

Selenium Webdriver

kreisfahrer

Published
with GitBook



Содержание

Introduction	0
Selenium Webdriver. Введение	1
WebDriver. Обзор и принцип работы	1.1
Основные методы Selenium Webdriver API	1.2
Типы локаторов	1.3
Ожидания	1.4
Пример использования Webdriver API	1.5
Selenium WebDriver. Сложные вопросы.	2
Локаторы. CSS, XPATH, JQUERY.	2.1
WebDriver API. Сложные взаимодействия.	2.2
Контроль за ходом теста. Кастомные ожидания, попапы, алерты, iframes.	2.3
DDT подход	2.4
Page Object Pattern. Архитектура тестового проекта.	3
Использование паттерна Page Object.	3.1
Альтернативные Page Object подходы.	3.2
Вспомогательные инструментаы.	3.3
Уровни абстракции. Создание кастомных элементов.	3.4
Архитектура. Основные элементы.	3.5
Selenium Grid и "headless" браузеры	4
Использование HtmlUnit драйвера в автотестировании	4.1
"Headless" тестирование с PhantomJS и SlimerJS	4.2
Grid. Настройка и использование.	4.3
Selenium Webdriver. Проблемные моменты	5
Вспомогательные интсрументы	5.1
Basic Authentification Window	5.2
Загрузка файла	5.3
Отправление файла (upload)	5.4
Логгирование в Selenium Webdriver	5.5
Скриншоты элементов и работа с изображением	5.6
Selenium Webdriver. Тестирование HTML5 веб приложений	6

Автоматизация Canvas элементов.	6.1
Автоматизация видео плеера.	6.2
Работа с web storage.	6.3
Selenium Webdriver. Расширение инструмента	7
Selenium "обертки" и расширения	7.1
Thucydides	7.2
Geb	7.3
Selenide	7.4
Репортинг	7.5
Selenium Webdriver. Тестирование клиентской производительности	8
Navigation timing API	8.1
Browser Mob Proxy	8.2
DynaTrace	8.3
HttpWatch	8.4
Selenium Webdriver. Тестирование на мобильных браузерах	9
Обзор инструментов	9.1
Установка и настройка Appium. Принципы и основы работы с инструментом	
Запуск тестов на десктоп и мобильных браузерах	9.3 9.2
Selenium Webdriver. Behavior-Driven Development.	10
Обзор методологии и инструментов на Java.	10.1
Cucumber JVM + Selenium Webdriver.	10.2
JBehave + Selenium Webdriver.	10.3
Алфавитный указатель	

Selenium

Selenium - это набор инструментов, предназначенных для автоматизации веб браузеров на различных платформах. **Selenium** может автоматизировать множество разнообразных браузеров на разных платформах, используя различные языки программирования и интегрируясь с разными тестовыми фреймворками.

Официальный сайт **Selenium** проекта: <http://docs.seleniumhq.org/>

Здесь можно найти большое количество первичной информации о **Selenium**, включая документацию, советы по использованию, текущий статус проекта, ссылки на исходный код, необходимые библиотеки, драйверы и многое другое.

Selenium Webdriver. Введение

Webdriver - популярный инструмент для управления реальным браузером, который можно использовать как для автоматизации тестирования веб приложений, так и для выполнения других рутинных задач, связанных с работой в вебе.

Кроме того, Webdriver - проект с открытым исходным кодом, поддерживает множество языков программирования и имеет большое сообщество пользователей.

WebDriver. Обзор и принцип работы

[Selenium Webdriver](#) - инструмент для автоматизации реального браузера, как локально, так и удаленно, наиболее близко имитирующий действия пользователя.

[Selenium 2](#) (или Webdriver) - последнее пополнение в пакете инструментов [Selenium](#) и является основным вектором развития проекта. Это абсолютно новый инструмент автоматизации. По сравнению с [Selenium RC](#) Webdriver использует совершенно иной способ взаимодействия с браузерами. Он напрямую вызывает команды браузера, используя родной для каждого конкретного браузера API. Как совершаются эти вызовы и какие функции они выполняют зависит от конкретного браузера. В то же время [Selenium RC](#) внедрял javascript код в браузер при запуске и использовал его для управления веб приложением. Таким образом, Webdriver использует способ взаимодействия с браузером более близкий к действиям реального пользователя.

Самое главное изменение новой версии [Selenium](#) - это Webdriver API.

[Selenium 1.0 \(RC\)](#) + WebDriver = [Selenium 2.0](#)

По сравнению с более старым интерфейсом он обладает рядом преимуществ:

- Интерфейс Webdriver был спроектирован более простым и выразительным;
- Webdriver обладает более компактным и объектно-ориентированным API;
- Webdriver управляет браузером более эффективно, а также справляется с некоторыми ограничениями, характерными для [Selenium RC](#), как загрузка и отправление файлов, попапы и диалоги.

Для работы с Webdriver необходимо 3 основных программных компонента:

1. *Браузер*, работу которого пользователь хочет автоматизировать. Это реальный браузер определенной версии, установленный на определенной ОС и имеющий свои настройки (по умолчанию или кастомные). На самом деле Webdriver может работать и с "ненастоящими" браузерами, но подробно о них позже.
2. Для управления браузером совершенно необходим *driver* браузера. Driver на самом деле является веб сервером, который запускает браузер и отправляет ему команды, а также закрывает его. У каждого браузера свой driver. Связано это с тем, что у каждого браузера свои отличные команды управления и реализованы они по-своему. Найти список доступных драйверов и ссылки для скачивания можно [на официальном сайте Selenium](#) проекта.
3. *Скрипт/тест*, который содержит набор команд на определенном языке программирования для драйвера браузера. Такие скрипты используют [Selenium Webdriver](#) bindings (готовые библиотеки), которые доступны пользователям на

различных языках.

Основные понятия и методы Selenium Webdriver API

Основными понятиями в Selenium Webdriver являются:

- Webdriver - самая важная сущность, ответственная за управление браузером. Основной ход скрипта/теста строится именно вокруг экземпляра этой сущности.
- WebElement - вторая важная сущность, представляющая собой абстракцию над веб элементом (кнопки, ссылки, инпута и др.). WebElement инкапсулирует методы для взаимодействия пользователя с элементами и получения их текущего статуса.
- By - абстракция над локатором веб элемента. Этот класс инкапсулирует информацию о селекторе(например, CSS), а также сам локатор элемента, то есть всю информацию, необходимую для нахождения нужного элемента на странице.

Сам процесс взаимодействия с браузером через Webdriver API довольно прост:

1. Нужно создать Webdriver:

```
WebDriver driver = new ChromeDriver();
```

При выполнении этой команды будет запущен Chrome, при условии, что он установлен в директорию по умолчанию и путь к ChromeDriver сохранен в системной переменной PATH.

2. Необходимо открыть тестируемое приложение (AUT), перейдя по url:

```
driver.get("http://mycompany.site.com");
```

Теоритически в хrome при этом должен открыться сайт компании.

3. Далее следует серия действий по нахождению элементов на странице и взаимодействию с ними:

```
By elementLocator = By.id("#element_id");  
WebElement element = driver.findElement(elementLocator));
```

Или более кратко:

```
WebElement element = driver.findElement(By.id("#element_id"));
```


После нахождения элемента, кликнем по нему:

```
element.click();
```

Далее следует совокупность похожих действий, как того требует сценарий.

4. В конце теста (часто также и в середине) должна быть какая-то проверка, которая и определит в конечном счете результат выполнения теста:

```
assertEquals("Webpage expected title", driver.getTitle());
```

Проверки может и не быть, если цель вашего скрипта - не тест, а выполнение какой-то рутины.

5. После теста надо закрыть браузер:

```
driver.quit();
```

Следует отметить, что для поиска элементов доступно два метода:

1. Первый - найдет только первый элемент, удовлетворяющий локатору:

```
WebElement element = driver.findElement(By.id("#element_id"));
```

2. Второй - вернет весь список элементов, удовлетворяющих запросу:

```
List<WebElement> elements = driver.findElements(By.name("elements_name"))
```

Более полную документацию Webdriver API можно найти по адресу:

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

Типы локаторов

Поскольку Webdriver - это инструмент для автоматизации веб приложений, то большая часть работы с ним это работа с веб элементами(WebElements). WebElements - ни что иное, как DOM объекты, находящиеся на веб странице. А для того, чтобы осуществлять какие-то действия над DOM объектами / веб элементами необходимо их точным образом определить(найти).

```
WebElement element = driver.findElement(By.<Selector>);
```

Таким образом в Webdriver определяется нужный элемент. By - класс, содержащий статические методы для идентификации элементов:

1.By.id

Пример:

```
<div id="element">
  <p>some content</p>
</div>
```

Поиск элемента:

```
WebElement element = driver.findElement(By.id("element_id"));
```

2.By.name

Пример:

```
<div name="element">
  <p>some content</p>
</div>
```

Поиск элемента:

```
WebElement element = driver.findElement(By.name("element_name"));
```

3.By.className

Пример:

```
<img class="logo">
```

Поиск элемента:

```
WebElement element = driver.findElement(By.className("element_class"));
```

4.By.TagName

Пример:

```
<div>
  <a class="logo" ref="...">...</a>
  <a class="support" ref="...">...</a>
</div>
```

Поиск элемента:

```
List<WebElement> elements = driver.findElements(By.tagName("a"));
```

5.By.LinkText

Пример:

```
<div>
  <a ref="...">text</a>
  <a ref="...">Another text</a>
</div>
```

Поиск элемента:

```
WebElement element = driver.findElement(By.linkText("text"));
```

6.By.PartialLinkText

Поиск элемента:

```
WebElement element = driver.findElement(By.partialLinkText("text"));
```

7.By.cssSelector

```
<div class='main'>
  <p>text</p>
  <p>Another text</p>
</div>
```

Поиск элемента:

```
WebElement element=driver.FindElement(By.cssSelector("div.main"));
```

8.By.XPath

```
<div class='main'>  
  <p>text</p>  
  <p>Another text</p>  
</div>
```

Поиск элемента:

```
WebElement element = driver.findElement(By.xpath("//div[@class='main']"));
```

Ожидания

Ожидания - неперенный атрибут любых UI тестов для динамических приложений. Нужны они для синхронизации работы [AUT](#) и тестового скрипта. Скрипт выполняется намного быстрее реакции приложения на команды, поэтому часто в скриптах необходимо дожидаться определенного состояния приложения для дальнейшего с ним взаимодействия.

Самый просто пример - переход по ссылке и нажатие кнопки:

```
driver.get("http://google.com");
driver.findElement(By.id("element_id")).click();
```

В данном случае необходимо дождаться пока не появится кнопка с `id = element_id` и только потом совершать действия над ней. Для этого и существуют ожидания.

Ожидания бывают:

1. Неявные ожидания - Implicit Waits - конфигурируют экземпляр WebDriver делать многократные попытки найти элемент (элементы) на странице в течении заданного периода времени, если элемент не найден сразу. Только по истечении этого времени WebDriver бросит `ElementNotFoundException`.

```
WebDriver driver = new FirefoxDriver();
driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
driver.get("http://some_url");
WebElement dynamicElement = driver.findElement(By.id("dynamicElement_id"));
```

Неявные ожидания обычно настраиваются сразу после создания экземпляра WebDriver и действуют в течении всей жизни этого экземпляра, хотя переопределить их можно в любой момент. К этой группе ожиданий также можно отнести неявное ожидание загрузки страницы:

```
driver.manage().timeouts().pageLoadTimeout(10, TimeUnit.SECONDS);
```

А также неявное ожидание отработки скриптов:

```
driver.manage().timeouts().setScriptTimeout(10, TimeUnit.SECONDS);
```

2. Явные ожидания - Explicit Waits - это код, который ждет наступления какого-то

события (чаще всего на [AUT](#)), прежде чем продолжит выполнение команд скрипта. Такое ожидание срабатывает один раз в указанном месте.

Самым худшим вариантом является использование `Thread.sleep(1000)`, в случае с которым скрипт просто будет ждать определенное количество времени. Это не гарантирует наступление нужного события либо будет слишком избыточным и увеличит время выполнения теста.

Более предпочтительно использовать `WebDriverWait` и `ExpectedCondition`:

```
WebDriver driver = new FirefoxDriver();
driver.get("http://some_url");
WebElement dynamicElement = (new WebDriverWait(driver, 10))
    .until(ExpectedConditions.presenceOfElementLocated(By.id("dynamicElement_id")));
```

В данном случае скрипт будет ждать элемента с `id = dynamicElement_id` в течении 10 секунд, но продолжит выполнение, как только элемент будет найден. При этом `WebDriverWait` класс выступает в роли конфигурации ожидания - задаем, как долго ждать события и как часто проверять его наступление. `ExpectedConditions` - статический класс, содержащий часто используемые условия для ожидания.

Пример использования Webdriver API

```
package org.openqa.selenium.example;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.support.ui.ExpectedCondition;
import org.openqa.selenium.support.ui.WebDriverWait;

public class WebDriverExample {
    public static void main(String[] args) {
        // Создаем экземпляр WebDriver
        // Следует отметить что скрипт работает с интерфейсом,
        // а не с реализацией.
        WebDriver driver = new FirefoxDriver();

        // Открываем гугл, используя драйвер
        driver.get("http://www.google.com");
        // По-другому это можно сделать так:
        // driver.navigate().to("http://www.google.com");

        // Находим элемент по атрибуту name
        WebElement element = driver.findElement(By.name("q"));

        // Вводим текст
        element.sendKeys("Selenium");

        // Отправляем форму, при этом драйвер сам определит как отправить форму по элемент
        element.submit();

        // Проверяем тайтл страницы
        System.out.println("Page title is: " + driver.getTitle());

        // Страницы гугл динамически отрисовывается с помощью javascript
        // Ждем загрузки страницы с таймаутом в 10 секунд
        (new WebDriverWait(driver, 10)).until(new ExpectedCondition<Boolean>() {
            public Boolean apply(WebDriver d) {
                return d.getTitle().toLowerCase().startsWith("selenium");
            }
        });

        // Ожидаем увидеть: "Selenium - Google Search"
        System.out.println("Page title is: " + driver.getTitle());

        // Закрываем браузер
        driver.quit();
    }
}
```


Selenium WebDriver. Сложные вопросы.

Локаторы. CSS, XPATH, JQUERY.

Ранее мы уже изучали локаторы, с помощью которых можно идентифицировать элементы на web странице. В этой главе мы пристальнее рассмотрим наиболее сложные из них, а именно CSS, Xpath и jQuery локаторы.


CSS локаторы

Многие браузеры реализуют CSS движок, чтобы разработчики смогли применять CSS таблицы в своих проектах. Что позволяет разделить между собой контент сраницы с её оформлением. В CSS есть паттерны, согласно которым стили, создаваемые разработчиком, применяются к элементам страницы (DOM). Эти паттерны называются локаторы (selectors). [Selenium WebDriver](#) использует тот же принцип для нахождения элементов. И он намного быстрее, чем поиск элементов на основе XPath, который мы рассмотрим чуть позже.

Давайте рассмотрим несколько базовых локаторов с использованием CSS:

Абсолютный путь:

```
WebElement userName = driver.findElement(By.cssSelector("html body div div form input"));
```



Относительный путь:

```
WebElement userName = driver.findElement(By.cssSelector("input"));
```

Поиск непосредственного дочернего элемента:

```
WebElement userName = driver.findElement(By.cssSelector("div > a"));
```

Поиск дочернего элемента любого уровня:

```
WebElement userName = driver.findElement(By.cssSelector("div a"));
```

Поиск по ID элемента:

```
WebElement userName = driver.findElement(By.cssSelector("input#username"));

//или

WebElement userName = driver.findElement(By.cssSelector("#username"));
```

Поиск по классу:

```
WebElement userName = driver.findElement(By.cssSelector("input.classname"));

//или

WebElement userName = driver.findElement(By.cssSelector(".classname"));
```

Поиск по значениям атрибутов html тегов:

```
WebElement previousButton = driver.findElement(By.cssSelector("img[alt='Previous']"));
```

Поиск по названию атрибутов:

```
List<WebElement> imagesWithAlt = driver.findElements(By.cssSelector("img[alt]"));
```

Поиск по началу строки:

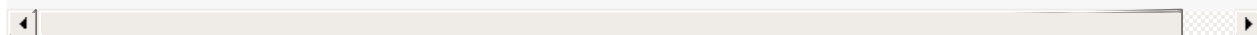
```
WebElement previousButton = driver.findElement(By.cssSelector("header[id^='page-']"));
```

Поиск по окончанию строки:

```
WebElement previousButton = driver.findElement(By.cssSelector("header[id$='page-']"));
```

Поиск по частичному совпадению строки:

```
WebElement previousButton = driver.findElement(By.cssSelector("header[id*='page-']"));
```



XPath локаторы

XPath (XML path) - это язык для нодов (nodes) в XML документе. Так как многие браузеры поддерживают XHTML, то мы можем использовать XPath для нахождения элементов на web страницах.

Важным различием между CSS и XPath локаторами является то, что, используя XPath, мы можем производить перемещение как в глубину DOM иерархии, так и возвращаться назад. Что же касается CSS, то тут мы можем двигаться только в глубину. Это значит, что с XPath можем найти родительский элемент, по дочернему.

