1. **Components of Selenium**:
   * Selenium comprises four major components:
     + **Selenium IDE (Integrated Development Environment)**: A complete IDE for Selenium tests. It’s implemented as a Firefox Add-On and Chrome Extension. You can record, edit, and debug functional tests using Selenium IDE. It was previously known as Selenium Recorder.
     + **Selenium RC (Remote Control)**: A server (written in Java) that accepts commands via HTTP. It allows writing automated tests for web applications in various programming languages. Selenium provides client drivers for languages like PHP, Python, Ruby, .NET, Perl, and Java.
     + **Selenium Grid**: Used with WebDriver to execute tests on remote systems.
2. **Selenium IDE**:
   * An essential tool in the Selenium Suite.
   * Allows recording, editing, and debugging of functional tests.
   * Implemented as a Firefox Add-On and Chrome Extension.
   * Previously known as Selenium Recorder.
   * Scripts are written in **Selenese**, a special test scripting language for Selenium.
3. **Selenium WebDriver**:
   * The heart of Selenium automation.`
   * Accepts commands and sends them to a browser.
   * Uses browser-specific drivers (e.g., ChromeDriver, GeckoDriver) to control browsers.
   * No need for a special server; WebDriver directly starts a browser instance.
   * Example code snippet to set up WebDriver in Java:
4. **Element Locators in Selenium**:
   * Common locators include:
     + **ID**: Find elements by their unique ID attribute.
     + **Name**: Locate elements by their name attribute.
     + **XPath**: Powerful but slower; uses path expressions to navigate XML documents.
     + **CSS Selector**: Similar to XPath but more concise.
     + **Class Name**: Locate elements by their class attribute.
     + **Link Text**: Find links by their visible text.
     + **Partial Link Text**: Locate links by partial visible text.
5. **Maximizing Window with Selenium WebDriver (Java)**:

**Java**

driver. manage(). window().maximize();

1. **Clearing Content of a Text Box**:

**Java**

WebElement textBox = driver.findElement(By.id("textbox\_id"));

textBox.clear();

1. **Executing JavaScript in Selenium WebDriver (Java)**:

**Java**

// Execute a JavaScript function

JavascriptExecutor jsExecutor = (JavascriptExecutor) driver;

jsExecutor.executeScript("alert('Hello, Selenium!')");

1. **Declare the drop-down element as an instance of the**Select**class**:

Select dropdown =

new Select(driver.findElement(By.id("your\_dropdown\_id")));

* + **Select the desired option** using one of the following methods:
    - selectByVisibleText("Option Text"): Selects an option by its visible text.
    - selectByValue("Option Value"): Selects an option by its value attribute.
    - selectByIndex(index): Selects an option by its index (0-based).

1. **Explain about the**Select**class and its methods:** The Select class in Selenium provides methods to interact with dropdowns. Here are some commonly used methods:
   * selectByVisibleText(String arg0): Selects an option by its visible text.
   * selectByIndex(int arg0): Selects an option by its index.
   * selectByValue(String arg0): Selects an option by its value attribute.
   * getOptions(): Returns a list of all options within the dropdown.
   * deselectAll(): Clears all selected entries (applicable for multi-select dropdowns).
2. **How to automate radio buttons in Selenium WebDriver?**

To automate radio buttons, locate the radio button element and use the click() method to select it.

1. **How to submit a form using Selenium WebDriver?** Locate the form element and use the submit() method to submit the form.

If the button is outside the form, click() must be used but if the button is inside the Form, either click() or submit();

1. **Difference between**findElement**and**findElements**methods.**

* findElement: Locates a single web element based on a unique locator strategy. Returns a WebElement. Returns the first matching element or throws an exception if not found.
* findElements : Locates multiple web elements based on the same criteria. Returns a list of WebElement objects. Returns a list of all matching elements (empty list if none found).

1. **How to count the total number of links on a page using Selenium WebDriver?**

Use the findElements(By.tagName("a")) to get all anchor tags (links) and then count the size of the list.

List<WebElements> element = driver.findElements(By.tagName(“a”));

1. **How to capture the page title using Selenium WebDriver?**

Use driver.getTitle() to retrieve the page title.

1. **How to store the current page URL using Selenium WebDriver?**

Use driver.getCurrentUrl() to get the current URL.

