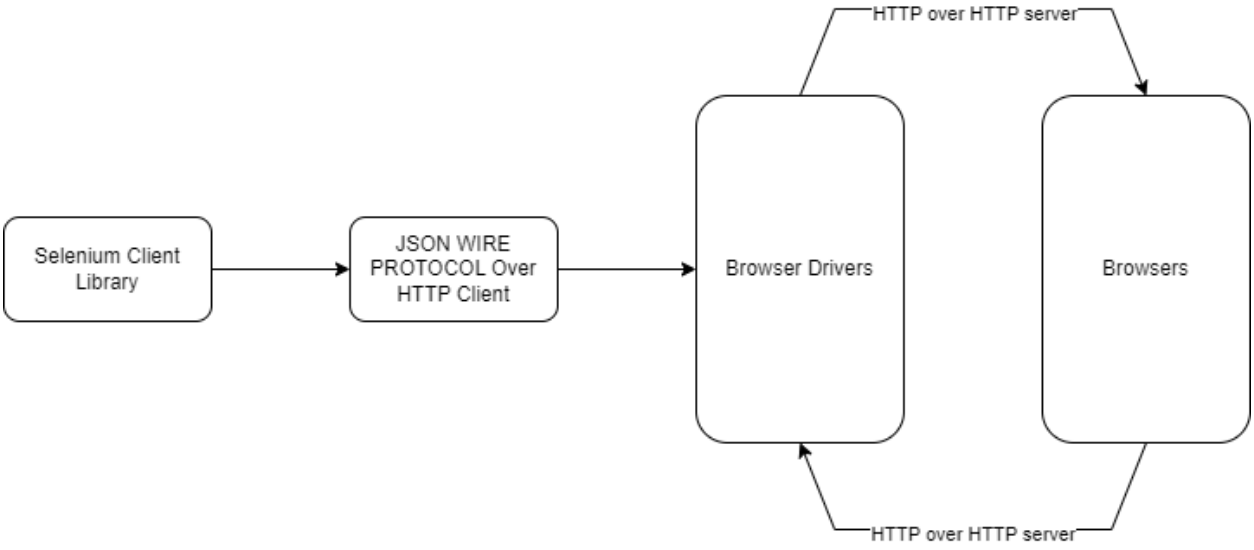

The architecture of Selenium WebDriver consists of the following:

- Selenium Client Library
- JSON WIRE PROTOCOL Over HTTP Client
- Browser Drivers
- Browsers

1- Selenium Client Library

The Selenium Client Library consists of various language libraries for Java, Ruby, Python, and other supported languages.

2- JSON WIRE PROTOCOL Over HTTP Client

JSON denotes Javascript Object Notation. This component of the Selenium WebDriver plays an important role in the Selenium automation process by transferring data between the server and a client on the web.

3- Browser Drivers

Browser drivers are used to carry out the communication between the Selenium WebDriver and the respective browser. The Browser drivers ensure that no details are revealed to the browser regarding the internal logic of the functionalities of the browser.

4- Browsers

As already discussed above, the browsers supported are Firefox, Safari, Chrome, and more.

Step 1: Set up the TestNG framework:

Create a new Java project in your preferred integrated development environment (IDE) like Eclipse or IntelliJ.

Add the necessary dependencies for Selenium WebDriver and TestNG in your project's build configuration or Maven/Gradle dependencies.

Step 2: Write TestNG test cases:

Create a new Java class, for example, "LoginTest," and annotate it with @Test from the TestNG framework.

Implement the necessary test methods within the class, representing different test scenarios related to the login functionality.

```
import org.openqa.selenium.WebDriver;
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;

public class LoginTest {
    private WebDriver driver;

    @BeforeMethod
    public void setUp() {
        // Set up WebDriver
        System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");
        driver = new ChromeDriver();
        driver.manage().window().maximize();
        // Other setup steps if required
    }

    @Test
    public void successfulLoginTest() {
        // Test steps for successful login
        driver.get("https://example.com/login");
        // Perform login actions
        // Assert login success
        Assert.assertEquals(driver.getTitle(), "Dashboard - My App");
```

```
    }

    @Test
    public void invalidLoginTest() {
        // Test steps for invalid login
        driver.get("https://example.com/login");
        // Perform invalid login actions
        // Assert error message
        Assert.assertTrue(driver.findElement(By.id("error-message")).isDisplayed());
    }

    @AfterMethod
    public void tearDown() {
        // Clean up WebDriver instance
        driver.quit();
    }
}
```

In this example, we have two test methods: successfulLoginTest() and invalidLoginTest(). The @Test annotation marks these methods as test cases.

The @BeforeMethod annotation is used to set up the WebDriver instance before each test method execution, ensuring a fresh state for each test.

The @AfterMethod annotation is used to clean up and quit the WebDriver after each test method, preventing resource leaks.

Step 3: Run the TestNG tests:

Right-click on the test class or the test suite file (XML file) containing the TestNG configuration.

Select the "Run as TestNG Test" option.

TestNG will execute the test methods in the specified order and generate the test reports. You can view the test results, including the status (pass/fail) and any assertions or exceptions thrown, in the TestNG report generated by the framework.

By utilizing TestNG, you can organize and manage your Selenium test cases effectively, perform assertions, set up preconditions, and generate detailed reports, enhancing the testing process for your web application.