

ТЕСТИРОВАНИЕ



ВЛАДИМИР ЯЗЫКОВ



ВЛАДИМИР ЯЗЫКОВ

Основатель UsefulWeb



neizerth@gmail.com



[@neizerth](https://t.me/neizerth)

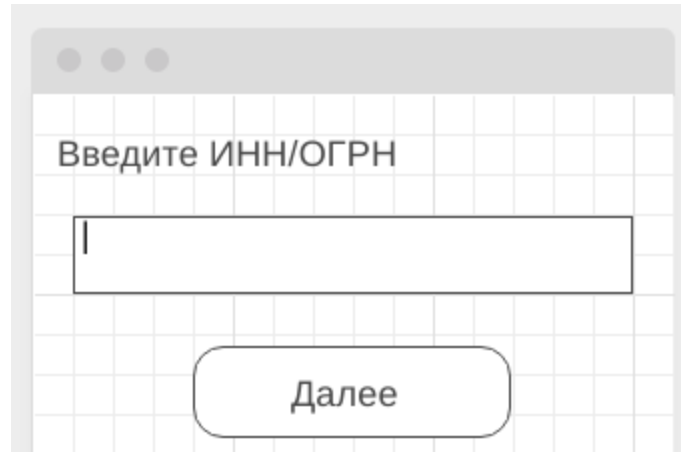


ПЛАН ЗАНЯТИЯ

1. [Пирамида тестирования](#)
2. [Unit-тестирование](#)
3. [E2E-тестирование](#)
4. [Компоненты](#)
5. [JSDOM](#)

ЗАДАЧА

Нужно реализовать форму для ввода и валидации ИНН/ОГРН, которая должна выглядеть следующим образом:

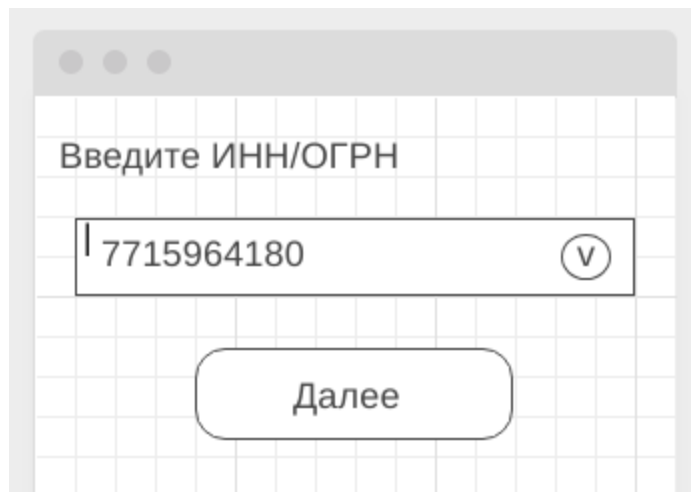


The image shows a mockup of a web form within a window frame. The window has a title bar with three standard macOS-style window control buttons (red, yellow, green). The form itself is set against a light gray grid background. At the top of the form, the text "Введите ИНН/ОГРН" is displayed in a dark gray font. Below this text is a single-line text input field with a thin black border. A small vertical cursor is visible at the beginning of the input field. At the bottom of the form is a rounded rectangular button with a thin black border and the text "Далее" centered inside it.

ВЛИДНОЕ ЗНАЧЕНИЕ

Валидация происходит при нажатии на кнопке «Далее».

При вводе валидного значения в поле формы появляется зелёная галочка:



Введите ИНН/ОГРН

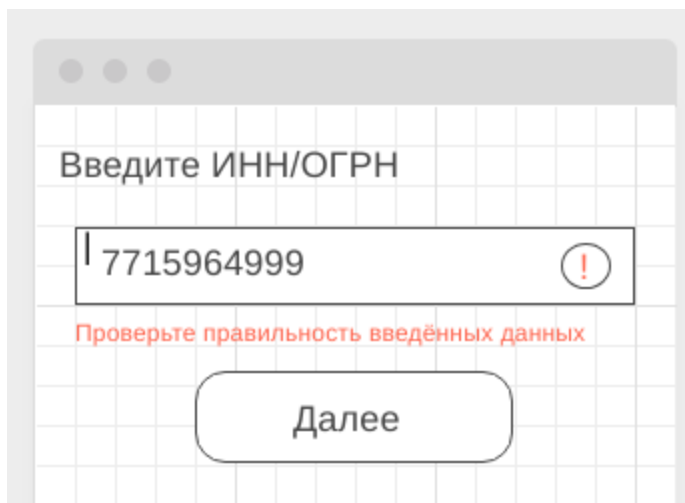
7715964180 ✓

Далее

The image shows a web form with a light gray grid background. At the top, there are three small gray circles. Below them is the text 'Введите ИНН/ОГРН'. Underneath is a text input field containing the value '7715964180'. To the right of the input field is a green checkmark icon inside a circle. Below the input field is a rounded rectangular button with the text 'Далее'.

