**📌 Section 1: Selenium Basics**

**Q1. What is Selenium?**

* Selenium is an **open-source automation testing tool** used to automate web applications across different browsers and platforms.
* It supports multiple languages like **Java, Python, C#, Ruby, JavaScript**.
* It cannot automate desktop or mobile apps (only web apps).

**Q2. What are the components of Selenium Suite?**

1. **Selenium IDE** → A browser plugin to record and playback scripts (mostly for beginners).
2. **Selenium RC (Remote Control)** → Deprecated.
3. **Selenium WebDriver** → Most widely used; interacts directly with browsers.
4. **Selenium Grid** → Used for parallel execution and cross-browser testing.

**Q3. What are the advantages of Selenium?**

* Open-source and free.
* Supports multiple languages and browsers.
* Large community support.
* Integrates well with frameworks like TestNG, JUnit, Cucumber, Maven, Jenkins, etc.

**Q4. What are the limitations of Selenium?**

* Cannot automate desktop applications.
* No built-in reporting mechanism.
* Cannot handle Captcha, OTP, Barcodes.
* Needs third-party tools for image/visual testing.

**Q5. What is Selenium WebDriver?**

* It is a **core component of Selenium** that provides an API to directly interact with web browsers.
* Unlike Selenium RC, WebDriver communicates directly with the browser without an intermediate server.

**Q6. How does Selenium WebDriver work internally?**

1. Your test script → WebDriver API → JSON Wire Protocol → Browser Driver (e.g., chromedriver.exe).
2. Browser Driver translates JSON commands into browser-native commands.
3. Browser executes the action and sends the response back to WebDriver.

**Q7. What are different WebDriver supported browsers?**

* Chrome → **ChromeDriver**
* Firefox → **GeckoDriver**
* Safari → **SafariDriver**
* Edge → **EdgeDriver**
* Internet Explorer → **IEDriverServer**

**Q8. What are the different types of locators in Selenium?**

* **ID**
* **Name**
* **Class Name**
* **Tag Name**
* **Link Text**
* **Partial Link Text**
* **XPath**
* **CSS Selector**

**Q9. Which locator is preferred in Selenium and why?**

* **ID** is preferred → because it is unique and fastest.
* If ID is not available, use **Name** or **CSS Selector**.
* **XPath** should be used only if nothing else works (slower).

**Q10. What is the difference between Absolute XPath and Relative XPath?**

* **Absolute XPath:** Starts from the root (/html/body/...) and gives the complete path.
  + Example: /html/body/div[1]/form/input[1]
  + ❌ Very fragile → breaks if structure changes.
* **Relative XPath:** Starts from the middle of the DOM and uses attributes.
  + Example: //input[@id='username']
  + ✅ More reliable.

**Q11. What is the difference between findElement() and findElements()?**

* findElement() → Returns the first matching element (throws **NoSuchElementException** if not found).
* findElements() → Returns a list of elements (returns **empty list** if not found).

**Q12. How do you handle dynamic elements in Selenium?**

* Use **contains(), starts-with(), ends-with()** in XPath.
* Example: //input[contains(@id,'username')]
* Or use **CSS substring matches** → [id^='user'], [id$='name'], [id\*='ser']

**Q13. What is the difference between getText() and getAttribute() in Selenium?**

* getText() → Fetches the **visible text** of an element.
* getAttribute("value") → Fetches the **attribute value** (like value, href, src, etc.).

**Q14. What is the difference between driver.close() and driver.quit()?**

* driver.close() → Closes the **current browser window**.
* driver.quit() → Closes **all windows** opened by WebDriver and ends the session.

**Q15. How do you handle multiple windows in Selenium?**

* Use getWindowHandles() to fetch all window IDs.
* Switch using driver.switchTo().window(windowID).

**Q16. How do you handle alerts and popups in Selenium?**

* Use driver.switchTo().alert() → provides:
  + accept() → Click OK
  + dismiss() → Click Cancel
  + getText() → Get alert message
  + sendKeys("text") → Enter text in prompt

**Q17. How do you handle dropdowns in Selenium?**

* Use the **Select class** (org.openqa.selenium.support.ui.Select).
* Methods:
  + selectByVisibleText()
  + selectByValue()
  + selectByIndex()

**Q18. How do you handle frames in Selenium?**

* driver.switchTo().frame(index)
* driver.switchTo().frame(name/id)
* driver.switchTo().frame(WebElement)
* To exit → driver.switchTo().defaultContent()

**Q19. How do you handle synchronization in Selenium?**

* **Implicit Wait** → applies globally, waits for a fixed time.
* **Explicit Wait** → waits until a specific condition (like visibility, clickable).
* **Fluent Wait** → polls at regular intervals until condition met or timeout.

**Q20. How do you capture screenshots in Selenium?**

File src = ((TakesScreenshot) driver).getScreenshotAs(OutputType.FILE);

FileUtils.copyFile(src, new File("screenshot.png"));

**Q21. Can Selenium handle Captchas or OTP?**

* ❌ No, Selenium cannot directly handle Captchas/OTP.
* ✅ Workarounds:
  + Use test environment where Captcha/OTP is disabled.
  + Use third-party APIs (OCR, SMS API).

**Q22. Can Selenium automate desktop applications?**

* ❌ No, Selenium cannot.
* ✅ Tools like **AutoIT**, **Sikuli**, **Robot class**, or **WinAppDriver** can be integrated.

**📌 Section 2: WebDriver Architecture**

**Q1. What is Selenium WebDriver?**  
Selenium WebDriver is a web automation framework that allows you to execute tests across different browsers. It provides an object-oriented API that directly communicates with browsers without requiring an intermediate server (unlike Selenium RC).

* It uses each browser’s **native automation engine**.
* Example: ChromeDriver → automates Chrome, GeckoDriver → automates Firefox.

**Q2. Explain the Selenium WebDriver architecture.**  
Selenium WebDriver follows a **three-layer architecture**:

1. **Selenium Client Libraries**
   * Language-specific bindings (Java, Python, C#, etc.).
   * Test scripts are written here.
2. **JSON Wire Protocol / W3C Protocol**
   * Acts as a communication layer between the client libraries and the browser drivers.
   * Converts client requests into JSON format.
3. **Browser Drivers**
   * Each browser has its own driver (e.g., ChromeDriver, GeckoDriver, EdgeDriver).
   * Driver receives the JSON commands and interacts with the browser using native automation APIs.
4. **Browsers**
   * Actual browsers like Chrome, Firefox, Edge, Safari execute the instructions and return responses back through the driver.

🔹 **Flow:** Test Script → Client Library → Protocol → Browser Driver → Browser.

**Q3. What is the role of the Browser Driver in WebDriver architecture?**

* The Browser Driver acts as a **bridge** between Selenium scripts and the browser.
* It translates commands from Selenium into a form the browser understands (via JSON Wire Protocol/W3C protocol).
* Example: ChromeDriver interacts with Chrome, GeckoDriver with Firefox.
* Drivers are **specific to each browser** and need to be downloaded separately.

**Q4. What is JSON Wire Protocol? Is it still used?**

* **JSON Wire Protocol** is a transport mechanism used by Selenium 2 (WebDriver) to send commands between client libraries and browser drivers in JSON format.
* Example: { "sessionId": "1234", "command": "findElement", "locator": "id=username" }.
* Since Selenium 4, WebDriver uses the **W3C WebDriver Protocol** instead of JSON Wire Protocol for better standardization.

**Q5. Difference between JSON Wire Protocol and W3C WebDriver Protocol?**

| **Aspect** | **JSON Wire Protocol** | **W3C WebDriver Protocol** |
| --- | --- | --- |
| Standardization | Proprietary to Selenium | W3C (World Wide Web Consortium) standard |
| Browser Compatibility | Sometimes caused inconsistencies | Works consistently across browsers |
| Selenium Version | Selenium 2 & 3 | Selenium 4 onwards |
| Communication | JSON-based commands | W3C standardized format |

**Q6. Why did Selenium shift from JSON Wire Protocol to W3C Protocol?**

* JSON Wire Protocol had **compatibility issues** between browsers.
* Each browser implemented features differently.
* W3C protocol ensures **cross-browser consistency** and makes Selenium more stable.

**Q7. Can you explain the step-by-step execution flow of a Selenium WebDriver test?**

1. **Test script** written in Java (or other language) is executed.
2. Selenium Client Library translates it into JSON commands.
3. These commands are sent to the Browser Driver (e.g., ChromeDriver).
4. Browser Driver passes the commands to the actual Browser.
5. Browser executes the command using its native engine.
6. Response is sent back to the driver → client library → test script.