Рассмотрим некоторые примеры:

Абсолютный путь:

```
WebElement userName = driver.findElement(By.xpath("html/body/div/div/form/input"));
```

Относительный путь:

```
WebElement userName = driver.findElement(By.xpath("//input"));
```

Поиск непосредственного дочернего элемента:

```
WebElement userName = driver.findElement(By.xpath("//div/a"));
```

Поиск дочернего элемента любого уровня:

```
WebElement userName = driver.findElement(By.xpath("//div//a"));
```

Поиск элемента по тексту:

```
WebElement userName = driver.findElement(By.xpath(".*[text()='Первая ссылка']/.."));
```

Поиск по значениям атрибутов:

```
WebElement userName = driver.findElement(By.xpath("//input[@id='username']"));
```

Поиск по названию атрибутов:

```
List<WebElement> imagesWithAlt = driver.findElements(By.xpath ("img[@alt]"));
```

Поиск родительского элемента:

```
WebElement userName = driver.findElement(By.xpath("//input[@id='username']/.."));
```

jQuery локаторы

Мы можем находить элементы, используя jQuery локаторы. Они используются, когда CSS локаторы изначально не поддерживаются браузерами.

Первое, что нужно сделать для их использования - проверить поддерживает ли приложение jQuery библиотеку. Если нет, то мы сами должны её добавить:

```
private static void addjQuery (JavascriptExecutor js) {  
  
    String script = "";  
  
    boolean needInjection = (Boolean)(js.executeScript("return this.$ === undefined;"));  
    if(needInjection) {  
        URL u = Resources.getResource("jquery.js");  
        try {  
            script = Resources.toString(u, Charsets.UTF_8);  
        } catch(IOException e) {  
            e.printStackTrace();  
        }  
        js.executeScript(script);  
    }  
}
```

Теперь нам нужен объект, с помощью которого мы смогли бы работать с JavaScript и, как следствие, с jQuery. Для этого мы будем использовать **JavaScriptExecutor**

```
WebDriver driver = new FirefoxDriver();  
driver.manage().window().maximize();  
  
JavascriptExecutor js = (JavascriptExecutor)driver;
```

Теперь всё готово, для того, чтобы мы могли использовать jQuery локаторы.

```
WebElement element = (WebElement) js.executeScript("return jQuery.find('#hplogo');");
```

А вот и сам локатор:

```
'#hplogo'
```

Так же мы можем использовать CSS локаторы внутри jQuery локаторов:

```
jQuery.find('input.someclass')
```

WebDriver API. Сложные взаимодействия.

[Selenium WebDriver](#) позволяет имитировать действия пользователя, начиная от простых движений мыши до сложных, перетягивание объекта. Все это позволяет реализовать класс **Actions**. Так же разработчики позаботились о том, чтобы мы могли создавать цепочку действий, используя этот класс. Рассмотрим некоторые возможности на следующих примерах.

Двойной щелчок на элементе:

```
@Test
public void testDoubleClick() throws Exception
{
    WebDriver driver = new ChromeDriver();
    driver.get("http://dl.dropbox.com/u/55228056/DoubleClickDemo.html");
    WebElement message = driver.findElement(By.id("message"));

    //Verify color is Blue
    assertEquals("rgb(0, 0, 255)",
        message.getCssValue("background-color").toString());
    Actions builder = new Actions(driver);
    builder.doubleClick(message).build().perform();

    //Verify Color is Yellow
    assertEquals("rgb(255, 255, 0)",
        message.getCssValue("background-color").toString());
    driver.close();
}
```

Перетягивание объекта:

```

@Test
public void testDragDrop() {
    driver.get("http://dl.dropbox.com/u/55228056/DragDropDemo.html");
    WebElement source = driver.findElement(By.id("draggable"));
    WebElement target = driver.findElement(By.id("droppable"));

    Actions builder = new Actions(driver);
    builder.dragAndDrop(source, target).perform();
    try
    {
        assertEquals("Dropped!", target.getText());
    } catch (Error e) {
        verificationErrors.append(e.toString());
    }
}

```

Другие полезные методы

Клик левой кнопкой мыши:

```

click()
click(WebElement onElement)

```

Клик с удержанием:

```

clickAndHold()
clickAndHold(WebElement onElement)

```

Правый клик:

```

contextClick()
contextClick(WebElement onElement)

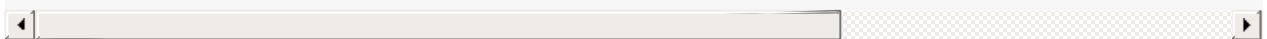
```

Пример работы с контекстным меню:

```

Actions builder = new Actions(driver);
builder.contextClick(webElement).sendKeys(Keys.ARROW_DOWN).sendKeys(Keys.ARROW_DOWN).send

```



Перетаскивание со смещением:

```

dragAndDropBy(WebElement source, int xOffset, int yOffset)

```

Нажатие и удержание клавиши и дальнейшее ее отпущание:

```
keyDown(Keys theKey) / keyDown(WebElement element, Keys key)
keyUp(Keys theKey) / keyUp(WebElement element, Keys key)
```

Смещение мыши:

```
moveByOffset(int xOffset, int yOffset)
```

Перемещение мыши на элемент:

```
moveToElement(WebElement toElement)
moveToElement(WebElement toElement, int xOffset, int yOffset)
```

Отпускание клавиши мыши:

```
release()
release(WebElement onElement)

//Вариант:
sendKeys(Keys.NULL)
```

Набор текста на клавиатуре:

```
sendKeys(java.lang.CharSequence... keysToSend)
sendKeys(WebElement element, java.lang.CharSequence... keysToSend)
```

Построение цепочки действий:

```
build()
```

Выполнение построенной цепочки действий:

```
perform()
```


Контроль за ходом теста. Кастомные ожидания, попапы, алерты, Iframes.

Ожидания

[Selenium WebDriver](#) имеет хороший набор стандартных "ожидалок"(waiters), но бывают случаи, когда их не достаточно. В этом случае мы можем написать собственную "ожидалку". В этом нам поможет класс ExpectedCondition.

Как же это работает? Давайте посмотрим на примере.

```

@Test
public void testExplicitWait()
{
    //Go to Sample Application
    WebDriver driver = new FirefoxDriver();
    driver.get("http://dl.dropbox.com/u/55228056/AjaxDemo.html");
    try {
        //Get the link for Page 4 and click on it, this will call AJAX code
        //for loading the contents for Page 4

        WebElement page4button = driver.findElement(By.linkText("Page4"));
        page4button.click();

        //Create Wait using WebDriverWait.
        //This will wait for 5 seconds for timeout before page4 element is found
        //Element is found in specified time limit test will move to the text step
        //instead of waiting for 10 seconds
        //Expected condition is expecting a WebElement to be returned
        //after findElement finds the
        //element with specified locator

        WebElement message = (new WebDriverWait(driver, 5))
            .until(new ExpectedCondition<WebElement>(){
                @Override
                public WebElement apply(WebDriver d) {
                    return d.findElement(By.id("page4"));
                }
            });
        assertTrue(message.getText().contains("Nunc nibh tortor"));
    } catch (NoSuchElementException e) {
        fail("Element not found!!");
        e.printStackTrace();
    } finally {
        driver.close();
    }
}

```

Например, если мы захотим что-то выполнить при появлении какого-либо элемента с использованием jQuery, то переопределяемый метод будет выглядеть так

```

WebElement element = (new WebDriverWait(driver, 10)).until(new ExpectedCondition<Boolean>
{
    public Boolean apply(WebDriver d) {
        JavascriptExecutor js = (JavascriptExecutor) d;
        return (Boolean)js.executeScript("return jQuery.active == 0");
    }
});

```

Еще одной распространенной задачей является работа с всплывающими окнами. Соответственно нам нужно переключиться на pop-up выполнить какие-либо действия(ввод данных, проверки и т д), а затем вернуться на родительское окно. Чтобы лучше в этом разобраться выполним простой пример:

```
@Test
public void testWindowPopup()
{
    //Save the WindowHandle of Parent Browser Window
    String parentWindowId = driver.getWindowHandle();

    //Clicking Help Button will open Help Page in a new Popup Browser Window
    WebElement helpButton = driver.findElement(By.id("helpbutton"));
    helpButton.click();
    try {
        //Switch to the Help Popup Browser Window
        driver.switchTo().window("HelpWindow");

        } catch (NoSuchWindowException e) {
            e.printStackTrace();
        }

    //Verify the driver context is in Help Popup Browser Window
    assertTrue(driver.getTitle().equals("Help"));

    //Close the Help Popup Window
    driver.close();

    //Move back to the Parent Browser Window
    driver.switchTo().window(parentWindowId);

    //Verify the driver context is in Parent Browser Window
    assertTrue(driver.getTitle().equals("Build my Car - Configuration"));
}
```

Работа с alert окнами

Проводя время в интернете вы не раз сталкивались с ситуациями, когда вы ошиблись при заполнении одного из полей или не выставили галочку где-нибудь, то появлялись окошки, требовавшие все исправить. Вот сейчас мы попробуем научиться их отлавливать и взаимодействовать с ними.

```
@Test
public void testSimpleAlert()
{
    //Clicking button will show a simple Alert with OK Button
    WebElement button = driver.findElement(By.id("simple"));
    button.click();

    try {
        //Get the Alert
        Alert alert = driver.switchTo().alert();

        //Get the Text displayed on Alert using getText() method of Alert class
        String textOnAlert = alert.getText();

        //Click OK button, by calling accept() method of Alert Class
        alert.accept();

        //Verify Alert displayed correct message to user
        assertEquals("Hello! I am an alert box!",textOnAlert);

    } catch (NoAlertPresentException e) {
        e.printStackTrace();
    }
}
```

Фреймы

Иногда разработчики хотят на одной странице использовать несколько окон или подокон. В разметке таких страниц вы обязательно встретите следующие теги `frameset` или `iframe`. Соответственно автоматизаторам необходимо уметь работать с ними. Помогать находить [Selenium](#) элементы в нужном подокне. Попробуем их различить с помощью имени или `id`. Рассмотрим пример:

```
@Test
public void testFrameWithIdOrName()
{
    //Activate the frame on left side using it's id attribute
    driver.switchTo().frame("left");

    //Get an element from the frame on left side and verify it's contents
    WebElement leftmsg = driver.findElement(By.tagName("p"));
    assertEquals("This is Left Frame", leftmsg.getText());

    //Activate the Page, this will move context from frame back to the Page
    driver.switchTo().defaultContent();

    //Activate the frame on right side using it's name attribute
    driver.switchTo().frame("right");

    //Get an element from the frame on right side and verify it's contents
    WebElement rightmsg = driver.findElement(By.tagName("p"));
    assertEquals("This is Right Frame", rightmsg.getText());

    //Activate the Page, this will move context from frame back to the Page
    driver.switchTo().defaultContent();
}
```

DDT подход

Рассмотрим классическую ситуацию - тестирование страницы логина на сайт. Нам нужно заполнить два поля username и password, а затем нажать кнопку sign in. Данный сценарий представится очень простым и написание для него теста не должно занять много времени. Но все мы прекрасно знаем, что данные подаваемые на вход могут быть различные: это и неверные username и password, и запрещенные символы, и просто поля могут оставить пустыми. Что же делать автоматизатору в таком случае? Писать отдельные тест для каждого возможного ввода? А если вариантов будет 200 или 300? Это чистой воды утопия. Чтобы избежать подобных проблем существует следующий подход к тестированию, который называется Data Driven Testing (DDT). DDT позволяет данные хранить отдельно от тестов. Наш написанный тест, каждый раз читает данные из хранилища (базаданных и т.д.) и выполняется, используя их. Продолжается это до тех пор пока тесты не будут запущены со всеми данными.

WebDriver предназначен для работы только лишь с API браузеров. Поэтому чтобы использовать в тестировании подход DDT нам понадобятся сторонние фреймворки. Рассмотрим наиболее популярные при написании JUnit и TestNG.

JUnit и DDT

JUnit — библиотека для модульного тестирования программного обеспечения на языке Java. Чтобы использовать его для DDT нам нужно будет "параметризировать" класс с тестами.

```
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.By;
import org.junit.*;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized.Parameters;
import org.junit.runners.Parameterized;
import static org.junit.Assert.*;
import java.util.Arrays;
import java.util.Collection;

@RunWith(value = Parameterized.class)
public class SimpleDDT {
    private static WebDriver driver;
    private static StringBuffer verificationErrors = new
        StringBuffer();
```

```

private String height;
private String weight;
private String bmi;
private String bmiCategory;

@Parameters
public static Collection testData() {
    return Arrays.asList(
        new Object[][] {
            {"160", "45", "17.6", "Underweight"},
            {"168", "70", "24.8", "Normal"},
            {"181", "89", "27.2", "Overweight"},
            {"178", "100", "31.6", "Obesity"}
        }
    );

    public SimpleDDT(String height, String weight, String bmi, String bmiCategory)
    {
        this.height = height;
        this.weight = weight;
        this.bmi = bmi;
        this.bmiCategory = bmiCategory;
    }

    @Test
    public void testBMICalculator() throws Exception {
        //Get the Height element and set the value using parameterised
        //height variable
        WebElement heightField = driver.findElement(By.name("heightCMS"));
        heightField.clear();
        heightField.sendKeys(height);

        //Get the Weight element and set the value using parameterised
        //Weight variable
        WebElement weightField = driver.findElement(By.name("weightKg"));
        weightField.clear();
        weightField.sendKeys(weight);

        //Click on Calculate Button
        WebElement calculateButton = driver.findElement(By.id("Calculate"));
        calculateButton.click();
        try {
            //Get the Bmi element and verify its value using parameterised
            //bmi variable
            WebElement bmiLabel = driver.findElement(By.name("bmi"));
            assertEquals(bmi, bmiLabel.getAttribute("value"));

            //Get the Bmi Category element and verify its value using
            //parameterised bmiCategory variable
            WebElement bmiCategoryLabel = driver.findElement(By.name("bmi_category"));
            assertEquals(bmiCategory, bmiCategoryLabel.
                getAttribute("value"));
        }
    }
}

```

```

        } catch (Error e) {
            //Capture and append Exceptions/Errors
            verificationErrors.append(e.toString());
            System.err.println("Assertion Fail "+ verificationErrors.
                toString());
        }
    }
}

```

TestNG

TestNG — это фреймворк для тестирования, написанный Java, он взял много чего с JUnit и NUnit, но он не только унаследовался от существующей функциональности JUnit, а также внедрения новых инновационных функций, которые делают его мощным, простым в использовании. Попробуем применить его:

```

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.By;
import org.testng.annotations.*;
import static org.testng.Assert.*;

public class TestNGDDT {
    private WebDriver driver;
    private StringBuffer verificationErrors = new StringBuffer();

    @DataProvider
    public Object[][] testData() {
        return new Object[][] {
            new Object[] { "160", "45", "17.6", "Underweight" },
            new Object[] { "168", "70", "24.8", "Normal" },
            new Object[] { "181", "89", "27.2", "Overweight" },
            new Object[] { "178", "100", "31.6", "Obesity" },
        };
    }

    @BeforeTest
    public void setUp() {
        // Create a new instance of the Firefox driver
        driver = new FirefoxDriver();
        driver.get("http://dl.dropbox.com/u/55228056/bmicalculator.html");
    }

    @Test(dataProvider = "testData")
    public void testBMICalculator(String height, String weight, String
        bmi, String category) {
        try {

```



```
        WebElement heightField = driver.findElement(By.name("heightCMS"));
        heightField.clear();
        heightField.sendKeys(height);

        WebElement weightField = driver.findElement(By.name("weightKg"));
        weightField.clear();
        weightField.sendKeys(weight);

        WebElement calculateButton = driver.findElement(By.id("Calculate"));
        calculateButton.click();

        WebElement bmiLabel = driver.findElement(By.name("bmi"));
        assertEquals(bmiLabel.getAttribute("value"), bmi);

        WebElement bmiCategoryLabel = driver.findElement(By.name("bmi_category"));
        assertEquals(bmiCategoryLabel.getAttribute("value"), category);

    } catch (Error e) {
        //Capture and append Exceptions/Errors
        verificationErrors.append(e.toString());
    }
}

@AfterTest
public void tearDown() {
    //Close the browser
    driver.quit();
    String verificationErrorString = verificationErrors.toString();
    if (!"".equals(verificationErrorString)) {
        fail(verificationErrorString);
    }
}
}
```

Page Object Pattern. Архитектура тестового проекта.

Использование паттерна Page Object.

Page object

Page Object - один из наиболее полезных и используемых архитектурных решений в автоматизации. Данный шаблон проектирования помогает инкапсулировать работу с отдельными элементами страницы, что позволяет уменьшить количество кода и его поддержку. Если, к примеру, дизайн одной из страниц изменён, то нам нужно будет переписать только соответствующий класс, описывающий эту страницу.

