SELENDINE

Ключевое для нас при взаимодействии с веб-страницей — это найти нужные нам элементы (что позволит затем взаимодействовать с ними).Посмотрим, что нам предлагает в этом плане Selenide

Как вы помните, для поиска элементов используются два метода:

1. $ — поиск одного элемента
2. $$ — поиск коллекции элементов

Оба эти методы являются перегруженными (т.е. их можно вызывать с разными типами параметров*).

Примечание:* это упрощённое представление перегрузки.

ПОИСК ПО CSS-СЕЛЕКТОРУ

Если в качестве аргумента передаётся строка (String), то она интерпретируется как CSS Selector (самый распространённый способ поиска).

КОМБИНАЦИИ СЕЛЕКТОРОВ

— S1, S2 : S1 и/или S2;

— S1 S2 : один из родителей — S1, дочерний — S2 (на любой глубине вложенности);

— S1 > S2 — непосредственный родитель S1, дочерний — S2;

— S1 + S2 — предыдущий «соседний»* — S1, следующий — S2;

— S1 ~ S2 — предыдущий «сестринский» — S1, один из следующих — S2.

Во всех случаях, кроме первого, выбирается элемент S2.

Примечание:* на том же уровне (т.е. один родитель)

Если мы посмотрим на разметку, то есть всего несколько вещей, за

которые мы можем «зацепиться» при выборе элемента (т.к. никакого

data-id , data-testid , data-test-id и т.д. нет и в помине):

— положение относительно формы (или всей страницы);

— атрибут name ;

— текст над полем ввода.

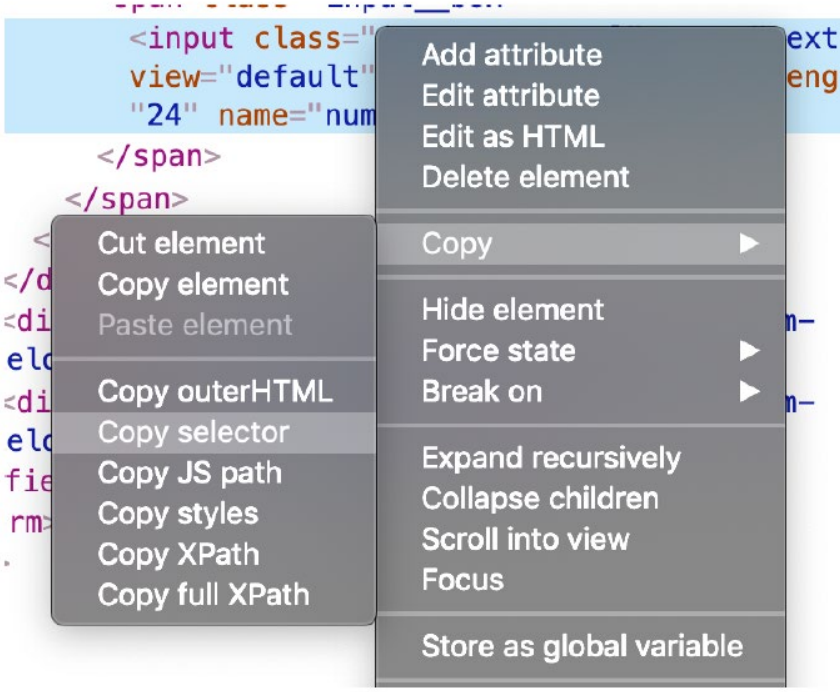
Примечание: пример специально разработан подобным образом, чтобы показать варианты работы без testid

.

РАЗМЕТКА

```
▼<form action="/" enctype="application/x-www-form-
urlencoded" method="post" autocomplete="on" class="form
form_size_m form_theme_alfa-on-white">
  ▼<fieldset>
    ▼<div class="form-field form-field_size_m form-
    field_theme_alfa-on-white">
      ▼<span class="input input_type_text
      input_view_default input_size_m
      input_width_available input_has-label
      input_theme_alfa-on-white">
        ▼<span class="input__inner">
          <span class="input__top">Номер счёта</span>
          ▼<span class="input__box">
            <input class="input__control" type="text"
            view="default" autocomplete="on" maxlength=
            "24" name="number" value> == $0
          </span>
        </span>
      </span>
    </div>
```

АБСОЛЮТНЫЙ СЕЛЕКТОР



АВТОГЕНЕРАЦИЯ

Обратите внимание на класс

.Registration_registrationForm 2mPnE .

