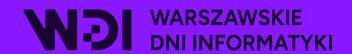
Mockowanie komunikacji z api

Najczęstsze błędy w testach i zaawansowane patterny

Łukasz Nowak

Senior software developer



Prelekcja wybrana w wyniku selekcji przez Radę Programową złożoną z uznanych liderów obszaru IT oraz Data Science.



vvarszawa, 04.04.2025 - 05.04.2025







Łukasz Nowak

Senior frontend dev



Łukasz Nowak



LukaszNowakPL

Dlaczego testy bywają flaky?

- Nieprzewidywalny condition race
- Aplikacja nie jest gotowa na test
- Wpadamy w pułapkę myślenia tunelowego
- Nieprawidłowo mockujemy
- Testujemy na niewłaściwym poziomie

Idealnie, w testach mockować powinniśmy tylko zewnętrzny świat, z którym się integrujemy.

Z czym integruje się frontend?

- Użytkownik assistive technologies, ekran, klawiatura, myszka
- Internet Url w przeglądarce
- Serwisy backendowe komunikacja api
- Pozostałe peryferyjne api (np. geolokalizacja)

Doktoryzacja z mockowania komunikacji z serwisami.

Przykładowe repozytorium

- Funkcjonalna React SPA + Backend
- Przykładowy kod dla każdego rozdziału
- Zawiera omawiany bad pattern i propozycję poprawy
- Poziomy testów:
 - Testy integracyjne Vitest
 - o Testy funkcjonalne Playwright



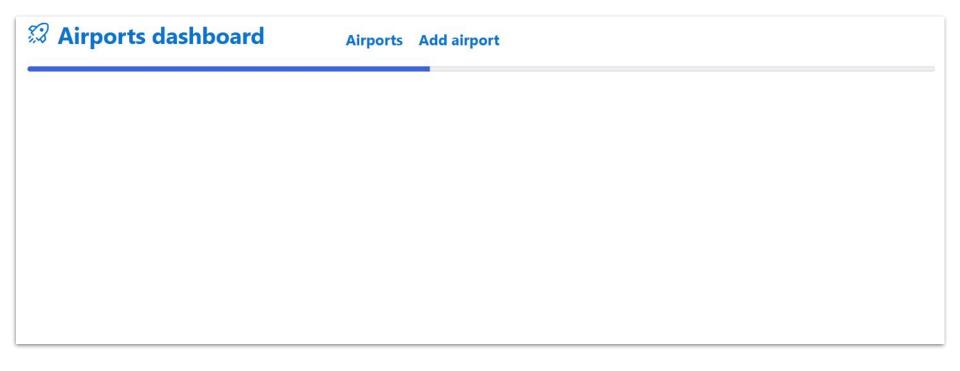
https://github.com/LukaszNowakPL/advanced-api-mocking-patterns

Brak mockowania komunikacji z api

Brak mockowania komunikacji z api

Nie mockujemy, bo nie wspiera bezpośrednio naszego testu

Testowanie na obecność loadera



Brak mockowania komunikacji z api

- Nie mockujemy, bo nie wspiera bezpośrednio naszego testu
- Komunikacja jest asynchroniczna, jej wynik może być dostępny dopiero po zakończeniu testu
- Powoduje losowo wyglądające błędy przy rozbudowanych suitach testowych

Jak to naprawić?

- Dodać tooling do mockowania komunikacji z api
- Zamockować połączenia wykonywane w trakcie testu

Mockowanie w MSW

```
// api-handlers/countries.ts
import {http, HttpResponse} from 'msw';
import {CountriesDto} from '../../src/api/rest/countries.dto';

export const countriesHandler = (responseData: CountriesDto, status = 200) =>
   http.get(`*/api/countries`, () => HttpResponse.json(responseData, {status}));
```

```
// AirportView.test.tsx
server.use(
   countriesHandler([countryData]),
   regionsHandler([]),
);
```

Mockowanie w Playwright

```
test.beforeEach (async ({page}) => {
   mockiavelli = await Mockiavelli.setup(page);
});
test('Airport addition journey', async ({page}) => {
   // And mocks of api calls triggered during the test
   const postAirportMock =
      mockiavelli.mockPOST('http://localhost:4000/api/airports', {
         status: 200,
         body: {},
   });
```

Jak to naprawić?

- Dodać tooling do mockowania komunikacji z api
- Zamockować połączenia wykonywane w trakcie testu
- Stworzyć listener informujący o brakującym mocku

Ustawienie listenera w MSW

Mockowanie nadmiarowe

Mockowanie nadmiarowe

- Błąd mylnie rozumianego patternu DRY
- Najczęstsze przypadki
 - w beforeEach() grupy testów
 - w setupie środowiska do mockowania
 - stosowanie wildcards, np. dla /airports/*

Hardcodowanie mocków na setupie MSW

```
export const server = setupServer(
  http.get(`*/api/countries`, () => HttpResponse.json(countriesList)),
  http.get(`*/api/airports`, () => HttpResponse.json(airportsList)),
  http.get(`*/api/airports/*`, () => HttpResponse.json(airport))
  // ...
);
```

Jak to naprawić?

