Aan de slag met Selenium WebDriver

Roy de Kleijn

Who am I ?

- Technical Test Consultant
- Trainer test automation
- Involved in R&D activities
- Blogs:
  - http://selenium.polteq.com
  - http://www.rdekleijn.nl

Goal of the workshop

Get familiar with the Selenium WebDriver API and implementing a structured maintainable automated testing framework.

Note: All the examples are written in Java, but are applicable to any strongly-typed programming language.

Goal of the workshop

Get familiar with the Selenium WebDriver API and implementing a structured maintainable automated testing framework.

Learn by simultaneous programming

Note: All the examples are written in Java, but are applicable to any strongly-typed programming language.

Agenda

- Selenium introduction

In practice:

- Setting up a project
- Create your first automated test
- Interact with the browser
- Locate Web Elements
- Interact with Web Elements
- Introducing design patterns

Agenda

- **Selenium introduction**

In practice:

- Setting up a project
- Create your first automated test
- Interact with the browser
- Locate Web Elements
- Interact with Web Elements
- Introducing design patterns

Capabilities of Selenium

Selenium is a set of tools to automate test scripts which simulate user actions in different browsers on different platforms.

Selenium components

- Selenium IDE (Integrated Development Environment)

  Software that provides support for the creation of test scripts

- Selenium-standalone server

  Execute Selenium IDE test scripts in different browsers

- WebDriver

  Create your test scripts in a preferred programming language

- Selenium GRID

  Execute test scripts on different environment and execute your test scripts in parallel

Selenium WebDriver



- Cross platform
  Unix, windows, mac
- Cross browser
  Internet explorer, firefox, chrome, opera, safari, htmlunit
- Cross language
  Java, .NET, ruby, python
- Mobile browser testing
  Android, iphone
- Object Oriented API
- Parallel testing







Selenium WebDriver

- Versatile

  feature, end-to-end, integration, acceptance, regression testing

Selenium WebDriver

- Manupilate cookies
- Take screenshots
- Simulate drag and drop
- Handle pop-up dialogs
- Execute javascipt

Selenium Grid

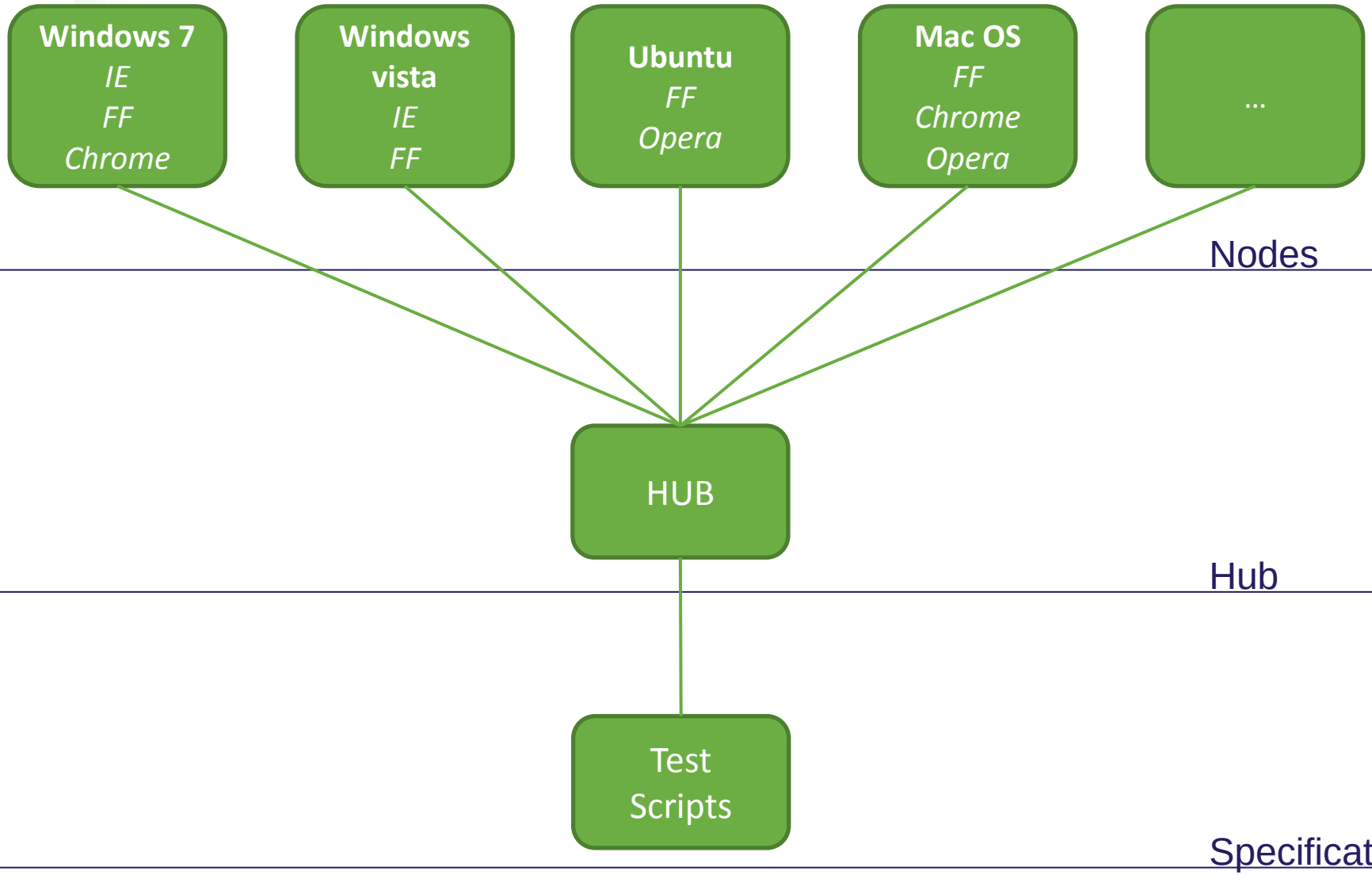
- Test execution on different environments
- Parallel testing
- Customized plugins