НЕВАЛИДНОЕ ЗНАЧЕНИЕ

При вводе невалидного значения в поле формы появляется красный восклицательный знак и сообщение об ошибке под полем ввода:



Введите ИНН/ОГРН

7715964999

Проверьте правильность введенных данных

Далее

The image shows a web form with a light gray background and a white grid pattern. At the top, there are three small gray circles. Below them is a label 'Введите ИНН/ОГРН'. Under the label is a text input field containing the value '7715964999'. To the right of the input field is a red circle with a white exclamation mark. Below the input field is a red error message 'Проверьте правильность введенных данных'. At the bottom of the form is a rounded rectangular button with the text 'Далее'.

ВАЛИДАЦИЯ

Отображение значка на поле ввода можно реализовать с помощью добавления класса `.valid` / `.invalid` соответственно.

А отображение сообщения об ошибке с помощью добавления в DOM элемента с текстом.

РАЗМЕТКА

```
1 <form data-widget="innogrn-form-widget">
2   <div class="form-control">
3     <label for="innorgn-input">Введите ИНН/ОГРН</label>
4     <input id="innorgn-input" data-id="innogrn-input" type="text">
5   </div>
6   <button data-id="innogrn-submit">Далее</button>
7 </form>
```




СТРАТЕГИЯ РЕАЛИЗАЦИИ

Мы научились с вами взаимодействовать с DOM деревом и реагировать на происходящие события. Поэтому при реализации подобного приложения проблем возникнуть не должно.

Возникает вопрос: как это всё протестировать?



ОЧЕВИДНЫЙ ВАРИАНТ

Понятно, что можно «прокликать» приложение вручную, но данный способ достаточно трудоёмок, а хотелось бы получить автоматизированное решение, которое можно подключить к CI/CD.

Важно: в данной лекции мы рассматриваем только функциональное тестирование, оставляя за скобками все остальные виды тестирования.



АВТОМАТИЗИРОВАННОЕ VS. РУЧНОЕ


Не стоит думать, что ручное тестирование в прошлом, и нужно автоматизировать всё.

У автоматизации есть как достоинства, так и недостатки, так же, как и у ручного тестирования.



ВОПРОС

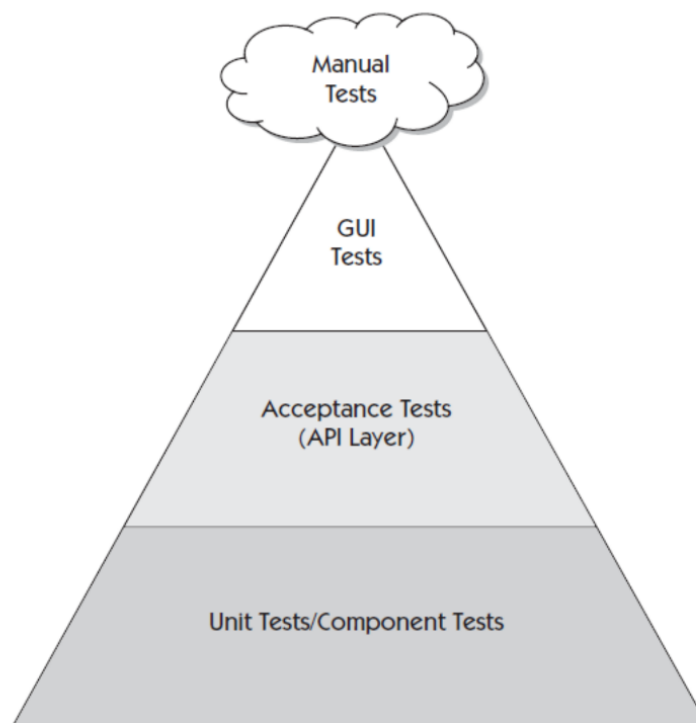
Какие предложения по автоматизации тестирования подобного приложения?