**Q8. What is the role of Selenium Grid in WebDriver architecture?**

* Selenium Grid allows **parallel test execution** across multiple browsers and machines.
* In architecture:
  + **Hub** → Central point controlling tests.
  + **Nodes** → Machines where browsers run.
* Test requests are sent from Selenium scripts to the **Hub**, which delegates them to the right **Node**.

**Q9. Do we need to install browsers for WebDriver to work?**

* Yes, the target browser (e.g., Chrome, Firefox) must be installed.
* WebDriver uses the installed browser’s **native APIs** to perform automation.

**Q10. What happens if the browser version and driver version mismatch?**

* WebDriver may **fail to launch** the browser or cause unexpected errors.
* Solution: Always use a **compatible driver version** matching your browser version.
* Tools like **WebDriverManager** (by Boni García) help manage driver versions automatically.

**📌 Section 3: Locators & WebElements**

**Q1. What are locators in Selenium?**

**Answer:**  
Locators are **mechanisms used to find elements** on a web page. They help Selenium identify and interact with elements like text boxes, buttons, links, etc.

**Q2. What are the different types of locators in Selenium?**

**Answer:**  
Selenium provides **8 locators**:

1. **id** – Finds element by unique id attribute.
2. **name** – Finds element by name attribute.
3. **className** – Finds element by class attribute.
4. **tagName** – Finds element by HTML tag (e.g., input, div).
5. **linkText** – Finds anchor (<a>) elements with exact link text.
6. **partialLinkText** – Finds <a> elements with partial text match.
7. **cssSelector** – Finds element using CSS expressions.
8. **xpath** – Finds element using XML path expressions.

**Q3. Which locator is the fastest and why?**

**Answer:**

* **id** is the fastest and most reliable locator because IDs are unique on a page.
* It directly maps to the DOM without requiring complex traversal.

**Q4. What is the difference between absolute XPath and relative XPath?**

**Answer:**

* **Absolute XPath** → Starts from the root (/html/body/...). Example:
* /html/body/div[1]/div[2]/input

❌ Fragile: Breaks if any DOM structure changes.

* **Relative XPath** → Starts from anywhere in the DOM (//). Example:
* //input[@id='username']

✅ More flexible and preferred in automation.

**Q5. Difference between findElement() and findElements()?**

**Answer:**

* findElement(By locator) → Returns the **first matching WebElement**. Throws NoSuchElementException if not found.
* findElements(By locator) → Returns a **List of WebElements** (can be empty if none found).

Example:

WebElement button = driver.findElement(By.id("submit")); // single element

List<WebElement> links = driver.findElements(By.tagName("a")); // multiple elements

**Q6. What is a WebElement in Selenium?**

**Answer:**

* A **WebElement** represents an HTML element on a web page.
* It provides methods to perform actions like click(), sendKeys(), getText(), etc.
* Example:
* WebElement username = driver.findElement(By.id("username"));
* username.sendKeys("testUser");

**Q7. What are some commonly used WebElement methods?**

**Answer:**

| **Method** | **Description** |
| --- | --- |
| click() | Clicks on element |
| sendKeys("text") | Enters text into input field |
| clear() | Clears input field |
| getText() | Gets visible text of element |
| getAttribute("attr") | Gets attribute value |
| isDisplayed() | Checks if element is visible |
| isEnabled() | Checks if element is enabled |
| isSelected() | Checks if checkbox/radio is selected |
| getCssValue("prop") | Gets CSS property value |

**Q8. What is the difference between getText() and getAttribute("value")?**

**Answer:**

* getText() → Retrieves **visible text** between opening and closing tags of an element.
* getAttribute("value") → Retrieves the **value attribute** of an element (useful for input fields).

Example:

<input type="text" id="name" value="John">

<p>Hello World</p>

driver.findElement(By.id("name")).getAttribute("value"); // John

driver.findElement(By.tagName("p")).getText(); // Hello World

**Q9. What are dynamic elements? How do you handle them?**

**Answer:**

* **Dynamic elements** have attributes (like id, name) that change every time the page loads.
* **Handling strategies:**
  + Use **relative XPath** with contains() or starts-with().
  + Use **CSS selectors** with \* or ^/$.
  + Example:
  + //input[contains(@id,'username')]
  + //button[starts-with(@id,'login')]

**Q10. Which is better: XPath or CSS Selector?**

**Answer:**

* **CSS Selector** is generally faster and cleaner.
* **XPath** is more powerful (supports traversal both ways: parent → child and child → parent).
* Use **CSS when possible**, but use **XPath when complex hierarchy is involved**.

**Q11. How do you handle elements with the same attributes?**

**Answer:**

* Use **indexing** in XPath:
* (//input[@type='text'])[2]
* Use **parent-child relationship**:
* //div[@id='login-form']//input[@type='password']

**Q12. How to check if an element exists on the page?**

**Answer:**

* Use findElements() (returns empty list if element not found):
* if(driver.findElements(By.id("username")).size() > 0) {
* System.out.println("Element exists");
* }

**Q13. What are shadow DOM elements? How to handle them in Selenium?**

**Answer:**

* Shadow DOM elements are inside a **shadow root** and are not accessible with normal locators.
* Handling requires **JavaScript Executor**:
* WebElement shadowHost = driver.findElement(By.cssSelector("#shadow-host"));
* SearchContext shadowRoot = shadowHost.getShadowRoot();
* WebElement elementInsideShadow = shadowRoot.findElement(By.cssSelector("#inside"));

**Q14. What is the difference between isDisplayed(), isEnabled(), and isSelected()?**

**Answer:**

* isDisplayed() → Returns **true** if element is visible on UI.
* isEnabled() → Returns **true** if element is enabled for interaction.
* isSelected() → Returns **true** if checkbox/radio/dropdown option is selected.

**📌 Section 4: WebDriver Commands**

**Q1. What are WebDriver commands in Selenium?**

**Answer:**  
WebDriver commands are the set of methods provided by the Selenium WebDriver API that allow testers to interact with browsers and web elements. They can be broadly categorized as:

1. **Browser Commands** (managing browser window, navigation, cookies, etc.)
2. **Navigation Commands** (forward, back, refresh, to)
3. **WebElement Commands** (click, sendKeys, clear, getText, etc.)
4. **Wait Commands** (implicit, explicit, fluent wait)

**Q2. What are the different categories of WebDriver commands?**

**Answer:**

1. **Browser Commands** → get(), getTitle(), getCurrentUrl(), getPageSource(), close(), quit().
2. **Navigation Commands** → navigate().to(), navigate().back(), navigate().forward(), navigate().refresh().
3. **WebElement Commands** → click(), sendKeys(), clear(), getText(), getAttribute(), isDisplayed(), isEnabled(), isSelected().
4. **Wait Commands** → implicitlyWait(), WebDriverWait (explicit), FluentWait.

**Q3. Explain the difference between close() and quit() in WebDriver.**

**Answer:**

* **driver.close()** → Closes the currently active browser window where the driver is running.
* **driver.quit()** → Quits the entire browser session, closing all windows opened by WebDriver.

**Example:**

driver.close(); // closes current tab/window

driver.quit(); // closes entire browser session

**Q4. How do you retrieve the current page title and URL?**

**Answer:**

* **getTitle()** → Returns the title of the current web page.
* **getCurrentUrl()** → Returns the URL of the current page.

**Example:**

String title = driver.getTitle();

String url = driver.getCurrentUrl();

System.out.println("Title: " + title);

System.out.println("URL: " + url);

**Q5. How do you fetch the page source in Selenium?**

**Answer:**

* **getPageSource()** → Retrieves the HTML source code of the current page.

**Example:**

String pageSource = driver.getPageSource();

System.out.println(pageSource);

⚠️ Not commonly used in automation scripts, but can help in debugging.

**Q6. What are Navigation commands in WebDriver?**

**Answer:**

* **navigate().to("url")** → Opens a new URL.
* **navigate().back()** → Moves one step back in browser history.
* **navigate().forward()** → Moves one step forward.
* **navigate().refresh()** → Refreshes the current page.

**Example:**

driver.navigate().to("https://google.com");

driver.navigate().back();

driver.navigate().forward();

driver.navigate().refresh();

**Q7. What are common WebElement commands and their usage?**

**Answer:**  
Some important **WebElement commands**:

1. click() → Clicks on an element.
2. sendKeys("text") → Enters text into input fields.
3. clear() → Clears the text from input fields.
4. getText() → Retrieves visible text of the element.
5. getAttribute("attr") → Retrieves the value of a given attribute.
6. isDisplayed() → Checks if the element is visible on the page.
7. isEnabled() → Checks if the element is enabled (usable).
8. isSelected() → Checks if a checkbox, radio button, or option is selected.

