Automated testing: API Automation

API is the acronym for application programming interface — a software intermediary that allows two applications to talk to each other. APIs are an accessible way to extract and share data within and across organizations.

API testing is basically black box testing which is simply concerned with the final output of the system under test. 1. Unit testing aims to verify whether the module delivers the required functionality. The development team monitors unit testing activity and makes necessary changes wherever required.

Response Codes (200, 404, 500, etc.)
1XX**Informational** – : Communicates transfer protocol-level information.
2XX **Success** – :Indicates that the client's request was accepted successfully.
3XX**Redirection** – :Indicates that the client must take some additional action in order to complete their request.
4XX**Client Error** – : This category of error status codes points the finger at clients.
5XX **Server Error** –:The server takes responsibility for these error status codes.

Http Get method

Used to request data from a specified resource.

```
@Test
    public void anotherget(){
        String response = given()
        .pathParam("id", 21)   // Step 3: set path parameter "id" to 21
        .when()                // Step 4: start defining the request
        .get("https://petstore.swagger.io/v2/pet/{id}") // Step 5: send a GET request to the specified
URL
        .then()                 // Step 6: start validating the response
        .statusCode(200)       // Step 7: assert that the HTTP status code is 200
        .body("id", equalTo(21)) // Step 8: assert that the JSON response contains an "id" field with a
value of 21
        .extract()             // Step 9: extract the response after validations
        .asPrettyString();
        System.out.println(response);
    }
```

HTTPS Post Method

Used to send data to create a new resource on the server.

```
@Test
public void createPet() {
    // Define the JSON body
    String newPet = "{¥n" +
            "    ¥"id¥": 21,¥n" +
            "    ¥"category¥": {¥n" +
            "        ¥"id¥": 0,¥n" +
            "        ¥"name¥": ¥"string¥"¥n" +
            "    },¥n" +
            "    ¥"name¥": ¥"doggie¥",¥n" +
            "    ¥"photoUrls¥": [¥n" +
            "        ¥"string¥"¥n" +
            "    ],¥n" +
            "    ¥"tags¥": [¥n" +
            "        {¥n" +
            "            ¥"id¥": 0,¥n" +
            "            ¥"name¥": ¥"string¥"¥n" +
            "        }¥n" +
            "    ],¥n" +
            "    ¥"status¥": ¥"available¥"¥n" +
            "}";

    // Send the POST request
    String response = given()
            .header("Content-Type", "application/json")  // Specify that the request body is JSON
            .body(newPet)  // Attach the JSON body to the request
            .when()
            .post("https://petstore.swagger.io/v2/pet")  // Send a POST request to create a new pet
            .then()
            .statusCode(200)  // Assert that the response status code is 200 (OK)
            .body("id", equalTo(21))  // Assert that the returned pet ID is 21
            .extract()
            .asPrettyString();  // Extract the response and format it as a pretty string

    // Print the response
    System.out.println(response);
}
```

HTTP Put Method

Used to update an existing resource.

SO when ever we need any update in the details then we use the Put Method.

- Use `POST` when you want to **add** a new resource.
- Use `PUT` when you want to **update** an existing resource (or create one at a specific URI).

The code of put and post is almost same, the only difference is that we got to use PUT isnstead of POST method.

HTTP Delete Method

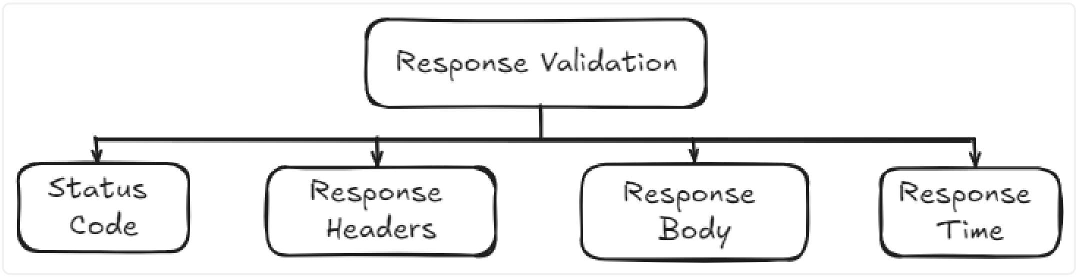
Delete is used to delete any existing data from the data set.

```
Response response = given()
        .pathParam("id",21)
        .when()
        .delete("https://petstore.swagger.io/v2/pet/{id}")
        .then()
        .statusCode(200)
        .log().all()
        .extract().response();
```

Making API requests involves sending data to or retrieving data from a server, which can be automated using RestAssured in Java. By mastering the use of methods like GET, POST, PUT, and DELETE, and learning to validate responses, you can build robust and reliable API tests.

Validating the responce in RestAssured FrameWork

Validating API responses is a crucial part of automated API testing, ensuring that the server returns the expected output based on the request sent.



Validating Status Code:

Status codes are the most basic form of validation. Every HTTP request returns a specific status code to indicate success, failure, or errors.

```
import io.restassured.RestAssured;
import static io.restassured.RestAssured.*;

public class ValidateStatusCode {

    public static void main(String[] args) {
        // Base URI of the API
        RestAssured.baseURI = "https://api.example.com";

        // Validate status code is 200 OK
        given()
            .when()
                .get("/students")
            .then()
                .statusCode(200)  // Validate status code
                .log().all();     // Log the response
    }
}
```

`.statusCode(200)`: This checks that the response status code is `200 OK`

Common Status Codes:

- `200 OK`: Request was successful.
- `201 Created`: Resource was created successfully (common for POST requests).
- `404 Not Found`: The requested resource does not exist.
- `500 Internal Server Error`: A server error occurred.

Validating Response Headers

Response headers provide important information about the response, such as content type, cache control, and authorization status.

```
import io.restassured.RestAssured;
import static io.restassured.RestAssured.*;

public class ValidateHeaders {

    public static void main(String[] args) {
        // Base URI of the API
        RestAssured.baseURI = "https://api.example.com";

        // Validate specific headers in the response
        given()
            .when()
                .get("/students/1")
            .then()
                .statusCode(200)
                .header("Content-Type", "application/json")  // Validate Content-Type header
                .header("Cache-Control", "no-cache")         // Validate Cache-Control header
                .log().all();
    }
}
```

Validating Response Body

The response body is the most important part of API validation, especially when working with JSON or XML data. We typically validate that the data returned matches our expectations.

so suppose we have a response for a specific request :

```
{
    "id": 1,
    "name": "John Doe",
    "age": 20,
    "class": "12th Grade"
}
```

Now we want to validate is the response is correct or not then,

```
import io.restassured.RestAssured;
import static io.restassured.RestAssured.*;
import static org.hamcrest.Matchers.*;

public class ValidateJsonResponse {

    public static void main(String[] args) {
        // Base URI of the API
        RestAssured.baseURI = "https://api.example.com";

        // Validate response body content
        given()
            .when()
                .get("/students/1")
            .then()
                .statusCode(200)
                .body("id", equalTo(1))               // Validate student ID is 1
                .body("name", equalTo("John Doe"))    // Validate name
                .body("age", equalTo(20))             // Validate age
                .body("class", equalTo("12th Grade")) // Validate class
                .log().all();                         // Log the entire response
    }
}
```

and when we have any nested response like

```
{
    "id": 1,
    "name": "John Doe",
    "age": 20,
    "address": {
        "city": "New York",
        "zip": "10001"
    }
}
```

So now as we can seee address have two responses to look for city and Zip, so now what we would do is

```
.statusCode(200)
.body("address.city", equalTo("New York"))  // Validate city in nested object
.body("address.zip", equalTo("10001"));     // Validate zip code
```

Validating Response Time

API performance is crucial, and we often need to ensure that the response time is within acceptable limits.

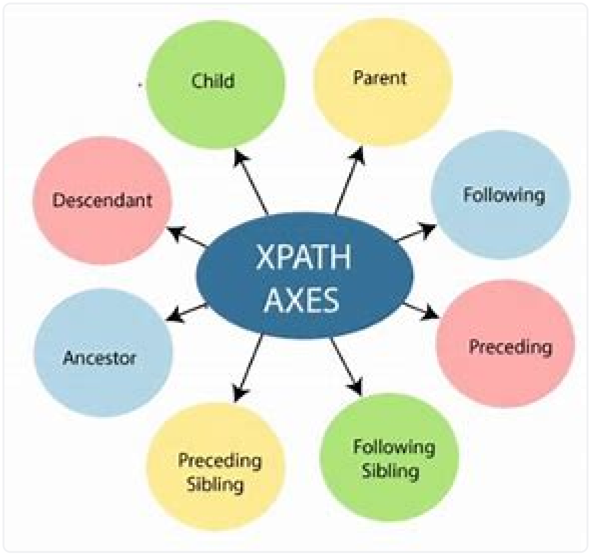
```
import io.restassured.RestAssured;
import static io.restassured.RestAssured.*;

public class ValidateResponseTime {

    public static void main(String[] args) {
        // Base URI of the API
        RestAssured.baseURI = "https://api.example.com";

        // Validate that the response time is less than 2000 ms (2 seconds)
        given()
            .when()
                .get("/students")
            .then()
                .statusCode(200)
                .time(lessThan(2000L))  // Validate response time in milliseconds
```

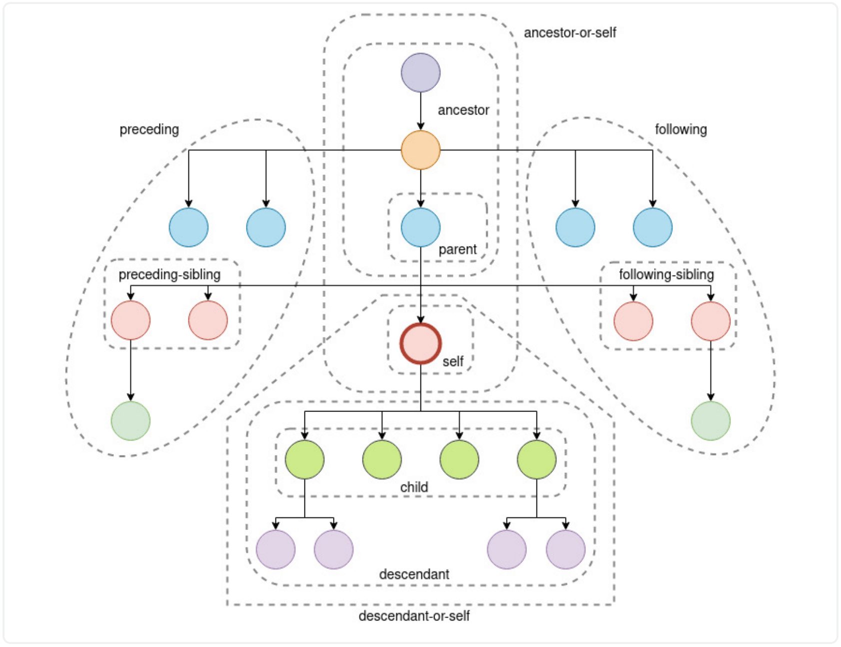
```
                    .log().all();
        }
    }
```

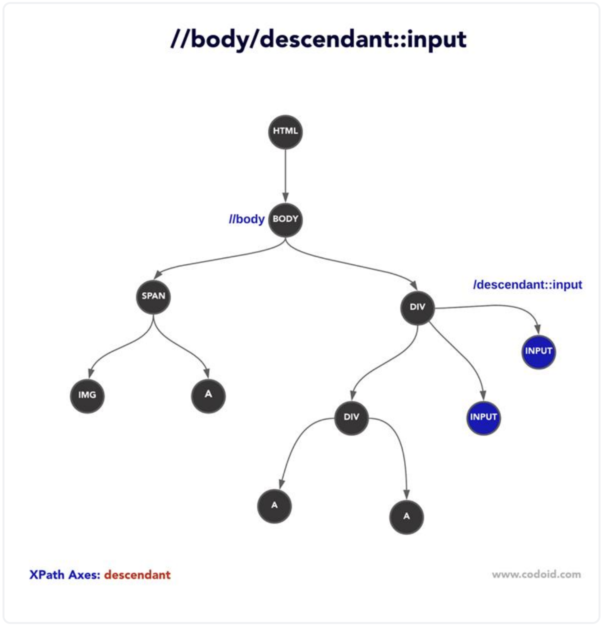
This code should be executed in or before 2000 milliseconds, or else it would fail the test

Dynamic X path

In Selenium, a **Dynamic XPath** refers to an XPath expression that is constructed to handle web elements whose attributes or positions may change dynamically during runtime. This is common in modern web applications where elements often have dynamic IDs, classes, or are generated dynamically.







Design Patterns

Singelton Design Pattern

Its ensure that only One instance should be created so that we can not created multiple instance for better performance.

The most important three rules to follow

- Private Constructor
- Static Member
- Static Method

- [ ] modifiy the code in terms of singleton Desing Pattern

Types of Design Patterns

- Factory(Creational Design Patterns)

  as the factory function, Factory Implementation focuses on making and using the functionalities in a way that the logic is not hold the value of transparency.
- Singleton
- Builder Design Pattern

  In the same sense as of building, the builder Design Pattern focuses on building/initializing the methodologies and function

builder

factory

singleton

Pom

Page Factory

refactoring.guru

Untitled

Selenium WebDriver

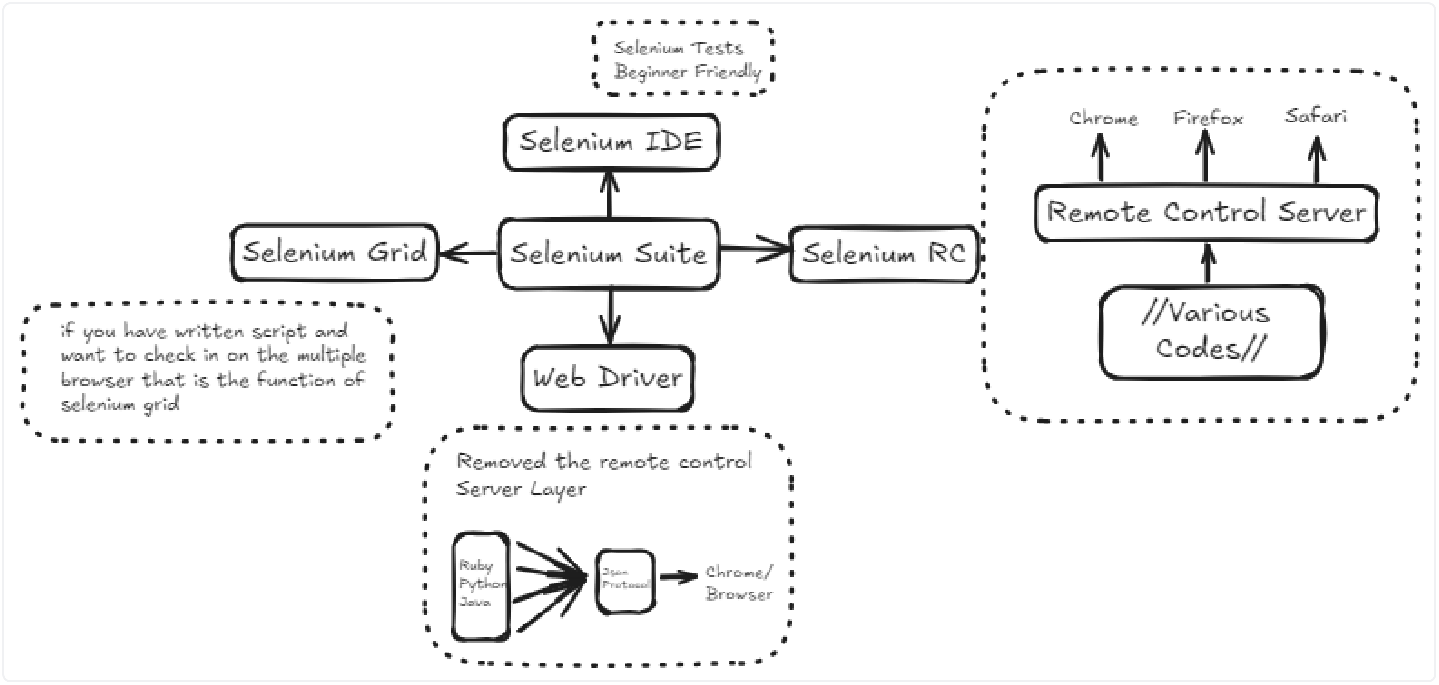
1. Selenium WebDriver

Selenium is Used to automate the UI Testing

We can also integrate selenium with different IDES.

Advantages of Selenium:

- Web Based Automation Tool
- Pen Source
- UI Automation
- Support Multiple OS
- Multiple Browsers
- Multiple languages



Selenium Tests Beginner Friendly
Selenium IDE
Selenium Grid
Selenium Suite
Selenium RC
Web Driver
if you have written script and want to check in on the multiple browser that is the function of selenium grid
Chrome
Firefox
Safari
Remote Control Server
//Various Codes//
Removed the remote control Server Layer
Ruby Python Java
Json Protocol
Chrome/ Browser

Everything Selenium does is send the browser commands to do something or send requests for information. Most of what you'll do with Selenium is a combination of these basic commands

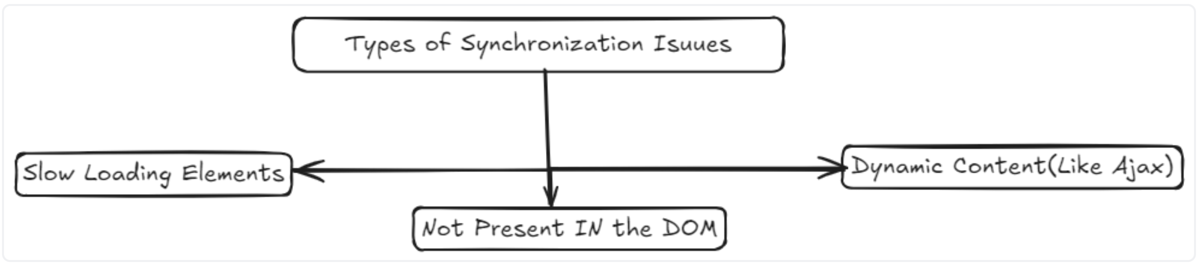
A very Basic Selenium Code

```
private WebDriver driver; //defining private so can use further
@BeforeEach
public void setup(){

System.setProperty("webdriver.chrome.driver","C:¥¥Users¥¥prabhatritesh_chaube¥¥Downloads¥¥chromedriver-
win64¥¥chromedriver.exe");
    driver = new ChromeDriver();
    driver.manage().window().maximize();
}

@Test
    public void selectone(){
        driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(20));
        driver.get("https://app.endtest.io/guides/docs/how-to-test-dropdowns/");
        WebElement select = driver.findElement(By.id("pets"));
        Select select1 = new Select(select);
        select1.selectByValue("rabbitt");
//        select1.selectByVisibleText("Rabbit");
//        select1.selectByIndex(5);

    }
 @AfterEach
    public void exit(){
        driver.quit();
    }
}
```

2. Synchronization

What are Synchronization Issues?

Synchronization issues arise when the WebDriver tries to interact with elements before they are ready (e.g., before they are loaded or clickable). These issues lead to flaky tests and frequent failures.

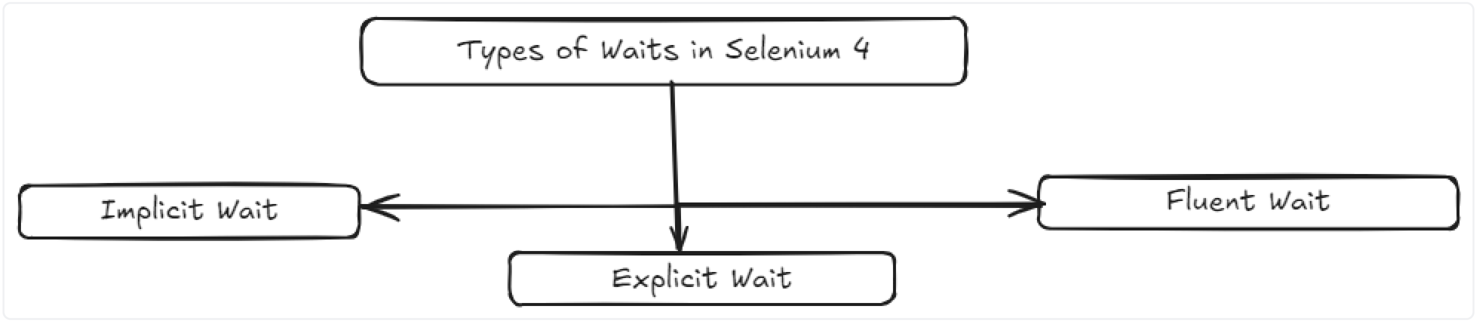


To ensure the WebDriver interacts with elements when they are in the correct state (loaded, clickable, etc.).

Common Solutions:

- General timeouts (implicit waits)
- Explicit waits
- Fluent waits
- Custom conditions

Waits



1.Implicit Wait

- Implicit Wait is used to set a default waiting time throughout your Selenium script. When you use implicit wait, it tells the WebDriver to wait for a certain amount of time before it throws a **NoSuchElementException**.

```
driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(20));
```

2.Explicit Wait

- Explicit Wait is used to wait for a specific condition to be true before proceeding further. It allows you to wait for a particular element or condition with a specific timeout.
- **Use Case:** Use explicit wait when you need to wait for a specific condition to occur before proceeding with the next step in the test script

```
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) {
        WebDriver driver = new ChromeDriver();
        driver.get(”https://example.com”);

        WebDriverWait wait = new WebDriverWait(driver, 20);
        WebElement element
=wait.until(ExpectedConditions.visibilityOfElementLocated(By.id(”exampleId”)));

        // Perform actions on the element
        element.click();

        driver.quit();
    }
}
```

3.Fluent Wait

- Fluent Wait is similar to explicit wait but with more flexibility. You can define the maximum amount of time to wait for a condition, as well as the frequency with which to check the condition, and you can also ignore specific types of exceptions while waiting.
- **Use Case**: Use fluent wait when you need more control over the wait conditions, such as checking for the condition at regular intervals and ignoring certain exceptions.

```
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.FluentWait;
import java.time.Duration;
import java.util.NoSuchElementException;

public class FluentWaitExample {
    public static void main(String[] args) {
        WebDriver driver = new ChromeDriver();
        driver.get(”https://example.com”);

        FluentWait<WebDriver> wait = new FluentWait<>(driver)
                .withTimeout(Duration.ofSeconds(30))
                .pollingEvery(Duration.ofSeconds(5))
                .ignoring(NoSuchElementException.class);

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

        // Perform actions on the element
        element.click();

        driver.quit();
    }
}
```

As we have all the wait types, why cant we just use the **implicitlyWait** all the time?

Limitations of implicitlyWait:

- It applies globally and cannot target specific conditions.
- It might not be efficient in scenarios with dynamic content.
- It could cause unnecessary waiting in situations where the element is ready sooner.

Implicit Wait	**Explicit Wait**
Global wait applied to all element searches.	Waits for specific conditions before proceeding.
Affects all elements in the WebDriver instance.	Applied to specific elements or conditions only.
`driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);`	`WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));` `wait.until(ExpectedConditions.visibilityOfElementLo` `y.id` `("elementId")));`
Less control – waits for all elements globally.	More control – can wait for specific elements or conditions like visibility, clickability, etc.
Universal timeout for all elements.	Waits for a specific timeout and only for a specifi condition.
None – only waits for elements to appear.	Can wait for various conditions (visibility, presen clickability, custom conditions, etc.).