Mockować każde połączenia per test

Mockowanie połączenia w teście

Mockowanie restrykcyjne

Mockowanie restrykcyjne

- Mockowanie tylko połączeń realnie wykonanych w teście
- Brak obsługi pozostałych połączeń będzie info w konsoli
- Konieczność potwierdzenia:
 - Parametrów ścieżki
 - Danych w body
 - Pozostałych części istotnych dla testu np. Search params, Headers, Cookies
- Pattern najdokładniej testujący integrację z backendem

Mockowanie restrykcyjne w MSW

```
export const addAirportHandler = (formData: AirportModel, status = 200) => {
    return http.post('*/api/airports', async ({request}) => {
        if (await isModelCorrect (formData, request)) {
            return HttpResponse.json({}, {status});
        }
    });
};
```

Porównanie obiektów

```
const isModelCorrect = async (formData: any, request) => {
  const requestBody = await request.clone().text();
  const stringifiedFormData = JSON.stringify(formData);

if (requestBody === stringifiedFormData) {
    return true;
} else {
    console.error('Body data does not match');
    return false;
}
};
```

Mockowanie restrykcyjne w Mockiavelli

```
// And new airport data
const airport: AirportModel = {
   name: 'test airport name',
   iata: 'TES',
   country id: countries[0].id,
   regions: [regionToSelect.id],
   vaccination notes: 'test vaccination notes',
//...
// Then POST api call is resolved with expected body
const postAirportRequest = await postAirportMock.waitForRequest();
expect (postAirportRequest .body) .toEqual (airport);
```

Stosowanie wzorca fabryka

Uciążliwe tworzenie danych zwracanych przez api

Pomysł: Obiekt bazowy do budowy mocków

```
// data-mocks.ts
const baseAirport: AirportModel = {
   name: 'test airport',
   iata: 'TES',
   country id: 1,
   regions: []
// Component.test.tsx
const spanishAirport = {
   ...baseAirport,
   country id: 2
```

Baza zbyt skomplikowana dla spread operator

```
const complicatedDto = {
   client: {
       name: 'test name',
       surname: 'test surname'
   contact: \{/* \dots */\},
   address: {
       permanent: \{/* \dots */\},
       temporary: { /* ... */}
```

Stosowanie wzorca fabryka

- Obiekt bazowy do budowy mocków
- Spread operator vs. dedykowana biblioteka

Przykład użycia Factory.ts

```
// factories/countries.ts
import * as Factory from 'factory.ts';

export const countryFactory = Factory.Sync.makeFactory<CountryDto>({
   id: 1,
   name: 'test country name',
   is_in_schengen: true,
});
```

```
// AirportView.test.tsx
// And country mock
const country = countryFactory.build({name: 'different country name' });

// And mocks of api calls triggered during the test
server.use(countriesHandler([country]) /* ... */);
```

Stosowanie wzorca fabryka

- Obiekt bazowy do budowy mocków
- Spread operator vs. dedykowana biblioteka
- Uwaga na pułapkę: Nigdy nie ufaj obiektowi bazowemu, zawsze nadpisuj to co chcesz później testować.

Błędne poleganie na obiekcie bazowym

```
// factories/countries.ts
import * as Factory from 'factory.ts';

export const countryFactory = Factory.Sync.makeFactory<CountryDto>({
   id: 1,
   name: 'test country name',
   is_in_schengen: true,
});
```

```
// AirportView.test.tsx
// And schengen country
const schengenLikeCountry = countryFactory.build({name: 'other country name'});
// And schengen-related test checks
// ...
```

Nadpisanie wszystkich testowanych elementów

```
export const countryFactory = Factory.Sync.makeFactory<CountryDto>({
   id: 1,
   name: 'test country name',
   is_in_schengen: true,
});
```

```
// And schengen country
const schengenCountry = countryFactory.build({
  name: 'test country name',
   is_in_schengen: true
});

// And schengen-related test checks
```

Użycie losowości do tworzenia mocków

Użycie losowości do tworzenia mocków

- Założenie: Cały obiekt bazowy jest losowy
- Efekt: Konieczność nadpisania danych używanych w teście
- Wynik: Obiekt bazowy zapewnia tylko zgodność z Dto/Modelem
 Odpowiedzialność za treść danych leży w warstwie testu
- Konieczność: Losowanie na poziomie enterprise

Przykład losowości danych z Faker library

```
import * as Factory from 'factory.ts';
import {faker} from '@faker-js/faker';
export const airportFactory = Factory.Sync.makeFactory<AirportDto>({
   id: faker.number.int(),
  name: faker.airline.airport().name,
   iata: faker.airline.airport().iataCode,
  country id: faker.number.int(),
   regions: faker.helpers.multiple(
       () => faker.number.int(), {count: faker.number.int(5)}
  ),
});
```

Nadpisanie danych używanych w teście

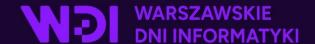
```
// And country data
const country = {
   id: 1, is in schengen: false,
   is passport required: true, is visa required: false
const nonSchengenCountryData = countryFactory.build(country)
// And airport data
const airport = {id: 99, name: 'test airport name', country: country.id}
const airportData = airportFactory.build(airport)
// And mocks of api calls triggered during the test
await countriesMock(page, [nonSchengenCountryData])
await airportMock(page, airport.id, airportData)
// And test for non-schengen airport ...
```

Co z tego mamy?

- Wady
 - Komplikuje proces tworzenia mocków
 - Komplikuje debugowanie testów
 - Wymaga sporego doświadczenia w pisaniu miarodajnych testów
- Zalety
 - Najlepiej oddaje nieprzewidywalność akcji użytkowników

Feedback

Zeskanuj kod i zostaw swoją opinię





Mockowanie komunikacji pomiędzy serwisami - najczęstsze błędy w testach i zaawansowane patterny

Łukasz Nowak

https://warszawskiedniinformatyki.pl/user.html#!/lecture/WDI25-36ea/rate

Linki

1. Mock Service Worker (MSW)

https://mswjs.io/

2. Mockiavelli

https://www.npmjs.com/package/mockiavelli

3. Factory.ts

https://www.npmjs.com/package/factory.ts

4. Faker

https://fakerjs.dev/