

| Sr. No. | Topics | Duration (Mins) | Session No (2 Hours) | Session No. (4 Hours) |
|---------|--|-----------------|----------------------|-----------------------|
| 1 | Introduction To Mobile Testing | 2 hrs | 1 | 1 |
| 2 | Introduction To Appium | 2 hrs | 2 | 1 |
| 3 | Automating the Application Management Actions | 4 hrs | 3 | 2 |
| | | | 4 | |
| 4 | Automating the Gestures and Key Events Handling using Appium | 4 hrs | 5 | 3 |
| | | | 6 | |
| 5 | Network Management And Performance Analysis using Appium | 4 hrs | 7 | 4 |
| | | | 8 | |
| 6 | Automating Hybrid and Native Apps | 4 hrs | 9 | 5 |
| | | | 10 | |

Topics

1. Introduction To Mobile Testing

Information

1. Define Mobile Automation Testing

- Mobile automation testing refers to the process of automating the testing of mobile applications across different devices, operating systems, and scenarios.
- It involves using specialized software tools or frameworks to execute test scripts and simulate user interactions on mobile devices.
- Mobile automation testing aims to ensure the quality, functionality, and performance of mobile applications by automating repetitive test scenarios.
- It helps streamline the testing process, increase test coverage, and improve efficiency compared to manual testing.

Key aspects of mobile automation testing include:

1. Test Script Creation:

- Test scripts are created using automation testing frameworks or tools.
- These scripts simulate user actions such as tapping, scrolling, inputting data, and interacting with various elements of the mobile application.

2. Cross-platform Testing:

- Mobile automation testing allows for testing applications across different platforms such as iOS and Android.
- It ensures consistent behavior and functionality across multiple devices and operating systems.

3. UI Testing:

- User Interface (UI) testing focuses on verifying the visual elements and interactions of the mobile application.
- It ensures that the app's interface is displayed correctly, buttons and menus function as intended, and navigation flows are smooth.

4. Functional Testing:

- Mobile automation testing verifies the functional aspects of the application, such as validating user inputs, checking data processing, and verifying the correctness of the output.
- It helps ensure that the application performs its intended tasks accurately.

5. Performance Testing:

- Mobile automation testing can include performance testing to evaluate the responsiveness, speed, and stability of the application under various load conditions.

- It helps identify performance bottlenecks, memory leaks, and other issues that could affect the app's performance.

6. Compatibility Testing:

- Mobile automation testing enables testing on a variety of devices, screen sizes, and resolutions.
- It ensures that the application is compatible and provides a consistent experience across different devices and configurations.

7. Integration Testing:

- Mobile automation testing allows for testing the integration of mobile applications with external systems or services.
- It ensures that the app can communicate and interact seamlessly with backend servers, databases, APIs, or other components.
- Benefits of mobile automation testing include faster execution of test cases, increased test coverage, improved accuracy, and the ability to perform repetitive tests consistently.
- It helps identify bugs, regressions, and compatibility issues early in the development cycle, leading to faster time to market and better user experiences.

2. Different Mobile Platforms

- There are several different mobile platforms available today, each with its own operating system and ecosystem.

1. Android:

- Developed by Google, Android is the most widely used mobile platform globally.
- It is based on the Linux kernel and offers an open-source operating system.
- Android devices are produced by various manufacturers and come in different shapes, sizes, and price ranges.
- Android provides access to a vast range of apps through the Google Play Store and supports customization and integration with Google services.

2. iOS:

- Developed by Apple, iOS is the operating system used exclusively on Apple devices, such as iPhone, iPad, and iPod Touch.
- iOS offers a seamless user experience, tight integration with other Apple products and services, and a curated App Store with strict quality control.
- It is known for its security, stability, and consistent performance across devices.

3. Windows:

- Microsoft's Windows operating system is also available for mobile devices.
- Windows Phone and Windows 10 Mobile were the previous mobile versions, but Microsoft has shifted its focus away from the mobile market.
- The availability of new Windows-based mobile devices and updates to the platform have become limited.

4. BlackBerry OS:

- Developed by BlackBerry Limited, BlackBerry OS was once a popular mobile platform known for its strong security features, business-focused capabilities, and physical keyboards.
- However, BlackBerry has transitioned to using Android as the primary operating system for its newer devices, making BlackBerry OS less prevalent in the current mobile landscape.

5. Tizen:

- Tizen is an open-source operating system primarily developed by Samsung, but it is also backed by other companies like Intel and the Linux Foundation.
- Tizen is used in a range of Samsung devices, including smartphones, smart TVs, and wearable devices.
- It offers a customizable user interface, compatibility with Android apps, and supports web-based application development.

6. KaiOS:

- KaiOS is a lightweight operating system designed for feature phones with limited hardware capabilities.
 - It focuses on providing essential smartphone-like features, internet connectivity, and support for popular apps like WhatsApp, Facebook, and Google Assistant.
 - KaiOS-powered devices target emerging markets and users who want a smartphone-like experience at an affordable price.
-
- Each platform has its own development tools, programming languages, and app distribution mechanisms.
 - Developers often need to consider the target platform and its specific requirements when creating mobile applications to ensure compatibility and optimal user experiences.

3. Types of mobile testing tours

- In mobile testing, a testing tour refers to a structured approach or technique for exploring and testing various aspects of a mobile application.
- There are several types of mobile testing tours that help ensure comprehensive testing coverage.

1. Functionality Tour:

- This tour focuses on testing the core functionality of the mobile application.
- It involves exploring different features, functionalities, and user interactions to ensure that the app behaves as expected.
- Testers examine each feature thoroughly, verifying inputs, outputs, and expected behaviour.

2. Usability Tour:

- Usability testing focuses on assessing the user-friendliness and ease of use of the mobile application.
- Testers evaluate the app's interface, navigation flow, responsiveness, and overall user experience.
- This tour aims to identify any usability issues or design flaws that may impact the app's usability.

3. Performance Tour:

- Performance testing focuses on evaluating the speed, responsiveness, and efficiency of the mobile application under various conditions.
- Testers examine factors such as load times, resource usage, network connectivity, and battery consumption to ensure that the app performs well and meets performance expectations.

4. Compatibility Tour:

- Compatibility testing ensures that the mobile application works correctly across different devices, operating systems, screen sizes, and resolutions.
- Testers verify that the app is compatible with a variety of mobile platforms, versions, and configurations to provide a consistent experience for all users.

5. Security Tour:

- Security testing aims to identify vulnerabilities, risks, and potential security breaches in the mobile application.
- Testers assess the app's security measures, authentication processes, data encryption, and protection against common security threats like unauthorized access or data leakage.

6. Localization Tour:

- Localization testing involves verifying that the mobile application works correctly in different languages, regions, and cultural contexts.
- Testers examine the app's language support, date and time formats, currency symbols, and any other localized elements to ensure accurate localization.

7. Interrupt Tour:

- The interrupt tour focuses on testing how the mobile application behaves when interrupted by various system events or external factors.

- Testers simulate interruptions like incoming calls, SMS, low battery, network disruptions, and app backgrounding to ensure that the app handles these events gracefully and maintains its state correctly.

8. Installation and Upgrade Tour:

- This tour focuses on testing the installation process and upgrading of the mobile application.
- Testers verify that the app installs correctly, updates smoothly, and migrates user data, settings, and preferences without any issues.
- These testing tours can be combined or customized based on the specific requirements of the mobile application and the testing objectives.
- They help ensure comprehensive testing coverage, improve the quality of the mobile application, and provide a better user experience.

4. Approaches to Mobile Testing

- There are several approaches to mobile testing, each with its own focus and objectives.
- The choice of approach depends on factors such as the nature of the mobile application, testing goals, available resources, and project requirements. Here are some common approaches to mobile testing:

1. Manual Testing:

- Manual testing involves human testers manually executing test cases and interacting with the mobile application.
- Testers simulate user actions, explore different functionalities, and verify the application's behaviour.
- Manual testing allows for flexibility, adaptability, and exploratory testing. It is suitable for early-stage testing, usability testing, ad-hoc testing, and scenarios that require human judgment.

2. Automated Testing:

- Automated testing involves using software tools or frameworks to automate the execution of test cases.
- Test scripts are created to simulate user interactions and verify the functionality of the mobile application.
- Automated testing provides repeatability, consistency, and faster test execution.
- It is suitable for regression testing, repetitive tasks, large-scale testing, and scenarios that require frequent testing cycles.

3. Emulator/Simulator Testing:

- Emulators and simulators are software-based tools that mimic the behaviour of mobile devices and operating systems on a computer.

- Emulator testing allows for testing the mobile application in a controlled and virtual environment.
- It enables testing on different device configurations, versions, and screen sizes without requiring physical devices.
- Emulator testing is cost-effective, scalable, and suitable for early-stage development, compatibility testing, and quick iterations.

4. Real Device Testing:

- Real device testing involves testing the mobile application on actual physical devices, such as smartphones and tablets.
- It allows for testing the application in real-world conditions, including various hardware specifications, network environments, and user interactions.
- Real device testing provides accurate results, real-time performance analysis, and device-specific testing.
- It is suitable for usability testing, performance testing, device-specific features, and final acceptance testing.

5. Crowd Testing:

- Crowd Testing involves leveraging a community of testers from diverse backgrounds and locations to test the mobile application.
- Testers from the crowd perform testing on different devices, platforms, and scenarios and provide feedback and bug reports.