ПИРАМИДА ТЕСТИРОВАНИЯ

ПИРАМИДА ТЕСТИРОВАНИЯ

Авторы книги «Agile Testing» предлагают следующий взгляд на тестирование:



Agile Testing: A Practical Guide for Testers and Agile Teams

РАЗБИРАЕМСЯ С ТЕРМИНОЛОГИЕЙ

Manual Tests — тесты, выполняемые вручную.

GUI Tests — тестирование с точки зрения конечного пользователя.

API Tests — тестирование API (чаще всего говорят в терминах интеграции с серверной частью).

Unit Tests — изолированное тестирование функциональности.

Мы не будем вдаваться в жаркие споры о терминологии и об уровнях пирамиды. Тем более на просторах сети можно найти достаточно много вариаций этой пирамиды.

В РАМКАХ КУРСА

В рамках нашего курса в части именно автоматизации мы будем рассматривать два крайних случая:

1. E2E — управление браузером из тестов для имитации действий пользователя и тестирования работоспособности приложения в целом (с точки зрения конечного пользователя).
2. Unit — тестирование изолированной функциональности:
 - в идеальном варианте — чистых функций с минимумом побочных эффектов;
 - в неидеальном случае — с минимумом моков.

TEST RUNNER & ASSERTIONS LIBRARY

Нам нужен инструмент для запуска тестов и проверки результатов.

В качестве такого инструмента мы будем использовать Jest, напоминаем шаги по установке:

```
$ npm install --save-dev jest babel-jest @babel/core @babel/cli @babel/preset-env  
$ npm install core-js@3
```

В `.babelrc`:

```
{  
  "presets": [["@babel/preset-env", {  
    "useBuiltIns": "usage",  
    "corejs": 3  
  }]]  
}
```



UNIT-ТЕСТИРОВАНИЕ



ВОПРОС

Что можно выделить в качестве «чистых функций», для тестирования которых достаточно передачи входных параметров и проверки возвращаемого результата?

ЧИСТЫЕ ФУНКЦИИ

1. Функция по проверке ИНН:

```
isValidInn(value)
```

2. Функция по проверке ОГРН:

```
isValidOgrn(value)
```

3. Функция по проверке ИНН/ОГРН:

```
isValidInnOrOgrn(value)
```

Важно: последнюю функцию можно реализовать как последовательный вызов первых двух. Поскольку она будет использовать только чистые функции, то и сама будет чистой функцией.

ПРОЦЕСС ТЕСТИРОВАНИЯ

Тестирование таких функций достаточно просто:

1. Выносим их в отдельный модуль.
2. Проверяем с помощью стандартных проверок Jest.

```
1  test('it should return true for valid organization Inn', () => {  
2    const input = '7715964180';  
3  
4    expect(isValidInn(input)).to...;  
5  });
```

НЕБОЛЬШИЕ РАЗЛИЧИЯ В ТЕСТАХ

Если тесты отличаются только входными и ожидаемыми результатами:

```
1 test.each([
2   ['true for valid organization Inn', '7715964180', true],
3   ['false for invalid organization Inn', '7715964999', false],
4   ...
5 ])(('it should be %s'), (_, input, expected) => {
6   expect(isValidInn(input)).toBe(expected);
7 });
```



E2E-ТЕСТИРОВАНИЕ

E2E-ТЕСТИРОВАНИЕ

Для тестирования в реальном браузерном окружении мы будем использовать [Puppeteer](#).

```
npm install --save-dev puppeteer
```


JEST V24

На данный момент есть некоторые проблемы интеграции новой версии Jest (v24) с Puppeteer (с использованием специального пакета `jest-puppeteer`), поэтому мы будем интегрировать их напрямую.

ПОРТ ДЛЯ WEBPACKDEVSERVER

Зафиксируем порт для Webpack DevServer (чтобы затем его можно было использовать в тестах) - файл `webpack.config.js`:

```
1 devServer: {  
2   port: 9000,  
3 },
```

ИМПОРТ И НАСТРОЙКА

```
1 import puppeteer from 'puppeteer';
2 jest.setTimeout(30000); // default puppeteer timeout
3 describe('INN/OGRN form', () => {
4   let browser = null;
5   let page = null;
6   const baseUrl = 'http://localhost:9000';
7   beforeAll(async () => {
8     browser = await puppeteer.launch({
9       headless: false, // show gui
10      slowMo: 100,
11      devtools: true, // show devTools
12    });
13    page = await browser.newPage();
14  });
15  afterAll(async () => {
16    await browser.close();
17  });
18  // test code here (next page)
19 });
```

Опции в методе `launch` нужно закомментировать при запуске в CI.