Основные преимущества:

- Разделение кода тестов и описания страниц
- Объединение всех действий по работе с веб-страницей в одном месте

Давайте посмотрим, как с этим работать. Для примера возьмём страницу логина, которая есть практически на всех сайтах. Создадим класс с описанием этой страницы, используя шаблон Page Object:

```

public class LoginPage {
    By usernameLocator = By.id("username");
    By passwordLocator = By.id("passwd");
    By loginButtonLocator = By.id("login");

    private final WebDriver driver;

    public LoginPage(WebDriver driver) {
        this.driver = driver;

        if (!"Login".equals(driver.getTitle())) {
            throw new IllegalStateException("This is not the login page");
        }
    }

    public LoginPage typeUsername(String username) {
        driver.findElement(usernameLocator).sendKeys(username);
        return this;
    }

    public LoginPage typePassword(String password) {
        driver.findElement(passwordLocator).sendKeys(password);
        return this;
    }

    public HomePage submitLogin() {
        driver.findElement(loginButtonLocator).submit();
        return new HomePage(driver);
    }

    public LoginPage submitLoginExpectingFailure() {
        driver.findElement(loginButtonLocator).submit();
        return new LoginPage(driver);
    }

    public HomePage loginAs(String username, String password) {
        typeUsername(username);
        typePassword(password);
        return submitLogin();
    }
}

```

```

}

```

Page Factory

Еще одним вариантом применения page object шаблона является использование класса Page Factory из библиотеки [Selenium](#). Давайте разберёмся, как с этим работать. Сначала нам нужно создать простой page object:

```
public class GoogleSearchPage {  
    private WebElement q;  
  
    public void searchFor(String text) {  
        q.sendKeys(text);  
        q.submit();  
    }  
}
```

Теперь чтобы всё работало корректно нам нужно инициализировать наш page object. Это выглядит так:

```
package org.openqa.selenium.example;  
  
import org.openqa.selenium.WebDriver;  
import org.openqa.selenium.WebElement;  
import org.openqa.selenium.htmlunit.HtmlUnitDriver;  
import org.openqa.selenium.support.PageFactory;  
  
public class UsingGoogleSearchPage {  
    public static void main(String[] args) {  
        WebDriver driver = new HtmlUnitDriver();  
  
        driver.get("http://www.google.com/");  
  
        GoogleSearchPage page = PageFactory.initElements(driver, GoogleSearchPage.class);  
  
        page.searchFor("Cheese");  
    }  
}
```

Альтернативные Page Object подходы.

Page Object и Thucydides

Одним из альтернативных подходов является использование фреймворка Thucydides. Он переводится как Фукидид, который был древнегреческим историком и полководцем, который во время Пелопоннесской войны прославился именно качественными репортажами, отсюда и название этого фреймворка.

Thucydides — это бесплатный проект с открытым исходным кодом. Разрабатываемый с 2011 года. Проект постоянно обновляется по мере добавления новой функциональности в новых версиях [Selenium](#).

Фреймворк реализует Page Object паттерн и позволяет уменьшить дублирование кода за счет использования еще одного типа классов между тестами и страницами, так называемых «степок».

Посмотрим, как выглядят класс "степок" и page object класс:

```
public class StepsinBook extends ScenarioSteps {
    public StepsinBook(Pages pages) {
        super(pages);
    }
    public BookPage getPageBook()
    {
        return  getPages().currentPageAt(BookPage.class);
    }
    @Step
    public void getMain(String url)
    {
        getPageBook().getMainPage(url);
    }
    @Step
    public void AllBooks()
    {
        getPageBook().allBooks();
    }
    @Step
    public void search(){
        getPageBook().search("Книга");
    }
    @Step
    public void catalog(){
        getPageBook().catalog();
    }
}
```

```
}

public class BookPage extends PageObject {
    @FindBy(linkText = "Все книги")
    WebElement allbooksButton;

    @FindBy(linkText = "Поиск")
    WebElement searchButton;

    @FindBy(name = "query")
    WebElement searchField;

    @FindBy(css = "button")
    WebElement searchBegin;

    public BookPage(WebDriver driver) {
        super(driver);
    }

    public void getMainPage(String url) {
        getDriver().get(url);
    }

    public void allBooks() {
        allbooksButton.click();
    }

    public void search(String searchWord) {
        searchButton.click();
        searchField.sendKeys(searchWord);
        searchBegin.click();
    }

}
```

Thucydides позволяет запускать тесты во всех браузерах, поддерживаемых [Selenium](#), и полностью берет на себя работу с драйвером, его настройку, запуск и остановку.

HTML Elements

Фреймворк HTML Elements — инструмент для простой работы с элементами веб-страниц в тестах.

HTML Elements позволяет собирать page-объекты как конструктор. Из типизированных элементов вы можете собирать нужные вам блоки, которые можно объединять, комбинировать друг с другом и собирать из них page-объекты. Это значительно

повышает степень переиспользования кода, делает его более читаемым и наглядным, а написание тестов — более простым.

Основной репозиторий: <https://github.com/yandex-qatools/html-elements>.

Документация: <https://github.com/yandex-qatools/html-elements/blob/master/README.md>.

Также документирован код проекта.

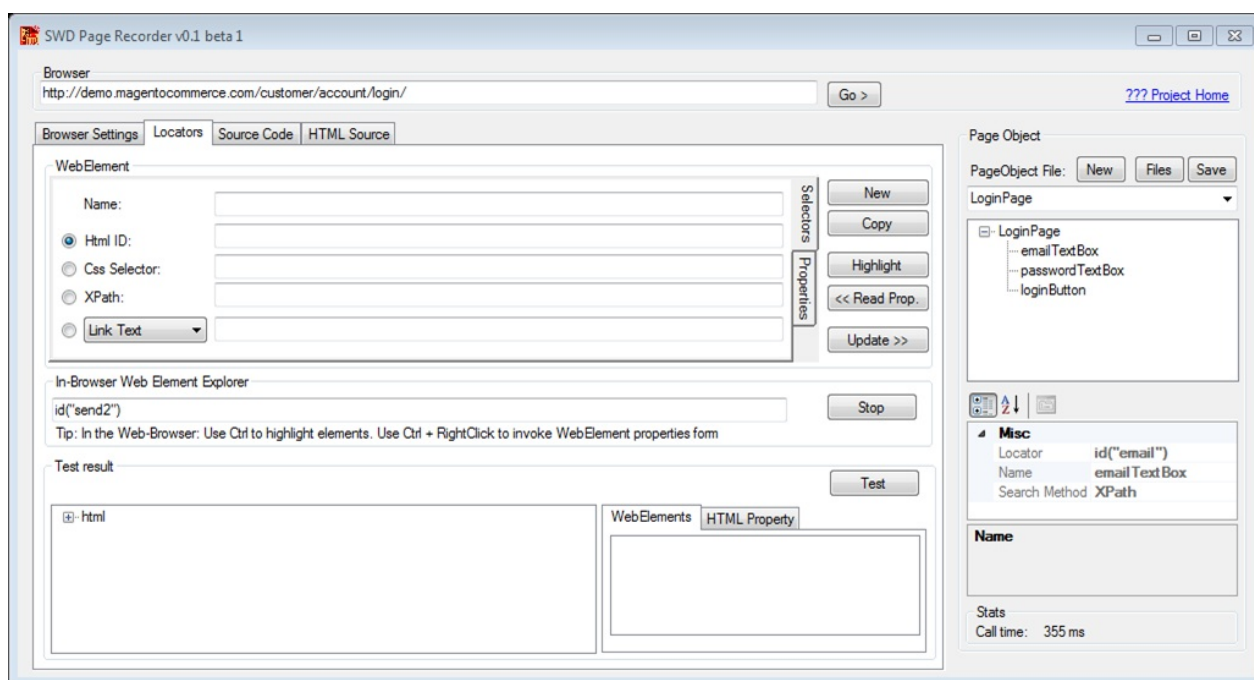
Примеры: <https://github.com/yandex-qatools/html-elements-examples>.

Вспомогательные инструмаенты.

SWD Page Recorder

SWD Page Recorder помогает записать локаторы элементов веб-страницы, отладить их в самом приложении и генерировать код PageObject-классов на C#, Java, Python, Ruby, Perl для дальнейшего использования в авто-тестах на [Selenium WebDriver](#). Данный инструмент позволяет не только найти необходимый локатор, но и оптимизировать его, и сгенерировать весь необходимый код для последующей вставки декларации элемента в код.

SWD Page Recorder - это единый инструментарий для работы с локаторами во всех браузерах, с которыми умеет работать WebDriver. Важным моментом является то, что Page Recorder тестирует всё через сам WebDriver – так что у вас не будет случаев, когда локатор, найденный другим путем – может не работать в WebDriver.



Домашняя страница: <http://swd-tools.com/>

Исходники: <https://github.com/dzharii/swd-recorder/releases>

Selenium Builder

Selenium Builder - это расширение для Firefox, позволяющие записывать тесты в бот-стиле. Работать с ним можно даже не зная языков программирования, как и с **Selenium** IDE.

Selenium Builder позволяет:

- записывать и проигрывать тесты
- поддерживает Selenium 1.0 и **Selenium** 2.0
- позволяет экспортировать записанные скрипты в языки программирования

Домашняя страница: <http://sebuilder.github.io/se-builder/>

Исходники: <https://github.com/sebuilder/se-builder>

Уровни абстракции. Создание кастомных элементов.

Уровни абстракции

В автоматизации часто применяются следующие абстракции:

- Page Object
- Page Element

Паттерн **Page Object** хорошо зарекомендовал себя в автоматизации. Основная идея – инкапсулировать логику поведения страницы в классе страницы. Таким образом, тесты будут работать не с низкоуровневым кодом, а с высокоуровневой абстракцией.

Плюсы Page Object:

- Разделение полномочий: вся логика страницы описывается в Page Object классах, а тестовые классы лишь используют их публичные методы и проверяют результат.
- DRY – все локаторы помещаются в одном месте
- Инкапсуляция работы с драйвером. Полезно при кросс-браузерном тестировании
- Page Objects позволяет записать локаторы в декларативном стиле

В классическом варианте, паттерн предполагает создание одного класса на одну страницу. Это может быть неудобно в ряде случаев:

- Использование кастомных элементов при создании веб-приложений
- Присутствие кастомных элементов на многих страницах

В решении этой проблемы может помочь использование наследования, но агрегация видится предпочтительнее. Поэтому лучше воспользоваться паттерном – Page Element. Page Elements – позволяет дробить страницу на более мелкие составляющие – блоки, виджеты и т.д. После чего эти блоки можно переиспользовать в нескольких страницах.

Кастомные элементы

WebDriver очень мощный инструмент, который позволяет выполнять различные действия над элементами страниц. Для этого используется класс WebElement. Но, как и всегда, не все так просто и замечательно. На практике автоматизаторы сталкиваются с тем, что разработчики используют кастомные элементы, с которыми

класс `WebElement` не работает. К примеру, возьмем написание тестов для таблиц. Стандартные классы `WebDriver` не работают с таблицами, потому нам нужно написать свой. В котором мы опишем работу с элементами таблицы (ячейки, строки и т. д.).

```
import java.util.List;

public class WebTable {
    private WebElement _webTable;

    public WebTable(WebElement webTable)
    {
        set_webTable(webTable);
    }

    public WebElement get_webTable() {
        return _webTable;
    }

    public void set_webTable(WebElement _webTable) {
        this._webTable = _webTable;
    }

    public int getRowCount() {
        List<WebElement> tableRows = _webTable.findElements(By.tagName("tr"));
        return tableRows.size();
    }

    public int getColumnCount() {
        List<WebElement> tableRows = _webTable.findElements(By.tagName("tr"));
        WebElement headerRow = tableRows.get(0);
        List<WebElement> tableCols = headerRow.findElements(By.tagName("td"));
        return tableCols.size();
    }

    public WebElement getCellEditor(int rowIdx, int colIdx, int editorIdx) throws NoSuchElementException
    {
        try {
            List<WebElement> tableRows = _webTable.findElements(By.tagName("tr"));
            WebElement currentRow = tableRows.get(rowIdx-1);
            List<WebElement> tableCols = currentRow.findElements(By.tagName("td"));
            WebElement cell = tableCols.get(colIdx-1);
            WebElement cellEditor = cell.findElements(By.tagName("input")).get(editorIdx);
            return cellEditor;
        } catch (NoSuchElementException e) {
            throw new NoSuchElementException("Failed to get cell editor");
        }
    }

    public WebElement getCellEditor(int rowIdx, int colIdx, int editorIdx) {
        List<WebElement> tableRows = _webTable.findElements(By.tagName("tr"));
        WebElement currentRow = tableRows.get(rowIdx-1);
        List<WebElement> tableCols = currentRow.findElements(By.tagName("td"));
        WebElement cell = tableCols.get(colIdx-1);
    }
}
```

```
        WebElement cellEditor = cell.findElements(By.tagName("input")).get(0);  
        return cellEditor;  
    }  
}
```

Архитектура. Основные элементы.

Что же такое архитектура? Это, по сути своей, структура разрабатываемой программы, описание того, как внутри устроены компоненты и как они друг с другом взаимодействуют. Кто-то может подумать, мол это же просто автотесты, зачем тут такие сложные процессы, проектирование архитектуры. Но это очень большое заблуждение.

Отсутствие архитектуры при проектировании тестового фреймворка приводит к многочисленным переписываниям кода, что в свою очередь требует серьезных временных затрат. Но, как говорится, "Время - деньги". Следствие всего этого потеря прибыли и недовольство заказчиков. Перейдем непосредственно к важным частям архитектуры.

Центральное место в архитектуре для автоматизации занимает паттерн Page Object, с которым вы уже знакомы. Он позволяет отделить логику описания веб-страниц от логики тестов. Так как же не будем забывать и о Page Element, который позволяет инкапсулировать работы с кастомными элементами.

Работая с паттерном Page Object, у нас возникает проблема: где удобнее хранить локаторы? Решить её можно создав UI Map. Для этого можно воспользоваться вариантами описанными ниже. Один из них это использовать файлы .properties.

.properties – текстовый формат и одноименное расширение имени файла, применяемое для сохр

Каждый параметр сохраняется как пара строк, первая содержит имя параметра (ключ), вторая значение параметра. Ключ и его значение в файле разделяются знаком равенства (ключ=значение). К примеру вы создали файл locators.properties, а его содержимое будет выглядеть так:

```
LoginPage.title>Login Page
LoginPage.userNameInput=id>Login
LoginPage.userPassInput=id>Password
LoginPage.loginButton=css=input[value>Login]
Grid.FieldFrom=xpath=//input[contains(@placeholder,'From')]
Grid.Datepicker=css=.datepicker
HomePage.title=Games
```

А чтобы использовать содержимое Properties файла, мы напишем парсер, который нам в этом поможет:

```
public class Locators {
    private static final Properties locators;

    private enum LocatorType{
        id, name, css, xpath, tag, text, partText;
    }

    static {
        locators = new Properties();
        InputStream is = Locators.class.getResourceAsStream("/locators.properties");
        try {
            locators.load(is);
        }
        catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }

    public static String title(String pageName) {
        return locators.getProperty(pageName);
    }

    public static By get(String locatorName) {
        String propertyValue = locators.getProperty(locatorName);
        return getLocator(propertyValue);
    }

    public static By get(String locatorName, String parameter) {
        String propertyValue = locators.getProperty(locatorName);
        return getLocator(String.format(propertyValue, parameter));
    }

    private static By getLocator(String locator){
        String[] locatorItems = locator.split("=",2);
        LocatorType locatorType = LocatorType.valueOf(locatorItems[0]);

        switch(locatorType) {

            case id :{
                return By.id(locatorItems[1]);
            }

            case name:{
                return By.name(locatorItems[1]);
            }

            case css:{
                return By.cssSelector(locatorItems[1]);
            }

            case xpath:{
                return By.xpath(locatorItems[1]);
            }
        }
    }
}
```

```

    }

    case tag:{
        return By.tagName(locatorItems[1]);
    }

    case text:{
        return By.linkText(locatorItems[1]);
    }

    case partText:{
        return By.partialLinkText(locatorItems[1]);
    }

    default:{
        throw new IllegalArgumentException("No such locator type: " + locatorItem);
    }
}
}

```

Другой способ организации работы с локаторами - это использование Html Elements, который объединяет элементы в блоки. Вот как выглядит пример такого блока:

```

@Name("Search form")
@Block(@FindBy(xpath = "//form"))
public class SearchArrow extends HtmlElement {
    @Name("Search request input")
    @FindBy(id = "searchInput")
    private TextInput requestInput;

    @Name("Search button")
    @FindBy(className = "b-form-button__input")
    private Button searchButton;

    public void search(String request) {
        requestInput.sendKeys(request);
        searchButton.click();
    }
}

```

Тогда Page Object будет описывать следующим образом:


```

public class SearchPage {
    private SearchArrow searchArrow;
    // Other blocks and elements here

    public SearchPage(WebDriver driver) {
        PageFactory.initElements(new HtmlElementDecorator(driver), this);
    }

    public void search(String request) {
        searchArrow.search(request);
    }

    // Other methods here
}

```

Еще одной важной частью архитектуры является создание Helper классов (вспомогательных классов). В них мы можем пометить методы, которые мы применяем на многих страницах, чтобы избежать дублирования кода, следуя принципу Don't repeat yourself. В такие классы можно поместить реализацию кастомных ожиданий или реализацию сложных действий:

```

public class Waiter {
    private static final int DEFAULT_TIME_OUT = 10;

    public static void waitFor(final ExpectedCondition condition){
        getWaiter().until(condition);
    }

    public static void waitForJquery(){
        getWaiter().until(new ExpectedCondition<Boolean>() {
            @Override
            public Boolean apply(WebDriver webDriver) {
                JavascriptExecutor js = (JavascriptExecutor) webDriver;
                return (Boolean) js.executeScript("return jQuery.active == 0");
            }
        });
    }

    //other waiters
}

```

Так же полезным окажется применение шаблон Flow. Он заключается в том, что описывая функционал веб-страниц, в методах мы возвращаем либо саму страницу, либо объект другой страницы. Такой подход еще больше похож на то, как работает пользователь: он либо выполняет действия на странице и нигде не переходит, либо нажав на какую-либо кнопку оказывается на другой странице веб-приложения.

