

Narzędzie wspierające tworzenie testów dla aplikacji webowych

Łukasz Kowalewski
Promotor: dr inż. Marcin Adamski

Spis treści

Spis treści	1
1 Wstęp	3
1.1 Cel i zakres pracy	3
1.2 Struktura pracy	4
2 Przykłady zastosowań testów	5
2.1 Testy jednostkowe	5
2.2 Testy integracyjne	6
2.3 Testy end-to-end (E2E)	6
3 Dostępne technologie	8
3.1 Frameworki testowe	8
3.2 Biblioteki wspomagające generowanie testów	9
3.3 Narzędzia CI/CD	10
4 Strategie generacji testów	13
4.1 Generowanie testów na podstawie rejestrowania akcji	13
4.2 Generowanie testów w oparciu o model systemu	14
5 Omówienie architektury aplikacji	16
5.1 Funkcjonalności i główne komponenty (Rozdział 5.1)	16
5.1.1 Zależność plików <code>spec.js</code> i <code>screenshotTestBuilder</code>	20
5.1.2 Rola <code>testManager</code> w zarządzaniu <code>spec.js</code> i codegenem	20
5.1.3 Działanie <code>mockApi</code> oraz <code>mockTest</code>	21
5.2 Szczegóły wybranych elementów (Rozdział 5.2)	21
5.2.1 API <code>screenshotTestBuilder</code> (metody <code>set</code> , <code>for</code> i inne) + przykład <code>spec.js</code>	21
5.2.2 Struktura plików <code>mock.js</code> (pola, za co odpowiadają) + przykład	23
5.2.3 Skrypty w <code>package.json</code>	24
5.2.4 <code>testManager</code>	25
6 Kluczowe mechanizmy i biblioteki	30
6.1 Zasady organizacji projektu	30
6.1.1 Struktura folderów i plików projektu; nazewnictwo zrzutów ekranu	30
6.1.2 Mechanizm tworzenia testów E2E przez <code>screenshotTestBuilder</code>	31
6.1.3 Mechanizm <code>testManager</code> z wykorzystaniem <code>codegen</code>	32
6.1.4 Mechanizm działania <code>mockTest</code>	33
6.2 Omówienie kluczowych bibliotek	34

6.2.1 Kluczowe miejsca w kodzie	35
7 Przykłady użycia aplikacji	41
7.1 Uruchamianie narzędzia w trybie interaktywnym	41
7.1.1 Przykładowy kod <code>home.spec.js</code>	46
7.1.2 Przykładowy kod <code>home.mock.js</code>	47
7.2 Integracja z procesem CI/CD	49
8 Podsumowanie	51
8.1 Wnioski końcowe	51
8.2 Możliwości dalszego rozwoju	51
Spis rysunków	53
Spis rysunków	53
Bibliografia	54

Rozdział 1

Wstęp

W niniejszym wstępie zostanie uzasadniona istotność tematu pracy inżynierskiej. Tytuł pracy – *Narzędzie wspierające tworzenie testów dla aplikacji webowych* – wskazuje główną funkcję tworzonego w jej ramach programu. Jego rolą jest dostarczenie użytkownikom narzędzia pozwalającego przygotowywać oprogramowanie i testy w krótszym czasie przy zachowaniu wysokiej jakości. W kolejnych rozdziałach omówione zostaną motywacje, problematyka zagadnienia oraz możliwe rozwiązania. Aby zrozumieć powody podjęcia takiego tematu, należy wykazać potrzebę tworzenia oprogramowania wspierającego proces testowania.

Testy stanowią kluczowy element zaawansowanych i dojrzałych systemów informatycznych oraz aplikacji. W dużych korporacjach istnieją specjalne działy zajmujące się wyłącznie tworzeniem i utrzymywaniem testów, a sam rynek pracy sygnalizuje wysoki popyt na programistów wyspecjalizowanych w tym obszarze. W takich organizacjach powstają zaawansowane narzędzia do automatyzacji testów, co jest zrozumiałe w kontekście ogromnych strat finansowych, jakie mogą wynikać z błędów w krytycznym oprogramowaniu.

Odmienne sytuacja bywa postrzegana w małych i średnich projektach, gdzie często brakuje dostatecznej motywacji (np. w postaci wysokich strat finansowych), by inwestować czas w rozbudowaną automatyzację testów. Celem tworzonej aplikacji jest ułatwienie generowania prostych testów w krótkim czasie, tak aby również mniejsze przedsięwzięcia mogły czerpać korzyści z automatyzacji, osiągając przy tym wyższe pokrycie testowe niewielkim nakładem pracy.

1.1 Cel i zakres pracy

Celem niniejszej pracy inżynierskiej jest opracowanie narzędzia, które znacząco usprawni proces tworzenia testów automatycznych w projektach webowych. Praca skupia się nie tylko na samej implementacji aplikacji, lecz także na analizie metod i dobrych praktyk w zakresie testowania.

Tworzone oprogramowanie ma rozwiązać przede wszystkim następujące problemy:

- **Wysoki próg wejścia** w automatyzację testów dla początkujących zespołów – narzędzie powinno dostarczyć przyjazne mechanizmy generowania przykładowych skryptów i scenariuszy testowych.
- **Długi czas przygotowywania testów** w małych i średnich projektach – planuje się zapewnić funkcje przyspieszające proces konfiguracji i pisania testów (np.

wstępne generowanie kodu testowego).

- **Integracja z istniejącymi technologiami** – narzędzie będzie wykorzystywać znane biblioteki (takie jak Selenium czy Playwright) oraz zapewni łatwą integrację z popularnymi pipeline’ami CI/CD.

Oczekiwanym rezultatem jest działające oprogramowanie, którego zastosowanie umożliwi efektywne tworzenie testów automatycznych oraz ich integrację z cyklem wytwarzania oprogramowania. Dokumentacja pracy przybliży zarówno aspekty teoretyczne (przegląd frameworków, strategii testowania), jak i praktyczne (omówienie kluczowych fragmentów kodu, przykłady uruchomienia testów oraz wdrożenia w środowisku CI/CD).

Zakres opracowania obejmuje:

- Analizę dostępnych narzędzi i bibliotek testowych dla aplikacji webowych,
- Zaprojektowanie i implementację modularnego narzędzia generującego testy,
- Przedstawienie możliwych sposobów dalszego rozwoju aplikacji, w tym plan rozbudowy funkcjonalności oraz integracji z innymi platformami.

1.2 Struktura pracy

W kolejnych rozdziałach przedstawiono najważniejsze aspekty projektowania i tworzenia narzędzia do wspierania testów automatycznych:

- **Rozdział 2** omawia przykłady zastosowań testów automatycznych, przedstawiając ich różnorodne formy (testy jednostkowe, integracyjne i end-to-end).
- **Rozdział 3** zawiera przegląd popularnych technologii i bibliotek wspierających testowanie aplikacji webowych, w tym frameworki testowe, narzędzia typu record-and-play czy rozwiązania CI/CD.
- **Rozdział 4** dotyczy strategii generowania testów, poczynając od podejść opartych na rejestrowaniu akcji, aż po wykorzystanie modeli aplikacji.
- **Rozdział 5** opisuje architekturę i interfejs projektowanej aplikacji, omawiając poszczególne moduły i sposób ich wzajemnej komunikacji.
- **Rozdział 6** poświęcony jest kluczowym elementom kodu źródłowego i zastosowanym bibliotekom.
- **Rozdział 7** prezentuje praktyczne przykłady użycia aplikacji w trybie interaktywnym oraz zintegrowanym w potoku CI/CD.
- **Rozdział 8** stanowi podsumowanie pracy, omawia wnioski końcowe i możliwe kierunki rozwoju narzędzia.

Rozdział 2

Przykłady zastosowań testów

Współczesne aplikacje webowe – niezależnie od skali – wymagają odpowiedniego poziomu kontroli jakości. Testy automatyczne są jednym z najbardziej efektywnych sposobów osiągnięcia tego celu. W niniejszym rozdziale omówiono podstawowe rodzaje testów stosowanych przy rozwoju oprogramowania, w szczególności aplikacji internetowych. Zwrócono przy tym uwagę zarówno na testy jednostkowe, integracyjne, jak i end-to-end (E2E). W bardziej złożonych projektach popularne jest podejście wielopoziomowe, pozwalające uniknąć błędów na różnych etapach tworzenia oprogramowania.

Testy automatyczne pełnią kluczową rolę w procesie ciągłej integracji i dostarczania (CI/CD). Ich cykliczne uruchamianie przed wdrożeniem minimalizuje ryzyko wprowadzenia wadliwych zmian. W dalszej części rozdziału zaprezentowano najistotniejsze cechy trzech podstawowych poziomów testów oraz omówiono korzyści i wyzwania wynikające z ich stosowania.

2.1 Testy jednostkowe

Testy jednostkowe (*unit tests*) dotyczą najmniejszych elementów oprogramowania, zazwyczaj pojedynczych funkcji czy metod. Ich celem jest weryfikacja poprawności działania konkretnego fragmentu kodu w oderwaniu od reszty systemu. Dzięki temu programiści mogą szybko wykrywać regresje w przypadku wprowadzania nowych funkcjonalności bądź zmian.

Charakterystyka i zalety testów jednostkowych

- **Wczesne wykrywanie błędów:** mały zakres testowanego kodu umożliwia łatwą diagnozę przyczyn problemów.
- **Szybkie uruchamianie:** testy jednostkowe są przeważnie mało zasobożerne, można je więc wykonywać nawet przy każdej kompilacji.
- **Wspieranie refaktoryzacji:** dobrze napisane testy jednostkowe pełnią rolę siatki bezpieczeństwa przy wprowadzaniu modyfikacji w kodzie.