АВТОГЕНЕРАЦИЯ КЛАССОВ

На самом деле, так работают специальные инструменты, позволяющие автоматически генерировать непересекающиеся имена классов.

В зависимости от того, как программист изменит стили (или класс), может быть автоматически сгенерирован новый класс.

Стоит отметить, что речь идёт именно о CSS классах

ИЗМЕНЕНИЕ СТРУКТУРЫ

Конечно, мы можем вручную модифицировать «абсолютный» путь так, чтобы убрать подводные камни, но стоит выбирать этот способ только тогда, когда нет способов лучше.

Кроме того, не стоит забывать, что при подобном подходе изменения структуры

страницы(добавление/удаление пары тегов) приведут к падению тестов.

АБСОЛЮТНЫЙ ПУТЬ

В итоге селектор будет выглядеть как-то так: `#root > div > div.Registration_registrationForm__2mPnE > form > fieldset > div:nth-child(1) > span > span > span.input__box > input`

АТРИБУТ `name`

Вариант с `name` был бы хорош, если бы на странице не было ещё одного элемента с тем же `name` (поле для ввода карты).

Но почему, если искать через консоль (с помощью синтаксиса `$$("[name='number']")`) находится только один вариант?

```
> $$("[name='number']")
  ▶ [input.input__control]
>
```

ОТОБРАЖЕНИЕ И СКРЫТИЕ ЭЛЕМЕНТОВ

Есть две ключевых техники отображения и скрытия элементов:

1. Динамическое удаление/создание элементов (элемент удаляется из DOM-дерева и заново добавляется в него при необходимости);

2. Изменения свойств, отвечающих за показ элемента (сам элемент остаётся в DOM-дереве).

В нашем случае используется первый вариант.

The **Document Object Model** (**DOM**) is a cross-platform and language-independent interface that treats an XML or HTML document as a tree structure wherein each node is an object representing a part of the document. The DOM represents a document with a logical tree. Each branch of the tree ends in a node, and each node contains objects. DOM methods allow programmatic access to the tree; with them one can change the structure, style or content of a document. Nodes can have event handlers attached to them. Once an event is triggered, the event handlers get executed.[2]

The principal standardization of the DOM was handled by the World Wide Web Consortium (W3C), which last developed a recommendation in 2004. WHATWG took over the development of the standard, publishing it as a living document. The W3C now publishes stable snapshots of the WHATWG standard.

DOM API

Браузер предоставляет так называемое DOM API, которое и позволяет пользовательским скриптам (чаще всего на JS) программно модифицировать дерево элементов и их свойства.

Вам важно разделять различные сценарии, используемые приложением, т.к. исходя из этого может меняться логика написания вами авто-теста*.

Примечание:* но если приложение изначально рассчитано на тестирование и использует data-test-id , то проблем и зависимостей будет гораздо меньше.

Q: Но если мы можем программно "подменять" кусочки DOM-дерева,можем ли мы подменять целиком страницу?

А: Почти. На самом деле, целиком подменять смысла нет, мы можем перестраивать большую часть "прозрачно" для пользователя. Так, что он будет думать, что переходит по обычным HTML-страницам.

SPA (Single Page Applications) — подход, при котором вместо загрузки нового HTML документа при каждой отправке формы, клике по ссылке и т.д., просто перестраивается дерево DOM.

Дерево генерируется программно с помощью JS и DOM API, а URL в адресной строке браузера меняется посредством JS и History API

Таким образом, у пользователя создаётся ощущение того, что он действительно переходит по страницам приложения (хотя на самом деле — это всего лишь иллюзия).

Ajax (also **AJAX** /ˈeɪdʒæks/; short for "Asynchronous JavaScript and XML")[1][2] is a set of web development techniques using many web technologies on the client-side to create asynchronous web applications. With Ajax, web applications can send and retrieve data from a server asynchronously (in the background) without interfering with the display and behaviour of the existing page. By decoupling the data interchange layer from the presentation layer, Ajax allows web pages and, by extension, web applications, to change content dynamically without the need to reload the entire page.[3] In practice, modern implementations commonly utilize JSON instead of XML.

Ajax is not a single technology, but rather a group of technologies. HTML and CSS can be used in combination to mark up and style information. The webpage can then be modified by JavaScript to dynamically display—and allow the user to interact with—the new information. The built-in XMLHttpRequest object, or since 2017 the new `fetch` function within JavaScript, is commonly used to execute Ajax on webpages, allowing websites to load content onto the screen without refreshing the page. Ajax is not a new technology, or a different language, just existing technologies used in new ways.

