Final Roadmap (Enhanced Version)**

1. Software Testing Basics
- SDLC & STLC.
- Manual vs. Automation Testing.
- Types of Testing:
- Functional (Unit, Integration, System, UAT).
- Non-Functional (Performance, Security, Usability).
- Regression, Smoke, Sanity, Exploratory Testing.
- Test Plan, Test Case, Test Strategy.
- Defect Life Cycle & Bug Tracking (JIRA, Bugzilla).
- Agile/Scrum Methodology.

2. Automation Testing Concepts
- Why Automation? ROI Analysis.
- Test Automation Frameworks (Data-Driven, Keyword-Driven, Hybrid, BDD).
- Common Automation Tools (Selenium, Cypress, Playwright, Appium).
- Headless vs. UI-based Testing.
- Parallel Execution & Cross-Browser Testing.
- CI/CD & Test Automation Integration (Jenkins, GitHub Actions, GitLab CI/CD).
- Challenges in Automation.

3. Selenium & Web Automation
- Selenium Architecture & WebDriver API.
- Locators & XPath Optimization.
- Handling Popups, Frames, Windows, Alerts, Cookies.
- Headless Browser Execution (Chrome, Firefox, Edge).
- Handling Dynamic Elements & AJAX Calls.
- Implicit, Explicit & Fluent Waits.
- Page Object Model (POM) & Page Factory.
- Parallel Execution using pytest
- Mobile Web Testing.

4. API Testing Basics
- Understanding REST & SOAP APIs.
- HTTP Methods & Status Codes.
- API Testing Tools (Postman, RestAssured, Requests in Python).
- Automating API Tests in CI/CD Pipelines.
- GraphQL Testing.
- Authentication Mechanisms (OAuth, JWT, API Keys).

5. Performance & Security Testing Basics
- Introduction to Load & Stress Testing (JMeter, Locust).
- Basics of OWASP & Security Testing.
- Security Testing Tools (OWASP ZAP, Burp Suite).
- Basics of Monitoring Tools (New Relic, Datadog).

6. Database & Backend Testing
- SQL Basics for Testers.
- Validating Data with Queries.
- Connecting Automation Scripts with Databases.
  - NoSQL Databases (MongoDB, Cassandra).

- Data Integrity Testing.

7. Test Reporting & Logging
- Generating Reports (Allure, Extent Reports).
- Logging in Automation (Log4j, Python Logging Module).
- Dashboard Creation (Grafana, Kibana).
- Error Handling.

Start---------->

1.1.1 SDLC (Software Development Life Cycle)

SDLC defines the **entire software development process** from ideation to deployment and maintenance. It consists of several phases:

1. **Requirement Analysis** – Understanding what the software should do.
2. **Planning** – Creating a roadmap and estimating costs.
3. **Design** – Architecting the system (UI, database, APIs, etc.).
4. **Development** – Writing the code.
5. **Testing** – Ensuring quality by running test cases.
6. **Deployment** – Releasing the software to production.
7. **Maintenance** – Fixing bugs and improving performance.

 Example:
A bank wants an online loan application system. The SDLC would involve gathering requirements (loan types, interest rates), designing the UI, coding the backend logic, testing different loan scenarios, deploying the app, and maintaining it by fixing issues.

1.1.2 STLC (Software Testing Life Cycle)

STLC is a **subset of SDLC** focused on testing. It ensures the software meets business and functional requirements.

1. **Requirement Analysis** – Understanding what needs to be tested.
2. **Test Planning** – Creating test strategies, estimating effort.
3. **Test Case Development** – Writing test cases and scripts.
4. **Test Environment Setup** – Setting up hardware/software for testing.
5. **Test Execution** – Running the test cases.
6. **Test Closure** – Analyzing test results and reporting defects.

 Example:
For the bank's loan system, STLC involves analyzing loan application scenarios, writing test cases\

(e.g., user enters invalid salary, incorrect loan tenure), setting up a test database, executing tests, logging bugs, and verifying fixes.

1.2 Manual vs. Automation Testing

Feature	Manual Testing	Automation Testing
Execution	Tester manually executes test cases.	Scripts run automatically.
Speed	Slower	Faster
Cost	Low initial cost but high in long-term	High initial cost but saves effort later
Accuracy	Prone to human errors	More accurate
Best For	Exploratory, UI, Ad-hoc testing	Regression, performance, API testing

 Example:

- **Manual Testing:** A tester manually checks whether a login form accepts valid credentials.
- **Automation Testing:** A Selenium script inputs username/password and verifies the login success message.

1.3 Types of Testing

1.3.1 Functional Testing **(Validates business logic)**

- **Unit Testing:** Tests individual components (e.g., a function that calculates interest).
- **Integration Testing:** Tests interactions between modules (e.g., login API working with the database).
- **System Testing:** Tests the full application (e.g., a complete loan application workflow).
- **User Acceptance Testing (UAT):** End-users verify if it meets business needs.

1.3.2 Non-Functional Testing **(Performance, security, usability)**

- **Performance Testing:** Load & stress testing (e.g., 1000 users applying for a loan at once).
- **Security Testing:** Checking vulnerabilities (e.g., SQL injection on login).
- **Usability Testing:** Evaluating user experience.

1.3.3 Other Important Types

- **Regression Testing:** Ensures new changes don’t break existing features.
- **Smoke Testing:** A quick check to verify if the build is testable.
- **Sanity Testing:** Focuses on verifying specific bug fixes.
- **Exploratory Testing:** Tester explores the system to find defects without predefined test cases.

MORE LATER IN DEPTH

1.4 Test Plan, Test Case, Test Strategy

Test Plan **– A document outlining what will be tested, when, how, and by whom.**

 Example: A test plan for a banking app might include testing login, fund transfer, loan application, etc.

Test Case **– A set of inputs, execution steps, and expected outputs for testing.**

 Example:

Test Case ID	Description	Steps	Expected Result
TC001	Verify login	1. Open app 2. Enter credentials 3. Click Login	User is logged in

Test Strategy **– High-level guidelines on how testing should be performed in a project.**

 Example: In Agile projects, the strategy might be to automate regression tests while manually testing new features.

