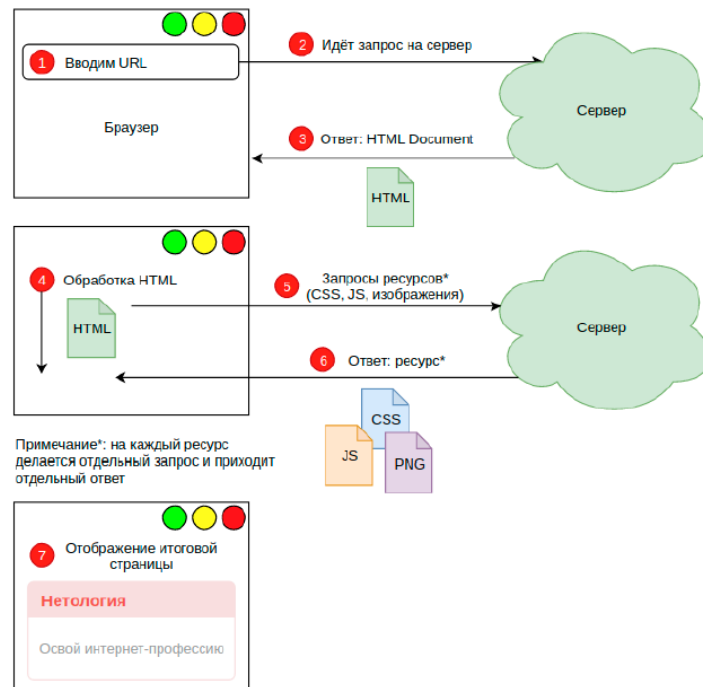


WEB INTERFACE TESTING

В рамках ручного тестирования мы используем веб-браузеры для воспроизведения вариантов взаимодействия пользователя.

В рамках автоматизированного тестирования нам нужно понимание общего механизма работы системы в целом, и инструменты, позволяющие автоматизировать необходимые операции.

ОБЩИЙ МЕХАНИЗМ РАБОТЫ



Что же мы можем автоматизировать?

А: Мы можем автоматизировать веб-браузер, мы можем автоматизировать взаимодействие на уровне HTTP.

Но в рамках нашего курса мы будем автоматизировать именно браузер.

При этом нам нужно иметь представление о ключевых технологиях, используемых при создании веб-интерфейсов.

JS — язык, использующий API браузера для добавления поведения к элементам или переопределения их поведения по умолчанию

Используя API браузера для добавления поведения к элементам или переопределения их поведения по умолчанию

Q: Что значит поведение по умолчанию?

А: У некоторых элементов есть поведение по умолчанию: например, клик по ссылке или отправка формы приводит к загрузке новой страницы и повторению всего процесса (который был на картинке). JS же позволяет сделать так, чтобы новая страница не загружалась, а данные отправлялись в «фоновом режиме» или не отправлялись вовсе

Итак, нам нужен инструмент, который бы позволил управлять веб-

браузером. При этом, не стоит забывать про ключевые особенности работы в веб-среде:

1. Много браузеров разных версий под разные ОС.

2. У пользователя могут быть различные характеристики системы: разрешение монитора, установленные расширения для браузера (например, скайп), медленное подключение и т.д.

3. Противодействие роботам — популярные сервисы всячески противодействуют роботизированному

использованию*, при этом, чаще всего, предоставляя отдельный бесплатно-платный API.

Крайне важно наличие разрешения на проведение тестирования! Никогда

не тестируйте никакой публичный сервис (тем более с использованием инструментов автоматизации),

если у вас нет письменного разрешения владельца, либо сервис специально не предназначен для отработки навыков автоматизированного тестирования.