Can slow down tests if elements load faster than the wait time.	More efficient, as it waits only for specific conditi then proceeds.
Will throw `NoSuchElementException` if an element is not found within the time limit.	Will throw `TimeoutException` if the condition is not within the specified time.
Should generally avoid combining with explicit waits to prevent conflicts.	Works best when used without implicit waits, for control of wait logic.

3. Locators

In Selenium WebDriver, **locators** and **selectors** are used to find and interact with elements on a webpage. Locating elements accurately is essential for automating UI testing.

1. Locators

WebDriver provides different types of locators to find elements in the DOM (Document Object Model). Some common locators include:

- **ID**: driver.findElement(By.id("submit-button"));
- **Name**: driver.findElement(By.name("username"));
- **Class Name**: driver.findElement(By.className("login-form"));
- **Tag Name**: driver.findElement(By.tagName("button"));
- **Link Text**: driver.findElement(By.linkText("Click Here"));
- **Partial Link Text**: driver.findElement(By.partialLinkText("Click"));
- **XPath**: driver.findElement(By.xpath("//button[@id='submit']"));
- **CSS Selector:** driver.findElement(By.cssSelector("input#username"));

2. DOM (Document Object Model)

The **DOM** represents the hierarchical structure of a webpage. Each HTML element on the page is a node in the DOM. WebDriver uses the DOM to locate elements. When a test interacts with an element (e.g., clicking a button), it's manipulating the corresponding element in the DOM.

Key concepts of the DOM:

- **Nodes**: Represent elements, attributes, or text within HTML.

- **Parent/Child Relationships**: DOM elements have parent-child or sibling relationships, which can be traversed to locate elements.

3. XPath

XPath (XML Path Language) is a query language used to locate elements in an XML-like structure such as the DOM.

Types of XPath:

- **Absolute XPath**: Provides a direct path from the root element.

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

Relative XPath: Starts from the current element or somewhere in the middle of the DOM.

```
//input[@name='username']
```

4. CSS Selectors

CSS selectors provide a concise way to select elements based on their attributes, id, or class. CSS selectors tend to be faster than XPath and are preferred when possible.

- **Element by ID**:driver.findElement(By.cssSelector("#username"));
- **Element by Class**:driver.findElement(By.cssSelector(".login-button"));
- **Element by Attribute:**driver.findElement(By.cssSelector("input[name='password']"));
- **Combining selectors**:driver.findElement(By.cssSelector("div#container .button"));

5. Reliable Locators

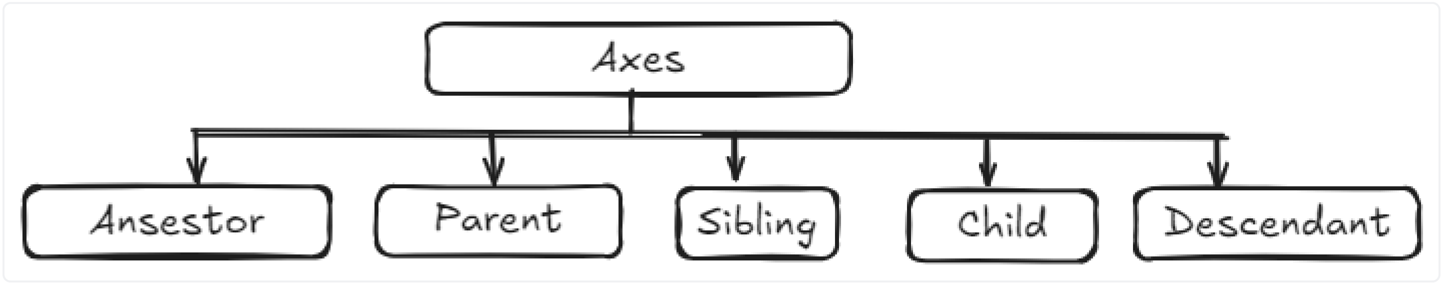
Choosing reliable locators is key to making your tests robust. Some tips for reliable locators:

- **Prefer IDs**: If an element has an `id`, it should be your first choice since it's unique.
- **Use Descriptive XPath or CSS Selectors**: Avoid using absolute paths like `/html/body/div`. Instead, prefer relative paths like `//div[@id='header']//input`.
- **Use Classes Carefully**: Classes may be shared across multiple elements, so ensure they are unique in the context of the element you're targeting.
- **Avoid Text-Based Locators**: `linkText()` or `partialLinkText()` may be unreliable if the text changes frequently.
- **Aria Labels**: Modern applications use ARIA labels, which can be used for accessible, stable locators:

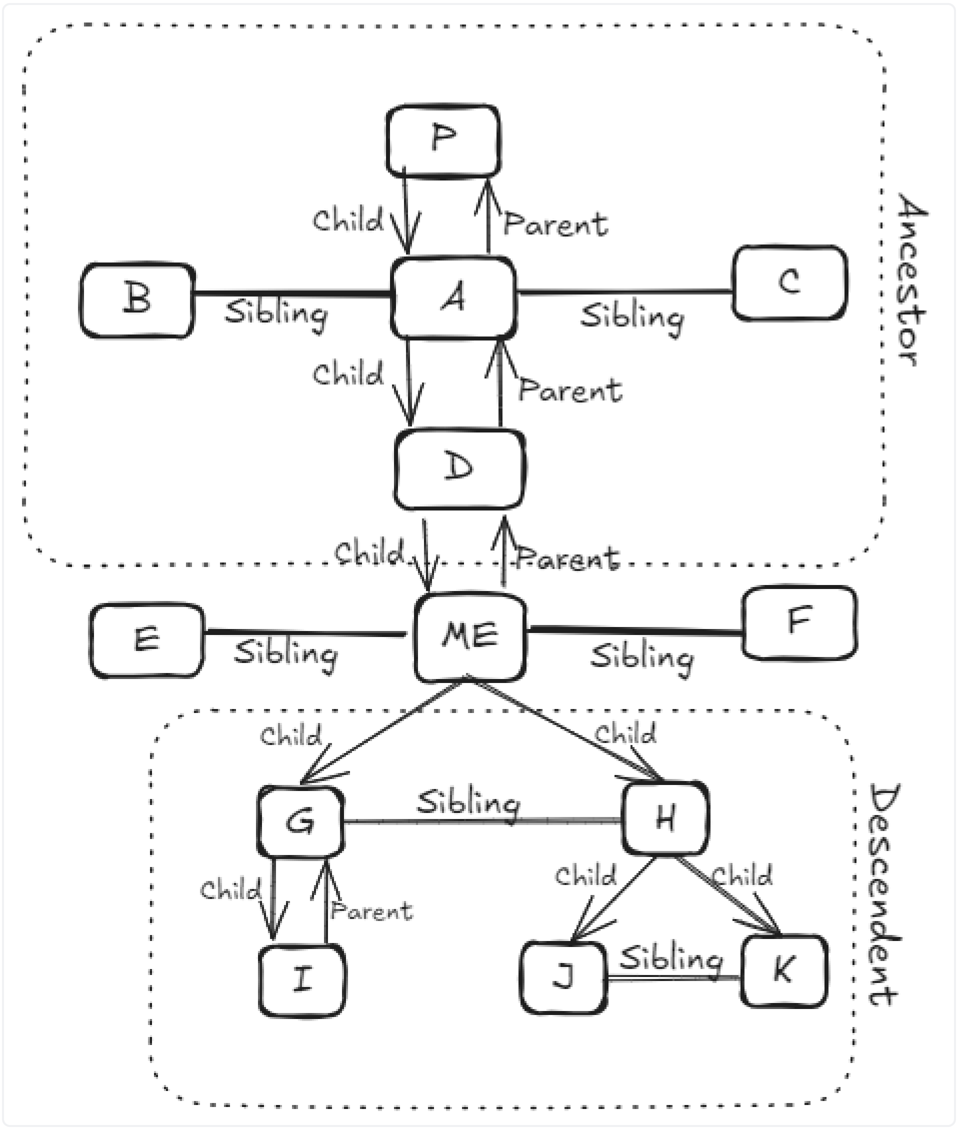
4. Xpath Axes

we use this where is no unique attribute attached to anything that we want to target.

So we act like the user to get most of the axes.



Now lets see the relationship with the element.



Ancestor

- ancestor
- ancestor-or-self

Descendant

- descendant
- descendant-or-self

Sibling

- Preceding-sibling
- following-sibling

Parent

- parent

Child

- child

```
//lable[text()='Email']/following-sibling::input[1]/Parent::div
```

```
//div[@class='container']/child::input[@type='text']
```

5. Page Object

POM

It is a design pattern in selenium that creates an Object Repository for storing all web elements.
Advantages:

- Makes code maintainable
- Makes code Readable
- Makes code reusable

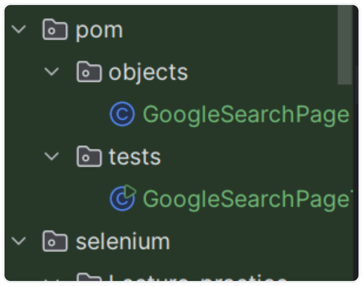
Basic Structure:

- Page Object Class: Contains methods that represent actions a user can perform on the page.
- Locators: Encapsulated within the page class, never exposed directly.
- Utility Methods: Should represent high-level actions (e.g., `login()`).

Rules:

- Avoid assertions inside page objects.
- Avoid overloading page objects with business logic.
- Keep interaction logic in page objects and validation in tests.

The Structure a POM folllows



Now the Objects would have the classes to interact with the element of the page we are looking for

```
package pom.objects;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
public class GoogleSearchPage {
    private static WebElement element;
    public static  WebElement searchBoxfilller(WebDriver driver){
        element = driver.findElement(By.name(”q”));
        return element;
    }
    public static WebElement clickSearch(WebDriver driver){
        element = driver.findElement(By.name(”btnK”));
        return element;
    }
}
```

And the test would obviosly be testing it, make surte to import the page of objects

```
package pom.tests;
import com.sun.xml.bind.v2.runtime.reflect.Lister;
import org.junit.jupiter.api.BeforeEach;
import org.openqa.selenium.Keys;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.annotations.BeforeSuite;
import pom.objects.GoogleSearchPage;

public class GoogleSearchPageTest {
    private static WebDriver driver;
    public static void main(String[] args) {

System.setProperty(”webdriver.chrome.driver”,”C:¥¥Users¥¥prabhatritesh_chaube¥¥Downloads¥¥chromedriver-win64¥¥chromedriver.exe”);
        driver = new ChromeDriver();
        driver.get(”https://www.google.co.in”);
        driver.manage().window().maximize();

        GoogleSearchPage.searchBoxfilller(driver).sendKeys("Bae on beach");
        GoogleSearchPage.clickSearch(driver).sendKeys(Keys.RETURN);

        driver.quit();
    }
}
```

Now we can go a step ahead and we can directly make all the functioning in one object class and test it in another.

And the Steps for the Same are:

a. Create A Class for each WebPage
b. Add object Locators
c. Add action Methods
d. Create Class for test case
e. Create object for page class
f. Refer Action Methods

and in that we Use

```
WebDriver driver;
public GoogleSearchUsingObject(WebDriver driver){
    this.driver = driver;
}
public static void main(String[] args) {

}
By locatebutton = By.xpath("//div[@class=¥"logo¥"]");
public  void clickbutton(WebDriver driver){
    driver.findElement(locatebutton).click();

}
```

and for the test we use :

```
private static WebDriver driver;
public static void main(String[] args) {

System.setProperty("webdriver.chrome.driver","C:¥¥Users¥¥prabhatritesh_chaube¥¥Downloads¥¥chromedriver-win64¥¥chromedriver.exe");
    driver = new ChromeDriver();
    driver.get("https://www.geeksforgeeks.org/problems/peak-element/1");
    GoogleSearchUsingObject performer = new GoogleSearchUsingObject(driver);
```

```
        performer.clickbutton(driver);
    }
```

Page Factory

page factory is used so that it can reduce the lines of codes, where pom is a way to separate objects and scripts the page factory is responsible for implementing it.

and we Use **@FindBY** and **@FindBYs** in the page factory

```
    @FindBy(xpath = ”//div[@class=¥”logo¥”]”) public WebElement clicker;
```

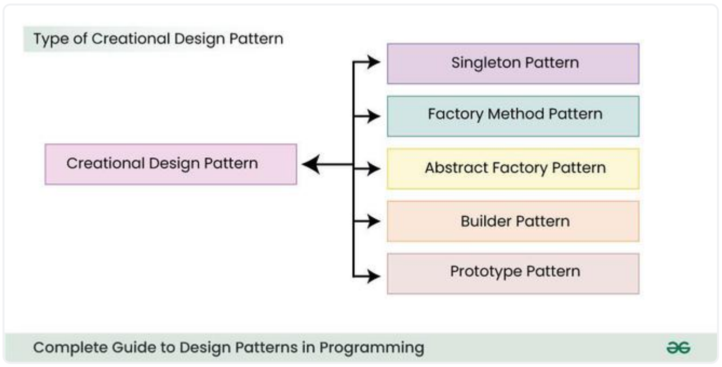
FindBY	FIndBys
Used to locate a single web element based on a specific locator strategy.	Used when you need to locate an element based on multiple conditions **sequentially**. It's essentially an "AND" condition between multiple locators.
@FindBy(id = "username")private WebElement usernameField;	@FindBys({ @FindBy(className = "form-group"), @FindBy(id = "username")}) private WebElement usernameField;

How can the POM be an antipattern:

- When Page Objects become too bloated or complex, with too many methods
- Sometimes, developers introduce methods in Page Objects that perform actions that span across multiple pages, which leads to Page Objects that are no longer representing a single web page.

Java Design Patterns : Creatonal

Creational design patterns are a category of design patterns in software development that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. They abstract the instantiation process and make it more flexible by decoupling the code from the actual object creation. These patterns promote reusability, scalability, and the efficient management of objects.


Type of Creational Design Pattern
Creational Design Pattern
Singleton Pattern
Factory Method Pattern
Abstract Factory Pattern
Builder Pattern
Prototype Pattern
Complete Guide to Design Patterns in Programming

Automated testing: API Automation

API is the acronym for application programming interface — a software intermediary that allows two applications to talk to each other. APIs are an accessible way to extract and share data within and across organizations.

API testing is basically black box testing which is simply concerned with the final output of the system under test. 1. Unit testing aims to verify whether the module delivers the required functionality. The development team monitors unit testing activity and makes necessary changes wherever required.

Response Codes (200, 404, 500, etc.)

1XX**Informational** – : Communicates transfer protocol-level information.

2XX **Success** – :Indicates that the client's request was accepted successfully.

3XX**Redirection** – :Indicates that the client must take some additional action in order to complete their request.

4XX**Client Error** – : This category of error status codes points the finger at clients.

5XX **Server Error** –:The server takes responsibility for these error status codes.

Http Get method

Used to request data from a specified resource.

```
@Test
    public void anotherget(){
        String response = given()
        .pathParam("id", 21)   // Step 3: set path parameter "id" to 21
        .when()                // Step 4: start defining the request
        .get("https://petstore.swagger.io/v2/pet/{id}") // Step 5: send a GET request to the specified
URL
        .then()                 // Step 6: start validating the response
        .statusCode(200)       // Step 7: assert that the HTTP status code is 200
        .body("id", equalTo(21)) // Step 8: assert that the JSON response contains an "id" field with a
value of 21
        .extract()             // Step 9: extract the response after validations
        .asPrettyString();
        System.out.println(response);
    }
```

HTTPS Post Method

Used to send data to create a new resource on the server.

```
@Test
public void createPet() {
    // Define the JSON body
    String newPet = ”{¥n” +
            ”    ¥”id¥”: 21,¥n” +
            ”    ¥”category¥”: {¥n” +
            ”        ¥”id¥”: 0,¥n” +
            ”        ¥”name¥”: ¥”string¥”¥n” +
            ”    },¥n” +
            ”    ¥”name¥”: ¥”doggie¥”,¥n” +
            ”    ¥”photoUrls¥”: [¥n” +
            ”        ¥”string¥”¥n” +
            ”    ],¥n” +
            ”    ¥”tags¥”: [¥n” +
            ”        {¥n” +
            ”            ¥”id¥”: 0,¥n” +
            ”            ¥”name¥”: ¥”string¥”¥n” +
            ”        }¥n” +
            ”    ],¥n” +
            ”    ¥”status¥”: ¥”available¥”¥n” +
            ”}”;

    // Send the POST request
    String response = given()
            .header(”Content-Type”, ”application/json”)  // Specify that the request body is JSON
            .body(newPet)  // Attach the JSON body to the request
            .when()
            .post(”https://petstore.swagger.io/v2/pet”)  // Send a POST request to create a new pet
            .then()
            .statusCode(200)  // Assert that the response status code is 200 (OK)
            .body(”id”, equalTo(21))  // Assert that the returned pet ID is 21
            .extract()
            .asPrettyString();  // Extract the response and format it as a pretty string

    // Print the response
    System.out.println(response);
}
```

HTTP Put Method

Used to update an existing resource.

SO when ever we need any update in the details then we use the Put Method.

- Use `POST` when you want to **add** a new resource.
- Use `PUT` when you want to **update** an existing resource (or create one at a specific URI).

The code of put and post is almost same, the only difference is that we got to use PUT isnstead of POST method.

HTTP Delete Method

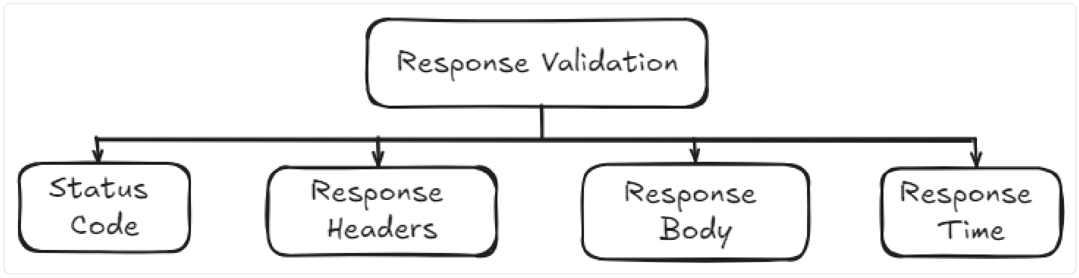
Delete is used to delete any existing data from the data set.

```
Response response = given()
        .pathParam("id",21)
        .when()
        .delete("https://petstore.swagger.io/v2/pet/{id}")
        .then()
        .statusCode(200)
        .log().all()
        .extract().response();
```

Making API requests involves sending data to or retrieving data from a server, which can be automated using RestAssured in Java. By mastering the use of methods like GET, POST, PUT, and DELETE, and learning to validate responses, you can build robust and reliable API tests.

Validating the response in RestAssured Framework

Validating API responses is a crucial part of automated API testing, ensuring that the server returns the expected output based on the request sent.



Validating Status Code:

Status codes are the most basic form of validation. Every HTTP request returns a specific status code to indicate success, failure, or errors.

```java
import io.restassured.RestAssured;
import static io.restassured.RestAssured.*;

public class ValidateStatusCode {

    public static void main(String[] args) {
        // Base URI of the API
        RestAssured.baseURI = "https://api.example.com";

        // Validate status code is 200 OK
        given()
            .when()
                .get("/students")
            .then()
                .statusCode(200)  // Validate status code
                .log().all();     // Log the response
    }
}
```

`.statusCode(200)`: This checks that the response status code is `200 OK`

Common Status Codes:

- `200 OK`: Request was successful.
- `201 Created`: Resource was created successfully (common for POST requests).
- `404 Not Found`: The requested resource does not exist.
- `500 Internal Server Error`: A server error occurred.

Validating Response Headers

Response headers provide important information about the response, such as content type, cache control, and authorization status.

```java
import io.restassured.RestAssured;
import static io.restassured.RestAssured.*;

public class ValidateHeaders {

    public static void main(String[] args) {
        // Base URI of the API
        RestAssured.baseURI = "https://api.example.com";

        // Validate specific headers in the response
        given()
            .when()
                .get("/students/1")
            .then()
                .statusCode(200)
                .header("Content-Type", "application/json")  // Validate Content-Type header
                .header("Cache-Control", "no-cache")         // Validate Cache-Control header
                .log().all();
    }
}
```

Validating Response Body

The response body is the most important part of API validation, especially when working with JSON or XML data. We typically validate that the data returned matches our expectations.

so suppose we have a response for a specific request :

```
{
    "id": 1,
    "name": "John Doe",
    "age": 20,
    "class": "12th Grade"
}
```

Now we want to validate is the response is correct or not then,

```
import io.restassured.RestAssured;
import static io.restassured.RestAssured.*;
import static org.hamcrest.Matchers.*;

public class ValidateJsonResponse {

    public static void main(String[] args) {
        // Base URI of the API
        RestAssured.baseURI = "https://api.example.com";

        // Validate response body content
        given()
            .when()
                .get("/students/1")
            .then()
                .statusCode(200)
                .body("id", equalTo(1))                // Validate student ID is 1
                .body("name", equalTo("John Doe"))    // Validate name
                .body("age", equalTo(20))             // Validate age
                .body("class", equalTo("12th Grade")) // Validate class
                .log().all();                         // Log the entire response
    }
}
```

and when we have any nested response like

```
{
    "id": 1,
    "name": "John Doe",
    "age": 20,
    "address": {
        "city": "New York",
        "zip": "10001"
    }
}
```

So now as we can see address have two responses to look for city and Zip, so now what we would do is

```
.statusCode(200)
.body("address.city", equalTo("New York"))  // Validate city in nested object
.body("address.zip", equalTo("10001"));     // Validate zip code
```

Validating Response Time

API performance is crucial, and we often need to ensure that the response time is within acceptable limits.

```
import io.restassured.RestAssured;
import static io.restassured.RestAssured.*;

public class ValidateResponseTime {

    public static void main(String[] args) {
        // Base URI of the API
        RestAssured.baseURI = "https://api.example.com";

        // Validate that the response time is less than 2000 ms (2 seconds)
        given()
            .when()
                .get("/students")
            .then()
                .statusCode(200)
                .time(lessThan(2000L))  // Validate response time in milliseconds
```

```
                    .log().all();
        }
    }
```

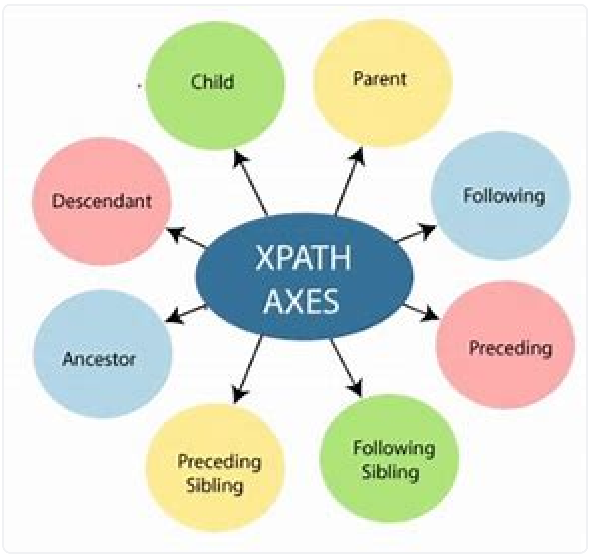
This code should be executed in or before 2000 milliseconds, or else it would fail the test

