Selenium WebDriver Recently Asked Interview Questions and Answers

Selenium WebDriver Basic Questions

1.What is Selenium?

Selenium is an **open-source automation testing tool** used primarily for **testing web applications** across different **browsers** (like Chrome, Firefox, Edge) and **operating systems** (Windows, macOS, Linux).

Here are some major advantages of using Selenium in web automation:

- **Cross-Browser Support**: Automates tests on multiple browsers like Chrome, Firefox, Safari, and Edge.
- **Cross-Platform Compatibility**: Works on different OS platforms.
- **Language Support**: Supports multiple programming languages such as **Java**, **Python**, **C#**, **Ruby**, and **JavaScript**.
- **Selenium Components**:
  - **Selenium WebDriver** – Interacts directly with the browser for automation.
  - **Selenium IDE** – A record-and-playback tool for creating quick tests.
  - **Selenium Grid** – Allows parallel test execution on multiple machines/browsers.
- **Integration Friendly**: Easily integrates with tools like **TestNG**, **JUnit**, **Maven**, **Jenkins**, and **Docker** for CI/CD pipelines.

2.Difference between Selenium 3 and Selenium 4?

Selenium 4 is a major upgrade over **Selenium 3**. The biggest change is that it now fully supports the **W3C WebDriver protocol**, which improves stability and compatibility across browsers. It introduces relative locators like **above(), below(),** and **near(),** making element identification easier. Selenium 4 allows **element-level screenshots**, which was not possible in Selenium 3.
There's also native support for **Chrome DevTools Protocol**, which lets you access network logs, console logs, and even mock geolocation. The Selenium Grid has been redesigned with better UI, Docker support, and parallel execution. Also, Selenium Manager helps with **automatic driver management**. Overall, Selenium 4 is more modern, stable, and developer-friendly.

Feature	Selenium 3	Selenium 4
WebDriver Protocol	JSON Wire Protocol	W3C WebDriver (Stable & Standard)
Browser Communication	Via protocol conversion layer	Direct, faster browser communication
Relative Locators	❌ Not available	✅ `above(),below(),near()` etc.
Element Screenshot	❌ Only full page	✅ `getScreenshotAs()` on specific WebElement

Chrome DevTools Protocol (CDP)	❌ Not available	✅ Native support – logs, network, performance monitoring
Selenium Grid	Basic hub-node setup	Redesigned UI, Docker support, parallel execution
Driver Management	Manual or WebDriverManager	✅ Built-in Selenium Manager
Debugging & Logs	Basic	Enhanced error messages and better console output
Selenium IDE	Outdated	Redesigned with better UI, multi-browser support
Documentation	Minimal	Improved and beginner-friendly

3.What are the different types of locators in Selenium?

ID, Name, ClassName, TagName, LinkText, PartialLinkText, CSS Selector, XPath

4.Difference between findElement() and findElements()?

Feature	`findElement()`	`findElements()`
Return Type	Returns a **single WebElement**	Returns a **List of WebElements**
When Element is Found	Returns the **first matching** element	Returns **all matching** elements
When No Element is Found	**Throws `NoSuchElementException`**	Returns an **empty list**
Use Case	When you're sure **only one element** is needed	When you want to work with **multiple similar elements** (e.g., rows, links, checkboxes)

5.How do you handle dropdowns in Selenium?

In Selenium, if the dropdown is built using the HTML <select> tag, we handle it using the **Select class** from org.openqa.selenium.support.ui.Select.
First, I locate the dropdown WebElement using any locator, then create a Select object and use methods like:

- selectByVisibleText("Option") – selects by the text visible to the user
- selectByValue("option1") – selects by the value attribute in the HTML
- selectByIndex(0) – selects by the index (starts from 0)

For non-<select> dropdowns (like custom UI dropdowns), I handle them using **click actions, keyboard interactions, or JavaScript**.

6.How to handle multiple windows in Selenium?

Use getWindowHandles() and switchTo().window(windowHandle) methods

7.What is the difference between get() and navigate().to()?

Both **driver.get()** and **driver.navigate().to()** are used to load a new web page in the browser.

