infogain

Selenium WebDriver

Session Objective

infogain
www.infogain.com

- Automation Testing Fundamental Concepts
- Introducing WebDriver and WebElements
- Exploring Advanced Interactions of WebDriver
- Exploring the Features of WebDriver
- Different Available WebDrivers
- Understanding WebDriver Events
- Exploring RemoteWebDriver and WebDriverBackedSelenium
- Understanding Selenium Grid
- Understanding PageObject Pattern

Understanding WebDriver Events

- Selenium WebDriver provides a very good framework for tracking the various events that happen while you're executing your test scripts using WebDriver.

EventFiringWebDriver and EventListener classes:

- **EventFiringWebDrive**r class is a wrapper around your normal WebDriver that gives the driver the capability to fire events.
- The **EventListener** class, on the other hand, waits to listen from **EventFiringWebDriver** and handles all of the events that are dispatched.
- There can be **more than one** listener waiting to hear from the EventFiringWebDriver class for an event to fire.
- All of the event listeners should be **registered with the EventFiringWebDriver** class to get notified.

Working with Event Listener

Step 1: **Create EventListener**

There are the following two ways to create an EventListener class:

- By implementing the **WebDriverEventListener** interface.
- By extending the **AbstractWebDriverEventListener** class provided in the WebDriver library

Step 2: **Creating a WebDriver instance**

- WebDriver driver = new FirefoxDriver();

Step 3: Creating EventFiringWebDriver and EventListener instances

- EventFiringWebDriver eventFiringDriver = new EventFiringWebDriver(driver);
- MyEventListener eventListener = new MyEventListener();

Registering EventListener with EventFiringWebDriver

- eventFiringDriver.register(eventListener);

Registering multiple EventListeners

```
eventFiringDriver.register(eventListener);
eventFiringDriver.register(eventListener2);
```

Unregistering EventListener with EventFiringWebDriver

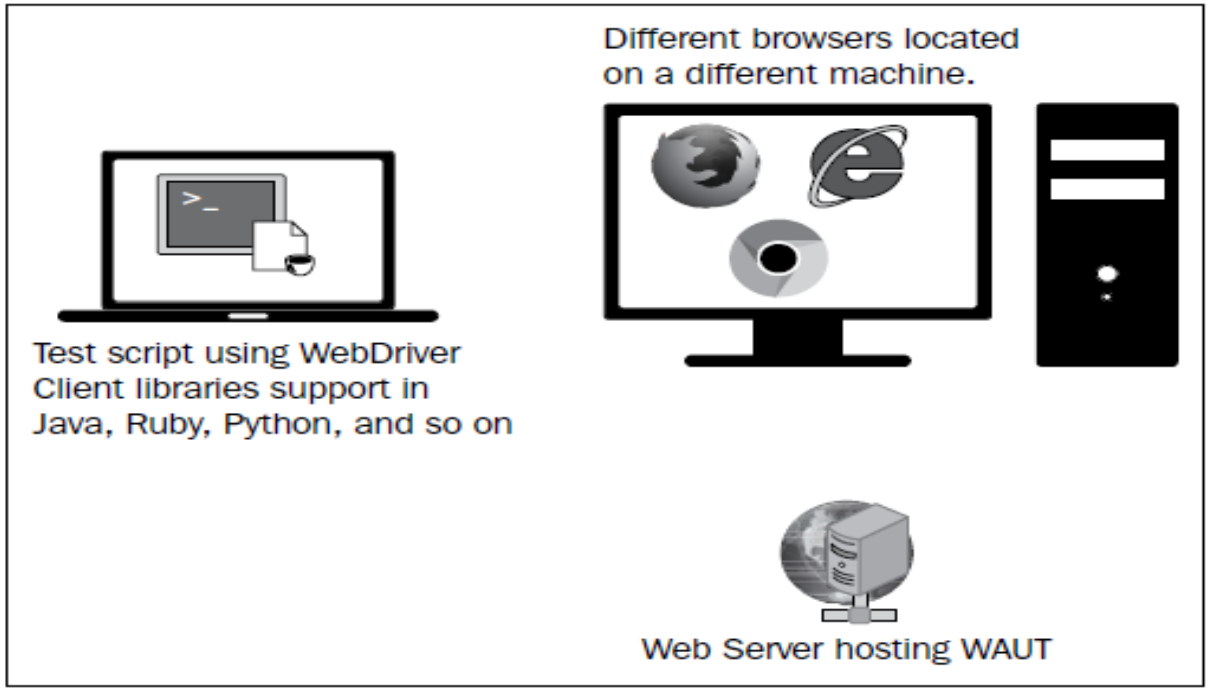
unregister(WebDriverEventListener eventListener)