- Crowd Testing offers a wide range of device coverage, real-world testing conditions, and access to a global pool of testers.
- It is suitable for exploratory testing, usability testing, localization testing, and gathering user feedback.

6. Beta Testing:

- Beta testing involves releasing the mobile application to a limited group of end-users or external testers to test it in real-world scenarios.
- Testers provide feedback, report bugs, and share their user experience.
- Beta testing helps identify issues, gather user feedback, and validate the application before its official release.
- It is suitable for user acceptance testing, compatibility testing, and collecting feedback on new features.
- It is important to consider a combination of these approaches based on the project requirements and goals.
- The choice of approach should align with the testing objectives, available resources, timeline, and budget to ensure comprehensive testing coverage and deliver a high-quality mobile application.

5. Levels of Mobile Testing

- Mobile testing can be conducted at various levels to ensure comprehensive coverage of the mobile application.

1. Unit Testing:

- Unit testing focuses on testing individual components or units of the mobile application in isolation.
- It involves testing functions, methods, or modules to ensure they behave as expected.
- Unit testing is typically performed by developers using frameworks like JUnit or XCTest.

2. Integration Testing:

- Integration testing verifies the interaction between different components, modules, or services within the mobile application.
- It ensures that the integrated parts work together correctly and communicate as intended.
- Integration testing may involve testing APIs, data exchange between modules, and dependencies on external systems.

3. Functional Testing:

- Functional testing ensures that the mobile application functions correctly and meets the specified requirements.

- Testers verify that each feature, functionality, and user interaction works as expected.
- It involves testing various scenarios, inputs, and outputs to validate the application's behaviour.

4. System Testing:

- System testing validates the mobile application as a whole in a simulated or real environment.
- It focuses on testing the interactions between the application, the operating system, and the hardware components.
- System testing covers different aspects, including performance, security, compatibility, and reliability of the entire system.

5. Usability Testing:

- Usability testing evaluates the user-friendliness, ease of use, and overall user experience of the mobile application.
- Testers assess the application's interface, navigation, responsiveness, and intuitiveness.
- Usability testing helps identify design flaws, user interface issues, and any usability barriers that may affect the user's satisfaction.

6. Performance Testing:

- Performance testing measures the responsiveness, stability, and efficiency of the mobile application under various conditions.
- It includes load testing, stress testing, and resource usage testing to ensure that the app performs well and meets performance expectations.
- Performance testing assesses factors such as response time, memory usage, network connectivity, and battery consumption.

7. Compatibility Testing:

- Compatibility testing ensures that the mobile application works correctly across different devices, operating systems, screen sizes, and resolutions.
- It involves testing on various combinations of hardware and software configurations to validate compatibility and consistent behavior across platforms.

8. Security Testing:

- Security testing focuses on identifying vulnerabilities, risks, and potential security breaches in the mobile application.
- Testers assess the app's security measures, authentication processes, data encryption, and protection against common security threats.
- Security testing helps ensure the app's robustness and protects user data and privacy.

9. Acceptance Testing:

- Acceptance testing involves validating the mobile application against predefined acceptance criteria or user expectations.
 - It is performed by stakeholders, clients, or end-users to ensure that the application meets their requirements and is ready for deployment.
 - Acceptance testing helps determine if the mobile application is suitable for release.
-
- These levels of mobile testing are not necessarily sequential and can overlap or be performed in parallel depending on the development methodology and project requirements.
 - A well-planned testing strategy considers a combination of these levels to ensure thorough testing coverage and deliver a high-quality mobile application.

6. Life cycle of mobile testing

- The life cycle of mobile testing typically follows a series of phases, each with specific activities and objectives.

1. Test Planning:

- In this initial phase, the testing team defines the scope, objectives, and testing requirements for the mobile application.

- Test planning involves identifying the target platforms, devices, testing approaches, and test tools.
- Test plans and strategies are developed, including the selection of testing techniques and methodologies.

2. Test Environment Setup:

- The testing team sets up the necessary test environment, which includes acquiring physical devices, emulators, simulators, and configuring the required software.
- The environment should replicate the target user's mobile devices, operating systems, network conditions, and other relevant factors to ensure accurate testing.

3. Test Case Design:

- Test cases are created based on the requirements and specifications of the mobile application.
- Test case design involves identifying test scenarios, inputs, expected outputs, and test data.
- Test cases should cover a range of functional and non-functional aspects, including various user interactions, edge cases, and error conditions.

4. Test Execution:

- Test execution is the phase where the actual testing takes place.

- Testers execute the test cases, either manually or using automated testing tools, to validate the functionality, performance, usability, compatibility, and security of the mobile application.
- Testers record the test results, including any defects or issues encountered during the execution.

5. Defect Management:

- Defects and issues discovered during the test execution phase are reported, tracked, and managed.
- The testing team assigns priorities and severities to each defect, and developers work on fixing them.
- Defects are retested to ensure successful resolution and closure.

6. Test Reporting:

- Throughout the testing life cycle, test reports are generated to provide an overview of the testing activities, progress, and results.
- Test reports include details on test coverage, test execution status, defect metrics, and overall quality assessment.
- These reports help stakeholders make informed decisions and track the progress of the testing effort.

7. Test Completion and Sign-Off:

- Once the test execution and defect resolution processes are complete, the testing team evaluates the

test results against the defined test objectives and acceptance criteria.

- If the application meets the desired quality standards, a sign-off is provided, indicating that the testing phase is complete and the application is ready for deployment.
- It's important to note that the mobile testing life cycle can vary based on project-specific requirements, methodologies, and organizational processes.
- Agile methodologies, for example, may involve iterative and continuous testing cycles throughout the development process.
- Additionally, continuous integration and continuous delivery (CI/CD) practices may integrate testing activities into the development pipeline, ensuring regular testing updates.

2. Introduction To Appium

Information

1. Introduction to Appium and Appium Architecture

A. Introduction to Appium:

- Appium is an open-source test automation framework used for mobile applications.
- It allows testers to automate mobile app testing across different platforms (such as Android and iOS) using a single codebase.
- Appium supports native, hybrid, and mobile web applications, making it a versatile tool for mobile test automation.
- It follows the WebDriver protocol, providing a familiar and standardized API for interacting with mobile devices and emulators.

B. Appium Architecture:

- The architecture of Appium consists of several key components that work together to facilitate mobile test automation.

1. Appium Server:

- The Appium Server is a critical component that acts as a bridge between the test scripts and the mobile devices/emulators.
- It receives commands from the test scripts, translates them into appropriate actions, and communicates with the mobile devices or emulators to execute those actions.
- The Appium Server can be started locally or accessed remotely, depending on the testing requirements.

2. WebDriver JSON Wire Protocol:

- Appium utilizes the WebDriver JSON Wire Protocol to enable communication between the Appium Server and the test scripts.
- This protocol defines a set of standardized HTTP RESTful API endpoints that allow test scripts to interact with the mobile devices or emulators.
- The test scripts send HTTP requests to the Appium Server, which then translates those requests into actions on the mobile device.

3. Desired Capabilities:

- Desired Capabilities are a set of key-value pairs that define the desired behaviour and configuration for the mobile session.
- They specify information such as the platform name (Android or iOS), device name, application path, automation name (UIAutomator, XCUITest), and other capabilities specific to the target platform.
- These capabilities are sent from the test script to the Appium Server during the session setup process.

4. Automation Driver:

- The Automation Driver is responsible for interacting with the native automation frameworks

provided by the respective platforms (such as UI Automator for Android and XCUI Test for iOS).

- It translates the WebDriver commands received from the Appium Server into actions that the native automation frameworks can understand and execute.
- The Automation Driver plays a crucial role in interacting with the UI elements, capturing responses, and providing feedback to the Appium Server.

5. Mobile Application:

- The mobile application under test is an essential component of the Appium architecture.
- The Appium Server interacts with the application by sending UI interaction commands and retrieving responses from the application.
- Appium supports testing native, hybrid, and mobile web applications, enabling testers to automate interactions with various elements, such as buttons, input fields, and gestures.
- Overall, the Appium architecture comprises the Appium Server, WebDriver JSON Wire Protocol, Desired Capabilities, Automation Driver, and the mobile application.
- These components work together to establish communication, execute commands, and automate

interactions between the test scripts and the mobile devices/emulators.

2. Appium Installation and Configuration

To install and configure Appium, follow these steps:

1. Prerequisites:

- Ensure that Node.js is installed on your machine.
- You can download Node.js from the official website (<https://nodejs.org>).
- Verify that the Java Development Kit (JDK) is installed. Appium requires JDK to run.
- You can download JDK from the Oracle website (<https://www.oracle.com/java/technologies/java-e-jdk11-downloads.html>).

2. Install Appium:

- Open a terminal or command prompt and run the following command to install Appium globally:

```
npm install -g appium
```

3. Install Appium Dependencies:

- Install the Appium server dependencies by running the following command:

```
npm install -g appium-doctor
```

- Run the appium-doctor command to check if all the dependencies are properly installed and configured.
- It will provide guidance on any missing dependencies or configuration issues.

4. Configure Appium:

- Create a new directory for your Appium project.
- Inside the project directory, create a new file named appium.config.js (or any other preferred name).
- Open the appium.config.js file and add the following basic configuration:

```
module.exports = {  
  port: 4723, // Appium server port  
  logLevel: 'info', // Log level (info, debug, error)  
  capabilities: [{  
    platformName: 'Android', // or 'iOS'  
  }],  
  desiredCapabilities: {}  
};
```

- Customize the configuration based on your specific requirements.