Selenium — инструмент автоматизации браузеров. Он позволяет нам автоматизированно управлять

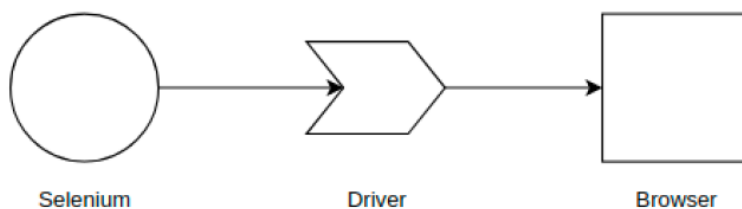
браузерами, а с какой целью (автоматизация тестирования или что-то ещё) — это уже целиком наше дело.

Кроме того, получается, что организация всей остальной инфраструктуры остаётся целиком в нашей сфере ответственности:

- инструмент сборки;
- организация тестов (фреймворк тестирования);
- assertions library.

Selenium предоставляет API, позволяющее взаимодействовать с браузерами из различных языков программирования. Конечно же, Java находится в числе таких языков.

Selenium использует следующую схему для управления браузерами:



Т.е. для выстраивания полной цепочки управления нам необходимы:

1. Сам Selenium;
2. Драйвер браузера;
3. Установленный браузер.

ВИДЖЕТ ЗАЯВКИ

Перед нами поставили задачу протестировать виджет заявки, который выглядит следующим образом:

Заказать обратный звонок

Как вас зовут

Номер вашего телефона

☐ Я соглашаюсь с условиями обработки моих персональных данных

Отправить

При успешной отправке:

✓ Ваша заявка успешно отправлена!

При неуспешной:

✗ При отправке заявки произошла ошибка!

BUILD.GRADLE

```
1  plugins {
2      id 'java'
3  }
4
5  group 'ru.netology'
6  version '1.0-SNAPSHOT'
7
8  sourceCompatibility = 1.8
9
10 repositories {
11     mavenCentral()
12 }
13
14 dependencies {
15     testImplementation 'org.junit.jupiter:junit-jupiter:5.5.1'
16     testImplementation 'org.seleniumhq.selenium:selenium-java:3.141.59'
17     testImplementation 'org.seleniumhq.selenium:selenium-chrome-driver:3.141.59'
18 }
19
20 test {
21     useJUnitPlatform()
22 }
```

через

```
System.setProperty("webdriver.chrome.driver", "/path/to/chromedriver");
```

Дальше возникает самый главный вопрос: а что мы собираемся тестировать?

Итак нам нужно:

1. Создать проект на базе Gradle;
2. Подключить JUnit;
3. Подключить Selenium;
4. Скачать chromedriver (и иметь установленный браузер Chrome);
5. Написать авто-тест.

Мы будем показывать все примеры на Chrome, в домашних работах вы также будете использовать и другие браузеры

Q: Зачем нам selenium-chrome-driver?

A: Если этой зависимости не будет, мы не сможем подключить chromedriver к нашему проекту*. Скачанный файл необходимо распаковать и положить в каталог, находящийся в списке путей переменной окружения PATH либо установить

ЗАГОТОВКА ТЕСТА

```
1 class CallbackTest {
2     private WebDriver driver;
3
4     @BeforeEach
5     void setUp() {
6         driver = new ChromeDriver();
7     }
8
9     @AfterEach
10    void tearDown() {
11        driver.quit();
12        driver = null;
13    }
14
15    @Test
16    void shouldTestSomething() {
17        throw new UnsupportedOperationException();
18    }
19 }
```

CHROMEDRIVER

```
class CallbackTest {
    private WebDriver driver;

    @BeforeAll
    static void setUpAll() {
        // убедитесь, что файл chromedriver.exe расположен именно в каталоге C:\tmp
        System.setProperty("webdriver.chrome.driver", "C:\\tmp\\chromedriver.exe");
    }

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

    @AfterEach
    void tearDown() {
        driver.quit();
        driver = null;
    }

    @Test
    void shouldTestSomething() {
        throw new UnsupportedOperationException();
    }
}
```

Время разработки тест-кейсов!

Постановка задачи, прямо скажем, ужасная, потому что никаких критериев того, когда должна выводиться страница «Успешно» или «Ошибка» нет никаких и без уточнения алгоритма работы мы не сможем написать нормальный тест.