1.5 Defect Life Cycle & Bug Tracking

Defect/Bug Life Cycle: The stages a defect goes through from detection to closure.

1. **New** – Reported by a tester.
2. **Assigned** – Developer is assigned to fix it.
3. **In Progress** – Developer starts working on it.
4. **Fixed** – Developer fixes the issue.
5. **Retest** – Tester verifies the fix.
6. **Closed** – If verified, defect is marked closed.

 Example: A tester finds that the login button is not working. They log a bug in JIRA, the developer fixes it, and the tester verifies it before closing the issue.

1.6 Agile/Scrum Methodology

Agile is an iterative development approach that focuses on customer feedback and rapid delivery. Scrum is a framework within Agile.

Scrum Terminology:

- **Sprint** – A time-boxed iteration (2 weeks).
- **Product Backlog** – List of features/bugs.
- **Sprint Backlog** – Items selected for the sprint.
- **Daily Standup** – Short meeting to discuss progress.
- **Sprint Review** – Demonstration of completed work.
- **Sprint Retrospective** – Discuss what went well and what to improve.

 Example: In a banking app project, the team plans to complete the **"loan eligibility calculator"** in a 2-week sprint. They meet daily to track progress, and at the end, they demonstrate the working feature to stakeholders.

Agile/Scrum promotes iterative development with sprints and regular feedback

---

Types of Software Testing (In-Depth Explanation)

Software testing is broadly classified into **Functional** and **Non-Functional Testing**. Within these, there are multiple subtypes.

1. Functional Testing (Validates Business Logic & Functional Requirements)

Functional testing ensures that the software works as intended, covering specific business requirements.

1.1 Unit Testing

- Tests individual components (functions, methods, classes).
- Ensures correctness of code at the lowest level.
- Typically automated using frameworks like PyTest (Python), JUnit (Java), or NUnit (C#).

 Example:
In a banking application, a function that calculates interest should return the correct value for different inputs:

```
python
CopyEdit
def calculate_interest(principal, rate, time):
    return (principal * rate * time) / 100

assert calculate_interest(1000, 5, 1) == 50  # Unit Test
```

1.2 Integration Testing

- Tests interaction between modules or systems.
- Ensures that APIs, databases, and third-party services communicate properly.
- Can be done using API calls, database queries, or UI interactions.

 Example:

- A **payment gateway API** in an e-commerce app should correctly deduct money from the customer and confirm the order.

Types of Integration Testing:

1. **Top-down approach** – Tests higher-level modules first.
2. **Bottom-up approach** – Tests lower-level modules first.
3. **Big Bang Testing** – All modules tested together.

1.3 System Testing

- Tests the entire application as a whole.
- Ensures compliance with business requirements.
- Involves UI, backend, database, API, and external integrations.

 Example:
A **hospital management system** should allow patients to book appointments, doctors to view records, and admins to generate reports.

1.4 User Acceptance Testing (UAT)

- Conducted by end-users before final deployment.
- Ensures the software meets business expectations.

 Example:
A **retail store owner** testing a POS system before launching in stores.

1.5 Smoke Testing

- A **quick sanity check** to see if the application is stable enough for detailed testing.
- Verifies critical features like login, dashboard, and basic workflows.

 Example:
In a **social media app**, checking if users can log in, post updates, and view notifications before starting detailed testing.

1.6 Sanity Testing

- Performed after bug fixes to confirm they are resolved without affecting other parts of the system.

 Example:
A **shopping cart bug fix** (items not being removed) is tested to confirm that removing items now works without breaking other features.

1.7 Regression Testing

- Ensures that new changes do not break existing functionality.
- Often automated due to repetitive nature.

 Example:
If a **profile update feature** is added to a mobile banking app, regression testing ensures login, fund transfer, and statements still work as before.

1.8 Exploratory Testing

- Testers use their creativity and domain knowledge to find unexpected issues.
- No predefined test cases.

 Example:
A tester randomly inputs **special characters in a signup form** to check if it breaks.

2. Non-Functional Testing (Validates Performance, Security, & Usability)

2.1 Performance Testing

Ensures the application performs well under various conditions.

2.1.1 Load Testing

- Tests how the system handles expected user loads.
- Done using tools like JMeter or Locust.

 Example:
A **ticket booking website** is tested with **500 concurrent users** to see if it slows down.

2.1.2 Stress Testing

- Puts the system under extreme conditions to check failure points.

 Example:
Testing a **stock trading platform** during peak trading hours.

2.1.3 Spike Testing

- Sudden increase in load to see if the system can handle traffic spikes.

 Example:
A **flash sale event** where website traffic jumps from **100 users to 10,000 in seconds**.

2.1.4 Endurance (Soak) Testing

- Tests system stability over an extended period.

 Example:
A **game server** running continuously for **48 hours** to check for memory leaks.

2.2 Security Testing

Ensures the application is safe from attacks and data breaches.

2.2.1 Vulnerability Testing

- Identifies security flaws in the system.
- Uses tools like **OWASP ZAP** and **Burp Suite**.

 Example:
Checking if **passwords are stored in plain text** instead of being hashed.

2.2.2 Penetration Testing

- Ethical hackers simulate attacks to find weaknesses.

 Example:
A security team **tries SQL injection attacks** on an e-commerce site’s login page.

2.2.3 Authentication Testing

- Ensures login mechanisms are secure.

 Example:
Verifying that an **OAuth-based login system** doesn't allow session hijacking.

2.3 Usability Testing

- Ensures the application is easy to use and user-friendly.
- Focuses on navigation, accessibility, and UI design.

 Example:
Testing if a **banking app’s fund transfer process** is intuitive for new users.

2.4 Compatibility Testing

- Ensures the software works across different **browsers, OS, and devices**.

 Example:
A web application is tested in **Chrome, Firefox, Edge, and Safari** to ensure consistency.