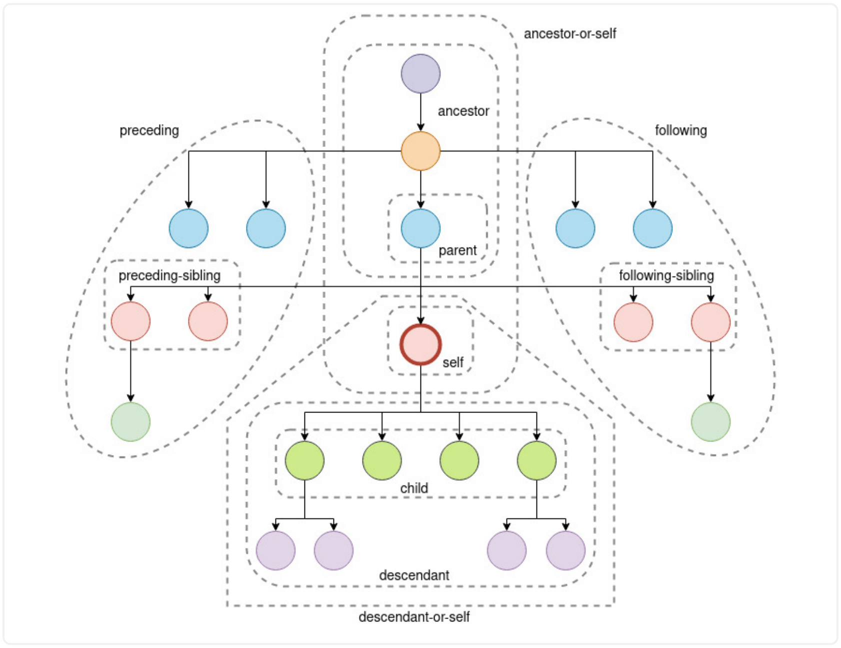
Difference in Query Parameters and Path Paramenter

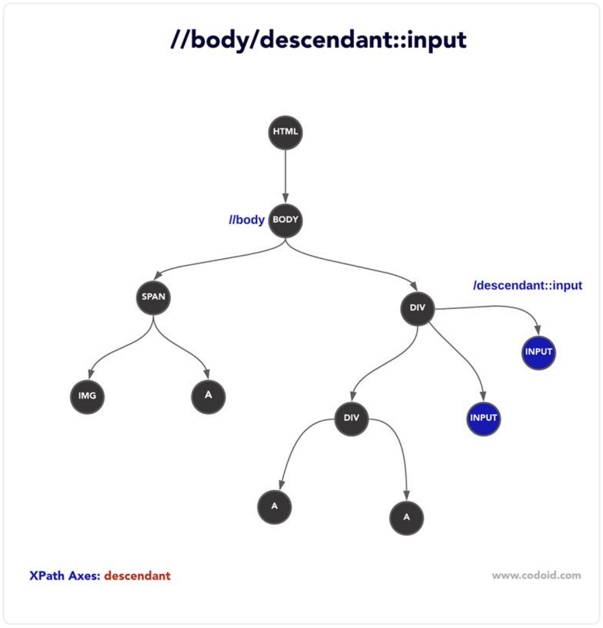
Path Paramenters	Query Parameters
Used to specify required values, such as resource IDs	Used to pass optional information (filters, pagination)
`/resource/{id}` (e.g., `/products/12345`)	`?key1=value1&key2=value2` (e.g., `?category=electronics`)

Dynamic X path

In Selenium, a **Dynamic XPath** refers to an XPath expression that is constructed to handle web elements whose attributes or positions may change dynamically during runtime. This is common in modern web applications where elements often have dynamic IDs, classes, or are generated dynamically.







Design Patterns

Singelton Design Pattern

Its ensure that only One instance should be created so that we can not created multiple instance for better performance.

The most important three rules to follow

- Private Constructor
- Static Member
- Static Method

- [ ] modifiy the code in terms of singleton Desing Pattern

Types of Design Patterns

- Factory(Creational Design Patterns)

  as the factory function, Factory Implementation focuses on making and using the functionalities in a way that the logic is not hold the value of transparency.
- Singleton
- Builder Design Pattern

  In the same sense as of building, the builder Design Pattern focuses on building/initializing the methodologies and function

builder

factory

singleton

Pom

Page Factory

refactoring.guru

Untitled

Selenium WebDriver

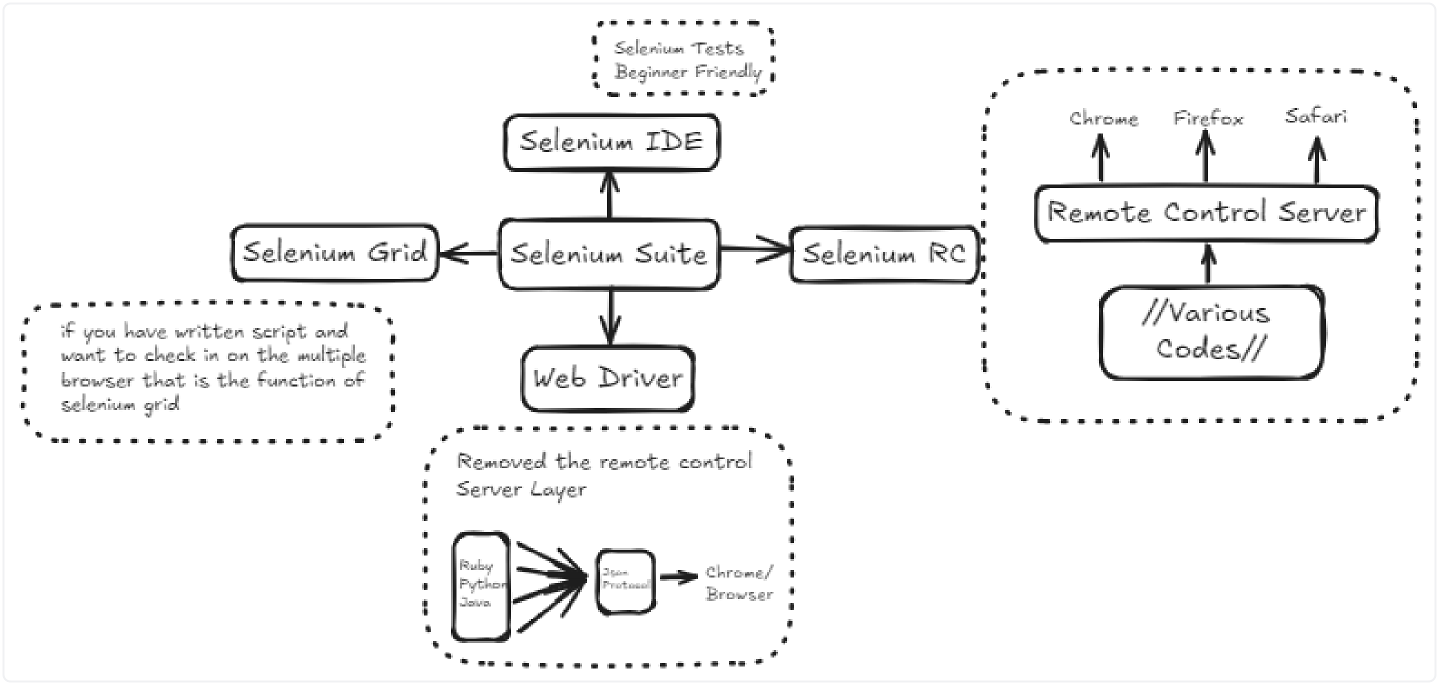
1. Selenium WebDriver

Selenium is Used to automate the UI Testing

We can also integrate selenium with different IDES.

Advantages of Selenium:

- Web Based Automation Tool
- Pen Source
- UI Automation
- Support Multiple OS
- Multiple Browsers
- Multiple languages



Everything Selenium does is send the browser commands to do something or send requests for information. Most of what you'll do with Selenium is a combination of these basic commands

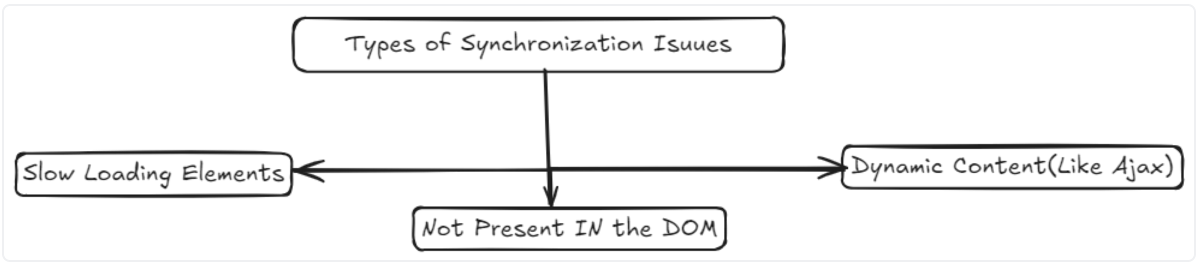
A very Basic Selenium Code

```
private WebDriver driver; //defining private so can use further
@BeforeEach
public void setup(){

System.setProperty("webdriver.chrome.driver","C:¥¥Users¥¥prabhatritesh_chaube¥¥Downloads¥¥chromedriver-
win64¥¥chromedriver.exe");
    driver = new ChromeDriver();
    driver.manage().window().maximize();
}

@Test
    public void selectone(){
        driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(20));
        driver.get("https://app.endtest.io/guides/docs/how-to-test-dropdowns/");
        WebElement select = driver.findElement(By.id("pets"));
        Select select1 = new Select(select);
        select1.selectByValue("rabbitt");
//        select1.selectByVisibleText("Rabbit");
//        select1.selectByIndex(5);

    }
 @AfterEach
    public void exit(){
        driver.quit();
    }
}
```

2. Synchronization

What are Synchronization Issues?

Synchronization issues arise when the WebDriver tries to interact with elements before they are ready (e.g., before they are loaded or clickable). These issues lead to flaky tests and frequent failures.

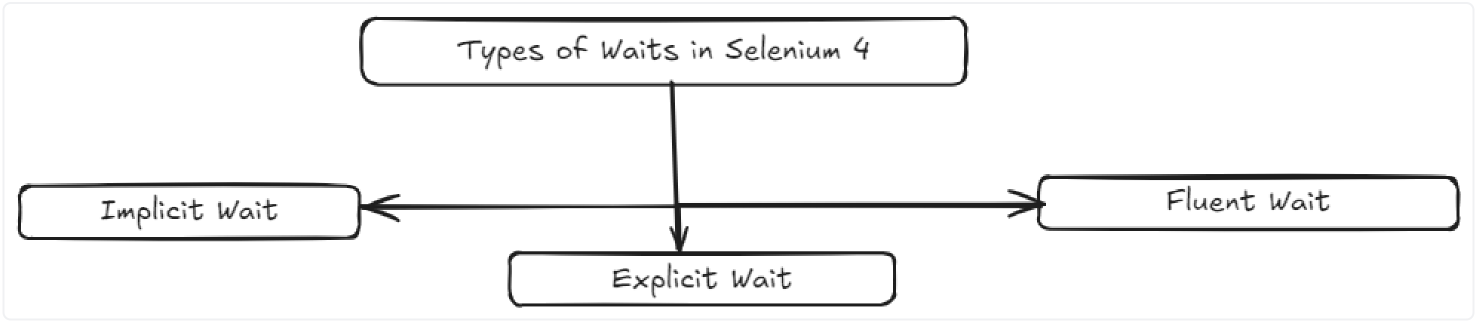


To ensure the WebDriver interacts with elements when they are in the correct state (loaded, clickable, etc.).

Common Solutions:

- General timeouts (implicit waits)
- Explicit waits
- Fluent waits
- Custom conditions

Waits



1.Implicit Wait

- Implicit Wait is used to set a default waiting time throughout your Selenium script. When you use implicit wait, it tells the WebDriver to wait for a certain amount of time before it throws a **NoSuchElementException**.

```
driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(20));
```

2.Explicit Wait

- Explicit Wait is used to wait for a specific condition to be true before proceeding further. It allows you to wait for a particular element or condition with a specific timeout.
- **Use Case:** Use explicit wait when you need to wait for a specific condition to occur before proceeding with the next step in the test script

```
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) {
        WebDriver driver = new ChromeDriver();
        driver.get("https://example.com");

        WebDriverWait wait = new WebDriverWait(driver, 20);
        WebElement element
=wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("exampleId")));

        // Perform actions on the element
        element.click();

        driver.quit();
    }
}
```

3.Fluent Wait

- Fluent Wait is similar to explicit wait but with more flexibility. You can define the maximum amount of time to wait for a condition, as well as the frequency with which to check the condition, and you can also ignore specific types of exceptions while waiting.
- **Use Case**: Use fluent wait when you need more control over the wait conditions, such as checking for the condition at regular intervals and ignoring certain exceptions.

```
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.FluentWait;
import java.time.Duration;
import java.util.NoSuchElementException;

public class FluentWaitExample {
    public static void main(String[] args) {
        WebDriver driver = new ChromeDriver();
        driver.get("https://example.com");

        FluentWait<WebDriver> wait = new FluentWait<>(driver)
                .withTimeout(Duration.ofSeconds(30))
                .pollingEvery(Duration.ofSeconds(5))
                .ignoring(NoSuchElementException.class);

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

        // Perform actions on the element
        element.click();

        driver.quit();
    }
}
```

As we have all the wait types, why cant we just use the **implicitlyWait** all the time?

Limitations of implicitlyWait:

- It applies globally and cannot target specific conditions.
- It might not be efficient in scenarios with dynamic content.
- It could cause unnecessary waiting in situations where the element is ready sooner.

Implicit Wait	**Explicit Wait**
Global wait applied to all element searches.	Waits for specific conditions before proceeding.
Affects all elements in the WebDriver instance.	Applied to specific elements or conditions only.
`driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);`	`WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));` `wait.until(ExpectedConditions.visibilityOfElementLo` `y.id` `("elementId")));`
Less control – waits for all elements globally.	More control – can wait for specific elements or conditions like visibility, clickability, etc.
Universal timeout for all elements.	Waits for a specific timeout and only for a specifi condition.
None – only waits for elements to appear.	Can wait for various conditions (visibility, presen clickability, custom conditions, etc.).

Can slow down tests if elements load faster than the wait time.	More efficient, as it waits only for specific conditi then proceeds.
Will throw `NoSuchElementException` if an element is not found within the time limit.	Will throw `TimeoutException` if the condition is not within the specified time.
Should generally avoid combining with explicit waits to prevent conflicts.	Works best when used without implicit waits, for control of wait logic.

3. Locators

In Selenium WebDriver, **locators** and **selectors** are used to find and interact with elements on a webpage. Locating elements accurately is essential for automating UI testing.

1. Locators

WebDriver provides different types of locators to find elements in the DOM (Document Object Model). Some common locators include:

- **ID**: driver.findElement(By.id("submit-button"));
- **Name**: driver.findElement(By.name("username"));
- **Class Name**: driver.findElement(By.className("login-form"));
- **Tag Name**: driver.findElement(By.tagName("button"));
- **Link Text**: driver.findElement(By.linkText("Click Here"));
- **Partial Link Text**: driver.findElement(By.partialLinkText("Click"));
- **XPath**: driver.findElement(By.xpath("//button[@id='submit']"));
- **CSS Selector:** driver.findElement(By.cssSelector("input#username"));
    - .className
    - .className
    - [type="text"]

2. DOM (Document Object Model)

The **DOM** represents the hierarchical structure of a webpage. Each HTML element on the page is a node in the DOM. WebDriver uses the DOM to locate elements. When a test interacts with an element (e.g., clicking a button), it's manipulating the corresponding element in the DOM.

Key concepts of the DOM:

- **Nodes**: Represent elements, attributes, or text within HTML.
- **Parent/Child Relationships**: DOM elements have parent-child or sibling relationships, which can be traversed to locate elements.

3. XPath

XPath (XML Path Language) is a query language used to locate elements in an XML-like structure such as the DOM.

Types of XPath:

- **Absolute XPath**: Provides a direct path from the root element.

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

Relative XPath: Starts from the current element or somewhere in the middle of the DOM.

```
//input[@name='username']
```

4. CSS Selectors

CSS selectors provide a concise way to select elements based on their attributes, id, or class. CSS selectors tend to be faster than XPath and are preferred when possible.

- **Element by ID**:driver.findElement(By.cssSelector("#username"));
- **Element by Class**:driver.findElement(By.cssSelector(".login-button"));
- **Element by Attribute:**driver.findElement(By.cssSelector("input[name='password']"));
- **Combining selectors**:driver.findElement(By.cssSelector("div#container .button"));

5. Reliable Locators

Choosing reliable locators is key to making your tests robust. Some tips for reliable locators:

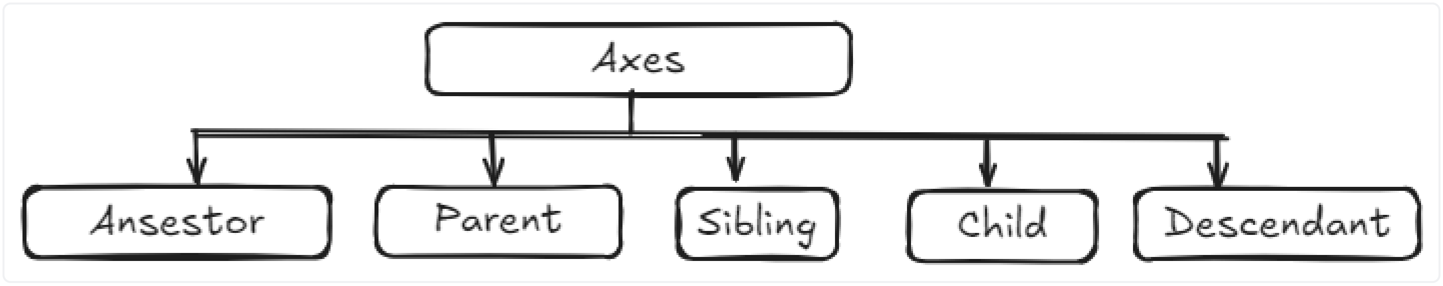
- **Prefer IDs**: If an element has an `id`, it should be your first choice since it's unique.
- **Use Descriptive XPath or CSS Selectors**: Avoid using absolute paths like `/html/body/div`. Instead, prefer relative paths like `//div[@id='header']//input`.
- **Use Classes Carefully**: Classes may be shared across multiple elements, so ensure they are unique in the context of the element you're targeting.
- **Avoid Text-Based Locators**: `linkText()` or `partialLinkText()` may be unreliable if the text changes frequently.
- **Aria Labels**: Modern applications use ARIA labels, which can be used for accessible, stable

locators:

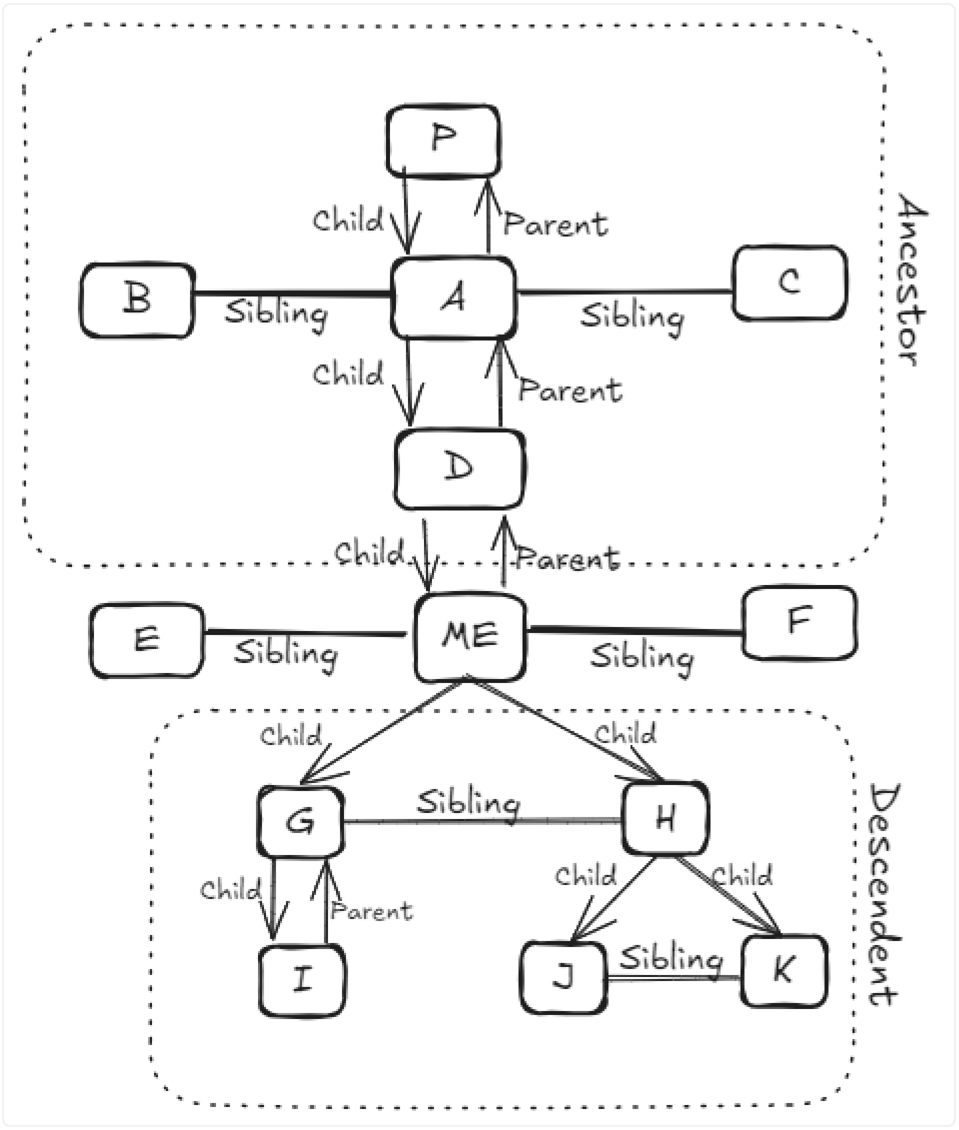
4. Xpath Axes

we use this where is no unique attribute attached to anything that we want to target.

So we act like the user to get most of the axes.



Now lets see the relationship with the element.



Ancestor

- ancestor
- ancestor-or-self

Descendant

- descendant
- descendant-or-self

Sibling

- Preceding-sibling
- following-sibling

Parent

- parent

Child

- child

```
//lable[text()='Email']/following-sibling::input[1]/Parent::div
```

```
//div[@class='container']/child::input[@type='text']
```

5. Page Object

POM

It is a design pattern in selenium that creates an Object Repository for storing all web elements.

Advantages:

- Makes code maintainable
- Makes code Readable
- Makes code reusable

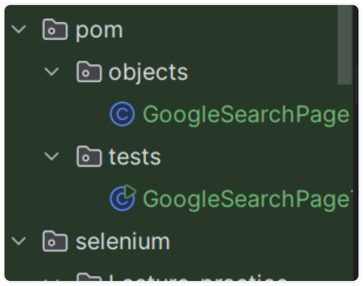
Basic Structure:

- Page Object Class: Contains methods that represent actions a user can perform on the page.
- Locators: Encapsulated within the page class, never exposed directly.
- Utility Methods: Should represent high-level actions (e.g., `login()`).

