**Alert:**

1. Simple Alert
2. Prompt Alert
3. Confirmation Alert\

**Simple Alert:**

A *simple alert* just has an***OK*** button on them. They are mainly used to display some information to the user. The very first alert on the test page is a simple alert. The following code will read the text from the *Alert* and then accept the alert.

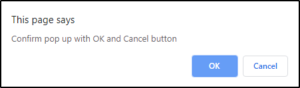


**Prompt Alert**:

In prompt alerts, you get an option to add a text field to the alert box. This is specifically used when some input is required from the user. You can use the ***sendKeys()***method to type some text in the Prompt alert box.



**Confirmation Alert**: This type of alert comes with an option to accept or dismiss the alert. In order to accept the alert, you can use the Alert.accept() and to dismiss, use Alert.dismiss()



* IAlert alert = driver.SwitchTo().Alert();

alert.Accept();

* IAlert alert = driver.SwitchTo().Alert();

alert.Dismiss();

* IAlert promptAlert = driver.SwitchTo().Alert();

promptAlert.SendKeys("Text to input");

* IAlert alert = driver.SwitchTo().Alert();

string alertText = alert.Text;

* // Replace "username" and "password" with actual credentials

IAlert authAlert = driver.SwitchTo().Alert();

authAlert.SetAuthenticationCredentials("username", "password");

* IAlert alert = driver.SwitchTo().Alert();

alert.AcceptAndClose();

* IAlert alert = driver.SwitchTo().Alert();

alert.DismissAndReturn();

* IAlert alert = driver.SwitchTo().Alert();

alert.AcceptAndReturn();

Windows:

List<string> windowHandles = new List<string>(driver.WindowHandles);

driver.SwitchTo().Window(windowHandles[1]);

driver.Navigate().GoToUrl("https://www.example2.com");

driver.SwitchTo().Window(windowHandles[0]);

// Now you're back on the first window/tab

driver.Navigate().GoToUrl("https://www.example.com");

**Get default window Handle:**

string currentWindowHandle = driver.CurrentWindowHandle;

driver.SwitchTo().Window(currentWindowHandle);

**Code to switch between two windows based on string:**

List<string> windowHandles = new List<string>(driver.WindowHandles);

string desiredWindowTitle = "Your Desired Window Title";

foreach (string handle in windowHandles)

{

driver.SwitchTo().Window(handle);

if (driver.Title == desiredWindowTitle)

{

}

}

**Dropdown list:**

using OpenQA.Selenium;

using OpenQA.Selenium.Chrome;

using System;

class Program

{

static void Main()

{

// Initialize the WebDriver (make sure to have the appropriate driver executable installed)

IWebDriver driver = new ChromeDriver();

// Navigate to a webpage with a custom dropdown

driver.Navigate().GoToUrl("https://example.com");

// Locate the custom dropdown and click it to expand the options

IWebElement customDropdown = driver.FindElement(By.Id("customDropdown"));

customDropdown.Click();

// Locate and click the desired option within the custom dropdown

IWebElement option = driver.FindElement(By.XPath("//div[@class='dropdown-option' and text()='Option Text']"));

option.Click();

}

}

**Normal Drop down with using select :**

<select name="dropdown">

<option value="1">Option 1</option>

<option value="2">Option 2</option>

<option value="3">Option 3</option>

</select>

IWebDriver driver = new ChromeDriver();

driver.Navigate().GoToUrl("https://example.com");

IWebElement dropdownElement = driver.FindElement(By.Id("dropdown-id"));

SelectElement select = new SelectElement(dropdownElement);

select.SelectByIndex(2); // By index

select.SelectByValue("option-value"); // By value

select.SelectByText("Option Text"); // By visible text

**Hover Over and Click**:

If the dropdown opens on hover, you can use Actions to hover over the dropdown trigger element and then click the desired option.

IWebElement dropdownTrigger = driver.FindElement(By.Id("dropdown-trigger"));

IWebElement optionElement = driver.FindElement(By.XPath("//div[@id='custom-dropdown']//div[text()='Option Text']"));

Actions actions = new Actions(driver);

actions.MoveToElement(dropdownTrigger).Click(optionElement).Perform()

**Keyboard Navigation**:

You can use Actions to simulate keyboard actions like arrow keys to navigate through the dropdown options.