**Example:**

WebElement searchBox = driver.findElement(By.name("q"));

searchBox.sendKeys("Selenium WebDriver");

searchBox.clear();

searchBox.sendKeys("Java");

System.out.println(searchBox.getAttribute("name"));

**Q8. How do you differentiate between getText() and getAttribute("value")?**

**Answer:**

* **getText()** → Returns visible text of the element on the web page.
* **getAttribute("value")** → Returns the hidden attribute value (like the value in an input box).

**Example:**

WebElement input = driver.findElement(By.id("username"));

input.sendKeys("John");

System.out.println(input.getText()); // prints nothing (input field has no inner text)

System.out.println(input.getAttribute("value")); // prints John

**Q9. How do you handle checkboxes and radio buttons in Selenium?**

**Answer:**

* Use isSelected() to check if it is already selected.
* Use click() to select or deselect (depending on element behavior).

**Example:**

WebElement checkbox = driver.findElement(By.id("subscribe"));

if(!checkbox.isSelected()) {

checkbox.click();

}

**Q10. How do you check if an element is visible or enabled before performing actions?**

**Answer:**

* **isDisplayed()** → returns true if element is visible.
* **isEnabled()** → returns true if element is enabled for interaction.

**Example:**

WebElement loginBtn = driver.findElement(By.id("login"));

if(loginBtn.isDisplayed() && loginBtn.isEnabled()) {

loginBtn.click();

}

**Q11. Difference between findElement() and findElements()?**

**Answer:**

* **findElement(By locator)** → returns the first matching element. Throws NoSuchElementException if not found.
* **findElements(By locator)** → returns a list of all matching elements. Returns an **empty list** if none are found (no exception).

**Example:**

WebElement element = driver.findElement(By.tagName("a"));

List<WebElement> allLinks = driver.findElements(By.tagName("a"));

**Q12. How do you handle dropdowns in Selenium WebDriver?**

**Answer:**  
Using the **Select class** (from org.openqa.selenium.support.ui.Select).

**Methods in Select class:**

* selectByVisibleText("Option")
* selectByValue("val")
* selectByIndex(2)

**Example:**

WebElement dropdown = driver.findElement(By.id("country"));

Select select = new Select(dropdown);

select.selectByVisibleText("India");

**📌 Section 5: Waits in Selenium**

**Q1. Why do we need waits in Selenium?**

**Answer:**  
Web applications often take time to load elements due to network delays, dynamic content, or AJAX calls. Without waits, Selenium might throw **NoSuchElementException** because the element isn’t ready when the script tries to interact with it.  
👉 Waits help **synchronize test execution with the browser**.

**Q2. What are the types of waits in Selenium?**

**Answer:**

1. **Implicit Wait** – Global wait for all elements.
2. **Explicit Wait** – Wait for a specific condition before proceeding.
3. **Fluent Wait** – A type of explicit wait with polling frequency and exception handling.

**A. Implicit Wait**

**Q3. What is Implicit Wait in Selenium?**

**Answer:**

* Implicit wait is applied globally for the entire WebDriver session.
* It tells WebDriver to wait for a certain amount of time before throwing NoSuchElementException.

**Q4. Example of Implicit Wait?**

driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);

driver.get("https://example.com");

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

**Q5. Pros & Cons of Implicit Wait?**

**Pros:**

* Simple to implement.
* Applies to all elements automatically.

**Cons:**

* Cannot be applied conditionally to specific elements.
* Slows down execution when elements are not present.

**B. Explicit Wait**

**Q6. What is Explicit Wait in Selenium?**

**Answer:**

* Explicit wait is applied to **specific elements** based on a condition.
* It waits until the condition is met (e.g., visibility, clickability, presence of element).

**Q7. Example: Explicit Wait with visibility condition?**

WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));

WebElement element = wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("password")));

element.sendKeys("mypassword");

**Q8. What are common ExpectedConditions used with Explicit Wait?**

**Answer:**

* visibilityOfElementLocated(By locator)
* elementToBeClickable(By locator)
* presenceOfElementLocated(By locator)
* textToBePresentInElementValue(By locator, "value")
* alertIsPresent()

**Q9. Example: Explicit Wait for Alert?**

WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));

Alert alert = wait.until(ExpectedConditions.alertIsPresent());

alert.accept();

**C. Fluent Wait**

**Q10. What is Fluent Wait in Selenium?**

**Answer:**

* Fluent Wait is an advanced form of Explicit Wait.
* It defines maximum timeout, polling interval, and exceptions to ignore.

**Q11. Example of Fluent Wait?**

Wait<WebDriver> wait = new FluentWait<>(driver)

.withTimeout(Duration.ofSeconds(20)) // max wait time

.pollingEvery(Duration.ofSeconds(2)) // check every 2 seconds

.ignoring(NoSuchElementException.class);

WebElement element = wait.until(driver -> driver.findElement(By.id("loginBtn")));

**Q12. Difference between Explicit Wait and Fluent Wait?**

| **Feature** | **Explicit Wait** | **Fluent Wait** |
| --- | --- | --- |
| Timeout | Yes | Yes |
| Polling Interval | Fixed (500 ms) | Customizable |
| Exception Handling | Default | Customizable |
| Usage | Specific conditions | Specific conditions with flexibility |

**D. Comparison of Waits**

**Q13. Difference between Implicit and Explicit Wait?**

| **Aspect** | **Implicit Wait** | **Explicit Wait** |
| --- | --- | --- |
| Scope | Applies globally | Applies to specific element |
| Conditions | Only for element presence | Multiple conditions (visibility, clickable, etc.) |
| Flexibility | Less flexible | Highly flexible |
| Usage | General delays | Targeted synchronization |

**Q14. Can we use Implicit Wait and Explicit Wait together?**

**Answer:**

* **Not recommended.** Mixing them may cause **unexpected wait times** and slow down execution.
* Best practice → use **Explicit Wait** for dynamic elements.

**E. Advanced Qs**

**Q15. What is Thread.sleep()? How is it different from Selenium waits?**

**Answer:**

* Thread.sleep(ms) is a Java method that pauses execution for fixed time.
* It is **not Selenium-specific** and does not check for element presence/visibility.
* Selenium waits are smarter since they **wait only until conditions are met** (saves execution time).

**Q16. Which wait is best to use in real projects?**

**Answer:**

* **Implicit Wait** → For simple static apps.
* **Explicit Wait** → For dynamic apps with AJAX.
* **Fluent Wait** → For highly dynamic apps where polling frequency needs control.

**📌 Section 6: Handling Different Web Elements**

**Q1. How do you handle text boxes in Selenium?**

**Answer:**

* Use sendKeys() to enter text.
* Use clear() before typing if the field already contains a value.

WebElement username = driver.findElement(By.id("username"));

username.clear();

username.sendKeys("testuser");

**Q2. How do you handle buttons in Selenium?**

**Answer:**

* Use click() to perform a click action.

WebElement loginBtn = driver.findElement(By.id("login"));

loginBtn.click();

**Q3. How do you handle checkboxes in Selenium?**

**Answer:**

* Use click() to select/deselect.
* Use isSelected() to verify state.

WebElement checkbox = driver.findElement(By.id("subscribe"));

if (!checkbox.isSelected()) {

checkbox.click();

}

**Q4. How do you handle radio buttons?**

**Answer:**

* Similar to checkboxes, but only one option can be selected within a group.

WebElement maleRadio = driver.findElement(By.id("male"));

if (!maleRadio.isSelected()) {

maleRadio.click();

}

**Q5. How do you handle dropdowns in Selenium?**

**Answer:**

* Selenium provides the Select class for <select> HTML elements.

WebElement countryDropdown = driver.findElement(By.id("country"));

Select country = new Select(countryDropdown);

// Different ways to select

country.selectByIndex(1);

country.selectByValue("IN");

country.selectByVisibleText("India");

**Q6. How do you handle multi-select dropdowns?**

**Answer:**

* Multi-select dropdown allows selecting multiple options.

WebElement multiSelect = driver.findElement(By.id("skills"));

Select select = new Select(multiSelect);

if (select.isMultiple()) {

select.selectByValue("Java");

select.selectByVisibleText("Python");

select.deselectByValue("Java"); // remove selection

}

**Q7. How do you handle file upload in Selenium?**

**Answer:**

* For <input type="file">, use sendKeys() with the file path.