Exploring RemoteWebDriver

- There is a high possibility that you may be working on Mac or Linux, but want to execute your tests on IE on a Windows machine.
- Executing test cases on a remote machine using RemoteWebDriver

RemoteWebDriver

- RemoteWebDriver is an implementation class of the WebDriver interface that a test script developer can use to execute their test scripts via the RemoteWebDriver server on a remote machine.
- There are two parts to RemoteWebDriver: **a server and a client.**



The test script is located on a local machine, while the browsers are installed on a remote machine.

RemoteWebDriver server

- The RemoteWebDriver server is a component that listens on a port for various requests from a RemoteWebDriver client.
- Once it receives the requests, it forwards them to any of the following: Firefox Driver, IE Driver, or Chrome Driver, whichever is asked.

Running the server :

java –jar selenium-server-standalone-2.45.0.jar

RemoteWebDriver client :

- It's nothing but the language-binding client libraries that serve as a RemoteWebDriver client.
- The client, as it used to when executing tests locally, translates your test script requests to JSON payload and sends them across to the RemoteWebDriver server using the JSON wire protocol.

Testing with Chrome Driver:

java -Dwebdriver.chrome.driver="C:/remote/chromedriver.exe" -jar selenium-server-standalone-2.45.0.jar

Understanding PageObject Pattern

infogain
www.infogain.com

- What is the PageObject pattern design?
- Good practices for designing PageObjects
- Extensions to the PageObject pattern
- An end-to-end example

PageObject pattern

- Whenever we are designing an automation framework for testing web applications, we have to accept the fact that the target application and its elements are bound to change.
- An efficient framework is one that needs minimal refactoring to adapt to new changes in the target application.
- Page Object Model is a design pattern to create Object Repository for web UI elements.
- Under this model, for each web page in the application there should be corresponding page class.
- This Page class will find the WebElements of that web page and also contains Page methods which perform operations on those WebElements.
- Name of these methods should be given as per the task they are performing.

@FindBy annotation

infogain
www.infogain.com

- An element in the PageObject is marked with the @FindBy annotation.
- It is used to direct the WebDriver to locate that element on a page. It takes the locating mechanism (that is, By Id or Name or Class Name) and the value of the element for that locating mechanism as input.
- There are two ways of using the **@FindBy** annotation:

- Usage 1 is shown as follows:

```
@FindBy(id="user_login")
WebElement userId;
```

- Usage 2 is shown as follows:

```
@FindBy(how=How.ID, using="user_login")
WebElement userId;
```

Understanding PageFactory

infogain
www.infogain.com

- Once the **PageObject** class declares elements using the FindBy annotation, you can instantiate that PageObject class and its elements using the **PageFactory** class.
- This class supports a static method named **initElements.**

The API syntax for this method is as follows:

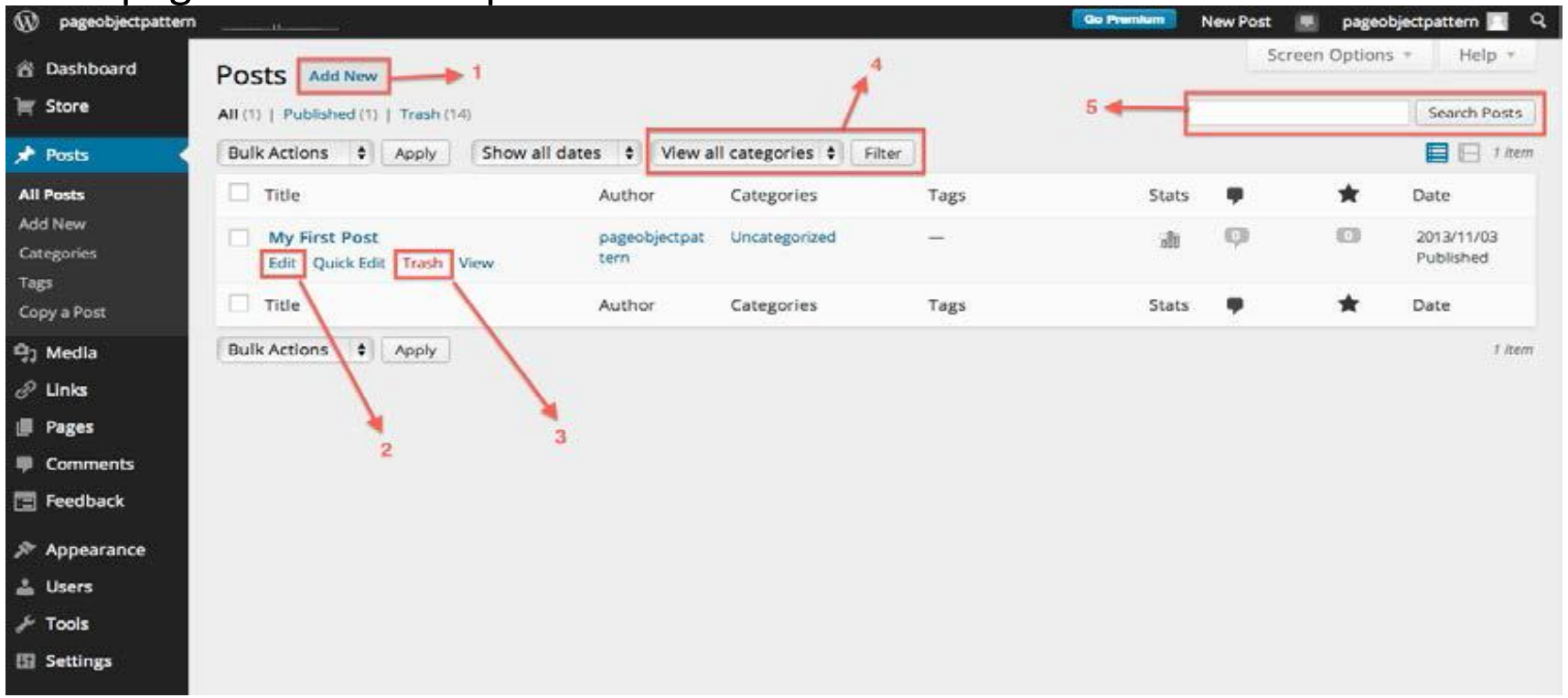
initElements(WebDriver driver, java.lang.Class PageObjectClass)

```
AdminLoginPage loginPage = PageFactory.initElements(driver,AdminLoginPage.class);
Thread.sleep(5000);
AllPostsPage allPostsPage = loginPage.login();
```

Good practices for the PageObjects design

infogain
www.infogain.com

- Consider a web page as a services provider



Always look for implied services

There are services that a page provides which can be identified very clearly on it.
There are some such services that are not visible on the page, but are implied. Like count number of posts.

Using PageObjects within a PageObject

- There will be many situations where you need to use **PageObject**s within a **PageObject.**
- Let us analyze that using a scenario on the **All Posts** page. When you click on **Add New** to add a new post, the browser actually navigates to a different page.
- So, you have to create two **PageObjects**, one for the **All Posts** page and another for the **Add New** page.

Methods in PageObjects as services and not as User Actions

- There might sometimes be confusion surrounding what methods make a PageObject.
- We have seen earlier that each PageObject should **contain User Services** as their methods. But quite often, we see some implementations of **PageObjects** in several test frameworks that constitute User Actions as their methods .

User Services :

- Create a new post
- Delete a post
- Edit a post

User Actions:

- Mouse click
- Typing text in a textbox
- Navigating to a page

Keeping the page-specific details off the test script

- The ultimate aim of the PageObject pattern design is to maintain the page-specific details, such as the IDs of the elements on the page, the way we reach a particular page in the application, and so on, away from the test script.
- Building your test framework using the PageObject pattern should allow you to keep your test scripts very generic and needing no modification each time the page implementation details change .

Understanding loadable components

infogain
www.infogain.com

- The loadable component is an extension to the PageObject pattern.
- The **LoadableComponen**t class in the WebDriver library will help test case developers make sure that the page or a component of the page is loaded successfully .
- The PageObject should extend this LoadableComponent abstract class and, as a result, it is bound to provide implementation for the following two methods:
  - **protected abstract void load()**
  - **protected abstract void isLoaded() throws java.lang.Error**

infogain

Thank You