Miejsce w cyklu życia aplikacji

Testy jednostkowe powstają zazwyczaj wraz z implementacją nowych funkcji, często w podejściu *Test-Driven Development* (TDD). Nawet w projektach niestosujących formalnie

TDD, testy jednostkowe są pisane równolegle bądź krótko po wprowadzeniu kluczowych metod. Dzięki temu deweloperzy na bieżąco weryfikują jakość kodu.

Znaczenie dla jakości oprogramowania

Choć testy jednostkowe nie wykrywają wszystkich możliwych błędów (w szczególności tych związanych z integracją czy kompleksową logiką biznesową), to stanowią podstawę solidnego procesu testowania. Ułatwiają utrzymanie wysokiej jakości kodu przez cały okres rozwoju aplikacji, zmniejszając liczbę nieoczekiwanych problemów w krytycznych częściach systemu.

2.2 Testy integracyjne

Testy integracyjne (*integration tests*) weryfikują poprawność współpracy pomiędzy różnymi komponentami systemu. W przeciwieństwie do testów jednostkowych, koncentrujących się na pojedynczych funkcjach, testy integracyjne sprawdzają, czy moduły wchodzące w skład aplikacji działają razem w sposób spójny i przewidywalny.

Główny cel i zakres

Podstawowym zadaniem testów integracyjnych jest upewnienie się, że wszystkie elementy systemu (np. warstwa serwerowa, baza danych, usługi zewnętrzne) współpracują zgodnie z oczekiwaniami. W aplikacjach webowych mogą obejmować m.in. testowanie komunikacji serwera z bazą, przepływ danych między mikrousługami czy integrację z API firm trzecich.

Przykłady zastosowań w aplikacjach webowych

- **Weryfikacja API i bazy danych:** testy integracyjne sprawdzają, czy żądania HTTP wysyłane przez front-end są prawidłowo obsługiwane w warstwie serwerowej oraz czy zwracane przez bazę dane są poprawne.
- **Integracje zewnętrzne:** jeśli aplikacja korzysta z usług płatności bądź map, testy integracyjne pozwalają zweryfikować, czy te usługi działają w oczekiwany sposób.
- **Reguły biznesowe po stronie serwera:** weryfikują sekwencje operacji wykonywanych przy współpracy wielu komponentów.

Korzyści i wyzwania

Zaletą testów integracyjnych jest możliwość szybkiego wykrywania błędów tam, gdzie różne moduły muszą ze sobą współdziałać. **Wyzwanie** stanowi zaś konieczność skonfigurowania środowisk testowych (bazy danych, serwerów, stubów dla usług zewnętrznych). Przy dużych projektach może to być czasochłonne i wymagające w utrzymaniu.

2.3 Testy end-to-end (E2E)

Testy end-to-end (*E2E*) to najbardziej rozbudowane testy funkcjonalne, w których symuluje się rzeczywiste zachowanie użytkownika końcowego (lub komunikację między sys-

temami) w całym przepływie aplikacji. Obejmują one wszystkie warstwy: od interfejsu użytkownika, przez warstwę serwera, aż po bazę danych i usługi zewnętrzne.

Na czym polega koncepcja testów E2E w aplikacjach webowych

Ideą testów E2E jest sprawdzenie działania całej aplikacji jako jednego spójnego rozwiązania. Taki test może obejmować:

1. Uruchomienie przeglądarki i przejście na stronę logowania.
2. Wprowadzenie danych logowania i przejście do kolejnej podstrony.
3. Wykonanie akcji biznesowych (np. zamówienie produktu).
4. Weryfikację wyników w bazie danych czy w komunikatach interfejsu.

Wszystko po to, by sprawdzić, czy aplikacja rzeczywiście działa zgodnie z wymaganiami i oczekiwaniami użytkownika.

Kiedy i dlaczego warto je stosować

- **Sprawdzenie kluczowych scenariuszy:** testy E2E pokrywają najważniejsze ścieżki biznesowe, zapewniając, że są one wolne od błędów krytycznych.
- **Realne warunki:** testy odzwierciedlają działania użytkownika końcowego, wykrywając problemy mogące pojawić się dopiero przy faktycznej interakcji z aplikacją.
- **Wysoka wiarygodność:** potwierdzenie poprawnego działania całości systemu.

Testy end-to-end są jednocześnie najwolniejsze i najbardziej wymagające w utrzymaniu, ponieważ trzeba uruchamiać wszystkie usługi składające się na system. Dlatego używa się ich głównie do kluczowych scenariuszy aplikacji i ostatecznej weryfikacji przed produkcyjnym wdrożeniem.

Przykładowe narzędzia

- **Selenium WebDriver** – klasyczne narzędzie automatyzujące przeglądarkę, dostępne w wielu językach programowania.
- **Cypress** – nowoczesny framework do testowania front-endu z szybkim sprzężeniem zwrotnym.
- **Playwright** – rozwijany przez Microsoft, obsługuje testy E2E dla Chromium, Firefox oraz WebKit, a także wiele języków programowania.
- **TestCafe** – rozwiązanie pozwalające pisać testy E2E w JavaScriptcie/TypeScriptcie, niewymagające instalowania dodatkowych sterowników przeglądarek.

Rozdział 3

Dostępne technologie

Rozdział ten omawia najpopularniejsze narzędzia i biblioteki wspierające proces testowania aplikacji webowych. Wybór właściwych frameworków i rozwiązań znacząco wpływa na wygodę tworzenia i utrzymywania testów, a także na łatwość integracji z otoczeniem projektowym.

3.1 Frameworki testowe

Framework testowy to zbiór narzędzi i bibliotek pozwalających pisać, organizować oraz uruchamiać testy w sposób zautomatyzowany. W przypadku testów aplikacji webowych frameworki te oferują wsparcie dla różnych języków programowania (Java, Python, JavaScript, C#) i rodzajów testów (jednostkowe, integracyjne, end-to-end).

Przegląd wybranych rozwiązań

- **JUnit / TestNG** (Java) – powszechnie używane w projektach Java. JUnit świetnie sprawdza się w testach jednostkowych, natomiast TestNG posiada bardziej rozbudowane funkcje konfiguracyjne i umożliwia równoległe uruchamianie testów.
- **Pytest** (Python) – cechuje się prostą składnią i bogatym ekosystemem wtyczek. Integruje się z wieloma innymi narzędziami, np. do raportowania.
- **Mocha / Jest** (JavaScript) – Mocha to elastyczny framework do testów asynchronicznych i synchronicznych, a Jest (tworzony przez Facebook) jest popularny w aplikacjach React.
- **Cucumber** – pozwala pisać testy w formie zrozumiałej dla nietechnicznych interesariuszy (składnia Gherkin), popularny w metodyce *Behavior-Driven Development*.
- **Playwright** – oprócz sterowania przeglądarką oferuje własny test runner, raportowanie i możliwość pisania testów w JavaScriptcie, Pythonie czy C#.

Kryteria wyboru

- **Język programowania** – warto wybrać framework naturalnie pasujący do języka wiodącego w projekcie.

- **Zakres testów** – dla testów front-endu React często wybiera się Jest, a do testów back-endu w Javie może lepiej sprawdzić się JUnit lub TestNG.
- **Integracja z CI/CD** – popularne narzędzia do ciągłej integracji (GitLab CI, GitHub Actions, Jenkins) posiadają często wtyczki czy gotowe przykłady integracji z określonym frameworkiem.
- **Spółeczność i dokumentacja** – dojrzałe rozwiązania z dużą społecznością ułatwiają rozwiązywanie problemów i zapewniają bogate zasoby przykładów.

3.2 Biblioteki wspomagające generowanie testów

Nowoczesne narzędzia do automatyzacji testów aplikacji webowych coraz częściej oferują opcje częściowego lub pełnego *generowania* skryptów testowych. Może to przybierać formę rejestrowania czynności wykonywanych w przeglądarce (tzw. podejście *record-and-play*) lub analizy kodu źródłowego w celu wygenerowania przykładowych testów. Takie rozwiązania mogą szczególnie przyspieszyć prace zespołów rozpoczynających przygodę z automatyzacją lub działających w mniejszych projektach.

Podejście *record-and-play*

Najłatwiejszą metodą generowania testów bywa *nagrywanie* czynności wykonywanych przez użytkownika w przeglądarce:

- **Selenium IDE** – wtyczka do przeglądarek Chrome i Firefox, nagrywająca akcje i eksportująca je do kodu w różnych językach (Java, Python, C#).
- **Playwright Codegen** – wbudowana w Playwright funkcja uruchamiająca przeglądarkę w trybie interaktywnym i generująca na tej podstawie gotowy kod testu (TypeScript, Python, C#, Java).
- **TestCafe Recorder** – dostępne w postaci wtyczek rozszerzenie do Chrome, nagrywające kliknięcia i wpisywane dane, a następnie tworzące skrypty w stylu TestCafe.

Podstawową zaletą *record-and-play* jest niski próg wejścia (łatwość generowania wstępnych testów E2E), natomiast wada to dość *sztywne* skrypty, wrażliwe na zmiany w interfejsie użytkownika.

Analiza kodu i generowanie szkieletów testów

Innym rozwiązaniem jest narzędziowe wspomaganie pisania testów poprzez analizę kodu aplikacji:

- **Narzędzia do analizy statycznej** – mogą wykrywać ścieżki wykonania w kodzie, wskazywać podejrzaną fragmenty i generować szkielety testów.
- **Boty testujące interfejs** – niektóre projekty open-source (*heuristics-based bots*) potrafią *klikać* w różne elementy według heurystyk, a następnie na tej podstawie generować wstępne skrypty E2E.

Podejścia te mogą wykryć scenariusze, o których człowiek mógłby nie pomyśleć, choć zazwyczaj wymagają dalszej konfiguracji czy refaktoryzacji kodu.