However, the main difference is in **flexibility**:

- driver.get("url") is a **simple method** used to launch a new URL and wait until the page is fully loaded.
- driver.navigate().to("url") **achieves the same result**, but it belongs to the navigate() interface, which also provides additional navigation methods like:
    - navigate().back() – Go to the previous page
    - navigate().forward() – Go to the next page
    - navigate().refresh() – Refresh the current page

So, if I just need to open a page, I can use get(). But when working with **browser history or navigation controls**, I prefer navigate().

8. How do you handle alerts and pop-ups?

In Selenium, we handle JavaScript alerts using the **Alert interface**. First, we switch the driver's focus to the alert using driver.switchTo().alert(), and then we can perform actions like **accept**, **dismiss**, or **read the alert text**.

9. How do you handle frames and iframes?

If a webpage contains a **frame or iframe**, Selenium **cannot directly access the elements inside it**. So, I use driver.switchTo().frame() to switch into that frame. This can be done by using the **frame's index**, **name or ID**, or a **WebElement** reference.

After interacting with the elements inside the frame, I use:

- driver.switchTo().parentFrame() – to go **one level up** (to the immediate parent frame)
- driver.switchTo().defaultContent() – to go **back to the main page** from any nested frame

10. What is the difference between driver.quit() and driver.close()?

- **driver.close()** is used to close the current browser window that the WebDriver is focused on. If multiple windows or tabs are open, it only closes the one in focus.
- **driver.quit()** is used to close all browser windows opened during the session and also ends the WebDriver session completely.

✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦

Selenium WebDriver Advanced Questions

11. What are the different types of waits in Selenium?

Implicit Wait, Explicit Wait, and Fluent Wait.

12. Implicit vs Explicit Wait vs Fluent wait - What's the difference?

Use **Implicit Wait** as a global default, **Explicit Wait** when I need to wait for a particular condition like visibility or clickability, and **Fluent Wait** when I need full control — like setting polling frequency or handling exceptions like NoSuchElementException

Feature	Implicit Wait	Explicit Wait	Fluent Wait
Applies To	All elements	Specific element/condition	Specific element/condition
How It Works	Waits for a fixed time before throwing exception	Waits until a condition is met	Waits until a condition is met with custom polling
Condition Support	❌ No (fixed time)	✅ Yes (like visibility, clickable, etc.)	✅ Yes
Polling Customization	❌ Not possible	❌ Not possible	✅ Can set polling frequency and exceptions to ignore

13. What is Fluent Wait, and when do you use it?

Fluent Wait is a type of wait in Selenium that allows you to **wait for a specific condition** with full control over:

- **Maximum timeout duration**
- **Polling interval** (how often Selenium checks the condition)
- **Exceptions to ignore** (like NoSuchElementException)

It’s useful for handling elements that appear **dynamically or with delays**, especially in **Ajax-heavy or slow-loading applications**.

14. How do you handle 'StaleElementReferenceException'?

I use JavaScriptExecutor when standard Selenium methods are unreliable — especially for complex or JavaScript-heavy web apps. It helps me interact with elements directly at the DOM level.
StaleElementReferenceException occurs when the **WebElement is no longer attached to the current DOM**. This usually happens when:

- The page is **refreshed**
- The **DOM is updated dynamically** (e.g., via AJAX or JavaScript)
- The **element is reloaded or replaced**

To handle this, I **re-locate the element just before interacting with it**. If the element is dynamic, I use a **try-catch block with a retry** or use **ExpectedConditions.refreshed()** to wait for a fresh version of the element before proceeding.

15. What is JavaScriptExecutor? How do you use it?

JavaScriptExecutor is an interface in Selenium that allows us to execute JavaScript code directly in the browser.
I use it when normal WebDriver commands don’t work or fail due to dynamic UI behavior.