Rules:

- Avoid assertions inside page objects.
- Avoid overloading page objects with business logic.
- Keep interaction logic in page objects and validation in tests.

The Structure a POM folllows



Now the Objects would have the classes to interact with the element of the page we are looking for

```
package pom.objects;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
public class GoogleSearchPage {
    private static WebElement element;
    public static  WebElement searchBoxfilller(WebDriver driver){
        element = driver.findElement(By.name("q"));
        return element;
    }
    public static WebElement clickSearch(WebDriver driver){
        element = driver.findElement(By.name("btnK"));
        return element;
    }
}
```

And the test would obviosly be testing it, make surte to import the page of objects

```
package pom.tests;
import com.sun.xml.bind.v2.runtime.reflect.Lister;
import org.junit.jupiter.api.BeforeEach;
import org.openqa.selenium.Keys;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.annotations.BeforeSuite;
import pom.objects.GoogleSearchPage;

public class GoogleSearchPageTest {
    private static WebDriver driver;
    public static void main(String[] args) {

System.setProperty("webdriver.chrome.driver","C:¥¥Users¥¥prabhatritesh_chaube¥¥Downloads¥¥chromedriver-
win64¥¥chromedriver.exe");
        driver = new ChromeDriver();
        driver.get("https://www.google.co.in");
        driver.manage().window().maximize();

        GoogleSearchPage.searchBoxfilller(driver).sendKeys(”Bae on beach”);
        GoogleSearchPage.clickSearch(driver).sendKeys(Keys.RETURN);

        driver.quit();
    }
}
```

Now we can go a step ahead and we can directly make all the functioning in one object class and test it in another.

And the Steps for the Same are:

a. Create A Class for each WebPage
b. Add object Locators
c. Add action Methods
d. Create Class for test case
e. Create object for page class
f. Refer Action Methods

and in that we Use

```
WebDriver driver;
public GoogleSearchUsingObject(WebDriver driver){
    this.driver = driver;
}
public static void main(String[] args) {

}
By locatebutton = By.xpath(”//div[@class=¥”logo¥”]”);
public  void clickbutton(WebDriver driver){
    driver.findElement(locatebutton).click();

}
```

and for the test we use :

```
private static WebDriver driver;
public static void main(String[] args) {

System.setProperty(”webdriver.chrome.driver”,”C:¥¥Users¥¥prabhatritesh_chaube¥¥Downloads¥¥chromedriver-
win64¥¥chromedriver.exe”);
    driver = new ChromeDriver();
    driver.get(”https://www.geeksforgeeks.org/problems/peak-element/1”);
    GoogleSearchUsingObject performer = new GoogleSearchUsingObject(driver);
```

```
        performer.clickbutton(driver);
    }
```

Page Factory

page factory is used so that it can reduce the lines of codes, where pom is a way to separate objects and scripts the page factory is responsible for implementing it.

and we Use **@FindBY** and **@FindBYs** in the page factory

```
    @FindBy(xpath = ”//div[@class=¥”logo¥”]”) public WebElement clicker;
```

FindBY	FIndBys
Used to locate a single web element based on a specific locator strategy.	Used when you need to locate an element based on multiple conditions **sequentially**. It's essentially an "AND" condition between multiple locators.
@FindBy(id = "username")private WebElement usernameField;	@FindBys({ @FindBy(className = "form-group"), @FindBy(id = "username")}) private WebElement usernameField;

How can the POM be an antipattern:

- When Page Objects become too bloated or complex, with too many methods
- Sometimes, developers introduce methods in Page Objects that perform actions that span across multiple pages, which leads to Page Objects that are no longer representing a single web page.

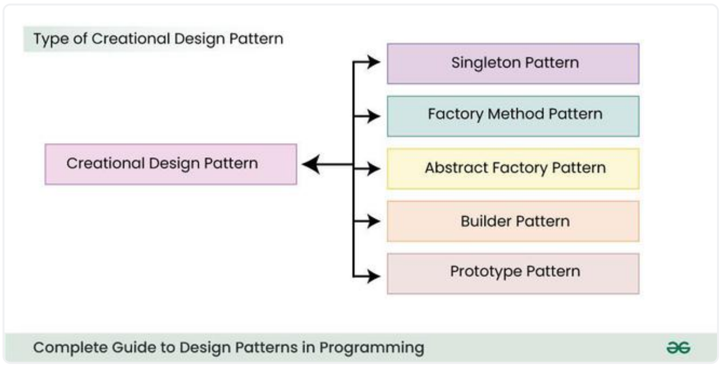
DIfference in Selenium Webdriver and Selenium RC

Selenium Webdriver	Selenium RC
Uses a simpler and more modern architecture; interacts directly with the browser.	Uses a server to translate commands from the client into browser commands.
Directly communicates with the browser via native OS methods.	Uses JavaScript to communicate with the browser through a server.

Supports all major browsers (Chrome, Firefox, IE, etc.) through browser-specific drivers.	Limited support; requires a server for each browser instance.

Java Design Patterns : Creatonal

Creational design patterns are a category of design patterns in software development that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. They abstract the instantiation process and make it more flexible by decoupling the code from the actual object creation. These patterns promote reusability, scalability, and the efficient management of objects.


Type of Creational Design Pattern
Creational Design Pattern
Singleton Pattern
Factory Method Pattern
Abstract Factory Pattern
Builder Pattern
Prototype Pattern
Complete Guide to Design Patterns in Programming

Factory Method Pattern:

In the Factory method pattern , we make an abstract class, then we define the abstract methods , for example we would define the methods of open(),save() and close().

Now we make two classes pdfDocument and WordDocument that would extend the abstract class so it would override all the methods we declared in the document Abstract class.

Then in main we call them using the object and it performs the task.

Abstract Factory Method:

Its the same as factory method, but now we make an interface that have a method to create the document and then we use the same to make more implementations.

Singleton :

We use singleton so that we can check if the object is null then return new object and if the object is not null then return the object we already have.

```
public class Singleton {
    // Static variable to hold the single instance
    private static Singleton instance;
    // Private constructor to prevent instantiation
    private Singleton() {
        // Initialization code here
    }
// Public method to provide access to the instance
    public static Singleton getInstance() {
        // Lazy initialization
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
    // Example method
    public void showMessage() {
        System.out.println("Hello from Singleton!");
    }
}
// Client code
public class SingletonExample {
    public static void main(String[] args) {
        // Access the singleton instance
        Singleton singleton = Singleton.getInstance();
        singleton.showMessage();
    }
}
```

Builder Pattern

The **Builder Method** (often referred to as the **Builder Pattern**) is a design pattern used to construct complex objects step by step. It provides a way to create objects that require many parameters or when the creation process is complex. This pattern is particularly useful when the object being constructed has numerous optional parameters, or when you want to separate the construction and representation of the object.

```
// Product class
class Car {
    private String make;
    private String model;
    private int year;
    private String color;

    // Constructor is private to prevent direct instantiation
    private Car(CarBuilder builder) {
        this.make = builder.make;
        this.model = builder.model;
        this.year = builder.year;
        this.color = builder.color;
    }

    @Override
    public String toString() {
        return "Car [make=" + make + ", model=" + model + ", year=" + year + ", color=" + color + "]";
    }

    // Static inner Builder class
    public static class CarBuilder {
        private String make;
        private String model;
        private int year;
        private String color;

        // Method to set the make
        public CarBuilder setMake(String make) {
            this.make = make;
            return this;
        }

        // Method to set the model
        public CarBuilder setModel(String model) {
            this.model = model;
            return this;
        }

        // Method to set the year
        public CarBuilder setYear(int year) {
            this.year = year;
            return this;
        }

        // Method to set the color
        public CarBuilder setColor(String color) {
            this.color = color;
            return this;
        }

        // Method to construct the Car object
        public Car build() {
            return new Car(this);
        }
    }
}

// Client code
public class BuilderPatternExample {
    public static void main(String[] args) {
        // Construct a Car object using the Builder pattern
        Car myCar = new Car.CarBuilder()
                .setMake(”Toyota”)
                .setModel(”Camry”)
                .setYear(2023)
                .setColor(”Red”)
                .build();

        // Print the Car object
        System.out.println(myCar);
    }
}
```

The Builder Pattern provides a flexible and efficient way to construct complex objects. It enhances code readability and maintainability by separating the construction process from the representation, making it easier to create objects with many parameters and configurations. This pattern is widely used in various applications, particularly when creating objects with optional or default settings.

TestNg

TestNg means Test Next Generation

get the Testng plugin, either thorugh dependencies or thoroush the plugin store.
now create a testng.xml that would have all the requred thing you need
create a class for testing, and supppose name it basistest in testng package
so now in the testngxml in the classs mention

```
<classes>
    <class name="testng.BasicTestNg"></class>
</classes>
```

and then run the tesng.xml(this is the test suite)

The basics testng code

```
package testng;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.Assert;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;

import java.time.Duration;

public class BasicTestNg {
    private static WebDriver driver;

    @BeforeMethod
    public void setup(){

System.setProperty("webdriver.chrome.driver","C:¥¥Users¥¥prabhatritesh_chaube¥¥Downloads¥¥chromedriver-win64¥¥chromedriver.exe");
        driver = new ChromeDriver();
        driver.manage().window().maximize();
        driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(10));
        driver.get("https://www.amazon.in/");
    }

    @Test
    public void checktitle(){
        String actual = driver.getTitle();
        String expected = ”Online Shopping site in India: Shop Online for Mobiles, Books, Watches, Shoes
and More - Amazon.in”;

        Assert.assertEquals(actual,expected);

    }

    @AfterMethod
    public void teardown(){
        driver.quit();
    }
}
```

TestNg Paramentres Annotations

this is used to send data from xml □ to testcases

```
<parameter name=”url” value=”www.example.com”></parameter>
```

use @Parameters({"url"}) before the @Test annotation and
when using pass a value for example

```
public void checkpackage(String myurl){
system.out.println(myurl)
//this would give output as www.example.com
```

@dataProvider annotaions

used to send java file □ Test cases

by using the **@dataprovider** annotation we can use a function to porvide a data to the test case and to the Test we would be providing it would become
@Test() □ @Test(dataProvider="function the gives the data")

```
@Test(dataProvider="function the gives the data")
public void takeinput(data types we are passing)
```

TesngListners

they are used to help us do what we do when the test fails

- make a class in which we want to do the test,
- now implement the ITestListner interface to the class
- override the methods we want to use, right click the screen and implemt / override it
- now in the testng.xml we would add

```
<listeners>
    <listener class-name="testng.Testlistners"></listener>
</listeners>
```

between start of suite and test name

Soft Assertions

When we use the assertion , if it is failing then the u[coming functions does not work, but when using soft assertion

```
SoftAssert a = new SoftAssert();
Assert.assertTrue(false) ☒ a.asserTrue(false)
```

a.assertAll() would store the output

parallel testng

when we run the testng.xml is runs in serial manner and now to do for parallel

```
<suite name="Suite" parallel='tests' thread-count="2">
```

this is for tests now if we want to run the classes parallelly then

```
<test name="test1" parallel="classes" thread-count = "2">
```

taking screenshot in testng

```
File f = ((TakeScreenShot)driver).getScreenshotas(OutputType.FILE);
Files.copy(f,new File(”location of file with name”))
```

`@BeforeMethod`	`@BeforeTest`
Runs before each test method in the class.	Runs before all test methods in a `<test>` block in `testng.xml`.
Before **every individual test method**.	Before the **first test method** in a `<test>` block.
Used for setting up preconditions for each test method (e.g., initializing data for each test).	Used for setting up configurations for all test methods in a `<test>` tag (e.g., global setup or initializing a browser).
Runs **once per test method**.	Runs **once per `<test>` block** in `testng.xml`.
Resetting state before each test method execution.	Setting up a database connection for all tests in a test group.
Applies to test methods in a **single class**.	Applies to all methods within the `<test>` defined in `testng.xml`.

@BeforeMethod

@AfterMethod

@Test

@BeforeSuite

@AfterSuite

@BeforeTest

@Afterest

@Parameters

@dataprovider

Assert.asserequalt(Actual,Expected)

Mock Tests

1. API Testing (Postman, RestAssured)

Theoretical Questions:

1. Explain the difference between `GET`, `POST`, `PUT`, and `DELETE` HTTP methods.
2. How does RestAssured handle authentication for API requests?

```
1. Basic Authentication
Basic Authentication is a simple authentication scheme built into the HTTP protocol. It uses a static
username and password combination.
2. OAuth 2.0 Authentication
RestAssured can handle OAuth 2.0 authentication, which is often used for APIs that require more secure
access.

RestAssured.given()
             .auth()
             .oauth2(accessToken) // Use OAuth2 token
              .when()
               .get("https://example.com/api/resource")
                .then()
                .statusCode(200);
```

1. What is the purpose of query parameters in a `GET` request? How are they different from path parameters?

Practical Questions:

1. Use Postman to test the following scenario:
   - Send a `GET` request with a query parameter to retrieve students whose age is greater than 18. Verify the response status code and body contains a list of students.
2. Write a RestAssured test to validate the following API:
   - Send a `POST` request to create a new student resource with JSON body. Validate that the response status code is `201 Created`.

---

2. Selenium WebDriver

Theoretical Questions:

1. What is Selenium WebDriver and how does it differ from Selenium RC?
2. Explain how to handle synchronization issues in Selenium.
3. What is the difference between `findElement()` and `findElements()` methods in Selenium?

Practical Questions:

1. Write a simple Selenium WebDriver test to automate the following scenario:
   - Navigate to a search engine, search for "Selenium WebDriver," and print the titles of the first five search results.
2. How would you handle a dynamic dropdown list using Selenium WebDriver?

---

3. Synchronization in Selenium

Theoretical Questions:

1. Why is using `Thread.sleep()` not recommended in Selenium tests for synchronization?
2. Compare and contrast implicit, explicit, and fluent waits in Selenium. When would you use each?

Practical Questions:

1. Implement an explicit wait in Selenium for an element that becomes visible after 10 seconds on the page.
2. Write a Selenium code to handle a synchronization issue where the page content loads at different times (use fluent wait).

---

4. Design Patterns

Theoretical Questions:

1. What is the Singleton pattern? How is it implemented in Java?
2. Describe the Page Object Model (POM) and explain why it's considered an antipattern in some cases.
3. Explain the Factory Method pattern and its use cases.

Practical Questions:

1. Implement the Builder pattern to create a `User` object with fields like `name`, `age`, `email`, and `address`.
2. Write a basic example of a Factory Method pattern to create different types of bank accounts (e.g., `SavingsAccount`, `CurrentAccount`).

---

5. TestNG Framework

Theoretical Questions:

1. What is the purpose of the `@DataProvider` annotation in TestNG? How is it used?
2. How does TestNG allow parameterization of test cases? Provide an example.

Practical Questions:

1. Write a TestNG test that verifies the login functionality for a web application. Use the `@DataProvider` to pass multiple sets of credentials.
2. Implement a TestNG test to run in parallel on multiple browsers (cross-browser testing).

---

6. Handling Locators and Page Object Model in Selenium

Theoretical Questions:

1. Explain the different types of locators in Selenium. When would you prefer using `CSS` selectors over `Xpath`?
2. What are the advantages of using the Page Object Model (POM) in Selenium?

Practical Questions:

1. Write Selenium code to locate a button by `Xpath` and click on it.
2. Implement a basic Page Object Model for a login page that includes methods to enter the username, password, and click the login button.

Practical Coding Challenges

1. **API Testing with RestAssured**: Write a RestAssured test to send a `DELETE` request to an API endpoint that deletes a student record. Validate the response status code is `200 OK`, and that the student no longer exists by sending a `GET` request.
2. **Synchronization in Selenium**: Write a Selenium test to handle a scenario where a webpage has a loading spinner that disappears when the content is ready. Use appropriate synchronization techniques to wait until the spinner disappears.
3. **TestNG – Parallel Execution**: Implement a TestNG test suite that runs two tests in parallel. One test should execute on Chrome and the other on Firefox.

Mock Test for Interview Preparation (Version 2)

---

1. API Testing (Postman, RestAssured)

Theoretical Questions:

1. What is the purpose of HTTP headers in API requests? Give examples of common headers.
2. Explain how you would test the performance of an API.
3. What are status codes in HTTP, and what do they indicate? Describe the significance of codes like `200`, `400`, `404`, and `500`.

Practical Questions:

1. In Postman, create a request that tests an API endpoint that returns a list of products. Ensure to validate the response time is within acceptable limits (e.g., <200 ms).
2. Write a RestAssured test that checks whether a specific product exists by sending a `GET` request to the endpoint and asserting that the response body contains the product name.

---

2. Selenium WebDriver

Theoretical Questions:

1. What is the difference between an absolute and a relative XPath?
2. Describe the concept of "headless" testing in Selenium and its advantages.
3. How do you handle alerts and pop-ups in Selenium?

Practical Questions:

1. Write a Selenium WebDriver script to automate the following:
   - Navigate to a login page, fill in the username and password, submit the form, and verify that the user is redirected to the homepage.
2. Write a Selenium code snippet to scroll to the bottom of a page and click a button that appears after scrolling.

---

3. Synchronization in Selenium

Theoretical Questions:

1. Explain how implicit waits and explicit waits differ. When should you use one over the other?
2. What are some common causes of synchronization issues in Selenium tests?

Practical Questions:

1. Implement a test using an implicit wait to wait for an element to appear before interacting with it. Demonstrate the difference between using implicit and explicit waits in two different methods.
2. Write Selenium code that uses a fluent wait to check for an element's visibility on a page, retrying for a maximum of 30 seconds.

---

4. Design Patterns

Theoretical Questions:

1. What is the Observer pattern? Provide a real-world example of its application.
2. Discuss the Strategy pattern and how it can be used to improve flexibility in your code.
3. Explain the Adapter pattern and when you would use it.

Practical Questions:

1. Implement a simple Observer pattern in Java where you have a `NewsPublisher` that notifies subscribers about news updates.
2. Create an example of the Strategy pattern where you have different algorithms for sorting an array of numbers.

---

5. TestNG Framework

Theoretical Questions:

1. How do you configure a TestNG suite? What XML elements are used in the configuration file?
2. Explain the use of `@BeforeClass` and `@AfterClass` annotations in TestNG.

Practical Questions:

1. Write a TestNG test that uses the `@BeforeMethod` annotation to set up the testing environment before each test case.
2. Implement a TestNG test with multiple `@Test` annotations that demonstrate the dependency between tests.

---

6. Handling Locators and Page Object Model in Selenium

Theoretical Questions:

1. What is the importance of using locators wisely in Selenium? What could happen if you use incorrect locators?
2. Describe how the Page Factory design pattern works in Selenium.

Practical Questions:

1. Write a Selenium code snippet to demonstrate how to find a checkbox element using a CSS selector and check it if it is not already checked.
2. Implement a Page Object Model for a registration page, including methods for entering user details and submitting the form.

Practical Coding Challenges

1. **API Testing with Postman**: Create a Postman collection that tests various endpoints of an API (e.g., `GET`, `POST`, `PUT`, `DELETE`). Include pre-request scripts to handle authentication tokens.
2. **Selenium Test with Waits**: Write a Selenium test that navigates to an e-commerce website, adds an item to the cart, and validates that the cart's item count has increased, using appropriate synchronization techniques.

3. **TestNG – Parameterization**: Create a TestNG test class that uses the `@DataProvider` annotation to run tests with different input data for a method that checks if a given string is a palindrome.

Mock Test for Interview Preparation (Version 3)

---

1. API Testing (Postman, RestAssured)

Theoretical Questions:

1. What are the key differences between REST and SOAP APIs?

Protocol	Primarily uses HTTP/HTTPS	Can use multiple protocols (HTTP, SMTP, etc.)
Data Format	Typically uses JSON or XML	Strictly uses XML

1. How would you validate the schema of a JSON response from an API?
2. Describe the process of authentication in APIs. What are some common methods?

Practical Questions:

1. Using Postman, create a test that checks if the response body of a `GET` request contains a specific field (e.g., `name` in a user object).
2. Write a RestAssured test case that validates the response time of an API endpoint is less than 300 milliseconds.

---

2. Selenium WebDriver

Theoretical Questions:

1. What is the role of the WebDriver in Selenium, and how does it differ from Selenium RC?
2. Explain how to handle file uploads in Selenium WebDriver.
3. What are the limitations of Selenium?

Practical Questions:

1. Write a Selenium script that navigates to a page, searches for a term, and verifies that the search results contain that term.

2. Create a Selenium code snippet that takes a screenshot of a webpage and saves it to a specified location.

---

3. Synchronization in Selenium

Theoretical Questions:

1. What are the potential issues of not using waits in Selenium tests?
2. Explain how you can handle dynamic web elements that load at different times.

Practical Questions:

1. Implement a Selenium test using an explicit wait to wait for an element (e.g., a button) to become clickable before proceeding.
2. Write a code example demonstrating the use of a `WebDriverWait` to wait for an element to be visible and then interact with it.

---

4. Design Patterns

Theoretical Questions:

1. Describe the Singleton pattern. What are its advantages and potential drawbacks?
2. What is the Factory pattern, and how does it differ from the Abstract Factory pattern?
3. Explain the Decorator pattern and give a use case where it would be beneficial.

Practical Questions:

1. Implement a Singleton class in Java that controls access to a resource, ensuring only one instance can be created.
2. Create a simple example of the Factory pattern to instantiate different types of vehicles (e.g., Car, Bike) based on a parameter.

---

5. TestNG Framework

Theoretical Questions:

1. What is the purpose of `@DependsOnMethods` in TestNG?
2. Explain the difference between `@Test(priority = 1)` and `@Test(dependsOnMethods = ”methodName”)`.

Practical Questions:

1. Write a TestNG test class with methods that demonstrate the use of `@AfterSuite` and `@BeforeSuite` annotations for setting up and tearing down resources.
2. Create a TestNG test that reads data from an external Excel file and uses it as input for the tests.

---

6. Handling Locators and Page Object Model in Selenium

Theoretical Questions:

1. What are some best practices for writing locators in Selenium?
2. How does the Page Object Model improve test maintainability?

Practical Questions:

1. Write a Selenium test that utilizes the Page Object Model for a login page, encapsulating the logic for entering credentials and clicking the login button.
2. Create a Page Object class for a search results page that includes methods to retrieve the number of results and click on a specific result.

Practical Coding Challenges

1. **API Testing with Postman**: Develop a comprehensive Postman collection to test a fictional online bookstore API, covering endpoints for books, authors, and categories. Include tests for status codes, response times, and data validation.
2. **Selenium Test with Complex Actions**: Write a Selenium test that performs the following actions:
   - Navigate to a webpage.
   - Fill out a form with various input fields, including dropdowns and checkboxes.
   - Submit the form and validate the success message.
3. **TestNG – Grouping Tests**: Create a TestNG test class that groups tests using the `@Test(groups = {"smoke"})` annotation. Ensure you run the grouped tests together and provide a summary of their execution.

Bonus Coding Challenges

1. **Multi-threading**: Implement a simple multi-threaded program in Java that prints numbers from 1 to 10, where each number is printed by a different thread.
2. **Data Structure Implementation**: Write a Java class that implements a basic stack data structure, including methods for push, pop, and checking if the stack is empty.

Automated testing: API Automation

API is the acronym for application programming interface — a software intermediary that allows two applications to talk to each other. APIs are an accessible way to extract and share data within and across organizations.