1. **How to simulate browser back and forward navigation?**
   * To go back: driver.navigate().back()
   * To go forward: driver.navigate().forward();
2. **Difference between single and double slash in XPath:**
   * Single slash (/): Selects the immediate child element.
   * Double slash (//): Selects any descendant element (not just immediate children).
3. **How to handle dynamic XPath?**
   * Use relative XPath with attributes that remain constant.
   * Use functions like contains(), starts-with(), or text() to handle dynamic values.
4. **Choosing the Right Locator**: When selecting a locator, consider the following options:
   * **ID**: Use when the element has a unique ID attribute.
   * **Name**: Suitable for elements with a unique name attribute.
   * **XPath**: Powerful and flexible, but can be slower. Useful for complex scenarios.
   * **CSS Selector**: Efficient and concise. Good for styling-based locators.
5. **Asserting Text on a Webpage**: To assert text on a webpage, use assertions from testing frameworks (e.g., TestNG, JUnit). For example:

String expectedText = "Welcome to Selenium!";

String actualText= driver.findElement(By.id("welcomeMessage")).getText();

Assert.assertEquals(actualText, expectedText);

Assertion(actualtext,expectedtext);

1. **Getting Element Attributes**:

You can retrieve an attribute value using getAttribute():

String hrefValue = driver.findElement(By.linkText("Learn More")).getAttribute("href");

1. **Double Clicking an Element**:

Use the Actions class to perform double-click actions:

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

Actions actions = new Actions(driver);

actions.doubleClick(element).perform();

1. **Drag and Drop**: Drag an element to a target location using dragAndDrop():

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

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

Actions actions = new Actions(driver);

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

1. **Synchronizing Application Window and Selenium**:
   1. Use implicit waits (driver.manage().timeouts().implicitlyWait()) to wait for elements to appear.
   2. Use explicit waits (WebDriverWait) for specific conditions (e.g., element visibility, presence).

26. **Implicit vs. Explicit Wait**:

* + **Implicit Wait**: Set once for the entire session. Waits for a specified time before throwing an exception if an element is not found.
  + **Explicit Wait**: Applied to specific elements. Waits for a certain condition to be met (e.g., element visibility) within a specified timeout.

1. **Handling Multiple Windows**:
   1. Use getWindowHandles() to get handles of all open windows.
   2. Switch between windows using driver.switchTo().window(windowHandle).
2. **Working with Pop-Up Windows**:
   1. Identify pop-up elements using locators.
   2. Switch to the pop-up window using driver.switchTo().window(windowHandle).
3. **Mouse Hover on an Element**:

Use Actions class for mouse interactions

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

Actions actions = new Actions(driver);

actions.moveToElement(element).perform();

1. **Capturing Screenshots**:

To take a screenshot:

TakesScreenshot ts = (TakesScreenshot) driver ;

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

File trg = new File(System.getProperty("user.dir")+"\\ScreenShot\\Photo1.png");

FileUtils.copyFile(src,trg);

1. **Getting Data from an Excel Sheet**:

To read data from an Excel sheet, you can use libraries like Apache POI. Here’s a basic example:

// Add Apache POI dependencies to your project

FileInputStream file = new FileInputStream("path/to/your/excel.xlsx");

XSSFWorkbook workbook = new XSSFWorkbook(file);

XSSFSheet sheet = workbook.getSheet("Sheet1");

// Read data from cells

String value = sheet.getRow(0).getCell(0).getStringCellValue();

1. **Reading Data from a Properties File**:

**Create a Properties File**:

First, create a .properties file (e.g., config.properties) that contains key-value pairs. Each line in the file represents a property, with the format: key=value.

**Load the Properties File**:

Use the FileReader class to read the properties file. Create an instance of Properties and load the file using the load() method.

**Access Property Values**:

Retrieve property values using the getProperty(key) method, where key corresponds to the property name

1. **Types of Automation Frameworks**:

There are several types of **test automation frameworks** commonly used in software testing. These frameworks provide structure, guidelines, and best practices for organizing and maintaining automated test scripts. Let’s explore some of most popular ones:

1. **Linear Scripting Framework**:
   * Also known as a **record-and-playback framework**.
   * Testers record their interactions with the application, and the tool generates corresponding scripts.
   * Simple to create but lacks flexibility and scalability.
2. **Modular Testing Framework**:
   * Breaks down the application into smaller modules or components.
   * Each module has its own test script.
   * Promotes reusability and maintainability.
   * Allows parallel development by different team members.
3. **Library Architecture Testing Framework**:
   * Organizes reusable functions (libraries) separately from test scripts.
   * Test scripts call these functions when needed.
   * Enhances code reusability and reduces redundancy.
4. **Data-driven Testing Framework**:
   * Separates test data from test logic.
   * Test scripts read data from external sources (e.g., Excel, CSV, databases).
   * Useful for testing multiple scenarios with different input data.
5. **Keyword Driven Testing Framework**:
   * Uses keywords or action words to define test steps.
   * Test scripts are written in a tabular format.
   * Keywords represent actions (e.g., “click,” “verify,” “input”).
   * Provides high-level abstraction and easy maintenance.
6. **Hybrid Testing Framework**:
   * Combines elements of multiple frameworks (e.g., data-driven and keyword-driven).
   * Offers flexibility to choose the best approach for different scenarios.
   * Commonly used in complex applications.
7. **Behavior Driven Development (BDD) Testing Framework**:
   * Focuses on collaboration between developers, testers, and business stakeholders.
   * Uses natural language (Gherkin syntax) to write test scenarios.
   * Tools like **Cucumber** or **SpecFlow** enable BDD.
8. **Selenium Grid**:

**Selenium Grid** is a powerful feature in **Selenium** that allows you to distribute and execute test cases across different machines and platforms. Let’s dive into how it works:

1. **Architecture of Selenium Grid**:
   * Selenium Grid consists of two main components:
     + **Hub**: The central point that receives test requests and distributes them to registered nodes.
     + **Node**: Instances of Selenium running on different machines (physical or virtual) that execute the test cases.
   * The hub routes test commands (in JSON format) to multiple registered nodes.
   * Nodes can be launched on various platforms and browsers.
2. **How It Works**:
   * You set up a hub (usually on your local machine) and register nodes (remote machines or VMs) with it.
   * The hub acts as a smart proxy server, directing test commands to the appropriate nodes.
   * When you trigger a test case, the hub identifies an available node based on desired capabilities (browser, OS, etc.).
   * The node launches the specified browser and executes the test case.
   * Test results are sent back to the hub, which aggregates them for reporting.
3. **Benefits of Selenium Grid**:
   * **Platform Independence**: Run test cases on different platforms (Windows, macOS, Linux) simultaneously.
   * **Parallel Execution**: Execute multiple test cases in parallel, saving time.
   * **Scalability**: Easily add more nodes to handle increased test loads.
   * **Distributed Testing**: Distribute testing across multiple machines for comprehensive coverage.
4. **Setting Up Selenium Grid**:
   * Download the **Selenium Standalone Server** (available as a single JAR file).
   * Store the JAR file on any drive.
   * Open the command prompt (cmd).
   * Register the hub by running:
   * java -jar selenium-server-standalone-3.8.1.jar -role hub
5. **Default Port ID in Selenium Grid**:

The default port for the Selenium Grid hub is **4444**.

1. **Page Object Model (POM)**:
   1. Design pattern to create an object repository for web elements.
   2. Advantages:
      1. **Modularity**: Separates test logic from page structure.
      2. **Reusability**: Page objects can be reused across tests.
      3. **Maintainability**: Changes to UI are localized to page objects.
2. **Page Factory**:
   1. An extension of POM.
   2. Uses annotations (@FindBy) to locate elements.
   3. Initializes elements lazily (when accessed) for better performance.
3. ***Page Factory:***

Page Factory is an advanced design pattern in Selenium that streamlines the creation of Page Objects. It enhances the readability and maintainability of test scripts by minimizing repetitive code.

It separates the test objects (web elements) from the test scripts and initializes these elements using annotations.

In essence, Page Factory optimizes the use of Page Objects in automation testing.

1. ***Advantages of Using TestNG***:

TestNG offers several advantages over other testing frameworks:

Parallel Execution: TestNG allows parallel execution of test methods, which significantly reduces test suite execution time.

Flexible Annotations: Annotations like @BeforeMethod, @AfterMethod, @BeforeClass, and @AfterClass provide fine-grained control over test execution.

Grouping and Prioritization: You can group test methods and set their priorities using annotations.

Data-Driven Testing: TestNG supports data-driven testing through data providers.

Listeners and Reporting: TestNG provides built-in listeners for custom reporting and handling test events.

