

Puppeteer

Часть 2

Василий Петров
Разработчик Python, JavaScript



Василий Петров

О спикере:

- Стаж работы в IT более 25 лет
- Разрабатывал корпоративные приложения
- Руководил проектами и IT-подразделениями
- Руководил собственным бизнесом
- Участвует в различных проектах с применением Python и JavaScript



Цели занятия

- 1 Узнаем, что такое кастомные функции, для чего они нужны
- 2 Разберёмся, чем нам помогут `.skip`, `.only`
- 3 Научимся генерировать данные без сторонних библиотек
- 4 Исследуем реализацию BDD с применением Cucumber

План занятия

- 1 [Тест со вводом текста](#)
- 2 [Custom commands](#)
- 3 [Генерация случайных данных](#)
- 4 [Cucumber + Puppeteer](#)
- 5 [Итоги](#)
- 6 [Домашнее задание](#)

Тест со вводом текста

1

Добавим ещё один тест со вводом текста

В дополнение к существующим тестам, напишем тест для выбора курса:

```
test("Should look for a course", async () => {  
  await page.goto("https://netology.ru/navigation");  
  const inputField = await page.$("input");  
  await page.waitForSelector("h1");  
  await inputField.focus();  
  await inputField.type("Тестировщик");  
  await page.keyboard.press("Enter");  
  const actual = await page.$eval("a[data-name]", (link) => link.textContent);  
  const expected = "Тестировщик ПО";  
  expect(actual).toContain(expected);  
});
```



Работа с конкретными тестами

Для упрощения работы с конкретными тестами мы можем использовать методы `.skip`, `.only`. Они позволят нам пропускать тесты или запускать конкретные тесты в режиме headed.

Во время работы над нашим тестом добавим `.only` и проверим, что этот тест работает так, как ожидается, а также отработывает верно, когда мы его сломаем:

```
test.only("Should look for a course", async () => {  
  ...  
});
```

Можно использовать как для `test()`, так и для `describe()`



Вопрос опытным тестировщикам

Посмотрев на наши тесты, что вы
предложите здесь улучшить?

Вопрос опытным тестировщикам

Можно избавиться от повторяющегося
кода, сделав тесты лучше читаемыми,
более стабильными и эффективными

Custom commands



2

Необходимость в кастомных командах

У нас становится всё больше тестов, они становятся всё сложнее для чтения, а также всё чаще повторяется код.

Следуя принципу DRY, мы можем вынести повторяющийся код в отдельные функции и организовать свою библиотеку кастомных команд (функций).

Для этого создадим в нашем проекте папку **/lib**.

В ней создадим файл для наших кастомных команд **commands.js**

Организуем экспорт кастомных команд

Все наши кастомные команды мы поместим в блок экспорта, чтобы они были доступны в тестах:

```
module.exports = {  
  
  }  
}
```



Добавляем кастомные команды

Первая команда — самая частая в UI тестах — клик по элементу.
На вход функция будет принимать сущность страницы и селектор:

```
module.exports = {  
  clickElement: async function(page, selector) {  
  }  
}
```



Добавляем обработку ошибок при выполнении команды

Если наш тест упадёт, нам важно быстро разобраться в причине и месте, где что-то пошло не так.

Чтобы облегчить наше с вами существование, рекомендуется добавлять обработку ошибок в наши кастомные функции, используя блок **try/catch**:

```
module.exports = {  
  clickElement: async function(page, selector) {  
    try {  
  
    } catch (error) {  
  
    }  
  }  
}
```



Добавляем обработку ошибок при выполнении команды

В блок `try` помещаем код действия, а в блок `catch` — сообщение, которое будет выведено в логах при ошибке во время выполнения этой команды:

```
try {  
    await page.waitForSelector(selector);  
    await page.click(selector);  
} catch (error) {  
    throw new Error(`Selector is not clickable: ${selector}`)  
}
```




Не забываем про автодополнение кода

Не забывайте пользоваться преимуществами среды разработки.
Используйте автодополнение кода:


try


 try

try

 trycatch

Try-Catch Stat...

