Wybrane elementy praktyki projektowania oprogramowania

Wykład 15/15
Testowanie

Wiktor Zychla 2023/2024

1 Spis treści

2 Testy jednostkowe

Testowanie kodu pozwala wykrywać błędy implementacji jeszcze przed wdrożeniem. Szczególnymi testami są testy jednostkowe czyli testy kodu, które są wyrażone w kodzie. Przygotowanie testów jednostkowych to próba odtworzenia takich przepływów kontroli w testowanym kodzie, które mogą wystąpić w rzeczywistości. Dobre testy jednostkowe charakteryzują się dużym tzw. *pokryciem* kodu, czyli uwzględnieniem możliwe dużej liczby możliwych danych testowych, na których kod wykonuje się różnymi ścieżkami wykonania.

W przypadku Node.js jest bardzo wiele frameworków wspierających testowanie. Jednym z nich jest mocha.

Framework najwygodniej zainstalować globalnie

```
npm install mocha -g
```

a następnie w podfolderze **test** dodać jeden lub wiele plików, które mocha wykona jako testy.

Przykład

```
var assert = require('assert');

function add() {
  return Array.prototype.slice.call(arguments).reduce(function(prev, curr) {
    return prev + curr;
  }, 0);
}

describe('Top level', function() {

  before(function() {
    // runs before all tests in this block
  });

  after(function() {
    // runs after all tests in this block
  });

  beforeEach(function() {
    // runs before each test in this block
  });

  afterEach(function() {
    // runs after each test in this block
  });

    describe('add()', function() {
        var tests = [
            {args: [1, 2],       expected: 3},
            {args: [1, 2, 3],    expected: 6},
```

```
            {args: [1, 2, 3, 4], expected: 10}
        ];

        tests.forEach(function(test) {
            it('correctly adds ' + test.args.length + ' args', function() {
            var res = add.apply(null, test.args);
            assert.equal(res, test.expected);
            });
        });
    });


    describe('Sublevel', function() {
        it( 'unit test function definition', function() {
            assert.equal( -1, [1,2,3].indexOf(4) );
        });
    });
});
```

Dla wygodny można w **package.json** w sekcji scripts zmapować klucz **test** na polecenie **mocha:**

```
  "scripts": {
    "test": "mocha"
  }
```

Wynik działania testu (czyli wykonania **npm test**)

```
Top level

  add()
    √ correctly adds 2 args
    √ correctly adds 3 args
    √ correctly adds 4 args
  Sublevel
    √ unit test function definition


  4 passing (29ms)
```

3 Testy aplikacji

Weźmy najprostszą aplikację:

```
var http = require('http');
var express = require('express');

var app = express();

app.set('view engine', 'ejs');
app.set('views', './views');

app.use( express.urlencoded({extended: true}));

app.get("/", (req, res) => {
    res.render("index", {email: ''});
});

app.post("/", (req, res) => {
    var email = req.body.email;
    sendEmail( email );
    res.render("index", {email})
})

app.listen(3000, () => {
    console.log('listening');
});

function sendEmail(email) {
    // tu jakieś efekty uboczne, wysłanie maila
    console.log( `email sent to ${email}`);
}
```

z jednym widokiem

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>

    <form method="post">
```

```
        <label>Email</label>
        <input name="email" value="<%= email %>" />
        <button>Send</button>