ТЕСТЫ

```
1 describe('INN/OGRN form', () => {
2     test('should add .valid class for valid inn', async () => {
3         await page.goto(baseUrl);
4         const form = await page.$('[data-widget=innogrn-form-widget]');
5         const input = await form.$('[data-id=innogrn-input]');
6         await input.type('7715964180');
7         const submit = await form.$('[data-id=innogrn-submit]');
8         submit.click();
9         await page.waitForSelector('[data-id=innogrn-input].valid');
10    });
11 });
```

ПОВЕДЕНИЕ VS. РЕАЛИЗАЦИЯ

В большинстве случаев можно начать долгую дискуссию по вопросу, можно ли считать добавления класса поведением или это всё-таки реализация.

И, возможно, стоит выносить состояние компонента (ошибка валидации) в `data-*`, либо «добираться» каким-либо образом до самого объекта компонента и проверять его.

Наша задача — обратить ваше внимание, что в большинстве случаев стоит тестировать именно поведение, а не конкретную реализацию (например, не текст сообщения об ошибке, размер шрифта и цвет). Хотя и в данном случае возможны разные точки зрения.



ДОКУМЕНТАЦИЯ

Ключевые классы:

- [Browser](#)
- [Page](#)
- [ElementHandle](#)

РАЗДЕЛЕНИЕ ТЕСТОВ

Положим наши тесты в каталог `test` и запустим через `npm test`:

```
1  "scripts": {  
2    "test": "jest --testPathPattern=./src/",  
3    "coverage": "jest --testPathPattern=./src/ --coverage",  
4    "e2e": "jest --testPathPattern=./e2e/"  
5  }
```



ЗАПУСК

```
npm run e2e
```




РАСШИРЕНИЕ ГРАНИЦ ДОЗВОЛЕННОГО

Границы между инструментами размываются, поэтому тот же Puppeteer умеет, например, перехватывать HTTP-запросы, что позволяет его использовать не только как инструмент E2E-тестирования.

МИНУСЫ PUPPETEER

Несмотря на то, что в headless-режиме тесты проходят достаточно быстро, есть ряд минусов:

1. Приходится тестировать страницу целиком.
2. Приходится постоянно держать запущенным сервер*.
3. Долгое время запуска.

* На самом деле настройку и запуск сервера обычно выносят в `setup` Jest'a и после старта сервера уже запускают тесты. Мы для упрощения этот момент не рассматриваем.



ПЛЮСЫ PUPPETEER

Ключевой плюс — тестирование в реальном окружении (никаких моков).



КОМПОНЕНТЫ



ИДЕЯ КОМПОНЕНТОВ

Поскольку писать всё приложение в виде одного скрипта — плохая идея, хотелось бы инкапсулировать логику формы (включая взаимодействие с DOM) в отдельном классе.

Это позволило бы переиспользовать этот класс, но остаются два вопроса:

1. Как тестировать?
2. Как постоянно следить за «актуальностью» разметки на всех страницах?

ТЕСТИРОВАНИЕ

Puppeteer позволяет нам работать через

```
page.goto(data:text/html,-content here-).
```

Но как запустить наш скрипт? Можно, конечно, использовать

```
page.evaluate
```

, но хотелось бы решения покрасивее.

Отказаться от Puppeteer и использовать моки? Но тогда не слишком ли много придётся писать моков?



АКТУАЛЬНОСТЬ РАЗМЕТКИ

А что если переложить создание необходимой для компонента разметки на плечи самого компонента?

Тогда останется только реализовать привязку к определённому DOM-элементу, в который наш компонент и будет отображаться.