IWebElement dropdown = driver.FindElement(By.Id("custom-dropdown"));

IWebElement optionElement = driver.FindElement(By.XPath("//div[@id='custom-dropdown']//div[text()='Option Text']"));

Actions actions = new Actions(driver);

actions.Click(dropdown).SendKeys(Keys.ArrowDown).SendKeys(Keys.Enter).Perform();

Mouse Actions:

**Clicking an Element:**

Actions actions = new Actions(driver);

IWebElement element = driver.FindElement(By.Id("element-id"));

actions.Click(element).Build().Perform();

**Double-Clicking an Element:**

Actions actions = new Actions(driver);

IWebElement element = driver.FindElement(By.Id("element-id"));

actions.DoubleClick(element).Build().Perform();

**Right-Clicking an Element (Context Click):**

Actions actions = new Actions(driver);

IWebElement element = driver.FindElement(By.Id("element-id"));

actions.ContextClick(element).Build().Perform();

**Hovering Over an Element:**

Actions actions = new Actions(driver);

IWebElement element = driver.FindElement(By.Id("element-id"));

actions.MoveToElement(element).Build().Perform();

**Dragging and Dropping an Element:**

Actions actions = new Actions(driver);

IWebElement sourceElement = driver.FindElement(By.Id("source-id"));

IWebElement targetElement = driver.FindElement(By.Id("target-id"));

actions.DragAndDrop(sourceElement, targetElement).Build().Perform();

* **Keyboard Actions:**

**Typing Text Using SendKeys:**

Actions actions = new Actions(driver);

IWebElement element = driver.FindElement(By.Id("element-id"));

actions.Click(element).SendKeys("Hello, World!").Build().Perform();

**Simulating Keyboard Shortcuts:**

Actions actions = new Actions(driver);

actions.KeyDown(Keys.Control).SendKeys("a").KeyUp(Keys.Control).Build().Perform();

* **Combining Actions:**

**Chaining Multiple Actions:**

Actions actions = new Actions(driver);

IWebElement element = driver.FindElement(By.Id("element-id"));

actions.MoveToElement(element).Click().SendKeys("Text").Build().Perform();

* **Advanced Interactions:**

**Click and Hold (e.g., for Drag-and-Drop):**

Actions actions = new Actions(driver);

IWebElement element = driver.FindElement(By.Id("element-id"));

actions.ClickAndHold(element).MoveByOffset(100, 0).Release().Build().Perform();

**Perform and Build of Action Class:**

**Build Method:**

The Build method is used to create a sequence of actions (an "action chain") by chaining together multiple actions. However, calling Build does not actually execute the actions; it prepares them for execution but doesn't perform them immediately.

**Perform Method:**

The Perform method is used to execute the sequence of actions that you've built using the Actions class. When you call Perform, it sends the entire sequence of actions to the browser, which then simulates those actions as if a user is interacting with the web page.

How do you locate and interact with a specific element (e.g., a button or checkbox) within a grid view using Selenium WebDriver in C#?

IWebDriver driver = new ChromeDriver();

// Navigate to the webpage with the grid view

driver.Navigate().GoToUrl("https://example.com");

// Locate the grid view

IWebElement gridView = driver.FindElement(By.Id("gridView"));

// Locate the row with Product ID "101"

IWebElement row = gridView.FindElement(By.XPath("//tr[td[text()='101']]"));

// Locate and click the "View" button in that row

IWebElement viewButton = row.FindElement(By.ClassName("btn-view"));

viewButton.Click();

**Boundary testing**

Boundary testing, also known as boundary value analysis or boundary testing technique, is a software testing method used to evaluate the behavior of an application or system at the boundaries of its input domain. The goal is to identify issues that may occur when input values are at the extreme limits, including the minimum and maximum values allowed.

Boundary testing is particularly valuable because many defects and unexpected behaviors often manifest at the edges of acceptable input ranges. By testing boundary values, you can uncover issues related to data validation, calculations, and system stability.

Here are some key aspects of boundary testing:

Boundary Conditions: Boundary testing focuses on testing values that are just inside, just outside, or precisely on the boundary conditions defined in the system's specifications. These conditions are often where problems arise.

Input Ranges: It applies to various types of input, including numerical values, date and time ranges, character strings, and any other input domains relevant to the application.