2.5 Localization & Globalization Testing

- Checks if the application works across different regions, languages, and time zones.

 Example:
An **e-commerce site** displaying **₹ (INR) in India** but **$ (USD) in the US**.

3. Other Important Testing Types

3.1 Alpha & Beta Testing

- **Alpha Testing**: Done in a controlled environment by internal testers.
- **Beta Testing**: Done by real users before the final release.

 Example:
A **new mobile banking app** is first tested internally (**Alpha**) and then released to select customers for feedback (**Beta**).

3.2 Accessibility Testing

- Ensures the application is usable by people with disabilities (e.g., screen readers, color contrast for the visually impaired).

 Example:
A government website ensuring **keyboard navigation** for users who cannot use a mouse.

3.3 Chaos Testing

- Random failures are introduced to test system resilience.
- Used in cloud and microservices environments.

 Example:
Netflix uses **Chaos Monkey** to randomly **shut down servers** to see how their system handles failures.

Summary of Testing Types

Type	Purpose	Example
Unit Testing	Test individual functions	Verify `calculate_interest()` function.
Integration Testing	Test module interactions	Ensure login API works with the database.
System Testing	Test full application	Check complete loan processing workflow.
UAT	End-user validation	Retail store owner testing POS system.
Performance Testing	Validate speed & stability	Check if a stock trading platform handles high traffic.
Security Testing	Prevent hacking & data leaks	Test for SQL injection on login pages.
Usability Testing	Ensure user-friendliness	Verify banking app UI is easy to navigate.
Regression Testing	Check new changes don’t break old features	Test that adding a profile update feature doesn’t break login.

Automation Testing Concepts (In-Depth Explanation)

Automation testing is the process of using tools and scripts to execute test cases without manual intervention. It helps improve efficiency, repeatability, and accuracy in software testing.

1. Why Automation? ROI Analysis

1.1 Benefits of Automation

✅Faster test execution.
✅Reduces manual effort and human errors.
✅Ensures repeatability and consistency.
✅Supports large-scale regression testing.
✅Enables parallel execution and CI/CD integration.

1.2 When to Automate?

✅**Stable Features** – Avoid automating frequently changing functionality.
✅**Repetitive Tests** – Login, form submission, data validation.
✅**High-Risk Features** – Payment gateway, authentication.

1.3 When NOT to Automate?

❌**Exploratory Testing** – Requires human intuition.
❌**UI Changes Frequently** – Automation scripts may fail often.
❌**One-Time Test Cases** – Not worth the automation effort.

2. Test Automation Frameworks

A test automation framework provides a structured approach to scripting, execution, and reporting.

2.1 Data-Driven Framework

- Uses external data sources (Excel, CSV, databases) to drive test cases.
- Suitable for tests requiring multiple sets of input data.

 Example: Testing a login page with different username-password combinations stored in an Excel file.

2.2 Keyword-Driven Framework

- Uses predefined keywords mapped to test actions.
- Non-technical users can write test cases.

 Example:

Keyword	Action	Element	Value
Click	Button	Login	
EnterText	Field	Username	admin

2.3 Hybrid Framework

- Combination of **Data-Driven** and **Keyword-Driven** approaches.
- Provides flexibility and reusability.

2.4 Behavior-Driven Development (BDD) Framework

- Uses **human-readable test cases** written in **Gherkin syntax** (`Given-When-Then`).
- Popular with tools like **Cucumber** and **pytest-bdd**.

 Example:

```
gherkin
CopyEdit
Feature: User Login
  Scenario: Successful login
    Given User is on login page
    When User enters valid credentials
    Then User is redirected to the dashboard
```

3. Cypress vs. Selenium (Comparison, Use Cases, & Differences)

Feature	Selenium	Cypress
Language Support	Multi-language (Java, Python, C#, JavaScript, etc.)	Only JavaScript/TypeScript
Browser Support	Chrome, Firefox, Edge, Safari	Chrome, Edge, Firefox
Architecture	Uses WebDriver API to interact with browsers	Runs inside the browser (faster execution)
Speed	Slower due to external WebDriver communication	Faster due to direct browser execution
Debugging	Relies on logs and breakpoints	Built-in debugging and time-travel feature
API Testing	Requires additional tools (RestAssured, Requests)	Has built-in API testing support
File Uploads & Downloads	Complex, requires third-party libraries	Native support
Mobile Testing	Can be used with Appium	No mobile support
Best Use Case	Large-scale cross-browser testing	Fast, modern web apps with JavaScript

3.1 Selenium (In-Depth Explanation)

Selenium is a widely used web automation tool that supports multiple programming languages and browsers.

Selenium WebDriver Architecture

1. **Test Script** – Written in Python, Java, or another language.
2. **Selenium WebDriver** – Interacts with browsers using HTTP requests.
3. **Browser Driver** – ChromeDriver, GeckoDriver, etc.
4. **Browser** – Chrome, Firefox, Edge.

Key Features of Selenium

✅Supports multiple programming languages.
✅Works with multiple browsers.
✅Allows for UI-based testing.
✅Integrates with CI/CD tools like Jenkins.

3.2 Cypress (In-Depth Explanation)

Cypress is a JavaScript-based automation tool designed for modern web applications.

Cypress Architecture

- Runs **inside** the browser, making it faster than Selenium.
- Provides **real-time reloading** and **automatic waiting** (no need for explicit waits).

Key Features of Cypress

✅Built-in waiting mechanism (handles AJAX calls automatically).
✅Fast execution inside the browser.
✅Real-time debugging and **time-travel debugging**.
✅Native API testing support.
✅Easy setup compared to Selenium.

4.1 What is Jenkins?

Jenkins is an open-source CI/CD tool that helps automate the build, test, and deployment process.

✅Supports **automation testing** integration.
✅Works with **Git, Docker, Kubernetes, Selenium, Cypress**.
✅Supports **plugins for test reporting**.