WebElement upload = driver.findElement(By.id("uploadFile"));

upload.sendKeys("C:\\Users\\Vishal\\Documents\\resume.pdf");

**Q8. How do you handle file download in Selenium?**

**Answer:**

* Selenium alone **cannot handle OS-level download dialogs**.
* Common approaches:
  + Change **browser preferences** to auto-download.
  + Use **Robot class / AutoIT / Desktop automation tools**.

Example (Chrome preferences):

HashMap<String, Object> prefs = new HashMap<>();

prefs.put("download.default\_directory", "C:\\Downloads");

ChromeOptions options = new ChromeOptions();

options.setExperimentalOption("prefs", prefs);

WebDriver driver = new ChromeDriver(options);

**Q9. How do you handle images in Selenium?**

**Answer:**

* Selenium cannot validate images visually.
* You can check:
  + isDisplayed() → image is visible.
  + getAttribute("src") → verify correct image URL.

WebElement logo = driver.findElement(By.id("logo"));

boolean isDisplayed = logo.isDisplayed();

String src = logo.getAttribute("src");

**Q10. How do you handle links in Selenium?**

**Answer:**

* Use click() to navigate.
* Get link text or URL with getText() and getAttribute("href").

WebElement link = driver.findElement(By.linkText("Contact Us"));

System.out.println(link.getAttribute("href"));

link.click();

**Q11. How do you handle tooltips in Selenium?**

**Answer:**

* Tooltips are usually in the title attribute.
* Use getAttribute("title").

WebElement tooltip = driver.findElement(By.id("helpIcon"));

String tipText = tooltip.getAttribute("title");

**Q12. How do you handle hidden elements in Selenium?**

**Answer:**

* Hidden elements (with CSS display:none) cannot be interacted with.
* We can:
  + Use **JavaScriptExecutor** to interact.
  + Or request developer to use proper attributes.

JavascriptExecutor js = (JavascriptExecutor) driver;

js.executeScript("document.getElementById('hiddenBtn').click();");

**Q13. How do you handle dynamic elements (changing IDs)?**

**Answer:**

* Use **relative locators (XPath, CSS)**.
* Use **starts-with, contains, normalize-space** in XPath.

WebElement dynamicBtn = driver.findElement(By.xpath("//button[contains(@id,'login')]"));

dynamicBtn.click();

**Q14. How do you handle sliders in Selenium?**

**Answer:**

* Use **Actions class** to drag slider.

WebElement slider = driver.findElement(By.id("slider"));

Actions move = new Actions(driver);

move.dragAndDropBy(slider, 50, 0).perform(); // move slider by 50px

**Q15. How do you handle calendars (date pickers)?**

**Answer:**

* Common approaches:
  1. **Send date as text** if input allows.
  2. **Click through calendar widget** and select the date.

driver.findElement(By.id("dateInput")).sendKeys("08/16/2025");

Or (dynamic selection via XPath):

driver.findElement(By.xpath("//td[text()='16']")).click();

**📌 Section 6B: Actions Class in Selenium**

**Q1. What is the Actions class in Selenium?**

* Actions class is used for **advanced user interactions** (mouse & keyboard events).
* Located in org.openqa.selenium.interactions.Actions.
* It generates complex actions like hover, drag-and-drop, right-click, double-click, etc.

**Q2. How do you perform mouse hover in Selenium?**

WebElement menu = driver.findElement(By.id("menu"));

Actions actions = new Actions(driver);

actions.moveToElement(menu).perform();

**Q3. How do you perform right-click (context click)?**

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

Actions actions = new Actions(driver);

actions.contextClick(element).perform();

**Q4. How do you perform double-click?**

WebElement button = driver.findElement(By.id("doubleClickBtn"));

Actions actions = new Actions(driver);

actions.doubleClick(button).perform();

**Q5. How do you perform drag-and-drop in Selenium?**

**Answer:**

WebElement source = driver.findElement(By.id("drag"));

WebElement target = driver.findElement(By.id("drop"));

Actions actions = new Actions(driver);

actions.dragAndDrop(source, target).perform();

Or by offset:

actions.dragAndDropBy(source, 100, 50).perform();

**Q6. How do you perform click and hold?**

**Answer:**

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

Actions actions = new Actions(driver);

actions.clickAndHold(element).pause(2000).release().perform();

**Q7. How do you perform keyboard actions using Actions class?**

**Answer:**

* Supports key press, release, and combined actions.

Actions actions = new Actions(driver);

// Press SHIFT and type HELLO in uppercase

WebElement input = driver.findElement(By.id("textBox"));

actions.keyDown(Keys.SHIFT).sendKeys(input, "hello").keyUp(Keys.SHIFT).perform();

**Q8. Difference between build().perform() and perform() in Actions?**

**Answer:**

* perform() → executes the action immediately.
* build().perform() → useful when combining **multiple actions** before execution.

Example:

Actions actions = new Actions(driver);

Action series = actions.moveToElement(el1).click().moveToElement(el2).click().build();

series.perform();

**Q9. Can Actions class handle multi-key combinations (like Ctrl+C, Ctrl+V)?**

**Answer:**  
Yes. Example:

Actions actions = new Actions(driver);

actions.keyDown(Keys.CONTROL).sendKeys("c").keyUp(Keys.CONTROL).perform(); // Copy

**Q10. What is the difference between Actions class and Robot class?**

**Answer:**

* **Actions class** → works only inside browser (web elements).
* **Robot class** → handles OS-level events (keyboard, mouse, file uploads).
* Example: Ctrl+S in Notepad cannot be done by Actions, but Robot can.

**📌 Section 7: Advanced Selenium**

**Q1. What is StaleElementReferenceException and how do you handle it?**

* This occurs when the WebElement is **no longer attached to the DOM** (page refreshed, navigated, or element re-rendered by JS).
* Example scenario: After clicking a button, the DOM reloads, making old element references invalid.

**Handling Approaches:**

1. **Re-locate the element** after DOM change:
2. WebElement element = driver.findElement(By.id("submit"));
3. element.click();
4. **Use Explicit Waits** (wait for element to be reattached).
5. WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));
6. WebElement element = wait.until(ExpectedConditions.presenceOfElementLocated(By.id("submit")));
7. Use **retry logic** in a loop if necessary.

**Q2. What is JavaScriptExecutor and when do you use it?**

* JavaScriptExecutor allows executing JavaScript directly in the browser from Selenium.
* Useful for cases when:
  + Elements are hidden but need interaction.
  + Performing scrolls.
  + Retrieving values not exposed directly.

**Example 1: Scroll into view**

JavascriptExecutor js = (JavascriptExecutor) driver;

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

js.executeScript("arguments[0].scrollIntoView(true);", element);

**Example 2: Click hidden element**

js.executeScript("arguments[0].click();", element);

**Q3. How do you take screenshots in Selenium?**

* Selenium provides TakesScreenshot interface.

File srcFile = ((TakesScreenshot) driver).getScreenshotAs(OutputType.FILE);

FileUtils.copyFile(srcFile, new File("screenshot.png"));

* You can also take element-level screenshots (Selenium 4 feature):

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

File screenshot = element.getScreenshotAs(OutputType.FILE);

**Q4. What are Page Load Strategies in Selenium?**  
Defines how Selenium waits for page load:

1. **normal (default)** – waits until all resources (JS, CSS, images) are loaded.
2. **eager** – waits until the initial HTML document is loaded and parsed (no need for images, stylesheets).
3. **none** – doesn’t wait for anything; returns control immediately.

**Usage:**

ChromeOptions options = new ChromeOptions();

options.setPageLoadStrategy(PageLoadStrategy.EAGER);

WebDriver driver = new ChromeDriver(options);

**Q5. What is Headless Browser Testing?**

* Running tests **without opening the browser UI** (faster & CI-friendly).
* Common use cases: CI/CD pipelines, running on servers without GUI.

**Example (Chrome Headless):**

ChromeOptions options = new ChromeOptions();

options.addArguments("--headless");

WebDriver driver = new ChromeDriver(options);

**Other headless drivers:**

* **HtmlUnitDriver** (lightweight, Java-based).
* **Firefox headless mode** also available.

**Q6. How do you handle Shadow DOM elements in Selenium?**

* Shadow DOM encapsulates elements (not accessible via normal locators).
* Must use **JavaScriptExecutor** to pierce shadow roots.