Why I use JavaScriptExecutor:

- To **scroll the page** when elements are not in view
- To **click elements** that are hidden, overlapped, or not clickable via .click()
- To **set or get values** from fields directly
- To **highlight elements** for debugging purposes
- To **fetch page properties** like title or URL via JS

```
JavascriptExecutor js = (JavascriptExecutor) driver;

// Scroll down
js.executeScript("window.scrollBy(0, 500);");

// Click an element using JS
js.executeScript("arguments[0].click();", element);

// Set value to input box
js.executeScript("arguments[0].value='Selenium';", inputBox);
```

16. How do you scroll a webpage using Selenium?

I generally prefer **JavaScriptExecutor** for scrolling, especially when working with lazy-loaded content or dynamically rendered pages. It gives me more control and works even when elements are not yet visible.
I use the Actions class when the application responds well to keyboard-based interactions like **PAGE_DOWN.**

1. Using JavaScriptExecutor (Most Common & Flexible)
I use JavaScriptExecutor when I want precise control over scrolling — vertically, horizontally, to an element, or to the bottom of the page.

```
// Scroll down by 500 pixels

((JavascriptExecutor) driver).executeScript("window.scrollBy(0, 500);");

// Scroll to the bottom of the page

((JavascriptExecutor) driver).executeScript("window.scrollTo(0, document.body.scrollHeight);");

// Scroll to a specific element

WebElement element = driver.findElement(By.id("footer"));

((JavascriptExecutor) driver).executeScript("arguments[0].scrollIntoView(true);", element);
```

2. Using Actions Class (When dealing with keyboard-based scrolling)

If the page responds to **keyboard inputs**, I use the Actions class to send PAGE_DOWN, ARROW_DOWN, or other keys.

```
Actions actions = new Actions(driver);

actions.sendKeys(Keys.PAGE_DOWN).build().perform();
```

17. How do you take a screenshot in Selenium?

I use the **TakesScreenshot** interface in Selenium. I cast the driver to **TakesScreenshot** and use **getScreenshotAs()** to capture the image. Then I save it using **FileUtils.**
I usually create a utility method for this and call it in failure scenarios or through listeners.

18. How do you handle file uploads in Selenium?

I handle file uploads by sending the **full file path** to the file input element using **sendKeys()**. This works when the element has **type="file"** in the HTML.

```
driver.findElement(By.id("upload")).sendKeys("C:\\path\\to\\file.txt");
```

19. How do you validate broken links in Selenium?

I collect all the anchor (<a>) tags using Selenium, extract their href attributes, and then use **Java's HttpURLConnection** to send an HTTP request to each URL.
If the response code is **400 or above**, I mark it as a broken link.

```
HttpURLConnection connection = (HttpURLConnection) new URL(url).openConnection();
connection.setRequestMethod("HEAD");

int statusCode = connection.getResponseCode();
```

20. How do you capture network logs in Selenium ?

To capture network logs in Selenium, I use **Chrome DevTools Protocol (CDP)** with **ChromeOptions** and **LoggingPreferences**.
This allows me to capture performance logs, including API calls, responses, status codes, and request URLs.

```
ChromeOptions options = new ChromeOptions();

LoggingPreferences logs = new LoggingPreferences();

logs.enable(LogType.PERFORMANCE, Level.ALL);

options.setCapability(CapabilityType.LOGGING_PREFS, logs);


WebDriver driver = new ChromeDriver(options);


LogEntries entries = driver.manage().logs().get(LogType.PERFORMANCE);

for (LogEntry entry : entries) {

   System.out.println(entry.getMessage());

}
```

✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦

Selenium Framework Questions for Best Practices

21. What is the Page Object Model (POM)?

Page Object Model (POM) is a design pattern in Selenium that helps us create **separate Java classes for each web page**, where we store all the web elements and the related actions (methods).
It improves **code readability, reusability, and maintainability**, especially in large test automation projects.

22. What is Page Factory? How is it different from POM?

Page Factory is an advanced version of the Page Object Model (POM) in Selenium. It simplifies object initialization by using annotations like **@FindBy** instead of repeatedly writing **driver.findElement().** One of its major advantages is **lazy loading**, meaning web elements are only located when they are actually used, not at the time of object creation. This improves test performance and reduces unnecessary overhead. Page Factory makes test code cleaner, more readable, and easier to maintain — especially in large-scale test automation frameworks.

23. What is the difference between @FindBy and driver.findElement()?

@FindBy is used **with Page Factory** to declare web elements in a clean and organized way. It supports **lazy loading**, meaning the element is located only when it's accessed — improving performance and reducing overhead.
On the other hand, **driver.findElement()** is a direct method that **immediately locates an element** on the page each time it's called.

