмуляторы и реальные девайсы одновременно
* Инспектор для записи и воспроизведения тестов
* Поддержка протокола JSON wire
* Большое комьюнити с советами
* Хорошая поддержка Android выше версии 4.1

### Минусы

* В iOS не поддерживается несколько сценариев в нескольких симуляторах одновременно
* Android ниже версии 4.1 не поддерживается (хотя таких приложений уже меньше 3%)
* В чем-то ограниченная поддержка гибридных приложений (но тут смотря что считать «гибридным»: можно переключаться между webview- и нативным контекстами, включив debug для WebView)
* Поддержка жестов до сих пор не идеальна
* Не очень полная документация
* Иногда проблемы с изображениями
* В версии для Windows иногда глючит Инспектор

### Сравнение Appium с Selendroid и Robotium

|  |  |  |
| --- | --- | --- |
| **Appium** | **Selendroid** | **Robotium** |
| Открытый код. Бесплатный | Открытый код. Бесплатный | Открытый код. Бесплатный |
| Поддерживает и iOS, и Android | Только Android | Только Android |
| Не нужно видеть код приложения и библиотеки | Нужен код приложения | Нужен код приложения |
| Не нужно реинсталлировать приложение | Нужно реинсталлировать | Небольшое изменение в коде Robotium приводит к полному ребилду |
| Поддерживает много фреймворков и языков программирования | Совместим с Selenium и Jenkins | Только Java. Не поддерживает Selenium |
| Большое доброе комьюнити | Хуже чем у Appium | Небольшое комьюнити |

### Сравнение Appium и Espresso

Как следует из описания Espresso на официальном сайте, тестовый фреймворк Espresso предоставляет API для UI-тестов пользовательских автоматизированных сценариев. Espresso отличается тем, что предназначен Google для функциональных тестов Android-интерфейсов. Espresso изначально приспособлен к автоматизации белого ящика, имея доступ к коду приложения.

Appium же, изначально делался кроссплатформенным и приспособленным к «черному ящику», имея дело с тестами «внешней стороны» приложения. Через фреймворк UIAutomator Appium запрашивает UI-элементы интерфейса.

## **Что такое локатор**

Локатор — обычный текст, которой идентифицирует себя как элемент DOM’а страницы. Простым языком: с помощью локатора на странице можно найти элементы. В случае CSS — локатор включает в себя набор уникальных атрибутов элемента, а в случае XPath — это путь по DOM’у к элементу.

У элемента может быть родитель и ребёнок. Родитель может быть один, а детей может быть несколько. Если детей несколько, то они являются соседями и каждый из них образует свою ось. 1 ребёнок = 1 ось со своими особенностями и своими вложенными элементами. А — родитель, B D E F W X Y — дети A. У каждого элемента есть свои дети, свои дальнейшие ветки, это и называется оси.

**Начнём с разбора неуникальных локаторов.** Если по локатору находятся 2 и более элементов, то такой локатор можно назвать неуникальным. Тест при обнаружении большого количества элементов по данному локатору упадёт или возьмёт первый.

**Уникальный, но non-suitable локатор**. Если мы в DevTools введем вышеуказанные названия, то найдется элемент. И здесь мы опускаемся до следующего уровня написания локаторов — уровня понятности, читаемости и надёжности локатора.

**Уникальные локаторы.**Они найдут только один элемент, они осмысленные, иногда читабельные и краткие. Как раз уникальные атрибуты — это class, id, name и подобные.

## **Поиск элементов с помощью XPath**

XPath в корне отличается от CSS как идеей, так и реализацией. XPath — это полноценный язык для поиска элементов в дереве, причём неважно каком, будь это XML или XHTML. Можно использовать XPath в веб-страницах, нативной мобильной вёрстке и других инструментах.

XPath — это путь от элемента к элементу. Можно представить, что структура тегов — это дерево каталогов, как в любой ОС.

