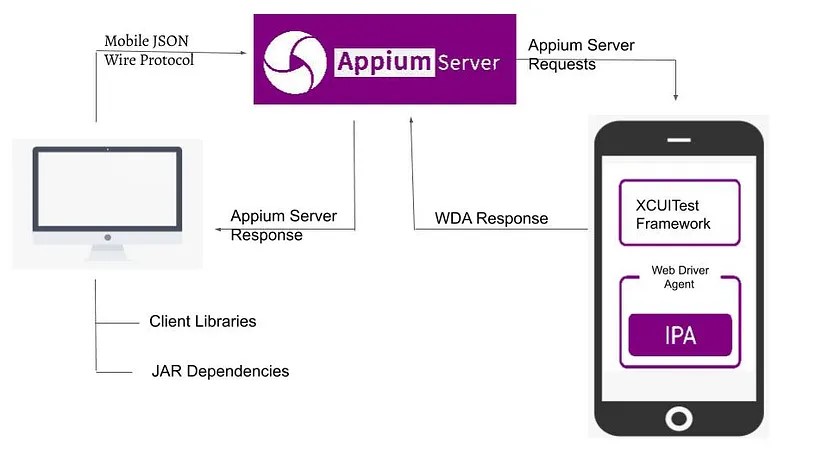
.What is an Appium

**Appium** is an open-source, cross-platform tool for automating mobile application testing on Android and iOS devices.It supports native, hybrid, and mobile web apps etc.By leveraging the WebDriver protocol, Appium enables developers and testers to write test scripts in multiple programming languages (like Java, Python, Ruby, and JavaScript).

Appium is an open source test automation framework used for mobile automation testing across different platforms such as iOS and Android. It also supports mobile web browser automation and has a wide variety of language support for automation.

# Appium Architecture



Appium is an HTTP server written in **[nodejs](https://nodejs.org/en/" \t "https://medium.com/swlh/_blank)** that exposes REST API. The client communicates with Appium server via REST API’s and it is handled by Mobile JSON Wire Protocol.

Steps that invloved how a client library converts user-written commands into REST API requests that are acceptable to Appium(to comver the user-written command in client libray JSON wire protocal play a crucial role)

### Step 1: Parsing User-written Commands

The client library first parses the user-written commands from the test script. This involves analyzing the syntax and semantics of the language used by the test script.

Step 2: Mapping Commands to REST API Methods

The client library maps the parsed commands to REST API methods that are supported by Appium. For example:

A command like driver.click(elementId) would be mapped to a GET request with a specific path, such as /element/<elementId>/click.

A command like driver.findElement(By.id("username")) would be mapped to a GET request with a path like /element/by/id/username.

### Step 3: Serializing Command Data into JSON Payload

The client library serializes the command data into a JSON payload that can be used as the body of a REST API request. For example:

For the driver.click(elementId) command, the JSON payload might look like this: {"click": {"elementId": "12345"}}.

For the driver.findElement(By.id("username")) command, the JSON payload might look like this: {"findElement": {"by": "id", "value": "username"}}.

### Step 4: Constructing REST API Request

The client library constructs the REST API request by combining the serialized JSON payload with the mapped REST API method and any other required parameters, such as authentication credentials, headers, and URL parameters.

### Step 5: Sending REST API Request to Appium Server

The client library sends the constructed REST API request to the Appium server using HTTP. The server then processes the request and responds accordingly.

### Step 6: Handling Server Response

The client library receives the response from the Appium server and handles it accordingly. For example, if the server responds with a JSON object containing the results of the command, the client library might parse the JSON object and update the test state accordingly.

Note: The JSON Wire Protocol provides a standardized framework that ensures consistent communication between client libraries and the Appium server. Each step in converting user-written commands into REST API requests leverages this protocol, from parsing commands and mapping them to appropriate REST API methods to serializing these commands as JSON payloads for HTTP requests, all the way to interpreting the responses in JWP-compliant JSON formats

## Platform automation behaviour

how does Appium map this protocol to automation behaviour on a wide range of platforms?

Appium doesn't! It leaves this responsibility up to a kind of software module called an Appium driver. The driver is kind of like a pluggable module for Appium that gives Appium the power to automate a particular platform (or set of platforms, depending on the goal of the driver). A driver's responsibility is to simply implement an Appium-internal interface representing the WebDriver protocol.

How it implements this interface is totally up to the driver, based on its strategy for making automation happen on a specific platform.

The Appium driver that supports iOS app automation is called the [XCUITest Driver](https://github.com/appium/appium-xcuitest-driver) because ultimately what it does is convert the WebDriver protocol to XCUITest library calls.

that drivers are independent, pluggable modules is that they work completely differently from one another. The tools and requirements for building and using drivers for different platforms are completely different. And so Appium lets you use just the drivers that you need for your automation tasks.

Appium has its very own [CLI for managing drivers](https://appium.io/docs/en/2.0/cli/extensions/). Management of drivers and plugins is handled by Appium's Extension CLI.

Appium provides access to automation capabilities for a given platform is that the Appium team (or anyone else**[3](https://appium.io/docs/en/2.0/intro/" \l "fn:3)**) writes a driver for that platform, implementing as much or little of the WebDriver protocol as desired. The driver can then be installed by anyone using Appium.

## **Placeholder and their meaning**

## **<ext-type>**: This refers to the "Extension type". It is a mandatory parameter that specifies the type of extension being used. The value of `<ext-type>` can be either:

## driver: This indicates that the extension is a driver, which is a module that provides a specific functionality to Appium.

## plugin: This indicates that the extension is a plugin, which is a module that extends the functionality of Appium.

Example: appium driver list (In this case extension type is a driver).

Example: appium plugin list

## All the Extension CLI commands can be used with either drivers or plugins, so it is essential to specify the type of extension being used.

## **<ext-name>**: This refers to the "Extension name". It is a short name of the extension that can be found in a call to `appium <ext-type> list`. This name is distinct from:

## The NPM (Node Package Manager) package name of the extension.

## The "install spec" of the extension (which is explained below).

Example: npm install -g appium (In this case extension name is a npm).

## The `<ext-name>` is used to identify the extension and perform actions on it, such as installing, updating, or removing it.

## **<install-spec>:** This refers to the "Install specification". It is a string used to indicate what extension Appium should install. The `<install-spec>` can be in various formats, such as:

## A URL pointing to a GitHub repository or a npm package.

## A npm package name.

## A local file path.

## A version number.

## The `<install-spec>` provides the necessary information for Appium to download and install the extension.

## **<install-source>**: This refers to the method that Appium should use to install an extension. The `<install-source>` can be one of the following:

## npm: Install the extension from the npm registry.

## github: Install the extension from a GitHub repository.

## local: Install the extension from a local file or directory.

## url: Install the extension from a URL.

## The `<install-source>` determines the installation method and provides the necessary information for Appium to retrieve the extension.

## Universal programming language access

Appium is ultimately a Node.js program.

the WebDriver specification is actually an HTTP-based protocol, meaning it is designed to be used over a network rather than within the memory of a single program.

The main benefits of this "client-server" architecture is that it allows the automation implementer (the thing doing the automation, in this case the 'server') to be completely distinct from the automation runner (the thing defining what automation should be done, in what steps, etc..., in this case the 'client'). Basically, all the "hard stuff" (actually figuring out how to make automation happen on a given platform) can be handled in one place by the server, and "thin" client libraries can be written in any programming language which simply encode HTTP requests to the server in language-appropriate way.