Важно понимать, что, в отличие от Unit-тестов, у вас может и не быть доступа к исходникам сервиса, чтобы «подсмотреть» логику там. Да и даже если они есть — это плохой подход, основываться не на требованиях, а на реализации. При уточнении нам пришла следующая информация: при заполнении всех полей (не важно, какой информацией) должна появляться страница успеха. Если же пользователь заполнил не все поля или на сервере что-то пошло не так, то выходит страница с ошибкой.

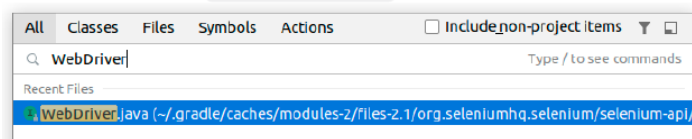
КЛЮЧЕВЫЕ ОПЕРАЦИИ

1. Загрузка страниц;
2. Поиск элементов на странице;
3. Взаимодействие с элементами

WEBDRIVER

```
public interface WebDriver extends SearchContext {  
    void get(String url);  
    String getCurrentUrl();  
    String getTitle();  
    List<WebElement> findElements(By by);  
    WebElement findElement(By by);  
    String getPageSource();  
    void close();  
    void quit();  
    ...  
}
```

Вы можете целиком посмотреть исходники нажав в IDEA два раза Shift и забив в поиск `WebDriver`:



Мы крайне рекомендуем читать исходники и комментарии в формате JavaDoc к ним, поскольку это первоисточник информации (в отличие от того, что вы можете найти в Интернете).

WEBELEMENT

```
public interface WebElement extends SearchContext, TakesScreenshot {  
    void click();  
    void submit();  
    void sendKeys(CharSequence... keysToSend);  
    void clear();  
    String getTagName();  
    String getAttribute(String name);  
    boolean isSelected();  
    boolean isEnabled();  
    String getText();  
    List<WebElement> findElements(By by);  
    WebElement findElement(By by);  
    boolean isDisplayed();  
    Point getLocation();  
    Dimension getSize();  
    Rectangle getRect();  
    String.getCssValue(String propertyName);  
}
```

заполнять поля ввода, отправлять форму и т.д.).

Selenium предлагает нам два ключевых интерфейса:

1. `WebDriver` — непосредственная работа с браузером, открытие страниц, вкладок, поиск элементов по всей странице (именно этот интерфейс имплементирует `ChromeDriver`);

2. `WebElement` — работа с HTML элементами.

Итак, мы можем управлять браузером, загружая страницы по определённому URL-адресу.

При загрузке страницы браузер обрабатывает HTML-код, создавая из HTML элементов объекты.

Мы можем искать эти объекты на странице и взаимодействовать с ними (например,

JAR

Мы будем предоставлять вам сервисы для тестирования в виде jar-архивов, для запуска которых нужно воспользоваться командой:

```
java -jar <имя_файла>.jar
```

При этом у вас должна быть установлена Java.

Все сервисы будут запускаться на порту 9999, т.е. в качестве URL'a для тестирования нужно будет использовать `http://localhost:9999`.

Чтобы что-то успеть увидеть, вы можете добавить вызов

ОТКРЫТИЕ СТРАНИЦЫ

Запустим наш сервис, после чего запустим авто-тест (который пока умеет только открывать страницу):

```
@Test
void shouldTestSomething() {
    driver.get("http://localhost:9999");
}
```

`Thread.sleep(5000)` в ваш тест или запуститься в режиме отладки.

Для того, чтобы пойти дальше, нам нужно разобрать три вещи:

НЕСОВМЕСТИМОСТЬ ВЕРСИЙ

В процессе запуска вы вполне можете получить следующее сообщение:

```
Starting ChromeDriver 77.0.3865.40 on port 3833
Only local connections are allowed.
Please protect ports used by ChromeDriver and related test frameworks to prevent access by malicious code.

session not created: This version of ChromeDriver only supports Chrome version 77
...
```

Это значит, вам нужно либо обновить Chrome до нужной версии, либо скачать более старую версию драйвера.