5. Start the Appium Server:

- Open a new terminal or command prompt window.
- Navigate to your project directory where the appium.config.js file is located.
- Run the following command to start the Appium server:

```
appium
```

- The Appium server will start running on the specified port, and the capabilities will be used to launch the specified device/emulator and the application.
- With these steps, you have installed and configured Appium on your machine.
- You can now write your test scripts using the preferred programming language (such as JavaScript, Java, or Python) and interact with the Appium server to automate testing on mobile devices or emulators.

3. Appium UI Automator viewer usage

- The Appium UI Automator Viewer is a tool that helps inspect and analyze the user interface (UI) elements of a mobile application.
- It provides a graphical representation of the application's UI hierarchy, allowing testers to identify UI elements and their attributes for automation purposes.
- Here's how you can use the Appium UI Automator Viewer:

1. Launch the Appium UI Automator Viewer:

- Open a command prompt or terminal.
- Navigate to the Android SDK's tools directory.

- The path may vary depending on the installation location.
- For example:

```
cd path/to/android-sdk/tools
```

- Run the following command to launch the UI Automator Viewer:

```
uiautomatorviewer
```

- The Appium UI Automator Viewer window will open.

2. Connect a Device or Emulator:

- Ensure that your Android device or emulator is connected to your computer via USB and is recognized by ADB (Android Debug Bridge).
- Start your application on the device or emulator that you want to inspect.

3. Capture the UI Hierarchy:

- In the Appium UI Automator Viewer window, click on the "Device Screenshot" button (camera icon) in the toolbar.
- The tool will capture a screenshot of the device's current screen.

4. Analyze the UI Hierarchy:

- The captured screenshot will be displayed in the UI Automator Viewer window.
- The left pane shows the UI hierarchy, representing the structure of the application's screens and UI elements.
- By expanding the tree nodes, you can navigate through the UI elements to explore their properties and attributes.

5. Inspect UI Elements:

- Click on any UI element in the hierarchy to view its details.
- The right pane will display the selected UI element's attributes, such as resource ID, class, text, content description, and other properties.
- Use these attributes to identify and interact with the UI elements in your Appium automation scripts.

6. Capture Screenshots and Export XML:

- To capture screenshots of specific UI elements, select the desired element and click on the "Device Screenshot" button again.
- To export the UI hierarchy as an XML file, click on the "Export" button (disk icon) in the toolbar.
- The Appium UI Automator Viewer is a valuable tool for understanding the structure and attributes of UI elements in a mobile application.
- It helps in identifying unique identifiers for automation, making it easier to write effective Appium test scripts.

4. Commonly used ADB commands

- ADB (Android Debug Bridge) is a versatile command-line tool that comes with the Android SDK.
- It allows you to communicate with and control Android devices or emulators connected to your computer.
- `adb devices`: Lists all the connected Android devices or emulators.
- `adb install <path_to_apk>`: Installs an APK file onto the connected device.
- `adb uninstall <package_name>`: Uninstalls an app from the connected device.

- `adb shell`: Opens a shell session on the connected device, allowing you to execute commands directly on the device.
- `adb logcat`: Displays the device's logcat logs, including debug messages, error messages, and other system logs. Useful for debugging and troubleshooting.
- `adb pull <remote_path> <local_path>`: Copies a file from the device to your local machine.
- `adb push <local_path> <remote_path>`: Copies a file from your local machine to the device.
- `adb shell am start -n <package_name>/<activity_name>`: Starts an activity of a specific app on the device.
- `adb shell am force-stop <package_name>`: Stops a running app on the device.
- `adb shell input keyevent <key_code>`: Sends a key event to the device, simulating a button press or key input.
- `adb shell screencap <file_path>`: Captures a screenshot of the device's screen and saves it to the specified file path on the device.
- `adb shell screenrecord <file_path>`: Records a video of the device's screen and saves it to the specified file path on the device.
- `adb shell dumpsys <service_name>`: Retrieves detailed information about a specific system service on the device. Useful for gathering system-related information.

- adb shell wm size: Retrieves the device's screen size and resolution.
- These are just a few examples of the commonly used ADB commands.
- ADB offers a wide range of commands and functionalities, allowing you to interact with and control Android devices or emulators for various testing and debugging purposes.

5. Starting the Appium server with advanced option

- When starting the Appium server, you can use advanced options to customize the server's behaviour and configuration.
1. Open a terminal or command prompt.
 - Use the Appium command to start the Appium server, followed by the desired advanced options.
 - Here's the basic syntax:

```
appium --<option1> <value1> --<option2> <value2> ...
```

2. Specify the advanced options based on your requirements.

- Here are some commonly used advanced options:
 - --address or -a: Specifies the IP address to which the Appium server should bind. By default, it binds to 0.0.0.0, which listens on all available network interfaces.
 - --port or -p: Specifies the port number on which the Appium server should run. By default, it runs on port 4723.
 - --log-level or -g: Sets the log level for the Appium server's output. Options include info, debug, warn, error, and silent.
 - --session-override: If enabled, allows a new session to override an existing session on the same server. By default, this option is disabled.
 - --relaxed-security: Disables Appium's security checks, allowing insecure operations. Use this option with caution and only in trusted environments.

- **--chromedriver-executable** or **-cp**: Specifies the path to the ChromeDriver executable if using the Chrome browser for testing on Android.
 - **--webdriveragent-port** or **-wda-port**: Specifies the port number on which the WebDriverAgent server should run if using iOS devices.
3. Add the desired advanced options to the Appium command.

- For example, to start the Appium server on a specific IP address and port, use the following command:

```
appium --address 127.0.0.1 --port 4725
```

4. Press Enter to execute the command.
- The Appium server will start with the specified advanced options.
 - You can now connect your Appium scripts to this server instance for test automation on the designated IP address and port.
 - Remember to refer to the Appium documentation for a comprehensive list of advanced options and their detailed explanations.

6. First Simple Code in Appium with Java

Here's an example of a simple Appium code written in Java to automate the launching of a mobile application and performing a basic interaction:

```
import io.appium.java_client.AppiumDriver;
import io.appium.java_client.MobileElement;
import io.appium.java_client.android.AndroidDriver;
import org.openqa.selenium.remote.DesiredCapabilities;
import java.net.MalformedURLException;
import java.net.URL;

public class AppiumExample {
    public static void main(String[] args) {
        // Set the desired capabilities
        DesiredCapabilities caps = new DesiredCapabilities();
        caps.setCapability("platformName", "Android");
        caps.setCapability("deviceName", "Your_Device_Name");
        caps.setCapability("appPackage", "com.example.app");
        caps.setCapability("appActivity",
"com.example.app.MainActivity");
    }
}
```

```
// Set the Appium server URL
URL appiumServerURL = null;
try {
    appiumServerURL = new
URL("http://127.0.0.1:4723/wd/hub");
} catch (MalformedURLException e) {
    e.printStackTrace();
}

// Create the Appium driver
AppiumDriver<MobileElement> driver = new
AndroidDriver<>(appiumServerURL, caps);

// Perform interaction with the app
MobileElement button =
driver.findElementById("com.example.app:id/button");
button.click();

// Quit the driver and close the app
driver.quit();

}
```

- In this example, we are using the Java client library for Appium to automate an Android application.
- Here's a breakdown of the code:
 - We import the necessary classes from the `io.appium.java_client` package for Appium automation and the `org.openqa.selenium.remote` package for `DesiredCapabilities`.
 - We create a `DesiredCapabilities` object and set the desired capabilities such as `platformName`, `deviceName`, `appPackage`, and `appActivity`. Modify these values to match your specific application.
 - We set the URL for the Appium server by providing the IP address and port number.
 - We create an instance of `AndroidDriver` by passing the Appium server URL and desired capabilities.
 - We find a button element using `driver.findElementById()` and perform a click action.
 - Finally, we call `driver.quit()` to close the driver and exit the application.
- Remember to have the Appium server running and the necessary setup, including connecting a device or starting an emulator, before executing the code.
- Also, ensure that you have the required dependencies and Appium Java client library included in your project.

7. Understanding Appium Inspector to find locators

- Appium Inspector is a tool that allows you to inspect and find locators for the UI elements of a mobile application.
- It provides a graphical interface to interactively explore the application's UI hierarchy and identify the properties and attributes of UI elements.
- Here's how you can use the Appium Inspector:

1. Launch the Appium Inspector:

- Open Appium Desktop or Appium GUI, depending on the version you are using.
- Start the Appium server if it's not already running.

2. Connect a Device or Emulator:

- Ensure that your mobile device or emulator is connected to your computer via USB and is recognized by the Appium server.

3. Configure Desired Capabilities:

- Specify the desired capabilities for your session in the Appium Inspector.
- These capabilities include the platform name (Android or iOS), device name, application path, automation name, and other relevant details.

4. Start a New Session:

- Click on the "Start Session" or "New Session" button in the Appium Inspector.
- This will initiate a connection between the Appium server and the device/emulator.

5. Inspect the UI Hierarchy:

- Once the session is started, the Appium Inspector will display the device's current screen.
- Use the mouse or touch interactions to navigate through the application's UI elements.
- You can expand and collapse tree nodes to explore the UI hierarchy.

6. Identify UI Element Properties:

- Click on a specific UI element in the hierarchy to select it.
- The Inspector will display the properties and attributes of the selected UI element.
- These may include resource ID, class, text, content description, coordinates, and other relevant details.