Minimum and Maximum Values: Testers explore the minimum and maximum values allowed by the system's design constraints. For example, if a text field allows between 1 and 100 characters, you'd test with inputs of 1 character, 100 characters, and 101 characters.

Edge Cases: Edge cases refer to values that are right on the edge of what's allowed. For example, if an age input field accepts values between 18 and 65, you'd test with values of 18, 19, 64, and 65.

Invalid Inputs: In addition to valid boundary values, you should test invalid inputs just beyond the boundary conditions. For example, for an age input field, test with negative values or values greater than 65 (assuming those are invalid inputs).

Error Handling: Pay attention to how the system handles boundary conditions. Does it reject invalid inputs appropriately? Does it produce accurate results for valid boundary inputs?

Documentation Review: Review the system's specifications and requirements to identify the boundary conditions explicitly defined by the project. These will guide your testing efforts.

Test Data Generation: Generate test data for both valid and invalid boundary conditions. Testers often use both manual data entry and automated testing tools to cover a wide range of cases.

Combinations: Sometimes, boundary conditions intersect. For example, if you have two fields for minimum and maximum values, you might test the scenario where the minimum value is greater than the maximum (an invalid combination).

Regression Testing: Include boundary tests as part of your regression testing suite to ensure that changes to the application don't introduce new boundary-related issues.

Boundary testing is applicable in various domains, including numeric inputs, date and time calculations, input field lengths, file size limits, and more. It's an essential technique for ensuring the robustness and reliability of software systems.

**Equivalence partitioning**

It is a software testing technique that involves dividing the input space of a system or component into groups of equivalent or similar input values. The idea is to test representative values from each partition to ensure that the software behaves consistently within each group. This technique helps reduce the number of test cases while still providing good coverage of the input space.

The key concept in equivalence partitioning is that if one test case from a partition passes, it is assumed that all other test cases within that partition will behave the same way, and there is no need to test them individually.

Here's an example of equivalence partitioning:

Scenario: Consider a simple login page for a web application. The application requires users to enter a username and password to log in.

Requirements:

Usernames should be between 6 and 12 characters long.

Passwords should be at least 8 characters long.

Both username and password fields are mandatory.

In this scenario, we can identify several equivalence partitions for testing:

Valid Usernames:

This partition includes valid usernames that are between 6 and 12 characters long.

Equivalence Class 1:

* Valid usernames (e.g., "user123", "john\_doe")
* Invalid Usernames: This partition includes usernames that do not meet the length requirement.

Equivalence Class 2:

* Usernames with fewer than 6 characters (e.g., "user")

Equivalence Class 3:

* Usernames with more than 12 characters (e.g., "very\_long\_username")

Valid Passwords:

This partition includes valid passwords that are at least 8 characters long.

Equivalence Class 4:

* Valid passwords (e.g., "password123", "securePwd")
* Invalid Passwords: This partition includes passwords that do not meet the length requirement.

Equivalence Class 5:

1. Passwords with fewer than 8 characters (e.g., "pwd123")

Equivalence Class 6:

1. Empty passwords (e.g., "")

Now, we can design test cases to cover each equivalence class:

Test Case 1: Valid username and valid password (e.g., "user123", "password123")

Test Case 2: Valid username and invalid password (e.g., "user123", "pwd")

Test Case 3: Invalid username and valid password (e.g., "user", "securePwd")

Test Case 4: Invalid username and invalid password (e.g., "very\_long\_username", "pwd")

Test Case 5: Valid username and empty password (e.g., "john\_doe", "")

Test Case 6: Empty username and valid password ("", "securePwd")

Test Case 7: Empty username and empty password ("", "")

Certainly, let's map equivalence classes to test cases in detail. Equivalence classes represent groups of input values that should behave similarly. Test cases are designed to cover each equivalence class, ensuring that you test various scenarios efficiently. In this example, we'll focus on mapping equivalence classes for a login form to specific test cases.

Equivalence Classes for Username Field:

Equivalence Class 1 (Valid Usernames):

Representative Values: "user123", "john\_doe"

These are valid usernames within the specified character length range.

Equivalence Class 2 (Invalid Usernames, less than 6 characters):

Representative Values: "abc", "x"

These usernames do not meet the minimum length requirement.