Особенностью xpath является начало селектора с / или //, что означает:

* / - переход поочередно по элементам
* // - возврат всех элементов по совпадению

|  |  |
| --- | --- |
| **Плюсы XPath** | **Минусы XPath** |
| - полноценный язык для поиска элементов не только в вебе, но и в других средах и документах  - позволяет перемещаться по дереву вниз и вверх  - гибко работает с осями элементов  - есть очень много функций, которые помогают в поиске локаторов, например, поиску по тексту в элементе или аналог normalize-space, который убирает пробелы у строки по бокам | - громоздкий  - нечитабельный  - сложен в освоении  - работает дольше, чем поиск по CSS, хоть и незначительно |

Поиск по Xpath

Сигнатура:

findElement(By.xpath(String XPath));

Поиск по name

Сигнатура метода:

findElement(By.name(String Name));

@AndroidFindBy(id = "ru.filit.mvideo.b2c.debug:id/find\_similar\_products\_button")  
private WebElement getSimilarBtn;  
  
@AndroidFindBy(id = "ru.filit.mvideo.b2c.debug:id/similar\_products")  
private WebElement shelfSimilarProductsBlock;  
  
@AndroidFindBy(xpath = "//\*[contains(@resource-id, 'similar\_products')]//\*[contains(@resource-id, 'collection\_title')]")  
private WebElement similarTitle;  
  
@AndroidFindBy(xpath = "//\*[(@resource-id = 'ru.filit.mvideo.b2c.debug:id/similar\_products')]//\*[@resource-id = 'ru.filit.mvideo.b2c.debug:id/product\_layout']")  
private List<MaterialCardWidget> similarProductCards;  
  
@AndroidFindBy(xpath = "//\*[(@resource-id = 'ru.filit.mvideo.b2c.debug:id/similar\_products')]//\*[@resource-id = 'ru.filit.mvideo.b2c.debug:id/collection\_see\_all']")  
private WebElement seeAllSimilar;  
  
@AndroidFindBy(id = "ru.filit.mvideo.b2c.debug:id/not\_available\_products\_button")  
private WebElement notAvailableBtn;  
  
@AndroidFindBy(xpath = "//\*[(@resource-id = 'ru.filit.mvideo.b2c.debug:id/similar\_products')]//\*[@resource-id = 'ru.filit.mvideo.b2c.debug:id/collection\_list']")  
private WebElement shelfSimilarProductsListBlock;  
  
@AndroidFindBy(xpath = "//\*[contains(@resource-id,'collection\_title')]")  
private WebElement shelfTitleElement;  
  
@AndroidFindBy(id = "ru.filit.mvideo.b2c.debug:id/accessories\_shelf")  
private WebElement productAccessoriesBlock;  
  
@AndroidFindBy(xpath = "//\*[contains(@resource-id, 'accessories\_shelf')]//\*[contains(@resource-id, 'collection\_title')]")  
@iOSXCUITFindBy(xpath = "//\*[contains(@label,'С этим товаром покупают')]")  
private WebElement accessoriesTitle;  
  
@AndroidFindBy(xpath = "//\*[(@resource-id = 'ru.filit.mvideo.b2c.debug:id/accessories\_shelf')]//\*[@resource-id = 'ru.filit.mvideo.b2c.debug:id/product\_layout']")  
private List<MaterialCardWidget> accessoryProductCards;  
  
@AndroidFindBy(xpath = "//\*[contains(@resource-id, 'accessories\_shelf')]//\*[contains(@resource-id, 'collection\_list')]")  
private WebElement accessoryListBlock;  
  
@AndroidFindBy(xpath = "//\*[(@resource-id = 'ru.filit.mvideo.b2c.debug:id/accessories\_shelf')]//\*[@resource-id = 'ru.filit.mvideo.b2c.debug:id/collection\_see\_all']")  
private WebElement seeAllAccessoryBtn;  
  
@AndroidFindBy(id = "ru.filit.mvideo.b2c.debug:id/upsale")  
private WebElement shelfUpsaleBlock;  
  
@AndroidFindBy(id = "ru.filit.mvideo.b2c.debug:id/upsale\_collection\_title")  
private WebElement upsaleTitle;  
  