7. Generate Locators:

- Based on the selected UI element and its properties, the Appium Inspector can generate various types of locators, such as XPath, ID, class name, and accessibility identifier.
- These locators can be used in your Appium automation scripts to interact with the UI element.

8. Copy Locators:

- Once you have identified the desired locator, you can copy it from the Appium Inspector and use it in your test automation code.
- The Appium Inspector provides a convenient way to explore the UI hierarchy and find locators for UI elements.

- It helps you understand the structure and properties of the application's UI, enabling you to write more reliable and maintainable Appium scripts

9. Automate an app on a virtual Android device

- To automate an app on a virtual Android device using Appium, follow these steps:

1. Set up the Android Virtual Device (AVD):

- Install Android Studio on your machine if you haven't already.
- Launch Android Studio and go to the AVD Manager.
- Create a new virtual device by selecting a device definition, choosing a system image (e.g., Android version), and configuring other desired settings.
- Start the virtual device from the AVD Manager.

2. Set up the Desired Capabilities:

```
DesiredCapabilities caps = new DesiredCapabilities();
caps.setCapability("platformName", "Android");
caps.setCapability("deviceName",
"Your_Virtual_Device_Name");
caps.setCapability("appPackage", "com.example.app");
caps.setCapability("appActivity",
"com.example.app.MainActivity");
caps.setCapability("automationName", "UiAutomator2");
```

3. Start the Appium Server:

- Launch the Appium server either through Appium Desktop or via the command line.
- Make sure the Appium server is running and listening on the default address and port (127.0.0.1:4723) or a custom address/port.

4. Create and Initialize the Appium Driver:

- Instantiate the Android Driver class with the desired capabilities and the Appium server URL. For example:

```
URL appiumServerURL = new  
URL("http://127.0.0.1:4723/wd/hub");  
  
AppiumDriver<MobileElement> driver = new  
AndroidDriver<>(appiumServerURL, caps);
```

5. Automate the App:

- Use the driver instance to interact with the virtual device and automate the desired app functionality.
- For example:

```
// Find and interact with UI elements  
  
MobileElement usernameField =  
driver.findElementById("com.example.app:id/usernameField");  
usernameField.sendKeys("testuser");  
  
MobileElement passwordField =  
driver.findElementById("com.example.app:id/passwordField");  
passwordField.sendKeys("testpassword");  
  
MobileElement loginButton =  
driver.findElementById("com.example.app:id/loginButton");  
loginButton.click();
```

6. Perform Assertions and Verifications:

- Implement assertions or verifications to validate the expected behavior of the app.
- For example:

```
MobileElement welcomeMessage =  
driver.findElement("com.example.app:id/welcomeMessage");  
  
String welcomeText = welcomeMessage.getText();  
Assert.assertEquals("Welcome, testuser!", welcomeText);
```

7. Clean Up and Quit the Driver:

- At the end of the test execution, make sure to quit the driver to release the resources and close the connection to the virtual device.
- For example:

```
driver.quit();
```

- Remember to update the desired capabilities with the appropriate values for your virtual device and the app you want to automate.
- The locators (e.g., IDs) used in the code examples should correspond to the UI elements within the target app.

- Ensure that the Appium server, virtual device, and AVD configurations are properly set up before running the Appium automation code.

3. Automating the Application Management Actions

Information

1. Automate App management such as Installing, Un-Installing, Reset, Closing the App in Background

- To automate app management tasks such as installing, uninstalling, resetting, and closing the app in the background using Appium, you can utilize the following methods:

1. Installing an App:

- Use the `installApp()` method of the Appium driver to install an application on the device or emulator.
- Example:

```
driver.installApp("path/to/app.apk");
```

2. Uninstalling an App:

- Use the `removeApp()` method of the Appium driver to uninstall an application from the device or emulator.
- Example:

```
driver.removeApp("com.example.app");
```

3. Resetting an App:

- Use the `resetApp()` method of the Appium driver to reset the application to its initial state.
- Example:

```
driver.resetApp();
```

4. Closing the App in Background:

- Use the `closeApp()` method of the Appium driver to close the application.
- Example:

```
driver.closeApp();
```

- To perform these actions, make sure you have the Appium server running, the desired capabilities properly set, and the Appium Java client library included in your project.
- Here's an example of automating the app management tasks in a sequential manner:

```
import io.appium.java_client.AppiumDriver;
import io.appium.java_client.MobileElement;
import io.appium.java_client.android.AndroidDriver;
import org.openqa.selenium.remote.DesiredCapabilities;
import java.net.MalformedURLException;
import java.net.URL;
public class AppManagementExample {
    public static void main(String[] args) {
        // Set the desired capabilities
        DesiredCapabilities caps = new
DesiredCapabilities();
        caps.setCapability("platformName", "Android");
        caps.setCapability("deviceName",
"Your_Device_Name");
        caps.setCapability("appPackage",
"com.example.app");
        caps.setCapability("appActivity",
"com.example.app.MainActivity");
```

```
// Set the Appium server URL  
URL appiumServerURL = null;  
try {  
    appiumServerURL = new  
URL("http://127.0.0.1:4723/wd/hub");  
} catch (MalformedURLException e) {  
    e.printStackTrace();  
}  
  
// Create the Appium driver  
AppiumDriver<MobileElement> driver = new  
AndroidDriver<>(appiumServerURL, caps);  
  
// Install App  
driver.installApp("path/to/app.apk");  
  
// Uninstall App  
driver.removeApp("com.example.app");  
  
// Install App again  
driver.installApp("path/to/app.apk");  
  
// Reset App  
driver.resetApp();  
  
// Close App in Background  
driver.closeApp();  
  
// Quit the driver  
driver.quit();  
}
```

- Remember to modify the desired capabilities with the appropriate values for your device and the app you want to manage.
- Additionally, ensure you have the correct path to the APK file for installation.
- Execute the code after starting the Appium server and connecting a device or starting an emulator. The app management tasks will be performed in sequence as defined in the code.
- Make sure to handle any exceptions and add appropriate error handling based on your specific requirements and test scenarios.

2. Implementing Assertion in Appium Scripts

- Implementing assertions in Appium scripts allows you to validate the expected behaviour or state of the application during test execution.
- You can use assertions to compare actual values with expected values and determine whether the test has passed or failed.
- Here's how you can implement assertions in Appium scripts using Java:

1. Import the necessary assertion class:

```
import org.testng.Assert;
```

2. Perform the actions and capture the actual result:

```
MobileElement element =  
driver.findElement("elementId");  
String actualText = element.getText();
```

3. Define the expected result:

```
String expectedText = "Expected Text";
```

4. Compare the actual and expected results using assertions:

a. Use assertEquals() to compare two values and assert that they are equal:

```
Assert.assertEquals(actualText, expectedText);
```

b. Use assertTrue() to assert that a condition is true:

```
Assert.assertTrue(actualText.contains(expectedText));
```

- c. Use assertFalse() to assert that a condition is false:

```
Assert.assertFalse(actualText.isEmpty());
```

```
Assert.assertNotNull(actualText);
```

- e. Use assertNull() to assert that a value is null:

```
Assert.assertNull(actualText);
```

- f. Use assertSame() to assert that two objects refer to the same object:

```
Assert.assertSame(actualObject, expectedObject);
```

g

```
Assert.assertNotSame(actualObject,  
expectedObject);
```

- These are some commonly used assertion methods provided by testing frameworks like TestNG or JUnit.
- You can choose the appropriate assertion method based on the nature of the assertion you want to make.
- Remember to import the necessary assertion class and handle any exceptions that may arise during assertion failures.
- Assertions help validate the correctness of your automation script and provide a way to assert the expected behaviour of the application being tested.

1. Appium Test with TestNG

- When using Appium for mobile test automation, you can integrate it with the TestNG testing framework to organize and execute your test cases efficiently.
- TestNG provides various features such as test configuration, parallel test execution, test data management, and reporting capabilities.

Step 1: Set Up TestNG and Appium

- Ensure that you have TestNG installed in your project. You can add TestNG as a dependency in your project's build file (e.g., Maven or Gradle) or by manually including the TestNG JAR files.
- Set up Appium by starting the Appium server and initializing the driver with desired capabilities.

Step 2: Write TestNG Test Cases

- Create a new Java class and annotate it with the @Test annotation from the TestNG framework.
- Write individual test methods within the class, each representing a specific test case.
- Utilize Appium API methods to interact with the mobile application and perform the desired actions.
- Add assertions or verifications to validate the expected behaviour or outcomes of the test cases.
- Example:

```
import io.appium.java_client.MobileElement;
import io.appium.java_client.android.AndroidDriver;
import org.openqa.selenium.remote.DesiredCapabilities;
import org.testng.Assert;
import org.testng.annotations.*;

public class AppiumTestWithTestNG {
    private AndroidDriver<MobileElement> driver;
```

```
driver = new AndroidDriver<>(new
URL("http://127.0.0.1:4723/wd/hub"), caps);
}

@AfterClass
public void tearDown() {
    // Quit the driver and close the app
    driver.quit();
}

@Test
public void testLogin() {
    // Test case for login functionality
    // ...
    // Assertion
    Assert.assertEquals(actual, expected);
}
```

Step 3: Configure TestNG XML File (Optional)

- You can create a TestNG XML file to configure your test suite, test groups, parallel execution, and other settings.
- Define the test classes or packages to include in the XML file.
- Configure parallel execution, data providers, listeners, and other TestNG features based on your requirements.
- Example TestNG XML File:

```
<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd">

<suite name="Mobile Test Suite">
    <test name="Appium Tests">
        <classes>
            <class>
```

Step 4: Run Tests with TestNG

- Run the test class directly from your IDE by right-clicking on the test class and selecting "Run as TestNG Test."
- Alternatively, run the TestNG XML file to execute multiple test classes or packages with specific configurations.
- TestNG provides detailed test execution reports, including pass/fail status, test timings, and other statistics.
- Note: Make sure you have the necessary dependencies, including TestNG and the Appium Java client library, configured in your project.