Easy Configuration: TestNG configuration via XML files (testng.xml) simplifies test suite setup.

1. ***Basic Annotations in TestNG:***

Some fundamental annotations used to run TestNG tests in Selenium include:

@Test: Marks a method as a test case.

@BeforeMethod: Executes before each test method.

@AfterMethod: Executes after each test method.

@BeforeClass: Runs once before all test methods in a class.

@AfterClass: Runs once after all test methods in a class.

@DataProvider: Supplies test data to test methods.

@Parameters: Passes parameters from testng.xml to test methods.

1. ***Difference Between @BeforeMethod and @BeforeClass:***

@BeforeMethod: Executes before each test method. Useful for setting up preconditions specific to individual test cases.

@BeforeClass: Runs once before all test methods in a class. Useful for setting up common preconditions for all test cases in the class.

1. ***Running Test Cases with Dependencies in Selenium using TestNG:***

Use the @dependsOnMethods attribute in the @Test annotation to define dependencies between test methods.

Ensure that the dependent method(s) execute successfully before the current method runs.

1. ***Running Test Cases in Groups in Selenium using TestNG:***

Group test methods using the groups attribute in the @Test annotation.

Define groups in the testng.xml file and include/exclude them as needed.

1. ***Default Priority of a Test Method in TestNG:***

The default priority of a test method in TestNG is 0. You can explicitly set priorities using the priority attribute.

1. ***Importance of testng.xml File:***

The testng.xml file serves as the configuration file for TestNG.

It defines test suites, test classes, test methods, parameters, listeners, and other settings.

Allows customization of test execution, parallelism, and data-driven testing.

Passing Parameters from testng.xml into Test Cases:

Define parameters in the testng.xml file using <parameter> tags.

Access these parameters in test methods using @Parameters annotation.

1. ***//Use of @Listener Annotation in TestNG://***

The @Listener annotation allows you to attach custom listeners to TestNG.

Listeners handle events like test start, test success/failure, and suite completion.

Useful for custom reporting, logging, and other test-related actions.

1. *Behavior-Driven Development (BDD):*
   * BDD is an **Agile software development process** derived from Test-Driven Development (TDD).
   * It focuses on describing the behavior of a system from the user’s perspective.
   * BDD encourages collaboration among developers, testers, business analysts, and product owners.
   * Key principles:
     + **Conversation**: All parties (customer, developer, tester, stakeholder) engage in a collaborative conversation to clarify system behavior.
     + **Concrete Examples**: Use simple language and concrete examples to illustrate system behavior.
     + **Automated Testing**: BDD emphasizes behavior over code implementation, making it ideal for automated testing.
   * BDD is facilitated through **natural language** (Gherkin) to express system behavior and expected outcomes.
2. *Cucumber:*
   * Cucumber is a BDD tool that uses **Gherkin language** (plain English text) to define features and scenarios.
   * **Advantages** of using Cucumber:
     + **Open Source**: Cucumber is free and doesn’t require licensing.
     + **IDE Integration**: Easily configure Cucumber with IDEs like Eclipse.
     + **Communication Bridge**: Bridges gaps among developers, testers, business analysts, and stakeholders.
     + **Executable Specifications**: Feature files serve as live documents and executable tests.
     + **Localization**: Supports multiple spoken languages.
     + **Structured Testing**: Defines requirements in a standard format (Gherkin).
3. *Main Files Required to Run a Cucumber Test Scenario:*

Every Cucumber project should have:

**Feature File**: Contains high-level descriptions of test scenarios in Gherkin format.

**Step Definition File**: Maps Gherkin steps to actual code (step implementations).

1. *Feature File:*
   * Describes a software feature and groups related scenarios.
   * Components:
     + **Feature**: High-level description of the feature.
     + **Scenario**: Describes steps and expected outcomes for a specific test case.
     + **Scenario Outline**: Reusable scenario with data-driven examples.
     + Keywords: Given, When, Then, And, But.
2. *Keywords Used in Cucumber for Writing a Scenario:*
   * **Primary Keywords**:
     + **Feature**: Describes the current test script.
     + **Scenario**: Represents a test case.
     + **Given, When, Then, And, But**: Steps within a scenario.
   * **Secondary Keywords**:
     + **Doc Strings (“""”)**: Used for large text blocks.
     + **Data Tables (“|”)**: Represent tabular data.
     + **Tags (“@”)**: Add metadata to scenarios.
     + **Comments (“#”)**: Provide additional context.
3. *Scenario Outline:*
   * A **Scenario Outline** in Cucumber is a powerful construct that allows you to write a single scenario with placeholders (variables) for input data. These placeholders are defined using the <variable> syntax within the scenario steps.
   * The purpose of a Scenario Outline is to run the same scenario with **multiple sets of inputs and expected outputs**. Instead of writing separate scenarios for each combination of input data, you can use a single outline and provide different values using an **Examples** table.