5. Challenges in Automation Testing

 Flaky Tests – UI elements change dynamically.
 Cross-Browser Issues – Different browser behaviors.
 Test Data Management – Handling dynamic data in automated tests.
 Test Maintenance – Scripts break with frequent UI updates.

1. What is Headless Testing?

Headless testing refers to running tests **without opening the browser UI**. The browser runs in the background, making test execution faster and more efficient.

✅**Used for:**

- Running tests in **CI/CD pipelines**.
- **Faster execution** as there is no UI rendering.
- Running tests on **servers** without a graphical interface.

2. What is UI-Based Testing?

UI-based testing executes tests **with the browser UI visible**, simulating real user interactions.

✅**Used for:**

- **Visual validation** (layout, CSS styles, animations).
- **Debugging** test failures more easily.
- **Manual intervention** when needed.

3. Headless vs. UI-Based Testing (Comparison Table)

Feature	Headless Testing	UI-Based Testing
Speed	Faster (no UI rendering)	Slower (renders full UI)
Resource Usage	Lower (less memory & CPU)	Higher (UI elements consume resources)
Best Use Case	CI/CD, background execution	Debugging, visual testing
Browser Support	Supported in Chrome, Firefox, Edge	All browsers
Debugging	Harder (no UI to inspect)	Easier (can see elements visually)
User Experience Testing	❌No	✅Yes
Performance Impact	Minimal	High (due to UI rendering)

2. Simple Diagram of Selenium WebDriver Flow

Your Script → Selenium WebDriver → Browser Driver → Browser

For example, if you want to open Google in Chrome using Selenium:

✅**Step 1:** Your script sends a command to **Selenium WebDriver**
✅**Step 2:** WebDriver sends the command to **ChromeDriver**
✅**Step 3:** ChromeDriver instructs **Google Chrome** to open `https://google.com`
✅**Step 4:** The browser executes the action

4. Why Do We Need a Browser Driver?

✅Selenium WebDriver **cannot** directly communicate with the browser.
✅Browser drivers (like ChromeDriver) act as a **translator** between Selenium and the browser.

Selenium Architecture & WebDriver API

1. Selenium Architecture Overview

Selenium WebDriver follows a **client-server architecture**, where the test script (client) communicates with the browser (server) through the WebDriver.

2. Components of Selenium WebDriver

1. **Selenium Client Libraries (Language Bindings)**
    - Selenium supports multiple languages like **Python, Java, JavaScript, C#, and Ruby**.
    - These libraries allow you to write scripts in different programming languages.
2. **JSON Wire Protocol** (for communication)
    - Selenium uses **JSON Wire Protocol** to send HTTP requests between the client and the browser driver.
    - It acts as a **messenger** between Selenium WebDriver and the browser driver.
3. **Browser Drivers** (ChromeDriver, GeckoDriver, etc.)
    - Every browser has its **own driver** (e.g., ChromeDriver for Chrome, GeckoDriver for Firefox).
    - These drivers translate Selenium commands into native browser commands.
4. **Browsers** (Chrome, Firefox, Edge, etc.)
    - The browser executes the commands and sends responses back to the WebDriver.

3. How Selenium WebDriver Works (Step-by-Step Flow)

Example: Opening Google in Chrome

1. The test script (written in Python, Java, etc.) sends a command like `driver.get("https://www.google.com")`.
2. Selenium WebDriver converts it into an HTTP request.
3. The request is sent to the **browser driver** (e.g., ChromeDriver).
4. The browser driver executes the command in the actual browser.
5. The response (e.g., page title) is sent back to Selenium WebDriver.
6. The WebDriver provides the response to the test script.

Locators are selectors used to find elements on a webpage.

Locator Type	Description	Example
ID	Finds an element by ID attribute	`driver.find_element(By.ID, "username")`
Name	Finds an element by name attribute	`driver.find_element(By.NAME, "email")`
Class Name	Finds elements by class name	`driver.find_element(By.CLASS_NAME, "btn")`
Tag Name	Finds elements by HTML tag	`driver.find_element(By.TAG_NAME, "input")`

Locator Type	Description	Example
Link Text	Finds links by text	`driver.find_element(By.LINK_TEXT, "Click Here")`
Partial Link Text	Finds links by partial text	`driver.find_element(By.PARTIAL_LINK_TEXT, "Click")`
CSS Selector	Uses CSS syntax for element selection	`driver.find_element(By.CSS_SELECTOR, "div#content p")`
XPath	Uses XML Path syntax for complex selection	`driver.find_element(By.XPATH, "//input[@id='username']")`

2. XPath Optimization

XPath is used for finding elements **when other locators fail**.

✅**Absolute XPath (Not Recommended)**

- Starts from the root element (`html`).

/html/body/div[1]/form/input[2]

Relative XPath (Recommended)

- Starts from the **desired element**, not the root.

3. Optimized XPath Techniques

1. **Using** `contains()` (find elements with partial attributes)

```
xpath
CopyEdit
//button[contains(text(),'Login')]
```

2. **Using** `starts-with()` (find elements with dynamic attributes)

```
xpath
CopyEdit
//input[starts-with(@id,'user_')]
```

3. **Using** `and/or` **for multiple conditions**

```
xpath
CopyEdit
//input[@type='text' and @class='form-control']
```

4. **Using** `following-sibling` (find next elements)

```
xpath
CopyEdit
//label[text()='Username']/following-sibling::input
```

Handling Popups, Frames, Windows, Alerts, and Cookies

1. Handling Popups (JavaScript Alerts)

Selenium can interact with JavaScript popups using `switch_to.alert`.

```
python
CopyEdit
alert = driver.switch_to.alert
alert.accept()  # Clicks OK
alert.dismiss()  # Clicks Cancel
```

2. Handling Frames (Switching to iframes)

Frames are used to embed content within a page. Selenium needs to **switch** to the frame before interacting with elements inside it.

```
python
CopyEdit
# Switch to a frame by index
driver.switch_to.frame(0)

# Switch to a frame by name or ID
driver.switch_to.frame("frameID")

# Switch back to the main page
driver.switch_to.default_content()
```