Zalety i wyzwania stosowania generatorów testów

Korzyści:

- *Oszczędność czasu* – szybkie stworzenie bazowej wersji skryptów testowych,
- *Niższy próg wejścia* – idealne dla nowych członków zespołu bądź mniej doświadczonych w automatyzacji,
- *Standaryzacja* – generatory często stosują uniwersalne wzorce, co ułatwia utrzymanie.

Wyzwania:

- *Brak elastyczności* – kod może być mocno zależny od konkretnego układu strony,
- *Konieczność refaktoryzacji* – wygenerowany kod bywa zbyt rozbudowany i wymaga uporządkowania,
- *Ciągłe aktualizacje* – przy zmianach w aplikacji część testów może przestać działać bez korekty generatora.

3.3 Narzędzia CI/CD

W większych projektach nie wystarcza jedynie lokalne uruchamianie testów. Testy automatyczne stają się integralną częścią cyklu ciągłej integracji i dostarczania (CI/CD), dzięki czemu każda nowa zmiana w kodzie jest automatycznie sprawdzana pod kątem jakości i stabilności. Poniżej omówiono kluczowe narzędzia CI/CD oraz sposoby włączania do nich testów.

GitLab CI

GitLab oferuje wbudowany mechanizm CI/CD oparty na pliku `.gitlab-ci.yml`. Aby uwzględnić w nim testy aplikacji webowych (np. napisane w Playwright czy Selenium), definiuje się:

- Etap testów (job) instalujący zależności,
- Polecenie uruchamiające testy (np. `npx playwright test`),
- Artefakty (raporty, zrzuty ekranu) pozwalające na łatwą inspekcję nieudanych uruchomień.

```
stages:
```

- build
- test

```
e2e_tests:
```

- stage: test
- image: mcr.microsoft.com/playwright:focal
- script:
 - npm ci

```
- npx playwright test
artifacts:
  when: on_failure
  paths:
    - playwright-report
```

GitHub Actions

GitHub Actions to popularne narzędzie do automatyzacji pracy w repozytorium GitHub, w tym do uruchamiania testów. Plik `.github/workflows/test.yml` konfiguruje *jobs*, definiując:

- Obraz bazowy (np. *ubuntu-latest*),
- Instalację wymagań,
- Uruchomienie testów,
- Publikację wyników (np. logów i raportów) jako *artifacts*.

Jenkins

Jenkins to jedno z bardziej konfigurowalnych narzędzi CI/CD typu open source. Służy do tworzenia *pipeline*'ów, w których:

- Pobiera się kod z repozytorium,
- Instaluje zależności i uruchamia testy,
- Generuje raporty (HTML, JUnit czy Allure),
- Prezentuje wyniki w interfejsie Jenkins po zakończeniu procesu.

Dostępnych jest wiele wtyczek, m.in. pozwalających na integrację z Selenium Grid czy Dockerem.

Kluczowe korzyści integracji testów z CI/CD

- **Stała kontrola jakości** – testy są wywoływane przy każdym *commit*,
- **Automatyczne raportowanie** – zespół ma bieżący wgląd w stan aplikacji,
- **Ciągłe dostarczanie** – w razie powodzenia testów, kod może zostać od razu wdrożony,
- **Elastyczność** – możliwość równoległego uruchamiania testów w różnych konfiguracjach.

Wyzwania i rozwój

- **Skonfigurowanie środowisk** – konieczne może być uruchamianie baz danych, mikroserwisów czy mocków,
- **Czas wykonania** – duże zestawy testów E2E wydłużają pipeline, co można łagodzić poprzez równoległe uruchamianie i skalowanie,
- **Utrzymanie** – wraz z rozwojem projektu rośnie liczba testów i ich złożoność, wymagając regularnych aktualizacji.

Rozdział 4

Strategie generacji testów

W poprzednich rozdziałach przedstawiono ogólne zasady automatycznego testowania aplikacji webowych i zaprezentowano wybrane technologie. Kolejnym krokiem jest omówienie różnych strategii generowania testów – od metod całkowicie automatycznych po podejścia półautomatyczne. Istnieje bowiem wiele sposobów tworzenia scenariuszy testowych, różniących się choćby poziomem ingerencji człowieka czy sposobem odwzorowania zachowań użytkownika.

W niniejszym rozdziale skoncentrujemy się na najczęściej stosowanych metodach generowania testów, w szczególności na podejściu *record-and-play* oraz rozwiązaniach opartych na modelu aplikacji. Dla mniejszych projektów liczy się często błyskawiczne stworzenie zestawu testowego, co motywuje do korzystania z narzędzi rejestrujących realne akcje w przeglądarce. W większych, bardziej zorganizowanych zespołach spotyka się narzędzia bazujące na formalnym opisie zachowania systemu, co pozwala generować obszerne zestawy testów pokrywające wiele nietypowych ścieżek.

4.1 Generowanie testów na podstawie rejestrowania akcji

Jednym z najbardziej intuicyjnych sposobów przyspieszających tworzenie testów automatycznych jest rejestrowanie czynności użytkownika w przeglądarce, często określane mianem podejścia *record-and-play*. Polega ono na „nagrywaniu” akcji wykonywanych przez testera (lub dewelopera) na stronie internetowej: kliknięć, wprowadzania danych w pola tekstowe czy przechodzenia między podstronami. Następnie narzędzie, które uczestniczy w tym procesie, generuje skrypt testowy w wybranym języku programowania.

Zalety podejścia *record-and-play*

- **Niski próg wejścia:** do przygotowania wstępnych testów nie trzeba znać szczegółowo frameworków testowych – wystarczy poprawnie wykonać scenariusze w przeglądarce.
- **Szybkie prototypowanie:** w ciągu kilku minut można uzyskać podstawowy plik testowy, który potem można udoskonalić.
- **Naturalne odwzorowanie zachowań użytkownika:** test powstaje na bazie realnego korzystania z aplikacji.

Wyzwania i ograniczenia

- **Nadmierna szczegółowość skryptu:** automatycznie wygenerowane testy są zwykle wrażliwe na nawet drobne zmiany w interfejsie.
- **Brak abstrakcji i trudniejsza konserwacja:** w *record-and-play* często nie ma wzorców typu Page Object, przez co kod może być trudniejszy w utrzymaniu.
- **Ograniczone pokrycie przypadków brzegowych:** narzędzia nagrywające skupiają się na głównych ścieżkach użytkownika.

Przykładowe narzędzia

- **Selenium IDE** – rozszerzenie do Chrome/Firefox, które umożliwia nagrywanie akcji i eksport skryptów (Java, Python, C#).
- **Cypress Recorder** – wtyczki do Chrome do rejestrowania czynności i generowania testów w stylu Cypress.
- **Playwright Codegen** – narzędzie wbudowane w Playwright, pozwala uruchomić przeglądarkę w trybie interaktywnym i generować testy w TypeScriptie, Pythonie czy .NET.

Podsumowując, podejście *record-and-play* świetnie nadaje się do szybkiego uzyskania bazowego zestawu testów, choć w dłuższej perspektywie wymaga zwykle wprowadzenia wzorców ułatwiających utrzymanie (np. Page Object Model).

4.2 Generowanie testów w oparciu o model systemu

Kolejnym sposobem automatycznego tworzenia skryptów testowych jest *Model-Based Testing* (MBT). W tym podejściu scenariusze testowe są generowane na bazie formalnego modelu systemu, opisującego możliwe stany i przejścia między nimi (np. w formie diagramów stanów lub sieci Petriego).

Zasada działania

1. **Budowa modelu:** przygotowanie schematu zachowania systemu (np. diagram stanów, opis przejść).
2. **Definicja danych testowych:** określenie danych wejściowych i oczekiwanych wyników dla poszczególnych akcji.
3. **Generowanie ścieżek:** narzędzie MBT automatycznie wyznacza ścieżki przejść w modelu, starając się osiągnąć ustalone kryterium pokrycia (np. wszystkie stany lub wszystkie przejścia).
4. **Konwersja na skrypty testowe:** każda ścieżka to osobny test. Narzędzie tłumaczy go na kod w wybranym frameworku.

Zalety i przykładowe zastosowania

- **Szersze pokrycie ścieżek:** automatyczna eksploracja modelu wychwytuje błędy w mniej typowych scenariuszach.
- **Synchronizacja z dokumentacją:** aktualizacja modelu w razie zmiany wymagań przekłada się na natychmiastowe odświeżenie testów.
- **Refaktoryzacja i konserwacja:** gdy model jest utrzymany w sposób czytelny, łatwiej zachować wysoką jakość testów w dłuższym horyzoncie.

Wyzwania i ograniczenia

- **Konieczność posiadania modelu:** przygotowanie i aktualizacja formalnego modelu systemu bywa pracochłonne.
- **Specjalistyczne narzędzia:** MBT wymaga często dedykowanego oprogramowania oraz znajomości specyficznych notacji.
- **Integracja z testami E2E:** wygenerowane testy trzeba nierzadko dostosować do konkretnych frameworków (Selenium, Playwright) i środowisk CI/CD.

Przykładowy proces

1. Przygotowanie diagramu stanów (np. logowanie, wyszukiwanie, wylogowanie).
2. Zdefiniowanie warunków przejść i danych (np. nazwy użytkowników, hasła).
3. Wygenerowanie ścieżek i tłumaczenie ich na skrypty testowe.
4. Uruchomienie testów w ramach istniejącego procesu CI/CD.

Podejście MBT jest szczególnie atrakcyjne w większych, długoterminowych projektach, gdzie sprawne utrzymanie pełnej dokumentacji i testów ma kluczowe znaczenie.