Вот, пожалуй, одни из основных архитектурных решений, которые упростят вам жизнь, как автоматизаторам. Однако хочется сразу сказать, что универсального решения нет. Все зависит от особенностей проекта, используемых технологий и желания заказчика.

Selenium Grid и "headless" браузеры

Использование HtmlUnit драйвера в автотестировании

HtmlUnit - это, на текущий момент, самая быстрая и легковесная реализация WebDriver. Однако он не графический. Вы не сможете следить за происходящим. Такое решение очень понравится программистам, но часто нам нужно, чтобы мы могли либо сами посмотреть, либо показать заказчику как работают тесты. Это нужно, скажем так, для психологической уверенности в работоспособности тестов, что всё работает, как задумано. Хотя у нас и есть реальные результаты запуска тестов, например, в виде html отчётов.

Создание драйвера ничем не отличается от прочих:

```
WebDriver driver = new HtmlUnitDriver();
```

Плюсы

- Самая быстрая реализация WebDriver.
- Платформенно-независимое решение, так как используется только Java.
- Поддержка JavaScript.

Минусы

- Эмулирует реализацию JavaScript браузера

HtmlUnit для эмуляции JavaScript использует движок Rhino, который не используется ни в одном из лидеров среди браузеров. Поэтому мы можем получить вполне рабочие тесты, но при запуске их на реальном браузере могут возникнуть проблемы.

Включение JavaScript

```
HtmlUnitDriver driver = new HtmlUnitDriver();  
driver.setJavascriptEnabled(true);
```

Остальные команды не отличаются, от используемых другими драйверами:

```
Select select = new Select(driver.findElement(By.xpath("//select")));  
  
select.deselectAll();  
  
driver.findElement(By.id("submit")).click();
```

"Headless" тестирование с PhantomJS и SlimerJS

PhantomJS

PhantomJS — это сборка движка WebKit без графического интерфейса, позволяющая в режиме консоли загружать веб-страницу, выполнять JavaScript, полноценно работать с DOM, Canvas, CSS, JSON и SVG. WebKit лежит в основе таких популярных браузеров как Chrome и Safari. Вы имеете возможность интеграции с различными фреймворками для тестирования JavaScript и веб-страниц начиная от Jasmine до WebDriver. Для работы с PhantomJS браузером был реализован драйвер на чистом JavaScript, который получил название GhostDriver. Вот пример теста, в котором он используется:

```

package ghostdriver;

import org.junit.Test;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.phantomjs.PhantomJSDriver;

import static junit.framework.TestCase.assertEquals;

public class PhantomJSCommandTest extends BaseTest {
    @Test
    public void executePhantomJS() {
        WebDriver d = getDriver();
        if (!(d instanceof PhantomJSDriver)) {
            // Skip this test if not using PhantomJS.
            // The command under test is only available when using PhantomJS
            return;
        }

        PhantomJSDriver phantom = (PhantomJSDriver)d;

        // Do we get results back?
        Object result = phantom.executePhantomJS("return 1 + 1");
        assertEquals(new Long(2), (Long)result);

        // Can we read arguments?
        result = phantom.executePhantomJS("return arguments[0] + arguments[0]", new Long(
        assertEquals(new Long(2), (Long)result);

        // Can we override some browser JavaScript functions in the page context?
        result = phantom.executePhantomJS("var page = this;" +
            "page.onInitialized = function () { " +
                "page.evaluate(function () { " +
                    "Math.random = function() { return 42 / 100 } " +
                "})" +
            "});");

        phantom.get("http://ariya.github.com/js/random/");

        WebElement numbers = phantom.findElement(By.id("numbers"));
        boolean foundAtLeastOne = false;
        for(String number : numbers.getText().split(" ")) {
            foundAtLeastOne = true;
            assertEquals("42", number);
        }
        assert(foundAtLeastOne);
    }
}

```

Официальный сайт: <http://phantomjs.org/>

Репозиторий с Ghost Driver: <https://github.com/detro/ghostdriver>

SlimerJS

SlimerJS - это скриптовый браузер для разработчика имеющий в своем арсенале движок эквивалентный последнему Firefox(Gecko), который лежит в основе ещё нескольких браузеров. Его также можно использовать вместе с GhostDriver, благодаря своей схожести с PhantomJS.

Официальный сайт: <http://slimerjs.org/>

Репозиторий на GitHub: <https://github.com/laurentj/slimerjs>

Grid. Настройка и использование.

Описание

Selenium Grid – это кластер, включающий в себя несколько **Selenium**-серверов. Он позволяет организовать распределённую сеть, позволяющую параллельно запускать много браузеров на разных машинах.

Selenium Grid работает следующим образом: имеется центральный сервер (hub), к которому подключены узлы (node). Работа с кластером осуществляется через hub, при этом он просто транслирует запросы узлам. Узлы могут быть запущены на той же машине, что и hub или на других.

Сервер и узлы могут работать под управлением разных операционных систем, на них могут быть установлены разные браузеры. Одна из задач **Selenium Grid** заключается в том, чтобы «подбирать» подходящий узел по типу браузера, версии, операционной системы и другим атрибутам, заданным при старте браузера.

Начало работы с **Selenium Grid**

Сначала мы должны запустить центральный сервер (hub). Это делается с помощью следующей команды:

```
java -jar selenium-server-standalone-2.11.0.jar -role hub
```

После запуска команды можно перейти на панель администрирования хаба по адресу:

```
http://localhost:4444/grid/
```

Запускаем узлы кластера:

```
java -jar selenium-server-standalone-2.11.0.jar -role webdriver -hub http://localhost:4444
```

Если хаб запущен на другой машине, нужно заменить адрес в параметр hub. Если запускается несколько узлов на одной машине, указывайте разные значения параметра port.

Центральный сервер (hub) можно настраивать через JSON файлы:


```
{
  "host": null,
  "port": 4444,
  "newSessionWaitTimeout": -1,
  "servlets" : [],
  "prioritizer": null,
  "capabilityMatcher": "org.openqa.grid.internal.utils.DefaultCapabilityMatcher",
  "throwOnCapabilityNotPresent": true,
  "nodePolling": 5000,

  "cleanUpCycle": 5000,
  "timeout": 300000,
  "maxSession": 25,
  "maxInstances": 25
}
```

Настройка узлов (nodes) дополнительно включает в себя информацию о поддерживаемых браузерах:

```
{
  "capabilities":
    [
      {
        "browserName": "firefox",
        "maxInstances": 5,
        "seleniumProtocol": "WebDriver"
      },

      {
        "browserName": "chrome",
        "maxInstances": 5,
        "seleniumProtocol": "WebDriver"
      }
    ],
  "configuration":
  {
    "proxy": "org.openqa.grid.selenium.proxy.DefaultRemoteProxy",
    "maxSession": 25,
    "port": 5555,
    "host": ip,
    "register": true,
    "registerCycle": 5000,
    "hubPort": 4444
  }
}
```

Selenium Webdriver. Проблемные моменты

При автоматизации веб приложений часто возникают различного рода технические сложности, такие как, например, работа с нативными окнами операционной системы или работа с изображениями. Также могут вызывать сложности операции с файлами (download/upload), аутентификация (Basic Authentication Window), получение логов. В этой главе будет рассмотрено, как эти проблемы решаются с использованием [Selenium Webdriver](#) и некоторых вспомогательных инструментов.

Вспомогательные инструменты

Autolt

Autolt - это бесплатный простой и легковесный инструмент для автоматизации графических windows приложений. Он построен на похожем на BASIC скриптовом языке, с помощью которого симулируются нажатия клавиш, движение мыши и манипуляции с окнами и контролами для автоматизации тех или иных задач.

В комплект также входит:

- Autolt Window Info - инструмент для получения информации об окне и контролах, их атрибутах, необходимых для взаимодействия с ними.
- SciTE4Autolt3 - инструмент для создания и редактирования скриптов.
- Aut2Exe - инструмент для компилирования Autolt скриптов в запускаемые .exe файлы.

Скачать последнюю версию инструмента, а также найти детальную информацию о нем можно на официальном сайте: <https://www.autoitscript.com/site/>

Autolt скрипт для работы с окном загрузки файла может выглядеть следующим образом:

```
WinWaitActive("Open", "", "20")
If WinExists("Open") Then
    ControlSetText("Open", "", "Edit1", $CmdLine[1])
    ControlClick("Open", "", "&Open")
EndIf
```

Скомпилировав скрипт в FileDownloadHandler.exe, его можно вызвать в тесте после появления диалога загрузки файла следующим образом:

```
Runtime.getRuntime().exec(new String[] {"FileDownloadHandler.exe", "\"C:\\Picture.png\""})
```

Browser Mob Proxy

Browser Mob Proxy - это бесплатный прокси-сервер для веб браузера, с помощью которого можно отслеживать трафик, перехватывать и модифицировать запросы, создавать черные и белые списки ресурсов, имитировать медленную скорость соединения, собирать данные о производительности. Browser Mob Proxy можно использовать вместе с [Selenium Webdriver](#) или независимо.

Управлять прокси-сервером можно напрямую через Java интерфейс или через REST API. В мавен проект добавить зависимость можно указав в pom.xml файле:

```
<dependency>
  <groupId>net.lightbody.bmp</groupId>
  <artifactId>browsermob-proxy</artifactId>
  <version>2.0-beta-8</version>
  <scope>test</scope>
</dependency>
```

или

```
<dependency>
  <groupId>net.lightbody.bmp</groupId>
  <artifactId>browsermob-proxy</artifactId>
  <version>2.0-beta-8</version>
  <scope>test</scope>
  <exclusions>
    <exclusion>
      <groupId>org.seleniumhq.selenium</groupId>
      <artifactId>selenium-api</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

если вы используете свою версию [Selenium Webdriver](#).

Сам тест же может выглядеть так:

```
import net.lightbody.bmp.core.har.Har;
import net.lightbody.bmp.proxy.ProxyServer;

import org.junit.Test;
import org.openqa.selenium.Proxy;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.remote.CapabilityType;
import org.openqa.selenium.remote.DesiredCapabilities;

public class SimpleTest {

    @Test
    public void bmpTest() throws Exception {
        // запуск прокси сервера
        ProxyServer server = new ProxyServer(4444);
        server.start();

        // получение Selenium proxy
        Proxy proxy = server.seleniumProxy();

        // конфигурация FirefoxDriver для использования прокси
        DesiredCapabilities capabilities = new DesiredCapabilities();
        capabilities.setCapability(CapabilityType.PROXY, proxy);

        WebDriver driver = new FirefoxDriver(capabilities);

        // открытие страницы
        driver.get(SOME_URL);

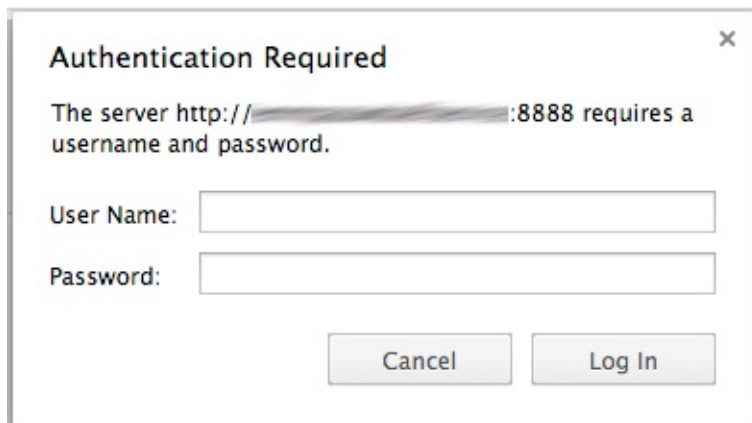
        // здесь основная часть теста

        driver.quit();
        server.stop();
    }
}
```

Официальный сайт Browser Mob Proxy <http://bmp.lightbody.net/>

Basic Authentication Window

Basic авторизация - это самый простой способ ограничения доступа к веб-приложениям и документам, предусмотренный стандартом протокола HTTP. При попытке обращения к таким ресурсам браузер формирует диалоговое окно, в котором предлагается ввести свой логин и пароль, после чего запрос выполняется повторно с предоставлением серверу данных для идентификации.



Сайты, использующие basic авторизацию, встречаются нечасто, чаще это веб-приложения, используемые для административных или корпоративных целей.

Основная проблема при тестировании таких приложений это нативные диалоговые окна, при появлении которых, также как и алертов, дальнейшее управление браузером с помощью [Selenium](#) становится просто невозможным.

Одним из способов избежать появления диалогового окна может быть использование специального URL формата <http://username:password@example.com/> для передачи учетных данных. Эта конструкция все еще поддерживается некоторыми браузерами, однако такое решение является официально устаревшим и иногда может отрицательно влиять на корректное выполнение дальнейших запросов на странице.

Browser Mob Proxy

При использовании basic аутентификации имя пользователя и пароль включаются в состав веб-запроса в стандартный HTTP заголовок "Authorization". Поэтому для успешного прохождения авторизации достаточно изменить HTTP заголовок перед отправкой на сервер. Сам [Selenium](#) не умеет манипулировать отправляемыми запросами, но для этих целей отлично подойдет прокси-сервер, в частности

BrowserMob Proxy в силу простоты его подключения. В случае basic авторизации `net.lightbody.bmp.proxy.ProxyServer` предоставляет метод для ее автоматического выполнения:

```
server.autoBasicAuthorization("example.com", "username", "password");
```

Первый аргумент `autoBasicAuthorization` это не URL, а доменное имя. Он не должен содержать `http://` или других частей URL. Для того, чтобы использовать учетные данные для любого домена нужно оставить первый параметр пустым.

Сам тест же может выглядеть так:

```
import net.lightbody.bmp.core.har.Har;
import net.lightbody.bmp.proxy.ProxyServer;

import org.junit.Test;
import org.openqa.selenium.Proxy;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.remote.CapabilityType;
import org.openqa.selenium.remote.DesiredCapabilities;

public class SimpleTest {

    @Test
    public void bmpTest() throws Exception {
        // запуск прокси сервера
        ProxyServer server = new ProxyServer(4444);
        server.autoBasicAuthorization("example.com", "username", "password");
        server.start();

        // получение Selenium proxy
        Proxy proxy = server.seleniumProxy();

        // конфигурация FirefoxDriver для использования прокси
        DesiredCapabilities capabilities = new DesiredCapabilities();
        capabilities.setCapability(CapabilityType.PROXY, proxy);

        WebDriver driver = new FirefoxDriver(capabilities);

        // открытие страницы
        driver.get("http://example.com/index");

        // здесь основная часть теста

        driver.quit();
        server.stop();
    }
}
```

Главное преимущество Browser Mob Proxy при работе с basic-аутентификацией - простота и кроссплатформенность (работает на Windows и Unix-based системах). Однако Browser Mob Proxy не предоставляет методов для работы с другими протоколами (Kerberos, NTLM).

AutoIT

Для автоматизации прохождения basic-авторизации можно также использовать AutoIt: нужно перехватить диалоговое окно, ввести учетные данные и подтвердить ввод. Казалось бы ничего сложного, но каждый браузер для авторизации формирует окна разных классов: Firefox использует свои диалоговые окна, IE — класс диалоговых окон Windows, а Chrome и вовсе не создает отдельного окна для ввода. В этом можно убедиться используя утилиту AutoIt Window Info. В единичном случае, например, для Firefox скрипт AutoIt может выглядеть следующим образом:

```
Local $classForBasicAuthWindow = "[CLASS:MozillaDialogClass]";

;ожидание появления окна 10 секунд
WinWait($classForBasicAuthWindow, "", 10)

If WinExists($classForBasicAuthWindow) Then
    WinActivate($classForBasicAuthWindow)

    ;имя пользователя
    Send($CmdLine[1] & "{TAB}");

    ;пароль
    Send($CmdLine[2] & "{ENTER}");

EndIf
```

В случае с авторизацией процесс AutoIt нужно запускать до открытия страницы драйвером, чтобы появившееся окно не заблокировало выполнение теста (скрипт скомпилирован в auth.exe):


```
public class SimpleBawTest {

    private final String username = "username";
    private final String password = "password";

    @Test
    public void bawTest() throws Exception {
        WebDriver driver = new FirefoxDriver();
        File autoIt = new File("/auth.exe");

        // запуск exe с передачей учетных данных в качестве параметра
        Process p = Runtime.getRuntime().exec(autoIt.getAbsolutePath()
        + " " + username + " " + password);

        driver.get("https://example.com");
        driver.findElement(By.className("sign-out"));
        driver.quit();
    }
}
```

AutoIt работает с графическим окном (UI), а не через протокол, поэтому с его помощью можно автоматизировать различные нативные окна, но его главный недостаток - он работает только на Windows платформе, что автоматически лишает тесты кроссплатформенности.

Загрузка файла

Иногда в автоматизации тестировании приходится проверять загрузку файлов (download). Главное правило, касающееся этого функционала: **избегайте загрузки файлов в ваших тестах**.

Дело в том, что как правило протестировать, то что вы хотите протестировать можно без непосредственного сохранения на диск файлов.

Пример:

```
package com.lazerycode.selenium.Tests;
import com.lazerycode.selenium.PageObjects.LoginPage;
import com.lazerycode.selenium.PageObjects.SecretFiles;
import com.lazerycode.selenium.SeleniumBase;
import com.lazerycode.selenium.Utility.FileDownloader;
import org.testng.annotations.Test;
import java.io.File;
import static com.lazerycode.selenium.Utility.CheckFileHash.getFileHash;
import static com.lazerycode.selenium.Utility.TypeOfHash.SHA1;
import static com.thoughtworks.selenium.SeleneseTestNgHelper.assertEquals;

public class LogInGetAFileAndCheckItTest extends SeleniumBase {

    @Test
    public void logInAndDownloadSecretFile() throws Exception {
        getDriverObject().get("http://localhost:8080/downloads/");
        LoginPage loginPage = new LoginPage();
        SecretFiles secretFilesPage = loginPage.logIn("foo", "bar");
        FileDownloader fileDownloader = new FileDownloader(getDriverObject());
        fileDownloader.setURI(secretFilesPage.getSecretFileHref());
        File secretFile = fileDownloader.downloadFile();
        int httpStatusCode = fileDownloader.getLastDownloadHTTPStatus();
        assertEquals(httpStatusCode, 200);
        assertEquals(getFileHash(secretFile, SHA1), ("781811ab9052fc61e109012acf5f22da89f"));
    }
}
```

Пример взят отсюда <https://github.com/Ardesco/What-Did-You-Download>. В нем используется кастомное расширение для Webdriver `FileDownloader`, который позволяет загрузить файл как временный по атрибуту `href="путь-к-файлу"`, убедиться, что загрузка была успешна (`assertEquals(httpStatusCode, 200);`), а также сравнить

содержимое файла с эталонным, сравнив их хеши (`assertEquals(getFileHash(secretFile, SHA1), ("781811ab9052fc61e109012acf5f22da89f2a5be"));`). Исходный код расширения можно найти по адресу <https://github.com/Ardesco/Powder-Monkey>

Отправление файла (upload)

Отправление (upload) файла обычно является довольно простой задачей. Она сводится к нахождению `input` элемента с атрибутом `type = "file"`. Далее нужно ввести путь к файлу и нажать кнопку "submit".

Например:

```
import org.openqa.selenium.*;
import java.net.URL;

public class SimpleFileUploadTest {
    @Test
    public void uploadTest() throws Exception {
        WebDriver driver = new FirefoxDriver(capabilities);
        driver.get("http://the-internet.herokuapp.com/upload");
        WebElement upload = driver.findElement(By.id("file-upload"));
        upload.sendKeys("your/path/here.file");
        driver.findElement(By.id("file-submit")).click();
        //make assertions here
        driver.quit();
    }
}
```

Однако есть одна особенность. Если тесты будут запускаться удаленно, то необходима дополнительная настройка при создании Webdriver:

```
DesiredCapabilities capabilities = DesiredCapabilities.firefox();
driver = new RemoteWebDriver(new URL(REMOTE_HUB_URL), capabilities);
driver.setFileDetector(new LocalFileDetector());
```

Метод `setFileDetector` говорит вебдрайверу, что файл загружается с локальной машины на удаленный сервер вместо обычного указания локального пути к файлу. В таком случае вебдрайвер отправит файл, закодированный в base64 формате, по JSON Wire протоколу на сервер прежде, чем вводить путь к файлу.

Логгирование в Selenium Webdriver

Логгирование важно как для разработчиков так и для пользователей Webdriver. Пользователи хотят иметь возможность быстро понять почему упал тест, в то время как разработчик хочет получить информацию, почему Webdriver упал. Хорошее логгирование может сделать дебаггинг намного проще.

Типы логов

Webdriver может быть использован в различных конфигурациях. К примеру, клиент запускающий тест (Java) может напрямую взаимодействовать с драйвером (локальный запуск), который контролирует браузер, или опосредованно через сервер (удаленный запуск). В зависимости от конфигурации могут быть доступны различные типы логов. Поэтому webdriver предоставляет возможность узнать типы поддерживаемых логов.

```
Set<String> logTypes = getWebDriver().manage().logs().getAvailableLogTypes();
```

В целом при дебаггинге определенной конфигурации было бы полезно получить доступ к логам каждого отдельного компонента конфигурации. А это: клиент (Java bindings), сервер (если Grid), драйвер и браузер. Все же логи всех компонентов не могут быть доступны во всех конфигурациях. Пример с использованием сервера (или не использованием) - самый простой, но логи также могут не поддерживаться или их невозможно будет получить из-за браузера. Впридачу к этому, иногда могут быть доступны и некоторые другие типы логов. Например, могут быть доступны логи, предназначенные для сбора данных о производительности на стороне клиента.

Таким образом, известные типы доступных для webdriver логов:

Типы логов	Значение
browser	Логи от javascript консоли браузера
client	Логи от клиентской части протокола Webdriver (например, Java bindings)
driver	Внутренние логи драйвера (например, логи FirefoxDriver)
performance	Логи относящиеся к производительности на странице (тайминги загрузки ресурсов)
server	Логи от Selenium сервера

Более подробно о типах логов и их получении можно найти на странице спецификации JsonWireProtocol <https://code.google.com/p/selenium/wiki/JsonWireProtocol>

Логи состоят из записей, каждая из которых содержит время (timestamp), уровень (log level) и сообщение.

Уровни логгирования (log levels)

Предназначение уровней логгирования - это предоставить пользователю способ фильтрации сообщений лога в зависимости от уровня интереса пользователя. Например, если цель логгирования - дебаггинг, то практически все сообщения будут нужны, в то время как при общем мониторинге понадобится меньше информации.

Для Webdriver были разработаны следующие уровни логгирования (в порядке возрастания детализированности):

- OFF: Логгирование отключено
- SEVERE: Сообщения об ошибках. К примеру, при неизвестной команде.
- WARNING: Предупреждения о том, что могло быть неверным, хоть ситуация и было успешно обработано. Например, перехваченное исключение.
- INFO: Сообщения информативного характера. Например, о полученных командах.
- DEBUG: Сообщения для дебаггинга. Например, информация о состоянии драйвера.
- ALL: Все сообщения. Это способ получить все сообщения независимо от его уровня.

Языки программирования вроде Java или Python предоставляют свои API логгирования со своими уровнями. Интерфейс вебдрайвера `Log` может работать с Java `LogType`, как в примере ниже. Однако следует иметь в виду, что внутри webdriver API уровень логгирования языка программирования должен быть преобразован в уровень логгирования Webdriver, прежде чем отправится запрос.

Получение логов

Если тип лога поддерживается, то его можно получить через интерфейс вебдрайвера. Для получения логов удаленного нода ([Selenium Grid](#)) используется wire protocol для его передачи на сервер (hub). При сценарии, один и тот же лог запрашивается несколько раз, одни и те же записи не будут получены дважды. В целях сохранения памяти после каждого запроса о получении лога, буфер этого лога обнуляется. Таким образом, каждое сообщение лога можно получить лишь раз независимо от количества попыток.

Для получения логов в Java следует использовать `Log` интерфейс:

```
Logs logs = driver.manage().logs();
LogEntries logEntries = logs.get(LogType.DRIVER);

for (LogEntry logEntry : logEntries) {
    System.out.println(logEntry.getMessage());
}
```

Более подробно об этом интерфейсе можно найти на

<http://selenium.googlecode.com/git/docs/api/java/org/openqa/selenium/logging/Logs.html>

Конфигурация логгирования

Бывают ситуации, когда логгирование необходимо полностью отключить, или сделать минимальным количество сообщений. Например, в случае, когда драйвер работает на устройстве с ограниченными ресурсами. Для поддержки таких ситуаций должна быть возможность конфигурировать логгирование как отдельная настройка в вебдрайвере. Реализовано это путем передачи драйверу списка пар тип лога - уровень логгирования. В большинстве реализаций клиентской части это сделано через

`DesiredCapabilities` класс.

Пример для Firefox:

```
LoggingPreferences logs = new LoggingPreferences();
logs.enable(LogType.BROWSER, Level.OFF);
logs.enable(LogType.CLIENT, Level.SEVERE);
logs.enable(LogType.DRIVER, Level.WARNING);
logs.enable(LogType.PERFORMANCE, Level.INFO);
logs.enable(LogType.SERVER, Level.ALL);

DesiredCapabilities desiredCapabilities = DesiredCapabilities.firefox();
desiredCapabilities.setCapability(CapabilityType.LOGGING_PREFS, logs);

WebDriver driver = new FirefoxDriver(desiredCapabilities);
```

Поведение по умолчанию - это собирать все сообщения и позволить пользователю отфильтровать их, как ему нужно. Если же для определенного типа логов было настроено другое поведение, то сообщения, более детализированного уровня, нежели установленный, не будут собираться.

Скриншоты элементов и работа с изображением

Интерфейс `TakesScreenshot` позволяет делать скриншоты целой страницы, текущего окна, видимой части страницы или всего дисплея, если это поддерживается браузером. Однако он не позволяет делать скриншоты отдельных элементов.

Мы можем расширить функциональность `TakesScreenshot` для возможности получения скриншотов элементов страницы с помощью Java Image API. Для этого можно использовать вспомогательный метод:

```
public static File captureElementBitmap(WebDriver driver, WebElement element) throws Exception {
    // Делаем скриншот страницы
    File screen = ((TakesScreenshot) driver).getScreenshotAs(OutputType.FILE);
    // Создаем экземпляр BufferedImage для работы с изображением
    BufferedImage img = ImageIO.read(screen);
    // Получаем размеры элемента
    int width = element.getSize().getWidth();
    int height = element.getSize().getHeight();
    // Создаем прямоугольник (Rectangle) с размерами элемента
    Rectangle rect = new Rectangle(0, 0, width, height);
    // Получаем координаты элемента
    Point p = element.getLocation();
    // Вырезаем изображение элемента из общего изображения
    BufferedImage dest = img.getSubimage(p.getX(), p.getY(), rect.width, rect.height);
    // Перезаписываем File screen
    ImageIO.write(dest, "png", screen);
    // Возвращаем File с изображением элемента
    return screen;
}
```

Сам тест с использованием этого метода может выглядеть так (при условии, что статический метод создан в классе `Utils`):


```
@Test
public void elementScreenshotTest(){
    WebDriver driver = new FirefoxDriver();
    driver.get(URL);
    WebElement element = driver.findElement(By.id("elem_id"));
    try {
        FileUtils.copyFile(Utils.captureElementBitmap(driver, element), new File("c:\\tmp
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Иногда в тестировании появляется необходимость сравнивать изображения. Например, для проверки загруженных иконок и изображений на сайте или для сравнения базового лэйаута на странице с текущим.

Webdriver обладает функционалом для снятия скриншотов, однако он не может сравнивать изображения. Для этого надо писать собственные расширения. В качестве примера рассмотрим следующий вспомогательный класс:

```

import java.awt.Image;
import java.awt.Toolkit;
import java.awt.image.PixelGrabber;

public class CompareUtil {
    public enum Result { Matched, SizeMismatch, PixelMismatch };
    static Result CompareImage(String baseFile, String actualFile) {
        Result compareResult = Result.PixelMismatch;
        Image baseImage = Toolkit.getDefaultToolkit().getImage(baseFile);
        Image actualImage = Toolkit.getDefaultToolkit().
            getImage(actualFile);
        try {
            PixelGrabber baseImageGrab = new PixelGrabber(baseImage, 0, 0, -1, -1, false)
            PixelGrabber actualImageGrab = new PixelGrabber(actualImage, 0, 0, -1, -1, fa
            int[] baseImageData = null;
            int[] actualImageData = null;
            if(baseImageGrab.grabPixels()) {
                int width = baseImageGrab.getWidth();
                int height = baseImageGrab.getHeight();
                baseImageData = new int[width * height];
                baseImageData = (int[])baseImageGrab.getPixels();
            }
            if(actualImageGrab.grabPixels()) {
                int width = actualImageGrab.getWidth();
                int height = actualImageGrab.getHeight();
                actualImageData = new int[width * height];
                actualImageData = (int[])actualImageGrab.getPixels();
            }
            System.out.println(baseImageGrab.getHeight() + "<>" +
                actualImageGrab.getHeight());
            System.out.println(baseImageGrab.getWidth() + "<>" +
                actualImageGrab.getWidth());
            if ((baseImageGrab.getHeight() != actualImageGrab.getHeight()) || (baseImageG
                compareResult = Result.SizeMismatch;
            }
            else if(java.util.Arrays.equals(baseImageData,actualImageData)){
                compareResult = Result.Matched;
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        return compareResult;
    }
}

```

Пример теста с использованием этого класса:

```
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.*;
import org.apache.commons.io.FileUtils;
import org.junit.*;
import static org.junit.Assert.*;
import java.io.File;

public class ScreenShotTest {
    public WebDriver driver;
    private StringBuffer verificationErrors = new StringBuffer();

    @Before
    public void setUp() throws Exception {
        // Create a new instance of the Firefox driver
        driver = new FirefoxDriver();
    }

    @Test
    public void imageCompareTest() throws Exception {
        String scrFile = "FILE_PATH";
        String baseScrFile = "BASE_FILE_PATH";
        // Заходим на страницу
        driver.get(URL);
        File screenshotFile = ((TakesScreenshot) driver).getScreenshotAs(OutputType.FILE);
        FileUtils.copyFile(screenshotFile, new File(scrFile));
        try {
            //Сравниваем изображение с базовым
            assertEquals(CompareUtil.Result.Matched, CompareUtil.CompareImage(baseScrFile
        } catch (Error e) {
            // Сохраняем ошибки в переменную
            verificationErrors.append(e.toString());
        }
    }

    @After
    public void tearDown() throws Exception {
        //Закрываем браузер
        driver.quit();
        String verificationErrorString = verificationErrors.toString();
        if (!"".equals(verificationErrorString)) {
            fail(verificationErrorString);
        }
    }
}
```

Selenium Webdriver. Тестирование HTML5 веб приложений

Автоматизация Canvas элементов.

Одним из нововведений HTML 5 был тег < canvas >, с помощью которого можно было создавать различные графики и рисунки используя JavaScript. Многие разработчики стали активно его применять при создании веб-приложений. Поэтому любой автоматизатор может легко оказаться на проекте, где нужно будет автоматизировать работу и с Canvas элементами. Так давайте поробуем с помощью [Selenium WebDiver](#) это сделать:

```
@Test
public void testHTML5CanvasDrawing() throws Exception {
    //Get the HTML5 Canvas Element
    WebElement canvas = driver.findElement(By.id("imageTemp"));

    //Select the Pencil Tool
    Select drawtool = new Select(driver.findElement(By.id("dtool")));
    drawtool.selectByValue("pencil");

    //Create a Action Chain for Draw a shape on Canvas
    Actions builder = new Actions(driver);
    builder.clickAndHold(canvas).moveByOffset(10, 50).
        moveByOffset(50,10).
        moveByOffset(-10, -50).
        moveByOffset(-50, -10).release().perform();

    //Get a screenshot of Canvas element after Drawing and
    //compare it to the base version to verify if the Drawing is performed
    FileUtils.copyFile(WebElementExtender.captureElementBitmap(canvas), new File("c:\\tmp
    assertEquals(CompareUtil.Result.Matched, CompareUtil.CompareImage("c:\\tmp\\base_post
}
```

Автоматизация видео плеера.

Ранее, чтобы посмотреть видео в интернете нужно было использовать различные плагины. При этом не существовало чего-то универсального. Однако, после выхода HTML 5 ситуация изменилась. Появился тег `< video >`, с помощью которого можно было добавлять легко видео на сайт. Соответственно, тестировать его тоже нужно. Для этого будем использовать класс `JavaScriptExecutor`. Напишем простой тест и посмотрим, как это работает:

```
@Test
public void testHTML5VideoPlayer() throws Exception {
    File scrFile = null;

    //Get the HTML5 Video Element
    WebElement videoPlayer = driver.findElement(By.id("vplayer"));

    //We will need a JavaScript Executor for interacting
    //with Video Element's
    //methods and properties for automation
    JavascriptExecutor jsExecutor = (JavascriptExecutor) driver;

    //Get the Source of Video that will be played in Video Player
    String source = (String) jsExecutor.executeScript("return arguments[0].currentSrc;",

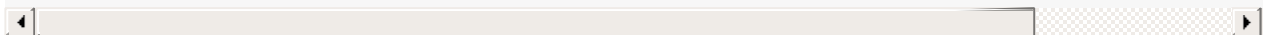
    //Get the Duration of Video
    long duration = (Long) jsExecutor.executeScript("return arguments[0].duration", video
    System.out.println(duration);

    //Verify Correct Video is loaded and duration
    assertEquals("http://html5demos.com/assets/dizzy.mp4", source);
    assertEquals(25, duration);

    //Play the Video
    jsExecutor.executeScript("return arguments[0].play()", videoPlayer);
    Thread.sleep(5000);

    //Pause the video
    jsExecutor.executeScript("arguments[0].pause()", videoPlayer);

    //Take a screen-shot for later verification
    scrFile = ((TakesScreenshot)driver).getScreenshotAs(OutputType.FILE);
    FileUtils.copyFile(scrFile, new File("c:\\tmp\\pause_play.png"));
}
```



Работа с web storage.