1. Как строятся HTML-элементы;

2. Как их обрабатывает браузер;

3. Как можно найти элементы на странице

СИНТАКСИС HTML-ЭЛЕМЕНТОВ

Перечень элементов определяется спецификацией [HTML](#), 4-ый раздел:

4.1.1. The `html` element

Categories:

None.

Contexts in which this element can be used:

As the document's [document element](#).

Wherever a subdocument fragment is allowed in a compound document.

Content model:

A `<head>` element followed by a `<body>` element.

Tag omission in text/html:

An `<html>` element's [start tag](#) can be omitted if the first thing inside the `<html>` element is not a [comment](#).

An `<html>` element's [end tag](#) can be omitted if the `<html>` element is not immediately followed by a [comment](#).

Content attributes:

[Global attributes](#).

`manifest` — Application cache manifest.

КЛЮЧЕВЫЕ
БЛОКИ
ОПИСАНИЯ

— Categories —
к каким
категориям
элемент
относится
(нужно для
следующих
двух блоков);

Отдельно выделяют два ключевых атрибута — `id` (уникальный идентификатор) и `class` (логическая группировка ряда элементов в единый класс, чаще всего с целью визуального оформления).

Атрибут `class` может содержать несколько значений, разделённых пробелом, например:

```
<button
  role="button"
  type="button"
  class="button button_view_extra button_size_m button_theme_alfa-on-white">
  <span class="button__content">
    <span class="button__text">Отправить</span>
  </span>
</button>
```

— Context —
внутри каких
элементов
может быть
вложен;

— Content
model — какие
элементы
может
содержать
внутри себя;

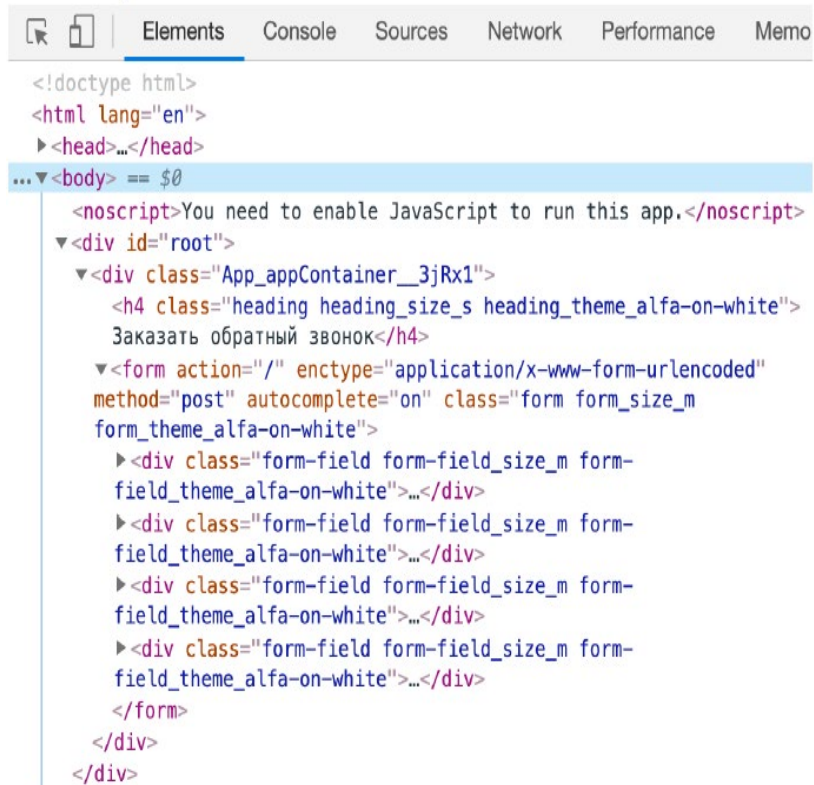
Т.е. у элемента `button` 4 класса: `button`, `button_view_extra`, `button_size_m`, `button_theme_alfa-on-white`.