**Example:**

JavascriptExecutor js = (JavascriptExecutor) driver;

WebElement shadowHost = driver.findElement(By.cssSelector("#shadow-host"));

SearchContext shadowRoot = (SearchContext) js.executeScript("return arguments[0].shadowRoot", shadowHost);

WebElement shadowElement = shadowRoot.findElement(By.cssSelector("#shadow-button"));

shadowElement.click();

**📌 Section 8: Test Automation Framework**

**Q1. What is a Test Automation Framework and why is it needed?**

**Answer:**

* A **framework** is a set of guidelines, rules, and reusable utilities that help in writing, organizing, and executing test scripts in a structured way.
* Benefits:
  + Improves **code reusability** and **readability**.
  + Promotes **maintainability** and **scalability**.
  + Separates **test logic** from **test data**.
  + Provides **reporting & logging**.
  + Supports **CI/CD integration**.

**Q2. What is the Page Object Model (POM)?**

**Answer:**

* A design pattern where each **page of the application** is represented as a **Java class**.
* **Elements** → defined as class variables.
* **Actions (methods)** → defined as class methods.

**Example:**

public class LoginPage {

WebDriver driver;

By username = By.id("username");

By password = By.id("password");

By loginBtn = By.id("login");

public LoginPage(WebDriver driver) {

this.driver = driver;

}

public void login(String user, String pass) {

driver.findElement(username).sendKeys(user);

driver.findElement(password).sendKeys(pass);

driver.findElement(loginBtn).click();

}

}

✅ Advantage: Reusable, easy to maintain, reduces code duplication.

**Q3. What is Page Factory? How is it different from POM?**

**Answer:**

* Page Factory is an **extension of POM** provided by Selenium.
* Uses **@FindBy annotations** instead of driver.findElement().
* PageFactory.initElements(driver, this) initializes elements automatically.

**Example:**

public class LoginPage {

@FindBy(id = "username") WebElement username;

@FindBy(id = "password") WebElement password;

@FindBy(id = "login") WebElement loginBtn;

public LoginPage(WebDriver driver) {

PageFactory.initElements(driver, this);

}

public void login(String user, String pass) {

username.sendKeys(user);

password.sendKeys(pass);

loginBtn.click();

}

}

✅ Cleaner, readable, but less flexible than plain POM in some cases.

**Q4. What is Data-Driven Testing? How do you implement it?**

**Answer:**

* A technique where **test data is stored externally** (Excel, CSV, DB, JSON, Properties file) and test scripts read from it.
* Avoids hardcoding and makes tests reusable for multiple inputs.

**Example (Excel with Apache POI):**

FileInputStream fis = new FileInputStream("data.xlsx");

XSSFWorkbook wb = new XSSFWorkbook(fis);

XSSFSheet sheet = wb.getSheet("Login");

String username = sheet.getRow(1).getCell(0).getStringCellValue();

✅ Supports **cross-browser**, **multiple test scenarios**, and **large datasets**.

**Q5. What is Keyword-Driven Testing?**

**Answer:**

* Uses **keywords (actions)** to represent test steps (e.g., "Click", "EnterText", "VerifyTitle").
* Keywords are mapped to reusable methods in code.
* Test cases can be stored in Excel, where each row is a step.

✅ Advantage: Testers with less coding knowledge can still design test cases.

**Q6. What is a Hybrid Framework?**

**Answer:**

* A **combination of Data-Driven + Keyword-Driven + POM/Page Factory**.
* Most companies use Hybrid frameworks because they provide:
  + **Reusability** (keywords).
  + **Flexibility** (data-driven).
  + **Maintainability** (POM).

**Q7. What kind of Utility Classes do you create in a framework?**

**Answer:**  
Common utilities include:

* **ExcelUtil** → Read/write Excel test data.
* **ConfigReader** → Read from config.properties.
* **WebDriverManager/DriverFactory** → Manage driver instances.
* **WaitUtils** → Explicit waits & fluent waits.
* **ScreenshotUtil** → Capture screenshots on failure.
* **LogUtil** → Logging (Log4j/SLF4J).

**Q8. How do you design a maintainable framework?**

**Answer:**  
Key principles:

1. **Separation of Concerns** → Keep test logic, locators, and data separate.
   * (POM for locators, Excel/DB for data, TestNG/JUnit for execution).
2. **Reusability** → Common utilities for waits, logging, reporting.
3. **Modularity** → Divide framework into clear modules (base, pages, utils, tests).
4. **CI/CD integration** → Compatible with Jenkins/GitHub Actions.
5. **Scalability** → Easy to add new test cases, new browsers.
6. **Reporting** → ExtentReports/Allure integrated.
7. **Error handling** → Retry logic, exception handling for stability.

✅ Example framework folder structure:

src/test/java

├── base

│ └── BaseTest.java

├── pages

│ └── LoginPage.java

├── tests

│ └── LoginTest.java

├── utils

│ ├── ExcelUtil.java

│ ├── ConfigReader.java

│ └── WebDriverFactory.java

resources

├── config.properties

└── testdata.xlsx

**📌 Section 9: TestNG / JUnit Integration**

**9.1 TestNG Annotations**

**Q1. What are the main TestNG annotations and their order of execution?**  
**A1.**  
Execution order in TestNG:

1. @BeforeSuite
2. @BeforeTest
3. @BeforeClass
4. @BeforeMethod
5. @Test
6. @AfterMethod
7. @AfterClass
8. @AfterTest
9. @AfterSuite

**Q2. Difference between @BeforeMethod and @BeforeClass?**

* @BeforeMethod: Runs **before each test method**. (e.g., open browser)
* @BeforeClass: Runs **once per class**. (e.g., login to app)

**Q3. Can we disable a test case in TestNG?**  
Yes, using @Test(enabled = false).

**Q4. How do you set priority of tests?**

@Test(priority = 1)

public void loginTest() { ... }

@Test(priority = 2)

public void purchaseTest() { ... }

**Q5. What happens if priority is not defined?**  
TestNG runs tests in **alphabetical order** of method names.

**Q6. How do you run a test multiple times?**  
Using invocationCount:

@Test(invocationCount = 5)

public void retryTest() { ... }

**Q7. Difference between dependsOnMethods and dependsOnGroups?**

* dependsOnMethods: Test will run only if specified methods pass.
* dependsOnGroups: Test will run only if specified group tests pass.

**9.2 Assertions (Hard vs Soft)**

**Q1. What are hard assertions in TestNG?**

* Stop execution immediately if assertion fails.

Assert.assertEquals(title, "Google");

**Q2. What are soft assertions in TestNG?**

* Collect all failures and report at end.

SoftAssert sa = new SoftAssert();

sa.assertEquals(title, "Google");

sa.assertTrue(element.isDisplayed());

sa.assertAll();

**Q3. When to use hard vs soft assertions?**

* Hard: Critical checks (e.g., login).
* Soft: Multiple verifications in same test (e.g., UI checks).

**9.3 TestNG XML (suite file)**

**Q1. What is the use of testng.xml?**

* Controls test execution: grouping, parallel runs, parameters, selective execution.

**Q2. How to run tests in parallel using testng.xml?**

<suite name="ParallelSuite" parallel="tests" thread-count="2">

<test name="Test1">

<classes>

<class name="tests.LoginTest"/>

</classes>

</test>

<test name="Test2">

<classes>

<class name="tests.SignupTest"/>

</classes>

</test>

</suite>

**Q3. How do you pass parameters via testng.xml?**

<parameter name="browser" value="chrome"/>

@Test

@Parameters("browser")

public void launchBrowser(String browser) { ... }

**Q4. Difference between parallel=tests vs parallel=methods?**

* tests: Runs entire <test> blocks in parallel.
* methods: Runs test methods inside same class in parallel.

**9.4 DataProviders**

**Q1. What is @DataProvider in TestNG?**

* Supplies test data (multiple sets) to a test method.

@DataProvider(name="loginData")

public Object[][] getData() {

return new Object[][] {{"user1","pass1"}, {"user2","pass2"}};

}

@Test(dataProvider="loginData")

public void loginTest(String username, String password) { ... }

**Q2. How is DataProvider different from @Parameters?**

* @Parameters: Static values from XML.
* @DataProvider: Dynamic, multiple sets from code/Excel.

**Q3. Can DataProvider run in parallel?**  
Yes, using parallel=true:

@DataProvider(name="testData", parallel=true)

**9.5 RetryAnalyzer**

**Q1. What is RetryAnalyzer in TestNG?**

* Reruns failed test automatically.

public class RetryAnalyzer implements IRetryAnalyzer {

int count = 0, maxTry = 2;

public boolean retry(ITestResult result) {

if (count < maxTry) {

count++;

return true;

}

return false;

}

}

@Test(retryAnalyzer = RetryAnalyzer.class)

public void flakyTest() { ... }

**Q2. Real-time use case of RetryAnalyzer?**

* Useful for flaky tests due to network, sync issues, environment instability.