 TypeError

 MimeTypeArray



Используем кастомную команду в тесте

Импортируем кастомную команду в тесты. Добавим следующую строку в файл с тестом:

```
const {clickElement} = require("../lib/commands.js");
```

Теперь мы можем использовать новую кастомную команду вместо старой имплементации клика на элемент:

```
await clickElement("header a + a");
```

Запустим снова наши тесты



Добавляем кастомные команды

Добавим ещё несколько полезных кастомных команд:

```
getText: async function (page, selector) {  
  try {  
    await page.waitForSelector(selector);  
    return await page.$eval(selector, (link) => link.textContent);  
  } catch (error) {  
    throw new Error(`Text is not available for selector: ${selector}`);  
  }  
}
```



Добавляем кастомные команды

Добавим ещё несколько полезных кастомных команд:

```
putText: async function (page, selector, text) {  
  try {  
    const inputField = await page.$(selector);  
    await inputField.focus();  
    await inputField.type(text);  
    await page.keyboard.press("Enter");  
  } catch (error) {  
    throw new Error(`Not possible to type text for selector: ${selector}`);  
  }  
}
```



Используем кастомные команды в тестах

Импортируем новые кастомные команды в тесты:

```
const {clickElement, putText, getText} = require("../lib/commands.js");
```

Заменяем кастомными командами соответствующий код в наших тестах:

```
const firstLinkText = await getText(page, "header a + a");  
...  
const actual = await getText(page, "h1");  
...  
await getText(page, "input", "тестировщик")  
...  
const actual = await getText(page, "a[data-name]");
```



Генерация рандомных данных



3

Необходимость в генерации

Почему нам необходимо генерировать данные самим, а не использовать faker, как мы делали ранее?



Необходимость в генерации

Почему нам необходимо генерировать данные самим, а не использовать faker, как мы делали ранее?

Иногда невозможно использовать сторонние библиотеки по ряду причин, поэтому приходится самим писать подобные утилиты.

Рассмотрим некоторые примеры



Генерируем строку

Добавим тест, в котором нам необходимо внести в поле логина строку из символов, чтобы проверить, что будет выдаваться ошибка в случае, если это не email:

```
test("Should show warning if login is not email", async () => {  
  await page.goto("https://netology.ru/?modal=sign_in");  
  
  await putText(page, 'input[type="email"]', "текст")  
});
```

Заменим строку «текст» на случайную строку



Напишем вспомогательную утилиту

Организуем место для вспомогательных утилит.

В папку `/lib` добавим файл `util.js`, где будем хранить наши вспомогательные утилиты.

Добавим экспорт и начнём писать функцию, которая возвращает рандомную строку:

```
module.exports = {  
  generateName: function(length) {  
    ...  
    return name;  
  }  
}
```



Напишем вспомогательную утилиту

Для реализации нам понадобится глобальный объект Math, который работает с математическими операциями:

```
generateName: function(length) {  
  let name = ""; //здесь будем хранить результат  
  let chars = 'abcdefgABCDEFGFG1234567890'; //возможные символы  
  let charLength = chars.length; //определяем длину  
  for(let i = 0; i < length; i++) { //запускаем цикл для формирования строки  
    name += chars.charAt(Math.floor(Math.random() * charLength))  
  }  
  return name;  
}
```



Генерируем строку

Импортируем новую функцию в тесты и добавляем её в нужное место:

```
test("Should show warning if login is not email", async () => {  
  await page.goto("https://netology.ru/?modal=sign_in");  
  
  await putText(page, 'input[type="email"]', generateName())  
  //assertions  
});
```

Запускаем тест



Перерыв



Cucumber + Puppeteer



4

BDD с Cucumber

Вероятно, что на некоторых проектах могут использоваться техники BDD на базе **Cucumber** и **Gherkin**.

С подходом BDD (Behavior Driven Development) мы уже знакомы с прошлого курса по автоматизации.

Рассмотрим BDD в стеке Puppeteer и Cucumber.