Selenium Grid

- Test execution on different environments
- Parallel testing
- Customized plugins

Reduce test execution time

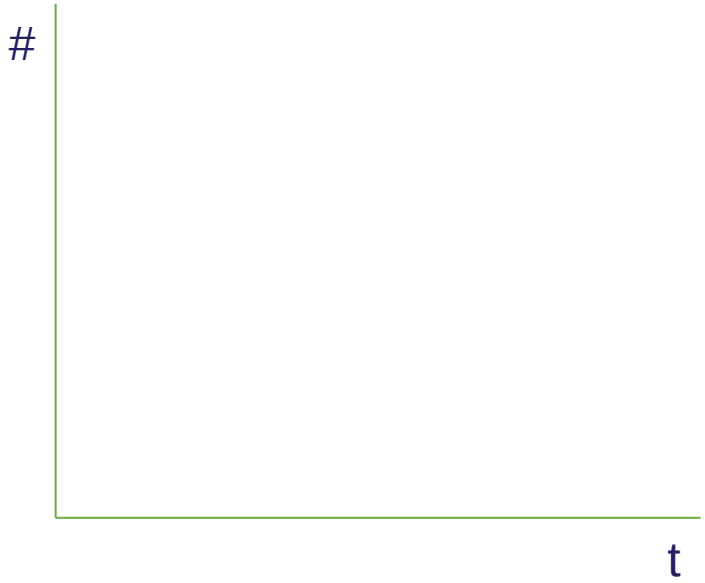
Selenium Grid



Node configuration

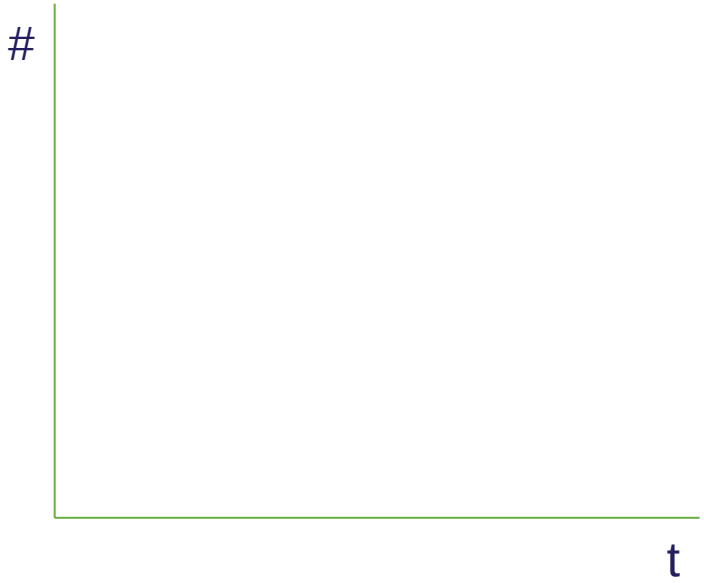
- Platfrom
  - Mac, Linux, android, vista, xp
- BrowserName
  - Firefox, chrome, internet explorer, safari, opera
- Version
  - Browser version