**9.6 Grouping Tests**

**Q1. How do you group tests in TestNG?**

@Test(groups="smoke")

public void loginTest() { ... }

@Test(groups={"regression","smoke"})

public void addToCartTest() { ... }

**Q2. How to include/exclude groups in XML?**

<groups>

<run>

<include name="smoke"/>

<exclude name="regression"/>

</run>

</groups>

**9.7 Parallel Execution**

**Q1. How do you handle thread safety in parallel execution?**

* Use ThreadLocal<WebDriver>.

**Q2. Issues with parallel execution in Selenium?**

* Multiple tests opening/closing browsers simultaneously → driver conflicts.
* Solution: Use ThreadLocal, proper teardown, and grid execution.

**9.8 TestNG Listeners**

**Q1. What are TestNG listeners?**

* Interfaces that listen to test events (start, success, failure).

**Q2. Common listeners used in Selenium?**

* ITestListener: Log/report test start/failure.
* ISuiteListener: Before/after suite.

public class ListenerExample implements ITestListener {

public void onTestFailure(ITestResult result) {

System.out.println("FAILED: " + result.getName());

}

}

**Q3. Real-world use of listeners?**

* Capture screenshots on failure.
* Integrate reporting tools (Allure/ExtentReports).

**9.9 TestNG vs JUnit**

**Q1. What’s the difference between TestNG and JUnit?**

| **Feature** | **TestNG ✅** | **JUnit ❌** |
| --- | --- | --- |
| Parallel execution | Yes | No |
| DataProvider | Yes | No |
| Dependency mgmt | Yes | Limited |
| Test Priorities | Yes | No |
| Rich reporting | Yes | Basic |

**Q2. Which one do you prefer for Selenium automation? Why?**

* TestNG → richer features (parallel execution, DataProviders, reporting).

**9.10 Real-Time Scenarios**

**Q1. If your test cases are failing randomly, how would you debug?**

* Check for **sync issues** → add waits.
* Check if tests are dependent → use dependsOnMethods.
* Add **RetryAnalyzer** for flaky ones.

**Q2. How do you design a maintainable TestNG framework?**

* Use testng.xml for execution control.
* Maintain BaseTest for setup/teardown.
* Use DataProvider for test data.
* Implement Listeners for logging/screenshots.
* Use groups for smoke/regression.

**📌 Section 10: Selenium Grid & Parallel Execution**

**10.1 Selenium Grid Basics**

**Q1. What is Selenium Grid? Why do we use it?**  
**Answer:**  
Selenium Grid is a component of Selenium that allows you to run test cases on multiple machines, browsers, and operating systems in parallel.

* It reduces execution time.
* Supports distributed testing.
* Ensures cross-browser and cross-platform compatibility.

**Q2. What are the main components of Selenium Grid 4?**  
**Answer:**  
Selenium Grid 4 has the following components:

1. **Hub** → Central point that manages test distribution.
2. **Nodes** → Machines where actual browsers/tests run.
3. **Event Bus** → Internal communication system (based on Redis).
4. **Session Map** → Keeps track of active sessions.
5. **Session Queue** → Maintains pending test requests.
6. **Distributor** → Assigns tests from queue to available nodes.
7. **Router** → Entry point for all WebDriver client requests.

**Q3. How is Selenium Grid 4 different from Selenium Grid 3?**  
**Answer:**

* **Grid 3:** Used Hub & Node architecture (with standalone Hub/Nodes).
* **Grid 4:** Merged architecture → everything can run in **standalone**, **hub-node**, or **distributed mode**.
* Uses **event-driven communication (Event Bus)**.
* Supports **Docker & Kubernetes natively**.
* Improved **parallel execution & observability (with GraphQL API)**.

**10.2 RemoteWebDriver Usage**

**Q4. What is RemoteWebDriver?**  
**Answer:**

* RemoteWebDriver is a class in Selenium that allows you to execute tests on remote machines (via Selenium Grid).
* It communicates with Grid Hub using JSON Wire Protocol or W3C protocol.

**Q5. How do you instantiate RemoteWebDriver?**  
**Answer:**

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.remote.DesiredCapabilities;

import org.openqa.selenium.remote.RemoteWebDriver;

import java.net.URL;

public class GridExample {

public static void main(String[] args) throws Exception {

DesiredCapabilities caps = new DesiredCapabilities();

caps.setBrowserName("chrome");

WebDriver driver = new RemoteWebDriver(

new URL("http://localhost:4444/wd/hub"), caps);

driver.get("https://www.google.com");

System.out.println(driver.getTitle());

driver.quit();

}

}

**Q6. Why do we need DesiredCapabilities or Options when using RemoteWebDriver?**  
**Answer:**  
They define the environment in which tests should run:

* Browser name & version.
* Platform (Windows/Linux/Mac).
* Additional browser-specific options (headless, incognito, etc.).

**10.3 Parallel Execution**

**Q7. How do you achieve parallel execution in TestNG with Selenium Grid?**  
**Answer:**  
Use **TestNG XML** with parallel attribute.

Example testng.xml:

<suite name="ParallelTests" parallel="tests" thread-count="3">

<test name="ChromeTests">

<parameter name="browser" value="chrome"/>

<classes>

<class name="tests.GoogleTest"/>

</classes>

</test>

<test name="FirefoxTests">

<parameter name="browser" value="firefox"/>

<classes>

<class name="tests.GoogleTest"/>

</classes>

</test>

</suite>

This will run **Chrome and Firefox tests simultaneously**.

**Q8. What are different levels of parallel execution supported by TestNG?**  
**Answer:**

1. parallel="methods" → Each test method runs in parallel.
2. parallel="classes" → Each test class runs in parallel.
3. parallel="tests" → Each <test> tag runs in parallel.

**Q9. How do you handle thread-safety issues in parallel execution?**  
**Answer:**

* Use **ThreadLocal WebDriver** so each test thread has its own driver instance.

Example:

public class DriverManager {

private static ThreadLocal<WebDriver> driver = new ThreadLocal<>();

public static WebDriver getDriver() {

return driver.get();

}

public static void setDriver(WebDriver driverInstance) {

driver.set(driverInstance);

}

public static void quitDriver() {

driver.get().quit();

driver.remove();

}

}

**10.4 Docker with Selenium Grid**

**Q10. Why use Docker with Selenium Grid?**  
**Answer:**

* Avoids manual setup of multiple nodes.
* Scales up/down easily.
* Provides consistency across environments.
* Integrates well with CI/CD pipelines.

**Q11. How do you run Selenium Grid using Docker?**  
**Answer:**

1. Install Docker.
2. Pull official Selenium images:
3. docker pull selenium/hub
4. docker pull selenium/node-chrome
5. docker pull selenium/node-firefox
6. Run Hub:
7. docker run -d -p 4444:4444 --name selenium-hub selenium/hub
8. Run Nodes and connect to Hub:
9. docker run -d --link selenium-hub:hub selenium/node-chrome
10. docker run -d --link selenium-hub:hub selenium/node-firefox

**Q12. What is Selenium Grid with Docker Compose?**  
**Answer:**

* Docker Compose allows you to define Hub and multiple Nodes in a **single docker-compose.yml file** and spin up the entire Grid setup with one command (docker-compose up).
* Makes test infrastructure setup easier in CI/CD pipelines.

**10.5 Advanced Concepts**

**Q13. What are the different Grid modes in Selenium Grid 4?**  
**Answer:**

1. **Standalone** → For local execution (hub + node in one).
2. **Hub-Node** → Classic mode with hub and multiple nodes.
3. **Distributed** → For large-scale execution (multiple services like EventBus, Distributor, Router, etc.).

**Q14. How do you monitor Selenium Grid?**  
**Answer:**

* Selenium Grid 4 provides a **UI Dashboard** at http://localhost:4444/ui.
* It shows active sessions, nodes, queue size, and available slots.

**Q15. How do you run tests in Selenium Grid on cloud platforms?**  
**Answer:**  
Use **RemoteWebDriver** with vendor’s Grid endpoint:

* **BrowserStack**: https://hub-cloud.browserstack.com/wd/hub
* **Sauce Labs**: https://ondemand.saucelabs.com/wd/hub
* Provide username, access key, and desired capabilities.