- By leveraging TestNG with Appium, you can efficiently organize and execute your mobile test cases, utilize TestNG's extensive features for test management and reporting, and integrate Appium with other testing frameworks or tools seamlessly.

4. Automating the Gestures and Key Events Handling using Appium

Information

1. Automate gestures on Android App such as Touch Action, Scroll, Tapping, Long press, Swiping and Orientation
 - To automate gestures on an Android app using Appium, you can utilize the TouchAction class and various methods available in the MobileElement class.
1. Touch Action (Tap):

```
TouchAction touchAction = new  
TouchAction(driver);  
  
MobileElement element =  
driver.findElement(By.id("elementId"));  
  
touchAction.tap(TapOptions.tapOptions().withEleme
```

2. Touch Action (Long Press):

```
TouchAction touchAction = new TouchAction(driver);
MobileElement element =
driver.findElement(By.id("elementId"));
touchAction.longPress(LongPressOptions.longPressOptions().withElement(ElementOption.element(element))).perform();
```

```
TouchAction touchAction = new
TouchAction(driver);
MobileElement startElement =
driver.findElement(By.id("startElementId"));
MobileElement endElement =
driver.findElement(By.id("endElementId"));
touchAction.press(ElementOption.element(startElement)).moveTo(ElementOption.element(endElement))
.release().perform();
```

4. Scroll:

```
MobileElement element =
driver.findElement(MobileBy.AndroidUIAutomator("new UiScrollable(new
UiSelector().scrollable(true)).scrollIntoView(new
UiSelector().text(\"ScrollText\"))"));
```

5. Orientation (Landscape):

```
driver.rotate(ScreenOrientation.LANDSCAPE);
```

6. Orientation (Portrait):

```
driver.rotate(ScreenOrientation.PORTRAIT);
```

- In the above examples, make sure to replace "elementId" with the actual ID or locator of the desired element in the app.
- Adjust the scroll and swipe examples based on your specific requirements.
- Note that for some gestures, you may need to import classes such as TouchAction, TapOptions,

`ElementOption`, `LongPressOptions`, `MobileBy`, `ScreenOrientation`, etc.

- These examples cover common gestures such as tapping, long pressing, swiping, scrolling, and changing the orientation of the device.
- You can combine these gestures and customize them as per your app's functionality and test requirements.

2. Automating Mouse Hovering like Move To, Double Click, Button Down, Button Up etc

- When automating mouse hovering and interactions using Appium, you can utilize the `Actions` class to perform actions like move to, double click, button down, button up, etc.
- However, please note that Appium primarily focuses on mobile automation, and mouse interactions are not directly supported for mobile apps.
- The following example demonstrates how you can automate mouse hovering using Selenium WebDriver (which is often used alongside Appium for web automation):

```
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.interactions.Actions;
public class MouseHoverExample {
    public static void main(String[] args) {
```

```
// Locate the element to hover over
WebElement element =
driver.findElement(By.id("elementId"));

// Create an instance of the Actions class
Actions actions = new Actions(driver);

// Move to the element
actions.moveToElement(element).perform();

// Perform other mouse interactions as needed
actions.doubleClick().perform();
actions.clickAndHold().perform();
actions.release().perform();

// Close the browser
driver.quit();

}
```

- In the above example, the Actions class from Selenium WebDriver is used to perform mouse interactions.
 - The moveToElement() method moves the mouse cursor to the specified element, doubleClick() performs a double click, clickAndHold() holds down the mouse button, and release() releases the mouse button.
-
- Please note that the above code is for web automation, and not all mouse actions are applicable to mobile automation with Appium.
 - In the context of mobile automation, gestures like tapping, swiping, and scrolling are typically used instead of mouse interactions.
-
- If you are looking for mouse automation specifically, consider using other tools and libraries like Sikuli or AutoIt, which are better suited for desktop-based automation.

3. Automate Android Key Events

- To automate Android key events using Appium, you can utilize the `AndroidKey` enum provided by the Appium Java client library.
- This enum contains a list of Android key codes that can be sent as key events to the device.

1. Pressing a Key:

```
import  
io.appium.java_client.android.nativekey.AndroidKey;  
  
import  
io.appium.java_client.android.nativekey.KeyEvent;  
  
// Press the BACK key  
  
driver.pressKey(new KeyEvent(AndroidKey.BACK));
```

- In this example, the `pressKey()` method is used to send a key event to the device.
- The `KeyEvent` class is used to construct the key event, and `AndroidKey.BACK` represents the BACK key code.

2. Long Pressing a Key:

```
import  
io.appium.java_client.android.nativekey.AndroidKey;  
  
import  
io.appium.java_client.android.nativekey.KeyEvent;  
  
// Long press the HOME key  
  
KeyEvent event = new  
KeyEvent(AndroidKey.HOME).withFlag(KeyEventFlag
```

- In this example, the KeyEventFlag.LONG_PRESS flag is added to the key event to indicate a long press action.
- You can modify the key code and flags based on the desired key event.

3. Sending Special Key Codes:

```
import  
io.appium.java_client.android.nativekey.AndroidKey;  
  
import  
io.appium.java_client.android.nativekey.KeyEvent;  
  
// Send a special key code: F1  
  
driver.pressKey(new KeyEvent(AndroidKey.F1));
```

- In this example, the key event is sent for a special key code (F1).
- You can refer to the AndroidKey enum for a list of available key codes.

- Make sure to import the necessary classes and adjust the key codes and flags based on your specific requirements.
- These examples demonstrate how to automate Android key events using Appium.
- Key events can be useful for simulating user interactions involving physical or virtual keys on the Android device.

4. Automate Drag and Drop, Pinch and Zoom, Alert gestures

- To automate drag and drop, pinch and zoom, and alert gestures in mobile apps using Appium, you can utilize the TouchAction class and various methods available in the Appium API.

1. Drag and Drop:

```
import io.appium.java_client.TouchAction;  
import  
io.appium.java_client.touch.offset.ElementOption;  
import org.openqa.selenium.WebElement;  
  
WebElement sourceElement =  
driver.findElement(By.id("sourceElementId"));  
  
WebElement targetElement =  
driver.findElement(By.id("targetElementId"));  
  
TouchAction touchAction = new TouchAction(driver);  
touchAction.longPress(ElementOption.element(source
```

- In this example, you locate the source and target elements for the drag and drop operation.
- Then, using the TouchAction class, you perform a long press on the source element, move it to the target element, and release the touch to complete the drag and drop action.

2. Pinch and Zoom:

```
import io.appium.java_client.MultiTouchAction;  
import io.appium.java_client.TouchAction;  
import io.appium.java_client.touch.WaitOptions;  
import  
io.appium.java_client.touch.offset.ElementOption;  
import org.openqa.selenium.WebElement;  
  
WebElement element =  
driver.findElement(By.id("elementId"));  
  
int centerX = element.getLocation().getX() +  
(element.getSize().getWidth() / 2);  
  
int centerY = element.getLocation().getY() +  
(element.getSize().getHeight() / 2);  
  
TouchAction<?> action1 = new  
TouchAction<>(driver);
```

```
.moveTo(ElementOption.point(centerX - 100,  
centerY - 100))  
.release();  
  
TouchAction<?> action2 = new TouchAction<>(driver);  
action2.press(ElementOption.element(element))  
.waitAction(WaitOptions.waitOptions(Duration.ofMilli  
s(1000)))  
.moveTo(ElementOption.point(centerX + 100,  
centerY + 100))  
.release();  
  
MultiTouchAction multiTouch = new  
MultiTouchAction(driver);  
multiTouch.add(action1).add(action2).perform();
```

- In this example, you locate the element to perform the pinch and zoom gesture.

- You calculate the center coordinates of the element, and then using TouchAction, you create two separate actions to move the element towards or away from the center.
- Finally, you use MultiTouchAction to combine and perform both actions simultaneously, simulating a pinch and zoom gesture.

3. Alert Gestures:

```
import io.appium.java_client.Alert;
Alert alert = driver.switchTo().alert();
alert.accept(); // Accept the alert
// OR
alert.dismiss(); // Dismiss the alert
```

- In this example, you switch to the alert context using driver.switchTo().alert().
- Then, you can either accept the alert by calling alert.accept() or dismiss it by calling alert.dismiss().
- Make sure to replace "sourceElementId," "targetElementId," and "elementId" with the actual

IDs or locators of the corresponding elements in your app.

- These examples cover automating drag and drop, pinch and zoom, and alert gestures in mobile apps using Appium.
- You can customize these examples based on your app's specific UI elements and automation requirements.