Selenium Grid



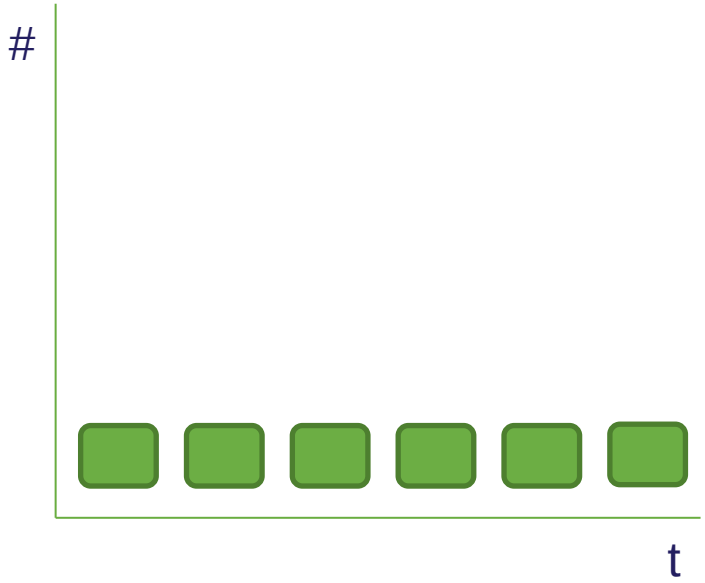
Without Selenium Grid

VS.



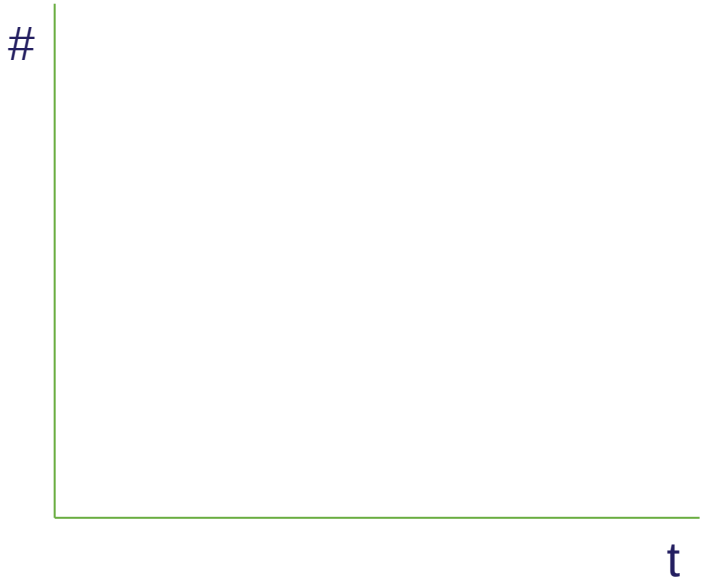
With Selenium Grid

Selenium Grid



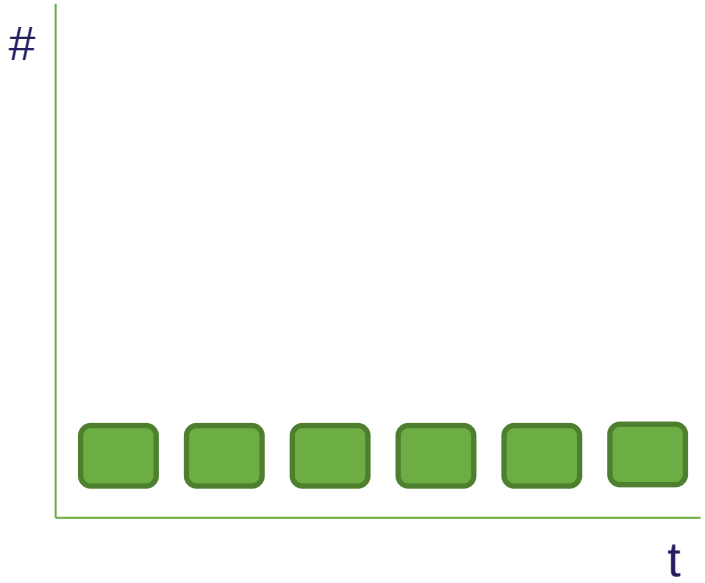
Without Selenium Grid

VS.