24. What are Selenium Grid and its advantages?

Selenium Grid is a tool that allows me to run tests in **parallel across multiple machines, browsers, and operating systems**, helping achieve **cross-browser testing** efficiently.
It follows a **Hub–Node architecture**, where the **Hub** manages the test distribution and the **Nodes** execute the tests on specified environments.
I use Selenium Grid to **scale test execution**, **reduce total execution time**, and ensure my application works consistently across different platforms.

25. How do you handle dynamic elements in Selenium?

I handle dynamic elements using **smart and flexible locators** like **dynamic XPath** or **CSS Selectors**. Instead of relying on exact attribute values, I use XPath functions like **contains**(), **starts-with(),** and **normalize-space()** to identify elements whose IDs or classes change frequently.

I avoid **unstable or hard-coded locators** that can break easily when the UI changes. Instead, I write **robust XPath expressions** using relative paths, parent-child or sibling relationships.

In complex or highly dynamic scenarios, I also use **explicit waits** or **JavaScriptExecutor** to ensure elements are available and ready for interaction

26. What is the role of Desired capabilities in Selenium?

DesiredCapabilities is used in Selenium to define a set of **key–value pairs** that specify browser properties, platform details, and custom preferences required for test execution.
It plays a key role when working with **Selenium Grid**, **RemoteWebDriver**, or cloud platforms like **BrowserStack** and **Sauce Labs**, where you need to specify the environment remotely.

I use D**esiredCapabilities** to configure the **browser name**, **version**, **platform**, and other advanced settings before initializing the driver.
However, since it's **deprecated in newer Selenium versions**, its functionality is now replaced by browser-specific classes like **ChromeOptions, FirefoxOptions**, etc.

27. What are the different types of Assertions used in Selenium?

In Selenium automation, we commonly use assertions from **TestNG** or **JUnit** to validate expected outcomes.
The two main types are:

- **Hard Assert** – Stops the test execution immediately if an assertion fails.
  Commonly used from org.testng.Assert.
- **Soft Assert** – Allows the test to continue even if an assertion fails, and reports all failures at the end using **assertAll().**
  Provided by org.testng.asserts.SoftAssert.

If using **JUnit**, the org.junit.Assert class provides only hard assertions.
Assertions help in validating actual vs expected results, and are crucial for verifying application behavior during test execution

28. What are the limitations of Selenium WebDriver?

Selenium WebDriver is a powerful tool for browser automation, but it has some limitations:

- ❌ **No support for desktop applications** – It only works with web-based applications.
- ❌ **No built-in reporting** – We need to integrate tools like TestNG, Allure, or ExtentReports.
- ❌ **Cannot handle CAPTCHA or barcode validation** – These require manual intervention or third-party solutions.
- ❌ **Limited to browser automation only** – It doesn't support mobile app automation directly; we use Appium for that.
- ❌ **No image-based testing** – It cannot validate visual content or UI layout precisely.

- ❌ **Handling dynamic content or browser popups may need workarounds** – Like using **Robot** class or **JavaScriptExecutor.**

29. How do you integrate Selenium with TestNG?

I integrate Selenium with **TestNG** to manage test execution, organize test flow, and generate detailed reports.
Here's how I do it:

- I use **TestNG annotations** like @BeforeMethod, @AfterMethod, and @Test to define the setup, teardown, and test logic.
- I create a **testng.xml file** to control which test classes to run, set priorities, group tests, and enable parallel execution.
- I use **TestNG assertions** (Assert.assertEquals, Assert.assertTrue, etc.) to validate expected vs actual results in the test cases.
- TestNG automatically generates an **HTML report** after execution, showing passed, failed, and skipped tests.

This integration makes my Selenium framework more modular, maintainable, and suitable for CI/CD pipelines.

30. What are TestNG Listeners, and how do you implement them?

TestNG Listeners are used to perform custom actions at different points in the test execution lifecycle. For example, you can execute specific code when a test starts, passes, fails, or is skipped. They are especially useful for enhancing reporting, logging, capturing screenshots on failures, and integrating third-party tools like **ExtentReports** or **Jira.**

To implement a listener, I create a separate class that implements **listener interfaces** like **ITestListener** (for test-level events) or **IExecutionListener** (for suite-level events). In that class, I override only the methods I need — such as **onTestFailure** or **onTestSuccess.**

Once the listener class is ready, I **register** it either by:

- Adding the @Listeners annotation to my test class, or
- Declaring it in the testng.xml configuration file.

