# **Appium Installation and Setup Guide**

## **1. Install Java JDK**

1. Download Java JDK from the [Oracle website](https://www.oracle.com/java/technologies/javase-downloads.html).
2. Install and set up the environment variables:
   * JAVA\_HOME = C:\Program Files\Java\jdk-<version>
   * Add JAVA\_HOME\bin to the PATH variable.

## **2. Install Android Studio**

1. Download and install Android Studio from the [official website](https://developer.android.com/studio).
2. Install the required SDKs and tools:
   * Open Android Studio > SDK Manager.
   * Install the latest SDK, SDK Tools, and AVD Manager.
3. Configure environment variables:
   * ANDROID\_HOME = C:\Users\<User>\AppData\Local\Android\Sdk
   * Add the following paths to PATH:
     + %ANDROID\_HOME%\platform-tools
     + %ANDROID\_HOME%\tools
     + %ANDROID\_HOME%\tools\bin

## **3. Install Appium**

1. Install Node.js from [Node.js official website](https://nodejs.org/).
2. Open Command Prompt and install Appium globally:  
    npm install -g appium
3. Verify installation:  
    appium -v

## **4. Start Appium Server**

Run the following command to start Appium with the correct base path:

**appium --base-path /wd/hub**

## **5. Install Appium Desktop (Optional)**

Download and install [Appium Desktop](https://github.com/appium/appium-desktop/releases) for a GUI-based server.

## **6. Install Appium Doctor**

Verify your setup using Appium Doctor:

npm install -g appium-doctor

appium-doctor

## **7. Install Appium Client Libraries**

For Java, install the following dependencies in your project:

<dependency>

<groupId>io.appium</groupId>

<artifactId>java-client</artifactId>

<version>8.0.0</version>

</dependency>

## **8. Create and Configure an Emulator**

1. Open Android Studio > AVD Manager.
2. Create a new virtual device and select the latest system image.
3. Start the emulator and verify using:  
    adb devices

## **9. Running the Appium Test Script**

Write your test script using Java and Appium:

DesiredCapabilities caps = new DesiredCapabilities();

caps.setCapability("platformName", "Android");

caps.setCapability("deviceName", "emulator-5554");

caps.setCapability("app", "path/to/your/app.apk");

AppiumDriver driver = new AndroidDriver(new URL("http://127.0.0.1:4723/wd/hub"), caps);

Current Capability set

**{**

**"appium:automationName": "UiAutomator2",**

**"appium:platformName": "Android",**

**"appium:platformVersion": "13",**

**"appium:deviceName": "\*\*\*\*\*\*\*\*\*\*",**

**"appium:app": "C:\\\*\*\*\*\*\\\*\*\*\*\\\*\*\*\*\*\*\*\\envoyHHT.apk"**

**}**

# **Introduction to HHT Mobile Automation- Using Appium Selenium**

This code is designed to automate the testing of our Android application using **Appium with Java in Eclipse**. It interacts with the app installed on an emulator, performing actions such as launching the app, interacting with UI elements, and validating expected results.

## **Code Breakdown:**

### **Setting Up the Appium Driver:**

* + The code initializes an **AndroidDriver** by defining the desired capabilities.
  + The capabilities specify the emulator/device name, the Android version, the app’s package and activity, and the Appium server details.
  + It ensures that Appium connects to the emulator and launches the application under test.

### **Executing Test Steps:**

* + The script interacts with the app by performing actions such as:  
    - Clicking on buttons
    - Entering text in input fields
    - Navigating through screens
    - Validating the presence of elements
  + It locates elements using **XPath, ID, Accessibility ID**, or other locator strategies.

### **Running the Test:**

* + Before executing the script, Appium must be started with the following command:

appium --base-path /wd/hub

* + This ensures that Appium listens for incoming requests and allows the script to communicate with the mobile device/emulator.
  + The test script then connects to the Appium server (http://localhost:4723/wd/hub) and starts executing the test cases.

### **Validation and Assertions:**

* + The script includes assertions to verify that actions were successful (e.g., checking if a button was clicked or text was entered correctly).
  + It logs test execution details, making it easier to debug issues if a test fails.

### **Test Completion & Reporting:**

* + After execution, the script cleans up by closing the driver.
  + Test results are logged to help analyze failures and debugging.

# HHT Automation Code Explanation

This document explains the structure and functionality of our **Appium Automation Code**, including the role of the **Test Runner** class.

## **Test Runner: Execution Flow & Design**

The TestRunner class is the **main entry point** for test execution. All test execution starts from here, ensuring a structured and modular workflow.

Execution Flow

1. **Initialization**
   * Instantiates core test modules (BaseTest, functional test classes).
   * BaseTest handles app lifecycle and authentication.
2. **Test Execution**
   * Opens the app → Logs in → Executes test modules sequentially → Logs out → Closes the app.
   * Enforces a structured, controlled test flow.
3. **Error Handling**
   * Captures exceptions (e.getCause(), e.getMessage()) for fault isolation.
   * Prevents execution failures from halting the suite.

## **Supporting Test Classes: Overview**

### **1. BaseTest**

Manages app lifecycle: opens the app, logs in, logs out, and closes the app. Ensures smooth execution flow.

### **2. FileTransfer**

Handles file-related operations within the app, ensuring seamless data movement.

### **3. Inventory\_Count**

Executes inventory counting functionalities, validating stock levels and updates.

### **4. Adjustments**

Manages inventory adjustments, ensuring correct stock modifications.

### **5. Deliveries**

Processes delivery transactions, verifying inbound stock accuracy.

### **6. PriceCheck**

Automates price verification to ensure correct pricing across inventory.

### **7. OrderInventory**

Handles inventory ordering workflows, ensuring stock replenishment.

### **8. ShelfLabels**

Manages shelf label operations, ensuring accurate product labeling.

Each class follows a **modular structure**, making the framework scalable and maintainable. 🚀

## **Conclusion**

This automation framework follows a structured approach, ensuring efficient and maintainable test execution. The **TestRunner** class serves as the central execution point, orchestrating multiple test modules. Each supporting class is modular, handling specific functionalities like file transfers, inventory management, and price checks.

By keeping the design **scalable and reusable**, this framework ensures smooth automation for app validation while maintaining flexibility for future enhancements. 🚀