Для начала решим, какой тест будем автоматизировать при помощи Cucumber



Задача

Хорошим кандидатом является тест с поиском курса:

```
test("Should look for a course", async () => {  
  await page.goto("https://netology.ru/navigation");  
  
  await putText(page, "input", "тестировщик");  
  
  const actual = await getText(page, "a[data-name]");  
  const expected = "Тестировщик ПО";  
  expect(actual).toContain(expected);  
});
```



Задача

Важно: в рамках курса мы изучаем подход BDD и выбираем тест для имплементации, не опираясь на бизнес-потребности. В реальной жизни мы обязаны использовать те или иные технологии в соответствии с потребностью бизнеса



Подключение Cucumber

Подключаем зависимости к нашему проекту. Также добавим библиотеку ассершенов Chai:

```
npm install cucumber  
npm install chai
```

Добавим скрипт для запуска тестов Cucumber в package.json:

```
"scripts": {  
  "test": "jest --watchAll --detectOpenHandles"  
  "cucumber": "cucumber-js"  
}
```



Подключение Cucumber

Добавляем папку **/features** для организации файлов Cucumber.

Внутри добавляем папку **/step_definitions** для реализации шагов.

Создаём первый файл с названием в формате ***.feature**



Подключение Cucumber: расширения

Для лучшей визуализации и удобной работы подключим расширения:



Cucumber (Gherkin) Full Support 2.15.1

VSCode Cucumber (Gherkin) Full Language Support + Fo...

Alexander Krechik



vscode-icons 10.1.1

Icons for Visual Studio Code

VSCode Icons Team



Подключение Cucumber: расширения

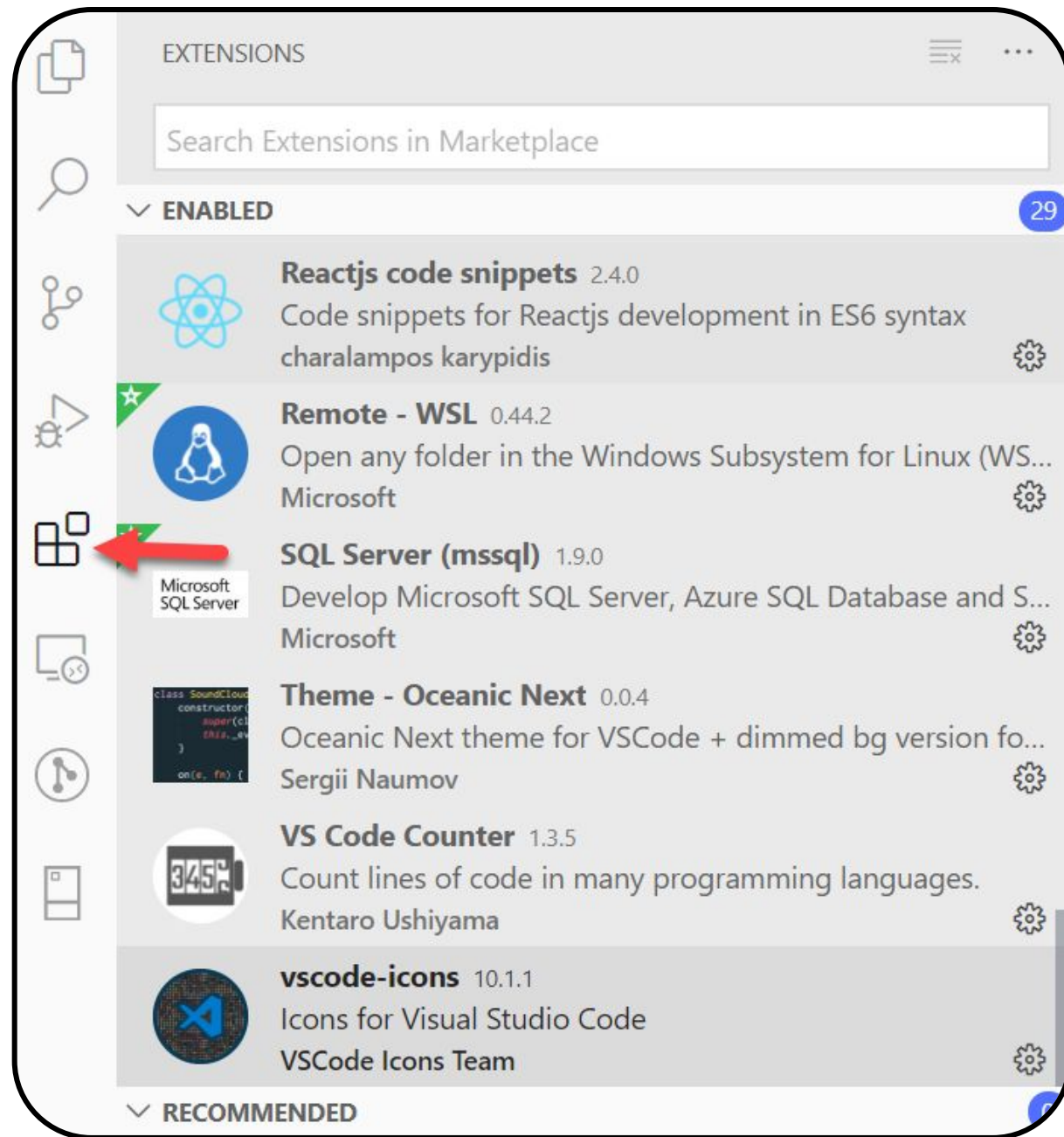
Есть несколько способов установки расширений:

- из VS Code (Ctrl+Shift+x) или меню Extensions
- с веб-сайта расширения
- из командной строки



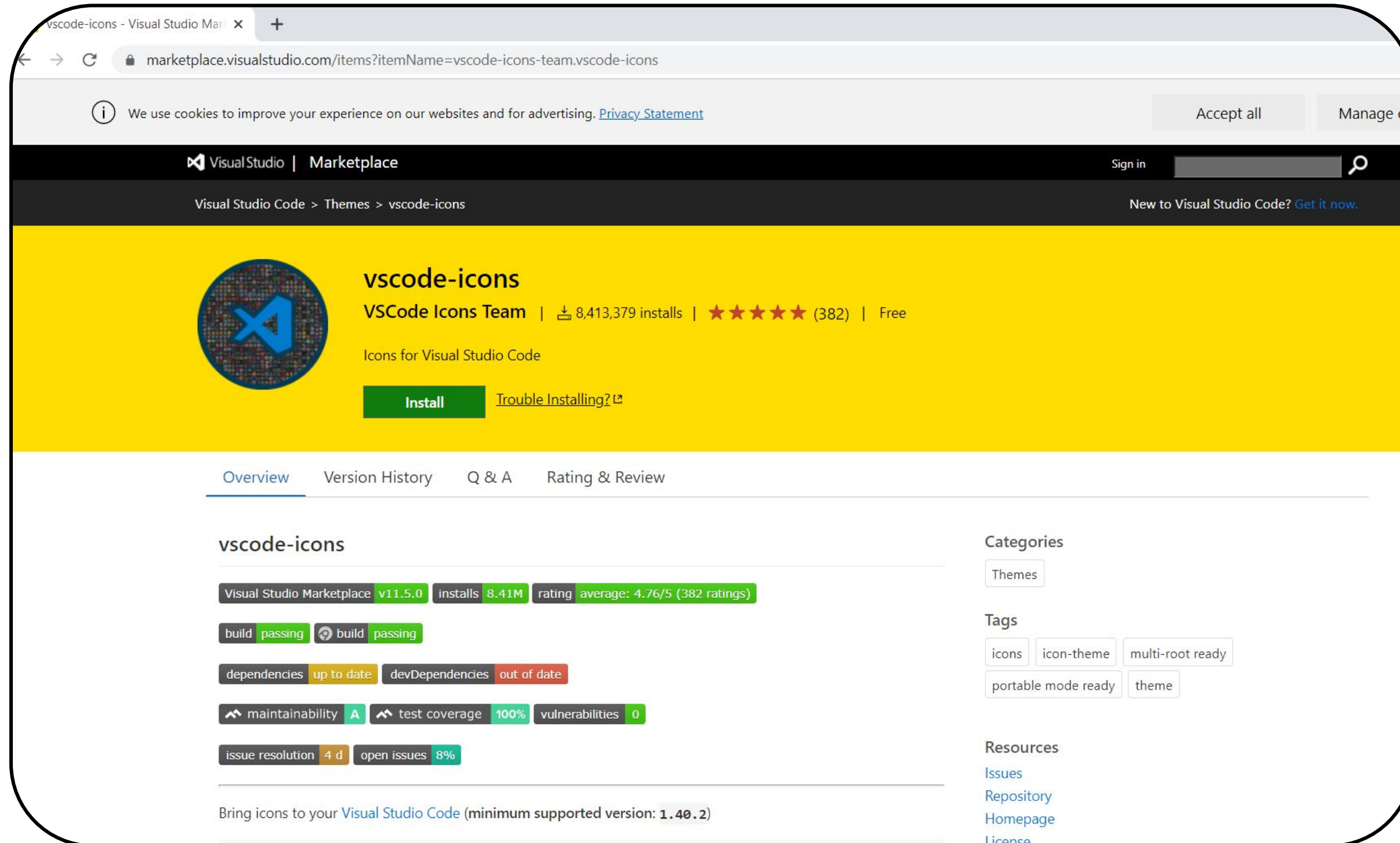
Подключение Cucumber: расширения

Из VS Code (Ctrl+Shift+x) или меню Extensions:



Подключение Cucumber: расширения

С веб-сайта расширения:



Подключение Cucumber: расширения

Из командной строки:

```
code --install-extension vscode-icons-team.vscode-icons
```