Cookie-файлы используются с самого начала применения JavaScript, поэтому хранение данных в веб-формате не является чем-то инновационным. Однако Web Storage является куда более мощным инструментом для хранения данных, так как обеспечивает больший уровень безопасности, повышенную скорость и простоту использования. Кроме того, в Web Storage можно хранить большие объемы данных. Ограничения определяются конкретным браузером, но обычно допускается использование от 5 до 10 Мбайт, что более чем достаточно для HTML-приложений. Еще одно преимущество — данные не загружаются с каждым запросом на сервер. Впрочем, существует одно ограничение — нельзя обобществлять веб-хранилища между браузерами: если вы сохранили данные в Safari, они будут недоступны Mozilla Firefox.

Теперь же посмотрим, что мы можем сделать с этим хранилищем данных, используя возможности WebDriver.

```
@Test
public void testHTML5LocalStorage() throws Exception {
    String lastName;
    JavascriptExecutor jsExecutor = (JavascriptExecutor) driver;

    //Get the current value of localStorage.lastname, this should be Smith
    lastName = (String) jsExecutor.executeScript("return localStorage.lastname;");
    assertEquals("Smith", lastName);
}
```

Selenium Webdriver. Расширение инструмента

Selenium "обертки" и расширения

Selenium Webdriver создавался максимально простым и незамысловатым с тем, чтобы поддерживать кроссфлатформенность и кроссбраузерность на эффективном уровне. Это должно означать, что большая часть функционала, которая востребована пользователями, не доступна "из коробки". Такова цена гибкости и простоты использования инструмента.

Взамен **Selenium Webdriver** предлагает легкость и множество способов расширения инструмента или агрегации в другие инструменты. Конечно это стало доступно прежде всего благодаря открытому исходному коду проекта, а также грамотной ООП архитектуре, построенной на интерфейсах и абстрактных классах.

Расширение инструмента подразумевает добавление в него нового, изначально не доступного функционала в зависимости от целей использования этого инструмента. Часто расширению подвергаются основные компоненты: Webdriver, WebElement или By.

Отличным примером расширения **Selenium Webdriver** является HTML Elements framework от yandex, который предложил сразу несколько идей для расширения Page Object паттерна:*

- Идея разбиения страницы на блоки - аннотация `@Block`
- Выделение разных типов элементов (`button` , `checkbox`)
- Возможность аннотировать методы как шаги - аннотация `@Step` для работы в стиле step-based

Официальный сайт инструмента <http://htmlelements.qatools.ru/>

Примеры можно найти на <https://github.com/yandex-qatools/htmlelements-examples>

Термин "обертка" (wrapper) означает то, что над Webdriver сделана надстройка и доступ к Webdriver API опосредован полностью либо частично. В качестве "оберток" для **Selenium Webdriver** на Java наиболее популярные:

- Thucydides
- Geb
- Selenide

Первый инструмент ориентирован прежде всего на автоматизацию приемочного тестирования, а также на красивые отчеты. Thucydides - это report-based фреймворк, позволяющий строить очень подробные html отчеты со скриншотами, шагами и другой информацией о тестах.

Geb использует скриптовый язык Groovy(JVM-based) и ориентирован на быстрое и лаконичное написание тестов.

Selenide похож на Geb, но написан на Java. Этот инструмент, кроме легких и лаконичных тестов, ориентирован также на эффективную работу с AJAX элементами.

Thucydides


Thucydides - это инструмент с открытым исходным кодом, ориентированный на эффективную автоматизацию приемочных тестов, а также на детализированную документацию и отчеты по проекту, построенные на базе этих тестов. Он работает вместе с JUnit и BDD инструментами, такими как JBehave and Cucumber-JVM, и предоставляет обширный API для автоматизированного тестирования в тесной интеграции с [Selenium Webdriver](#).

Thucydides разработан для решения следующих задач:


- Написание более гибких тестов, которые легче поддерживать
- Получение иллюстрированных, исчерпывающих (story-based) отчетов
- Ясная привязка тестов к требованиям
- Измерение покрытия требований

Thucydides workflow


Шаг 1: Определение требований и приемочных критериев

	<p>Thucydides начинается с требований, которые нужно реализовать. Для каждого требования есть приемочные критерии, которые лучше разъясняют требование. Приемочные критерии автоматизирует фукидид.</p>
---	---

Шаг 2: Моделирование требований

	<p>С помощью фукидида вы строите простую модель ваших требований на языке Java. Есть несколько способов моделирования требований, включая обычный Java класс, используя конвенцию структуры директорий или интегрируясь с сторонними инструментами, вроде Jira. Такой подход позволяет разработчику явным образом указать, какое требование тестирует каждый из тестов, а фукидиду - отслеживать тестируемые фичи и требования.</p>
---	---

Шаг 3: Автоматизация приемочного тестирования

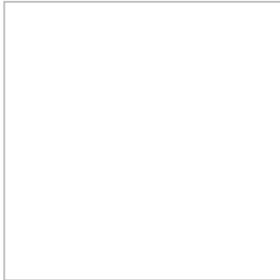
	<p>Далее описываются приемочные критерии языком бизнес-домена, а автоматизаторы имплементируют их с помощью BDD, таких как JBehave или Cucumber-JVM, или с помощью Java и JUnit, так чтобы фукидид мог их запускать, но со статусом pending (тело теста не реализовано).</p>
---	---

Шаг 4: Имплементация тестов



Автоматизаторы теперь могут **имплементировать приемочные критерии** в форме тестов для реального **AUT**. Тесты можно делить на **степы** для лучшей **читабельности** и **легкой поддержки**. Для тестирования веб приложений используется **Selenium Webdriver**.

Шаг 5: Отчет о результатах теста



Thucydides позволяет строить **детализированные отчеты** о результатах запуска тестов, включая:

- **Историю** для каждого теста
- **Скриншот** для каждого степа в тесте
- **Результат** выполнения теста, включая **время** и **сообщения об ошибках**

Шаг 6: Отчет о покрытии требований



Кроме отчетов о выполнении тестов фукидид также предоставляет информацию о:

- Количестве **протестированных** требований
- Количестве **выполненных** требований
- количестве **требований, которые предстоит выполнить**

Шаг 7: Отчет о прогрессе на проекте



Фукидид также предоставляет информацию по истории и прогрессе проекта:

- Изменение количества разработанных фич во времени
- Изменение количества имплементированных и протестированных фич во времени
- Изменение количества упавших тестов во времени

Как видно Thucydides - довольно сложный инструмент, построенный вокруг концепции BDD и приемочного тестирования, использующий Webdriver для тестирования веб приложений.

Maven зависимость:

```
<dependency>  
  <groupId>net.thucydides</groupId>  
  <artifactId>thucydides</artifactId>  
  <version>0.9.273</version>  
</dependency>
```

Более детальную информацию о нем вы найдете на официальном сайте - <http://thucydides.info/>

Geb

Geb - это инструмент для автоматизации браузера, написанный на скриптовом языке Groovy (JVM-based) и использующий [Selenium Webdriver](#) для автоматизации браузера, JQuery селекторы для локации элементов и page object паттерн. В рамках автотестирования он легко интегрируется с различными тестовыми фреймворками, как JUnit, TestNG, Spock.

В сравнении с Webdriver API, geb предоставляет более удобный интерфейс в следующих областях:

- работа с экземпляром Webdriver (создание, настройка, переходы, уничтожение)
- нахождение элементов (JQuery локаторы)
- page object паттерн
- ожидания
- взаимодействия со страницей
- работа с AJAX элементами
- интеграция с build инструментами (maven, gradle, grails)
- интеграция с облачными сервисами (Sauce labs, Browser Stack)

Пример теста на Geb:

```
import geb.Browser

Browser.drive {
    go "http://myapp.com/login"

    assert $("h1").text() == "Please Login"

    $("form.login").with {
        username = "admin"
        password = "password"
        login().click()
    }

    assert $("h1").text() == "Admin Section"
}
```

Geb разрабатывался для удобной и быстрой автоматизации. Полее подробно об этом инструменте можно найти на официальном сайте <http://www.gebish.org/>

Selenide

Selenide - это "обертка" для [Selenium Webdriver](#), ориентированная, как и Geb, на быструю и лаконичную автоматизацию тестирования, но написанная на Java.

Преимущества этого инструмента:

- Более удобный и лаконичный API
- Доступ к Webdriver
- Селекторы в стиле JQuery
- Работа с AJAX элементами
- Автостарт и уничтожение браузера

Пример теста:

```
@Test
public void testLogin() {
    open("/login");
    $(By.name("user.name")).type("johny");
    $("#submit").click();
    $("#username").shouldHave(text("Hello, Johny!"));
}
```

Maven зависимость:

```
<dependency>
  <groupId>com.codeborne</groupId>
  <artifactId>selenide</artifactId>
  <version>2.10</version>
</dependency>
```

Официальный сайт: <http://selenide.org/>

Репортинг

Репортинг - построение читаемого отчета о результатах выполнения тестов. В отчете прежде всего должна присутствовать информация о:

- Количестве запущенных тестов
- По каждому тесту:
 - его название
 - название сьюта
 - результат
 - время выполнения
 - сообщение об ошибке, если тест упал
- Количество успешно выполненных тестов
- Количество упавших тестов
- Время выполнения всех тестов
- Дата и время запуска

Эту информацию должен содержать любой отчет о выполнении тестов. Также отчет может содержать дополнительную информацию, полезную для дебаггинга:

- Полный стек трейс, если тест упал
- Последовательность степов (test story) и их результат
- Встроенные скриншоты на каждый степ и/или при падении теста
- Встроенные логи и/или другие attachments (любые текстовы и медиа файлы, включая html dump страницы)
- Значения переменных при DDT
- Значения переменных тестового окружения (версия приложения, браузер и другое)

Кроме этого отчет может содержать отчеты о покрытии требований и другую визуальную информацию:

- Чарты и графики о результатах выполнения тестов
- Чарты и графики о покрытии требований
- Другая визуальная информация сводного характера

Информация о результатах теста получается и сохраняется после запуска тестов тест раннером. Часто это xml файл. Для тестовых фреймворков XUnit xml файл с результатами имеет один и тот же формат, что позволяет инструментам, строящим визуализированный отчет (чаще всего html), работать с ним единообразно.

Таким образом информация о результатах прохождения тестов собирается и сохраняется при запуске тестов, но визуализироваться может на следующих этапах:

- Сразу после запуска непосредственным тест раннером (например, testng или junit)
- При запуске билд инструментом (surefire plugin для maven)
- При запуске через CI (соответствующие плагины)
- Отдельный инструмент для генерации отчета (вручную)

Кроме того, если набор информации, которую собирает тест раннер, вам кажется неполным, то можно использовать различные test report фреймворки, которые собирает больше информации и строят более красивые и полные отчеты либо создать такой инструмент самому. Как правило, такие инструменты состоят из 3 компонентов:

1. Адаптер для тест раннера (listener). Этот модуль подключается к тест раннеру (непосредственному или встроенному в билд инструмент) и собирает информацию о выполнении тестов независимо от встроенного в тест раннер механизма.
2. Модель - это информация о формате хранения собранной информации о выполнении тестов (структура xml файла, например)
3. Генератор html репорта на основе имеющейся модели по информации, сохраненной в конкретном xml файле.

Примером такого репорт фреймворка может служить ReportNG, который подключается и работает с TestNG тест раннером. Пример его конфигурации с maven:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.7</version>
      <configuration>
        <testSourceDirectory>${basedir}/src/main/java/</testSourceDirectory>
        <testClassesDirectory>${project.build.directory}/classes/</testClassesDir
        <properties>
          <property>
            <name>usedefaultlisteners</name>
            <value>>false</value>
          </property>
          <property>
            <name>listener</name>
            <value>org.uncommons.reportng.HTMLReporter, org.uncommons.reportn
          </property>
        </properties>
        <workingDirectory>${project.basedir}</workingDirectory>
        <argLine>-Dmaven.browser</argLine>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Пример репорта можно увидеть по адресу

<http://reportng.uncommons.org/sample/index.html> Официальный сайт фреймворка

<http://reportng.uncommons.org/>

Минус этого фреймворка заключается в том, что он работает только с TestNG, но не работает с JUnit. Более универсальный фреймворк такого же типа - allure framework разработанный в yandex. Allure framework умеет работать как с TestNG, так и с JUnit. Более того, он может работать с BDD инструментами (Cucumber) и с тест раннерами на других языках (JUnit - C#, pytest - python и другие). Такая универсальность делает его отличным выбором для любого проекта по автоматизации (пожалуй кроме случая с thucydides).

Подключается к maven allure следующим образом:

```
<build>
  plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.16</version>
      <configuration>
        <testSourceDirectory>${basedir}/src/main/java/</testSourceDirectory>
        <testClassesDirectory>${project.build.directory}/classes/</testClassesDir
        <testFailureIgnore>>false</testFailureIgnore>
        <argLine>-javaagent:${settings.localRepository}/org/aspectj/aspectjweaver
        </argLine>
        <properties>
          <property>
            <name>listener</name>
            <value>ru.yandex.qatools.allure.testng.AllureTestListener</value>
          </property>
        </properties>
        <argLine>-Dmaven.browser</argLine>
      </configuration>
      <dependencies>
        <dependency>
          <groupId>org.aspectj</groupId>
          <artifactId>aspectjweaver</artifactId>
          <version>${aspectj.version}</version>
        </dependency>
      </dependencies>
    </plugin>
  </plugins>
</build>
```

Также есть библиотека, предоставляющая дополнительный функционал, призванный сделать репорт еще более читаемым и полным.

```
<dependency>
  <groupId>ru.yandex.qatools.allure</groupId>
  <artifactId>allure-testng-adaptor</artifactId>
  <version>${allure.version}</version>
</dependency>
```

А также модуль для генерации html репорта:

```
<reporting>
  <excludeDefaults>true</excludeDefaults>
  <plugins>
    <plugin>
      <groupId>ru.yandex.qatools.allure</groupId>
      <artifactId>allure-maven-plugin</artifactId>
      <version>LATEST</version>
    </plugin>
  </plugins>
</reporting>
```

При такой конфигурации maven проекта достаточно запустить тесты (`mvn clean test`), чтобы сгенерировался xml файл в отдельной папке с информацией, собранной allure-testNG адаптером. Команда `mvn site` сгенерирует html отчет в папке site. Пример отчета, а также полную документацию и примеры проектом вы найдете на сайте инструмента: <http://allure.qatools.ru/>

Selenium Webdriver. Тестирование клиентской производительности

Клиентская производительность - это скорость работы, отзывчивость веб приложения и его отдельных функциональных элементов на стороне клиента (в браузере).

Производительность на стороне клиента характеризуют такие параметры, как время загрузки страницы, скорость отработки скриптов или время рендеринга элементов. Имея эту информацию в наличии, можно судить о том, где именно производительность приложения низка, что в свою очередь позволяет оптимизировать ее наилучшим образом.

Иногда кастомные контролы, изображения и медиа контент серьезно снижают производительность приложения. Используя [Selenium Webdriver](#) вместе с другими инструментами можно измерять производительность веб приложения и определить узкие места, негативно влияющие на ощущения пользователя при работе с приложением.

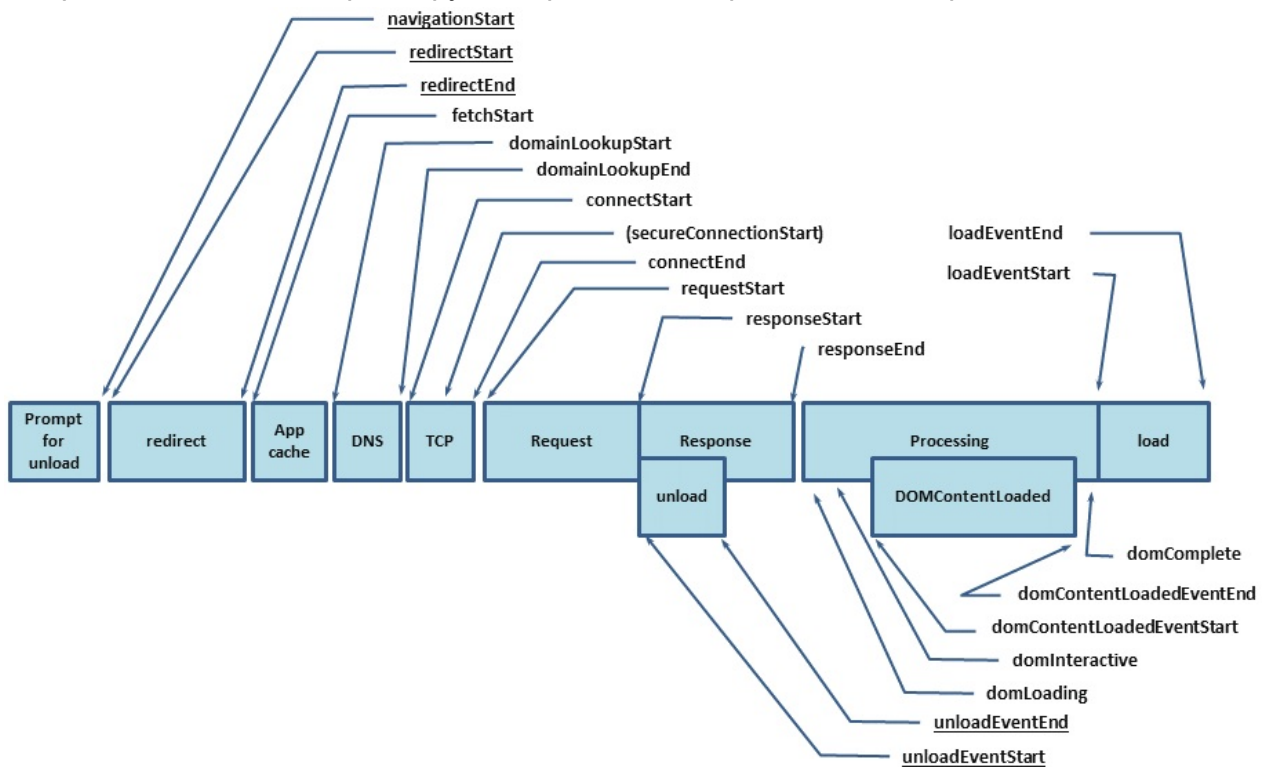
Navigation timing API

Navigation timing - javascript API для измерения производительности веб приложений, утвержденный организацией W3C в качестве стандарта.