3. Handling Multiple Windows (Switching Tabs)

When a new tab/window opens, you need to switch to it.

```
python
CopyEdit
# Get all window handles
windows = driver.window_handles

# Switch to the new window
driver.switch_to.window(windows[1])

# Close the current window
driver.close()
```

4. Handling Cookies

```
python
CopyEdit
# Get all cookies
cookies = driver.get_cookies()

# Add a cookie
driver.add_cookie({"name": "session", "value": "abc123"})

# Delete all cookies
driver.delete_all_cookies()
```

1. What is Headless Execution?

Headless mode runs the browser **without a GUI (no visible window)**.
✅**Faster execution**
✅**Reduces resource consumption**
✅**Useful for CI/CD (Jenkins, Docker, Cloud testing)**

options = Options() options.add_argument("--headless") # Enable headless mode options.add_argument("--disable-gpu") driver = webdriver.Chrome(options=options)

****

1. What Are Dynamic Elements?

Dynamic elements change frequently on a webpage due to **AJAX calls, JavaScript updates, or user interactions**.
Example:

- Search suggestions (Google autocomplete).
- Loading spinners that replace content dynamically.
- Notification popups appearing after an action.

2. Problems with Dynamic Elements

✅The element may not be present immediately.
✅The element's attributes (ID, class, text) may change dynamically.
✅The element may disappear and reappear multiple times.

3. Solutions for Handling Dynamic Elements

A. Use `WebDriverWait` for AJAX Loading

AJAX (Asynchronous JavaScript and XML) updates content **without reloading the page**.
Selenium **cannot detect when AJAX finishes**, so we use **Explicit Waits**.

from selenium.webdriver.common.by import By

from selenium.webdriver.support.ui import WebDriverWait

from selenium.webdriver.support import expected_conditions as EC

 # Wait for a dynamic element (up to 10 seconds)

wait = WebDriverWait(driver, 10)

dynamic_element = wait.until(EC.presence_of_element_located((By.ID, "dynamicContent")))

B. Retry Clicking (Handling StaleElementException)

Sometimes elements disappear and reappear, causing a `StaleElementReferenceException`.

Solution: Re-locate the element before interacting

```
from selenium.common.exceptions import StaleElementReferenceException

for _ in range(5):  # Retry up to 5 times
    try:
        button = driver.find_element(By.ID, "retryButton")
        button.click()
        break
    except StaleElementReferenceException:
        print("Element changed, retrying...")
```

2. Types of Waits in Selenium

Wait Type	Description	Best Use Case
Implicit Wait	Waits for a fixed time before throwing `NoSuchElementException`	Simple scenarios where all elements load in a set time
Explicit Wait	Waits until a condition is met (e.g., element is clickable)	Best for handling AJAX elements and popups
Fluent Wait	Similar to Explicit Wait but retries every few milliseconds	Best for elements that appear intermittently

3. Implicit Wait in Selenium

- Applies to **all elements** globally.
- If an element is not found, Selenium keeps **retrying** until timeout.

❌**Drawback**: Slows down tests if the element appears quickly.

4. Explicit Wait in Selenium

- Waits **only** for a specific element.
- Uses `WebDriverWait` and `expected_conditions`.

✅**Best for handling AJAX elements.**

5. Fluent Wait in Selenium

- Like Explicit Wait but checks the element **at intervals** instead of waiting continuously.

```
from selenium.webdriver.support.wait import FluentWait
from selenium.common.exceptions import NoSuchElementException
import time

wait = FluentWait(driver) \
```

```
    .with_timeout(15) \  # Maximum 15 seconds
    .polling_every(1) \  # Check every 1 second
    .ignoring(NoSuchElementException)

element = wait.until(lambda driver: driver.find_element(By.ID, "dynamicElement"))
element.click()
```

✅**Reduces test execution time by polling instead of waiting blindly.**

Page Object Model (POM) & Page Factory

1. What is Page Object Model (POM)?

POM is a design pattern that **separates the test script from the UI elements**.
✅**Improves code maintainability**
✅**Reduces redundancy**
✅**Easier debugging and scalability**

2. Structure of POM

In POM, we create separate **Page Classes** for each webpage.

----------XCRO=----------

If Selenium **finds the element** but **cannot click it**, the possible reasons are:

1. **The element is not yet interactable** (e.g., hidden behind a loader).
2. **Another element (like an overlay) is blocking it**.
3. **The page has refreshed or updated the DOM** after finding the element.
4. **A race condition** where Selenium finds an old version of the element (causing a `StaleElementReferenceException`).
5. This error occurs when **Selenium finds an element, but it is no longer valid in the DOM** before interacting with it.

.ElementClickInterceptedException

 Why? Another element (e.g., overlay, popup, loader) is blocking the click.

Fix: Scroll into view before clicking

✅**Fix: Wait for the element to be clickable**

2. NoSuchElementException

 Why? The element is **not found** in the DOM.

3. `TimeoutException`

 Why? The element **did not appear within the wait time**.

4. `InvalidElementStateException`

 Why? The element is **not in a state** to be interacted with (e.g., disabled).

5. `WebDriverException: target window already closed`

 Why? Selenium is trying to interact with a window that was closed.

For stale elements use for loop and try excepte

□ Common Situations Where `StaleElementReferenceException` Occurs

1□ Page Reload or Navigation

If the page reloads, all elements are **destroyed and recreated** in the DOM. Selenium loses reference to the old elements, making them stale.

2□ DOM Changes Due to JavaScript

If JavaScript dynamically updates the page (e.g., AJAX, React updates), the elements might be **removed and recreated**, making previous references stale.

 Example Scenario:

- A search results list updates dynamically.
- The old elements get replaced with new ones.
- Trying to interact with the old elements results in a stale error.

Table Rows or Lists Dynamically Changing

If a table or list updates dynamically (e.g., removing or adding rows), the references to old elements become stale.

Parallel Execution Using pytest

Parallel execution helps reduce test execution time by running multiple tests simultaneously. In `pytest`, we use the `pytest-xdist` plugin to run tests in parallel.