XHR & FETCH

При необходимости взаимодействия с сервером из JS с помощью AJAX- запросов и других технологий отправляются запросы и принимаются ответы без перезагрузки страницы.

Таким образом, HTML, CSS и JS на самом деле загружаются только при первой загрузке страницы*, а в дальнейшем между сервером и браузером передаются только данные.

Примечание:* конечно же, здесь есть различные варианты

Попробуем написать авто-тест, который проходит процедуру регистрации по номеру счёта.

Сценарий следующий:

1. Переключаемся на вкладку «По номеру счёта»;

АВТО-ТЕСТ

```
@Test
void shouldRegisterByAccountNumber() {
  open("http://localhost:9999");
  $$(".tab-item").last().click();
  $("[name='number']").setValue("40550100012346138564");
  $("[name='phone']").setValue("+792000000000");
  $("button").click(); // кликаем на первой попавшейся кнопке :-)
  // TODO:
}
```

2. Заполняем поле номер счёта: 40550100012346138564;

3. Заполняем поле телефон: +792000000000;

4. Нажимаем на кнопку «Продолжить»;

5. Ждём появления надписи, что всё успешно;

6. Ждём перехода на страницу «Личный кабинет».

ELEMENTS COLLECTION

`$$` возвращает `ElementsCollection`, который обладает большим количеством полезных методов (а кроме того, наследуется от `AbstractList`, т.е. содержит все методы `List`).

```
1 ElementsCollection exclude(Condition)
2 ElementsCollection filter(Condition)
3 SelenideElement find(Condition)
4 SelenideElement first()
5 SelenideElement last()
6 ElementsCollection should*
```

ОСОБЕННОСТИ ПОИСКА

На самом деле поиск происходит только тогда, когда мы взаимодействуем с элементом (элементами) либо выполняем проверки `should*`, т.е. если мы просто напишем:

```
1 @Test
2 void shouldRegisterByAccountNumber() {
3   open("http://localhost:9999");
4   $$(".tab-item").last();
5   // никакого поиска производится не будет, пока не вызовем click или should*
6 }
```

Это относится и к `$`, и к `$$`.

Давайте подумаем, а как реальный пользователь будет заполнять нашу форму?

Вряд ли он будет искать «последнюю вкладку» (или чего хуже по data- test-id).

Пользователь ищет по тексту.

Но мы с вами говорили, что искать по тексту чревато нестабильностью тестов (если текст меняется).

В чём здесь противоречие?

На предыдущем слайде мы использовали достаточно опасный приём: мы сами построили гипотезу поведения пользователя (с этим всё ок), но не проверили её.

JAVADOC

Все подобные особенности Selenide описаны в JavaDoc к методам (F4 в IDEA на имени метода):

```
/**
* Locates the first element matching given CSS selector
* ATTENTION! This method doesn't start any search yet!
* @param cssSelector any CSS selector like "input[name='first_name']" or "#messages .new_message"
* @return SelenideElement
*/
public static SelenideElement $(String cssSelector) {
  return getSelenideDriver().find(cssSelector);
}
```

Обязательно читайте JavaDoc к методам и классам Selenide перед их использованием.

Если мы будем на основании непроверенных на реальных пользователях гипотезах основывать дальнейшие

решения — это ничем не лучше (а возможно и хуже), чем основывать их на гороскопах или подбрасывании монет.

Запомните: вы не пользователь! Вы заранее не знаете как он будетработать (даже несмотря на то, что вы можете быть в этом уверены.Противоречий между поиском по тексту и поиском по data-test-id

нет, т.к. данные подходы решают разные задачи:

— data-test-id обеспечивает стабильность;

— текст обеспечивает эмуляцию действий пользователя.

Скорее всего, вам придётся совмещать их, чтобы обеспечить и нужный уровень стабильности (найти ключевой блок и работать внутри него, а не по всей странице) и по тексту (эмулировать действия пользователя).

Но ведь Selenium (когда мы его рассматривали) не умел искать по тексту.

Можно ли как-то искать по тексту?

На самом деле, Selenium не умел искать при помощи CSS Selector’ов, но вполне себе умел при помощи XPath.

Например, для поиска по тексту можно использовать следующее выражение: "//*[text()='По номеру счёта']" .