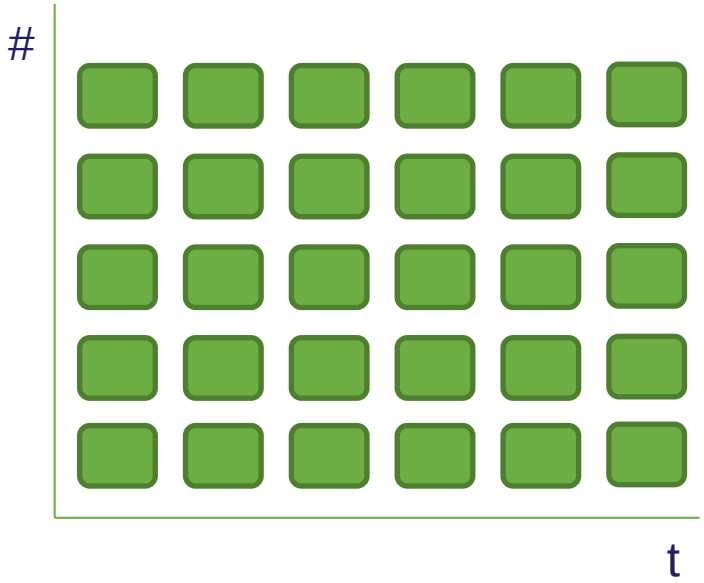
With Selenium Grid

Selenium Grid



Without Selenium Grid

vs.



With Selenium Grid

Customized plugins

- Prioritizer
  - Order the available test scripts in the HUB based on priority

Agenda

- Selenium introduction

In practice:

- **Setting up a project**
- Create your first automated test
- Interact with the browser
- Locate Web Elements
- Interact with Web Elements
- Introducing design patterns

Setting up a project

Prerequisites:

- Java JDK

(http://www.oracle.com/technetwork/java/javase/downloads/index.html)

- Eclipse

Download the appropriate xx-bit version

(http://eclipse.org/downloads/packages/eclipse-ide-java-developers/keplerr)

Start eclipse and create workspace

- Start eclipse
  - Create workspace

    Thick the box to always use this workspace

  - Go to workbench 

TestNG eclipse plugin

Install the plugin or update the TestNG plugin

- In eclipse: `Help` -> `Eclipse Marketplace`
  - Search for '`TestNG`'

Create a project

Maven is a software project management tool. We use it mainly for managing dependencies and running our tests.

- Open context menu of package explorer
  - Choose: `New` -> `Other...` and select `Maven` -> `Maven Project`
- Choose: Create a simple project, as we can skip the project template section

Create a project

Enter a Group Id and Artifact Id as they are used to give your project an unique identification.

Group Id will identify your project uniquely across all projects
Artifact Id is the name of the generated jar file

Example:
Group Id: `org.workshop.webdriver`
Artifact Id: `testshop`

Create a project

A default project structure is generated with all the source folders and a pom (Project Object Model) file.

The pom file contains project configurations / dependencies and plugins that can be executed.

Add dependencies

We can start adding two dependencies. The testing framework (TestNG) we use and Selenium WebDriver to interact with the browser.

- Open the context menu of the pom file and choose: `Maven -> Add Dependency`
- Now you can search for '`selenium-java`' and select the latest version

- Add another dependency, called '`testng`' and select the latest version

Now both dependencies are part of our project.

Add plugin

We need a maven plugin to execute our tests, called Surefire

- Open the context menu of the pom file and choose: `Maven -> Add Plugin`
- Now you can search for '`maven-surefire-plugin`' and select the latest version

Now you are able to execute your tests by maven

There is an example pom-file on the USB stick which you can use to copy its contents.