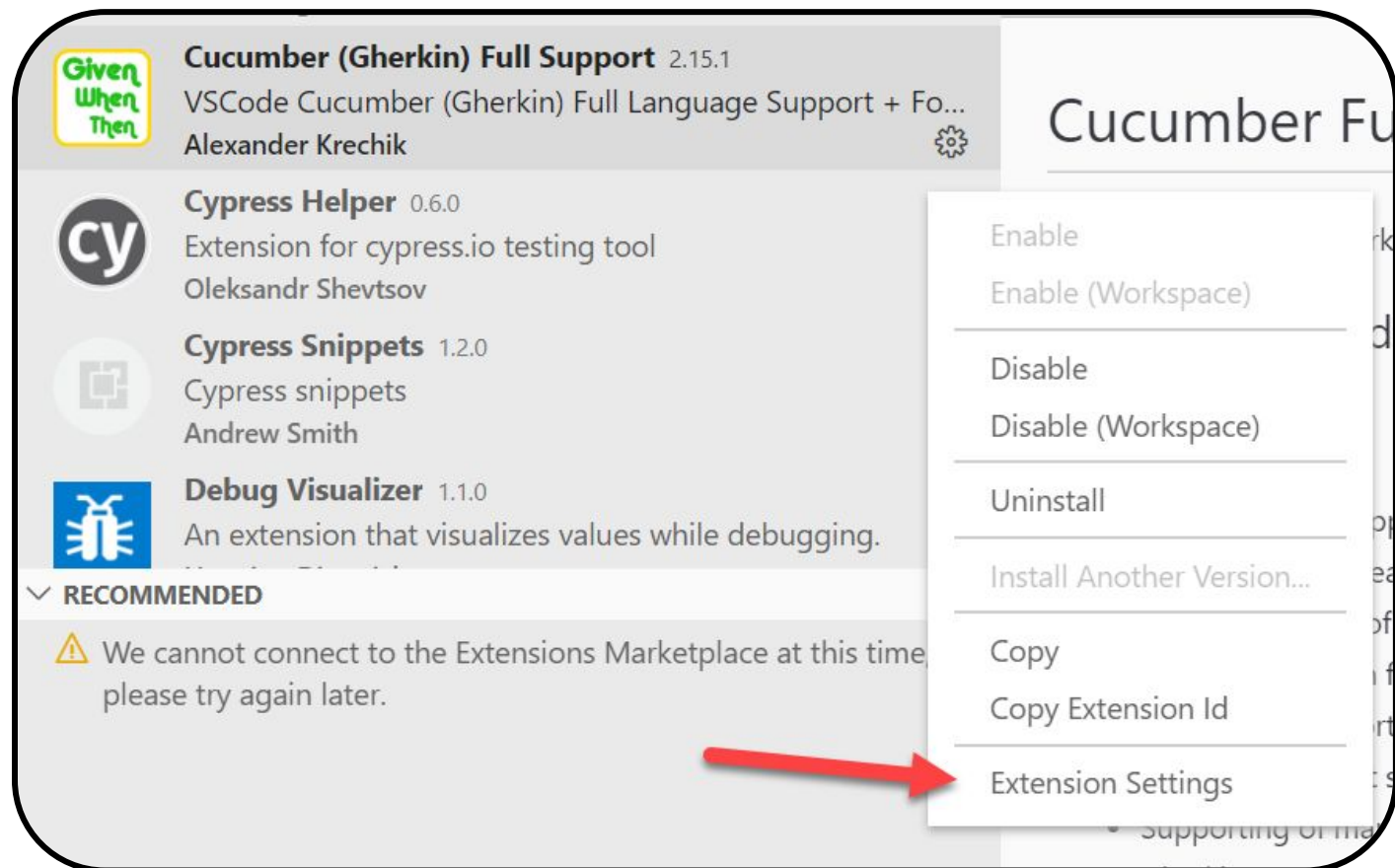


Подключение Cucumber: расширения

Подключим расширения разными способами:

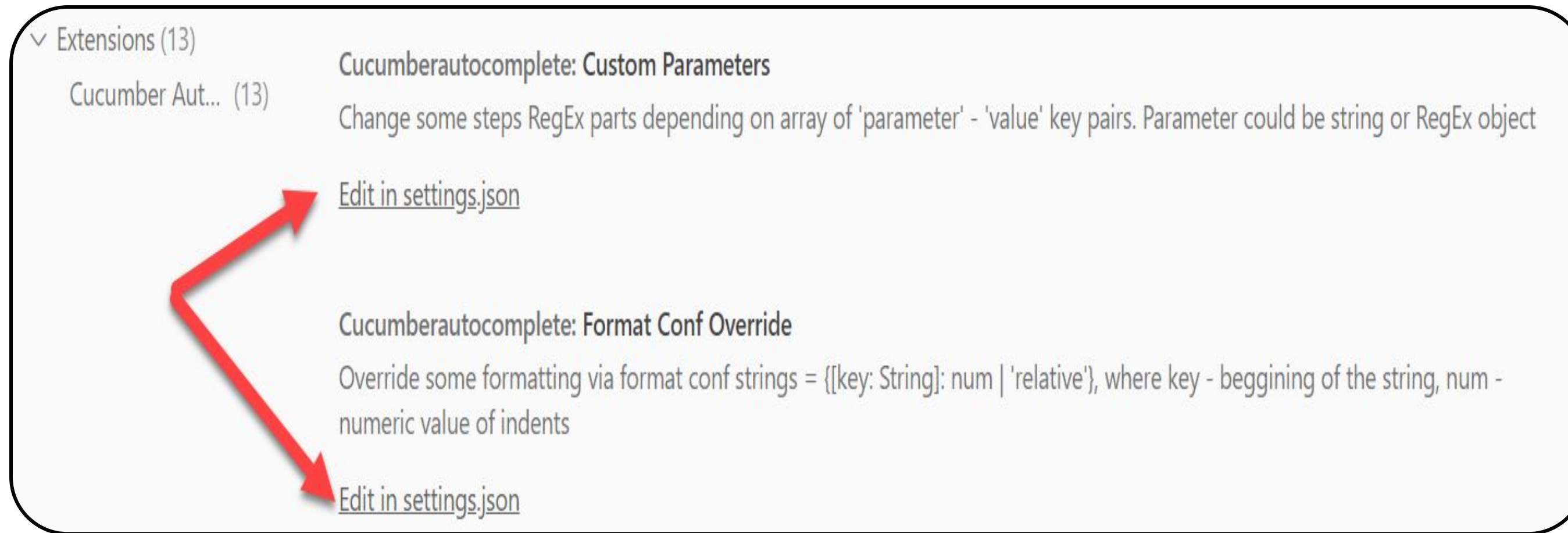


При необходимости мы можем изменить настройки расширений в меню Extensions:



Подключение Cucumber: расширения

Настройки расширений и всего проекта хранятся в файле **settings.json**:



Также открыть настройки можно, используя **Ctrl+Shift+p -> Open Settings**.

