



# ТЕСТИРОВАНИЕ ВЕБ-ИНТЕРФЕЙСОВ. SELENIDE



АРТЕМ РОМАНОВ



# АРТЕМ РОМАНОВ

Инженер по обеспечению качества



[romanov-artyom](#)



[@Artem\\_Telegram](#)



[Романов Артём](#)



# ПЛАН ЗАНЯТИЯ

1. [Selenide](#)
2. [Поиск элементов](#)
3. [Модификация DOM](#)
4. [Timeout](#)
5. [Изменение видимости элементов](#)
6. [Итоги](#)



**SELENIDE**



# SELENIDE

Мы уже познакомились с тем, как запускать Selenide и выполнять базовые манипуляции: клики и установка значения полей.

Кроме того, посмотрели на базовые ассерты, предлагаемые Selenide.

Сегодня наша задача: изучить этот инструмент глубже и посмотреть на более расширенные техники.

# ЗАДАЧА

Наша задача — протестировать возможность регистрации по номеру счёта в интернет банке.

По умолчанию показывается форма регистрации по номеру карты:

## Регистрация



Мы гарантируем безопасность ваших данных

ПО НОМЕРУ КАРТЫ

ПО НОМЕРУ СЧЁТА

Номер карты

---

Месяц

---

Год

---

Мобильный телефон

---

Продолжить

# ЗАДАЧА

Форма регистрации через номер счёта показывается только после клика на вкладку «По номеру счёта»:

## Регистрация



Мы гарантируем безопасность ваших данных

ПО НОМЕРУ КАРТЫ

ПО НОМЕРУ СЧЁТА

Номер счёта

Мобильный телефон

Продолжить

# ЗАДАЧА

Кроме того, после ввода всех данных есть «небольшая» задержка в несколько секунд, при которой происходит валидация введённых данных:

## Регистрация



Мы гарантируем безопасность ваших данных

ПО НОМЕРУ КАРТЫ

ПО НОМЕРУ СЧЁТА

Номер счёта

0000 0000 0000 0000 0000

Мобильный телефон

+7 000 000 00 00



Подождите...





# УПРОЩЕНИЯ

На этой лекции мы будем рассматривать упрощённую форму: без верификации по SMS и без возможных блокировок при многократном неправильном вводе кода верификации.

Об этих вопросах мы поговорим на следующих лекциях.



# ПОИСК ЭЛЕМЕНТОВ



## ПОИСК ЭЛЕМЕНТОВ

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.

# ПОИСК ЭЛЕМЕНТОВ

Как вы помните, для поиска элементов используются два метода:

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

Оба эти метода 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).

# CSS SELECTORS

Ключевые селекторы:

- По id: `#id`;
- По классу `.class`;
- По атрибуту:
  - Наличие атрибута: `[attr]`;
  - Атрибут с определённым значением: `[attr='value']` \*.

Примечание: \* `value` можно и не заключать в кавычки, если оно не содержит пробелов.

# 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 –  $S2$ ;
- $S1 \sim S2$  – предыдущий «сестринский» –  $S1$ , один из следующих –  $S2$ .

Во всех случаях, кроме первого, выбирается элемент  $S2$ .