Navigation timing предоставляет простой и прямой способ получения точных данных о загрузке страницы (page navigation) и событиях при загрузке страницы (load events). Этот API доступен в IE 9, firefox, chrome и webkit-based браузерах.

Доступ к API можно получить через свойства интерфейса `window.performance.timing` с помощью javascript. Каждый атрибут объекта `performance.timing` хранит время того или иного навигационного события, когда был послан запрос на сервер (request), в миллисекундах в формате UTC (в миллисекундах с первого января 1970 года). Нуль означает, что событие произошло.

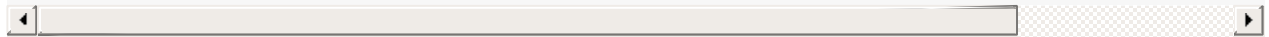
Очередность событий при загрузке страницы изображена на диаграмме:



Более подробно про эти события можно прочитать в самом стандарте Navigation Timing: <http://www.w3.org/TR/navigation-timing/>

Пример:

```
@Test
public void testLogin() {
    Webdriver driver = new FirefoxDriver();
    driver.get(SOME_URL);
    JavascriptExecutor js = (JavascriptExecutor) driver;
    // Получаем время Load Event End (окончание загрузки страницы)
    long loadEventEnd = (Long) js.executeScript("return window.performance.timing.loadEventEnd");
    // Получаем Navigation Event Start (начало перехода)
    long navigationStart = (Long) js.executeScript("return window.performance.timing.navigationStart");
    // Разница между Load Event End и Navigation Event Start - это время загрузки страницы
    System.out.println("Page Load Time is " + (loadEventEnd - navigationStart)/1000 + " s");
}
```



Browser Mob Proxy

Ранее мы уже упоминали этот инструмент, но сейчас нас интересует его возможность собирать данные о производительности. Browser Mob Proxy может собирать данные о производительности и сохранять их в формате HAR.

HAR - это архив, содержащий данные о навигации по world wide web (или [AUT](#)), в том числе и данные о клиентской производительности, в формате JSON определенной структуры.

Сохраняются данные следующим образом:

```
import net.lightbody.bmp.core.har.Har;
import net.lightbody.bmp.proxy.ProxyServer;

import org.junit.Test;
import org.openqa.selenium.Proxy;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.remote.CapabilityType;
import org.openqa.selenium.remote.DesiredCapabilities;

public class SimpleTest {

    @Test
    public void bmpTest() throws Exception {
        // запуск прокси сервера
        ProxyServer server = new ProxyServer(4444);
        server.autoBasicAuthorization("example.com", "username", "password");
        server.start();

        // получение Selenium проку
        Proxy proxy = server.seleniumProxy();

        // конфигурация FirefoxDriver для использования проку
        DesiredCapabilities capabilities = new DesiredCapabilities();
        capabilities.setCapability(CapabilityType.PROXY, proxy);

        WebDriver driver = new FirefoxDriver(capabilities);

        // создание HAR с меткой
        server.newHar("mypage.ru");

        // открытие страницы
        driver.get("http://example.com/index");

        // получение данных HAR
        Har har = server.getHar();

        // здесь будет обработка полученных данных
        // например, сохранение файл
        proxy.getHar().writeTo(new File("D://Test.har"));

        driver.quit();
        server.stop();
    }
}
```

Полученный HAR файл содержит JSON данные, которые можно использовать как текстовую информацию, либо просмотреть в удобном представлении, используя один из существующих инструментов для визуализации (HAR viewer). Например, по адресу <http://www.softwareishard.com/har/viewer/>

Получить информацию можно и без сохранения HAR файла. Java класс

`net.lightbody.bmp.core.har.HarLog` предоставляет набор методов для выборочного получения нужной информации:

```
Har har = proxy.getHar();

// получить информацию о браузере
System.out.println(har.getLog().getBrowser().getName());
System.out.println(har.getLog().getBrowser().getVersion());

// список всех обработанных запросов
for (HarEntry entry : har.getLog().getEntries()) {

    System.out.println(entry.getRequest().getUrl());
    // время ожидания ответа от сервера в миллисекундах
    System.out.println(entry.getTimings().getWait());
    // время чтения ответа от сервера в миллисекундах
    System.out.println(entry.getTimings().getReceive());
}
```

Инструменты для тестирования производительности

Существует множество различных инструментов для тестирования производительности веб приложений. Среди наиболее популярных можно назвать:

- DynaTrace
http://www.compuware.com/en_us/application-performance-management/products/dynatrace-free-trial.html
- HttpWatch
<http://www.httpwatch.com>

DynaTrace

Compuware DynaTrace предоставляет довольно широкий перечень платных услуг для тестирования производительности приложений. Полный список можно найти на официальном сайте, указанном выше.

Для наших целей (автоматизация) нужен инструмент dynaTrace AJAX Edition, который можно использовать при автоматизации в частности вместе с [Selenium Webdriver](#). Загрузить инструмент, а также его подробное описание можно найти по адресу http://www.compuware.com/en_us/application-performance-management/products/ajax-free-edition/overview.html

DynaTrace AJAX Edition работает в браузерах:

- Internet Explorer 8-11
- Firefox 17-30

На платформах:

- Windows Vista
- Windows 7
- Windows 8, 8.1

На ряду с бесплатной версией можно купить premium edition, которая обладает расширенным функционалом.

При установке DynaTrace автоматически установит эддоны для IE и Firefox. После окончания установки следует убедиться, что эддоны включены.

При запуске тестов необходимо:

1. Запустить dynaTrace AJAX Edition перед запуском тестов.

2. Установить глобальные переменные окружения:

```
SET DT_AE_AGENTACTIVE=true
SET DT_AE-AGENTNAME=Firefox
```

Это можно сделать как в IntelliJ Idea (run => edit configurations => Environment variables => "+"), если вы запускаете тесты через IntelliJ Idea, или, используя команды для командной строки, указанные выше, если вы запускаете тесты через командную строку (например, с помощью maven), или же просто установив переменные вручную в windows.

3. Использовать браузер с профилем по умолчанию (default):

```
@Before
public void setUp() throws Exception
{
    // Создаем экземпляр профиля и получаем профиль "по умолчанию"
    ProfilesIni profile = new ProfilesIni();
    FirefoxProfile ffprofile = profile.getProfile("default");
    // Создаем драйвер, передав ему профиль "по умолчанию"
    driver = new FirefoxDriver(ffprofile);
}
```

В результате запуска тестов в окне dynaTrace AJAX Edition под опцией Session вы увидите данные о производительности, собранные в ходе теста, а также рекомендации по оптимизации производительности приложения.

Start Page Performance Report

URL	Rank	First Impressio...	onLoad Tim...	Fully Loaded [ms]	On Server [ms]	On Client [ms]	Ø Interactive...	Remarks
http://news.yahoo.com/	C(71)	112	4175	8757	8120	1483	0	50 Requests, 2 Redirects, 28 uncached, 116ms in Wai...
http://news.yahoo.com/us	B(88)	50	2615	4118	3321	997	0	1 Redirects, 15 uncached, 108ms in Wait, 4 single res...
http://news.yahoo.com/business	C(79)	2625	2937	5107	7216	1007	0	16 uncached, 91ms in Wait, 5 single resource domai...
http://news.yahoo.com/world	B(89)	83	2488	3886	5921	1099	0	1 Redirects, 14 uncached, 155ms in Wait, 4 single res...
http://news.yahoo.com/entertainment	B(87)	1957	2170	3603	5963	1083	0	13 uncached, 131ms in Wait, 4 single resource doma...
http://news.yahoo.com/sports	B(81)	2129	2579	4075	5156	1291	0	47 Requests, 1 Redirects, 23 uncached, 138ms in Wai...
http://news.yahoo.com/technology	B(88)	1263	2276	4341	4525	1085	0	15 uncached, 100ms in Wait, 5 single resource doma...

Summary (C) Caching (D) Network (A) Server-Side (E) JavaScript/AJAX (C) Timeline KPI's

C(71) is the calculated JavaScript/AJAX Performance Rank
 Optimize the JavaScript handlers and AJAX/XHR calls to improve end-user experience for this page.
 Read more on [Best Practices on JavaScript/AJAX Performance Optimization](#) on the dynaTrace Community Portal

See how your rank compares to Alexa 1000 sites

A B C D E F
 You are here: C(71)

There are 15 JS handlers and jQuery-like calls taking longer than 20ms coming from 6 JS files. 7 script blocks impact the rank
 Reducing execution time and invocation count of these handlers [Selectors](#) can improve end-user experience.
 Click through the identified handlers ,methods and identify who called them (Back Traces) and what they call (Forward Traces).

Contributor	Invocations	Total Sum [ms]
<script>	1	731.45
load event on <document>	1	313.59
getElementsByClassName("yn-menu", undefined, und...	1	158.06
getElementsByClassName("dynamic_ad", undefined, ...	1	157.88
getElementsByClassName("filter-controls", undefined,...	1	148.01
getElementsByClassName("darla_ad", undefined, und...	1	146.76
getElementsByClassName("yn-menu-blog", undefine...	1	144.56
load event on <document>	1	140.52
getElementsByClassName("highlight-viewer", undefi...	1	140.47
load event on <document>	1	60.78
unload event on <document>	1	57.34
load event on <document>	1	48.52
getElementsByClassName("showtt", "A", "bd", undefi...	1	30.00
getElementsByClassName("toggle-control", "a", <div>...	1	22.86
load event on <document>	1	20.66

Back Traces	Invocations	Exec [ms]
<anonymous>	1	16.11
<script>	1	16.11

```
function ()
{
    var e = new YAHOO.News.Menu({
        activeClass: "menu-active",
        menuToolClass: "yn-menu",
        addMenuMasks: "div"
    });
    var d = YAHOO.util.Dom.getElementsByClassName("yn-menu");
    for(var c = 0; c < d.length; c++)
```

HttpWatch

Еще одним популярным инструментом для тестирования производительности является HttpWatch. Узнать о инструменте или скачать последнюю версию можно по адресу <http://www.httpwatch.com/>

Как и DynaTrace, HttpWatch собирает различные метрики клиентской производительности для анализа проблем с производительностью javascript, выполнения запросов, передачи ресурсов и загрузки страницы.

HttpWatch работает в браузерах:

- Internet Explorer 6-11
- Firefox 25-32

На платформах:

- Windows XP
- Windows Vista
- Windows 7
- Windows 8, 8.1

Заявленная поддержка языков программирования:

- C#
- Ruby
- Javascript

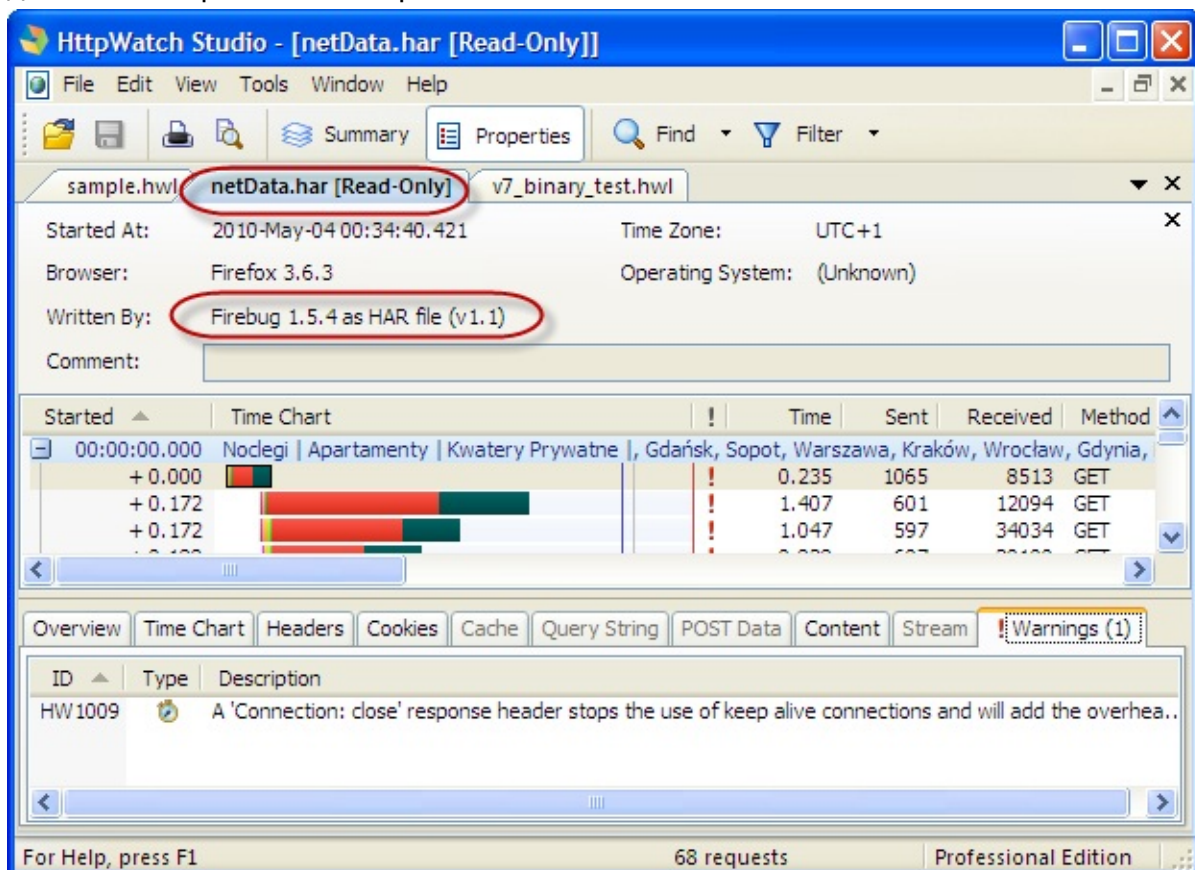
Для скачивания доступны платная и бесплатная версии. Различие между ними доступно по адресу <http://www.httpwatch.com/editions.htm>. При установке HttpWatch установит плагины для браузеров, которые нужно настроить для автоматических тестов:

На вкладке **reporting** установите радиокнопку **Start Recording when HttpWatch is opened** и отметьте чекбокс **Stop recording when HttpWatch is closed**. Изменения сохраните.

Пример кода на C# (HttpWatch.dll находится в директории установки инструмента):

```
// Создаем контроллер HttpWatch
Controller control = new Controller();
//Создаем экземпляр IE драйвера
IWebDriver driver = new InternetExplorerDriver();
// Создаем уникальный титул стартовой страницы, чтобы HttpWatch мог подключиться
string uniqueTitle = Guid.NewGuid().ToString();
IJavaScriptExecutor js = driver as IJavaScriptExecutor;
js.ExecuteScript("document.title = '" + uniqueTitle + "';");
// Подключаем HttpWatch к созданному экземпляру IE драйвера
Plugin plugin = control.AttachByTitle(uniqueTitle);
// Открываем окно HTTPWatch
plugin.OpenWindow(false);
// Переходим по url
driver.Navigate().GoToUrl("YourUrlHere");
// Выполняем действия
driver.FindElement(...).SendKeys(...);
driver.FindElement(...).SendKeys(...);
driver.FindElement(...).Click();
// Сохраняем данные, собранные HttpWatch, в файл
plugin.Log.Save("C:\\report.html");
// Закрываем браузер
driver.Close();
```

Получив данные и сохранив их в файле, вы можете просмотреть их с помощью HttpWatch Studio, которая устанавливается вместе с HttpWatch. Репорт содержит данные и метрики как на скриншоте:



Стоит также отметить тот факт, что HttpWatch репорты можн экспортировать в другие форматы, в том числе и HAR.