— Tag omission — могут ли отсутствовать открывающий/закрывающий теги;

— Content attributes — какие атрибуты могут быть у элемента

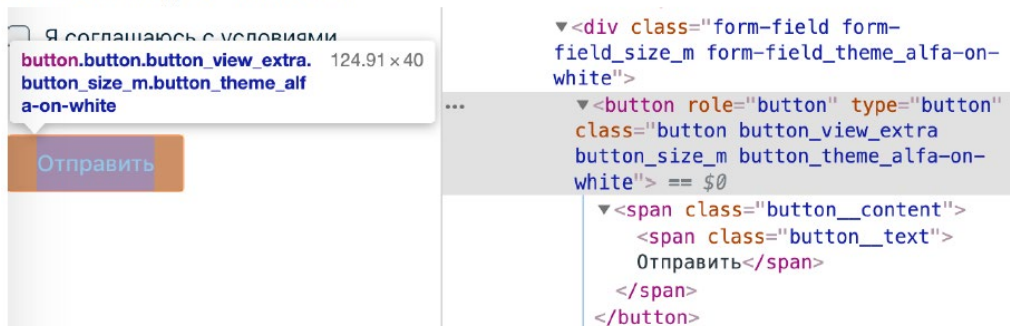
Важно запомнить следующую вещь: даже если HTML написан с нарушением всех правил спецификации, браузер его всё равно отобразит, при этом элементы создаст так, как посчитает нужным. Поэтому вам всегда нужно ориентироваться не на исходный код страницы (то, как написал разработчик), а на представление, полученное браузером (см. Developer Tools).

Посмотрим на представление разметки в Developer Tools (F12 или Ctrl + Shift + I):

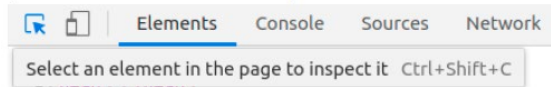


```
<!doctype html>
<html lang="en">
  <head>...</head>
  <body> == $0
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root">
      <div class="App_appContainer_3jRx1">
        <h4 class="heading heading_size_s heading_theme_alfa-on-white">
          Заказать обратный звонок</h4>
        <form action="/" enctype="application/x-www-form-urlencoded"
          method="post" autocomplete="on" class="form form_size_m
          form_theme_alfa-on-white">
          <div class="form-field form-field_size_m form-
            field_theme_alfa-on-white">...</div>
          <div class="form-field form-field_size_m form-
            field_theme_alfa-on-white">...</div>
          <div class="form-field form-field_size_m form-
            field_theme_alfa-on-white">...</div>
          <div class="form-field form-field_size_m form-
            field_theme_alfa-on-white">...</div>
        </form>
      </div>
    </div>
```

Обратите внимание, при наведении на конкретный узел в дереве сам элемент подсвечивается:



Кроме того, вы можете выбрать на странице элемент и найти его в дереве (либо на элементе правой кнопкой мыши — «Посмотреть код»):



СТРАТЕГИЯ ПОИСКА

1. id (по атрибуту id) не подходит (и добавить мы его не может — вспомните лекцию про

Unit-тестирование);

2. linkText и partialLinkText (по тексту гиперссылки — тег a) не подходит;

3. name (по атрибуту name) не подходит;

4. tagName (по названию тега) подходит;

5. xpath (пока не знаем, что такое);

6. classname (по атрибуту class) подходит ;

7. cssSelector (пока не знаем, что такое).

Дальше — всё просто, мы знаем, что из себя представляет поле ввода Имени:

```
<input class="input__control" type="text"
      view="default" autocomplete="on" value="">
```

Нам нужно:

1. Найти его;
2. Ввести в него текст.