5.Network Management and Performance Analysis using Appium

Information

1. Read the Notifications from the Notification Bar
 - To read notifications from the notification bar in Android using Appium, you can use a combination of Appium's capabilities and the Android-specific methods.
1. Set the Required Capabilities:
 - When configuring your Appium session, make sure to set the necessary capabilities to interact with the notification bar.
 - For example:

```
DesiredCapabilities capabilities = new  
DesiredCapabilities();  
capabilities.setCapability("platformName", "Android");  
capabilities.setCapability("deviceName",  
"your_device_name");  
// Add other desired capabilities
```

2. Open the Notification Shade:

- Use the Appium `openNotifications()` method to open the notification shade:

```
driver.openNotifications();
```

3. Access the Notification Elements:

- Once the notification shade is open, you can locate and interact with the notification elements as you would with any other elements on the screen.
- Use the appropriate locator strategy, such as `By.id`, `By.xpath`, or `By.className`, to locate the desired elements within the notification shade.

4. Read the Notification Content:

- Retrieve the notification content by accessing the text or attributes of the notification elements.
- For example, you can use the `getText()` method to retrieve the text content of the notification.

- Here's an example that demonstrates opening the notification shade and reading the content of the first notification:

```
// Open the notification shade  
driver.openNotifications();  
  
// Locate the notification element  
WebElement notification =  
driver.findElement(By.xpath("//android.widget.LinearLayout[1]"));  
  
// Read the notification content  
String notificationContent = notification.getText();  
System.out.println("Notification Content: " +  
notificationContent);
```

- Note that the XPath expression used in the example (`//android.widget.LinearLayout[1]`) targets the first notification in the notification shade.

- You may need to adjust the XPath or use a different locator strategy based on the structure and properties of the notification elements in your app.

- By following these steps, you can read notifications from the notification bar using Appium.
- During an interview, you can further discuss handling multiple notifications, verifying specific notification details, and adapting the approach to handle different notification scenarios.

2. Automate Networking Configurations such as to Toggle Airplane Mode, Toggle Data, Toggle Wi-Fi

- To automate networking configurations such as toggling Airplane Mode, Data, and Wi-Fi in Android using Appium, you can utilize the Appium driver's capabilities along with the Android-specific methods.
- Here's an example of how you can achieve this:

1. Set the Required Capabilities:

- When configuring your Appium session, make sure to set the necessary capabilities to interact with the device's networking configurations.
- For example:

```
DesiredCapabilities capabilities = new  
DesiredCapabilities();  
  
capabilities.setCapability("platformName", "Android");  
capabilities.setCapability("deviceName",  
"your_device_name");  
  
// Add other desired capabilities
```

2. Toggle Airplane Mode:

- Use the Appium driver's `toggleAirplaneMode()` method to enable or disable Airplane Mode:

```
driver.toggleAirplaneMode();
```

3. Toggle Data:

- Use the Appium driver's `toggleData()` method to enable or disable mobile data:

```
driver.toggleData();
```

4. Toggle Wi-Fi:

- Use the Appium driver's `toggleWifi()` method to enable or disable Wi-Fi:

```
driver.toggleWifi();
```

- Here's an example that demonstrates toggling Airplane Mode, Data, and Wi-Fi:

```
// Toggle Airplane Mode  
driver.toggleAirplaneMode();  
  
// Toggle Data  
driver.toggleData();  
  
// Toggle Wi-Fi  
driver.toggleWifi();
```

- Note that these methods rely on the Appium driver's capabilities to interact with the device's networking configurations.
- Make sure to refer to the Appium documentation and the specific version of the Appium Java client you are using for accurate method usage.

- By using these methods, you can automate networking configurations such as toggling Airplane Mode, Data, and Wi-Fi in Android using Appium.
- During an interview, you can discuss additional network-related automation scenarios, handling network-related assertions, and verifying the changes in networking configurations after toggling.

3. Automating the App Interactions such as Lock and Unlock and rotate the Android Phone

- Automating app interactions such as locking and unlocking the device and rotating the Android phone can be achieved using Appium's capabilities and the Android-specific methods.

1. Lock and Unlock the Device:

- To automate locking and unlocking the device, you can use the Appium driver's lockDevice() and unlockDevice() methods.
- Here's how you can do it:

```
// Lock the device  
driver.lockDevice();
```

```
// Unlock the device  
driver.unlockDevice();
```

- The lockDevice() method locks the device screen, simulating the user action of pressing the power button.
- The unlockDevice() method unlocks the device screen, simulating the user action of swiping or entering the device's unlock pattern or PIN.

2. Rotate the Android Phone:

- To automate rotating the Android phone to different orientations, you can use the Appium driver's 'rotate' method.
- Here's an example:

```
import io.appium.java_client.ScreenOrientation;

// Rotate the device to landscape mode
driver.rotate(ScreenOrientation.LANDSCAPE);

// Rotate the device to portrait mode
driver.rotate(ScreenOrientation.PORTRAIT);
```

- The rotate method allows you to specify the desired orientation using the 'ScreenOrientation' enum, which includes values such as LANDSCAPE and PORTRAIT.
- Note that the availability of these capabilities may vary depending on the version of Appium, the Appium client library, and the device or emulator being used.
- During an interview, you can discuss other app interactions, such as taking screenshots, launching other apps, or interacting with device settings.
- Additionally, you can highlight the importance of these interactions in testing different scenarios, handling device-specific behaviour, and handling any potential challenges that may arise during automation.

4. Get the Performance Data of an Android App

- To retrieve performance data of an Android app using Appium, you can leverage the Appium driver's capabilities along with the Android-specific methods.

1. Set the Required Capabilities:

- When configuring your Appium session, make sure to set the necessary capabilities to enable performance data collection.
- For example:

```
DesiredCapabilities capabilities = new  
DesiredCapabilities();  
capabilities.setCapability("platformName", "Android");  
capabilities.setCapability("deviceName",  
"your_device_name");  
capabilities.setCapability("automationName",  
"uiautomator2");  
capabilities.setCapability("adbExecTimeout", "50000");  
capabilities.setCapability("androidInstallTimeout",  
"20000");  
capabilities.setCapability("performanceLoggingPrefs",  
"{"enablePerformanceLogging": true}");  
capabilities.setCapability("adbPort", "5037");  
// Add other desired capabilities
```

In the above capabilities, the performanceLoggingPrefs capability is set to enable performance logging.

2. Start Performance Monitoring:

- After launching the app, you need to start the performance monitoring by using the Appium driver's 'startPerformanceRecord' method.
- Here's an example:

```
driver.startPerformanceRecord();
```

navigating through different screens, executing specific actions, or conducting tests.

- During these interactions, the performance data will be captured.

4. Stop Performance Monitoring and Retrieve Data:

- After the interactions are completed, you can stop the performance monitoring and retrieve the collected data.
- Use the Appium driver's stopPerformanceRecord method to stop monitoring and retrieve the performance data.
- Here's an example:

```
List<LogEntry> performanceData =  
driver.stopPerformanceRecord(PerfDataType.ALL);
```

- The ‘stopPerformanceRecord’ method returns a list of ‘LogEntry’ objects that represent the collected performance data.
 - You can specify the desired performance data types (e.g., PerfDataType.CPU, PerfDataType.MEMORY) to retrieve specific metrics.
 - Iterate through the ‘LogEntry’ objects to extract and analyze the performance data as per your requirements.
-
- Please note that the availability and format of performance data may vary depending on the device, Android version, and the app itself.
 - During an interview, you can discuss further performance data analysis techniques, integrating performance testing into your automation framework, and leveraging tools like Appium Studio or external performance monitoring tools to capture and analyze performance data in a more comprehensive manner.

6. Automating Hybrid and Native Apps

Information

1. Hybrid and Native App Features

- Hybrid and native apps are two different approaches to mobile app development, each with its own set of features and characteristics.

A. Hybrid App Features:

1. Cross-Platform Compatibility:

- Hybrid apps are built using web technologies such as HTML, CSS, and JavaScript.
- They can run on multiple platforms, including iOS and Android, without requiring significant modifications.

2. Code Reusability:

- Hybrid apps allow for code reuse across different platforms, as they are developed using web technologies.
- Developers can write the code once and deploy it on multiple platforms, saving time and effort.

3. Web Technology Stack:

- Hybrid apps utilize web technologies, enabling developers to leverage existing web development skills and tools.
- They can use frameworks like Apache Cordova (PhoneGap) or Ionic to access device capabilities through JavaScript APIs.

4. Faster Development Cycle:

- Hybrid apps often have shorter development cycles compared to native apps.
- This is because developers can write and test code using web development tools and frameworks, and then package it as a native app for different platforms.

5. Offline Functionality:

- Hybrid apps can leverage local storage capabilities and caching mechanisms to provide offline functionality.
- Users can access and interact with certain app features even without an active internet connection.

B. Native App Features:

1. Native User Experience:

- Native apps are built specifically for a particular platform using the platform's native programming language (e.g., Java or Kotlin for Android, Objective-C, or Swift for iOS).
- This allows them to provide a seamless and optimized user experience, conforming to the platform's UI and interaction patterns.

2. Access to Native APIs and Features:

- Native apps have direct access to the device's hardware and software features, including camera, GPS, accelerometer, and push notifications.
- This allows developers to create highly integrated and feature-rich apps that can leverage the full capabilities of the device.

3. Performance and Speed:

- Native apps are known for their superior performance and speed, as they are optimized for the specific platform.
- They can take full advantage of the device's hardware and utilize low-level APIs, resulting in smooth and responsive user experiences.

4. App Store Integration:

- Native apps can be distributed through official app stores (e.g., Google Play Store for Android, App Store for iOS), providing a streamlined distribution process and access to a wider user base.