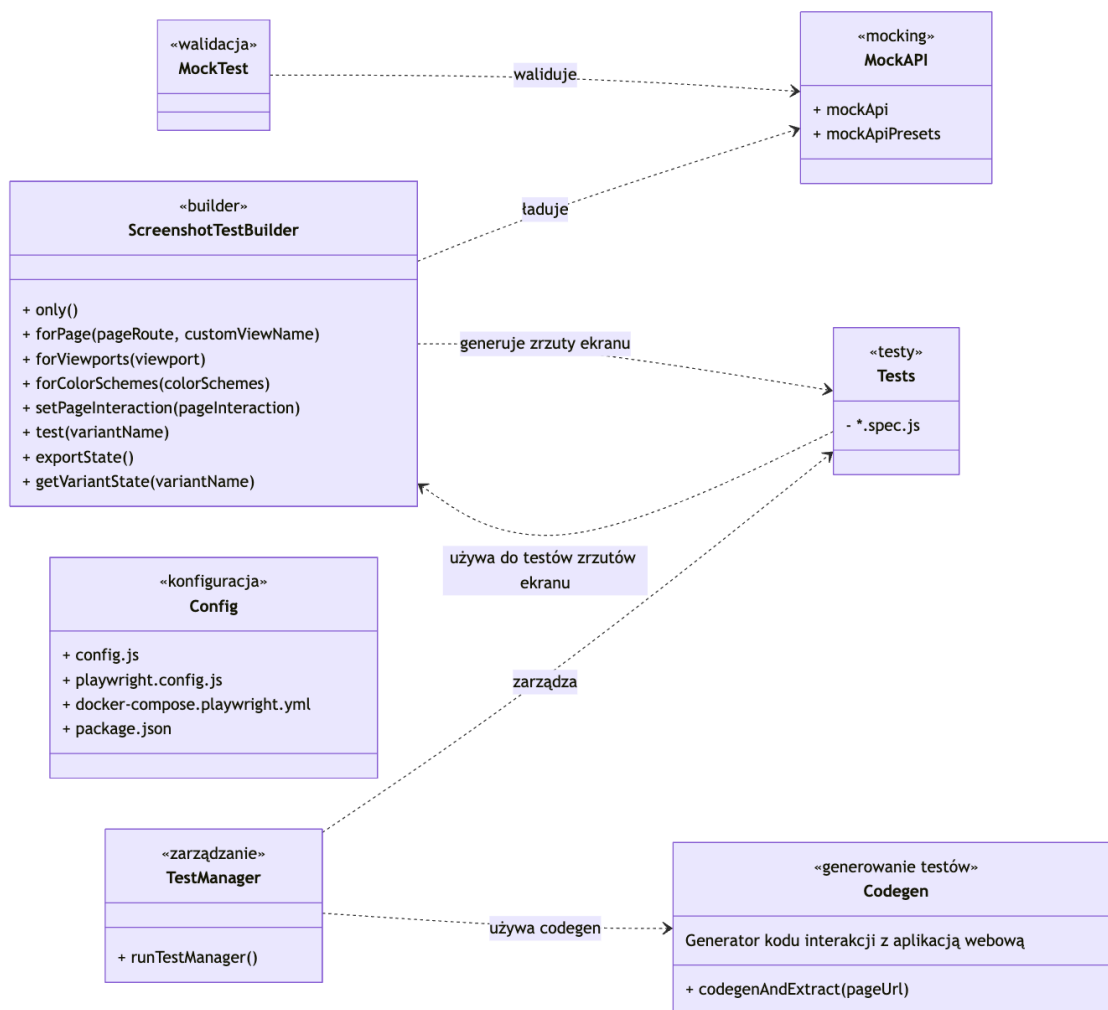
Rozdział 5

Omówienie architektury aplikacji

Niniejszy rozdział opisuje główne moduły tworzonego narzędzia, jego funkcjonalności oraz sposób, w jaki poszczególne komponenty współpracują ze sobą. W szczególności zaprezentowano tu dwa diagramy UML ilustrujące strukturę klas oraz przypadki użycia (z perspektywy użytkownika i systemu CI/CD). Przedstawiono również zależności między najważniejszymi plikami projektowymi (`spec.js`) a klasą `screenshotTestBuilder`, która stanowi klucz do automatycznego tworzenia testów zrzutów ekranu.

5.1 Funkcjonalności i główne komponenty (Rozdział 5.1)

W projekcie zaimplementowano moduły i usługi odpowiadające za tworzenie, wykonywanie oraz zarządzanie testami automatycznymi. Ilustruje to *diagram klas* (rys. 5.1) oraz *diagram przypadków użycia* (rys. 5.2), uwzględniające zarówno perspektywę architektury, jak i interakcje użytkownika z narzędziem.

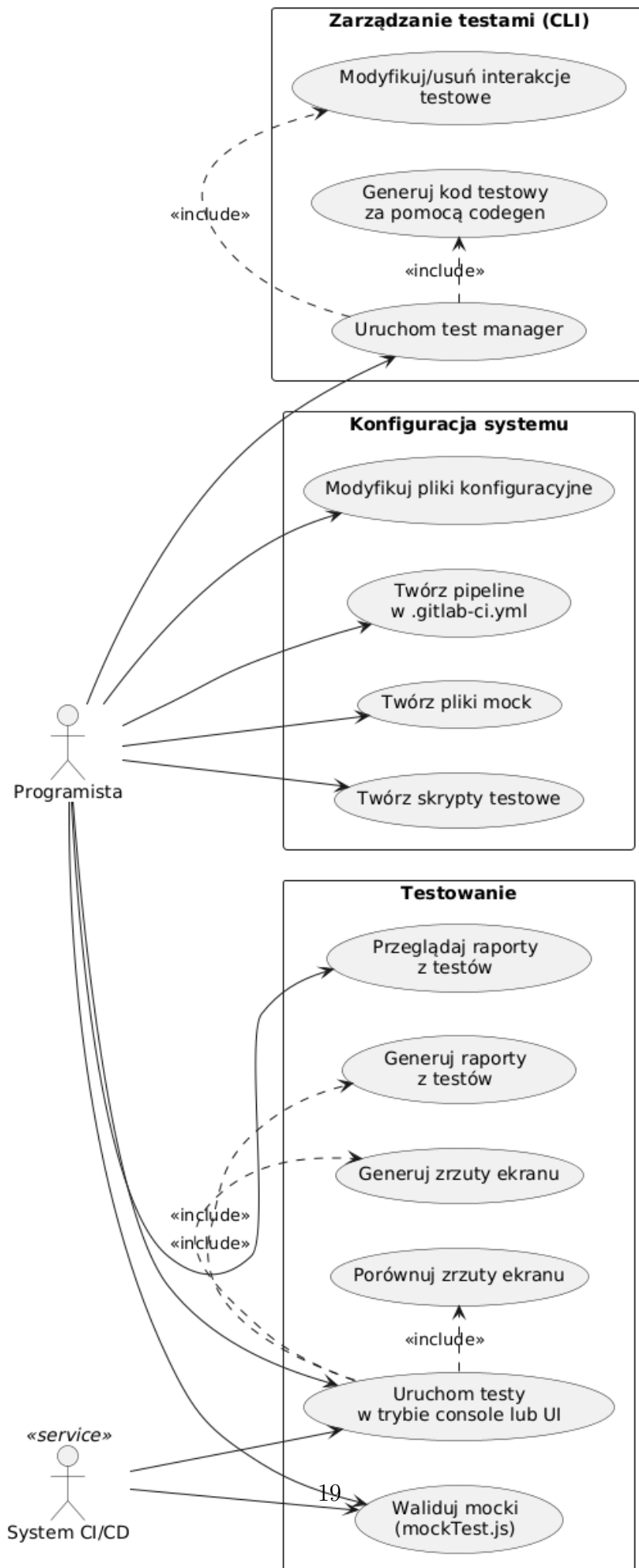


Rysunek 5.1: Diagram klas przedstawiający główne komponenty systemu

Najważniejsze elementy widoczne na powyższym diagramie to:

- **ScreenshotTestBuilder** – centralny komponent służący do definiowania parametrów testów wizualnych (np. rodzaje widoków, warianty kolorów, interakcje na stronie). Udostępnia metody umożliwiające budowanie kompletnego scenariusza testowego.
- **Tests** (zawartość plików `*.spec.js`) – zestaw testów wykorzystujących przygotowane warianty i funkcje **ScreenshotTestBuilder**. Każdy test może generować zrzuty ekranu w różnych konfiguracjach (urządzenie *desktop/mobile*, tryb *light/dark*, konkretna przeglądarka itp.).
- **TestManager** – narzędzie CLI do uruchamiania testów, zarządzania nimi (np. edycja wariantów, usuwanie lub dodawanie interakcji) oraz współpracy z **codegen**.
- **Codegen** – moduł pozwalający w półautomatyczny sposób wygenerować sekwencje interakcji z aplikacją (np. kliknięcia i wypełnianie formularzy), które następnie mogą być użyte w testach `spec.js`.
- **Config** – pliki konfiguracyjne projektu (np. ustawienia rozdzielczości, ścieżki URL), dzięki którym w prosty sposób można dostosować testy do różnych środowisk.

- **MockAPI** i **MockTest** – mechanizmy związane z symulowaniem wywołań API oraz walidacją, czy zdefiniowane dane **mock** są zgodne ze strukturą faktycznych odpowiedzi serwera.



Na diagramie przypadków użycia (rys. 5.2) widoczna jest pełna interakcja pomiędzy:

- **Programistą** – tworzy i modyfikuje pliki konfiguracyjne, definiuje testy (`spec.js`) oraz może uruchamiać menedżer testów (`TestManager`).
- **Systemem CI/CD** – automatycznie uruchamia testy (m.in. testy zrzutów ekranu i walidacje `mock`), generuje raporty i zarządza pipeline’em.

Zarówno człowiek (Programista), jak i *System CI/CD* korzystają z tych samych głównych operacji: uruchamianie testów, weryfikacja `mock` i przeglądanie raportów. Ponadto Programista może w dowolnym momencie wywołać `TestManager`, by zmodyfikować scenariusze testowe (np. dodać nową interakcję na stronie).