    </form>

</body>
</html>
```

To co istotne dzieje się tu we fragmencie

```
app.post("/", (req, res) => {
    var email = req.body.email;
    sendEmail( email );
    res.render("index", {email})
})
```

funkcja odczytuje dane wprowadzone przez użytkownika

- Funkcja odczytuje dane wprowadzone przez użytkownika (wejście)
- Funkcja używa danych z wejścia do wysłania maila (tu: informacja na konsoli ale wyobraźmy sobie że to jest wysłanie maila)

Jak napisać test jednostkowy dla takiej konstrukcji?

Problemy:

1. **app.post** w ogóle nie nadaje się bezpośrednio do testowania
2. callback podawany jako parametr w **app.post** zależy od obiektów **request** i **response**
3. sam callback odwołuje się do logiki wyjścia która ma skutki uboczne (wysłanie maila) i w teście jednostkowym te skutki mogą być niedopuszczalne

Rozwiązanie: **architektura portów i adapterów**

Wejście: port wejścia, odseparowany od **request** i **response**

Wyjście: dla każdego wyjścia ze skutkami ubocznymi należy mieć mechanizm jego podmiany na jego *zastępnik* którego można użyć w teście jednostkowym

3.1 Refaktoryzacja wejścia

Refaktoryzujmy wejście otaczając kod aplikacji dodatkową funkcją/obiektem, przyjmującym jawną listę parametrów:

```
function rootPost( email ) {
```

```
    sendEmail( email );
}
```

i wywołujmy taką funkcję z callbacka

```
app.post("/", (req, res) => {
    var email = req.body.email;
    rootPost( email );
    res.render("index", {email})
})
```

Zmiana jest pozornie nieduża, ale krytyczna dla architektury testów:

1. Testom jednostkowym zostanie poddana funkcja **rootPost**
2. Gdyby była potrzeba przekazania jej większej liczby parametrów lub gdyby ona zwracała wyniki – to zostanie zmodyfikowana
3. Dla każdego skonfigurowanego na poziomie aplikacji express punktu końcowego i dla każdego rodzaju żądania (GET/POST) można utworzyć taką osobną funkcję, której potem można użyć w teście jednostkowym

3.2 Refaktoryzacja wyjścia

Refaktoryzacja wyjścia jest trudniejsza. Mówimy tu o użyciu w kodzie aplikacji zewnętrznych zasobów, takich jak baza danych, mechanizm wysyłania maili, inne żądania sieciowe, muszą pojawić się ich *zastępniki* nie mające skutków ubocznych (na przykład zamiast prawdziwej funkcji wysyłania maila – funkcja która nie wysyła maila tylko odnotowuje jego wysłanie).

Jak wprowadzić możliwość podmiany funkcji wyjścia na jej zastępnik w taki sposób żeby funkcja obsługująca żądanie nie musiała być modyfikowana?

Jedno z rozwiązań to tzw. **dostawca usług**

To element kodu, który jest fabryką funkcji/obiektów i może być przekonfigurowany w zależności od potrzeb. Przykład prostego dostawcy usług

```
class ServiceProvider
{
    constructor() {
        this.services = {}
    }

    registerService( name, service ) {
        this.services[name] = service;
    }
```

```
    resolveService( name ) {
        if ( this.services[name] ) {
            return this.services[name];
        }

        throw `No service registered for ${name}`;
    }
}

module.exports = new ServiceProvider();
```

Jego użycie w aplikacji:

```
var sp = require("./sp");

...
sp.registerService('email', sendEmail );
```

Dzięki temu, w funkcji obsługującej żądanie można napisać

```
function rootPost( email ) {
    var emailSender = sp.resolveService('email');
    emailSender( email );
}
```

Proszę zwrócić uwagę, że ta funkcja jest uniwersalna – nie zmieni się jeśli w aplikacji na czas testów jednostkowych zostanie skonfigurowana inna usługa.

Teraz można już napisać test:

```
var assert = require('assert');
var sp = require('../sp');

function rootPost( email ) {
  var emailSender = sp.resolveService('email');
  emailSender( email );
}

describe('Testy logiki aplikacji', function () {

  it('POST do widoku głównego', function () {
```

```
    // przygotowanie testu
    var isEmailSent = false;

    sp.registerService('email', (email) => {
      isEmailSent = true;
    })

    // przeprowadzenie testu
    rootPost('foo')

