SUGGESTED INTERVIEW QUESTIONS FOR AUTOMATION TESTING

# 1. What are the best candidates for the test automation? (which test cases you’ll automate first?)

* To get rid of most of manual efforts on automation, we utilize automation scripts that can run repeated and recurring tasks. So as aligned with the testing levels, and considering the build has excitingly constructed. After manual exploration, we would pick regression suite first from the existing build and then proceed with others.
* The business logic for that change would be the rank 2
* The challenging cases from the Suite would be rank 3.

# 2. Page object model and how do you implement it?

* It is the design pattern that emphasizes the maintainability and readability, in which the automation test scripts were organized such that each web elements and its actions for the particular page is maintained within the class that treated as the section or web page.
* We initially creating the project structure, in this case we have below,
* Tests
* Pages
* Utilities
* Conftest.py
* Requirement.txt
* Test\_config.yaml
* Then we define the classes under pages folder that explains the business logic of the web automation.
* Then we will write tests under /tests folder where we can invoke methods and classes from Pages classes.
* If required, we implement the logics for data validations and other external supporting features under Utilities folder
* The setup will be done on conftest.py which would suppose to be first step if we are implement newly

# 3. Distinguish Implicit and Explicit wait and how do we implement in python and Java.

* Implicit Wait sets a default waiting time for all elements in the WebDriver instance. If an element is not immediately found, the driver will keep trying to locate it until the specified timeout expires.
* Explicit Wait allows you to wait for a specific condition to be met for a particular element before proceeding. It is more flexible and precise compared to Implicit Wait.

|  |  |
| --- | --- |
| PyThon | Java |
| from selenium import webdriver  driver = webdriver.Chrome()  driver.implicitly\_wait(10)  # Waits up to 10 seconds for elements to appear  driver.get("https://example.com")  element = driver.find\_element\_by\_id("username") # Will wait up to 10 seconds if the element isn't immediately found  driver.quit() | from selenium import webdriver  from selenium.webdriver.common.by import By  from selenium.webdriver.support.ui import WebDriverWait  from selenium.webdriver.support import expected\_conditions as EC  driver = webdriver.Chrome()  driver.get("https://example.com")  wait = WebDriverWait(driver, 10)  # Wait up to 10 seconds for conditions  # Wait until the element is visible  element = wait.until(EC.visibility\_of\_element\_located((By.ID, "username")))  element.send\_keys("test\_user")  driver.quit() |
| import org.openqa.selenium.WebDriver;  import org.openqa.selenium.chrome.ChromeDriver;  public class ImplicitWaitExample {  public static void main(String[] args) {  System.setProperty("webdriver.chrome.driver", "path\_to\_chromedriver");  WebDriver driver = new ChromeDriver();    // Implicit wait  driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);  driver.get("https://example.com");  driver.findElement(By.id("username")).sendKeys("test\_user"); // Will wait up to 10 seconds  driver.quit();  }  } | import org.openqa.selenium.By;  import org.openqa.selenium.WebDriver;  import org.openqa.selenium.WebElement;  import org.openqa.selenium.chrome.ChromeDriver;  import org.openqa.selenium.support.ui.ExpectedConditions;  import org.openqa.selenium.support.ui.WebDriverWait;  public class ExplicitWaitExample {  public static void main(String[] args) {  System.setProperty("webdriver.chrome.driver", "path\_to\_chromedriver");  WebDriver driver = new ChromeDriver();  driver.get("https://example.com");    // Explicit wait  WebDriverWait wait = new WebDriverWait(driver, 10);  WebElement element = wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("username")));  element.sendKeys("test\_user");  driver.quit();  }  } |

Table . Implicit VS Explicit wait on Java and Python

# 4. How do you handle dynamic or constantly changing elements?

- Find the locator or path which doesn’t change over time such as static elements.

- Collab with dev team to identify the web elements and either figure out the web element how to locate it or ask them to implement that we need to compose static element for that web element.

- If possible, the team can share us the repo where we can identify the web element using hooks (however we didn’t try yet throughout my work experience but got to know from another team).

# 5. How do you organize your test cases to ensure the coverages and easy maintenance?

POM is most famous and also easily maintainable and readable design pattern we use to organize and prioritize the test cases.

To get tracked with which is automated or not, we utilize requirement traceability matrix that can easily figure out which is automated existing, what are the changes can be done on the script and which are need to implement the tests.

# 6. How do you integrate automated test cases into CI CD pipeline?

After installation, setup and integeration of relevant CI CD tools (Git-Actions and Jenkins in our case),

* 1. We create the Jenkins Pipeline Job
  2. We compose the Jenkins Pipeline script on Jenkins File, where each steps corresponding to the commands that can run tests automatically.
  3. Then we build triggers where we use Git Commit to commit the local changes to the respositroy
  4. Then we navigate to run the test whenever required and it will generate reports that can send to clients for update.

Using Git-Actions,

1. We create workflow under Github actions by generating YAML script
2. Compose the script of workflow that runs for every push or pull requests

# 7. How to Handle authentication and assassin management in your test automation?

Authentication and authorization can be managed in test automation using Python. The process involves simulating user authentication, securely managing credentials, and verifying that the proper access controls are enforced.

 **Basic Authentication**

* Often requires passing a username and password directly in the request header.
* Example with Python's requests library:

python

CopyEdit

import requests

url = "https://example.com/api"

response = requests.get(url, auth=("username", "password"))

print(response.status\_code)

 **Token-Based Authentication**

* Requires obtaining a token (e.g., JWT) via login and using it in subsequent requests.
* Example with requests:

python

CopyEdit

import requests

# Step 1: Get the token

auth\_url = "https://example.com/api/login"

login\_data = {"username": "user", "password": "pass"}

response = requests.post(auth\_url, json=login\_data)

token = response.json()["token"]

# Step 2: Use the token in requests

api\_url = "https://example.com/api/resource"

headers = {"Authorization": f"Bearer {token}"}

api\_response = requests.get(api\_url, headers=headers)

print(api\_response.status\_code)

 **OAuth 2.0 Authentication**

* Involves multiple steps, including redirecting to an authorization server and exchanging authorization codes for tokens.
* Libraries like requests-oauthlib can simplify this process.

python

CopyEdit

from requests\_oauthlib import OAuth2Session

client\_id = "your\_client\_id"

client\_secret = "your\_client\_secret"

authorization\_base\_url = "https://example.com/oauth/authorize"

token\_url = "https://example.com/oauth/token"

oauth = OAuth2Session(client\_id)

authorization\_url, state = oauth.authorization\_url(authorization\_base\_url)

print(f"Go to this URL to authorize: {authorization\_url}")

redirect\_response = input("Paste the full redirect URL here: ")

token = oauth.fetch\_token(token\_url, authorization\_response=redirect\_response, client\_secret=client\_secret)

print(f"Access token: {token}")

# 8. How to deal with Flaky Test cases?

 **Rerun Analysis**: Rerun the test multiple times to observe inconsistent results.

 **Test Result Trends**: Monitor test history in CI/CD tools (e.g., Jenkins, GitHub Actions) to identify unstable tests.

 **Error Patterns**: Look for common error messages or logs across failures.