Here’s how it works:

* + Feature: Calculator
  + As a user
  + I want to use a calculator to add numbers
  + So that I don't need to add myself
  + Scenario Outline: Add two numbers
  + Given I have a calculator
  + When I add <number1> and <number2>
  + Then the result should be <expectedResult>
  + Examples:
  + | number1 | number2 | expectedResult |
  + | 2 | 3 | 5 |
  + | 10 | 15 | 25 |

In this example, the same steps are executed with different input values, making it more concise and maintainable.

1. *Step Definition File:*
   * The **Step Definition file** (also known as the **glue code**) is where you implement the actual code that corresponds to the steps defined in your feature files.
   * Each step in your feature file (e.g., Given, When, Then) needs a corresponding method in the Step Definition file.
   * For example, if your feature file has the step: Given I have a calculator, your Step Definition file should contain a method that initializes the calculator.
   * The Step Definition file bridges the gap between the plain text steps in the feature file and the actual automation code (e.g., Java, Ruby, etc.).
2. *Background Keyword:*
   * The **Background** keyword in Cucumber allows you to define a set of common steps that are executed before every scenario in a feature file.
   * It’s useful for setting up preconditions or shared context for multiple scenarios.
   * For instance, if your feature file involves logging in before performing various actions, you can define the login steps in the Background section.
   * The Background steps are executed before each scenario, ensuring consistency across scenarios.
3. *Hooks in Cucumber:*
   * **Hooks** in Cucumber are special methods that run at specific points during the test execution lifecycle.
   * They allow you to perform setup (e.g., opening a browser), teardown (e.g., closing resources), or other custom actions.
   * Common hook points include **Before** (runs before each scenario), **After** (runs after each scenario), and others.
   * Hooks are defined in the Step Definition file and can be used to manage state, set up test data, or perform cleanup.
4. *Running Multiple Scenarios:*
   * To run multiple scenarios in Cucumber, you typically execute the entire feature file.
   * You can run all scenarios in a feature file using the command-line interface or by configuring your test runner (e.g., JUnit, TestNG).
   * For example, if you have a feature file named calculator.feature, you can run it using:
   * cucumber features/calculator.feature
   * The test runner will execute all scenarios defined in that feature file.
5. *What Is Maven?*
   * **Maven** is based on the concept of a **Project Object Model (POM)**. It simplifies the build process, similar to **ANT**, but with more advanced features.
   * Key features of Maven:
     + **POM Files**: These XML files contain project-related information such as dependencies, source directories, plugins, and goals. When you execute Maven commands, it reads the pom.xml file for configuration.
     + **Dependencies and Repositories**: Maven manages external Java libraries (dependencies) required for your project. It downloads them from central repositories and stores them in your local repository.
     + **Build Life Cycles, Phases, and Goals**: A build life cycle consists of phases, and each phase contains goals. Maven commands can target specific lifecycles, phases, or goals.
     + **Build Profiles**: Profiles allow you to build your project with different configurations (e.g., local, development, test).
     + **Build Plugins**: Plugins perform specific tasks during the build process.
   * In summary, Maven simplifies day-to-day work for Java developers and enhances project comprehension.
6. *Why Should We Use Maven?*
   * **Project Management**: Maven handles processes like releases, distribution, reporting, builds, documentation, and source code management (SCMs).
   * **Dependency Management**: It automates the compilation of source code and manages external libraries.
   * **Consistent Builds**: Maven ensures consistent builds across different environments.
   * **Modularity**: Projects can be modularized, making it easier to manage complex applications.
   * **Centralized Repositories**: Maven integrates with remote repositories for efficient dependency resolution.
   * **Code Quality**: Maven supports code quality checks and testing.
   * **Transparent Migration**: It simplifies project migration and upgrades.
   * **Comprehensive Tool**: Maven covers various aspects of software project management.