In practice

Let’s see how it works…

Test website: http://selenium.polteq.com/testshop/

Agenda

- Selenium introduction

In practice:

- Setting up a project
- **Create your first automated test**
- Interact with the browser
- Locate Web Elements
- Interact with Web Elements
- Introducing design patterns

Create a test class

Make sure your class (containing the tests) is named like *Test.java, then it will be automatically run by Surefire

Open the context menu of the source folder `src/test/java` and choose `New` -> `Class`

Specify a package and a class name

Example:

Package name: `org.workshop.test`

Class name: `FirstTest`

Create a test method

Annotate a method with @Test or you can do it once on class level

```
@Test
public void openSite() {
    // Create a webdriver instance to control the browser
    WebDriver driver = new FirefoxDriver();
    // Open a website
    driver.get("http://selenium.polteq.com/testshop/");
    // Assert the browsers title
    Assert.assertEquals(driver.getTitle(), "TestShop");
    // Quit the browser
    driver.quit();
}
```

Execute your test script

There are two ways of executing your test script

- Open the context menu of the just created test method and select `Run As -> TestNg Test`
- Surefire will automatically execute all the tests in *Test.java classes if we run '`test`' goal

Take screenshot on failure

```
 if (!result.isSuccess()) {
      File scrFile = ((TakesScreenshot) driver)
          .getScreenshotAs(OutputType.FILE);
      try {
        String fileName = result.getName() + UUID.randomUUID();
        File targetFile = new File("target/screenshots/" +
fileName + ".jpg");
        FileUtils.copyFile(scrFile, targetFile);
        result.setAttribute("screenshot", "<a target='blank'
href='../screenshots/" + fileName + ".jpg'>Screenshot</a>");
        Reporter.setCurrentTestResult(result);
        Reporter.log("<a target='blank' href='" +
targetFile.getAbsolutePath() + "'> Screenshot</a>");
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
```

Analyze the results

TestNG test results location:

`Project\test-output\index.html`

Maven test results location:

`Project\target\surefire-reports\index.html`

Introduce pre and post methods

- TestNG is a very flexible testing framework which comes with a lot before and after annotations.

- In previous test script you can see that we open and close a browser within the test script.

Open Browser

```
// Create a webdriver instance to control the browser
WebDriver driver = new FirefoxDriver();
```

Close Browser

```
// Quit the browser
driver.quit();
```

Introduce pre and post methods

```
public class BeforeAfterTest {
  private WebDriver driver;
  @BeforeMethod
  public void setUp() {
    driver = new FirefoxDriver();
    driver.get("http://selenium.polteq.com/testshop/");
  }
  @AfterMethod
  public void tearDown() {
    driver.quit();
  }
  @Test
  public void openSite() {
    Assert.assertEquals(driver.getTitle(), "TestShop");
  }
}
```

Introduce pre and post methods

A field which we can use over all the methods in this class

```
public class BeforeAfterTest {
  private WebDriver driver;
  @BeforeMethod
  public void setUp() {
    driver = new FirefoxDriver();
    driver.get("http://selenium.polteq.com/testshop/");
  }
  @AfterMethod
  public void tearDown() {
    driver.quit();
  }
  @Test
  public void openSite() {
    Assert.assertEquals(driver.getTitle(), "TestShop");
  }
}
```

Introduce pre and post methods

```
public class BeforeAfterTest {
  private WebDriver driver;
  @BeforeMethod
  public void setUp() {
    driver = new FirefoxDriver();
    driver.get("http://selenium.polteq.com/testshop/");
  }
  @AfterMethod
  public void tearDown() {
    driver.quit();
  }
  @Test
  public void openSite() {
    Assert.assertEquals(driver.getTitle(), "TestShop");
  }
}
```

A field which we can use over all the methods in this class

@BeforeMethod runs before every test methods

Introduce pre and post methods

```
public class BeforeAfterTest {
  private WebDriver driver;
  @BeforeMethod
  public void setUp() {
    driver = new FirefoxDriver();
    driver.get("http://selenium.polteq.com/testshop/");
  }
  @AfterMethod
  public void tearDown() {
    driver.quit();
  }
  @Test
  public void openSite() {
    Assert.assertEquals(driver.getTitle(), "TestShop");
  }
}
```

A field which we can use over all the methods in this class

@BeforeMethod runs before every test methods

@AfterMethod runs after every test methods

Introduce pre and post methods

@BeforeSuite

@AfterSuite

Introduce pre and post methods

@BeforeSuite

    @BeforeClass

    @AfterClass

@AfterSuite

Introduce pre and post methods

@BeforeSuite
    @BeforeClass
        @BeforeMethod
            @Test
        @AfterMethod
        [...]
        @BeforeMethod
            @Test
        @AfterMethod
    @AfterClass
@AfterSuite

Execute by group name

Add `groups` attribute to the `@Test` annotation

Example:

```
@Test(groups = {"initialtest"})
```

- For group execution we need to add `alwaysRun= true` to `@Before*` and `@After*` annotations. Then this configuration method will be run regardless of what groups it belongs to.