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

# 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>
    </span>
  </div>

```



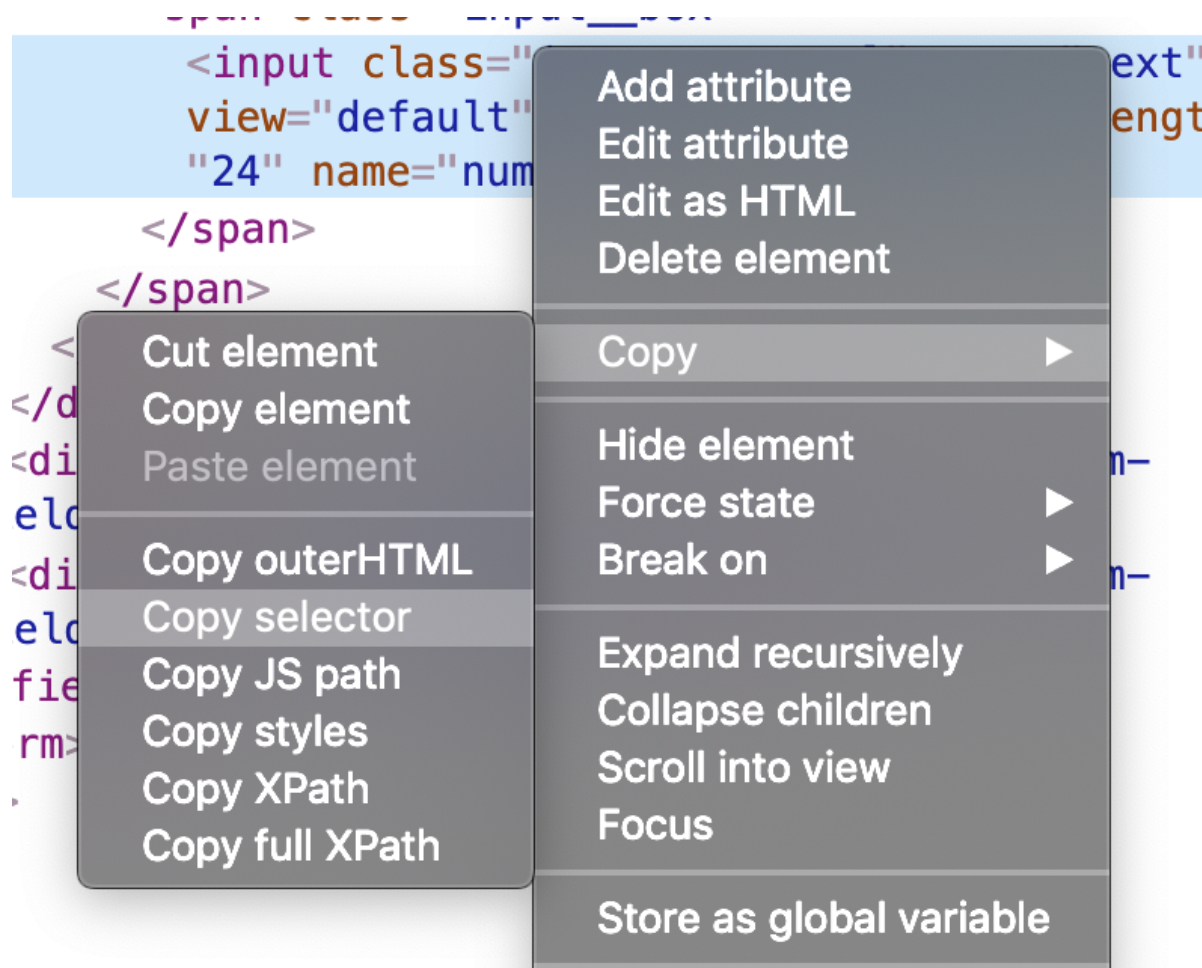
# ВЫБОР СЕЛЕКТОРА

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`.

# АБСОЛЮТНЫЙ СЕЛЕКТОР



# 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`

Вопрос к аудитории: как вы думаете, чем плох этот вариант?

# АВТОГЕНЕРАЦИЯ

Обратите внимание на класс

```
.Registration_registrationForm__2mPnE.
```

Как вы думаете, его придумывал человек (особенно последнюю часть)?



# 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.

## 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.



# 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.



# SPA

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

A: 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

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).



## 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.



# МОДИФИКАЦИЯ DOM



# АВТО-ТЕСТ

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

Сценарий следующий:

1. Переключаемся на вкладку «По номеру счёта»;
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».

# ABTO-TEST

```
1  @Test
2  void shouldRegisterByAccountNumber() {
3      open("http://localhost:9999");
4      $$(".tab-item").last().click();
5      $("[name='number']").setValue("40550100012346138564");
6      $("[name='phone']").setValue("+792000000000");
7      $("button").click(); // кликаем на первой попавшейся кнопке :- )
8      // TODO:
9  }
```

# 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 `$$`.

# JAVADOC

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

```
1  /**
2   * Locates the first element matching given CSS selector
3   * ATTENTION! This method doesn't start any search yet!
4   * @param cssSelector any CSS selector like "input[name='first_name']" or "#messages .new_message"
5   * @return SelenideElement
6   */
7  public static SelenideElement $(String cssSelector) {
8      return getSelenideDriver().find(cssSelector);
9  }
```

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

# ПОИСК ПО ТЕКСТУ

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?

# ГИПОТЕЗЫ

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

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 будет работать (даже несмотря на то, что вы можете быть в этом уверены).

# ПОИСК ПО ТЕКСТУ

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).

## ПОИСК ПО ТЕКСТУ

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']"`.



# XPATН

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`.



# ХРАТН

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`:

```
1 By withText(String elementText)
2 By byText(String elementText)
3 By byAttribute(String attributeName, String attributeValue)
4 By by(String attributeName, String attributeValue)
5 By byTitle(String title)
6 By byValue(String value)
7 By byName(String name)
8 By byXPath(String xpath)
9 By byLinkText(String linkText)
10 By byPartialLinkText(String partialLinkText)
11 By byId(String id)
12 By byCssSelector(String css)
13 By byClassName(String className)
```

При этом игнорируются незначачие пробелы и другие «мусорные СИМВОЛЫ».

# ABTO-TEST