*Maven POM File (Project Object Model)****:***

* The **POM** is an **XML file** named pom.xml that resides in the base directory of a Maven project.
* It serves as the **central configuration file** for your project, containing essential information such as:
  + **Project coordinates**: Group ID, Artifact ID, and Version.
  + **Project dependencies**: Libraries and frameworks required by your project.
  + **Build settings**: Compilation, packaging, and plugins.
  + **Project description**, **licenses**, and other metadata.
* The POM defines the **project’s identity, structure, and lifecycle**.

***Maven Repositories:***

* Repositories hold **build artifacts** (e.g., JAR files) and **dependencies**.
* There are two main types:
  + **Local Repository**: A directory on your machine where Maven caches downloaded artifacts and stores temporary build files.
  + **Remote Repository**: Accessed via various protocols (e.g., file://, https://), these can be:
    - **Central Repository**: The default repository for downloading dependencies.
    - **Custom/internal repositories**: Set up within your organization for sharing private artifacts.

***Maven Dependency****:*

* A **dependency** in Maven refers to an external library or module that your project relies on.
* Dependencies are specified in the POM file under the <dependencies> section.
* Maven **automatically downloads** these dependencies from repositories (local or remote) during the build process.
* Transitive dependencies (dependencies of dependencies) are also managed by Maven.

***Maven Clean:***

* + mvn clean is a **Maven command**.
  + When you run mvn clean, it performs the following actions:
    - **Deletes** all previously compiled Java .class files and other resources (like .properties) in your project.
    - Starts the build process from a **clean slate**.
    - Then, it **compiles**, **tests**, and **packages** your Java project.
    - Optionally, it can **install** or **copy** the built .jar or .war file into your local Maven repository.

***Git:***

* **Git** is a **Distributed Version Control System (DVCS)** used to save different versions of files (or sets of files) so that any version can be retrieved at will.
* It also makes it easy to record and compare different file versions.
* With Git, you can track changes, collaborate with others, and manage your codebase efficiently.

*** Version Control System (VCS):***

* A **Version Control System** (also known as **source control**) is a tool used to track, create, and manage changes to software code.
* It helps developers store every change they make to a file at different stages, allowing them and their teammates to retrieve those changes later.
* Essentially, it keeps a historical record of code modifications.

*** Git Commands and Their Functions:***

1. git clone: Downloads an existing repository from a remote server.
   * Syntax: git clone <repository\_url>
2. git commit: Saves changes to the local repository.
   * Syntax: git commit -m "Commit message"
3. git pull: Downloads and integrates changes from a remote repository.
   * Syntax: git pull origin <branch\_name>
4. git push: Publishes local changes to a remote server.
   * Syntax: git push origin <branch\_name>
5. git merge: Integrates changes from one branch into another.
   * Syntax: First, switch to the target branch: git checkout <target\_branch>

Then, merge changes: git merge <source\_branch>

1. git checkout: Switches between branches or restores files.
   * Syntax (switching branches): git checkout <branch\_name>
   * Syntax (restoring files): git checkout <commit\_hash> -- <file\_path>

*** Creating a Repository in Git:***

* To create a new Git repository:
  1. Navigate to the desired directory using the command line.
  2. Run git init to initialize a new repository.
  3. Add your code files using git add.
  4. Commit the changes using git commit.
  5. Optionally, connect your local repository to a remote repository (e.g., on GitHub) using git remote add origin <repository\_url>.

*** git push Command:***

* The git push command is used to upload local commits to a remote repository (such as GitHub).
* After making changes and committing them locally, you can push those changes to the remote repository using git push.
* For example: git push origin main pushes the changes from the local main branch to the remote repository.