Execute by group name

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.14</version>
      <configuration>
            <groups>initialtest</groups>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Execute your test with maven

Assertions

Compare result with expectation

```
Assert.assertEquals(actual, expected, message);

Assert.assertTrue(condition, message);

Assert.assertFalse(condition, message);
```

Hamcrest provides a library of flexible matcher objects.
http://code.google.com/p/hamcrest/

Useful eclipse shortcuts

Shortcut	Explanation
CTRL + 1	Shows quick fixes for common issues, like missing semi colons, import issues, missing declarations.
CTRL + Shift + O	Fix missing imports
CTRL + Shift + F	Auto formatting
Ctrl + Space	Content Assist
CTRL + 7	Comment line

In practice

Let’s see how it works…

Test website: http://selenium.polteq.com/testshop/

Agenda

- Selenium introduction

In practice:

- Setting up a project
- Create your first automated test
- **Interact with the browser**
- Locate Web Elements
- Interact with Web Elements
- Introducing design patterns

WebDriver interface

Launch different browsers

```
driver.get()
      .getCurrentUrl()
      .getPageSource()
      .getTitle()
driver.manage().window().setSize()
      .manage().window().setPosition()
driver.navigate().back()
      .navigate().forward()
      .navigate().to()
      .navigate().refresh()
```

And more ...

Reference: http://selenium.googlecode.com/git/docs/api/java/index.html

Launch Firefox

FirefoxDriver is part of WebDriver

```
@Test
public void launchFirefox() {
  // This will open firefox
  WebDriver driver = new FirefoxDriver();

  // Open a webpage
  driver.get("http://selenium.polteq.com/testshop/");

  // Close the browser
  driver.close();
}
```

Launch Chrome

- ChromeDriver is a separate executable
- Maintained by the Chromium team
- Download url executable: http://code.google.com/p/chromedriver/downloads/list

Launch Chrome

```
@Test
public void launchChrome() {
  // We have to set a path property
  System.setProperty("webdriver.chrome.driver",
      "src/test/resources/chromedriver.exe");

  // This will open firefox
  WebDriver driver = new ChromeDriver();

  // Open a webpage
  driver.get("http://selenium.polteq.com/testshop/");

  // Close the browser
  driver.close();
}
```

Launch Internet Explorer

- InternetExplorerDriver is a separate executable
- Download url executable: http://code.google.com/p/selenium/downloads/list

Launch Internet Explorer

```
@Test
public void launchInternetExplorer() {
  // We have to set a path property
  System.setProperty("webdriver.ie.driver",
      "src/test/resources/IEDriverServer.exe");

  // This will open firefox
  WebDriver driver = new InternetExplorerDriver();

  // Open a webpage
  driver.get("http://selenium.polteq.com/testshop/");

  // Close the browser
  driver.close();
}
```

Navigation

```
Navigation nav = driver.navigate();
nav.to(url)
nav.back()
nav.forward()
```

Implicitly wait method

Polling the DOM for X seconds

```
driver.manage().timeouts().implicitlyWait(20, TimeUnit.SECONDS);
```

Execute javascript

```
JavascriptExecutor js = (JavascriptExecutor) driver;
js.executeScript("return document.title");
```

In practice

Let’s see how it works…

Test website: http://selenium.polteq.com/testshop/

Agenda

- Selenium introduction

In practice:

- Setting up a project
- Create your first automated test
- Interact with the browser
- **Locate Web Elements**
- Interact with Web Elements
- Introducing design patterns

Locators

Locators are the way to tell Selenium with which element we like to do something.