    // sprawdzenie poprawności

    assert.equal( true, isEmailSent );
  });
});
```

4 Testy interfejsu użytkownika

4.1 Selenium

Okazuje się, że testowanie aplikacji internetowych nie jest takie łatwe – stosunkowo łatwo przetestować to co dzieje się na serwerze ale jak testować silnik przeglądarki, to co widzi i z czym pracuje użytkownik?

Na te pytania długo nie było odpowiedzi.

Przez długi czas jednym z rozwiązań było korzystanie z interfejsów automatyzujących aplikacje okienkowe. W Windows był to interfejs UI Automation API. W taki sposób daje się automatyzować proste aplikacje, ale już nie – złożoną zawartość w przeglądarce.

Krokiem naprzód była pierwsza wersja platformy Selenium (2005). Pomysł był karkołomny – wykorzystać niskopoziomową wiedzę na temat tego jak działają popularne przeglądarki i przygotować API do sterowania nimi. Pomysł był zawodny, nowa wersja przeglądarki mogła powodować problemy w działaniu Selenium.

W 2012 roku podjęto udaną próbę ustandaryzowania sposobu komunikacji narzędzia z przeglądarkami – zamiast je „hackować" zaproponowano protokół komunikacyjny, WebDriver, Specjalna wersja przeglądarki (dostępne na stronie Selenium) implementują ten protokół i pozwalają na sterowanie nimi z poziomu kodu (napisanego w zasadzie w dowolnym języku).

Implementacji protokołu dla Node.js jest wiele, w tym np. Web Driver IO.

```
var webdriverio = require('webdriverio');
var options = { desiredCapabilities: { browserName: 'chrome' } };
var client = webdriverio.remote(options);
client
    .init()
    .url('https://duckduckgo.com/')
    .setValue('#search_form_input_homepage', 'WebdriverIO')
    .click('#search_button_homepage')
    .getTitle().then(function(title) {
        console.log('Title is: ' + title);
        // outputs: "Title is: WebdriverIO (Software) at DuckDuckGo"
    })
    .end();
```

4.2 Puppeteer

Puppeteer reprezentuje inne podejście – ogranicza się do jednej przeglądarki (Chromium) ale za to:

- Udostępnia całe jej API

- Pozwala automatyzować przeglądarkę bez tworzenia jej okna (w tym: bez potrzeby posiadania pulpitu w interaktywnej sesji użytkownika!)

```
const puppeteer = require('puppeteer');

(async () => {
  const browser = await puppeteer.launch();
  const page = await browser.newPage();
  await page.goto('https://example.com');
  await page.screenshot({path: 'example.png'});

  await browser.close();
})();
```

4.3 Playwright

Framework Playwright łączy elementy do tej pory występujące fragmentarycznie we wcześniejszych podejściach, m.in.:

- Wsparcie różnych przeglądarek
- Tryb headless
- Generator kodu

Aby utworzyć pusty projekt, należy w konsoli wydać polecenie