Equivalence Class 3 (Invalid Usernames, more than 12 characters):

Representative Values: "very\_long\_username", "thisisareallylongusername"

These usernames exceed the maximum length allowed.

Equivalence Classes for Password Field:

Equivalence Class 4 (Valid Passwords):

Representative Values: "password123", "securePwd"

These are valid passwords meeting the minimum character length requirement.

Equivalence Class 5 (Invalid Passwords, less than 8 characters):

Representative Values: "pwd123", "1234567"

These passwords do not meet the minimum length requirement.

Mapping Equivalence Classes to Test Cases:

Now, let's map each equivalence class to specific test cases based on the representative values:

Test Cases for Equivalence Class 1 (Valid Usernames):

Test Case 1: Valid username and valid password

Username: "user123"

Password: "password123"

Test Case 2: Valid username and valid but short password

Username: "user123"

Password: "pwd123"

Test Case 3: Valid username and invalid, very long password

Username: "user123"

Password: "thisisareallylongpassword"

Test Cases for Equivalence Class 2 (Invalid Usernames, less than 6 characters):

Test Case 4: Invalid short username and valid password

Username: "abc"

Password: "securePwd"

Test Case 5: Invalid short username and invalid short password

Username: "x"

Password: "x"

Test Cases for Equivalence Class 3 (Invalid Usernames, more than 12 characters):

Test Case 6: Invalid very long username and valid password

Username: "very\_long\_username"

Password: "securePwd"

Test Cases for Equivalence Class 4 (Valid Passwords):

Test Case 7: Valid username and empty password

Username: "user123"

Password: ""

Test Case 8: Empty username and valid password

Username: ""

Password: "securePwd"

Test Case 9: Empty username and empty password

Username: ""

Password: ""

Test Cases for Equivalence Class 5 (Invalid Passwords, less than 8 characters):

Test Case 10: Valid username and invalid short password

Username: "john\_doe"

Password: "x"

These test cases are designed to cover each equivalence class, ensuring that you test various scenarios efficiently without having to test every possible username and password combination individually. Each test case represents a specific scenario that helps verify the behavior of the login form based on the defined equivalence classes.

**Decision table:**

Let's explore a practical example of decision table testing in the context of a simple online shopping cart system. In this scenario, the decision to be tested is whether or not to apply a discount to a customer's purchase based on various conditions.

Decision: Apply a discount to a customer's purchase.

Conditions:

Customer Type (Regular, Premium)

Total Purchase Amount

Special Promotion (Yes, No)

Outcomes:

Apply a 10% discount.

Apply a 5% discount.

No discount applied.

Decision Table:

Customer Type Purchase Amount Promotion Outcome

Regular $100 or more Yes Apply 10% discount

Regular $100 or more No Apply 5% discount

Regular Less than $100 Yes Apply 5% discount

Regular Less than $100 No No discount applied

Premium $100 or more Yes Apply 10% discount

Premium $100 or more No Apply 10% discount

Premium Less than $100 Yes Apply 5% discount

Premium Less than $100 No No discount applied

Generated Test Cases:

Test Case 1: Regular customer, total purchase amount $120, special promotion: Yes.

Expected Outcome: Apply a 10% discount.

Test Case 2: Regular customer, total purchase amount $120, special promotion: No.

Expected Outcome: Apply a 5% discount.

Test Case 3: Regular customer, total purchase amount $80, special promotion: Yes.

Expected Outcome: Apply a 5% discount.

Test Case 4: Regular customer, total purchase amount $80, special promotion: No.

Expected Outcome: No discount applied.

Test Case 5: Premium customer, total purchase amount $120, special promotion: Yes.

Expected Outcome: Apply a 10% discount.

Test Case 6: Premium customer, total purchase amount $120, special promotion: No.

Expected Outcome: Apply a 10% discount.

Test Case 7: Premium customer, total purchase amount $80, special promotion: Yes.

Expected Outcome: Apply a 5% discount.

Test Case 8: Premium customer, total purchase amount $80, special promotion: No.

Expected Outcome: No discount applied.

By using decision table testing, you systematically cover various combinations of customer types, purchase amounts, and special promotions, ensuring that the software correctly applies discounts according to the specified conditions. This approach helps achieve comprehensive test coverage while managing complexity.