Tools

- Firefox
  - Firebug (http://getfirebug.com)
  - FirePath (FireFox add-on)
- Internet explorer
  - F12
- Chrome
  - F12
- Opera
  - http://dev.opera.com/articles/view/opera-developer-tools/
- Safari
  - Preferences -> advanced

- All browsers
  - Firebug light (http://getfirebug.com/firebuglite)

Firefox Add-ons

Firebug & Firepath

- Display sourcecode
- Get to know element names

Inspect elements on the page

Locate a textfield

Example:

```
<input type="text" class="first" name="first"
   id="firstTextfield">
```


```
driver.findElement(By.cssSelector("input#firstTextfield"))
```

Locate table cells

Example:

```
<table border="1" id="simpleTable">
    <tr>
        <td>1.1</td>
       <td>1.2</td>
    </tr>
    <tr>
        <td>2.1</td>
       <td>2.2</td>
    </tr>
</table>
```

1.1	1.2
2.1	2.2

```
driver.findElement(By
    .cssSelector("table#simpleTable tr:nth(2) td:nth(2)"))
```

Result: 2.2

Locate a list item

Example:

```
<ul id="list">
   <li>Selenium IDE</li>
   <li>WebDriver</li>
   <li>Selenium Grid</li>
</ul>
```

```
driver.findElement(By
        .cssSelector("ul#list li:nth(2)"))
```

Result: WebDriver

In practice

Let’s see how it works…

Test website: http://selenium.polteq.com/testshop/

Agenda

- Selenium introduction

In practice:

- Setting up a project
- Create your first automated test
- Interact with the browser
- Locate Web Elements
- **Interact with Web Elements**
- Introducing design patterns

Launch different browsers

driver.findElement(By.*id("element")).sendKeys()*

.click()

.getAttribute()

.getCssValue()

.submit()

.isDisplayed()

.isEnabled()

.isSelected()

And more ...

Reference: http://selenium.googlecode.com/git/docs/api/java/index.html

In practice

Let’s see how it works…

Test ideas: Login, register, search, contact

Test website: http://selenium.polteq.com/testshop/

Agenda

- Selenium introduction

In practice:

- Setting up a project
- Create your first automated test
- Interact with the browser
- Locate Web Elements
- Interact with Web Elements
- **Introducing design patterns**

Problems that arise

- Unmaintainable
- Unreadable test scripts
- Creation of test scripts is time consuming
- **Code duplication**

From problem to solution

```java
@Test
public void isItemInOrderListNONPOM() throws InterruptedException {
    driver.get("http://selenium.polteq.com/prestashop/authentication.php");
    driver.findElement(By.cssSelector("input#email")).sendKeys(
            "roy.dekleijn@polteq.com");
    driver.findElement(By.cssSelector("input#passwd")).sendKeys("1qazxsw2");
    driver.findElement(By.cssSelector("input#SubmitLogin")).click();
    driver.findElement(By.cssSelector("a[title='Orders']")).click();

    List<WebElement> els = driver.findElements(By
            .cssSelector("table#order-list tbody tr"));
    for (WebElement temp : els) {
        System.out.println(temp.findElement(
                By.cssSelector("td[class='history_link bold'] a"))
                .getText());
    }
}
```

```java
@Test
public void isItemInOrderList() {
    List<OrderHistoryResult> results = new AuthenticationPage(driver).get()
            .setAccountEmail("roy.dekleijn@polteq.com")
            .setAccountPassword("1qazxsw2").clickLoginButton()
            .navigateToMyOrdersPage().getResults();

    for (OrderHistoryResult temp : results) {
        System.out.println(temp.getTitle());
    }

}
```

Solution

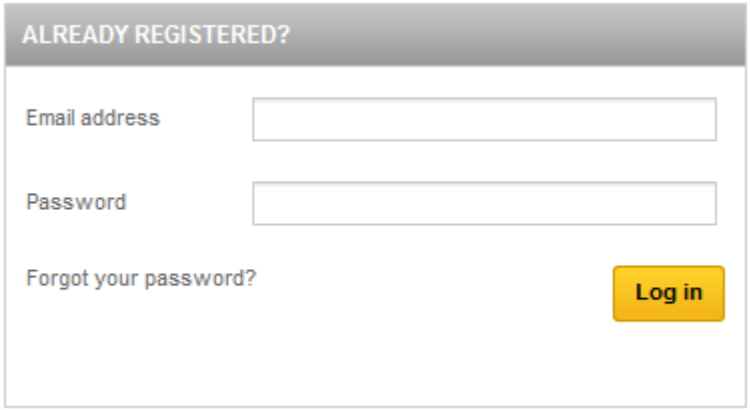
Each page contains only a part of the total functionality available on the website