```
npm init playwright@latest
```

Testy znajdują się w folderze /tests. W zależności od wyboru, tworzy się je w jednym ze wspieranych języków.

Warto przestudiować ważniejsze fragmenty dokumentacji, m.in.:

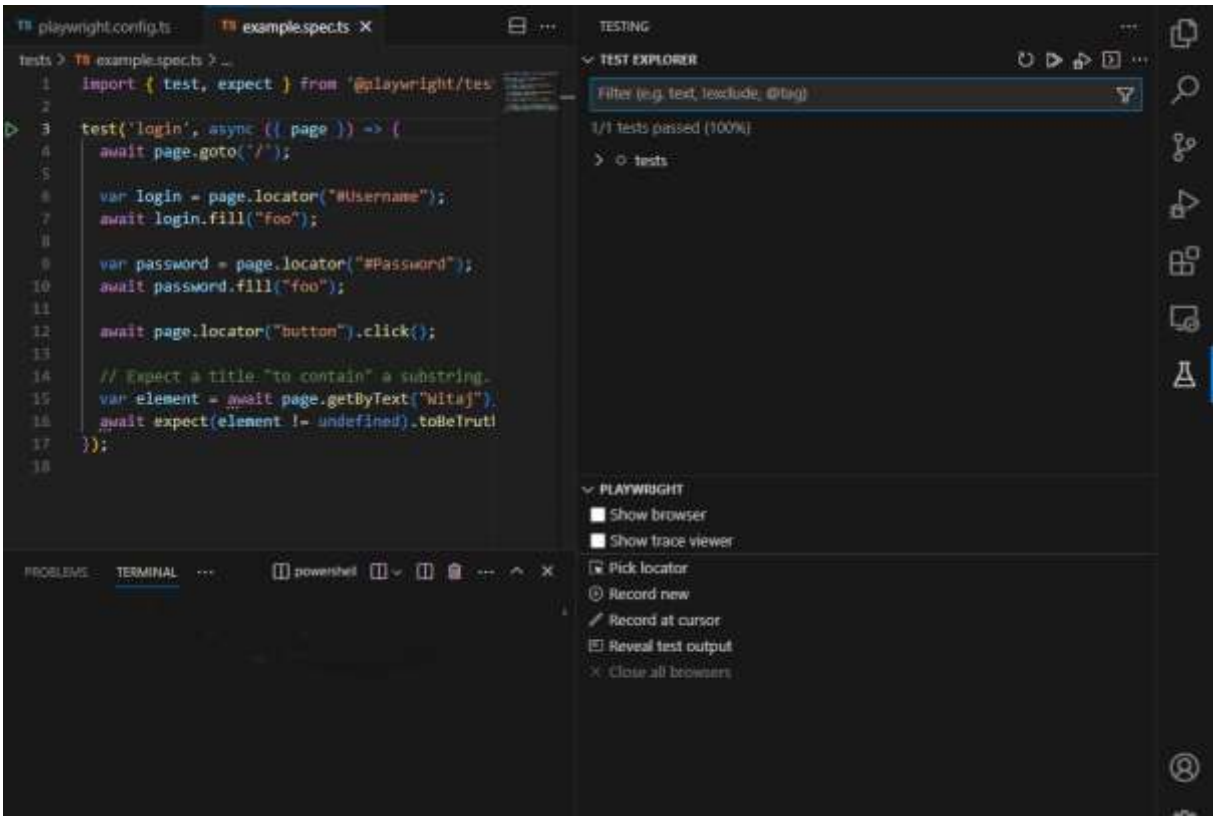
- Akcje
- Asercje

Przykładowy test do aplikacji z poprzedniego rozdziału poniżej. Przed uruchomieniem testu należy w **playwright.config.ts** zmienić **baseURL** na url odpowiadający ścieżce aplikacji.

```
import { test, expect } from '@playwright/test';

test('login', async ({ page }) => {
  await page.goto('/');

  var login = page.locator("#Username");
  await login.fill("foo");

  var password = page.locator("#Password");
  await password.fill("foo");

  await page.locator("button").click();

  // Expect a title "to contain" a substring.
  var element = await page.getByText("Witaj");
  await expect(element != undefined).toBeTruthy();
});
```

Tworzenie własnych testów możliwe jest w trybie pisania kodu, lub przez użycie generatora.

Generator dostępny jest w VS Code na zakładce testów:


playwright.config.ts
example.spec.ts
tests > example.spec.ts > ...
import { test, expect } from '@playwright/tes
test('login', async ({ page }) => {
await page.goto('/');
var login = page.locator("#Username");
await login.fill("foo");
var password = page.locator("#Password");
await password.fill("foo");
await page.locator("button").click();
// Expect a title "to contain" a substring.
var element = await page.getByText("Witaj");
await expect(element != undefined).toBeTrut
});
PROBLEMS
TERMINAL
powershell
TESTING
TEST EXPLORER
Filter (e.g. text, !exclude, @tag)
1/1 tests passed (100%)
tests
PLAYWRIGHT
Show browser
Show trace viewer
Pick locator
Record new
Record at cursor
Reveal test output
Close all browsers