5.1.1 Zależność plików `spec.js` i `screenshotTestBuilder`

Wszystkie pliki testowe oznaczone rozszerzeniem `.spec.js` w tym projekcie opierają się na funkcjonalnościach dostarczanych przez klasę `ScreenshotTestBuilder`. Oznacza to, że każdy test:

1. Inicjalizuje `ScreenshotTestBuilder` (np. wywołując konstruktor i metody konfiguracyjne `forPage`, `setPageInteraction` itp.),
2. Definiuje warianty testów (`test()` z nazwą wariantu bądź bez),
3. Przechowuje i wykorzystuje stan zdefiniowany przez `ScreenshotTestBuilder`, aby ostatecznie wygenerować zrzuty ekranu w różnych kombinacjach ustawień.

Ścisła zależność między `spec.js` a `screenshotTestBuilder` polega na tym, że:

- `screenshotTestBuilder` udostępnia metody pozwalające na łatwe ustawienie dowolnych parametrów testu (np. `viewport`, tryb kolorów, interakcje użytkownika),
- `*.spec.js` korzystają z tych metod w celu „zbudowania” gotowego scenariusza testowego,
- Każdy nowy plik `.spec.js` w praktyce *musi* używać `ScreenshotTestBuilder`, dzięki czemu konfiguracja testów pozostaje spójna w całym projekcie.

5.1.2 Rola `testManager` w zarządzaniu `spec.js` i `codegenem`

`testManager` pełni w projekcie funkcję nadrzędnego narzędzia do zarządzania plikami `.spec.js` oraz wspiera Programistę w tworzeniu i modyfikowaniu interakcji testowych. Po uruchomieniu w trybie wiersza poleceń (CLI) pozwala m.in.:

- **Przeglądać istniejące testy** – użytkownik dostaje listę dostępnych plików `.spec.js` i może wybrać konkretny wariant testu do edycji.
- **Dodawać lub usuwać interakcje na stronie** – `testManager` umożliwia szybkie wstawienie nowych kroków (np. kliknięcie przycisku, wypełnienie formularza) lub wyeliminowanie dotychczasowych akcji.

- **Korzystać z *codegen*** – integracja z *Playwright codegen* umożliwia półautomatyczne generowanie scenariuszy testowych. Codegen nie jest częścią autorską tego projektu, lecz stanowi wbudowane narzędzie *Playwright* do rejestrowania czynności w przeglądarce. Dzięki `testManager`, Programista może wywołać `codegen`, który „nagrywa” kroki wykonywane na stronie, po czym wygenerowany kod może zostać łatwo włączony do pliku `.spec.js`.

Głównym zadaniem `testManager` jest więc ułatwienie nawigacji w wielu plikach testowych i wsparcie w zarządzaniu różnymi wariantami testów (np. różne interakcje na tej samej stronie). Rozwiązuje to problem ręcznego dopisywania fragmentów kodu w `.spec.js`, bo dzięki integracji z `codegen` i mechanizmami CLI większość operacji można wykonać przy pomocy interaktywnych pytań i odpowiedzi.

5.1.3 Działanie `mockApi` oraz `mockTest`

`mockApi` zapewnia w projekcie mechanizm definiowania sztucznych (testowych) odpowiedzi serwera. Dla wybranych endpointów API można zdefiniować, jakie dane mają być zwracane, dzięki czemu testy nie muszą opierać się na żywym środowisku produkcyjnym czy deweloperskim. Każda paczka `mock` (np. `/home/home.mock.js`) zawiera:

- **Dane do zwrócenia** – statyczne lub wygenerowane, opisujące np. przykładowy obiekt JSON, treść HTML czy obraz,
- **Parametry endpointu** – wskazanie żądanej ścieżki (np. `/branding/`) lub zapytania (`?query=test`),
- **Presety** – grupowanie kilku `mock` w zestawy, które można jednocześnie załadować w testach (np. `mockApiPresets.default`).

`mockTest` to z kolei moduł sprawdzający, czy zdefiniowane `mocki` są zgodne z prawdziwymi danymi, jakie zwraca produkcyjne API. Jego zadaniem jest:

- **Pobranie realnej odpowiedzi** z wybranego endpointu w środowisku docelowym,
- **Porównanie struktury** z danymi zadeklarowanymi w `mockApi` (sprawdzana jest m.in. zgodność typu pól czy format obrazów),
- **Raportowanie ewentualnych różnic** – w razie niezgodności `mockTest` wskazuje, który `mock` wymaga aktualizacji, by odzwierciedlać rzeczywisty stan API.

5.2 Szczegóły wybranych elementów (Rozdział 5.2)

5.2.1 API `screenshotTestBuilder` (metody `set`, `for` i inne) + przykład `spec.js`

W tej części opisano najważniejsze metody dostępne w klasie `ScreenshotTestBuilder`. Dzięki nim można w przejrzysty sposób zdefiniować konfigurację testów wizualnych:

- `only()` – pozwala oznaczyć dany zestaw testów jako wykonywany *wyłącznie* (z pominięciem innych).

- `forPage(pageRoute, customViewName?)` – określa, jaką trasę (URL) ma odwiedzić test i ew. jaką nazwę widoku nadać danej konfiguracji (np. "home").
- `forViewports(viewport[])` – ustawia listę viewportów (np. ["desktop", "mobile"]).
- `forColorSchemes(colorSchemes[])` – ustawia listę trybów kolorystycznych (np. ["light", "dark"]).
- `setPageInteraction(async (page) => {...})` – umożliwia zdefiniowanie akcji wykonywanych na stronie przed zrobieniem zrzutu ekranu (np. kliknięcie przycisku).
- `test(variantName?)` – finalizuje konfigurację i tworzy właściwe testy (w kontekście Playwright). Parametr `variantName` służy do rozróżnienia różnych wariantów testu (np. "booking").
- `exportState()` – zwraca obiekt z aktualnymi ustawieniami (np. `pageRoute`, `viewport`, `colorSchemes`, `pageInteraction` itp.). Mechanizm wykorzystywany np. przez `testManager`.
- `getVariantState(variantName)` – zwraca zapisany stan konfiguracji dla wybranego `variantName`.

Przykładowy plik `example.spec.js`:

```
//@ts-check
import ScreenshotTest from '../screenshotTestBuilder.js';

new ScreenshotTest()
  .forPage('/', 'home')           // Ustawia główny adres / i nazywa ten
    ↪ widok "home"
  .forViewports(['desktop', 'mobile'])
  .forColorSchemes(['light', 'dark'])
  .setPageInteraction(async (page) => {
    await page.click('text=Akceptuj Cookies');
  })
  .test('accept-cookies')         // Pierwszy wariant testu (nazwa
    ↪ "accept-cookies")

// Drugi wariant testu, inna interakcja
.setPageInteraction(async (page) => {
  await page.click('text=Otwórz Menu');
})
.test('open-menu');
```

W powyższym przykładzie zdefiniowano dwa warianty testu (`accept-cookies` oraz `open-menu`). Każdy z nich zostanie uruchomiony w dwóch trybach widoku (*desktop*, *mobile*) oraz dla dwóch schematów kolorów (*light*, *dark*). Łącznie wygeneruje to kilka zrzutów ekranu.

5.2.2 Struktura plików `mock.js` (pola, za co odpowiadają) + przykład

Pliki `.mock.js` odpowiadają za definiowanie odpowiedzi *mockowanych* endpointów API. Każdy taki plik eksportuje zazwyczaj dwa obiekty:

- `mockApi` – zawiera definicje poszczególnych endpointów (np. `branding`, `room`, `images`).
- `mockApiPresets` – grupuje definicje w postaci zestawów, np. `default`, `booking`, `error`.

Kluczowe pola w każdej definicji endpointu to:

- `endpoint` – ścieżka w API, np. `'room/'`.
- `query` – dodatkowe parametry zapytania (`'?page=1'`).
- `data` – treść, jaka zostanie zwrócona (obiekt JSON, tekst HTML, bufor obrazu).
- `contentType` – typ *Content-Type* odpowiedzi (`'application/json'`, `'image/jpeg'`, `'text/html'`).
- `apiUrl` (opcjonalne) – jeśli mock ma wskazywać na inny adres bazowy niż domyślny.

Przykładowy plik `simple.mock.js`:

```
export const mockApi = {
  branding: {
    default: {
      endpoint: 'branding/',
      query: '',
      data: {
        name: 'Hotel Testowy',
        description: 'Najlepszy hotel w okolicy',
      },
      contentType: 'application/json'
    }
  },
  images: {
    logo: {
      endpoint: 'images/logo.png',
      query: '',
      data: Buffer.from([0x89, 0x50, 0x4e, 0x47]), // przykładowy bajt
      contentType: 'image/png'
    }
  }
};

export const mockApiPresets = {
  default: [
    mockApi.branding.default,
```



```

    mockApi.images.logo
  ]
};

```

Dzięki `mockApiPresets.default` w testach można wczytać oba endpointy jednocześnie, co pozwala symulować środowisko bez konieczności korzystania z prawdziwej infrastruktury.

5.2.3 Skrypty w `package.json`

W pliku `package.json` zdefiniowano skrypty, które ułatwiają uruchamianie testów oraz zarządzanie projektem. Poniżej przedstawiono zawartość sekcji `"scripts"` wraz z krótkim omówieniem:

```

{
  "scripts": {
    "test:e2e": "playwright test",
    "test:e2e:docker": "docker-compose -f docker-compose.playwright.yml
      ↪ run --rm console",
    "test:e2e:ui": "playwright test --ui",
    "test:e2e:ui:docker": "docker-compose -f
      ↪ docker-compose.playwright.yml run --rm ui-mode",
    "manager": "node testManager.js",
    "test:mock": "node mockTest.js"
  }
}

```

- `"test:e2e"` – uruchamia testy end-to-end za pomocą Playwright w trybie konsolowym.
- `"test:e2e:docker"` – podobne uruchomienie testów E2E, lecz w kontenerze (zdefiniowanym w `docker-compose.playwright.yml`) i z poleceniem `console`.
- `"test:e2e:ui"` – wywołuje testy Playwright w trybie interfejsu graficznego (`-ui`), umożliwiając przeglądanie wyników testów w czasie rzeczywistym.
- `"test:e2e:ui:docker"` – analogicznie jak wyżej, ale uruchamia się w środowisku Docker.
- `"manager"` – wywołuje główne narzędzie CLI, czyli `testManager.js`, służące do interaktywnego zarządzania plikami `.spec.js` i wariantami testów.
- `"test:mock"` – uruchamia `mockTest.js`, który weryfikuje poprawność `mockApi` w porównaniu do realnego API.