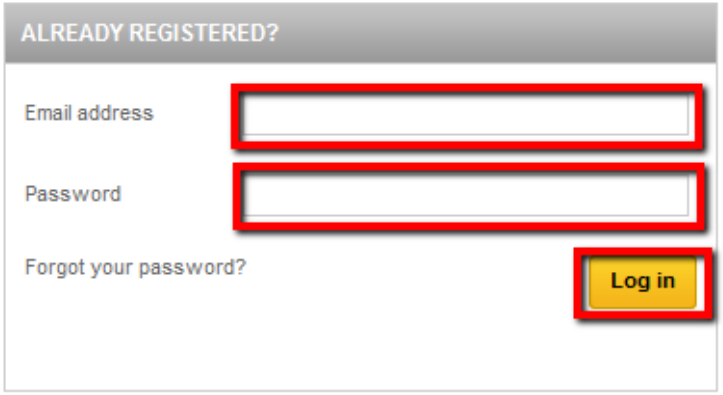
Put page specific functionality in a class with a corresponding name

Step-by-step plan

1. Identify necessary WebElements
2. Create a class
3. Define WebElements in corresponding classes
4. Model the functionality of a page into methods
5. Model the page flow by setting returntypes

Identify necessary WebElements





Create a class

A class with the name of the page extending from `LoadableComponent`

```
public class HomePage extends LoadableComponent<HomePage> {
  private WebDriver driver;

  public HomePage(WebDriver driver) {
    this.driver = driver;
    PageFactory.initElements(driver, this);
  }
```

Define WebElements

On class level (above the methods)

```
@FindBy(how = How.CSS, using = "a.login")
private WebElement loginLink;
```

Model the functionality

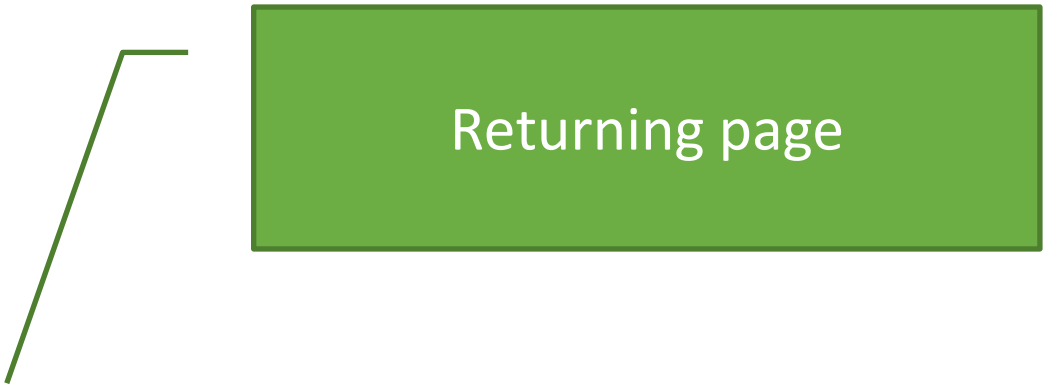
```
public void clickOnLoginLink() {
    loginLink.click();
    return new LoginPage(driver);
  }
```

Model the page flow

- Prerequisite:
  - Multiple pages are modelled

- Modify returntype
  - The returntype is the name of the page (class) where you are navigating towards
  - Use the current class name, if you stay on the same page

Model the page flow




```
public LoginPage clickOnLoginLink() {
   loginLink.click();
   return new LoginPage(driver);
}
```

In practice

Let’s see how it works…

Test website: http://selenium.polteq.com/testshop/

What we have learned today

- Basics of TestNG testing framework
- Basic usage of matchers
- Using the Selenium WebDriver API
- Create maintainable test scripts using design patterns

Reference

- Selenium Javadoc

http://selenium.googlecode.com/git/docs/api/java/index.html

- TestNG documentation

http://testng.org/doc/documentation-main.html

Thank you!

Source available on GitHub:
https://github.com/roydekleijn/workshop/

Get your FREE copy using coupon code: TESTNETNL

(coupon code valid for 1 months)

Book: https://leanpub.com/LearningSelenium/

Twitter: https://twitter.com/TheWebTester

LinkedIn: http://www.linkedin.com/in/roydekleijn

Website: http://rdekleijn.nl