Selenium Webdriver. Тестирование на мобильных браузерах

Обзор инструментов

В последнее время начало набирать обороты такое направление, как мобильная автоматизация, которая включает в себя и атоматизацию мобильных веб-приложений. Как и любое новое напраление, оно порадило вместе с собой множество инструментов для решения его насучных задач. Вот сейчас мы и рассмотрим некоторые из них: Ranorex, Monkey Talk и Appium. Два первых мы рассмотрим в чисто познавательном плане, а на последнем остановимся подробнее, поскольку он очень активно набирает популярность.

Ranorex

Ranorex Automation Tools — это полноценная среда разработки, а также набор инструментов и библиотек для написания тестов. Она позволяет автоматизировать следующие виды приложений:

- Desktop
- Web
- Mobile (в том числе Mobile web)

Нас в данном случае интересует последний тип. На официмальном сайте есть примео иллюстрирующий работу одновременно с Web и Web Mobile. Данная среда предоставляет следующие возможности:

1. Поддержка динамически генерируемых графических элементов управления (контролов)
2. Настраиаемая система поиска контролов
3. Простая поддержка тестов, основанных на данных (Data Driven Testing)
4. Возможность разрабатывать свои модули (фреймворки) и использовать их при разработке тестов на C#
5. Поддержка запуска тестов на сервере Continuous Integration (TeamCity)
6. Генерация информативных отчетов по результату прогона тестов
7. Возможность интеграции тестов с тест-кейсами системы тест-менджмента (TMS)
8. Простота изучения и использования тестирующими

Monkey Talk

Monkey Talk - это инструмент для мобильного тестирования, который служит для написания тестов под Android и iOS. В отличие от выше описанного, этот инструмент предназначен только для мобильного тестирования. Monkey Talk довольно прост в освоении благодаря подробным гайдам с пояснениями и скриншотами. Благодаря собственной IDE, с возможностью Record\Play решений, легко осваивается ручными тестировщиками.

Плюсы:

- Распространяется бесплатно
- Возможность создание тестов под 2 платформы (iOS & Android)
- Использование полноценного языка высокого уровня (Java API)

Минусы:

- Необходимость исходников тестируемого приложения
- Нельзя использовать привычные локаторы, такие как CSS и Xpath (использует собственные)

Установка и настройка Appium.

Принципы и основы работы с инструментом

Appium – инструмент автоматизации мобильных приложений, использующих Webdriver API. Appium – HTTP сервер, который создает и управляет сессиями Webdriver.

Если мы хотим использовать Appium на Windows, то нам нужно воспользоваться Appium.exe.

Установка:

1. С помощью [Appium.exe](#).
2. Используя NPM:
 - Устанавливаем NPM: C:\node> npm install express -g (C:\Node - место установки node.js)
 - Запускаем NPM и выполняем команду: \$ npm install appium

Настройка:

1. Первым делом мы устанавливаем [Node.js](#) (выше 0.10 версии)
2. Затем устанавливаем [Android SDK](#). С поддержкой API Level 17 или выше. Создаем переменную окружения ANDROID_HOME, куда добавляем пути к папкам tools и platform-tools.
3. Устанавливаем Java JDK и прописываем к нему пути в переменной JAVA_HOME.
4. Далее нам нужно установить [Apache Ant](#). Так же добавляем путь к его папке в переменную окружения - PATH.
5. После чего устанавливаем [Apache Maven](#). Создаем две переменные M2HOME и M2, куда соответственно вписываем пути к папке Maven и паке bin, внутри неё.

Запуск тестов на десктоп и мобильных браузерах

Для запуска тестов с помощью Appium нужно в начале настроить драйвер, а для этого нужно ему выставить правильные свойства.

Для запуска тестов под Android свойства драйвера будут следующие:

```
public class AndroidTest {

    private WebDriver driver;

    @Before
    public void setUp() throws Exception {
        File classpathRoot = new File(System.getProperty("user.dir"));
        File appDir = new File(classpathRoot, "../../apps/ApiDemos/bin");
        File app = new File(appDir, "ApiDemos-debug.apk");
        DesiredCapabilities capabilities = new DesiredCapabilities();
        capabilities.setCapability("device", "Android");
        capabilities.setCapability(CapabilityType.BROWSER_NAME, "");
        capabilities.setCapability(CapabilityType.VERSION, "4.2");
        capabilities.setCapability(CapabilityType.PLATFORM, "MAC");
        capabilities.setCapability("app", app.getAbsolutePath());
        capabilities.setCapability("app-package", "com.example.android.apis");
        capabilities.setCapability("app-activity", ".ApiDemos");
        driver = new SwipeableWebDriver(new URL("http://127.0.0.1:4723/wd/hub"), capabilities);
    }

    //Other Test methods...
}
```

Если же мы хотим тесты на мобильнос Safari, то:

```
@Before
public void setUp() throws Exception {
    DesiredCapabilities capabilities = new DesiredCapabilities();
    capabilities.setCapability("device", "iPhone Simulator");
    capabilities.setCapability("version", "6.1");
    capabilities.setCapability("app", "safari");
    driver = new RemoteWebDriver(new URL("http://127.0.0.1:4723/wd/hub"),
        capabilities);
    driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS);
}
```

Selenium Webdriver. Behavior-Driven Development.

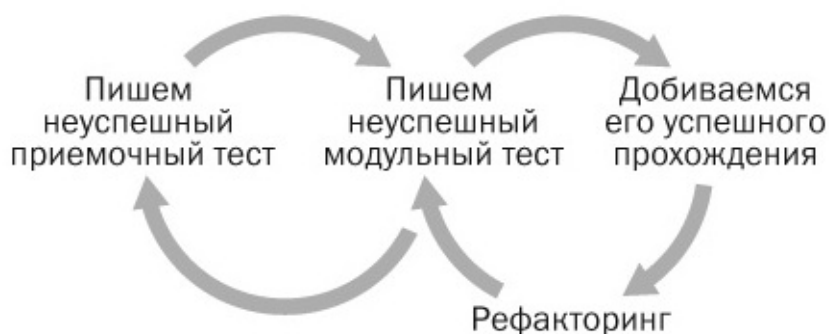
Обзор методологии и инструментов на Java.

В эпоху «сначала тестирование» поведение пользователя выражалось в основном через модульные тесты, написанные на языке системы, т. е. на языке, непонятном пользователю. С появлением методик разработки на основе поведений (Behavior-Driven Development, BDD) эта динамика меняется. Используя методики BDD, вы можете создавать автоматизированные тесты на языке бизнеса, в то же время сохраняя связь с реализуемой системой.

Конечно, был создан целый ряд инструментов, помогающих реализовать BDD в процессе разработки. К ним относятся Cucumber-JVM в Java, SpecFlow для Microsoft .NET Framework и еще много для этих и других языков. Инструменты реализации BDD подхода упрощают написание и выполнение спецификаций в средах разработки, а использование [Selenium WebDriver](#) позволяет управлять браузером для комплексного автоматизированного тестирования системы.

Многие считают BDD надмножеством TDD, но не его заменой. Ключевая разница — фокусировка на начальном проекте и создании тестов. Но основное внимание уделяется не тестам модулей или объектов, как в TDD, а целям пользователей и пошаговым операциям, предпринимаемым ими для достижения этих целей. Поскольку мы больше не начинаем с тестов малых модулей, меньше проявляется склонность размышлять над деталями проекта или использовать более мелкие части. Вместо этого документируются исполняемые спецификации, которые контролируют мою систему. В результате по-прежнему получаются модульные тесты, но в BDD поощряется подход «от внешнего к внутреннему» (outside-in approach), который начинается с полного описания подлежащей реализации функции.

Цикл разработки на основе поведений (BDD):



Cucumber JVM + Selenium Webdriver.

Cucumber JVM - это один из популярных инструментов реализации Behavior Driven Development (BDD) подхода в Java. Этот инструмент позволяет создавать тесты любому участнику проектной команды. Для этого используют язык Gherkin, в котором основными являются следующие слова: Given, When и Then. Тесты, созданные таким образом, хранятся в файлах с расширением ".feature".

Зависимости для Maven проекта:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>FundTransfer</groupId>
    <artifactId>FundTransfer</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <dependencies>
        <dependency>
            <groupId>info.cukes</groupId>
            <artifactId>cucumber-java</artifactId>
            <version>1.0.14</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>info.cukes</groupId>
            <artifactId>cucumber-junit</artifactId>
            <version>1.0.14</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.10</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.seleniumhq.selenium</groupId>
            <artifactId>selenium-java</artifactId>
            <version>2.25.0</version>
        </dependency>
    </dependencies>
</project>
```

Сценарий написанный с помощью языка Gherkin:

```

Feature: Customer Transfer's Fund
  As a customer,
  I want to transfer funds
  so that I can send money to my friends and family
Scenario: Valid Payee
  Given the user is on Fund Transfer Page
  When he enters "Jim" as payee name
  And he enters "100" as amount
  And he Submits request for Fund Transfer
  Then ensure the fund transfer is complete with "$100
  transferred successfully to Jim!!" message
Scenario: Invalid Payee
  Given the user is on Fund Transfer Page
  When he enters "Jack" as payee name
  And he enters "100" as amount
  And he Submits request for Fund Transfer
  Then ensure a transaction failure message "Transfer
  failed!! 'Jack' is not registered in your List of Payees"
  is displayed
Scenario: Account is overdrawn past the overdraft limit
  Given the user is on Fund Transfer Page
  When he enters "Tim" as payee name
  And he enters "1000000" as amount
  And he Submits request for Fund Transfer
  Then ensure a transaction failure message "Transfer
  failed!! account cannot be overdrawn" is displayed

```

Создание java класса:

```

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.By;
import cucumber.annotation.*;
import cucumber.annotation.en.*;
import static org.junit.Assert.assertEquals;

public class FundTransferStepDefs {
    protected WebDriver driver;

    @Before
    public void setUp() {
        driver = new ChromeDriver();
    }

    @Given("the user is on Fund Transfer Page")
    public void The_user_is_on_fund_transfer_page() {
        driver.get("http://dl.dropbox.com/u/55228056/fundTransfer.html");
    }

    @When("he enters \"([^\"]*)\" as payee name")

```

```

    public void He_enters_payee_name(String payeeName) {
        driver.findElement(By.id("payee")).sendKeys(payeeName);
    }

    @And("he enters \"([^\"]*)\" as amount")
    public void He_enters_amount(String amount) {
        driver.findElement(By.id("amount")).sendKeys(amount);
    }

    @And("he Submits request for Fund Transfer")
    public void He_submits_request_for_fund_transfer() {
        driver.findElement(By.id("transfer")).click();
    }

    @Then("ensure the fund transfer is complete with \"([^\"]*)\" message")
    public void Ensure_the_fund_transfer_is_complete(String msg) {
        WebElement message = driver.findElement(By.id("message"));
        assertEquals(message.getText(), msg);
    }

    @Then("ensure a transaction failure message \"([^\"]*)\" is displayed")
    public void Ensure_a_transaction_failure_message(String msg) {
        WebElement message = driver.findElement(By.id("message"));
        assertEquals(message.getText(), msg);
    }

    @After
    public void tearDown() {
        driver.close();
    }
}

```

И для изменения настроек Cucumber-JVM мы добавим класс конфигурации:

```

package fundtransfer.test;

import cucumber.junit.Cucumber;
import org.junit.runner.RunWith;

@RunWith(Cucumber.class)
@Cucumber.Options(format = {"pretty", "html:target/cucumber-htmlreport",
    "json-pretty:target/cucumber-report.json"})
public class RunCukesTest {

}

```


JBehave + Selenium Webdriver.

JBehave – это еще один из инструментов пользующихся популярностью, когда речь заходить о BDD подходе на проекте. Он похож на Cucumber-JVM, так же использует язык Gherkin для написания тестовых сценариев. Только расширение для файлов здесь уже ".story". Вот пример такого файла:

```
Narrative: I should be able to Calculate my Body Mass Index

Scenario: I should see my BMI after entering Height and Weight

When I open BMI Calculator Home Page
When I enter height as '181'
When I enter weight as '80'
When I click on the Calculate button
Then I should see bmi as '24.4' and category as 'Normal'
```

Следующим шагом, после создания тестового сценария, является написание java класса, котрый может выглядеть следующим образом:

```

import junit.framework.Assert;
import org.jbehave.core.annotations.Then;
import org.jbehave.core.annotations.When;

import org.openqa.selenium.By;
import org.openqa.selenium.WebElement;

public class Bmi extends StoryBase {

    @When("I open BMI Calculator Home Page")
    public void IOpen(){
        driver.get("http://dl.dropbox.com/u/55228056/bmicalculator.html");
    }

    @When("I enter height as '$height'")
    public void IEnterHeight(String height){
        WebElement heightCMS = driver.findElement(By.id("heightCMS"));
        heightCMS.sendKeys(height);
    }

    @When("I enter weight as '$weight'")
    public void IEnterWeight(String weight){
        WebElement weightKg = driver.findElement(By.id("weightKg"));
        weightKg.sendKeys(weight);
    }

    @When("I click on the Calculate button")
    public void IClickOnTheButton(){
        WebElement button = driver.findElement(By.id("Calculate"));
        button.click();
    }

    @Then("I should see bmi as '$bmi_exp' and category as '$bmi_category_exp'")
    public void IShouldBmiAndCategory(String bmi_exp, String bmi_category_exp){
        WebElement bmi = driver.findElement(By.id("bmi"));
        Assert.assertEquals(bmi_exp, bmi.getAttribute("value"));
        WebElement bmi_category = driver.findElement(By.id("bmi_category"));
        Assert.assertEquals(bmi_category_exp, bmi_category.getAttribute("value"));
        driver.quit();
    }
}

```

После чего мы должны создать конфигурационный файл для корректного взаимодействия [Selenium](#) и JBehave:

```
import java.util.List;
import org.jbehave.core.configuration.Configuration;
import org.jbehave.core.configuration.MostUsefulConfiguration;
import org.jbehave.core.io.LoadFromClasspath;
import org.jbehave.core.junit.JUnitStory;
import org.jbehave.core.reporters.Format;
import org.jbehave.core.reporters.StoryReporterBuilder;
import org.jbehave.core.steps.InstanceStepsFactory;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;

public abstract class StoryBase extends JUnitStory {

    protected final static WebDriver driver = new FirefoxDriver();

    @Override
    public Configuration configuration() {
        return new MostUsefulConfiguration()
            .useStoryLoader(new LoadFromClasspath(this.getClass().getClassLoader()))
            .useStoryReporterBuilder(new StoryReporterBuilder()
                .withDefaultFormats()
                .withFormats(Format.HTML, Format.CONSOLE)
                .withRelativeDirectory("jbehave-report"))
    }

    @Override
    public List candidateSteps() {
        return new InstanceStepsFactory(configuration(), this).createCandidateSteps();
    }
}
```

Алфавитный указатель

AUT

Application under Test - тестируемое приложение

[7.2. Thucydides](#) [8.2. Browser Mob Proxy](#)
[1.2. Основные методы Selenium Webdriver API](#) [1.4. Ожидания](#)

Selenium

набор инструментов, предназначенных для автоматизации веб браузеров на различных платформах. Selenium может автоматизировать множество различных браузеров на разных платформах, используя различные языки программирования и интегрируясь с различными тестовыми фреймворками.

[10.3. JBehave + Selenium Webdriver.](#) [5.2. Basic Authentification Window](#)
[5.5. Логгирование в Selenium Webdriver](#) [3.2. Альтернативные Page Object подходы.](#)
[3.1. Использование паттерна Page Object.](#) [3.3. Вспомогательные инструмаенты.](#)
[4. Selenium Grid и "headless" браузеры](#) [4.3. Grid. Настройка и использование.](#)
[0. Introduction](#) [6.1. Автоматизация Canvas элементов.](#)
[7.1. Selenium "обертки" и расширения](#)
[2.3. Контроль за ходом теста. Кастомные ожидания, попапы, алерты, Iframes.](#)
[1.1. WebDriver. Обзор и принцип работы](#)

Selenium Webdriver

инструмент для автоматизации реального браузера, как локально, так и удаленно, наиболее близко имитирующий действия пользователя. Selenium Webdriver входит в состав Selenium 2.0 вместе с Selenium RC.

[10. Selenium Webdriver. Behavior-Driven Development.](#)
[10.2. Cucumber JVM + Selenium Webdriver.](#) [10.3. JBehave + Selenium Webdriver.](#)
[10.1. Обзор методологии и инструментов на Java.](#)
[5.5. Логгирование в Selenium Webdriver](#) [3.3. Вспомогательные инструмаенты.](#)
[5. Selenium Webdriver. Проблемные моменты](#)
[6. Selenium Webdriver. Тестирование HTML5 веб приложений](#) [7.3. Geb](#)

- 7.4. Selenide 7.1. Selenium "обертки" и расширения
- 7. Selenium Webdriver. Расширение инструмента 7.2. Thucydides
- 9. Selenium Webdriver. Тестирование на мобильных браузерах 8.3. DynaTrace
- 8. Selenium Webdriver. Тестирование клиентской производительности
- 5.1. Вспомогательные инструменты 2. Selenium WebDriver. Сложные вопросы.
- 2.3. Контроль за ходом теста. Кастомные ожидания, попапы, алерты, Iframes.
- 2.1. Локаторы. CSS, XPATH, JQUERY.
- 2.2. WebDriver API. Сложные взаимодействия. 1. Selenium Webdriver. Введение
- 1.2. Основные методы Selenium Webdriver API
- 1.1. WebDriver. Обзор и принцип работы