1. *Continuous Integration and Deployment (CI/CD):*
   * **Continuous Integration (CI)**: It’s a practice where developers integrate their code into a shared repository multiple times a day. The goal is to catch integration bugs early and ensure that the code integrates smoothly with existing code.
   * **Continuous Delivery (CD)**: This focuses on reducing friction in the deployment process. It automates the steps required to deploy a build, allowing code to be released safely at any time.
   * **Continuous Deployment**: Takes CD one step further by automatically deploying code changes whenever they are made.
2. *CI Tools:*
   * CI tools automate the merge, build, and testing phases of the development process. They integrate with version control systems, code repositories, and other DevOps tools.
   * Some popular CI tools include:
     + **Jenkins**: An open-source automation server that supports building, deploying, and automating projects.
     + **CircleCI**: Supports rapid software development and publishing.
     + **GitLab CI**: Integrates with GitLab for CI/CD.
     + **TeamCity**: A powerful CI server by JetBrains.
     + **Buddy**: Offers automated parallelization and quick tests.
3. *CI/CD Pipeline:*
   * A CI/CD pipeline is a series of steps that deliver a new version of software. It includes building, testing, and deploying code.
   * Automation is key to CI/CD pipelines, minimizing human error and ensuring consistent releases.
4. *Jenkins:*
   * Jenkins is an open-source automation server written in Java.
   * It supports continuous integration and continuous delivery processes.
   * Key features:
     + Easy installation and upgrade.
     + Extensible with a vast plugin ecosystem.
     + Distributed builds with master-slave architecture.
     + Integrates with various tools in the CI/CD toolchain.
5. *Jenkins with Selenium:*
   * Jenkins integrates seamlessly with Selenium for test automation.
   * Benefits:
     + Automates test execution.
     + Provides reports and notifications.
     + Enables continuous testing as part of the CI/CD pipeline.
6. *Creating a Job in Jenkins:*
   * In Jenkins, a job represents a task or a set of tasks.
   * Steps:

Log in to Jenkins.

Click “New Item.”

Choose the type of job (e.g., Freestyle project, Pipeline).

Configure the job settings, including source code management, build triggers, and post-build actions.

1. *Configuring Automatic Builds in Jenkins:*

Configure build triggers (e.g., SCM polling, webhook triggers).

Specify build steps (e.g., compile, test, package).

Jenkins will automatically trigger builds based on defined conditions.

* 1. ***Default Properties in SOAPUI****:*

When you create a new project in **SOAPUI**, it comes with some **inbuilt default properties**. These properties are available as soon as you create a project. While they might vary based on the specific context (such as project type or configuration), here are some common default properties you’ll encounter:

**Project Properties**:

These are global properties associated with the entire project. They can be accessed and modified from any test case or test step within the project.

Examples of project properties include:

project.name: The name of the project.

project.version: The version of the project.

project.environment: The environment (e.g., development, staging, production).

project.baseURL: The base URL for API requests.

**Test Suite Properties**:

Each test suite within a project has its own set of properties.

Examples of test suite properties include:

testsuite.name: The name of the test suite.

testsuite.description: A brief description of the test suite.

testsuite.parallel: Whether to run test cases in parallel within the suite.

testsuite.timeout: The maximum time allowed for executing the entire suite.

**Test Case Properties**:

Similar to test suites, individual test cases also have properties.

Examples of test case properties include:

testcase.name: The name of the test case.

testcase.description: A description of the test case.

testcase.tags: Tags associated with the test case (useful for categorization).

testcase.priority: Priority level (e.g., high, medium, low).

**Test Step Properties**:

Each test step (e.g., REST Request, SOAP Request) within a test case has its own properties.

Examples of test step properties include:

teststep.name: The name of the test step.

teststep.endpoint: The specific endpoint for the request.

teststep.timeout: Timeout duration for the request.

teststep.assertions: Assertions to validate the response.

**Environment Properties**:

These properties are specific to the environment (e.g., development, staging) in which your tests run.

You can define environment-specific variables (e.g., API keys, database connection strings) as properties.