@AndroidFindBy(xpath = "//\*[(@resource-id = 'ru.filit.mvideo.b2c.debug:id/upsale')]//\*[@resource-id = 'ru.filit.mvideo.b2c.debug:id/product\_layout']")  
private List<MaterialCardWidget> upsaleProductCards;  
  
@AndroidFindBy(id = "ru.filit.mvideo.b2c.debug:id/upsale\_collection\_see\_all")  
private WebElement seeAllUpsaleBtn;  
  
@AndroidFindBy(xpath = "//\*[contains(@resource-id, 'upsale')]//\*[contains(@resource-id, 'collection\_list')]")  
private WebElement upsaleListBlock;

### Верификация изменения состояния элементов

Пожалуй, эта практика одна из самых важных в мобильной автоматизации в принципе, потому что в приложениях очень редко встречаются статические элементы, которые сразу же отображаются на экране и всегда на нём доступны. Чаще мы сталкиваемся с тем, что загрузка элементов занимает какое-то время.

Например, в случае медленного интернета элементы, получаемые с сервера, отображаются с существенной задержкой. И если мы попытаемся их проверить до того, как они появятся, тесты будут падать.

Таким образом, прежде чем начать проверять элементы, нам необходимо дождаться их появления на экране. Естественно, эта проблема не нова и существуют стандартные решения.

**Ожидания** — не самый приятный процесс, где бы он ни происходил. Ожидания в автотестах — отдельная боль. Избавиться от них невозможно, но есть способы организовать их работу в более удобной форме, нежели Thread.sleep().

Implicit Wait

Implicit Wait, или неявное ожидание, — пожалуй, самый популярный способ ожидания в Selenium благодаря своей легкости в использовании.

Чтобы использовать Implicit Wait в автотестестах, достаточно:

установить его всего 1 раз,

указать вручную лимит ожидания.

После того, как команда исполнится, Implicit Wait будет действовать на протяжении всего пробега автотестов и ожидать указанное время прежде, чем выбросить NoSuchElementException (или не выбрасывать, если необходимый элемент на странице найден). Не устанавливать Implicit Wait равносильно нулевому лимиту времени, и исключение пробросится сразу.

Explicit Wait

Explicit wait, или явное ожидание, чаще используется для ожидания определенного условия, которое должно быть выполнено прежде, чем тест пойдет дальше.

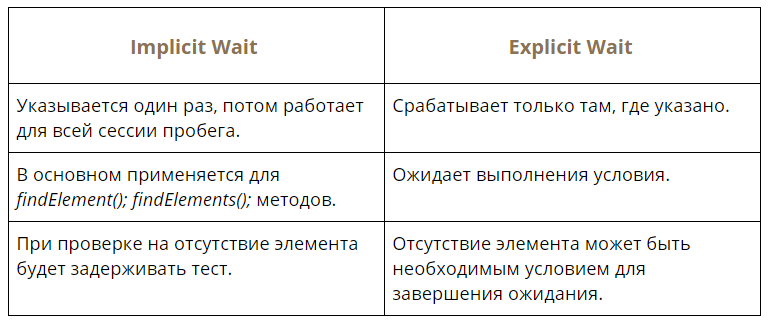
О явном ожидании стоит помнить следующие вещи:

ожидание сработает именно там, где оно указано;

как и неявному ожиданию, ему необходимо указать лимит времени;

ожидает выполнения необходимого условия

## Разница между Implicit и Explicit Wait

[](https://media.tproger.ru/uploads/2020/11/table.png)

## Написание собственных ожиданий

Ожидания Selenium не всегда способны удовлетворить потребности тестировщика. В таких случаях мы можем сами написать методы, которые удержат автотесты от падения. Работать кастомные ожидания будут по тому же принципу, что и Explicit Wait, т.е. срабатывать в той части теста, в которой указаны, а условие, которого необходимо дождаться, нужно написать самим как некий аналог класса ExpectedConditions. Хранить ли свои условия в отдельном классе или в том же, где и написанный способ ожидания, и нужно ли разделять метод ожидания и условие или достаточно будет оставить их прописанными в едином методе, зависит от условий проекта и предпочтений тестировщика.