**📌 Section 11: Selenium with Maven/Gradle & CI/CD**

**11.1 Maven Basics (Dependencies, Surefire Plugin)**

**Q1. What is Maven, and why is it used in Selenium projects?**  
**A:**

* Maven is a **build automation and project management tool** based on POM (Project Object Model).
* It simplifies handling **dependencies** (like Selenium, TestNG, Apache POI), build lifecycle, and reporting.
* Instead of downloading JAR files manually, we just add them in pom.xml, and Maven fetches them from the repository.

**Q2. What is the role of pom.xml in Maven?**  
**A:**

* It defines the project’s configuration:
  + Dependencies (libraries like Selenium, TestNG).
  + Plugins (Surefire, Compiler, etc.).
  + Build and reporting settings.

Example snippet:

<dependencies>

<dependency>

<groupId>org.seleniumhq.selenium</groupId>

<artifactId>selenium-java</artifactId>

<version>4.20.0</version>

</dependency>

</dependencies>

**Q3. What is the Maven Surefire Plugin, and why is it important in Selenium testing?**  
**A:**

* Surefire Plugin is used to **execute unit tests** and **integration tests**.
* It integrates with frameworks like TestNG or JUnit to run Selenium test classes.
* It also generates test reports.

Example config:

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>3.0.0</version>

<configuration>

<suiteXmlFiles>

<suiteXmlFile>testng.xml</suiteXmlFile>

</suiteXmlFiles>

</configuration>

</plugin>

**Q4. What is the difference between Maven and Gradle?**  
**A:**

* **Maven**: XML-based, rigid lifecycle, widely used.
* **Gradle**: Groovy/Kotlin-based DSL, faster build (incremental), more flexible.
* In Selenium projects, Maven is more common, but Gradle is rising in popularity.

**11.2 Integration with Jenkins (Running Selenium Tests from CI Pipeline)**

**Q5. Why do we integrate Selenium tests with Jenkins?**  
**A:**

* Jenkins enables **Continuous Integration (CI)**:
  + Automatically trigger Selenium tests after code commit.
  + Schedule builds (nightly regression runs).
  + Provide detailed HTML/XML reports.

**Q6. How do you run Selenium tests in Jenkins?**  
**A:**  
Steps:

1. Install Jenkins and required plugins (Maven, TestNG Results).
2. Create a new Jenkins job (freestyle or pipeline).
3. Pull code from **GitHub/GitLab/Bitbucket**.
4. Configure **Maven build step** with clean test.
5. Publish **TestNG reports**.

**Q7. What is a Jenkins Pipeline, and how do you use it for Selenium?**  
**A:**

* Jenkins Pipeline (Jenkinsfile) defines CI/CD as **code**.
* Example:

pipeline {

agent any

stages {

stage('Checkout') {

steps {

git 'https://github.com/username/selenium-framework.git'

}

}

stage('Build') {

steps {

sh 'mvn clean install'

}

}

stage('Test') {

steps {

sh 'mvn test'

}

}

}

post {

always {

junit '\*\*/target/surefire-reports/\*.xml'

}

}

}

**Q8. How do you schedule Selenium test execution in Jenkins?**  
**A:**

* Use **Build Triggers → Build periodically**.
* Example: H 2 \* \* \* → runs tests daily at 2 AM.

**Q9. What challenges do you face when running Selenium tests in Jenkins?**  
**A:**

* Browser setup on Jenkins server.
* Headless execution issues.
* Handling parallel execution with Grid/Docker.
* Managing large reports and logs.

**11.3 Running Selenium Tests in GitHub Actions / GitLab CI**

**Q10. What are GitHub Actions, and how do they help in Selenium automation?**  
**A:**

* GitHub Actions is a **CI/CD platform** built into GitHub.
* You can trigger Selenium tests on push/PR events.
* Provides free runners (Ubuntu, Windows, Mac).

**Q11. How do you configure Selenium tests in GitHub Actions?**  
**A:**

* Create .github/workflows/selenium-tests.yml:

name: Selenium Tests

on: [push, pull\_request]

jobs:

test:

runs-on: ubuntu-latest

steps:

- name: Checkout Code

uses: actions/checkout@v2

- name: Set up JDK

uses: actions/setup-java@v2

with:

java-version: '17'

- name: Run Tests

run: mvn clean test

**Q12. Can Selenium tests run in GitHub Actions headlessly?**  
**A:**

* Yes, by using Chrome headless mode.
* Example in DriverManager:

ChromeOptions options = new ChromeOptions();

options.addArguments("--headless", "--no-sandbox");

WebDriver driver = new ChromeDriver(options);

**Q13. How is GitLab CI different from GitHub Actions for Selenium?**  
**A:**

* Both provide CI/CD pipelines.
* GitHub Actions is **native to GitHub**.
* GitLab CI uses .gitlab-ci.yml.

Example GitLab config:

stages:

- test

selenium\_test:

image: maven:3.8.5-openjdk-17

stage: test

script:

- mvn clean test

**Q14. What are the benefits of using CI/CD for Selenium tests?**  
**A:**

* Faster feedback loop.
* Avoids manual execution.
* Detects bugs early.
* Enables **parallel execution** and **cross-browser testing**.
* Ensures **regression safety** before production release.

**📌 Section 12: Selenium & APIs / Databases**

**🔹 12.1 Why UI automation + API automation together?**

**Q1. Why do we combine UI and API automation in testing?**  
👉 To validate the system end-to-end. UI only tests frontend behavior, but APIs and DB ensure backend consistency.  
Example: After submitting a form in UI, check DB entry and API response.

**Q2. What are benefits of integrating UI + API automation?**

* Faster root cause analysis (UI failure vs backend failure).
* Reduces flaky UI test dependency.
* Ensures frontend and backend are in sync.
* Improves test coverage (validation at multiple layers).

**Q3. Can you give a real-world scenario?**

* Flight booking on MakeMyTrip:
  + **UI:** Enter source/destination and click search.
  + **API:** Validate /searchFlights API returns same data shown on UI.
  + **DB:** Verify booked ticket stored correctly in booking table.

**🔹 12.2 JDBC Integration for DB Validation**

**Q4. How can Selenium tests interact with databases?**  
👉 Using **JDBC (Java Database Connectivity)** API.

**Steps:**

1. Load the JDBC driver.
2. Establish connection (DriverManager.getConnection).
3. Create Statement or PreparedStatement.
4. Execute query (executeQuery / executeUpdate).
5. Validate results.

**Q5. Example: Validate login data from DB after UI action**

import java.sql.\*;

public class DBValidationExample {

public static void main(String[] args) throws Exception {

// 1. Connect to DB

Connection con = DriverManager.getConnection(

"jdbc:mysql://localhost:3306/testdb", "root", "password");

// 2. Create statement

Statement stmt = con.createStatement();

// 3. Execute query

ResultSet rs = stmt.executeQuery("SELECT \* FROM users WHERE username='testUser'");

// 4. Validate result

while (rs.next()) {

String dbPassword = rs.getString("password");

System.out.println("Password in DB: " + dbPassword);

}

con.close();

}

}

**Q6. Which databases can we connect with JDBC in Selenium tests?**

* MySQL
* Oracle
* PostgreSQL
* SQL Server
* SQLite

**Q7. What are challenges with DB validation in Selenium tests?**

* DB schema changes may break tests.
* Requires DB access (sometimes restricted in real projects).
* Test data setup and cleanup is critical.

**🔹 12.3 Hybrid Tests: Validate UI result vs DB vs API response**

**Q8. How can we design hybrid tests?**  
👉 Use Selenium + JDBC + API libraries (like RestAssured) together.

**Example Flow:**

* Perform action in UI (Selenium).
* Fetch backend response via API (RestAssured).
* Fetch DB record (JDBC).
* Compare all three values.

**Q9. Example: Place an order and validate across UI, API, DB**

// UI Action: Place order using Selenium

driver.findElement(By.id("placeOrderBtn")).click();

// API Validation

Response apiResponse = given().get("https://api.shop.com/orders/123");

String apiStatus = apiResponse.jsonPath().getString("status");

// DB Validation

Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/shopdb","root","password");

Statement stmt = con.createStatement();

ResultSet rs = stmt.executeQuery("SELECT status FROM orders WHERE order\_id=123");

rs.next();

String dbStatus = rs.getString("status");

// Compare results

Assert.assertEquals(apiStatus, "CONFIRMED");

Assert.assertEquals(dbStatus, "CONFIRMED");

**Q10. What’s the advantage of hybrid validation?**

* Ensures consistency between **UI display, API response, and DB storage**.
* Detects mismatches quickly (e.g., API shows “CONFIRMED” but UI shows “FAILED”).
* Builds trust in system reliability.