5. Enhanced Security:

- Native apps can utilize platform-specific security features and frameworks, providing enhanced data protection and secure user interactions.
- It's important to note that hybrid and native apps have their own advantages and considerations.
- The choice between them depends on factors such as development timeline, target audience, required features, and performance requirements.

2. Desired Capabilities for Native and Hybrid Apps

- The desired capabilities for native and hybrid apps in Appium may have some similarities, but there are also some specific capabilities for each type of app.

A. Desired Capabilities for Native Apps:

1. `platformName`: Specifies the mobile platform for the app (e.g., "Android" or "iOS").

2. `deviceName`: Specifies the name of the target device or emulator on which the app will be executed.
3. `app`: Specifies the path or URL of the app's installation file (APK for Android or IPA for iOS).
4. `automationName`: Specifies the automation technology or framework to be used (e.g., "UiAutomator2" for Android, "XCUITest" for iOS).
5. `udid`: Specifies the unique device identifier (UDID) of the target device when multiple devices are connected.
6. `platformVersion`: Specifies the version of the mobile platform (e.g., "10.0" for Android, "14.5" for iOS).
7. `appPackage`: Specifies the package name of the app for Android.
8. `appActivity`: Specifies the main activity name of the app for Android.
9. `bundleId`: Specifies the bundle identifier of the app for iOS.

B. Desired Capabilities for Hybrid Apps:

1. `platformName`: Same as for native apps, specifies the mobile platform for the app.
2. `deviceName`: Same as for native apps, specifies the name of the target device or emulator.
3. `app`: Same as for native apps, specifies the path or URL of the app's installation file.
4. `automationName`: Same as for native apps, specifies the automation technology or framework to be used.

5. `browserName`: Specifies the browser or webview to be used for running the hybrid app (e.g., "Chrome" for Android, "Safari" for iOS).
 6. `chromedriverExecutable`: Specifies the path to the ChromeDriver executable file for running the hybrid app on Android.
 7. `safariInitialUrl`: Specifies the initial URL to load when running the hybrid app on iOS with Safari.
- These are just some of the commonly used desired capabilities for native and hybrid apps in Appium.
 - The specific capabilities and their values may vary depending on the app, platform, and testing requirements.
 - It's important to consult the Appium documentation and the specific platform's capabilities documentation for accurate and up-to-date information on desired capabilities.

3. Activity Lifecycle Testing for Native App

- Activity lifecycle testing is crucial for ensuring the proper behavior and stability of native Android apps.
- It involves testing the app's behavior during different stages of its lifecycle, such as when the app is launched, paused, resumed, or terminated.

1. Launching the App:

- Test the app's behavior when launching it for the first time.

- Verify that the main activity is displayed correctly.
- Validate any initialization processes or data loading.

2. Pausing and Resuming the App:

- Test the app's behavior when it is paused and then resumed.
- Simulate scenarios where the app is put in the background, such as by pressing the Home button.
- Verify that the app properly preserves its state and data when resumed.

3. Handling Configuration Changes:

- Test the app's behavior when configuration changes occur, such as device rotation or language change.
- Verify that the app handles configuration changes gracefully without losing data or causing crashes.
- Validate that the app correctly adapts its UI and resources to the new configuration.

4. Navigating Between Activities:

- Test the app's behavior when navigating between different activities within the app.
- Verify that the app maintains proper state and data when switching activities.
- Validate any data transfer or communication between activities.

5. Back Button and Activity Stack:

- Test the app's behavior when the back button is pressed.
- Verify that the app navigates back to the correct previous activity or exits the app when necessary.
- Validate the behavior of the activity stack, ensuring that the app maintains the correct order of activities.

6. Termination and Restarting:

- Test the app's behavior when it is terminated, either manually or by the system.
- Verify that the app can be properly restarted and resumes from the correct state.
- Validate any data persistence or restoration processes.
- During activity lifecycle testing, it's important to simulate different scenarios and edge cases to ensure the app behaves as expected under various conditions.
- This can be achieved through manual testing or by automating the test scenarios using Appium or other testing frameworks.
- Additionally, it's essential to consider device-specific behavior, handle any exceptions or errors gracefully, and ensure compatibility with different Android versions and device configurations.

1. Steps to Automate a Native Android App using Java

- To automate a native Android app using Java and Appium, you can follow these steps:

1. Set Up the Environment:

- Install Java Development Kit (JDK) on your machine.
- Set up the Java environment variables.
- Install an Integrated Development Environment (IDE) like Eclipse or IntelliJ.

2. Set Up Appium:

- Install Node.js on your machine.
- Install Appium using npm (Node Package Manager) by running the command: ‘npm install -g appium’.
- Install Appium server dependencies using the command: ‘appium-doctor –android’.

3. Set Up Android SDK and Emulator:

- Install Android Studio and set up the Android SDK.
- Create an Android Virtual Device (AVD) emulator using the AVD Manager in Android Studio.

4. Create a New Java Project:

- Open your IDE and create a new Java project.

- Configure the project to use the necessary dependencies, such as the Appium Java client library and the Selenium WebDriver.

5. Write Test Scripts:

- Create a new test class or package within the Java project.
- Import the required libraries, including the Appium Java client library.
- Write test scripts using Java and the Appium Java client APIs.
- Use the desired capabilities to specify the platform, device, app path, and other configurations.

6. Set Up Appium Driver:

- Initialize the Appium driver in your test script using the desired capabilities.
- Connect to the Appium server using the server URL and desired capabilities.

7. Interact with the Native App:

- Use the Appium driver's methods to locate and interact with the native app's elements, such as clicking buttons, entering text, or verifying text content.
- Utilize Appium's built-in UIAutomatorViewer or Appium Inspector to inspect elements and generate locators for the app's UI.

8. Run and Debug the Tests:

- Set up the desired test configuration in your IDE.
- Run the test scripts to execute the automated tests on the native Android app.
- Analyze test results and debug any issues that may arise.

9. Enhance Test Automation:

- Implement test frameworks, such as TestNG or JUnit, to manage test cases and generate test reports.
- Utilize design patterns and best practices for test automation, such as Page Object Model (POM) or Page Factory, to enhance maintainability and reusability.

10. Continuously Refine and Maintain Tests:

- Continuously update and maintain your test scripts as the native Android app evolves.
- Incorporate error handling, assertions, and verification points to ensure the accuracy and reliability of the tests.
- Regularly review and refactor your test scripts to improve efficiency and readability.
- Remember to adapt these steps to your specific project requirements and adjust them based on the native Android app's characteristics and testing needs.

Assignments :

1. Toggle Wi-Fi and Capture Network Latency

- Objective: Automate the toggling of Wi-Fi and capture network latency using Appium.
- Tasks:
- Automate the toggling of Wi-Fi using the Appium driver's toggleWifi() method.
- After toggling Wi-Fi, perform actions that require network connectivity and capture the network latency for each action.
- Use appropriate techniques or tools to measure network latency, such as performance monitoring libraries or network profiling tools.
- Verify that the network latency is within acceptable limits using assertions or verifications.

2. Automating Drag and Drop Gesture

- Objective: Automate the drag and drop gesture on a mobile app using Appium.
- Tasks:
- Write an Appium automation script to locate the source and target elements on the screen for the drag and drop action.
- Use the drag and drop gesture to perform the action by dragging the source element and dropping it onto the target element.
- Validate the outcome of the drag and drop action using assertions or verifications.

3. Test Plan for a Mobile Application:

- Develop a comprehensive test plan for testing a mobile application of your choice.
- Include the following sections:
 - i. Introduction and scope of testing
 - ii. Testing objectives and goals
 - iii. Test environment setup (devices, emulators, tools)
 - iv. Test strategy and approaches (functional, performance, usability, etc.)
 - v. Test coverage and prioritization
 - vi. Test data management
 - vii. Test execution and reporting
 - viii. Defect management
 - ix. Risks and mitigation strategies

4. Cross-platform App Automation

- Task: Write an automated test script to perform the following actions on a cross-platform app (e.g., developed using Flutter or React Native) using Appium and Java:
 - Launch the cross-platform app on both Android and iOS devices/emulators.
 - Perform navigation through different screens by interacting with buttons or menus.
 - Validate the correctness of displayed data or elements.
 - Execute specific functionality unique to each platform, such as taking a picture using the device's camera.
 - Test any platform-specific features or behaviors, such as push notifications or geolocation.

- Verify that the app maintains consistent behavior and UI across both Android and iOS.
- Generate a detailed test report capturing the test execution results for both platforms.

5. Synchronizing Tests with Fluent Wait

- Objective: Use Fluent Wait to implement synchronization in Appium and handle dynamic elements.
- Tasks:
- Write an Appium automation script with a Fluent Wait to wait for a specific condition to be met before proceeding further.
- Use Fluent Wait to handle dynamic elements that may take varying amounts of time to load.
- Include assertions or verifications to validate the condition or presence of the element after the Fluent Wait.
- Test the effectiveness of Fluent Wait by incorporating scenarios with different waiting times for elements.