□ Why Parallel Execution?

- **Faster execution:** Tests run on multiple cores instead of sequentially.
- **Efficient resource utilization:** Uses multiple CPUs or machines.

- **Better scalability:** Helps in CI/CD pipelines to test faster.

pytest -n 4 # Runs tests on 4 parallel processes

Running Parallel Tests with Fixtures

If tests share resources (e.g., opening a browser), use `scope="session"` in `pytest.fixture` to **reuse** the browser across tests.

```
import pytest
from selenium import webdriver

@pytest.fixture(scope="session")
def browser():
    driver = webdriver.Chrome()
    yield driver
    driver.quit()

def test_google(browser):
    browser.get("https://www.google.com")
    assert "Google" in browser.title

def test_bing(browser):
    browser.get("https://www.bing.com")
    assert "Bing" in browser.title
```

Running Selenium Tests in Mobile Mode

You can simulate a mobile browser in **desktop Chrome** using **Chrome DevTools Protocol (CDP**

```
from selenium import webdriver

mobile_emulation = {
    "deviceName": "iPhone X"
}

options = webdriver.ChromeOptions()
options.add_experimental_option("mobileEmulation", mobile_emulation)
```

```
driver = webdriver.Chrome(options=options)

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

print(driver.title)

driver.quit()
```

API Testing Basics

API (Application Programming Interface) testing ensures that software applications communicate correctly over the internet or within systems. It verifies **functionality, performance, security, and reliability** of APIs.

Understanding REST & SOAP APIs

APIs mainly follow two architectures: **REST (Representational State Transfer)** and **SOAP (Simple Object Access Protocol)**.

□ REST API

- **Lightweight & fast**: Uses JSON for data exchange.
- **Stateless**: Server does not store client session.
- **Follows HTTP methods**: GET, POST, PUT, DELETE.
- **Uses URIs (Uniform Resource Identifiers)** to identify resources.

□ SOAP API

- **Uses XML format** (not JSON).
- **More secure** but **slower** than REST.
- **Follows strict messaging rules**.
- **Uses WSDL (Web Services Description Language)** for operations.

□ REST vs. SOAP

Feature	REST	SOAP
Data Format	JSON, XML, HTML	Only XML
Speed	Faster	Slower
Security	OAuth, JWT	WS-Security
Used In	Web services, mobile apps	Banking, enterprise apps

□ HTTP Methods & Status Codes

□ HTTP Methods

Method	Purpose	Example
GET	Retrieve data	`GET /users/1`

Method	Purpose	Example
POST	Create a new resource	`POST /users`
PUT	Update an existing resource	`PUT /users/1`
DELETE	Remove a resource	`DELETE /users/1`

Code	Meaning	Example Scenario
200 OK	Success	Successfully fetched user data
201 Created	Resource created	New user added to DB
400 Bad Request	Invalid input	Missing required fields in request
401 Unauthorized	Authentication required	Login required
403 Forbidden	No access	User has no permission
404 Not Found	Resource missing	User does not exist
500 Internal Server Error	Server crashed	API bug or issue

```
response = requests.get("https://jsonplaceholder.typicode.com/users/1")


if response.status_code == 200:
    print("Success:", response.json())
elif response.status_code == 404:
    print("User not found")
else:
    print("Error:", response.status_code)
```

□ 3. Requests Library (Python API Testing)

- **Best for**: Simple and fast API testing in Python.
- **Features**:
  ✅Supports all HTTP methods.
  ✅Works with JSON payloads.
  ✅Can be integrated into **Pytest for automation**.

✅**Example: GET & POST Requests in Python**

```
import requests


# GET request
response = requests.get("https://jsonplaceholder.typicode.com/users/1")
print(response.json())  # Print response
```

```
# POST request
payload = {"name": "John", "email": "john@example.com"}
response = requests.post("https://jsonplaceholder.typicode.com/users", json=payload)
print(response.status_code, response.json())
```

Authentication Mechanisms (OAuth, JWT, API Keys)

APIs often require authentication for security.

□ 1. API Keys

- Simple **key-based authentication**.
- Sent in **headers or query params**.

2. OAuth 2.0

- **Token-based authentication** for security.
- Used by **Google, Facebook, GitHub APIs**.
- **Steps:**
  1. User logs in → API sends a request to **OAuth provider**.
  2. OAuth provider generates an **Access Token**.
  3. API sends **Bearer Token** in headers.

□ 3. JWT (JSON Web Token)

- Token-based **stateless authentication**.
- Used in **secure web apps & APIs**.
- JWT structure: **Header + Payload + Signature**.

1□ Header (Specifies Algorithm & Type) 2□ Payload (Contains Claims & Data) **3□ Signature (Ensures Security & Integrity)**

Generated using **HMAC SHA256** or **RSA**.

How JWT Works (Authentication Flow)

JWT is commonly used for **stateless authentication**.

□ JWT Authentication Process

1□ **User logs in** → Sends credentials (`username`, `password`).
2□ **Server verifies credentials** → Generates a JWT token.
3□ **Client stores JWT** → In local storage or cookies.

4 **Client makes API requests** → Sends JWT in **Authorization header**.
5 **Server validates JWT** → Grants or denies access.

Types of Performance Testing

Type	Purpose
Load Testing	Tests system behavior under expected user load.
Stress Testing	Tests system beyond expected limits (crash behavior).
Spike Testing	Tests system response to sudden traffic spikes.
Scalability Testing	Checks how well the system scales with increased workload.
Endurance Testing	Tests system stability over a long period.

Locust (Python-based, easy to script for scalable load testing)

```
from locust import HttpUser, task, between


class WebsiteUser(HttpUser):
    wait_time = between(1, 3)  # Users wait between requests


    @task
    def test_home_page(self):
        self.client.get("/")  # Simulating user visiting home page


    @task
    def test_login(self):
        self.client.post("/login", json={"username": "user", "password": "pass"})
```

```
locust -f locustfile.py --host=http://yourwebsite.com
```

Security Testing Basics

Security testing identifies **vulnerabilities, weaknesses, and risks** in an application. The goal is to **prevent security breaches**.