API testing is basically black box testing which is simply concerned with the final output of the system under test. 1. Unit testing aims to verify whether the module delivers the required functionality. The development team monitors unit testing activity and makes necessary changes wherever required.

Response Codes (200, 404, 500, etc.)
1XX**Informational** – : Communicates transfer protocol-level information.
2XX **Success** – :Indicates that the client's request was accepted successfully.
3XX**Redirection** – :Indicates that the client must take some additional action in order to complete their request.
4XX**Client Error** – : This category of error status codes points the finger at clients.
5XX **Server Error** –:The server takes responsibility for these error status codes.

Http Get method

Used to request data from a specified resource.

```
@Test
    public void anotherget(){
        String response = given()
        .pathParam("id", 21)   // Step 3: set path parameter "id" to 21
        .when()                // Step 4: start defining the request
        .get("https://petstore.swagger.io/v2/pet/{id}") // Step 5: send a GET request to the specified
URL
        .then()                 // Step 6: start validating the response
        .statusCode(200)       // Step 7: assert that the HTTP status code is 200
        .body("id", equalTo(21)) // Step 8: assert that the JSON response contains an "id" field with a
value of 21
        .extract()             // Step 9: extract the response after validations
        .asPrettyString();
        System.out.println(response);
    }
```

HTTPS Post Method

Used to send data to create a new resource on the server.

```
@Test
public void createPet() {
    // Define the JSON body
    String newPet = ”{¥n” +
            ”    ¥”id¥”: 21,¥n” +
            ”    ¥”category¥”: {¥n” +
            ”        ¥”id¥”: 0,¥n” +
            ”        ¥”name¥”: ¥”string¥”¥n” +
            ”    },¥n” +
            ”    ¥”name¥”: ¥”doggie¥”,¥n” +
            ”    ¥”photoUrls¥”: [¥n” +
            ”        ¥”string¥”¥n” +
            ”    ],¥n” +
            ”    ¥”tags¥”: [¥n” +
            ”        {¥n” +
            ”            ¥”id¥”: 0,¥n” +
            ”            ¥”name¥”: ¥”string¥”¥n” +
            ”        }¥n” +
            ”    ],¥n” +
            ”    ¥”status¥”: ¥”available¥”¥n” +
            ”}”;

    // Send the POST request
    String response = given()
            .header(”Content-Type”, ”application/json”)  // Specify that the request body is JSON
            .body(newPet)  // Attach the JSON body to the request
            .when()
            .post(”https://petstore.swagger.io/v2/pet”)  // Send a POST request to create a new pet
            .then()
            .statusCode(200)  // Assert that the response status code is 200 (OK)
            .body(”id”, equalTo(21))  // Assert that the returned pet ID is 21
            .extract()
            .asPrettyString();  // Extract the response and format it as a pretty string

    // Print the response
    System.out.println(response);
}
```

HTTP Put Method

Used to update an existing resource.

SO when ever we need any update in the details then we use the Put Method.

- Use `POST` when you want to **add** a new resource.
- Use `PUT` when you want to **update** an existing resource (or create one at a specific URI).

The code of put and post is almost same, the only difference is that we got to use PUT isnstead of POST method.

HTTP Delete Method

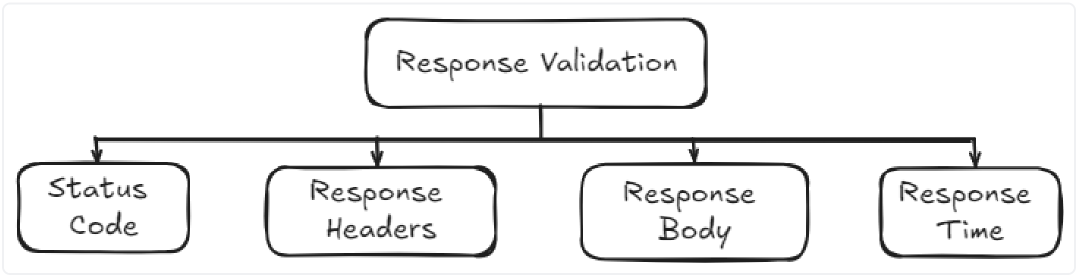
Delete is used to delete any existing data from the data set.

```
Response response = given()
        .pathParam(”id”,21)
        .when()
        .delete(”https://petstore.swagger.io/v2/pet/{id}”)
        .then()
        .statusCode(200)
        .log().all()
        .extract().response();
```

Making API requests involves sending data to or retrieving data from a server, which can be automated using RestAssured in Java. By mastering the use of methods like GET, POST, PUT, and DELETE, and learning to validate responses, you can build robust and reliable API tests.

Validating the responce in RestAssured FrameWork

Validating API responses is a crucial part of automated API testing, ensuring that the server returns the expected output based on the request sent.



Validating Status Code:

Status codes are the most basic form of validation. Every HTTP request returns a specific status code to indicate success, failure, or errors.

```
import io.restassured.RestAssured;
import static io.restassured.RestAssured.*;

public class ValidateStatusCode {

    public static void main(String[] args) {
        // Base URI of the API
        RestAssured.baseURI = "https://api.example.com";

        // Validate status code is 200 OK
        given()
            .when()
                .get("/students")
            .then()
                .statusCode(200)  // Validate status code
                .log().all();     // Log the response
    }
}
```

`.statusCode(200)`: This checks that the response status code is `200 OK`

Common Status Codes:

- `200 OK`: Request was successful.
- `201 Created`: Resource was created successfully (common for POST requests).
- `404 Not Found`: The requested resource does not exist.
- `500 Internal Server Error`: A server error occurred.

Validating Response Headers

Response headers provide important information about the response, such as content type, cache control, and authorization status.

```
import io.restassured.RestAssured;
import static io.restassured.RestAssured.*;

public class ValidateHeaders {

    public static void main(String[] args) {
        // Base URI of the API
        RestAssured.baseURI = "https://api.example.com";

        // Validate specific headers in the response
        given()
            .when()
                .get("/students/1")
            .then()
                .statusCode(200)
                .header("Content-Type", "application/json")  // Validate Content-Type header
                .header("Cache-Control", "no-cache")         // Validate Cache-Control header
                .log().all();
    }
}
```

Validating Response Body

The response body is the most important part of API validation, especially when working with JSON or XML data. We typically validate that the data returned matches our expectations.

so suppose we have a response for a specific request :

```
{
    "id": 1,
    "name": "John Doe",
    "age": 20,
    "class": "12th Grade"
}
```

Now we want to validate is the response is correct or not then,

```
import io.restassured.RestAssured;
import static io.restassured.RestAssured.*;
import static org.hamcrest.Matchers.*;

public class ValidateJsonResponse {

    public static void main(String[] args) {
        // Base URI of the API
        RestAssured.baseURI = "https://api.example.com";

        // Validate response body content
        given()
            .when()
                .get("/students/1")
            .then()
                .statusCode(200)
                .body("id", equalTo(1))               // Validate student ID is 1
                .body("name", equalTo("John Doe"))    // Validate name
                .body("age", equalTo(20))             // Validate age
                .body("class", equalTo("12th Grade")) // Validate class
                .log().all();                         // Log the entire response
    }
}
```

and when we have any nested response like

```
{
    "id": 1,
    "name": "John Doe",
    "age": 20,
    "address": {
        "city": "New York",
        "zip": "10001"
    }
}
```

So now as we can seee address have two responses to look for city and Zip, so now what we would do is

```
.statusCode(200)
.body("address.city", equalTo("New York"))  // Validate city in nested object
.body("address.zip", equalTo("10001"));     // Validate zip code
```

Validating Response Time

API performance is crucial, and we often need to ensure that the response time is within acceptable limits.

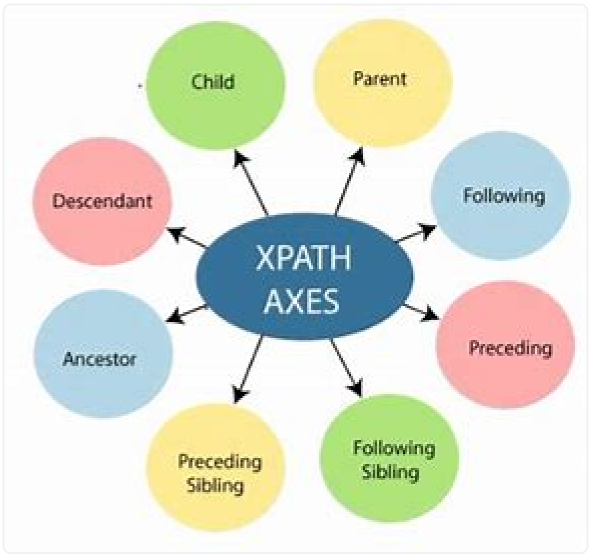
```
import io.restassured.RestAssured;
import static io.restassured.RestAssured.*;

public class ValidateResponseTime {

    public static void main(String[] args) {
        // Base URI of the API
        RestAssured.baseURI = "https://api.example.com";

        // Validate that the response time is less than 2000 ms (2 seconds)
        given()
            .when()
                .get("/students")
            .then()
                .statusCode(200)
                .time(lessThan(2000L))  // Validate response time in milliseconds
```

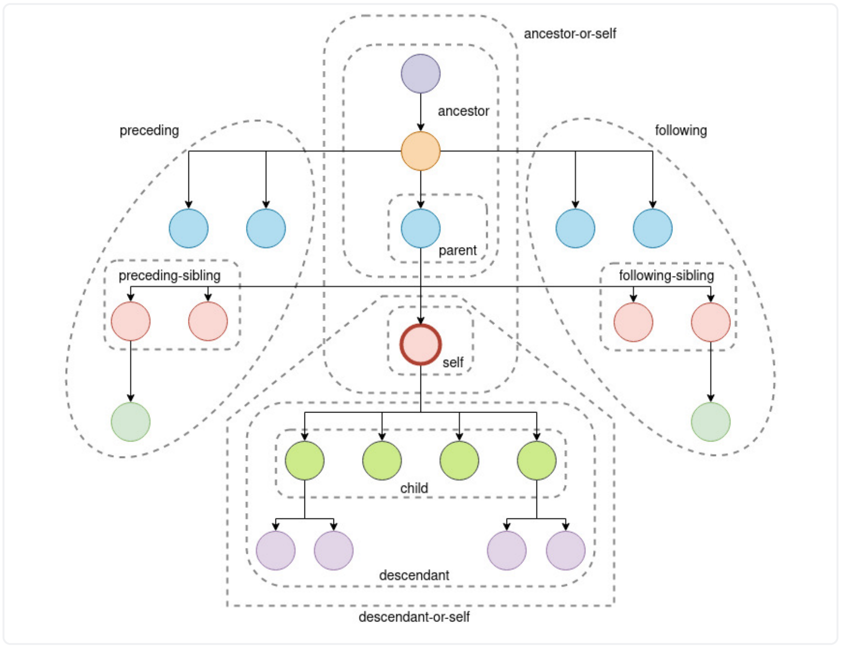
```
                    .log().all();
        }
    }
```

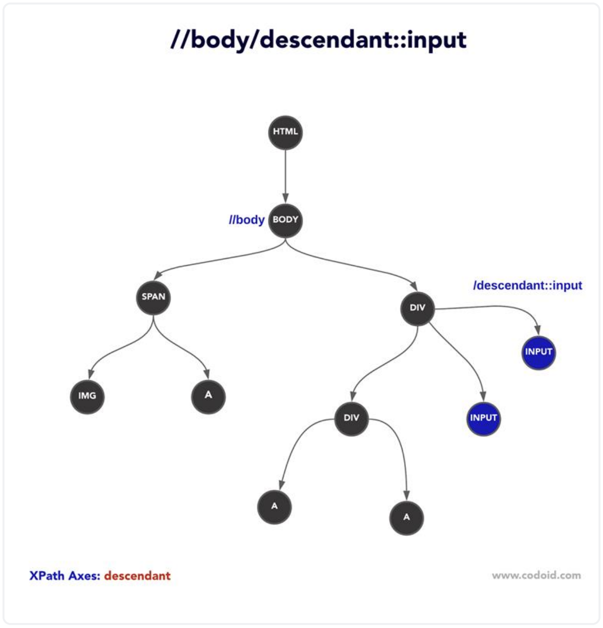
This code should be executed in or before 2000 milliseconds, or else it would fail the test

Dynamic X path

In Selenium, a **Dynamic XPath** refers to an XPath expression that is constructed to handle web elements whose attributes or positions may change dynamically during runtime. This is common in modern web applications where elements often have dynamic IDs, classes, or are generated dynamically.







Design Patterns

Singelton Design Pattern

Its ensure that only One instance should be created so that we can not created multiple instance for better performance.

The most important three rules to follow

- Private Constructor
- Static Member
- Static Method

- [ ] modifiy the code in terms of singleton Desing Pattern

Types of Design Patterns

- Factory(Creational Design Patterns)

    as the factory function, Factory Implementation focuses on making and using the functionalities in a way that the logic is not hold the value of transparency.
- Singleton
- Builder Design Pattern

    In the same sense as of building, the builder Design Pattern focuses on building/initializing the methodologies and function

builder

factory

singleton

Pom

Page Factory

refactoring.guru

Untitled

Selenium WebDriver

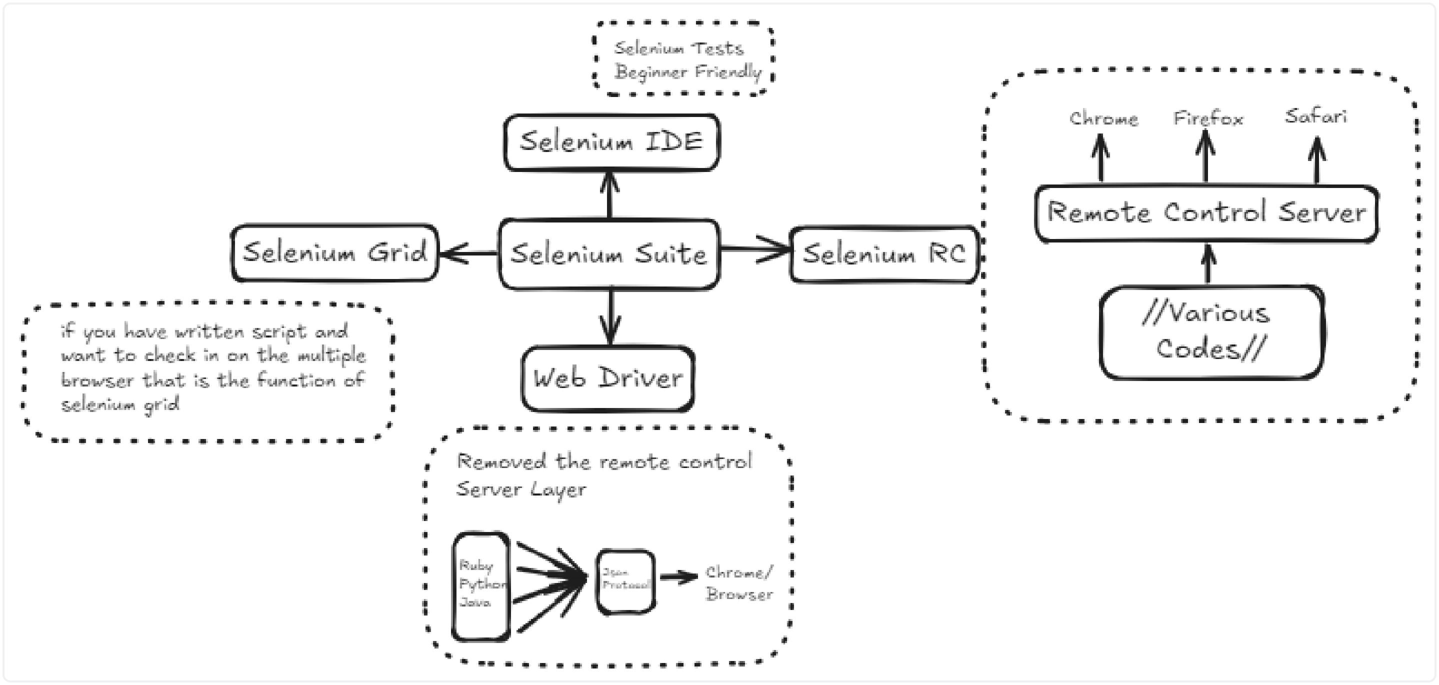
1. Selenium WebDriver

Selenium is Used to automate the UI Testing

We can also integrate selenium with different IDES.

Advantages of Selenium:

- Web Based Automation Tool
- Pen Source
- UI Automation
- Support Multiple OS
- Multiple Browsers
- Multiple languages


Selenium Tests Beginner Friendly
Selenium IDE
Selenium Grid
Selenium Suite
Selenium RC
Web Driver
if you have written script and want to check in on the multiple browser that is the function of selenium grid
Chrome
Firefox
Safari
Remote Control Server
//Various Codes//
Removed the remote control Server Layer
Ruby Python Java
Json Protocol
Chrome/ Browser

Everything Selenium does is send the browser commands to do something or send requests for information. Most of what you'll do with Selenium is a combination of these basic commands

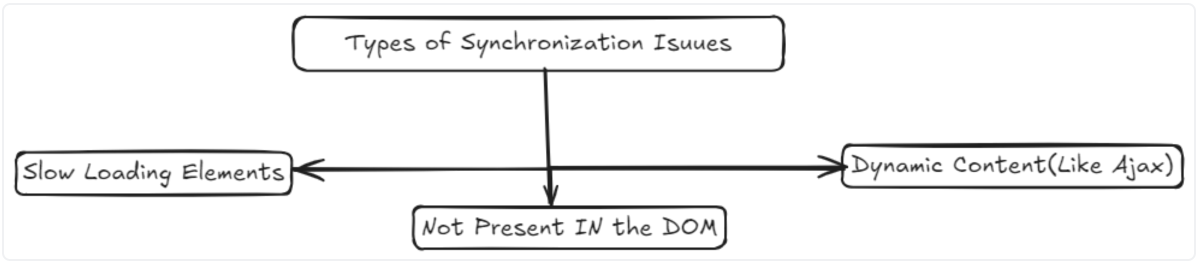
A very Basic Selenium Code

```
private WebDriver driver; //defining private so can use further
@BeforeEach
public void setup(){

System.setProperty(”webdriver.chrome.driver”,”C:¥¥Users¥¥prabhatritesh_chaube¥¥Downloads¥¥chromedriver-
win64¥¥chromedriver.exe”);
    driver = new ChromeDriver();
    driver.manage().window().maximize();
}

@Test
    public void selectone(){
        driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(20));
        driver.get(”https://app.endtest.io/guides/docs/how-to-test-dropdowns/”);
        WebElement select = driver.findElement(By.id(”pets”));
        Select select1 = new Select(select);
        select1.selectByValue(”rabbitt”);
//        select1.selectByVisibleText(”Rabbit”);
//        select1.selectByIndex(5);

    }
 @AfterEach
    public void exit(){
        driver.quit();
    }
}
```

2. Synchronization

What are Synchronization Issues?

Synchronization issues arise when the WebDriver tries to interact with elements before they are ready (e.g., before they are loaded or clickable). These issues lead to flaky tests and frequent failures.

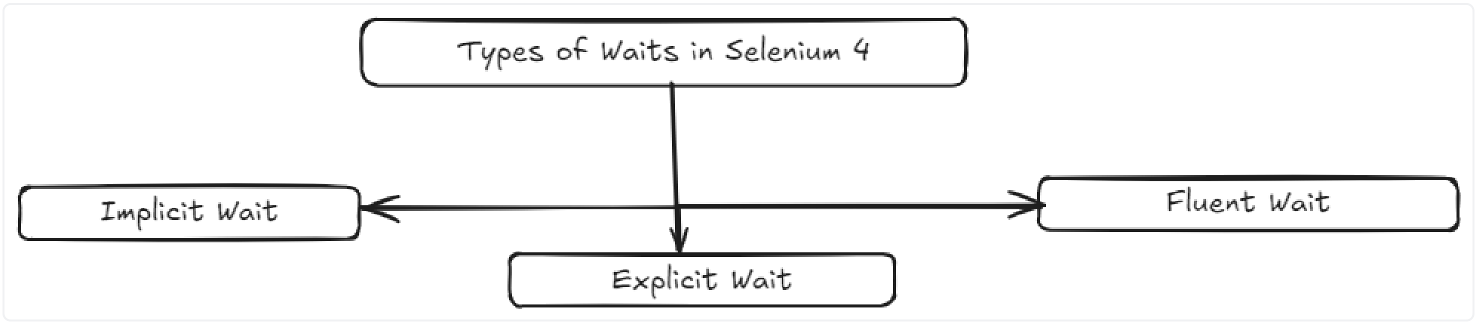


To ensure the WebDriver interacts with elements when they are in the correct state (loaded, clickable, etc.).

Common Solutions:

- General timeouts (implicit waits)
- Explicit waits
- Fluent waits
- Custom conditions

Waits



1.Implicit Wait

- Implicit Wait is used to set a default waiting time throughout your Selenium script. When you use implicit wait, it tells the WebDriver to wait for a certain amount of time before it throws a **NoSuchElementException**.

```
driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(20));
```

2.Explicit Wait

- Explicit Wait is used to wait for a specific condition to be true before proceeding further. It allows you to wait for a particular element or condition with a specific timeout.
- **Use Case:** Use explicit wait when you need to wait for a specific condition to occur before proceeding with the next step in the test script

```
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) {
        WebDriver driver = new ChromeDriver();
        driver.get("https://example.com");

        WebDriverWait wait = new WebDriverWait(driver, 20);
        WebElement element
=wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("exampleId")));

        // Perform actions on the element
        element.click();

        driver.quit();
    }
}
```

3.Fluent Wait

- Fluent Wait is similar to explicit wait but with more flexibility. You can define the maximum amount of time to wait for a condition, as well as the frequency with which to check the condition, and you can also ignore specific types of exceptions while waiting.
- **Use Case**: Use fluent wait when you need more control over the wait conditions, such as checking for the condition at regular intervals and ignoring certain exceptions.

```
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.FluentWait;
import java.time.Duration;
import java.util.NoSuchElementException;

public class FluentWaitExample {
    public static void main(String[] args) {
        WebDriver driver = new ChromeDriver();
        driver.get("https://example.com");

        FluentWait<WebDriver> wait = new FluentWait<>(driver)
                .withTimeout(Duration.ofSeconds(30))
                .pollingEvery(Duration.ofSeconds(5))
                .ignoring(NoSuchElementException.class);

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

        // Perform actions on the element
        element.click();

        driver.quit();
    }
}
```

As we have all the wait types, why cant we just use the **implicitlyWait** all the time?

Limitations of implicitlyWait:

- It applies globally and cannot target specific conditions.
- It might not be efficient in scenarios with dynamic content.
- It could cause unnecessary waiting in situations where the element is ready sooner.

Implicit Wait	**Explicit Wait**
Global wait applied to all element searches.	Waits for specific conditions before proceeding.
Affects all elements in the WebDriver instance.	Applied to specific elements or conditions only.
`driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);`	`WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));` `wait.until(ExpectedConditions.visibilityOfElementLo` `y.id` `("elementId")));`
Less control – waits for all elements globally.	More control – can wait for specific elements or conditions like visibility, clickability, etc.
Universal timeout for all elements.	Waits for a specific timeout and only for a specifi condition.
None – only waits for elements to appear.	Can wait for various conditions (visibility, presenc clickability, custom conditions, etc.).