```
1  @Test
2  void shouldRegisterByAccountNumber() {
3      open("http://localhost:9999");
4      $(byText("По номеру счёта")).click();
5      $("[name='number']").setValue("4055 0100 0123 4613 8564");
6      $("[name='phone']").setValue("+792000000000");
7      $(byText("Продолжить")).click();
8      // TODO:
9  }
```

# 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*` ).

# Condition

```
1  @Test
2  void shouldRegisterByAccountNumber() {
3      open("http://localhost:9999");
4      $$(".tab-item").find(exactText("По номеру счёта")).click();
5      $("[name='number']").setValue("4055 0100 0123 4613 8564");
6      $("[name='phone']").setValue("+792000000000");
7      $$("button").find(exactText("Продолжить")).click();
8      ...
9  }
```

# 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:

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

# WAITING

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

```
1  @Test
2  void shouldRegisterByAccountNumber() throws InterruptedException {
3      open("http://localhost:9999");
4      $$(".tab-item").find(exactText("По номеру счёта")).click();
5      $("[name='number']").setValue("4055 0100 0123 4613 8564");
6      $("[name='phone']").setValue("+792000000000");
7      $$("button").find(exactText("Продолжить")).click();
8      $(withText("Успешная авторизация")).shouldBe(visible);
9      $(byText("Личный кабинет")).shouldBe(visible);
10 }
```

# VISIBLE VS EXISTS

**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).

## byText VS withText

**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

Element not found {with text: Успешная авторизация}

Expected: visible

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

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

Timeout: 4 s.



# SCREENSHOTS

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



**TIMEOUT**

# TIMEOUT

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`.

# TIMEOUT

**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

`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

```
1  @Test
2  void shouldRegisterByAccountNumber() throws InterruptedException {
3      open("http://localhost:9999");
4      $$(".tab-item").find(exactText("По номеру счёта")).click();
5      $("[name='number']").setValue("4055 0100 0123 4613 8564");
6      $("[name='phone']").setValue("+792000000000");
7      $$("button").find(exactText("Продолжить")).click();
8      $(withText("Успешная авторизация")).waitUntil(visible, 5000);
9      $(byText("Личный кабинет")).waitUntil(visible, 5000);
10 }
```



# TIMEOUT

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.





# **ИЗМЕНЕНИЕ ВИДИМОСТИ ЭЛЕМЕНТОВ**

## СКРЫТИЕ И ПОКАЗ ЭЛЕМЕНТОВ

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

Остался второй сценарий: скрывание и показ элементов.

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

# СКРЫТИЕ И ПОКАЗ ЭЛЕМЕНТОВ

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).

# СКРЫТИЕ И ПОКАЗ ЭЛЕМЕНТОВ

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

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

## РАБОТА СО СКРЫТЫМ ЭЛЕМЕНТОМ

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

```
1  @Test
2  void shouldRegisterByAccountNumber() throws InterruptedException {
3      open("http://localhost:9999");
4      $$("[name='number']").last().setValue("4055 0100 0123 4613 8564");
5      $("[name='phone']").setValue("+792000000000");
6      $$("button").find(exactText("Продолжить")).click();
7      ...
8  }
```

## РАБОТА СО СКРЫТЫМ ЭЛЕМЕНТОМ

```
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.

# РАБОТА СО СКРЫТЫМ ЭЛЕМЕНТОМ

```
1  @Test
2  void shouldRegisterByAccountNumber() {
3      open("http://localhost:9999");
4      $$(".tab-item").find(exactText("По номеру счёта")).click();
5      $$("[name='number']").last().setValue("4055 0100 0123 4613 8564");
6      $$("[name='phone']").last().setValue("+792000000000");
7      $$("button").find(exactText("Продолжить")).click();
8      $(withText("Успешная авторизация")).waitUntil(visible, 5000);
9      $(byText("Личный кабинет")).waitUntil(visible, 5000);
10 }
```



# ИТОГИ





# ИТОГИ

Сегодня мы рассмотрели несколько очень важных тем:

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.

## ПОДДЕРЖИВАЕМОСТЬ ТЕСТОВ

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 странице — 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`.

# ВНЕШНИЕ СИСТЕМЫ И ТЕСТОВЫЕ ДАННЫЕ

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).

Об этих проблемах мы и поговорим на следующей лекции.



# ДОМАШНЕЕ ЗАДАНИЕ

Давайте посмотрим ваше [домашнее задание](#).

- Вопросы по домашней работе задаем в чате Slack!
- Задачи можно сдавать по частям.
- Зачет по домашней работе проставляется после того, как приняты **все задачи**.



**Задавайте вопросы и напишите отзыв о лекции!**

**АРТЕМ РОМАНОВ**

 [romanov-artyom](#)

 [@Artem\\_Telegram](#)

 [Романов Артём](#)