**Synchronization in Selenium with C#:**

Synchronization in Selenium refers to the techniques and strategies used to ensure that automated tests interact with web elements and the web application in a stable and reliable manner, regardless of the variations in web page loading times, element rendering, or server responses. It's crucial because web applications are dynamic, and elements may load or change at different speeds.

Why Synchronization is Important:

Varying Page Load Times: Web pages may load at different speeds based on factors like network latency, server load, or complex JavaScript rendering. Without synchronization, tests may attempt to interact with elements before they are fully loaded, leading to test failures.

Asynchronous Operations: Modern web applications often use asynchronous techniques, such as AJAX requests, to load data or update parts of a page. Synchronization is necessary to wait for these asynchronous operations to complete.

Dynamic Content: Elements on a web page may appear, disappear, or change dynamically based on user interactions or data updates. Synchronization helps tests adapt to such changes.

How to Achieve Synchronization in Selenium with C#:

**Implicit Waits:**

Implicit waits are set at the driver level and apply globally to all find element commands.They instruct the WebDriver to wait for a certain amount of time before throwing a NoSuchElementException if the element is not found.

Example:

driver.Manage().Timeouts().ImplicitWait = TimeSpan.FromSeconds(10);

**Explicit Waits:**

Explicit waits allow you to wait for a specific condition to be met before proceeding with the test.You can use conditions like ExpectedConditions.ElementIsVisible,ExpectedConditions.ElementToBeClickable, etc., to wait for specific element states.

Example

WebDriverWait wait = new WebDriverWait(driver, TimeSpan.FromSeconds(10));

IWebElement element = wait.Until(ExpectedConditions.ElementIsVisible(By.Id("element-id")));

**Fluent Waits:**

Fluent waits combine implicit and explicit waits to provide more fine-grained control.

They allow you to define the polling interval and timeout duration, making them flexible for dynamic scenarios.

Example:

DefaultWait<IWebDriver> fluentWait = new DefaultWait<IWebDriver>(driver);

fluentWait.Timeout = TimeSpan.FromSeconds(30);

fluentWait.PollingInterval = TimeSpan.FromMilliseconds(500);

fluentWait.IgnoreExceptionTypes(typeof(NoSuchElementException));

IWebElement element = fluentWait.Until(ExpectedConditions.ElementIsVisible(By.Id("element-id")));

**Page Load Timeout:**

driver.Manage().Timeouts().PageLoad = TimeSpan.FromSeconds(30);

The choice between using explicit waits or fluent waits in Selenium depends on the specific requirements and behavior of your web application and test scenarios. Here are some guidelines to help you decide which one to use:

**When to USE :**

**Use Explicit Waits When:**

You Have a Specific Condition to Check: If you know exactly what condition you need to wait for, such as an element becoming clickable or visible, explicit waits are a straightforward choice. You can specify the condition directly.

Fixed and Predictable Timing: When the timing of an element's appearance or change is fixed and predictable, explicit waits with a static timeout are effective. For example, waiting for a login button to become clickable after clicking a "Sign In" link.

Simple Waiting Scenarios: If your synchronization needs are straightforward and do not involve complex or dynamic behavior, explicit waits are easier to set up and use.

**Use Fluent Waits When:**

Conditions Are Gradual or Variable: When elements may appear, change, or become available gradually or with varying timings, fluent waits are more appropriate. You can set a dynamic timeout and polling interval to handle these situations.

Non-Blocking Waits: Fluent waits are non-blocking, meaning they allow your test script to continue execution during the wait, checking the condition periodically. This is useful for scenarios where you want to multitask while waiting.

Fine-Tuned Synchronization: Fluent waits provide more control over synchronization, making them suitable for scenarios where precise control over the waiting process is needed. You can adjust the polling interval to match the behavior of the application.

Handling Dynamic Elements: If your application involves dynamic elements or AJAX requests, fluent waits are often the better choice because they allow you to wait for elements to appear or change over time.

In many cases, a combination of both explicit and fluent waits may be used within the same test suite, depending on the specific scenarios you encounter. You can choose the synchronization approach that best fits each individual test case.

Additionally, consider the maintainability of your tests. Explicit waits with clear and descriptive conditions may be easier to read and understand for simple scenarios, while fluent waits provide more control for complex synchronization needs. The key is to select the synchronization method that aligns with your application's behavior and the timing requirements of your tests.