Can slow down tests if elements load faster than the wait time.	More efficient, as it waits only for specific conditi then proceeds.
Will throw `NoSuchElementException` if an element is not found within the time limit.	Will throw `TimeoutException` if the condition is not within the specified time.
Should generally avoid combining with explicit waits to prevent conflicts.	Works best when used without implicit waits, for control of wait logic.

3. Locators

In Selenium WebDriver, **locators** and **selectors** are used to find and interact with elements on a webpage. Locating elements accurately is essential for automating UI testing.

1. Locators

WebDriver provides different types of locators to find elements in the DOM (Document Object Model). Some common locators include:

- **ID**: driver.findElement(By.id("submit-button"));
- **Name**: driver.findElement(By.name("username"));
- **Class Name**: driver.findElement(By.className("login-form"));
- **Tag Name**: driver.findElement(By.tagName("button"));
- **Link Text**: driver.findElement(By.linkText("Click Here"));
- **Partial Link Text**: driver.findElement(By.partialLinkText("Click"));
- **XPath**: driver.findElement(By.xpath("//button[@id='submit']"));
- **CSS Selector:** driver.findElement(By.cssSelector("input#username"));

2. DOM (Document Object Model)

The **DOM** represents the hierarchical structure of a webpage. Each HTML element on the page is a node in the DOM. WebDriver uses the DOM to locate elements. When a test interacts with an element (e.g., clicking a button), it's manipulating the corresponding element in the DOM.

Key concepts of the DOM:

- **Nodes**: Represent elements, attributes, or text within HTML.

- **Parent/Child Relationships**: DOM elements have parent-child or sibling relationships, which can be traversed to locate elements.

3. XPath

XPath (XML Path Language) is a query language used to locate elements in an XML-like structure such as the DOM.

Types of XPath:

- **Absolute XPath**: Provides a direct path from the root element.

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

Relative XPath: Starts from the current element or somewhere in the middle of the DOM.

```
//input[@name='username']
```

4. CSS Selectors

CSS selectors provide a concise way to select elements based on their attributes, id, or class. CSS selectors tend to be faster than XPath and are preferred when possible.

- **Element by ID**:driver.findElement(By.cssSelector("#username"));
- **Element by Class**:driver.findElement(By.cssSelector(".login-button"));
- **Element by Attribute:**driver.findElement(By.cssSelector("input[name='password']"));
- **Combining selectors**:driver.findElement(By.cssSelector("div#container .button"));

5. Reliable Locators

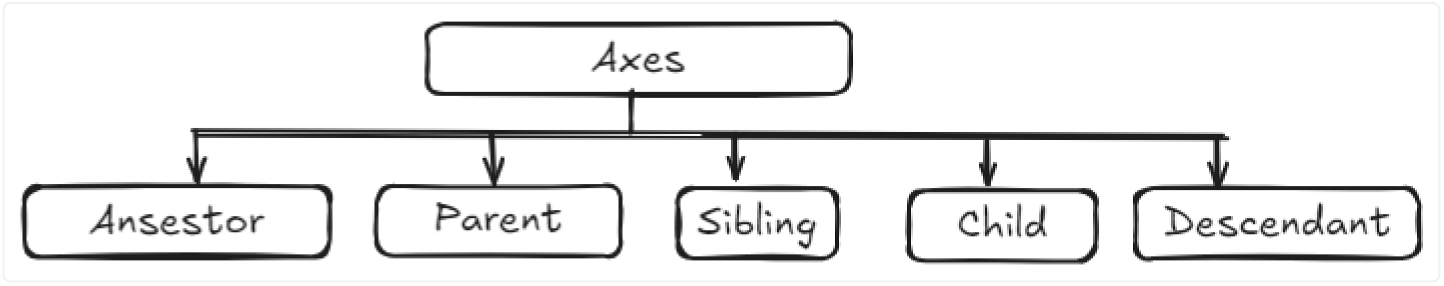
Choosing reliable locators is key to making your tests robust. Some tips for reliable locators:

- **Prefer IDs**: If an element has an `id`, it should be your first choice since it's unique.
- **Use Descriptive XPath or CSS Selectors**: Avoid using absolute paths like `/html/body/div`. Instead, prefer relative paths like `//div[@id='header']//input`.
- **Use Classes Carefully**: Classes may be shared across multiple elements, so ensure they are unique in the context of the element you're targeting.
- **Avoid Text-Based Locators**: `linkText()` or `partialLinkText()` may be unreliable if the text changes frequently.
- **Aria Labels**: Modern applications use ARIA labels, which can be used for accessible, stable locators:

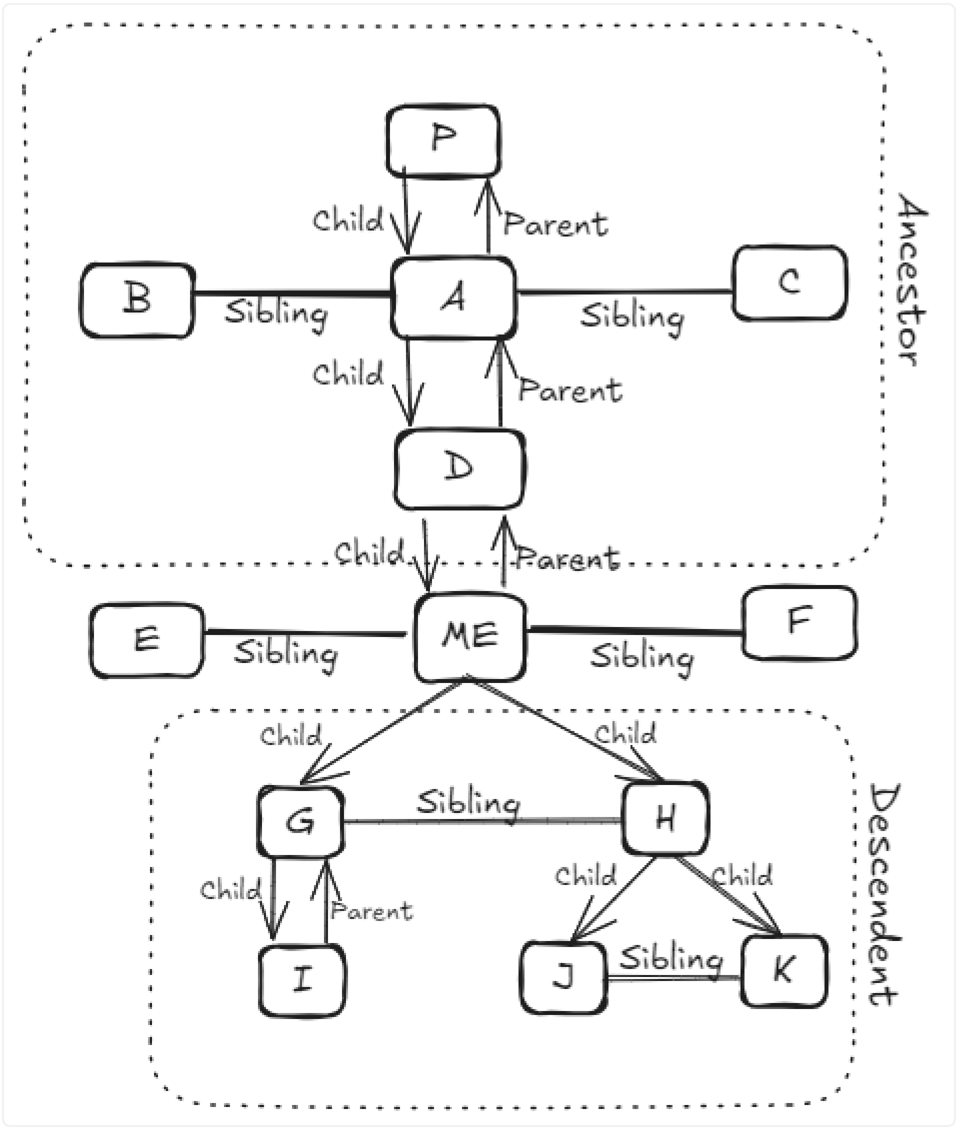
4. Xpath Axes

we use this where is no unique attribute attached to anything that we want to target.

So we act like the user to get most of the axes.



Now lets see the relationship with the element.



Ancestor

- ancestor
- ancestor-or-self

Descendant

- descendant
- descendant-or-self

Sibling

- Preceding-sibling
- following-sibling

Parent

- parent

Child

- child

```
//lable[text()='Email']/following-sibling::input[1]/Parent::div
```

```
//div[@class='container']/child::input[@type='text']
```

5. Page Object

POM

It is a design pattern in selenium that creates an Object Repository for storing all web elements.
Advantages:

- Makes code maintainable
- Makes code Readable
- Makes code reusable

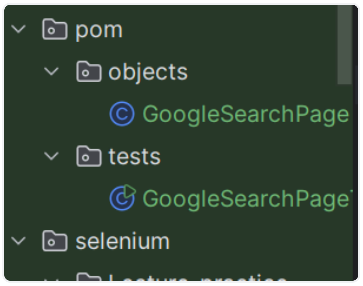
Basic Structure:

- Page Object Class: Contains methods that represent actions a user can perform on the page.
- Locators: Encapsulated within the page class, never exposed directly.
- Utility Methods: Should represent high-level actions (e.g., `login()`).

Rules:

- Avoid assertions inside page objects.
- Avoid overloading page objects with business logic.
- Keep interaction logic in page objects and validation in tests.

The Structure a POM folllows



Now the Objects would have the classes to interact with the element of the page we are looking for

```
package pom.objects;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
public class GoogleSearchPage {
    private static WebElement element;
    public static  WebElement searchBoxfilller(WebDriver driver){
        element = driver.findElement(By.name("q"));
        return element;
    }
    public static WebElement clickSearch(WebDriver driver){
        element = driver.findElement(By.name("btnK"));
        return element;
    }
}
```

And the test would obviosly be testing it, make surte to import the page of objects

```
package pom.tests;
import com.sun.xml.bind.v2.runtime.reflect.Lister;
import org.junit.jupiter.api.BeforeEach;
import org.openqa.selenium.Keys;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.annotations.BeforeSuite;
import pom.objects.GoogleSearchPage;

public class GoogleSearchPageTest {
    private static WebDriver driver;
    public static void main(String[] args) {

System.setProperty("webdriver.chrome.driver","C:¥¥Users¥¥prabhatritesh_chaube¥¥Downloads¥¥chromedriver-
win64¥¥chromedriver.exe");
        driver = new ChromeDriver();
        driver.get("https://www.google.co.in");
        driver.manage().window().maximize();

        GoogleSearchPage.searchBoxfilller(driver).sendKeys("Bae on beach");
        GoogleSearchPage.clickSearch(driver).sendKeys(Keys.RETURN);

        driver.quit();
    }
}
```

Now we can go a step ahead and we can directly make all the functioning in one object class and test it in another.

And the Steps for the Same are:

a. Create A Class for each WebPage
b. Add object Locators
c. Add action Methods
d. Create Class for test case
e. Create object for page class
f. Refer Action Methods

and in that we Use

```
WebDriver driver;
public GoogleSearchUsingObject(WebDriver driver){
    this.driver = driver;
}
public static void main(String[] args) {

}
By locatebutton = By.xpath("//div[@class=¥"logo¥"]");
public  void clickbutton(WebDriver driver){
    driver.findElement(locatebutton).click();

}
```

and for the test we use :

```
private static WebDriver driver;
public static void main(String[] args) {

System.setProperty("webdriver.chrome.driver","C:¥¥Users¥¥prabhatritesh_chaube¥¥Downloads¥¥chromedriver-win64¥¥chromedriver.exe");
    driver = new ChromeDriver();
    driver.get("https://www.geeksforgeeks.org/problems/peak-element/1");
    GoogleSearchUsingObject performer = new GoogleSearchUsingObject(driver);
```

```
        performer.clickbutton(driver);
    }
```

Page Factory

page factory is used so that it can reduce the lines of codes, where pom is a way to separate objects and scripts the page factory is responsible for implementing it.

and we Use **@FindBY** and **@FindBYs** in the page factory

```
    @FindBy(xpath = ”//div[@class=¥”logo¥”]”) public WebElement clicker;
```

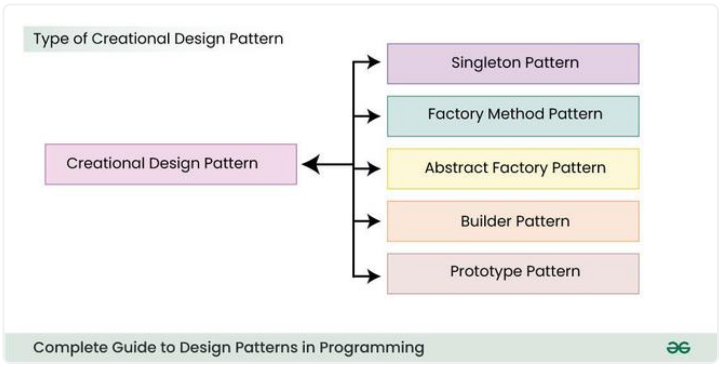
FindBY	FIndBys
Used to locate a single web element based on a specific locator strategy.	Used when you need to locate an element based on multiple conditions **sequentially**. It's essentially an "AND" condition between multiple locators.
@FindBy(id = "username")private WebElement usernameField;	@FindBys({ @FindBy(className = "form-group"), @FindBy(id = "username")}) private WebElement usernameField;

How can the POM be an antipattern:

- When Page Objects become too bloated or complex, with too many methods
- Sometimes, developers introduce methods in Page Objects that perform actions that span across multiple pages, which leads to Page Objects that are no longer representing a single web page.

Java Design Patterns : Creatonal

Creational design patterns are a category of design patterns in software development that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. They abstract the instantiation process and make it more flexible by decoupling the code from the actual object creation. These patterns promote reusability, scalability, and the efficient management of objects.


Type of Creational Design Pattern
Creational Design Pattern
Singleton Pattern
Factory Method Pattern
Abstract Factory Pattern
Builder Pattern
Prototype Pattern
Complete Guide to Design Patterns in Programming

Automated testing: API Automation

API is the acronym for application programming interface — a software intermediary that allows two applications to talk to each other. APIs are an accessible way to extract and share data within and across organizations.

API testing is basically black box testing which is simply concerned with the final output of the system under test. 1. Unit testing aims to verify whether the module delivers the required functionality. The development team monitors unit testing activity and makes necessary changes wherever required.

Response Codes (200, 404, 500, etc.)
1XX**Informational** – : Communicates transfer protocol-level information.
2XX **Success** – :Indicates that the client's request was accepted successfully.
3XX**Redirection** – :Indicates that the client must take some additional action in order to complete their request.
4XX**Client Error** – : This category of error status codes points the finger at clients.
5XX **Server Error** –:The server takes responsibility for these error status codes.

Http Get method

Used to request data from a specified resource.

```
@Test
    public void anotherget(){
        String response = given()
        .pathParam("id", 21)   // Step 3: set path parameter "id" to 21
        .when()                // Step 4: start defining the request
        .get("https://petstore.swagger.io/v2/pet/{id}") // Step 5: send a GET request to the specified
URL
        .then()                 // Step 6: start validating the response
        .statusCode(200)       // Step 7: assert that the HTTP status code is 200
        .body("id", equalTo(21)) // Step 8: assert that the JSON response contains an "id" field with a
value of 21
        .extract()             // Step 9: extract the response after validations
        .asPrettyString();
        System.out.println(response);
    }
```

HTTPS Post Method

Used to send data to create a new resource on the server.

```
@Test
public void createPet() {
    // Define the JSON body
    String newPet = "{¥n" +
            "    ¥"id¥": 21,¥n" +
            "    ¥"category¥": {¥n" +
            "        ¥"id¥": 0,¥n" +
            "        ¥"name¥": ¥"string¥"¥n" +
            "    },¥n" +
            "    ¥"name¥": ¥"doggie¥",¥n" +
            "    ¥"photoUrls¥": [¥n" +
            "        ¥"string¥"¥n" +
            "    ],¥n" +
            "    ¥"tags¥": [¥n" +
            "        {¥n" +
            "            ¥"id¥": 0,¥n" +
            "            ¥"name¥": ¥"string¥"¥n" +
            "        }¥n" +
            "    ],¥n" +
            "    ¥"status¥": ¥"available¥"¥n" +
            "}";

    // Send the POST request
    String response = given()
            .header("Content-Type", "application/json")  // Specify that the request body is JSON
            .body(newPet)  // Attach the JSON body to the request
            .when()
            .post("https://petstore.swagger.io/v2/pet")  // Send a POST request to create a new pet
            .then()
            .statusCode(200)  // Assert that the response status code is 200 (OK)
            .body("id", equalTo(21))  // Assert that the returned pet ID is 21
            .extract()
            .asPrettyString();  // Extract the response and format it as a pretty string

    // Print the response
    System.out.println(response);
}
```

HTTP Put Method

Used to update an existing resource.

SO when ever we need any update in the details then we use the Put Method.

- Use `POST` when you want to **add** a new resource.
- Use `PUT` when you want to **update** an existing resource (or create one at a specific URI).

The code of put and post is almost same, the only difference is that we got to use PUT isnstead of POST method.

HTTP Delete Method

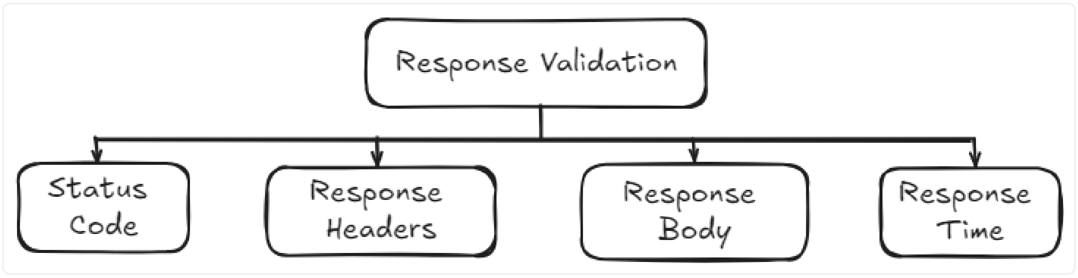
Delete is used to delete any existing data from the data set.

```
Response response = given()
        .pathParam(”id”,21)
        .when()
        .delete(”https://petstore.swagger.io/v2/pet/{id}”)
        .then()
        .statusCode(200)
        .log().all()
        .extract().response();
```

Making API requests involves sending data to or retrieving data from a server, which can be automated using RestAssured in Java. By mastering the use of methods like GET, POST, PUT, and DELETE, and learning to validate responses, you can build robust and reliable API tests.

Validating the response in RestAssured Framework

Validating API responses is a crucial part of automated API testing, ensuring that the server returns the expected output based on the request sent.



Validating Status Code:

Status codes are the most basic form of validation. Every HTTP request returns a specific status code to indicate success, failure, or errors.

```
import io.restassured.RestAssured;
import static io.restassured.RestAssured.*;

public class ValidateStatusCode {

    public static void main(String[] args) {
        // Base URI of the API
        RestAssured.baseURI = "https://api.example.com";

        // Validate status code is 200 OK
        given()
            .when()
                .get("/students")
            .then()
                .statusCode(200)  // Validate status code
                .log().all();     // Log the response
    }
}
```

`.statusCode(200)`: This checks that the response status code is `200 OK`

Common Status Codes:

- `200 OK`: Request was successful.
- `201 Created`: Resource was created successfully (common for POST requests).
- `404 Not Found`: The requested resource does not exist.
- `500 Internal Server Error`: A server error occurred.

Validating Response Headers

Response headers provide important information about the response, such as content type, cache control, and authorization status.

```
import io.restassured.RestAssured;
import static io.restassured.RestAssured.*;

public class ValidateHeaders {

    public static void main(String[] args) {
        // Base URI of the API
        RestAssured.baseURI = "https://api.example.com";

        // Validate specific headers in the response
        given()
            .when()
                .get("/students/1")
            .then()
                .statusCode(200)
                .header("Content-Type", "application/json")  // Validate Content-Type header
                .header("Cache-Control", "no-cache")         // Validate Cache-Control header
                .log().all();
    }
}
```

Validating Response Body

The response body is the most important part of API validation, especially when working with JSON or XML data. We typically validate that the data returned matches our expectations.

so suppose we have a response for a specific request :

```
{
    "id": 1,
    "name": "John Doe",
    "age": 20,
    "class": "12th Grade"
}
```

Now we want to validate is the response is correct or not then,

```
import io.restassured.RestAssured;
import static io.restassured.RestAssured.*;
import static org.hamcrest.Matchers.*;

public class ValidateJsonResponse {

    public static void main(String[] args) {
        // Base URI of the API
        RestAssured.baseURI = "https://api.example.com";

        // Validate response body content
        given()
            .when()
                .get("/students/1")
            .then()
                .statusCode(200)
                .body("id", equalTo(1))               // Validate student ID is 1
                .body("name", equalTo("John Doe"))    // Validate name
                .body("age", equalTo(20))             // Validate age
                .body("class", equalTo("12th Grade")) // Validate class
                .log().all();                         // Log the entire response
    }
}
```

and when we have any nested response like

```
{
    "id": 1,
    "name": "John Doe",
    "age": 20,
    "address": {
        "city": "New York",
        "zip": "10001"
    }
}
```

So now as we can see address have two responses to look for city and Zip, so now what we would do is

```
.statusCode(200)
.body("address.city", equalTo("New York"))  // Validate city in nested object
.body("address.zip", equalTo("10001"));     // Validate zip code
```

Validating Response Time

API performance is crucial, and we often need to ensure that the response time is within acceptable limits.

```
import io.restassured.RestAssured;
import static io.restassured.RestAssured.*;

public class ValidateResponseTime {

    public static void main(String[] args) {
        // Base URI of the API
        RestAssured.baseURI = "https://api.example.com";

        // Validate that the response time is less than 2000 ms (2 seconds)
        given()
            .when()
                .get("/students")
            .then()
                .statusCode(200)
                .time(lessThan(2000L))  // Validate response time in milliseconds
```

```
                    .log().all();
        }
    }
```

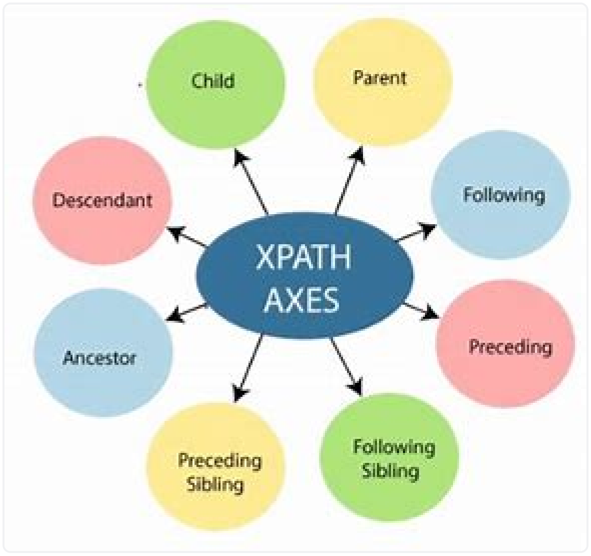
This code should be executed in or before 2000 milliseconds, or else it would fail the test

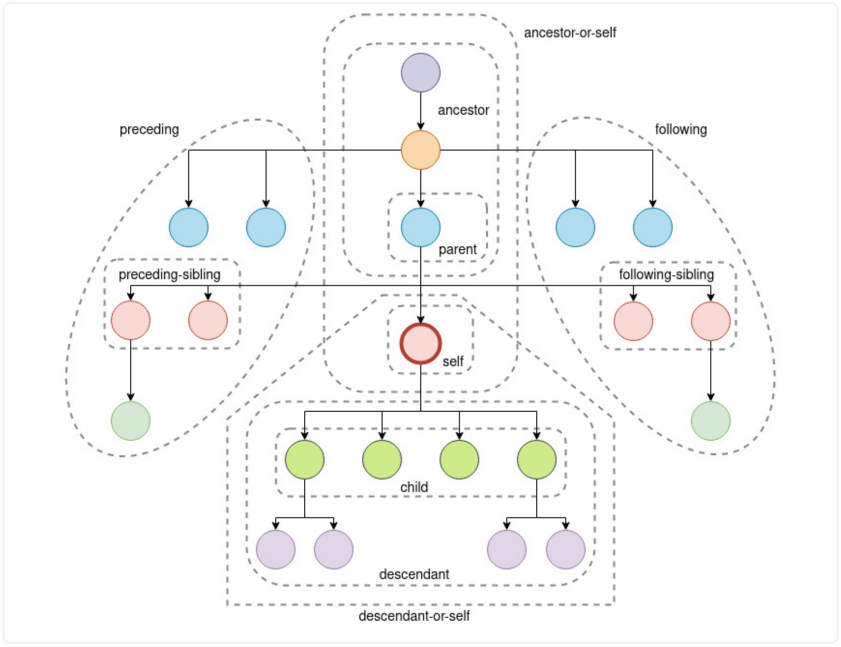
Difference in Query Parameters and Path Paramenter

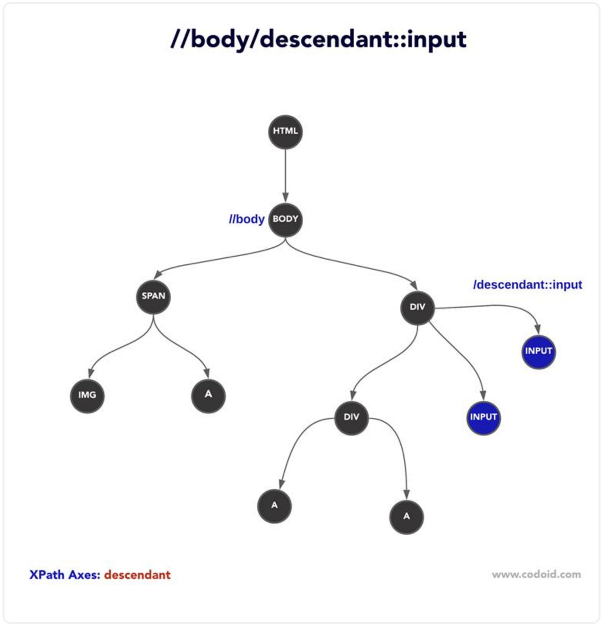
Path Paramenters	Query Parameters
Used to specify required values, such as resource IDs	Used to pass optional information (filters, pagination)
`/resource/{id}` (e.g., `/products/12345`)	`?key1=value1&key2=value2` (e.g., `?category=electronics`)

Dynamic X path

In Selenium, a **Dynamic XPath** refers to an XPath expression that is constructed to handle web elements whose attributes or positions may change dynamically during runtime. This is common in modern web applications where elements often have dynamic IDs, classes, or are generated dynamically.