СТРАТЕГИЯ ПОИСКА

Вспомним класс `By`, который будет позволять нам искать:

```
public abstract class By {
    public static By id(String id) { }
    public static By linkText(String linkText) { }
    public static By partialLinkText(String partialLinkText) { }
    public static By name(String name) { }
    public static By tagName(String tagName) { }
    public static By xpath(String xpathExpression) { }
    public static By className(String className) { }
    public static By cssSelector(String cssSelector) { }
```

КАК ИЩЕТ SELENIUM

Поиск осуществляется по следующему алгоритму:

1. Если ищем внутри документа все элементы `findElements`, то возвращается список из элементов, упорядоченный на основании того, как элементы встречаются в самом документе (самый первый наверху).
2. Если ищем внутри документа один элемент `findElement`, то возвращается первый найденный (см. п.1).

```
1 WebElement nameInput = driver.findElement(By.className("input__control"));
2 nameInput.sendKeys("Василий");
3 // либо просто
4 driver.findElement(By.className("input__control")).sendKeys("Василий");
```

Если ищем внутри другого элемента, то алгоритм такой же, как для документа, только мы ищем внутри дочерних элементов, а не всего документа.

Q: Но мы же видим чекбокс?

A: На самом деле, в современном вебе есть ряд элементов, которые достаточно плохо стилизуются:

— чекбоксы

— радио-переключатели

— выпадающие списки (select)

— поля выбора файлов

— и т.д.

Поэтому веб-разработчики с помощью различных трюков скрывают настоящие элементы и

Если вы искали один элемент, и вдруг так случилось, что этого элемента на странице не оказалось, то вы получите исключение

`NoSuchElementException` (в примере добавлен суффикс 404):

```
org.openqa.selenium.NoSuchElementException: no such element:
Unable to locate element: {"method":"css selector","selector":".input__control\404"}

...

ru.netology.selenium.CallbackTest > shouldTestSomething() FAILED
org.openqa.selenium.NoSuchElementException at CallbackTest.java:34
```

Если же вы искали все элементы (`findElementsBy`), то вы просто получите пустой список.

ОБЩИЙ КОД

```
@Test
void shouldSubmitRequest() {
    driver.get("http://localhost:9999");
    List<WebElement> elements = driver.findElements(By.className("input__control"));
    elements.get(0).sendKeys("Василий");
    elements.get(1).sendKeys("+79270000000");
    driver.findElement(By.className("checkbox__control")).click();
    driver.findElement(By.className("button")).click();
    String text = driver.findElement(By.className("alert-success")).getText();
    assertEquals("Ваша заявка успешно отправлена!", text.trim());
}
```

ElementClickInterceptedException

```
org.openqa.selenium.ElementClickInterceptedException: element click intercepted:
Element <input class="checkbox__control" type="checkbox" autocomplete="off" value="">
is not clickable at point (304, 229).
```

```
Other element would receive the click: <span class="checkbox__box">...</span>
```

Q: Что не так? Ведь элемент действительно существует?

A: Дело в том, что Selenium пытается вести как пользователь - а пользователь не может кликать на невидимые элементы или элементы, закрытые другими элементами.

эмулируют их поведение, "подсовывая" пользователю фейковые элементы. Вот на них-то пользователь и будет кликать.

Q: Как это распознать и что с этим делать?

A: На начальном этапе - методом проб и ошибок.

В целом же, нужно изучать веб-разработку. Так или иначе, если вы собираетесь быть автоматизатором,

то вы постоянно должны быть в курсе:

— как происходит веб-разработка

— какие подходы, приёмы и инструменты используются

ОБРАТИТЕ ВНИМАНИЕ

1. Наш тест выглядит из серии в первое найденное на странице поля ввода введите 'Василий', а во

второе — '+7....' — если бы вы написали такой кейс для ручного

ОБЩИЙ КОД

```
@Test
void shouldSubmitRequest() {
    driver.get("http://localhost:9999");
    List<WebElement> elements = driver.findElements(By.className("input__control"));
    elements.get(0).sendKeys("Василий");
    elements.get(1).sendKeys("+79270000000");
    driver.findElement(By.className("checkbox__box")).click();
    driver.findElement(By.className("button")).click();
    String text = driver.findElement(By.className("alert-success")).getText();
    assertEquals("Ваша заявка успешно отправлена!", text.trim());
}
```

тестирования, вам бы вряд ли дали позитивный фидбек;

2. Мы не используем assert'ы на каждом шагу, т.к. если что-то не получилось, мы получим Exception, который обрушит наш тест;

3. В строке assertEquals чаще всего будет крыться «бомба» на будущее, т.к. при любом изменении текста наш тест будет падать (а фразы меняются очень часто)*.