**Table in Gherkin Language:**

**Creating a Table Object**:

**[Given(@"the following products are added to the shopping cart:")]**

**public void GivenTheFollowingProductsAreAddedToTheShoppingCart(Table table)**

**{**

**}**

**Accessing Rows and Columns:**

You can access the rows and columns of the table using the following methods:

table.Rows: Returns a collection of TableRow objects, where each TableRow represents a row in the table.

table.HeaderRow: Returns the header row of the table as a TableRow object.

**Iterating Through Rows:**

foreach (var row in table.Rows)

{

var productName = row["Product"]; // Access data in the "Product" column

var quantity = row["Quantity"]; // Access data in the "Quantity" column

// Your code here

}

**Converting to a Collection:**

var products = table.CreateSet<Product>();

**Converting to a Dictionary:**

var dataDictionary = table.CreateDynamicSet();

Given some step with a table:

| Header1 | Header2 | Header3 |

| Data1 | Data2 | Data3 |

| Data4 | Data5 | Data6 |

| Data7 | Data8 | Data9 |

In this format:

Each row is enclosed within vertical bars (|).

***The first row is often used as the header row and contains column names or labels (e.g., Header1, Header2, Header3).***

Subsequent rows contain data for the corresponding columns (e.g., Data1, Data2, Data3).

**why we have (.\*) for step definition in c# selenium**

In C# with Selenium and SpecFlow (or other BDD frameworks), the (.\*) is a regular expression pattern used in step definition methods to capture dynamic values from the Gherkin scenarios. It's a way to make step definitions more flexible and reusable because it allows you to match different values for a specific part of the step.

Here's how it works:

Regular Expression (Regex): (.\*) is a regex pattern. In regex, . matches any character, and \* means "zero or more occurrences." So, (.\*) matches any sequence of characters.

Parameter Capturing: When you use (.\*) in a step definition, it captures the part of the step that matches this pattern. For example:

[Given(@"the user searches for '(.\*)'")]

public void GivenTheUserSearchesFor(string searchTerm)

{

// 'searchTerm' will capture the value provided in the Gherkin step

}

In the above step definition, if your Gherkin step is "Given the user searches for 'laptop'," the value "laptop" will be captured as the searchTerm parameter.

Flexibility: Using (.\*) allows you to match various inputs for the same step, making your step definitions versatile. You can reuse the same step definition for different scenarios by capturing and using the dynamic values provided in the Gherkin steps.

Parameter Passing: The captured values can be passed as arguments to methods or used for validation and interactions with your application. This makes your automation code more adaptable to different scenarios.

Here's a summary of how (.\*) is used:

(.\*) is a regex pattern used to capture dynamic values.

It allows you to make your step definitions more flexible and reusable.Captured values can be used as method parameters in your step definition code.It's a common practice in BDD frameworks like SpecFlow to handle varying input values in Gherkin scenarios.

**When someone asks about the framework used in your C# Selenium project**

**Modularity, Abstraction, POM, Test Data Management, Version Control and Collaboration**:

**Modularity:**

Example: In an e-commerce testing framework, you have separate modules or classes for different pages like HomePage, ProductPage, CartPage, and CheckoutPage. Each module contains methods and locators related to that specific page. This modularity allows you to update or add tests for individual pages without affecting others.

**Abstraction or POM:**

public class LoginPage

{

private IWebDriver driver;

public LoginPage(IWebDriver driver)

{

this.driver = driver;

}

public void NavigateToLoginPage(string url)

{

driver.Navigate().GoToUrl(url);

}

public void Login(string username, string password)

{

driver.FindElement(By.Id("username")).SendKeys(username);

driver.FindElement(By.Id("password")).SendKeys(password);

driver.FindElement(By.CssSelector("button[type='submit']")).Click();

}

public bool IsLoggedIn()

{

WebDriverWait wait = new WebDriverWait(driver, TimeSpan.FromSeconds(10));

return wait.Until(d => d.Url.Contains("/dashboard"));

}

}

IWebDriver driver = new ChromeDriver();

LoginPage loginPage = new LoginPage(driver);

loginPage.NavigateToLoginPage("https://example.com/login");

loginPage.Login("myusername", "mypassword");

Assert.IsTrue(loginPage.IsLoggedIn());

**Configuration management:**

In a real-world Selenium test automation framework, configuration management plays a crucial role in managing various settings and parameters needed for your tests. Here's an example of how you can implement configuration management with different types of configurations in a C# Selenium framework.