Dzięki tym skryptom można sprawnie kontrolować różne typy testów (E2E w kontenerze czy lokalnie, testy mocków) oraz korzystać z `testManager` bez konieczności ręcznego wpisywania dłuższych poleceń.

5.2.4 testManager

Moduł `testManager` pełni rolę interfejsu CLI do zarządzania plikami testowymi (`.spec.js`) i ich wariantami. Pozwala m.in.:

- Wybierać plik `.spec.js` z dostępnej listy,
- Odczytywać i modyfikować warianty testów w danym pliku,
- Dodawać lub usuwać interakcje użytkownika (`pageInteraction`),
- Uruchamiać *Playwright codegen* w celu półautomatycznego wygenerowania nowych interakcji.

Poniżej pokazano przykładowe ekrany z działania `testManager`:

```
suduvis@suduvis-V0C:~/Projects/test-builder$ npm run manager

> manager
> node testManager.js

Welcome to testManager CLI!

? Select a test file: (Use arrow keys)
> tests/cookie/cookie.spec.js
  tests/home/home.spec.js
  _____
  Exit
```

Rysunek 5.3: Główne menu `testManager` po uruchomieniu: użytkownik wybiera plik `.spec.js` lub opcję *Exit*.

Po wybraniu konkretnego pliku `.spec.js`, `testManager` wyświetla dostępne warianty testów:

```
suduvis@suduvis-V0C:~/Projects/test-builder$ npm run manager

> manager
> node testManager.js

Welcome to testManager CLI!

✓ Select a test file: tests/home/home.spec.js
? Select a test variant: (Use arrow keys)
> newInteraction
  anotherVariant
  booking
  main
  _____
  Back to file selection
```

Rysunek 5.4: Menu wyboru wariantu w `testManager`. Można też wrócić do wyboru pliku `.spec.js`.

Dla wybranego wariantu `testManager` prezentuje bieżącą konfigurację (widok, interakcje, wymiary itp.) w formie tabeli:

Current config for this variant:

Property	Value
pageRoute	/
viewName	home
viewport	desktop, mobile
colorSchemes	light, dark
viewPortResolution	{ "desktop": { "width": 1396, "height": 480 }, "mobile": { "width": 600, "height": 480 } }
customDarkCSS	body { background: grey; }
onlyThis	false
pageInteraction	await page.getByRole('button', { name: 'Book this room' }).nth(1).click(); await page.getByPlaceholder('Firstname').click(); await page.getByPlaceholder('Firstname').fill('kk');

? What would you like to do with setPageInteraction? (Use arrow keys)

Generate new code via codegen + override setPageInteraction

Remove setPageInteraction

> Back to variant selection

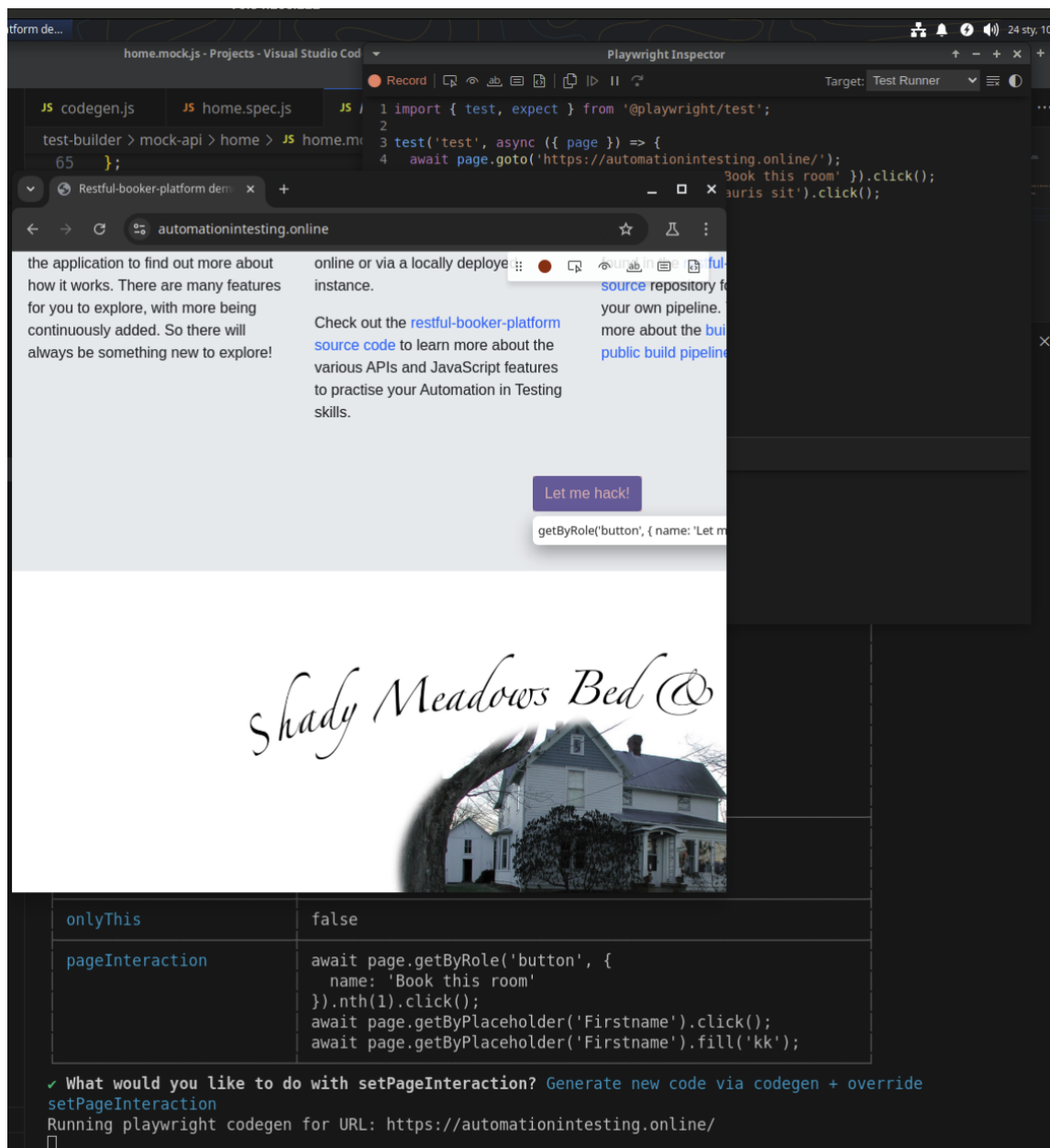
Exit

Rysunek 5.5: Przykładowa tabelaryczna prezentacja stanu wariantu w `testManager`.

Następnie użytkownik może skorzystać z opcji:

- **Generate new code via codegen + override setPageInteraction:** wywołuje wewnętrznie *Playwright codegen*, który otwiera przeglądarkę i rejestruje nowe czynności wykonywane przez testera. Po zakończeniu nagrywania `testManager` wstawia wygenerowany kod do `.spec.js`, zastępując dotychczasowy.
- **Remove setPageInteraction:** usuwa z pliku `.spec.js` definicję akcji użytkownika dla bieżącego wariantu.
- **Back to variant selection / Exit:** pozwala wrócić do poprzedniego kroku lub wyjść z narzędzia.

Rys. 5.6 ilustruje moment, w którym `testManager` uruchamia przeglądarkę i *Playwright codegen*, a użytkownik może „nagrać” nową sekwencję interakcji.



Rysunek 5.6: Przykład równoczesnego działania CLI `testManager` i *Playwright codegen* (przeglądarka).

Podsumowując, `testManager` upraszcza proces edycji i tworzenia testów:

1. Interaktywnie prowadzi użytkownika przez wybór plików `.spec.js` i wariantów,
2. Umożliwia szybkie przełączanie się między wariantami testów (bez konieczności ręcznego grzebania w kodzie),
3. Zapewnia integrację z *Playwright codegen*, dzięki czemu można łatwo „nagrać” nowe czynności na stronie i dołączyć je do testu,
4. Pozwala na podgląd aktualnej konfiguracji (rozmiary ekranu, tryby kolorystyczne, akcje `pageInteraction` itp.) w czytelnej tabeli.

Dzięki temu programiści oraz testerzy mogą w prosty sposób zarządzać dużą liczbą testów wizualnych (screen testów) i błyskawicznie wprowadzać zmiany w interakcjach wykonywanych przez test.

Rozdział 6

Kluczowe mechanizmy i biblioteki

W niniejszym rozdziale omówione zostaną kluczowe elementy rozwiązania wspierającego automatyzację testów aplikacji webowych. Przedstawiono tutaj sposób organizacji zasobów projektowych (folderów i plików), zasady definiowania nazw zrzutów ekranu oraz kluczowe mechanizmy umożliwiające generowanie i wykonywanie testów E2E w oparciu o narzędzie **Playwright**. Opisano też rolę menedżera testów (**testManager**) wraz z integracją mechanizmu **codegen**, a także koncepcję weryfikacji plików **mock** względem faktycznych odpowiedzi serwera (**mockTest**). Na zakończenie rozdziału podkreślono, jakich bibliotek użyto do implementacji poszczególnych modułów oraz wskazano najważniejsze miejsca w kodzie, w których zaimplementowano prezentowane mechanizmy.