Types of Security Testing

Type	Purpose
Vulnerability Testing	Finds security flaws in code (SQL injection, XSS, CSRF).
Penetration Testing	Simulates hacker attacks to identify security gaps.

Type	Purpose
Authentication Testing	Tests login mechanisms, token security (JWT, OAuth).
Session Management Testing	Ensures sessions are secure against hijacking.

✅Burp Suite (Professional Penetration Testing)

Burp Suite is a **powerful penetration testing** tool for web security.

 Captures HTTP requests (like Postman, but for security).
 Intercepts & Modifies requests to test vulnerabilities.
 Supports **Brute-force, SQL injection, and XSS detection**.

Example: Testing Login Security with Burp Suite

1. **Start Burp Suite** → Enable "Intercept Mode".
2. **Login to the Web App** → Capture the request.
3. **Modify the Request** → Try SQL Injection (`admin' OR '1'='1`).
4. **Send & Analyze the Response** → See if it bypasses authentication.

✅Python Logging Module (For Selenium & API Tests)

Instead of printing messages to the console (`print()`), use structured logging.

 Setup Logging in Selenium (Python)

1. **Basic logging setup:**

```
import logging


logging.basicConfig(filename="test_log.log", level=logging.INFO, format="%(asctime)s
- %(levelname)s - %(message)s")


logging.info("Test Execution Started")

logging.warning("Payment data mismatch detected")

logging.error("Element not found error")
```

Key Concepts of BDD

Term	Meaning
Feature File	Contains test scenarios written in Gherkin syntax (`.feature` file).
Gherkin Language	A simple language to define test cases (Given-When-Then).
Step Definitions	The actual implementation of the Gherkin steps using Selenium, API calls, etc.
BDD Frameworks	`Cucumber (Java)`, `Behave (Python)`, `SpecFlow (.NET)`, `Jest + Cypress (JS)`

```
Feature: Login to Web Application
```

As a user,

I want to log in to the web application

So that I can access my dashboard.

Scenario: Successful Login

Given I am on the login page

When I enter valid credentials

And I click the login button

Then I should be redirected to the dashboard

Feature: API Testing for User Login

Scenario: Valid Login

Given I have a valid API endpoint

When I send a POST request with valid credentials

Then I should receive a status code 200

And the response should contain a valid token

>>>>>PYTHON IMPLEMENTATION

```
from behave import given, when, then
from selenium import webdriver
from selenium.webdriver.common.by import By

@given('the user is on the login page')
def open_login_page(context):
    context.driver = webdriver.Chrome()
    context.driver.get("https://example.com/login")

@when('they enter "{username}" and "{password}"')
def enter_credentials(context, username, password):
    context.driver.find_element(By.ID, "username").send_keys(username)
```

```
    context.driver.find_element(By.ID, "password").send_keys(password)


@when('they click the login button')
def click_login_button(context):
    context.driver.find_element(By.ID, "login-button").click()


@then('they should see the dashboard')
def verify_dashboard(context):
    assert "Dashboard" in context.driver.page_source
    context.driver.quit()
```

□ Database & Backend Testing for Automation Testers (Using Python)

Database testing ensures that **data is correctly stored, retrieved, and validated** in the backend. Since you're using **Python**, we will focus on database testing using **SQL, NoSQL (MongoDB), and Python libraries like** `sqlite3`, `pymysql`, **and** `pymongo`.

□ Common SQL Operations

Operation	SQL Query Example
Select Data	`SELECT * FROM users;`
Filter Data	`SELECT * FROM users WHERE age > 25;`
Insert Data	`INSERT INTO users (id, name, age) VALUES (1, 'Alice', 30);`
Update Data	`UPDATE users SET age = 31 WHERE id = 1;`
Delete Data	`DELETE FROM users WHERE id = 1;`
Join Tables	`SELECT orders.id, users.name FROM orders INNER JOIN users ON orders.user_id = users.id;`

□ **Use case in automation testing:**

- Verify if **data is correctly inserted after user actions** (e.g., placing an order).
- Ensure **data consistency** across different tables.
- Fetch expected data from the database and compare it with UI/API data.

2□ Validating Data with Queries (Python + MySQL)

```
import pymysql

# Connect to MySQL database
connection = pymysql.connect(
    host='localhost',
    user='root',
    password='password',
    database='test_db'
)
cursor = connection.cursor()

# Query data
cursor.execute("SELECT name, age FROM users WHERE id=1;")
result = cursor.fetchone()  # Fetch one row

# Validate data
expected_name = "Alice"
expected_age = 30

assert result[0] == expected_name, f"Expected {expected_name}, but got {result[0]}"
assert result[1] == expected_age, f"Expected {expected_age}, but got {result[1]}"

print("✅Data validation passed!")

# Close connection
cursor.close()
connection.close()
```

2nd example:

```
import pymysql
from selenium import webdriver
```

```
# Step 1: Place an order via Selenium (UI Automation)
driver = webdriver.Chrome()
driver.get("https://example.com")
driver.find_element("id", "order-button").click()
order_id = driver.find_element("id", "order-id").text  # Get order ID from UI
driver.quit()

# Step 2: Fetch order details from DB
connection = pymysql.connect(host='localhost', user='root', password='password',
database='test_db')
cursor = connection.cursor()
cursor.execute(f"SELECT status FROM orders WHERE order_id={order_id};")
db_status = cursor.fetchone()[0]

# Step 3: Validate DB result
assert db_status == "Confirmed", f"Expected 'Confirmed', but got {db_status}"
print("✅Order status is correctly updated in DB.")

# Close connection
cursor.close()
connection.close()
```

Pymongo:

```
from pymongo import MongoClient

# Connect to MongoDB
client = MongoClient("mongodb://localhost:27017/")
db = client["test_db"]
collection = db["users"]

# Fetch user data
user = collection.find_one({"email": "test@example.com"})
```

Validate data

expected_name = "Alice"

assert user["name"] == expected_name, f"Expected {expected_name}, but got {user['name']}"

print("✅MongoDB validation passed!")