Design Patterns

Singelton Design Pattern

Its ensure that only One instance should be created so that we can not created multiple instance for better performance.

The most important three rules to follow

- Private Constructor
- Static Member
- Static Method

- [ ] modifiy the code in terms of singleton Desing Pattern

Types of Design Patterns

- Factory(Creational Design Patterns)

  as the factory function, Factory Implementation focuses on making and using the functionalities in a way that the logic is not hold the value of transparency.
- Singleton
- Builder Design Pattern

  In the same sense as of building, the builder Design Pattern focuses on building/initializing the methodologies and function

builder

factory

singleton

Pom

Page Factory

refactoring.guru

Untitled

Selenium WebDriver

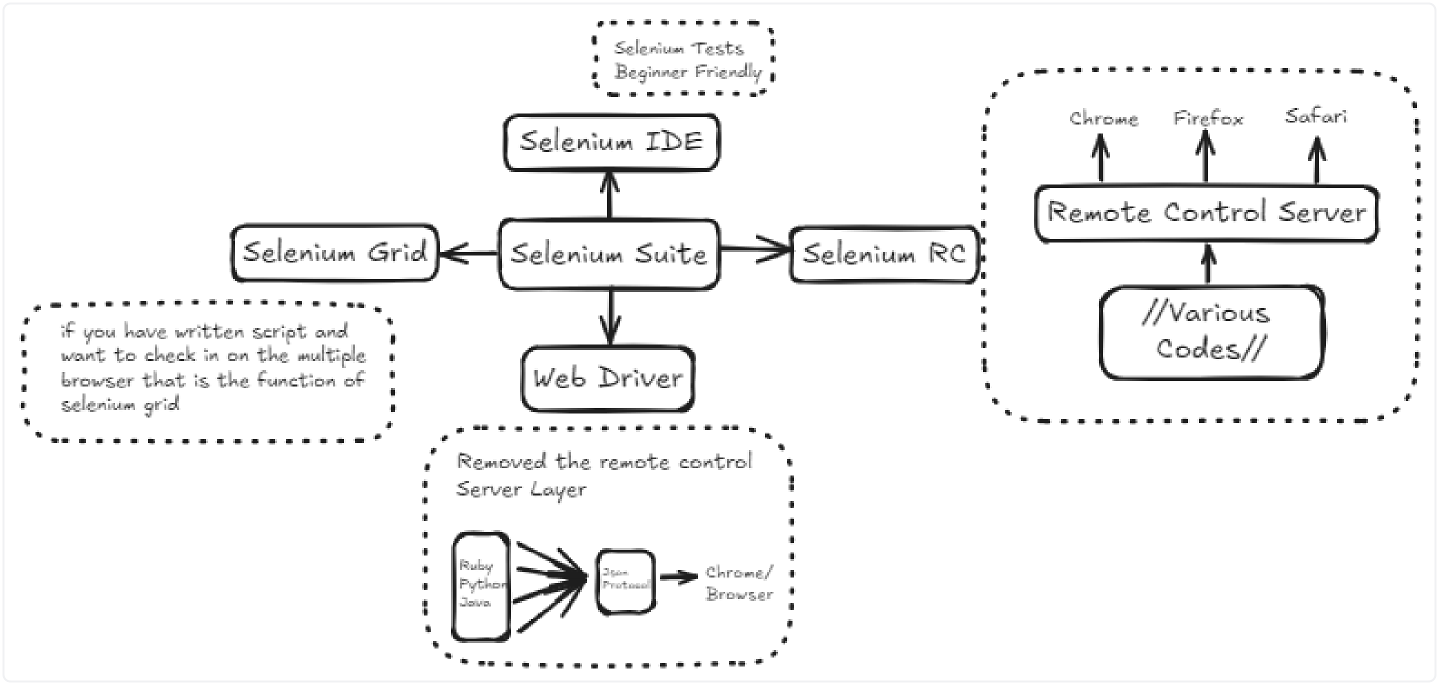
1. Selenium WebDriver

Selenium is Used to automate the UI Testing

We can also integrate selenium with different IDES.

Advantages of Selenium:

- Web Based Automation Tool
- Pen Source
- UI Automation
- Support Multiple OS
- Multiple Browsers
- Multiple languages


Selenium Tests Beginner Friendly
Selenium IDE
Selenium Grid
Selenium Suite
Selenium RC
Web Driver
if you have written script and want to check in on the multiple browser that is the function of selenium grid
Chrome
Firefox
Safari
Remote Control Server
//Various Codes//
Removed the remote control Server Layer
Ruby
Python
Java
Json Protocol
Chrome/ Browser

Everything Selenium does is send the browser commands to do something or send requests for information. Most of what you'll do with Selenium is a combination of these basic commands

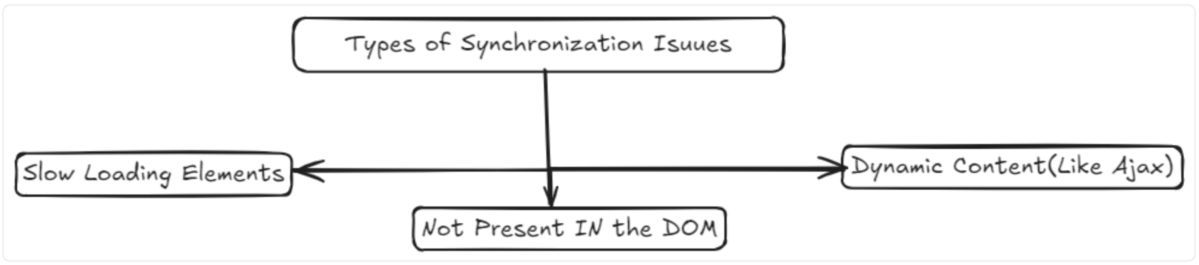
A very Basic Selenium Code

```
private WebDriver driver; //defining private so can use further
@BeforeEach
public void setup(){

System.setProperty("webdriver.chrome.driver","C:¥¥Users¥¥prabhatritesh_chaube¥¥Downloads¥¥chromedriver-
win64¥¥chromedriver.exe");
    driver = new ChromeDriver();
    driver.manage().window().maximize();
}

@Test
    public void selectone(){
        driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(20));
        driver.get("https://app.endtest.io/guides/docs/how-to-test-dropdowns/");
        WebElement select = driver.findElement(By.id("pets"));
        Select select1 = new Select(select);
        select1.selectByValue("rabbitt");
//        select1.selectByVisibleText("Rabbit");
//        select1.selectByIndex(5);

    }
 @AfterEach
    public void exit(){
        driver.quit();
    }
}
```

2. Synchronization

What are Synchronization Issues?

Synchronization issues arise when the WebDriver tries to interact with elements before they are ready (e.g., before they are loaded or clickable). These issues lead to flaky tests and frequent failures.

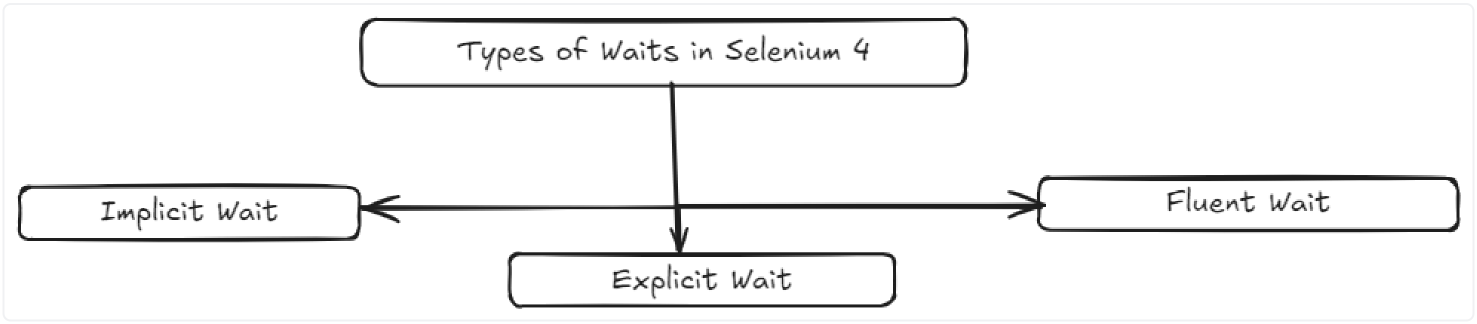


To ensure the WebDriver interacts with elements when they are in the correct state (loaded, clickable, etc.).

Common Solutions:

- General timeouts (implicit waits)
- Explicit waits
- Fluent waits
- Custom conditions

Waits



1.Implicit Wait

- Implicit Wait is used to set a default waiting time throughout your Selenium script. When you use implicit wait, it tells the WebDriver to wait for a certain amount of time before it throws a **NoSuchElementException**.

```
driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(20));
```

2.Explicit Wait

- Explicit Wait is used to wait for a specific condition to be true before proceeding further. It allows you to wait for a particular element or condition with a specific timeout.
- **Use Case:** Use explicit wait when you need to wait for a specific condition to occur before proceeding with the next step in the test script

```
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) {
        WebDriver driver = new ChromeDriver();
        driver.get("https://example.com");

        WebDriverWait wait = new WebDriverWait(driver, 20);
        WebElement element
=wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("exampleId")));

        // Perform actions on the element
        element.click();

        driver.quit();
    }
}
```

3.Fluent Wait

- Fluent Wait is similar to explicit wait but with more flexibility. You can define the maximum amount of time to wait for a condition, as well as the frequency with which to check the condition, and you can also ignore specific types of exceptions while waiting.
- **Use Case**: Use fluent wait when you need more control over the wait conditions, such as checking for the condition at regular intervals and ignoring certain exceptions.

```
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.FluentWait;
import java.time.Duration;
import java.util.NoSuchElementException;

public class FluentWaitExample {
    public static void main(String[] args) {
        WebDriver driver = new ChromeDriver();
        driver.get("https://example.com");

        FluentWait<WebDriver> wait = new FluentWait<>(driver)
                .withTimeout(Duration.ofSeconds(30))
                .pollingEvery(Duration.ofSeconds(5))
                .ignoring(NoSuchElementException.class);

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

        // Perform actions on the element
        element.click();

        driver.quit();
    }
}
```

As we have all the wait types, why cant we just use the **implicitlyWait** all the time?

Limitations of implicitlyWait:

- It applies globally and cannot target specific conditions.
- It might not be efficient in scenarios with dynamic content.
- It could cause unnecessary waiting in situations where the element is ready sooner.

Implicit Wait	**Explicit Wait**
Global wait applied to all element searches.	Waits for specific conditions before proceeding.
Affects all elements in the WebDriver instance.	Applied to specific elements or conditions only.
`driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);`	`WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));` `wait.until(ExpectedConditions.visibilityOfElementLo` `y.id` `("elementId")));`
Less control – waits for all elements globally.	More control – can wait for specific elements or conditions like visibility, clickability, etc.
Universal timeout for all elements.	Waits for a specific timeout and only for a specifi condition.
None – only waits for elements to appear.	Can wait for various conditions (visibility, presen clickability, custom conditions, etc.).

Can slow down tests if elements load faster than the wait time.	More efficient, as it waits only for specific conditi then proceeds.
Will throw `NoSuchElementException` if an element is not found within the time limit.	Will throw `TimeoutException` if the condition is not within the specified time.
Should generally avoid combining with explicit waits to prevent conflicts.	Works best when used without implicit waits, for control of wait logic.

3. Locators

In Selenium WebDriver, **locators** and **selectors** are used to find and interact with elements on a webpage. Locating elements accurately is essential for automating UI testing.

1. Locators

WebDriver provides different types of locators to find elements in the DOM (Document Object Model). Some common locators include:

- **ID**: driver.findElement(By.id("submit-button"));
- **Name**: driver.findElement(By.name("username"));
- **Class Name**: driver.findElement(By.className("login-form"));
- **Tag Name**: driver.findElement(By.tagName("button"));
- **Link Text**: driver.findElement(By.linkText("Click Here"));
- **Partial Link Text**: driver.findElement(By.partialLinkText("Click"));
- **XPath**: driver.findElement(By.xpath("//button[@id='submit']"));
- **CSS Selector:** driver.findElement(By.cssSelector("input#username"));
    - .className
    - .className
    - [type="text"]

2. DOM (Document Object Model)

The **DOM** represents the hierarchical structure of a webpage. Each HTML element on the page is a node in the DOM. WebDriver uses the DOM to locate elements. When a test interacts with an element (e.g., clicking a button), it's manipulating the corresponding element in the DOM.

Key concepts of the DOM:

- **Nodes**: Represent elements, attributes, or text within HTML.
- **Parent/Child Relationships**: DOM elements have parent-child or sibling relationships, which can be traversed to locate elements.

3. XPath

XPath (XML Path Language) is a query language used to locate elements in an XML-like structure such as the DOM.

Types of XPath:

- **Absolute XPath**: Provides a direct path from the root element.

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

Relative XPath: Starts from the current element or somewhere in the middle of the DOM.

```
//input[@name='username']
```

4. CSS Selectors

CSS selectors provide a concise way to select elements based on their attributes, id, or class. CSS selectors tend to be faster than XPath and are preferred when possible.

- **Element by ID**:driver.findElement(By.cssSelector("#username"));
- **Element by Class**:driver.findElement(By.cssSelector(".login-button"));
- **Element by Attribute:**driver.findElement(By.cssSelector("input[name='password']"));
- **Combining selectors**:driver.findElement(By.cssSelector("div#container .button"));

5. Reliable Locators

Choosing reliable locators is key to making your tests robust. Some tips for reliable locators:

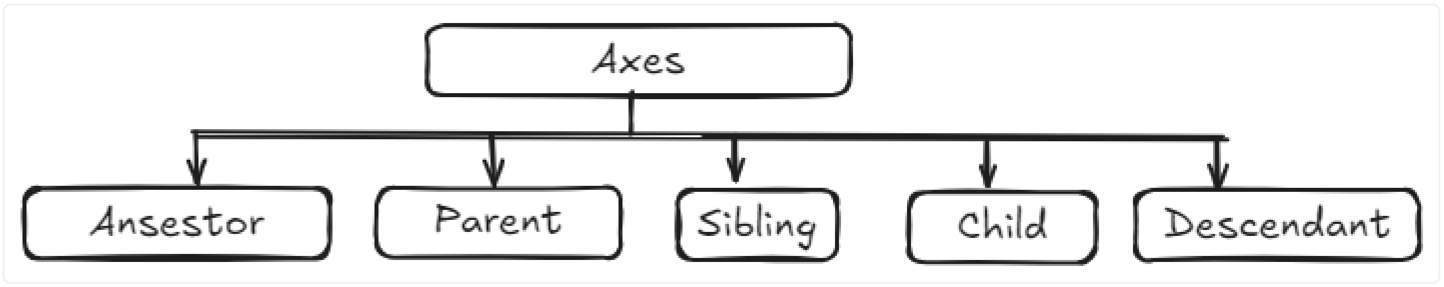
- **Prefer IDs**: If an element has an `id`, it should be your first choice since it's unique.
- **Use Descriptive XPath or CSS Selectors**: Avoid using absolute paths like `/html/body/div`. Instead, prefer relative paths like `//div[@id='header']//input`.
- **Use Classes Carefully**: Classes may be shared across multiple elements, so ensure they are unique in the context of the element you're targeting.
- **Avoid Text-Based Locators**: `linkText()` or `partialLinkText()` may be unreliable if the text changes frequently.
- **Aria Labels**: Modern applications use ARIA labels, which can be used for accessible, stable

locators:

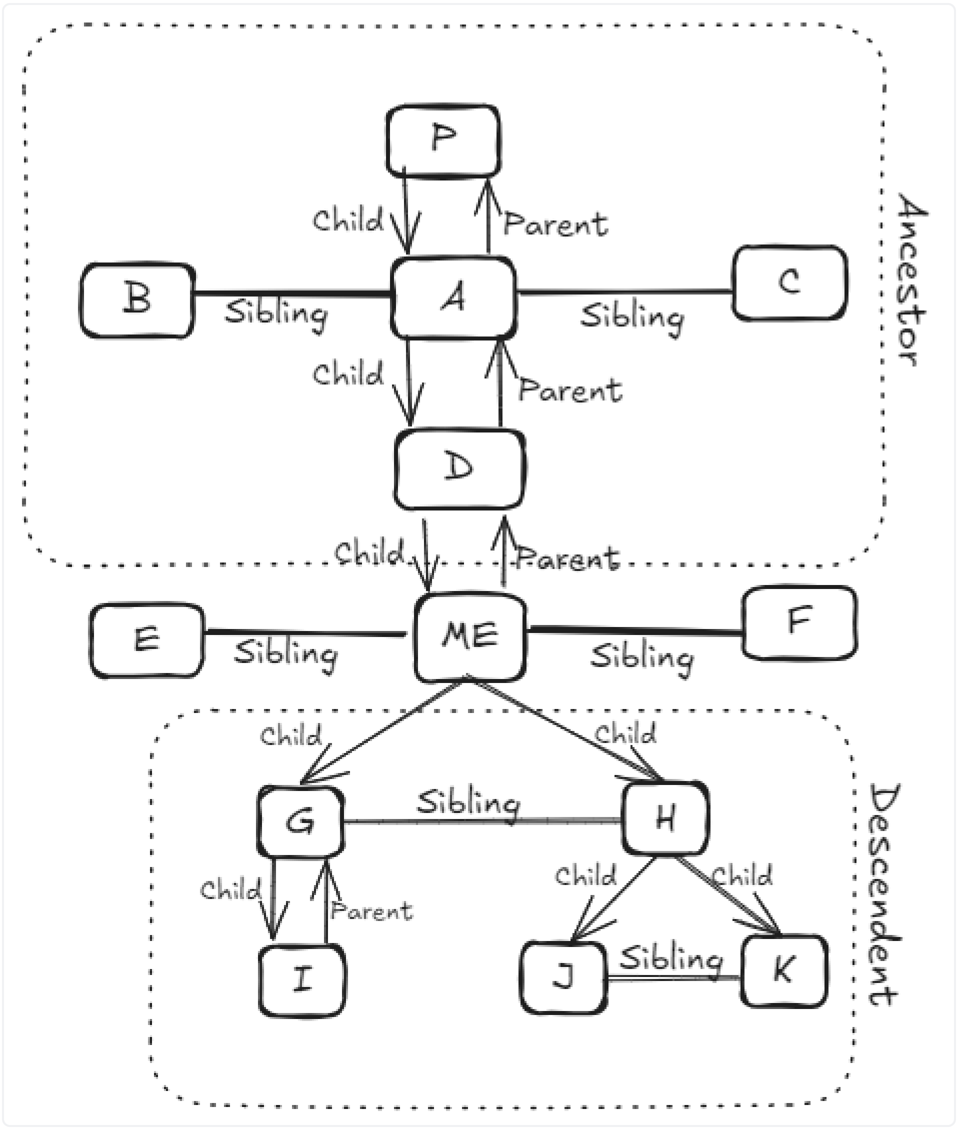
4. Xpath Axes

we use this where is no unique attribute attached to anything that we want to target.

So we act like the user to get most of the axes.



Now lets see the relationship with the element.



Ancestor

- ancestor
- ancestor-or-self

Descendant

- descendant
- descendant-or-self

Sibling

- Preceding-sibling
- following-sibling

Parent

- parent

Child

- child

```
//lable[text()='Email']/following-sibling::input[1]/Parent::div
```

```
//div[@class='container']/child::input[@type='text']
```

5. Page Object

POM

It is a design pattern in selenium that creates an Object Repository for storing all web elements.

Advantages:

- Makes code maintainable
- Makes code Readable
- Makes code reusable

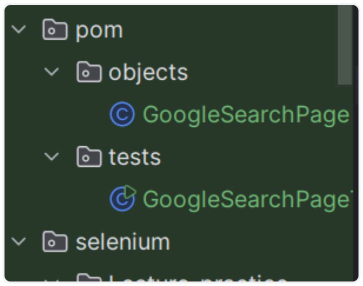
Basic Structure:

- Page Object Class: Contains methods that represent actions a user can perform on the page.
- Locators: Encapsulated within the page class, never exposed directly.
- Utility Methods: Should represent high-level actions (e.g., `login()`).