6.1 Zasady organizacji projektu

6.1.1 Struktura folderów i plików projektu; nazewnictwo zrzutów ekranu

W projekcie wyróżnić można kilka istotnych katalogów, zorganizowanych w taki sposób, aby ułatwić utrzymanie testów, mocków oraz plików konfiguracyjnych:

- **/tests** – tutaj znajdują się właściwe testy `.spec.js` wraz z wygenerowanymi zrzutami ekranu. Każdy scenariusz testowy może zostać osadzony w dedykowanym podfolderze (np. `/tests/home`, `/tests/cookie`), co pozwala na tematyczne grupowanie testów.
 - Każdy `.spec.js` test może generować wiele zrzutów ekranu. Zrzuty te, zgodnie z konfiguracją, przechowywane są w tym samym folderze, co plik testowy (np. `home.spec.js` obok plików `home-desktop-light-chromium.png`).
- **/mock (mock-api)** – w tym katalogu znajdują się pliki `.mock.js` opisujące zachowania poszczególnych endpointów (np. `home.mock.js`, `cookie.mock.js`). Dzięki nim testy mogą działać bez konieczności nawiązywania połączenia z realnym środowiskiem serwerowym.
- **config.js**, **playwright.config.js**, **docker-compose.playwright.yml** – pliki odpowiedzialne za konfigurację środowiska, ustawienia **Playwright** oraz ewentualną orkiestrację w kontenerach **Docker**.

- `testManager.js`, `screenshotTestBuilder.js`, `mockTest.js` – najistotniejsze skrypty odpowiedzialne za mechanizmy opisywane w niniejszym rozdziale, tj. zarządzanie testami, generowanie zrzutów ekranu i weryfikację mock.

Sposób nazywania zrzutów ekranu. Wygenerowane zrzuty ekranu w katalogu `/tests` posiadają nazwy zgodne ze schematem:

`viewName-viewport-colorscheme-variant-przegladarka.png`

gdzie:

- `viewName` – nazwa widoku bądź strony, np. `home` lub `cookie`;
- `viewport` – informacja o rodzaju wyświetlacza, np. `desktop` bądź `mobile`;
- `colorscheme` – `light` lub `dark`, zależnie od tematu kolorystycznego;
- `variant` – opcjonalna etykieta, np. `booking` oznaczająca konkretny scenariusz w obrębie tej samej strony;
- `przegladarka` – wskazanie, w jakiej przeglądarce dany zrzut został wykonany (np. `chromium`, `firefox`, `webkit`).

Przykładowo nazwa `home-desktop-light-booking-chromium.png` oznacza, że jest to zrzut strony `home`, wykonany w trybie `desktop`, przy jasnej kolorystyce (`light`) i wariantie testu `booking`, w przeglądarce `chromium`. Taki sposób nazywania ułatwia szybkie odnalezienie odpowiedniej kombinacji parametrów wśród wielu plików screenshotów.

6.1.2 Mechanizm tworzenia testów E2E przez `screenshotTestBuilder`

Proces tworzenia testów E2E z wykorzystaniem `Playwright` jest w projekcie wspierany przez komponent `screenshotTestBuilder`. Pomysł polega na tym, aby w jednym miejscu deklarować *parametry* testu (np. jaką stronę należy odwiedzić, w jakich wymiarach przeglądarki, z jakim schematem kolorów, jakie interakcje powinny zostać wykonane), a następnie na tej podstawie automatycznie zbudować gotowy skrypt `.spec.js`.

W praktyce wygląda to następująco:

1. **Inicjacja buildera:** w pliku `.spec.js` tworzy się nowy obiekt `ScreenshotTest()`, co stanowi punkt wyjścia do konfiguracji testu.
2. **Ustawianie parametrów:** za pomocą metod takich jak `forPage()`, `forViewports()`, `forColorSchemes()` programista wskazuje, jakie warianty (viewports, tryby kolorystyczne) mają zostać uwzględnione.
3. **Opcjonalne interakcje:** `setPageInteraction()` pozwala zdefiniować akcje wykonywane na stronie (np. kliknięcia, wypełnianie formularza) przed zrobieniem zrzutu ekranu.
4. **Deklaracja testu:** wywołanie `test()` zamyka konfigurację i generuje serię rzeczywistych testów `Playwright` (po jednym na każdą kombinację viewportu, trybu kolorystycznego, nazwy wariantu itd.).

5. **Uruchomienie:** w trakcie wykonywania testów (`npx playwright test`) aplikacja odwiedza wskazaną stronę, wywołuje zadane interakcje i zapisuje zrzuty ekranu w odpowiednich plikach (zgodnie z opisanym wcześniej schematem nazewnictwa).

Dzięki temu mechanizmowi programista nie musi tworzyć wielu osobnych plików testowych dla każdej kombinacji rozdzielczości, koloru czy wariantu scenariusza. Zamiast tego jeden builder potrafi wygenerować i uruchomić szereg analogicznych testów o wspólnych parametrach. Dodatkowo, `ScreenshotTestBuilder` pozwala na *eksport* konfiguracji (np. by wykorzystać ją w `testManager`), co znacznie ułatwia edycję i utrzymanie testów w dłuższym okresie.

6.1.3 Mechanizm `testManager` z wykorzystaniem codegen

Mechanizm, w którym `testManager` wykorzystuje codegen do tworzenia skryptu z interakcją na stronie, bazujący na obiekcie `page`, przebiega następująco:

1. **Wybór pliku i wariantu testu w CLI:** Użytkownik uruchamia `testManager` w trybie wiersza poleceń. Narzędzie skanuje katalog `/tests`, odnajduje wszystkie pliki `.spec.js` i wyświetla ich listę. Po wybraniu konkretnego pliku, `testManager` pokazuje dostępne warianty testu (zdefiniowane metodą `test('nazwaWariantu')` w kodzie `.spec.js`).
2. **Wywołanie opcji generowania interakcji:** Gdy użytkownik chce zmodyfikować lub dodać nową sekwencję interakcji w danym wariantcie, wybiera w menu opcję „Generate new code via codegen + override `setPageInteraction`”.
3. **Uruchomienie *Playwright codegen*:** `testManager` w tle wykonuje polecenie `npx playwright codegen <URL>`, gdzie `<URL>` to z reguły `basePageUrl + pageRoute` (ustalone w konfiguracji testu). Otwiera się przeglądarka w trybie interaktywnym oraz dodatkowe okno z wygenerowanym kodem.
4. **Rejestrowanie akcji użytkownika:** Wszystkie kliknięcia, wypełnienia pól formularza, nawigacja itp. są w czasie rzeczywistym zapisywane w kodzie JavaScript/TypeScript (zależnie od ustawień). *Playwright codegen* generuje m.in. sekwencje wywołań `page.click()`, `page.fill()` i tym podobnych.
5. **Zapisanie wygenerowanego kodu:** Po zakończeniu „nagrywania” interakcji, codegen tworzy tymczasowy plik, z którego `testManager` odczytuje potrzebne fragmenty (zwykle samą zawartość funkcji asynchronicznej wykorzystującej obiekt `page`).
6. **Nadpisanie `setPageInteraction` w pliku `.spec.js`:** `testManager` analizuje obecny kod `.spec.js`, odnajduje właściwy blok `setPageInteraction(async (page) => {...})` dla wybranego wariantu testu i wstawia tam nowo wygenerowaną treść (czyli ciąg wywołań `page.XXX()`).
7. **Formatowanie i zapis zmian:** Na końcu `testManager` może wywołać `prettier` (jeśli jest dostępny), aby kod testu zachował spójny styl. Zmodyfikowany plik `.spec.js` zostaje nadpisany, co oznacza, że kolejny `test:e2e` będzie wykonywał już zaktualizowane kroki.

8. **Potwierdzenie w CLI:** Użytkownik widzi w `testManager` informację, że interakcja została zmieniona. Może teraz w dowolnym momencie uruchomić `npx playwright test` (albo skorzystać z pipeline’u CI) i sprawdzić działanie nowego scenariusza.

Dzięki temu procesowi `testManager` *integruje* się z wbudowanym w Playwright narzędziem `codegen`, pozwalając w szybki sposób tworzyć lub modyfikować *krok po kroku* akcje `page`, które zostaną wykonane w testach. Jest to szczególnie przydatne dla osób, które nie chcą ręcznie pisać skryptów interakcji z przeglądarką, a wolą nagrać je przez klikanie w interfejs.

6.1.4 Mechanizm działania `mockTest`

Moduł `mockTest` sprawdza, czy pliki `.mock.js` (odpowiedzialne za symulowanie wywołań API) są zgodne ze strukturą i danymi zwracanymi przez realne API. Mechanizm jego działania można podzielić na następujące kroki:

1. **Wczytanie wszystkich plików `.mock.js`:** `mockTest` rekurencyjnie przegląda katalog `mock-api` i importuje każdy plik definiujący `mockApi` oraz `mockApiPresets`. Dzięki temu uzyskuje listę wszystkich endpointów, które w testach mogą być symulowane.
2. **Dla każdego endpointu - pobranie faktycznej odpowiedzi API:** `mockTest` odczytuje z `mockApi` m.in. `endpoint`, `query` oraz ewentualny `apiUrl`, a następnie wykonuje żądanie HTTP do prawdziwego serwera (zdefiniowanego np. w `config.js` jako `baseApiUrl`).
3. **Porównanie *Content-Type* i danych:**
 - Jeśli `contentType` to obraz (`image/png`, `image/jpeg`), `mockTest` pobiera dane w postaci binarnej i porównuje bajt po bajcie z tym, co jest w `mockApi`.
 - Jeśli `contentType` to `application/json`, `mockTest` konwertuje odpowiedź na obiekt JSON i sprawdza, czy struktura (schemat) odpowiada danym określonym w `mockApi`. Wykorzystywane są do tego takie narzędzia jak `ajv` (JSON schema validator) czy `quicktype-core` (do generowania schematu na podstawie przykładowych danych).
4. **Walidacja różnic i raportowanie:** Jeśli `mockTest` wykryje, że *struktura* JSON różni się od tej, którą deklaruje `mockApi` (np. brakuje jakiegoś pola w obiekcie, ma inny typ lub nazwa klucza się nie zgadza), wówczas generuje komunikat o błędzie. Analogicznie w przypadku obrazów – jeśli zawartość bajtów nie pasuje, test również uznawany jest za niepoprawny.
5. **Informacja zwrotna na temat ewentualnych poprawek:** Programista po uruchomieniu `node mockTest.js` dostaje w konsoli informację, czy wszystkie mocki są zgodne z aktualnym stanem API, a jeśli nie – w których konkretnie polach czy plikach `mock.js` pojawiły się rozbieżności.