* 1. *Important Functionalities of SOAPUI****:***
* **SOAPUI** is a versatile tool for testing web services. Here are some of its key functionalities:
  + **Functional Testing**: You can create and execute functional tests for both **RESTful APIs** and **SOAP Web Services**. Validate the correctness of your services.
  + **Performance Testing**: Measure how well your services perform under load. Create load tests by right-clicking a functional test and running it as a load test.
  + **Interoperability Testing**: Test the compatibility of your services across different platforms and technologies.
  + **Regression Testing**: Ensure that changes or updates to your services do not break existing functionality.
  + **Service Simulation (Mocking)**: Create mock services to mimic real services before they are implemented. Useful for testing without waiting for actual services to be available.
  + **Data-Driven Testing**: Use the **DataSource TestStep** to read and loop test data from external sources like Excel, XML, JDBC, and files.
  + **Test Debugging**: Follow test flows step-by-step with the **ReadyAPI Test Debugging** interface.
  + **Web Services Coverage Analysis**: Dynamically analyze how well your service contract (SOAP or REST) is covered by your functional tests.
  1. ***What Is SOAPUI?****:*
* **SOAPUI** is a tool for testing web services. It supports both **SOAP Web Services** and **RESTful Web Services** (HTTP-based services).
* Key points about **SOAPUI**:
  + It’s an **open-source** tool with a commercial companion called **ReadyAPI** (for mission-critical web services).
  + You can perform functional testing, performance testing, interoperability testing, regression testing, and more.
  + **Drag-and-drop test creation** makes it easy to create even complex test scenarios.
  + You can simulate web services, record tests for later use, create code stubs from WSDL, and generate REST specifications (WADL) from recorded communication.
  1. ***Role of XML, SOAP, WSDL, and UDDI in Web Services:***
* **Web services** enable different applications to communicate and exchange data over the internet. They rely on standardized protocols to create interoperable applications. Here’s the role of each component:
  + **XML (eXtensible Markup Language)**:
    - XML is a markup language used to structure and represent data.
    - In web services, XML is the primary format for exchanging information between applications.
    - It provides a common syntax for data representation, making it language-agnostic.
  + **SOAP (Simple Object Access Protocol)**:
    - SOAP is an XML-based messaging protocol for communication between applications.
    - It defines a standard way for applications to exchange structured information over the internet.
    - SOAP requests and responses are structured as XML documents.
  + **WSDL (Web Services Description Language)**:
    - WSDL is an XML-based language used to describe web services and their operations.
    - It defines how to interact with a web service without accessing its source code.
    - Components in a WSDL document include definitions, types, messages, port types, bindings, and services.
  + **UDDI (Universal Description, Discovery, and Integration)**:
    - UDDI is a directory service that facilitates the discovery and integration of web services.
    - It allows businesses to publish and find information about available services.
    - UDDI provides a registry where service providers can list their services, making them accessible to potential consumers.
  1. ***Assertions in SoapUI:***
* **Assertions** validate whether the response received from a web service meets expected criteria.
* Commonly used assertions in SoapUI include:
  + **Contains Assertion**:
    - Searches for the existence of a specified string in the response.
    - Supports regular expressions.
  + **Not Contains Assertion**:
    - Searches for the non-existence of a specified string.
    - Also supports regular expressions.
  + **XPath Match Assertion**:
    - Validates whether an XPath expression matches a part of the response.
  + **Scripting Assertions**:
    - Allows custom validation using Groovy scripts.
  + **XQuery Match Assertion**:
    - Validates whether an XQuery expression matches a part of the response.
* Example of a **Contains Assertion**:
  + Suppose we want to check if the string ‘46’ exists in the response:
  + <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  + <soap:Body>
  + <HelloWorldResponse xmlns="http://example.com/">
  + <HelloWorldResult>Hello, John Doe!</HelloWorldResult>
  + </HelloWorldResponse>
  + </soap:Body>
  + </soap:Envelope>
  + If the response contains ‘46’, the assertion passes; otherwise, it fails.
  1. ***Web Services and Communication Channels:***
* **Web services** allow applications to communicate and exchange data over the internet.
* Communication channels available for web services include:
  + **HTTP/HTTPS**: The most common protocol for web services, using standard HTTP methods (GET, POST, PUT, DELETE).
  + **SOAP over HTTP**: SOAP messages transmitted via HTTP.
  + **RESTful APIs**: Use HTTP methods (GET, POST, PUT, DELETE) to interact with resources (URLs).
  + **JMS (Java Message Service)**: For asynchronous communication using message queues.
  + **AMF (Action Message Format)**: Used in Adobe Flash applications.
  + **WebSocket**: Provides full-duplex communication over a single TCP connection.
  + **MQTT (Message Queuing Telemetry Transport)**: Lightweight protocol for IoT devices.
  + **SMTP (Simple Mail Transfer Protocol)**: Used for email-based communication.
  + **FTP (File Transfer Protocol)**: For transferring files between systems.
  1. ***SoapUI Automation:***
* **SoapUI** is an open-source testing tool for API testing, particularly SOAP and REST web services.
* It supports functional tests, security tests, and virtualization.
* **Automation in SoapUI** involves:
  + Creating test cases and test steps.
  + Defining assertions to validate responses.
  + Running tests automatically using the TestRunner (command-line or within SoapUI).
  + Using Groovy scripting for custom logic.
  + Integrating with other tools and frameworks.