Rules:

- Avoid assertions inside page objects.
- Avoid overloading page objects with business logic.
- Keep interaction logic in page objects and validation in tests.

The Structure a POM folllows



Now the Objects would have the classes to interact with the element of the page we are looking for

```
package pom.objects;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
public class GoogleSearchPage {
    private static WebElement element;
    public static  WebElement searchBoxfilller(WebDriver driver){
        element = driver.findElement(By.name("q"));
        return element;
    }
    public static WebElement clickSearch(WebDriver driver){
        element = driver.findElement(By.name("btnK"));
        return element;
    }
}
```

And the test would obviosly be testing it, make surte to import the page of objects

```
package pom.tests;
import com.sun.xml.bind.v2.runtime.reflect.Lister;
import org.junit.jupiter.api.BeforeEach;
import org.openqa.selenium.Keys;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.annotations.BeforeSuite;
import pom.objects.GoogleSearchPage;

public class GoogleSearchPageTest {
    private static WebDriver driver;
    public static void main(String[] args) {

System.setProperty("webdriver.chrome.driver","C:¥¥Users¥¥prabhatritesh_chaube¥¥Downloads¥¥chromedriver-
win64¥¥chromedriver.exe");
        driver = new ChromeDriver();
        driver.get("https://www.google.co.in");
        driver.manage().window().maximize();

        GoogleSearchPage.searchBoxfilller(driver).sendKeys("Bae on beach");
        GoogleSearchPage.clickSearch(driver).sendKeys(Keys.RETURN);

        driver.quit();
    }
}
```

Now we can go a step ahead and we can directly make all the functioning in one object class and test it in another.

And the Steps for the Same are:

a. Create A Class for each WebPage
b. Add object Locators
c. Add action Methods
d. Create Class for test case
e. Create object for page class
f. Refer Action Methods

and in that we Use

```
WebDriver driver;
public GoogleSearchUsingObject(WebDriver driver){
    this.driver = driver;
}
public static void main(String[] args) {

}
By locatebutton = By.xpath("//div[@class=¥"logo¥"]");
public  void clickbutton(WebDriver driver){
    driver.findElement(locatebutton).click();

}
```

and for the test we use :

```
private static WebDriver driver;
public static void main(String[] args) {

System.setProperty("webdriver.chrome.driver","C:¥¥Users¥¥prabhatritesh_chaube¥¥Downloads¥¥chromedriver-
win64¥¥chromedriver.exe");
    driver = new ChromeDriver();
    driver.get("https://www.geeksforgeeks.org/problems/peak-element/1");
    GoogleSearchUsingObject performer = new GoogleSearchUsingObject(driver);
```

```
        performer.clickbutton(driver);
    }
```

Page Factory

page factory is used so that it can reduce the lines of codes, where pom is a way to separate objects and scripts the page factory is responsible for implementing it.

and we Use **@FindBY** and **@FindBYs** in the page factory

```
    @FindBy(xpath = ”//div[@class=¥”logo¥”]”) public WebElement clicker;
```

FindBY	FIndBys
Used to locate a single web element based on a specific locator strategy.	Used when you need to locate an element based on multiple conditions **sequentially**. It's essentially an "AND" condition between multiple locators.
@FindBy(id = "username")private WebElement usernameField;	@FindBys({ @FindBy(className = "form-group"), @FindBy(id = "username")}) private WebElement usernameField;

How can the POM be an antipattern:

- When Page Objects become too bloated or complex, with too many methods
- Sometimes, developers introduce methods in Page Objects that perform actions that span across multiple pages, which leads to Page Objects that are no longer representing a single web page.

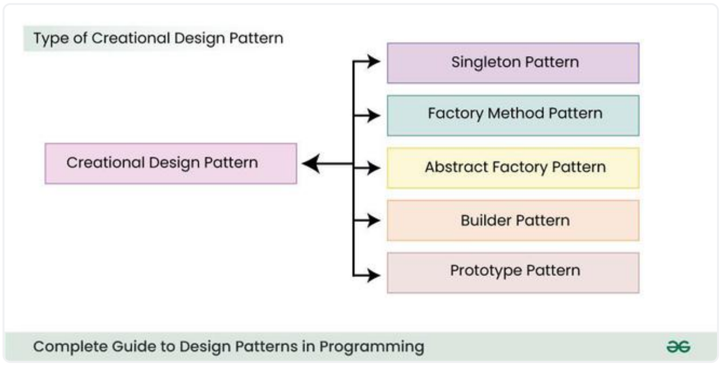
DIfference in Selenium Webdriver and Selenium RC

Selenium Webdriver	Selenium RC
Uses a simpler and more modern architecture; interacts directly with the browser.	Uses a server to translate commands from the client into browser commands.
Directly communicates with the browser via native OS methods.	Uses JavaScript to communicate with the browser through a server.

Supports all major browsers (Chrome, Firefox, IE, etc.) through browser-specific drivers.	Limited support; requires a server for each browser instance.

Java Design Patterns : Creatonal

Creational design patterns are a category of design patterns in software development that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. They abstract the instantiation process and make it more flexible by decoupling the code from the actual object creation. These patterns promote reusability, scalability, and the efficient management of objects.


Type of Creational Design Pattern
Creational Design Pattern
Singleton Pattern
Factory Method Pattern
Abstract Factory Pattern
Builder Pattern
Prototype Pattern
Complete Guide to Design Patterns in Programming

Factory Method Pattern:

In the Factory method pattern , we make an abstract class, then we define the abstract methods , for example we would define the methods of open(),save() and close().

Now we make two classes pdfDocument and WordDocument that would extend the abstract class so it would override all the methods we declared in the document Abstract class.

Then in main we call them using the object and it performs the task.

Abstract Factory Method:

Its the same as factory method, but now we make an interface that have a method to create the document and then we use the same to make more implementations.

Singleton :

We use singleton so that we can check if the object is null then return new object and if the object is not null then return the object we already have.

```
public class Singleton {
    // Static variable to hold the single instance
    private static Singleton instance;
    // Private constructor to prevent instantiation
    private Singleton() {
        // Initialization code here
    }
// Public method to provide access to the instance
    public static Singleton getInstance() {
        // Lazy initialization
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
    // Example method
    public void showMessage() {
        System.out.println("Hello from Singleton!");
    }
}
// Client code
public class SingletonExample {
    public static void main(String[] args) {
        // Access the singleton instance
        Singleton singleton = Singleton.getInstance();
        singleton.showMessage();
    }
}
```

Builder Pattern

The **Builder Method** (often referred to as the **Builder Pattern**) is a design pattern used to construct complex objects step by step. It provides a way to create objects that require many parameters or when the creation process is complex. This pattern is particularly useful when the object being constructed has numerous optional parameters, or when you want to separate the construction and representation of the object.

```
// Product class
class Car {
    private String make;
    private String model;
    private int year;
    private String color;

    // Constructor is private to prevent direct instantiation
    private Car(CarBuilder builder) {
        this.make = builder.make;
        this.model = builder.model;
        this.year = builder.year;
        this.color = builder.color;
    }

    @Override
    public String toString() {
        return "Car [make=" + make + ", model=" + model + ", year=" + year + ", color=" + color + "]";
    }

    // Static inner Builder class
    public static class CarBuilder {
        private String make;
        private String model;
        private int year;
        private String color;

        // Method to set the make
        public CarBuilder setMake(String make) {
            this.make = make;
            return this;
        }

        // Method to set the model
        public CarBuilder setModel(String model) {
            this.model = model;
            return this;
        }

        // Method to set the year
        public CarBuilder setYear(int year) {
            this.year = year;
            return this;
        }

        // Method to set the color
        public CarBuilder setColor(String color) {
            this.color = color;
            return this;
        }

        // Method to construct the Car object
        public Car build() {
            return new Car(this);
        }
    }
}

// Client code
public class BuilderPatternExample {
    public static void main(String[] args) {
        // Construct a Car object using the Builder pattern
        Car myCar = new Car.CarBuilder()
                .setMake("Toyota")
                .setModel("Camry")
                .setYear(2023)
                .setColor("Red")
                .build();

        // Print the Car object
        System.out.println(myCar);
    }
}
```

The Builder Pattern provides a flexible and efficient way to construct complex objects. It enhances code readability and maintainability by separating the construction process from the representation, making it easier to create objects with many parameters and configurations. This pattern is widely used in various applications, particularly when creating objects with optional or default settings.

TestNg

TestNg means Test Next Generation

get the Testng plugin, either thorugh dependencies or thoroush the plugin store.
now create a testng.xml that would have all the requred thing you need
create a class for testing, and supppose name it basistest in testng package
so now in the testngxml in the classs mention

```
<classes>
    <class name="testng.BasicTestNg"></class>
</classes>
```

and then run the tesng.xml(this is the test suite)

The basics testng code

```
package testng;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.Assert;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;

import java.time.Duration;

public class BasicTestNg {
    private static WebDriver driver;

    @BeforeMethod
    public void setup(){

System.setProperty("webdriver.chrome.driver","C:¥¥Users¥¥prabhatritesh_chaube¥¥Downloads¥¥chromedriver-
win64¥¥chromedriver.exe");
        driver = new ChromeDriver();
        driver.manage().window().maximize();
        driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(10));
        driver.get("https://www.amazon.in/");
    }

    @Test
    public void checktitle(){
        String actual = driver.getTitle();
        String expected = ”Online Shopping site in India: Shop Online for Mobiles, Books, Watches, Shoes
and More - Amazon.in”;

        Assert.assertEquals(actual,expected);

    }

    @AfterMethod
    public void teardown(){
        driver.quit();
    }
}
```

TestNg Paramentres Annotations

this is used to send data from xml □ to testcases

```
<parameter name=”url” value=”www.example.com”></parameter>
```

use @Parameters({"url"}) before the @Test annotation and
when using pass a value for example

```
public void checkpackage(String myurl){
system.out.println(myurl)
//this would give output as www.example.com
```

@dataProvider annotaions

used to send java file □ Test cases

by using the **@dataprovider** annotation we can use a function to porvide a data to the test case and to the Test we would be providing it would become
@Test() □ @Test(dataProvider="function the gives the data")

```
@Test(dataProvider=”function the gives the data”)
public void takeinput(data types we are passing)
```

TesngListners

they are used to help us do what we do when the test fails

- make a class in which we want to do the test,
- now implement the ITestListner interface to the class
- override the methods we want to use, right click the screen and implemt / override it
- now in the testng.xml we would add

```
<listeners>
    <listener class-name=”testng.Testlistners”></listener>
</listeners>
```

between start of suite and test name

Soft Assertions

When we use the assertion , if it is failing then the u[coming functions does not work, but when using soft assertion

```
SoftAssert a = new SoftAssert();
Assert.assertTrue(false) ☒ a.asserTrue(false)
```

a.assertAll() would store the output

parallel testng

when we run the testng.xml is runs in serial manner and now to do for parallel

```
<suite name=”Suite” parallel=’tests’ thread-count=”2”>
```

this is for tests now if we want to run the classes parallelly then

```
<test name=”test1” parallel=”classes” thread-count = ”2”>
```

taking screenshot in testng

```
File f = ((TakeScreenShot)driver).getScreenshotas(OutputType.FILE);
Files.copy(f,new File(”location of file with name”))
```

`@BeforeMethod`	`@BeforeTest`
Runs before each test method in the class.	Runs before all test methods in a `<test>` block in `testng.xml`.
Before **every individual test method**.	Before the **first test method** in a `<test>` block.
Used for setting up preconditions for each test method (e.g., initializing data for each test).	Used for setting up configurations for all test methods in a `<test>` tag (e.g., global setup or initializing a browser).
Runs **once per test method**.	Runs **once per** `<test>` **block** in `testng.xml`.
Resetting state before each test method execution.	Setting up a database connection for all tests in a test group.
Applies to test methods in a **single class**.	Applies to all methods within the `<test>` defined in `testng.xml`.

@BeforeMethod

@AfterMethod

@Test

@BeforeSuite

@AfterSuite

@BeforeTest

@Afterest

@Parameters

@dataprovider

Assert.asserequalt(Actual,Expected)

Mock Tests

1. API Testing (Postman, RestAssured)

Theoretical Questions:

1. Explain the difference between `GET`, `POST`, `PUT`, and `DELETE` HTTP methods.
2. How does RestAssured handle authentication for API requests?

```
1. Basic Authentication
Basic Authentication is a simple authentication scheme built into the HTTP protocol. It uses a static
username and password combination.
2. OAuth 2.0 Authentication
RestAssured can handle OAuth 2.0 authentication, which is often used for APIs that require more secure
access.

RestAssured.given()
             .auth()
             .oauth2(accessToken) // Use OAuth2 token
              .when()
               .get("https://example.com/api/resource")
                .then()
                .statusCode(200);
```

1. What is the purpose of query parameters in a `GET` request? How are they different from path parameters?

Practical Questions:

1. Use Postman to test the following scenario:
   - Send a `GET` request with a query parameter to retrieve students whose age is greater than 18. Verify the response status code and body contains a list of students.
2. Write a RestAssured test to validate the following API:
   - Send a `POST` request to create a new student resource with JSON body. Validate that the response status code is `201 Created`.

---

2. Selenium WebDriver

Theoretical Questions:

1. What is Selenium WebDriver and how does it differ from Selenium RC?
2. Explain how to handle synchronization issues in Selenium.
3. What is the difference between `findElement()` and `findElements()` methods in Selenium?

Practical Questions:

1. Write a simple Selenium WebDriver test to automate the following scenario:
   - Navigate to a search engine, search for "Selenium WebDriver," and print the titles of the first five search results.
2. How would you handle a dynamic dropdown list using Selenium WebDriver?

---

3. Synchronization in Selenium

Theoretical Questions:

1. Why is using `Thread.sleep()` not recommended in Selenium tests for synchronization?
2. Compare and contrast implicit, explicit, and fluent waits in Selenium. When would you use each?

Practical Questions:

1. Implement an explicit wait in Selenium for an element that becomes visible after 10 seconds on the page.
2. Write a Selenium code to handle a synchronization issue where the page content loads at different times (use fluent wait).

---

4. Design Patterns

Theoretical Questions:

1. What is the Singleton pattern? How is it implemented in Java?
2. Describe the Page Object Model (POM) and explain why it's considered an antipattern in some cases.
3. Explain the Factory Method pattern and its use cases.

Practical Questions:

1. Implement the Builder pattern to create a `User` object with fields like `name`, `age`, `email`, and `address`.
2. Write a basic example of a Factory Method pattern to create different types of bank accounts (e.g., `SavingsAccount`, `CurrentAccount`).

---

5. TestNG Framework

Theoretical Questions:

1. What is the purpose of the `@DataProvider` annotation in TestNG? How is it used?
2. How does TestNG allow parameterization of test cases? Provide an example.

Practical Questions:

1. Write a TestNG test that verifies the login functionality for a web application. Use the `@DataProvider` to pass multiple sets of credentials.
2. Implement a TestNG test to run in parallel on multiple browsers (cross-browser testing).

---

6. Handling Locators and Page Object Model in Selenium

Theoretical Questions:

1. Explain the different types of locators in Selenium. When would you prefer using `CSS` selectors over `Xpath`?
2. What are the advantages of using the Page Object Model (POM) in Selenium?

Practical Questions:

1. Write Selenium code to locate a button by `Xpath` and click on it.
2. Implement a basic Page Object Model for a login page that includes methods to enter the username, password, and click the login button.

Practical Coding Challenges

1. **API Testing with RestAssured**: Write a RestAssured test to send a `DELETE` request to an API endpoint that deletes a student record. Validate the response status code is `200 OK`, and that the student no longer exists by sending a `GET` request.
2. **Synchronization in Selenium**: Write a Selenium test to handle a scenario where a webpage has a loading spinner that disappears when the content is ready. Use appropriate synchronization techniques to wait until the spinner disappears.
3. **TestNG – Parallel Execution**: Implement a TestNG test suite that runs two tests in parallel. One test should execute on Chrome and the other on Firefox.

Mock Test for Interview Preparation (Version 2)

---

1. API Testing (Postman, RestAssured)

Theoretical Questions:

1. What is the purpose of HTTP headers in API requests? Give examples of common headers.
2. Explain how you would test the performance of an API.
3. What are status codes in HTTP, and what do they indicate? Describe the significance of codes like `200`, `400`, `404`, and `500`.

Practical Questions:

1. In Postman, create a request that tests an API endpoint that returns a list of products. Ensure to validate the response time is within acceptable limits (e.g., <200 ms).
2. Write a RestAssured test that checks whether a specific product exists by sending a `GET` request to the endpoint and asserting that the response body contains the product name.

---

2. Selenium WebDriver

Theoretical Questions:

1. What is the difference between an absolute and a relative XPath?
2. Describe the concept of "headless" testing in Selenium and its advantages.
3. How do you handle alerts and pop-ups in Selenium?

Practical Questions:

1. Write a Selenium WebDriver script to automate the following:
    - Navigate to a login page, fill in the username and password, submit the form, and verify that the user is redirected to the homepage.
2. Write a Selenium code snippet to scroll to the bottom of a page and click a button that appears after scrolling.

---

3. Synchronization in Selenium

Theoretical Questions:

1. Explain how implicit waits and explicit waits differ. When should you use one over the other?
2. What are some common causes of synchronization issues in Selenium tests?

Practical Questions:

1. Implement a test using an implicit wait to wait for an element to appear before interacting with it. Demonstrate the difference between using implicit and explicit waits in two different methods.
2. Write Selenium code that uses a fluent wait to check for an element's visibility on a page, retrying for a maximum of 30 seconds.

---

4. Design Patterns

Theoretical Questions:

1. What is the Observer pattern? Provide a real-world example of its application.
2. Discuss the Strategy pattern and how it can be used to improve flexibility in your code.
3. Explain the Adapter pattern and when you would use it.

Practical Questions:

1. Implement a simple Observer pattern in Java where you have a `NewsPublisher` that notifies subscribers about news updates.
2. Create an example of the Strategy pattern where you have different algorithms for sorting an array of numbers.

---

5. TestNG Framework

Theoretical Questions:

1. How do you configure a TestNG suite? What XML elements are used in the configuration file?
2. Explain the use of `@BeforeClass` and `@AfterClass` annotations in TestNG.

Practical Questions:

1. Write a TestNG test that uses the `@BeforeMethod` annotation to set up the testing environment before each test case.
2. Implement a TestNG test with multiple `@Test` annotations that demonstrate the dependency between tests.

---

6. Handling Locators and Page Object Model in Selenium

Theoretical Questions:

1. What is the importance of using locators wisely in Selenium? What could happen if you use incorrect locators?
2. Describe how the Page Factory design pattern works in Selenium.

Practical Questions:

1. Write a Selenium code snippet to demonstrate how to find a checkbox element using a CSS selector and check it if it is not already checked.
2. Implement a Page Object Model for a registration page, including methods for entering user details and submitting the form.

Practical Coding Challenges

1. **API Testing with Postman**: Create a Postman collection that tests various endpoints of an API (e.g., `GET`, `POST`, `PUT`, `DELETE`). Include pre-request scripts to handle authentication tokens.
2. **Selenium Test with Waits**: Write a Selenium test that navigates to an e-commerce website, adds an item to the cart, and validates that the cart's item count has increased, using appropriate synchronization techniques.

3. **TestNG – Parameterization**: Create a TestNG test class that uses the `@DataProvider` annotation to run tests with different input data for a method that checks if a given string is a palindrome.

Mock Test for Interview Preparation (Version 3)

---

1. API Testing (Postman, RestAssured)

Theoretical Questions:

1. What are the key differences between REST and SOAP APIs?

Protocol	Primarily uses HTTP/HTTPS	Can use multiple protocols (HTTP, SMTP, etc.)

Data Format	Typically uses JSON or XML	Strictly uses XML

1. How would you validate the schema of a JSON response from an API?
2. Describe the process of authentication in APIs. What are some common methods?

Practical Questions:

1. Using Postman, create a test that checks if the response body of a `GET` request contains a specific field (e.g., `name` in a user object).
2. Write a RestAssured test case that validates the response time of an API endpoint is less than 300 milliseconds.

---

2. Selenium WebDriver

Theoretical Questions:

1. What is the role of the WebDriver in Selenium, and how does it differ from Selenium RC?
2. Explain how to handle file uploads in Selenium WebDriver.
3. What are the limitations of Selenium?

Practical Questions:

1. Write a Selenium script that navigates to a page, searches for a term, and verifies that the search results contain that term.

2. Create a Selenium code snippet that takes a screenshot of a webpage and saves it to a specified location.

---

3. Synchronization in Selenium

Theoretical Questions:

1. What are the potential issues of not using waits in Selenium tests?
2. Explain how you can handle dynamic web elements that load at different times.

Practical Questions:

1. Implement a Selenium test using an explicit wait to wait for an element (e.g., a button) to become clickable before proceeding.
2. Write a code example demonstrating the use of a `WebDriverWait` to wait for an element to be visible and then interact with it.

---

4. Design Patterns

Theoretical Questions:

1. Describe the Singleton pattern. What are its advantages and potential drawbacks?
2. What is the Factory pattern, and how does it differ from the Abstract Factory pattern?
3. Explain the Decorator pattern and give a use case where it would be beneficial.

Practical Questions:

1. Implement a Singleton class in Java that controls access to a resource, ensuring only one instance can be created.
2. Create a simple example of the Factory pattern to instantiate different types of vehicles (e.g., Car, Bike) based on a parameter.

---

5. TestNG Framework

Theoretical Questions:

1. What is the purpose of `@DependsOnMethods` in TestNG?
2. Explain the difference between `@Test(priority = 1)` and `@Test(dependsOnMethods = ”methodName”)`.

Practical Questions:

1. Write a TestNG test class with methods that demonstrate the use of `@AfterSuite` and `@BeforeSuite` annotations for setting up and tearing down resources.
2. Create a TestNG test that reads data from an external Excel file and uses it as input for the tests.

---

6. Handling Locators and Page Object Model in Selenium

Theoretical Questions:

1. What are some best practices for writing locators in Selenium?
2. How does the Page Object Model improve test maintainability?

Practical Questions:

1. Write a Selenium test that utilizes the Page Object Model for a login page, encapsulating the logic for entering credentials and clicking the login button.
2. Create a Page Object class for a search results page that includes methods to retrieve the number of results and click on a specific result.

Practical Coding Challenges

1. **API Testing with Postman**: Develop a comprehensive Postman collection to test a fictional online bookstore API, covering endpoints for books, authors, and categories. Include tests for status codes, response times, and data validation.
2. **Selenium Test with Complex Actions**: Write a Selenium test that performs the following actions:
   - Navigate to a webpage.
   - Fill out a form with various input fields, including dropdowns and checkboxes.
   - Submit the form and validate the success message.
3. **TestNG – Grouping Tests**: Create a TestNG test class that groups tests using the `@Test(groups = {”smoke”})` annotation. Ensure you run the grouped tests together and provide a summary of their execution.

Bonus Coding Challenges

1. **Multi-threading**: Implement a simple multi-threaded program in Java that prints numbers from 1 to 10, where each number is printed by a different thread.
2. **Data Structure Implementation**: Write a Java class that implements a basic stack data structure, including methods for push, pop, and checking if the stack is empty.