XPath (XML Path Language) — язык запросов к элементам XML-документа. Но в нашем случае, браузеры его поддерживают и для HTML-документов.

В консоли браузера или Selenide вы можете попробовать при помощи выражения $x .

Глубоко копать в XPath мы не будем, скажем лишь, что он достаточно тяжёл для восприятия, но иногда используется в автоматизации при отсутствии альтернатив.

Несмотря на то, что вышло уже несколько стандартов, самым поддерживаемым остаётся 1.0, вышедший в 1999 году.

SELECTORS

Авторы Selenide понимали, что возможность искать по тексту критически важна, поэтому добавили соответствующий метод поиска с помощью вспомогательного класса `Selectors`:

```
By withText(String elementText)
By byText(String elementText)
By byAttribute(String attributeName, String attributeValue)
By by(String attributeName, String attributeValue)
By byTitle(String title)
By byValue(String value)
By byName(String name)
By byXpath(String xpath)
By byLinkText(String linkText)
By byPartialLinkText(String partialLinkText)
By byId(String id)
By byCssSelector(String css)
By byClassName(String className)
```

АВТО-ТЕСТ

```
@Test
void shouldRegisterByAccountNumber() {
  open("http://localhost:9999");
  $(byText("По номеру счёта")).click();
  $("[name='number']").setValue("4055 0100 0123 4613 8564");
  $("[name='phone']").setValue("+792000000000");
  $(byText("Продолжить")).click();
  // TODO:
}
```

REAL WORLD

Нужно отметить, что в реальных условиях вы будете просить `data-test-id` на определённый блок на странице и уже внутри него искать свои элементы:

```
@Test
void shouldRegisterByAccountNumber() {
  open("http://localhost:9999");
  $$(".tab-item").find(exactText("По номеру счёта")).click();
  SelenideElement block = $("[data-testid='registration']");
  block.$("[name='number']").setValue("4055 0100 0123 4613 8564");
  block.$("[name='phone']").setValue("+792000000000");
  block.$$("button").find(exactText("Продолжить")).click();
  // TODO
```

Expected: visible

Screenshot: file:./build/reports/tests/1570...0.png

Page source: file:./build/reports/tests/1570...0.html

Timeout: 4 s.

Condition

На самом деле, так искать не очень удобно, т.к. мы ищем буквально по всем элементам на странице без возможности выбора только из нужной группы.

Condition — класс, предоставляющий возможность задавать условия (как поиска, так и для assert 'ов вроде should*).

Q: в чём разница между visible и exists .

A: visible означает, что элемент видим (пользователю в том числе), exists означает, что элемент просто существует на странице (но при этом может быть невидимым).

Q: в чём разница между byText и withText ?

A: byText ищет по соответствию текста (с игнорированием незначащих пробелов), а withText ищет по наличию текста (т.е. по подстроке).

Element not found {with text: Успешная авторизация}

WAITING

Теперь нам нужно дождаться появления сначала информационного сообщения о том, что всё прошло успешно, затем, что загрузился личный кабинет:

```
@Test
void shouldRegisterByAccountNumber() throws InterruptedException {
  open("http://localhost:9999");
  $$(".tab-item").find(exactText("По номеру счёта")).click();
  $("[name='number']").setValue("4055 0100 0123 4613 8564");
  $("[name='phone']").setValue("+792000000000");
  $$("button").find(exactText("Продолжить")).click();
  $(withText("Успешная авторизация")).shouldBe(visible);
  $(byText("Личный кабинет")).shouldBe(visible);
}
```

Обратите внимание, что помимо скриншота (на котором видно, что идётзагрузка) есть и снимок в формате HTML, который мы можем проанализировать на наличие элементов.

Итак, что же случилось? По факту, если тестировать руками, то с указанными данными регистрация успешно проходит.

На самом деле, Selenide работает следующим образом: он опрашивает страницу с определённым интервалом на наличие элемента(или выполнение условий)* в течение фиксированного таймаута (по умолчанию — 4 секунды).

Важно: поиск выполняется только при взаимодействии с элементом или проверки условий should .

Q: почему бы просто не поставить «бесконечный» таймаут? Или очень большой?

A: бесконечный нельзя, т.к. во-первых, это противоречит логике — ни один пользователь не ждёт бесконечно, а во-вторых, тогда в системе CI мы получим «незавершающиеся тесты».

Мы, конечно: можем выставить «глобальный» таймаут через Configuration.timeout (в мс), но лучше для любого случая, превышающего 4 секунды, выставлять конкретное время (по-хорошему,это должно быть в требованиях).