Примечание*: на этот счёт есть разные мнения - должны мы проверять элемент по доп.характеристикам

(классы, атрибуты) или по содержимому. Если подумать, то реальный пользователь будет ориентироваться именно по содержимому (в данном случае - тексту сообщения).

Промежуточные выводы пока неутешительны:

1. Слишком много работы по настройке инфраструктуры;
2. Дурацко выглядящие тест-кейсы, проверяющие только то, что мы написали;
3. Нам всё-таки сначала вручную нужно проделать всё, только потом, проанализировав, автоматизировать;
4. Мы пока даже не думали о генерации тестовых данных, сетевых задержках, недоступных сервисах и управлении состоянием.

CSS-селекторы — это специальный язык выражений, позволяющий осуществлять поиск на странице. Описаны

в спецификации CSS Selectors. Текущая версия (в разработке) — 4, но Selenium требует, как минимум, поддержки 2-ой версии.

Ключевые для нас:

- input (либо любое другое название тега) — селектор по тегу;
- #username — селектор по id;
- .input control — селектор по классу;

- [type] — селектор по наличию атрибута;
- [type="submit"] — селектор по наличию атрибута со значением.

Из CSS-селекторов можно создавать комбинации, позволяющие более точно выбирать элементы на странице:

- S1S2 (input.input control) — только те поля ввода, у которых есть класс input control ;
- S1 S2 (form input) — только те поля ввода, которые расположены внутри form на любой глубине вложенности;
- и т.д.

Но ни то, ни другое кардинально не решает проблем. Хотя, если другого выхода нет, приходится пользоваться этим инструментарием.

Безусловно, лучшей практикой является назначение уникальных для всей страницы (либо в конкретной её области) идентификаторов для элементов.

id для этого не очень подходит, т.к. для больших порталных систем один и тот же виджет может встречаться несколько раз.

data-test-id

Например, вот так может выглядеть форма, размеченная с помощью data-test-id:

```
<form action="/" enctype="application/x-www-form-urlencoded" method="post" autocomplete="on"
class="..." data-test-id="callback-form">
  <div class="...">
    <span class="..." data-test-id="name">
      <span class="input__inner">
        <span class="input__top">Как вас зовут</span>
        <span class="input__box">
          <input class="input__control" type="text" view="default" autocomplete="on" value="">
        </span>
      </span>
    </span>
  </div>
  ...
</form>
```

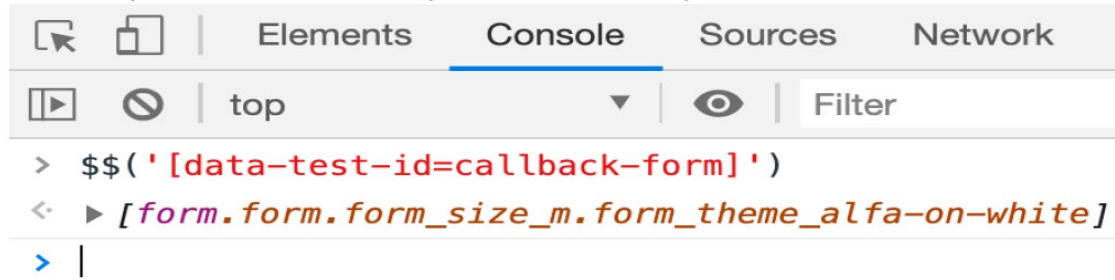
Спецификация HTML разрешает создавать собственные атрибуты с префиксом data-, например, data-id (или data-testid, data-test-id).

Ключевое здесь — добиться от программистов включения в код этих

Примечание*: обратите внимание - не всегда программисты выставляют идентификаторы туда, куда нужно (и не всегда у них есть физически возможность это сделать, например, если они используют библиотеку).

идентификаторов. Без них вам будет очень тяжело, т.к. ваши тесты будут хрупкими, а следовательно, неэффективными.