COMPONENT

```
1 export default class InnOgrnFormWidget {
2   constructor(parentEl) {
3     this.parentEl = parentEl;
4   }
5
6   static get markup() {
7     return `
8     <form data-widget="innogrn-form-widget">
9       <div class="form-control">
10         <label for="innorgn-input">Введите ИНН/ОГРН</label>
11         <input id="innorgn-input" data-id="innogrn-input" type="text">
12       </div>
13       <button data-id="innogrn-submit">Далее</button>
14     </form>
15     `;
16   }
17
18   static get inputSelector() {
19     return '[data-id=innogrn-input]';
20   }
21   ...
22 }
```


COMPONENT

```
1  export default class InnOgrnFormWidget {
2    ...
3
4    static get submitSelector() {
5      return '[data-id=innogrn-submit]';
6    }
7
8    bindToDOM() {
9      this.parentEl.innerHTML = this.constructor.markup;
10     const submit = this.parentEl.querySelector(this.constructor.submitSelector);
11     submit.addEventListener('click', evt => this.onSubmit(evt));
12   }
13
14   onSubmit(evt) {
15     evt.preventDefault();
16     const inputEl = this.parentEl.querySelector(this.constructor.inputSelector);
17     if (isValidInn(inputEl.value)) {
18       inputEl.classList.add('valid');
19     } else {
20       inputEl.classList.add('invalid');
21     }
22   }
23 }
```



ТЕСТИРОВАНИЕ

Перейдя к подобной концепции компонентов, можно внедрить промежуточный вариант тестирования, когда мы будем тестировать только функциональность данного компонента, не рассматривая всю страницу целиком.

В рамках данного курса мы подобное тестирование будем называть «компонентным», подразумевая взаимодействие конкретного компонента с DOM.

Осталось подобрать инструмент для тестирования.



JSDOM

JSDOM



“A JavaScript implementation of the WHATWG DOM and HTML standards, for use with Node.js”

Готовая реализация API (в том числе для целей тестирования).

Позволяет отказаться от необходимости писать моки.

JSDOM

Вольный перевод:



JavaScript-реализация WHATWG-стандарта DOM и HTML для использования с Node.js

JSDOM И JEST

JSDOM уже идёт в составе Jest. Поэтому мы можем использовать его без предварительной настройки*, так как она произведена в пакете `jest-environment-jsdom`.

Ключевое замечание, которое стоит сделать: Jest использует версию JSDOM 11 для обеспечения совместимости.

* Если вы не настраивали другой `testEnvironment`.

ГЛОБАЛЬНЫЕ ОБЪЕКТЫ

В рамках интеграции с Jest нам предоставляются глобальные объекты `document` и `window`, с которыми мы можем взаимодействовать:

```
1  test('should render self', () => {
2    document.body.innerHTML = '<div id="container"></div>';
3
4    const container = document.querySelector('#container');
5    const widget = new InnOgrnFormWidget(container);
6
7    widget.bindToDOM();
8
9    expect(container.innerHTML).toEqual(InnOgrnFormWidget.markup);
10 });
```

JSDOM INPUT

Генерировать собственные события мы пока не умеем, поэтому будем устанавливать значение через свойство `value`:

```
1  test('should validate input', () => {
2    document.body.innerHTML = '<div id="container"></div>';
3
4    const container = document.querySelector('#container');
5    const widget = new InnOgrnFormWidget(container);
6
7    widget.bindToDOM();
8
9    const input = container.querySelector(InnOgrnFormWidget.inputSelector);
10   input.value = '7715964180';
11
12   const submit = container.querySelector(InnOgrnFormWidget.submitSelector);
13   submit.click();
14
15   expect(input.classList.contains('valid')).toBeTruthy();
16 });
```




ВОПРОСЫ

1. Из какого интерфейса берётся метод `click`?
2. Какие ещё там есть методы?



JSDOM & MOCKS

В зависимости от логики вашего теста, вы можете по-прежнему устанавливать Mock'и на необходимые функции и проверять только интеграцию вашей логики и DOM API.

E2E VS. UNIT VS. COMPONENT

Почему всё не покрыть E2E-тестами, ведь они максимально показывают «правдоподобный результат».

Стоимость и скорость. Мы идём на компромиссы для уменьшения стоимости и увеличения скорости.



ИТОГИ



ИТОГИ

1. Повторили Unit-тесты.
2. Поговорили о E2E-тестах на базе Puppeteer.
3. Посмотрели на использование JSDOM.



ТРЕБОВАНИЯ К ДЗ

Ко всем лекциям по прежнему будет требоваться unit-тестирование только функций и объектов, не взаимодействующих с DOM.

E2E тестирование через Puppeteer и тестирование с использованием JSDOM будет требоваться только в ряде лекций (чтобы облегчить вам выполнение ДЗ).



Задавайте вопросы и напишите отзыв о лекции!

ВЛАДИМИР ЯЗЫКОВ



neizerth@gmail.com



[@neizerth](https://www.instagram.com/neizerth)