Interview Questions :

1. What is the difference between Implicit, Explicit, and Fluent waits in Appium test automation?
 - Implicit, Explicit, and Fluent waits are different methods used to synchronize tests in Appium.

i. Implicit Wait:

- Implicit wait is a global wait applied to all elements in the Appium session.
- It sets a maximum time limit for the driver to wait for an element to appear before throwing an exception.
- Implicit waits are defined once and affect all subsequent element interactions.
- They are useful when waiting for elements that may take some time to load or appear.

ii. Explicit Wait:

- Explicit wait allows you to wait for a specific condition to be met before proceeding further in the test.
- It provides more fine-grained control over synchronization by allowing you to define a condition and a maximum wait time.
- The driver waits until the condition is satisfied or the timeout is reached.
- Explicit waits are helpful when you need to wait for specific elements or conditions during test execution.

iii. Fluent Wait:

- Fluent wait is a dynamic wait that provides additional flexibility compared to implicit and explicit waits.

- It allows you to define both the polling interval and the maximum wait time.
- Fluent wait polls the DOM at regular intervals until the defined condition is met or the timeout is reached.
- It is useful when you need to customize the wait behaviour, such as setting a specific interval or ignoring specific exceptions.

2. How do you automate gestures such as touch action, scroll, tapping, long press, swiping, and orientation changes in Appium?

- To automate gestures in Appium, you can use various Appium API methods and classes.

i. Touch Action:

- You can use the TouchAction class to perform touch-based gestures.
- For example, you can create a TouchAction instance, specify the action (e.g., press, moveTo, release), and then call the perform method to execute the action on the desired element or screen coordinates.

ii. Scroll:

- Appium provides multiple ways to automate scrolling.
- One common approach is using the mobileBy.AndroidUIAutomator method to construct a scrollable UI selector and then using driver.findElement to locate the element for scrolling.

- You can then perform a scroll action using TouchAction or other scroll-specific methods.

iii. Tapping:

- Tapping can be automated by locating the desired element using Appium's element locating methods (e.g., By.id, By.xpath) and then performing a tap action using the click method on the found element.

iv. Long Press:

- Similar to tapping, long press can be automated by locating the element and then performing a long press action using the longPress method from the TouchAction class.

v. Swiping:

- Swiping gestures can be achieved by using TouchAction to perform a sequence of actions such as press, moveTo, and release.
- You can specify the starting and ending coordinates or elements to perform the swipe action.

vi. Orientation:

- To automate orientation changes, you can use the rotate method from the WebDriver class. By passing the desired ScreenOrientation (e.g., LANDSCAPE,

PORTRAIT), you can change the orientation of the device or emulator.

3. What are the challenges you may face in mobile automation testing, and how do you overcome them?

- In mobile automation testing, some common challenges include device fragmentation, frequent OS updates, and handling different screen sizes/resolutions.
- To overcome these challenges, I would:
- Use device farms or cloud-based testing platforms to access a wide range of devices for testing.
- Regularly update the automation frameworks and tools to ensure compatibility with the latest OS versions.
- Implement responsive design testing to verify the application's behaviour on various screen sizes.
- Prioritize testing on the most popular devices and OS versions based on market share and user analytics.

4. What are the main advantages of hybrid apps over native apps?

Hybrid apps offer several advantages over native apps, including:

- a. Cross-platform compatibility:
 - Hybrid apps can run on multiple platforms, such as iOS and Android, using a single codebase.

- This eliminates the need for separate development efforts for each platform.
- b. Code reusability:
- With hybrid apps, developers can write the code once using web technologies like HTML, CSS, and JavaScript, and deploy it across multiple platforms, saving time and effort.
- c. Faster development cycle:
- Hybrid app development typically has a faster development cycle as compared to native apps.
 - Since developers can reuse code and leverage existing web development skills, it results in quicker app development and deployment.
- d. Lower maintenance cost:
- Hybrid apps require less maintenance effort since changes or updates can be made to the codebase, and the changes are automatically reflected across all platforms.
- e. Simplified distribution:
- Hybrid apps can be distributed easily through app stores or even as web apps, allowing for wider accessibility and reach.

5. How can you retrieve performance data of an Android app using Appium?

To retrieve performance data of an Android app using Appium, you can follow these steps:

- i. Set the Required Capabilities:

- When configuring your Appium session, make sure to set the necessary capabilities to enable performance data collection. For example, you can use the capability `performanceLoggingPrefs` with the value `{"enablePerformanceLogging": true}`.

ii. Start Performance Monitoring:

- After launching the app, start the performance monitoring using the Appium driver's `startPerformanceRecord` method.
- This method enables the collection of performance data.

iii. Perform Interactions:

- Execute various interactions in the app that you want to analyze performance for, such as scrolling, tapping, or data retrieval.

iv. Stop Performance Monitoring and Retrieve Data:

- After the interactions are completed, stop the performance monitoring and retrieve the collected data.
- Use the Appium driver's `stopPerformanceRecord` method, passing the desired performance data type(s) (e.g., `PerfDataType.CPU`, `PerfDataType.MEMORY`) as an argument.

v. Analyze Performance Data:

- Once the performance data is retrieved, you can analyze it based on the collected metrics, such as CPU usage, memory consumption, or network traffic.
- Utilize appropriate techniques or tools to interpret and draw insights from the performance data.
- Here's an example:

```
// Start performance monitoring
driver.startPerformanceRecord();

// Perform interactions in the app

// Stop performance monitoring and retrieve data
List<LogEntry> performanceData =
driver.stopPerformanceRecord(PerfDataType.ALL);

// Analyze the performance data
// Perform calculations or extract metrics of interest
```

6. How can the Appium UI Automator Viewer help in finding locators for UI elements in a mobile application?

- The Appium UI Automator Viewer provides a graphical representation of the UI hierarchy of a

mobile application, making it easier to find and inspect UI elements.

- It offers the following benefits for finding locators:
 - i. Visual representation:
 - The UI Automator Viewer displays the application's UI hierarchy in a tree-like structure, allowing testers to visualize and navigate through the elements.
 - ii. Element properties:
 - Testers can select a specific UI element to view its properties and attributes, such as resource ID, class, text, and content description.
 - These properties can be used to identify and locate the UI element in automation scripts.
 - iii. Locator generation:
 - The UI Automator Viewer can generate various types of locators, such as XPath, ID, class name, and accessibility ID, based on the selected UI element.
 - These locators can be copied and used in Appium automation scripts to interact with the element.

7. What are the key advantages of native apps over hybrid apps?

- Native apps offer several advantages over hybrid apps, including:
 - i. Native user experience:
 - Native apps provide a seamless and optimized user experience as they are specifically designed for a particular platform.

- They conform to the platform's UI and interaction patterns, resulting in better performance and user satisfaction.
- ii. Access to device capabilities:
- Native apps have direct access to the device's hardware and software features, such as camera, GPS, accelerometer, and push notifications.
 - This allows developers to create highly integrated and feature-rich apps that can leverage the full capabilities of the device.
- iii. Performance and speed:
- Native apps are known for their superior performance and speed.
 - They are optimized for the specific platform, utilizing low-level APIs and hardware acceleration, which results in smoother animations, faster response times, and overall better performance.
- iv. App store integration:
- Native apps can be distributed through official app stores like the Google Play Store for Android or the App Store for iOS.
 - This provides a streamlined distribution process, enhanced discoverability, and access to a larger user base.
- v. Enhanced security:
- Native apps can utilize platform-specific security features and frameworks, providing enhanced data protection and secure user interactions.

- They can leverage the device's security measures like biometric authentication, sandboxing, and encryption to ensure the privacy and integrity of user data.

8. How can you automate mouse hovering and interactions like move to, double click, button down, and button up using Appium or Selenium WebDriver?

- Mouse hovering and interactions can be automated using the Actions class in Selenium WebDriver or Appium (for web automation).
 - i. Move To:
 - You can use the moveToElement method of the Actions class to move the mouse cursor to a specific element.
 - First, create an instance of the Actions class, chain the moveToElement method, and provide the desired element as the parameter.
 - Finally, call the perform method to execute the action.
 - ii. Double Click:
 - To automate double-clicking, use the doubleClick method of the Actions class.
 - Similar to moving the cursor, create an instance of the Actions class, chain the doubleClick method, and provide the element to perform the double click action on.
 - Finally, call the perform method to execute the action.

iii. Button Down and Button Up:

- Button down and button up actions can be used for scenarios where you need to press and release a mouse button.
- You can use the clickAndHold method to simulate pressing down a button and the release method to release the button.
- Again, create an instance of the Actions class, chain the desired methods, and provide the element as needed.
- Finally, call the perform method to execute the action.

9. How can you handle different versions of Appium in your test automation setup?

- Stay updated with the latest stable version of Appium and its corresponding documentation.
- Prioritize compatibility with the target mobile devices and OS versions.
- Maintain a well-defined configuration management system to manage Appium installations.
- Test your automation scripts on different versions of Appium to ensure compatibility and identify any version-specific issues.
- Stay connected with the Appium community and forums to stay informed about updates, issues, and best practices.

10.How can you automate the toggling of Airplane Mode using Appium?

- Automating the toggling of Airplane Mode in Android using Appium involves utilizing the Appium driver's capabilities and the Android-specific methods.
- Here's an overview of the steps:

i. Set the Required Capabilities:

- When configuring your Appium session, make sure to set the necessary capabilities to interact with the device's networking configurations.
- For example, you can use the capability automationName with the value uiautomator2.

ii. Toggle Airplane Mode:

- Use the Appium driver's toggleAirplaneMode() method to automate the toggling of Airplane Mode.
- This method simulates the user action of toggling the Airplane Mode switch in the device's settings.

- Here's an example:

```
// Toggle Airplane Mode  
driver.toggleAirplaneMode();
```