В Developer Tools можно тестировать css-селекторы:



Selenide — это
фреймворк
для

data-test-id

```
@Test
void shouldSubmitRequest() {
    driver.get("http://localhost:9999");
    WebElement form = driver.findElement(By.cssSelector("[data-test-id=callback-form]"));
    form.findElement(By.cssSelector("[data-test-id=name] input")).sendKeys("Василий");
    form.findElement(By.cssSelector("[data-test-id=phone] input")).sendKeys("+79270000000");
    form.findElement(By.cssSelector("[data-test-id=agreement]")).click();
    form.findElement(By.cssSelector("[data-test-id=submit]")).click();
    String text = driver.findElement(By.className("alert-success")).getText();
    assertEquals("Ваша заявка успешно отправлена!", text.trim());
    Thread.sleep(5000);
}
```

Уже получше, но видно, что мы замучаемся каждый раз писать
`findElement(By.cssSelector(...))`.

Кроме того, осталась нерешённой проблема с ручным скачиванием драйвера. Неплохо бы это оптимизировать.

ElementNotInteractableException

Q: Зачем мы ищем `input (By.cssSelector("[data-test-id=name] input"))`? Разве нельзя просто в элемент с `data-test-id=name` ввести текст?

A: К сожалению - нет, Selenium позволяет вводить текст в поля ввода, но не в элементы `span`.

В рамках курса мы будем использовать именно Selenide и практиковаться в использовании этого API. Ключевое для нас следующее:

1. Сам скачивает драйвер;

автоматизированного
тестирования веб-

приложений на
основе Selenium
WebDriver, дающий
следующие
преимущества*:

- Изящный API;
- Поддержка Ajax для стабильных тестов;
- Мощные селекторы;
- Простая конфигурация.

SELENIDE API

Описание API
представляет из себя
буквально одну
страницу, которую
вам рекомендуем
посмотреть.

build.gradle

```
plugins {  
    id 'java'  
}  
  
group 'ru.netology'  
version '1.0-SNAPSHOT'  
  
sourceCompatibility = 1.8  
  
repositories {  
    mavenCentral()  
}  
  
dependencies {  
    testImplementation 'org.junit.jupiter:junit-jupiter:5.5.1'  
    testImplementation 'com.codeborne:selenide:5.3.1'  
}  
  
test {  
    useJUnitPlatform()  
}
```

```
1 class CallbackTest {  
2     @Test  
3     void shouldSubmitRequest() {  
4         open("http://localhost:9999");  
5         SelenideElement form = $("[data-test-id=callback-form]");  
6         form.$("[data-test-id=name] input").setValue("Василий");  
7         form.$("[data-test-id=phone] input").setValue("+79270000000");  
8         form.$("[data-test-id=agreement]").click();  
9         form.$("[data-test-id=submit]").click();  
10        $(".alert-success").shouldHave(exactText("Ваша заявка успешно отправлена!"));  
11    }  
12 }
```

2. \$('selector') для поиска одного элемента, \$\$('selector')

для поиска нескольких;

3. Используем SelenideElement ;

4. Включает матчеры для проверок.

Кроме того, содержит всякие плюшки вроде создания скриншотов при падении теста (попробуйте поменять

«отправлена» на «принята»).

SelenideElement наследуется от WebElement , добавляя к нему достаточно много различных методов

Сегодня мы рассмотрели очень много важных тем и выявили ключевые

проблемы при автоматизации.

Кроме того, посмотрели, как некоторые из этих проблем решаются с помощью Selenide.

Не забывайте, что Selenide это надстройка над Selenium, поэтому мы продолжим обсуждать некоторые моменты, касающиеся именно Selenium, т.к. они будут проецироваться на Selenide, например:

```
ru.netology.selenium.CallbackTest > shouldSubmitRequest() FAILED
```

```
com.codeborne.selenide.ex.ElementNotFound at CallbackTest.java:17
```

```
Caused by: org.openqa.selenium.NoSuchElementException at CallbackTest.java:17
```