**Scenario:** Let's assume you're building an automation framework for testing an e-commerce website. You need to manage configurations related to test environments (e.g., URLs, browser configurations), test data, and timeouts.

**Types of Configurations:**

Environment Configuration: This includes settings like URLs for different test environments (e.g., development, staging, production).

Browser Configuration: Configure the browser and its options (e.g., headless mode, browser type).

Test Data Configuration: Manage test data (e.g., usernames, passwords) that your tests will use.

Timeout Configuration: Define timeouts for various actions like page loads, element waits, and script executions.

|  |  |
| --- | --- |
| **Environment Configuration** | **{**  **"Environment": {**  **"BaseUrl": "https://example.com",**  **"EnvironmentType": "staging"**  **}**  **}** |
| **Browser Configuration** | **{**  **"Browser": {**  **"BrowserType": "chrome",**  **"HeadlessMode": false**  **}**  **}** |
| **Test Data Configuration** | **{**  **"UserData": {**  **"ValidUser": {**  **"Username": "testuser",**  **"Password": "password123"**  **},**  **"InvalidUser": {**  **"Username": "invaliduser",**  **"Password": "invalidpassword"**  **}**  **}**  **}** |
| **Timeout Configuration** | **{**  **"Timeouts": {**  **"ElementWaitTimeout": 10,**  **"PageLoadTimeout": 30,**  **"ScriptTimeout": 15**  **}**  **}** |
|  |  |

using Microsoft.Extensions.Configuration;

public class ConfigurationManager

{

private readonly IConfiguration \_config;

public ConfigurationManager()

{

var builder = new ConfigurationBuilder()

.SetBasePath(Directory.GetCurrentDirectory())

.AddJsonFile("appsettings.json");

\_config = builder.Build();

}

public string GetBaseUrl()

{

return \_config.GetSection("Environment")["BaseUrl"];

}

public string GetEnvironmentType()

{

return \_config.GetSection("Environment")["EnvironmentType"];

}

public string GetBrowserType()

{

return \_config.GetSection("Browser")["BrowserType"];

}

public string GetUserData(string userType, string field)

{

return \_config.GetSection("UserData")[userType][field];

}

public int GetElementWaitTimeout()

{

return int.Parse(\_config.GetSection("Timeouts")["ElementWaitTimeout"]);

}

public int GetPageLoadTimeout()

{

return int.Parse(\_config.GetSection("Timeouts")["PageLoadTimeout"]);

}

public int GetScriptTimeout()

{

return int.Parse(\_config.GetSection("Timeouts")["ScriptTimeout"]);

}

}

// Initialize ConfigurationManager

var configManager = new ConfigurationManager();

// Use configurations in your tests

string baseUrl = configManager.GetBaseUrl();

string browserType = configManager.GetBrowserType();

bool headlessMode = configManager.IsHeadlessModeEnabled();

string validUsername = configManager.GetUserData("ValidUser", "Username");

string validPassword = configManager.GetUserData("ValidUser", "Password");

int elementWaitTimeout = configManager.GetElementWaitTimeout();

// Use these values to set up your Selenium WebDriver, perform actions, and make assertions.

using OpenQA.Selenium;

using OpenQA.Selenium.Support.UI;

public class LoginPage

{

private IWebDriver \_driver;

private WebDriverWait \_wait;

private ConfigurationManager \_configManager;

public LoginPage(IWebDriver driver)

{

\_driver = driver;

\_wait = new WebDriverWait(driver, TimeSpan.FromSeconds(10));

\_configManager = new ConfigurationManager();

}

public void NavigateToLoginPage()

{

\_driver.Navigate().GoToUrl(\_configManager.GetBaseUrl());

}

public void EnterUsername(string username)

{

\_driver.FindElement(By.Id("username")).SendKeys(username);

}

public void EnterPassword(string password)

{

\_driver.FindElement(By.Id("password")).SendKeys(password);

}

public void ClickLoginButton()

{

\_driver.FindElement(By.Id("login-button")).Click();

}

}