Показать **settings.json**



Подключение Cucumber: добавляем настройки

Добавим некоторые настройки:

- папка, где будем реализовывать шаги:

```
"cucumberautocomplete.steps": "./features/step_definitions/*.js"
```

- сделаем доступ к нашим шагам более гибким. Нам будут предлагаться шаги даже в случае несовпадения ключевых слов:

```
"cucumberautocomplete.strictGherkinCompletion": false
```

Больше настроек можно исследовать [здесь](#)



Первый тест Cucumber

Теперь мы готовы написать наш первый тест Cucumber.
В файле `search.feature` создаём сценарий:

Feature: Search a course

Scenario: Should search by text

Given user is on `"/navigation"` page

When user search by `"тестировщик"`

Then user sees the course suggested `"Тестировщик ПО"`



Первый тест Cucumber

Запустим тест командой, которую мы добавили в package.json:

```
npm run cucumber
```



Первый тест Cucumber

Получаем ошибку, так как добавленные нами шаги не привязаны к тестовым методам:

? Given user is on "/navigation" page

Undefined. Implement with the following snippet:

```
Given('user is on {string} page', function (string) {  
  // Write code here that turns the phrase above into concrete actions  
  return 'pending';  
});
```

...



Первый тест Cucumber

Воспользуемся шаблоном, который предложил нам Cucumber, и создадим тестовые методы для шагов.

Для этого добавим файл `/step_definitions/search.step.js`.

Импортируем необходимые зависимости

```
const {Given, When, Then, Before, After} = require('cucumber');  
const puppeteer = require('puppeteer');  
const expect = require('chai');
```



Первый тест Cucumber

Добавим предложенные шаблоны для имплементации шагов, не забывая добавить `async` для функции:

```
Given("user is on {string} page", async function (string) {  
  // Write code here that turns the phrase above into concrete actions  
  return "pending";  
});  
When("user search by {string}", async function (string) {  
  // Write code here that turns the phrase above into concrete actions  
  return "pending";  
});  
Then("user sees the course suggested {string}", async function (string) {  
  // Write code here that turns the phrase above into concrete actions  
  return "pending";  
});
```



Первый тест Cucumber

Перенесём сюда наши проверки:

```
Given("user is on {string} page", async function (string) {  
    return await page.goto(`https://netology.ru${string}`);  
});  
  
When("user search by {string}", async function (string) {  
    return await putText(page, "input", string);  
});  
  
Then("user sees the course suggested {string}", async function (string) {  
    const actual = await getText(page, "a[data-name]");  
    expect(actual).contain(string);  
});
```



Первый тест Cucumber

Добавим также хуки Before и After:

```
Before(async function () {  
  const browser = await puppeteer.launch({ headless: false, slowMo: 50 });  
  const page = await browser.newPage();  
  this.browser = browser;  
  this.page = page;  
  
});  
  
After(async function () {  
  if (this.browser) {  
    await this.browser.close();  
  }  
  
});
```

Cucumber работает с контекстом, поэтому, чтобы обращаться к текущей сущности page или браузеру, мы должны использовать ключевое слово `this`, указывая, что мы работаем внутри этого класса



Первый тест Cucumber

Добавляем this в тесты, в те места, где работаем со страницей:

```
Given("user is on {string} page", async function (string) {  
  return await this.page.goto(`https://netology.ru${string}`, {setTimeout: 10000});  
});  
  
When("user search by {string}", async function (string) {  
  return await putText(this.page, "input", string);  
});  
  
Then("user sees the course suggested {string}", async function (string) {  
  const actual = await getText(this.page, "a[data-name]");  
  const expected = string;  
  expect(actual).contain(expected);  
});
```



Итоги

- 1 Познакомились с комбинациями Puppeteer с другими фреймворками
- 2 Узнали, как реализовать BDD
- 3 Научились реализовывать кастомные функции
- 4 Разобрались с функциями .skip, .only
- 5 Поработали с Cucumber

Домашнее задание

Давайте посмотрим ваше домашнее задание:

- Вопросы по домашней работе задавайте в чате группы
- Задачи можно сдавать **по частям**
- Зачёт по домашней работе проставляется после того, как приняты **все задачи**

**Задавайте вопросы
и пишите отзывы
о лекции**