Configuration — ключевой класс конфигурации Selenide.

В JavaDoc к нему расписаны возможные опции конфигурации Selenide (как программной, так и аргументами командной строки).

Обязательно изучите его!

TIMEOUT

```
@Test
void shouldRegisterByAccountNumber() throws InterruptedException {
  open("http://localhost:9999");
  $$(".tab-item").find(exactText("По номеру счёта")).click();
  $("[name='number']").setValue("4055 0100 0123 4613 8564");
  $("[name='phone']").setValue("+792000000000");
  $$("button").find(exactText("Продолжить")).click();
  $(withText("Успешная авторизация")).waitUntil(visible, 5000);
  $(byText("Личный кабинет")).waitUntil(visible, 5000);
}
```

Что это значит для нас: это значит, что «зелёные тесты» проходят быстро (т.к.) мы ничего не ждём, а «красные тесты» падают медленно, т.к. и мы ждём либо по таймауту по умолчанию (4 секунды), либо по выставленному нами.

И только если в течение этого таймаута не удаётся найти нужный нам элемент (либо дождаться выполнения условия), Selenide обрушит тест.

Итак, мы разобрали первый сценарий работы с динамическими страницами — объекты создаются и удаляются с помощью DOM API*.

Остался второй сценарий: скрытие и показ элементов.

Примечание:* стоит отметить, что современные фреймворки, например, React, делают это более эффективно, лишь обновляя необходимые свойства элементов и дочерние элементы

Для того, чтобы скрыть элемент (не показывать его пользователю), не

обязательно удалять его со страницы.

Можно средствами JS модифицировать следующие CSS-свойства элемента:

— visibility — значения visible / hidden ;

— display — значения block / none ;

— позиционирование — вынести элемент за пределы страницы или

«закрыть» его другим элементом (расположив другой элемент сверху)

Возьмём другую версию приложения, где переключение по вкладкам реализовано простым скрытием:

```
> $$("[name='number']")
◂ ▶ (2) [input.input__control, input.input__control]
```

Но раз элемент `уже` доступен, почему бы не попробовать обратиться к нему сразу?

```
@Test
void shouldRegisterByAccountNumber() throws InterruptedException {
  open("http://localhost:9999");
  $$("[name='number']").last().setValue("4055 0100 0123 4613 8564");
  $("[name='phone']").setValue("+792000000000");
  $$("button").find(exactText("Продолжить")).click();
  ...
}
```

```
Element should be visible or transparent:
visible or have css value opacity=0
{[name='number'].last}
...
Timeout: 4 s.
```

Т.е. в данном случае Selenide подождал положенные 4 секунды в надежде на то, что элемент всё-таки станет видимым (или прозрачным) и после этого «обрушил» наш тест, сообщив нам о том, что взаимодействовать мы можем только с видимыми элементами.

```
@Test
void shouldRegisterByAccountNumber() {
  open("http://localhost:9999");
  $$(".tab-item").find(exactText("По номеру счёта")).click();
  $$("[name='number']").last().setValue("4055 0100 0123 4613 8564");
  $$("[name='phone']").last().setValue("+792000000000");
  $$("button").find(exactText("Продолжить")).click();
  $(withText("Успешная авторизация")).waitUntil(visible, 5000);
  $(byText("Личный кабинет")).waitUntil(visible, 5000);
}
```

1. Поговорили о SPA и о том, как строится отображение различных блоков в современных приложениях;
2. Посмотрели на то, как Selenide ищет элементы и работает с таймаутами;
3. Выяснили, что со скрытыми элементами работать нельзя.

Кроме того, вы можете отметить, что обе приведённых реализации теста (особенно вторая) являются «ужасными».Происходит это в первую очередь от того, что нам фактически не за что зацепиться на cnранице — приложение изначально не заточено на удобство тестирования.Поэтому старайтесь сразу внедрять в свои проекты практики по выделению нужных блоков страницы с помощью data-test-id .Пока мы тестировали достаточно простые интерфейсы, но давайте подумаем, а что если в это приложение добавится две функции:

1. Подтверждение по SMS;

2. Блокировка при трехкратном неверном вводе кода подтверждения.

Тогда во весь встанет проблема интеграции с внешними системами (SMS) и управления тестовыми данными (т.к. если мы запустим тест, блокирующий учётную запись, то использовать её уже не сможем)