□ Data Integrity Testing in Automation (With Python)

□ What is Data Integrity Testing?

Data integrity testing ensures that data in a database remains:

1. **Accurate** → The data should match expected values.
2. **Consistent** → Data should not change unexpectedly.
3. **Valid** → Data must conform to defined formats and constraints.
4. **Reliable** → No missing, duplicate, or corrupted data.

1□ Types of Data Integrity Testing

Type	Description	Example Check
Entity Integrity	Ensures each row has a unique identifier (**Primary Key**).	No two rows should have the same **ID**.
Referential Integrity	Ensures relationships between tables are maintained (**Foreign Keys**).	Deleting a user should also remove their orders.
Domain Integrity	Ensures values are within allowed constraints.	`age` should not be negative.
User-Defined Integrity	Business rules are correctly implemented.	A user cannot withdraw more money than available.

2□ Common Data Integrity Issues

- ❌**Duplicate Records** (Multiple entries of the same data)
- ❌**Missing or Null Values** (Essential fields left blank)
- ❌**Orphan Records** (Foreign key references missing data)
- ❌**Data Type Mismatch** (String stored where an integer is expected)
- ❌**Incorrect Relationships** (Broken references between tables)
- ❌**Outdated or Stale Data** (Old data not being updated)

3 Data Integrity Testing With SQL (Python Examples)

✅1. Check for Duplicate Records

Duplicate records can cause incorrect reporting and calculations

```
import pymysql

connection = pymysql.connect(host='localhost', user='root', password='password', database='test_db')
cursor = connection.cursor()

# Find duplicate emails
cursor.execute("SELECT email, COUNT(*) FROM users GROUP BY email HAVING COUNT(*) > 1;")
duplicates = cursor.fetchall()

assert len(duplicates) == 0, f"❌Found duplicate records: {duplicates}"
print("✅No duplicate users found.")

cursor.close()
connection.close()
```

PYTEST + SQL

```
import pymysql
import pytest

@pytest.fixture(scope="module")
def db_connection():
    connection = pymysql.connect(host='localhost', user='root', password='password', database='test_db')
    yield connection
    connection.close()
```

```
def test_no_duplicate_users(db_connection):
    cursor = db_connection.cursor()
    cursor.execute("SELECT email, COUNT(*) FROM users GROUP BY email HAVING COUNT(*) > 1;")
    duplicates = cursor.fetchall()
    assert len(duplicates) == 0, f"❌Found duplicate records: {duplicates}"

def test_no_orphan_orders(db_connection):
    cursor = db_connection.cursor()
    cursor.execute("""
    SELECT orders.user_id FROM orders
    LEFT JOIN users ON orders.user_id = users.id
    WHERE users.id IS NULL;
    """)
    orphans = cursor.fetchall()
    assert len(orphans) == 0, f"❌Found orphan orders: {orphans}"

def test_valid_ages(db_connection):
    cursor = db_connection.cursor()
    cursor.execute("SELECT age FROM users;")
    ages = [row[0] for row in cursor.fetchall()]
    invalid_ages = [age for age in ages if not isinstance(age, int) or age < 0]
    assert not invalid_ages, f"❌Found invalid ages: {invalid_ages}"
```

What is the difference between Smoke, Sanity, and Regression Testing?

Type	Description
Smoke Testing	Basic checks if the build is stable for testing.
Sanity Testing	Focuses on new functionality verification.
Regression Testing	Ensures existing features still work after changes.

✅**Short Answer:**

- **Smoke** → Basic stability check
- **Sanity** → Quick new feature check
- **Regression** → Ensures old features work

How do you ensure test scripts are maintainable?

✅**Detailed Answer:**

1. **Use POM (Page Object Model)** for modularity.
2. **Avoid hardcoded values** (use config files).
3. **Use parameterized tests** (Pytest/JUnit).
4. **Follow naming conventions** & structure.
5. **Use logging & reporting** (Allure, Extent Reports).

Introduction to Locust for Performance Testing in Python

Locust is a **Python-based performance testing tool** that helps simulate users on a website or API to measure how it performs under load.

1 Why Use Locust?

✓ **Lightweight** – Uses Python, easy to write & scale.
✓ **Distributed Testing** – Can simulate thousands of users.
✓ **Real User Behavior** – Uses Python functions instead of complex scripts.
✓ **Web UI & CLI** – Monitor performance in real-time.

2 Installing Locust

First, install Locust using pip:

```
bash
CopyEdit
pip install locust
```

3 Writing Your First Locust Test

A Locust test is written as a Python script where you define **user behavior**.

 Basic Locust Test for a Website

```
python
CopyEdit
from locust import HttpUser, task, between

class MyWebsiteUser(HttpUser):
    wait_time = between(1, 3)  # Wait time between requests (1-3 sec)

    @task
    def home_page(self):
        self.client.get("/")  # Simulate a GET request to home page

    @task
    def about_page(self):
        self.client.get("/about")  # Simulate a GET request to About page
```

4 Running the Locust Test

Save the script as `locustfile.py` and run:

```
bash
CopyEdit
locust
```

Now, open `http://localhost:8089` in a browser to **start testing.**

5 Key Locust Concepts

Concept	Description
`HttpUser`	Represents a simulated user that makes HTTP requests
`@task`	Defines what actions the user performs
`self.client.get()`	Sends a GET request
`self.client.post()`	Sends a POST request
`wait_time = between(1, 3)`	Random wait between requests
`locust -f locustfile.py`	Starts the test

6 Simulating Multiple Users

Modify `locustfile.py` to set **concurrent users** and **spawn rate**:

```
bash
CopyEdit
locust --headless -u 100 -r 10 -t 1m
```

- `-u 100` → Simulate **100 users**
- `-r 10` → Spawn **10 users per second**
- `-t 1m` → Run for **1 minute**