This setup allows me to automate actions like taking screenshots when a test fails, logging test results to a file, or sending email reports — all without modifying the main test logic.

✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦✦

Selenium Execution & Debugging questions

31. How do you run Selenium tests in headless mode?

To run Selenium tests in **headless mode**, configure the browser to run without a GUI using **ChromeOptions** or **FirefoxOptions**, and pass the "headless" argument. This enables test execution in the background without opening a visible browser window.

Headless mode is particularly useful in **CI/CD environments**, **Docker containers**, or servers without a graphical interface. It is commonly used in automated pipelines like **Jenkins** or **GitHub Actions** to speed up execution and save system resources.
Functionally, headless mode behaves the same as regular mode, but without displaying the browser UI.

32. How do you handle authentication pop-ups in Selenium?

Basic authentication pop-ups (browser-based) can be handled by embedding credentials directly into the URL, using the format:
http://username:password@your-url.com

If the browser blocks this approach (as some modern browsers do), tools like **Robot class**, **AutoIT**, or **Sikuli** can be used to interact with the OS-level pop-up windows.

33. How to execute parallel tests in Selenium?

Parallel test execution in Selenium can be achieved in two main ways:

- **Using TestNG**:
  Configure parallel execution in the testng.xml file by setting the parallel attribute (e.g., "tests" or "methods") and defining the thread-count. This allows multiple test classes or methods to run simultaneously.

- **Using Selenium Grid**:
  Distribute test execution across multiple machines or browser instances by connecting to a central Hub and executing tests on different Nodes in parallel. This is ideal for cross-browser and cross-platform testing.

Parallel execution helps reduce total test execution time and improves efficiency in large-scale automation suites.

34. How do you handle CAPTCHA in Selenium?

CAPTCHA is designed to prevent automation, so **it cannot be reliably automated using Selenium**.
In real-world testing, there are a few ways to handle it:

- **Disable CAPTCHA in the test environment**, if possible, to allow smooth automation.

- **Use manual intervention** for that step in rare cases where automation must pause.
- For complex scenarios, **third-party services** like **2Captcha**, **DeathByCaptcha**, or **OCR-based solutions** can be used to bypass CAPTCHA — though this is not recommended for secure or production systems.

The best practice is to ask the development team to **bypass or disable CAPTCHA in staging/test environments**, as the goal is to test the application's functionality — not the CAPTCHA itself.

35. What are the different ways to maximize a browser window in Selenium/TestNG?

Selenium provides multiple ways to maximize or control the browser window size:

- **Using driver.manage().window().maximize();**
  This is the most common and simple way to maximize the window to full screen.
- **Using driver.manage().window().setSize(new Dimension(width, height));**
  This is useful when you want to set a custom browser size, such as for responsive testing or simulating different screen resolutions.
- **Using driver.manage().window().fullscreen();**
  This launches the browser in full-screen mode, which is different from maximize

36. What would you do if a Selenium test runs fine locally but fails in CI (e.g., Jenkins)?

If a Selenium test passes locally but fails in CI (like Jenkins), I would:

- Check for environment differences (headless mode, browser versions, screen resolution).
- Run tests in headless mode with a fixed window size on CI.
- Make sure the right browser drivers are installed and configured in CI.
- Add explicit waits to handle slower loading in CI.
- Capture logs and screenshots on failures to debug easily.
- Remove any hardcoded paths or dependencies that exist only locally.

37. How do you debug a failing Selenium test?

To debug a failing test, I:

- Check the test logs and stack traces for errors.
- Capture screenshots at failure points to see what the browser displayed.
- Use browser developer tools (DevTools) to inspect elements and confirm locators.
- Add additional logging or breakpoints if running in debug mode.

- Verify environment settings like browser version, driver compatibility, and network issues.

This systematic debugging helps identify and fix test failures quickly.

38. What steps do you take if a Selenium test intermittently fails (flaky test)?

- Add or increase explicit waits to handle dynamic page loads.
- Review and improve element locators to make them more reliable.
- Check for any dependencies on test data or environment that might cause inconsistency.
- Re-run tests Multiple times to detect failure patterns and trace the cause.