**Q11. Where do we use this in real projects?**

* Banking apps → UI balance, API balance, DB balance must match.
* E-commerce → Order ID in UI vs API vs DB.
* Travel booking → Ticket confirmation in all three layers.

**📌 Section 13: Debugging & Best Practices**

**🔹 13.1 Common Selenium Exceptions**

**Q1. What are common Selenium exceptions you’ve faced?**  
👉 Some frequently encountered exceptions:

* **NoSuchElementException** – Locator not found on the page.
* **TimeoutException** – Explicit wait exceeded before element appeared.
* **ElementClickInterceptedException** – Another element (popup, overlay) blocking the click.
* **StaleElementReferenceException** – DOM updated, reference to old element lost.
* **ElementNotInteractableException** – Element found but not ready for interaction.
* **SessionNotCreatedException** – Mismatch between driver & browser version.

**Q2. How do you handle NoSuchElementException?**

* Verify locator correctness (use unique locators).
* Add explicit waits to handle late loading.
* Debug with driver.getPageSource() to confirm presence of element.

**Q3. How do you handle StaleElementReferenceException?**

* Re-locate element before interacting.
* Use retry logic.  
  Example:

public WebElement safeFindElement(By locator) {

for (int i = 0; i < 3; i++) {

try {

return driver.findElement(locator);

} catch (StaleElementReferenceException e) {

System.out.println("Retrying...");

}

}

throw new RuntimeException("Element not stable: " + locator);

}

**🔹 13.2 Handling Synchronization Issues**

**Q4. What causes synchronization issues in Selenium?**

* Page loads slower than expected.
* AJAX / dynamic elements appear late.
* JavaScript-based UI updates not instant.

**Q5. What are the solutions?**

* **Implicit Wait:** Sets default wait for element search.
* **Explicit Wait:** Wait until condition (visibility, clickable).
* **Fluent Wait:** Polls at intervals with custom timeout.

Example (Explicit Wait):

WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));

WebElement button = wait.until(ExpectedConditions.elementToBeClickable(By.id("submitBtn")));

button.click();

**🔹 13.3 Writing Reusable Utilities**

**Q6. Why write reusable utilities?**

* Avoid code duplication.
* Improve readability & maintainability.
* Abstract common waits, clicks, screenshots.

**Q7. Example utility class (WaitHelper):**

public class WaitHelper {

WebDriver driver;

WebDriverWait wait;

public WaitHelper(WebDriver driver, int timeout) {

this.driver = driver;

this.wait = new WebDriverWait(driver, Duration.ofSeconds(timeout));

}

public WebElement waitForElementVisible(By locator) {

return wait.until(ExpectedConditions.visibilityOfElementLocated(locator));

}

public void clickElement(By locator) {

waitForElementVisible(locator).click();

}

}

**Q8. Example: How do you use this utility in tests?**

WaitHelper helper = new WaitHelper(driver, 10);

helper.clickElement(By.id("loginBtn"));

**🔹 13.4 Debugging Failed Tests (Logs, Screenshots)**

**Q9. How do you debug failed Selenium tests?**

1. Capture **logs** (console + framework logs).
2. Take **screenshots** at failure points.
3. Use **test reports** (ExtentReports, Allure).
4. Re-run test with debug mode in IDE.
5. Add temporary Thread.sleep(2000) to reproduce issue (not as solution).

**Q10. Example: Capture screenshot on failure (TestNG ITestListener):**

public class ScreenshotListener implements ITestListener {

@Override

public void onTestFailure(ITestResult result) {

TakesScreenshot ts = (TakesScreenshot) result.getTestContext()

.getAttribute("WebDriver");

File src = ts.getScreenshotAs(OutputType.FILE);

try {

FileUtils.copyFile(src, new File("screenshots/" + result.getName() + ".png"));

} catch (IOException e) {

e.printStackTrace();

}

}

}

**Q11. How do logs help in debugging?**

* Logs provide stack trace of exception.
* Framework logs highlight which step failed.
* Helps distinguish environment issue vs script issue.

**🔹 13.5 Best Practices in Debugging & Writing Selenium Tests**

**Q12. What are Selenium best practices you follow?**

* Use **Page Object Model (POM)** for maintainability.
* Use **descriptive locators** (avoid fragile XPath).
* Add **explicit waits**, not hard sleeps.
* Always handle **synchronization**.
* Keep **test data externalized** (Excel, JSON, DB).
* Capture **screenshots & logs** for failures.
* Use **retry logic** for flaky tests.
* Integrate with **CI/CD pipeline** for early detection.

**📌 Section 14: Latest & Advanced Topics**

**🔹 14.1 Selenium with Docker**

**Q1. Why use Docker with Selenium?**

* Provides **isolated environments** for running tests.
* No dependency conflicts (driver/browser versions).
* Scales tests easily (multiple containers = parallel execution).
* Works well with **CI/CD pipelines**.

**Q2. What is Selenium Grid with Docker?**

* Instead of installing Selenium Grid manually, you run **containers**:
  + selenium/hub → central hub.
  + selenium/node-chrome / selenium/node-firefox → browser nodes.

**Q3. Example: Run Selenium Grid using Docker Compose:**

version: "3"

services:

hub:

image: selenium/hub:4.0.0

ports:

- "4444:4444"

chrome:

image: selenium/node-chrome:4.0.0

depends\_on:

- hub

environment:

- SE\_EVENT\_BUS\_HOST=hub

**Q4. How do you connect tests to Docker Grid?**

WebDriver driver = new RemoteWebDriver(

new URL("http://localhost:4444/wd/hub"),

new ChromeOptions()

);

**🔹 14.2 Selenium + BDD (Cucumber + Gherkin)**

**Q5. What is BDD in automation testing?**

* **BDD (Behavior Driven Development)** bridges communication between business & technical teams.
* Uses **Gherkin language** (Given-When-Then) for writing test scenarios.

**Q6. Example: Cucumber Feature File (Login.feature):**

Feature: Login functionality

Scenario: Successful login with valid credentials

Given User is on login page

When User enters valid username and password

And clicks on login button

Then User should be redirected to homepage

**Q7. Example: Step Definition in Java:**

@Given("User is on login page")

public void userOnLoginPage() {

driver.get("https://app.example.com/login");

}

@When("User enters valid username and password")

public void enterCredentials() {

driver.findElement(By.id("username")).sendKeys("testuser");

driver.findElement(By.id("password")).sendKeys("password123");

}

**Q8. Why integrate Selenium with Cucumber?**

* **Readable test cases** for non-technical stakeholders.
* Keeps scenarios **business-focused** while code handles automation.

**🔹 14.3 Selenium Alternatives: Playwright & Cypress**

**Q9. What is Playwright?**

* Microsoft’s browser automation tool.
* Supports **multiple browsers** (Chromium, Firefox, WebKit).
* Has **auto-waiting** (less flaky than Selenium).
* Supports **modern web apps (React, Angular)** better.

**Q10. What is Cypress?**

* End-to-end JS testing framework.
* Runs inside the browser (fast).
* Provides **time-travel debugging** (can see what happened at each step).
* Great for **front-end developers**, but **limited to Chrome-family browsers**.

**Q11. When is Selenium still better?**

* When you need **multi-language support** (Java, Python, C#, etc.).
* When you need **cross-browser, cross-device** automation.
* For **enterprise-level test suites** with large CI/CD setups.
* For **integration with legacy systems** (Cypress/Playwright are newer).

**🔹 14.4 Cloud-based Selenium Execution (BrowserStack, SauceLabs)**

**Q12. Why use cloud platforms for Selenium tests?**

* Avoids maintaining **local infrastructure**.
* Provides **real devices & browsers** on-demand.
* Supports **parallel test execution** at scale.
* Useful for **cross-browser testing** (Safari on Mac, IE on Windows).

**Q13. Example: Running Selenium test on BrowserStack:**

DesiredCapabilities caps = new DesiredCapabilities();

caps.setCapability("browserName", "Chrome");

caps.setCapability("browserVersion", "latest");

caps.setCapability("os", "Windows");

caps.setCapability("osVersion", "11");

WebDriver driver = new RemoteWebDriver(

new URL("https://USERNAME:ACCESS\_KEY@hub-cloud.browserstack.com/wd/hub"),

caps

);

driver.get("https://www.google.com");

**Q14. Difference between BrowserStack & SauceLabs?**

* **BrowserStack** → More focus on **real devices** (mobile testing).
* **SauceLabs** → Stronger in **scalability** & **enterprise CI/CD integrations**.