Istotną zaletą `mockTest` jest to, że *uniemożliwia* długotrwałe utrzymywanie nieaktualnych plików `mock.js`, co zapobiega sytuacji, w której testy przechodzą tylko dlatego, że

używają zastarzałej, błędnej symulacji API. W momencie, gdy rzeczywiste API ulega zmianie (np. dodaje nowe pola w JSON), od razu wiadomo, że `mockTest` tego nie przepuści i wymusi dostosowanie `mockApi` do nowego formatu.

6.2 Omówienie kluczowych bibliotek

W projekcie wykorzystano szereg bibliotek oraz narzędzi, które wspomagają proces automatyzacji testów aplikacji webowych, zarówno od strony przygotowywania i uruchamiania scenariuszy E2E, jak i zarządzania nimi czy walidacji danych zwracanych przez API. Poniżej przedstawiono najważniejsze z nich wraz z opisem roli, jaką pełnią w systemie:

- **Playwright** (`@playwright/test`) – podstawowa technologia do testów E2E w tym projekcie. Umożliwia uruchamianie testów w przeglądarkach Chromium, Firefox oraz WebKit. Dostarcza wbudowanego *test runnera*, obsługę zrzutów ekranu, generowanie raportów (także w trybie interfejsu graficznego) oraz narzędzie *codegen* do interaktywnego rejestrowania akcji użytkownika.
- **Inquirer** – biblioteka do tworzenia interaktywnych menu w wierszu poleceń (CLI). Z jej pomocą zaimplementowano `testManager`, który pyta użytkownika o wybór plików `.spec.js`, wariantów testów oraz pozwala na zarządzanie interakcjami.
- **Chalk** – pakiet do kolorowania i formatowania tekstu w konsoli. Dzięki niemu `testManager` może wyświetlać wyróżnione komunikaty (np. z powodzeniem, ostrzeżenia, błędy), zwiększając czytelność i wygodę obsługi narzędzia.
- **Prettier** – narzędzie do automatycznego formatowania kodu JavaScript/TypeScript. Po nagraniu akcji w *Playwright codegen*, `testManager` może wywołać `prettier` w celu znormalizowania stylu wygenerowanego kodu i wstawienia go w pliku `.spec.js`. Ułatwia to utrzymanie jednolitego stylu całego repozytorium.
- **Babel** (w szczególności `@babel/parser`, `@babel/traverse`, `@babel/types` i `@babel/generator`) – zestaw narzędzi do parsowania kodu JavaScript i dokonywania w nim modyfikacji w postaci drzewa AST (*Abstract Syntax Tree*). W `testManager` wykorzystuje się Babel do:
 1. Odczytu istniejących plików `.spec.js` i odnajdywania miejsc, w których występują wywołania `test()` oraz `setPageInteraction(...)`.
 2. Nadpisywania fragmentów kodu, np. wstawiania nowo wygenerowanej sekwencji akcji z *codegen* (tj. ciągu wywołań `page.click(...)`, `page.fill(...)` itp.).
 3. Generowania zaktualizowanego pliku testowego, który zostaje ostatecznie zapisany na dysku.
- **cli-table3** – pakiet pozwalający wyświetlać w konsoli dane w formie tabelarycznej. W `testManager` używany do prezentacji aktualnej konfiguracji wybranego wariantu testu (np. rozmiar viewportu, kolor, nazwa interakcji). Ułatwia to szybki wgląd w kluczowe parametry.

- **AJV** (*Another JSON Validator*) oraz **ajv-formats** – biblioteki do walidacji struktury danych w formacie JSON. Służą one `mockTest` do sprawdzania, czy dane zwracane przez `mockApi` pokrywają się ze schematem danych realnego API.
- **quicktype-core** – narzędzie umożliwiające na podstawie przykładowego JSON-u (pobranego z produkcyjnego API) automatyczne wyprowadzenie schematu (*JSON Schema*), który następnie jest weryfikowany z `mockApi`. `mockTest` korzysta z *quicktype-core* do inferencji tego schematu w locie i porównywania go z danymi zadeklarowanymi w pliku `.mock.js`.
- **node-fetch** – biblioteka do wykonywania zapytań HTTP po stronie Node.js. Wykorzystywana w `mockTest` przy pobieraniu realnych danych z API, aby sprawdzić aktualną strukturę i porównać ją z zawartością `mockApi`.
- **Docker / Docker-Compose** – środowisko konteneryzacji (z plikiem `docker-compose.playwright.yml`), w którym można uruchomić testy E2E w odizolowanym kontenerze. Pozwala to zapewnić spójne i powtarzalne warunki uruchomieniowe (m.in. zdefiniowana wersja `mcr.microsoft.com/playwright`).

Wszystkie wymienione wyżej biblioteki są zdefiniowane w pliku `package.json` (sekcja `dependencies`) i instalowane wraz z projektem (`npm install`). Ich dobór podyktowany był dążeniem do zapewnienia:

- **Spójnego ekosystemu testowego** – *Playwright* gwarantuje solidną podstawę do automatyzacji E2E, wspieraną przez narzędzia do walidacji i generowania scenariuszy.
- **Wysokiej czytelności i automatyzacji** – *Inquirer*, *Chalk* i *cli-table3* umożliwiają tworzenie rozbudowanego, przyjaznego CLI, a *Babel* i *Prettier* pozwalają na automatyczną modyfikację i formatowanie kodu.
- **Łatwej integracji z różnymi środowiskami** – *Docker-Compose* w połączeniu z obrazem `playwright` ułatwia uruchamianie testów na dowolnej maszynie, w tym w pipeline'ach CI/CD.

Zastosowanie powyższych bibliotek i narzędzi w sposób synergiczny pozwoliło na uzyskanie rozwiązania, w którym zarówno interaktywna konfiguracja, jak i sam proces testowania (od generowania kodu aż po sprawdzanie poprawności danych) działają płynnie i elastycznie.

6.2.1 Kluczowe miejsca w kodzie

W poniższym podrozdziale zaprezentowano i omówiono cztery kluczowe fragmenty kodu źródłowego projektu, które w szczególny sposób ilustrują mechanizmy związane z tworzeniem testów, generowaniem interakcji i mockowaniem odpowiedzi API. Są to:

- metoda `#mockApiCall`,
- metoda `test(variantName)`,
- funkcja `inferSchemaFromJSON(jsonData)`,

- funkcja `codegenAndExtract(pageUrl)`.

Każdy fragment został przytoczony w formie kodu oraz wzbogacony o komentarz wyjaśniający jego działanie i zastosowanie w projekcie.

Metoda `#mockApiCall`

```
async #mockApiCall(page) {
  const mockApi = await getMockDataFor(this.#viewName);
  const { mockApiPresets } = mockApi;

  for (const {
    endpoint,
    data,
    contentType,
    customQuery = '',
    apiUrl,
  } of mockApiPresets.default) {
    await page.route(
      `${apiUrl || config.baseApiUrl}/${endpoint}${customQuery}`,
      async (route) => {
        if (contentType === 'text/html') {
          await route.fulfill({
            contentType: 'text/html',
            body: data,
          });
        } else if (contentType?.startsWith('image/')) {
          await route.fulfill({
            contentType: contentType,
            body: data,
          });
        } else {
          await route.fulfill({ json: data });
        }
      }
    );
  }
}
```

Opis działania:

Powyższa metoda `#mockApiCall` (zaimplementowana w klasie `ScreenshotTestBuilder`) umożliwia podstawienie zdefiniowanych wcześniej *mocków* API zamiast prawdziwych wywołań sieciowych. Dzięki temu test nie musi odwoływać się do realnego serwera. Mechanizm w kolejnych krokach:

1. Odczytuje dane z pliku `.mock.js` przypisanego do aktualnie testowanego widoku (`this.#viewName`).
2. Iteruje po wszystkich wpisach w `mockApiPresets.default`, które opisują m.in. `endpoint`, `contentType` i same `data`.

3. Dla każdej ścieżki `endpoint` (oraz parametrów `query`) rejestruje `page.route`, aby przechwycić żądanie do API w trakcie testu.
4. Wywołuje `route. fulfill(...)`, zwracając przygotowany *mock*:
 - tekst/HTML (`'text/html'`),
 - obraz (`'image/...'`) lub
 - obiekt JSON.

Dzięki temu możliwe jest wiarygodne testowanie *front-endu* bez dostępu do prawdziwych zasobów sieciowych, a wyniki testów (np. zrzuty ekranu) nie są zależne od stanu środowiska zewnętrznego.

Metoda `test(variantName)`

```
test(variantName) {
  if (!this.#pageRoute) throw new Error('Page route is not set');

  const isCli = process.env.SCREENSHOT_TEST_BUILDER_CLI === 'true';

  if (!isCli) {
    const testFunction = this.#onlyThis ? t.only : t;

    const testCases = variantName ? [variantName] : [null];
    const testState = {
      pageInteraction: this.#pageInteraction,
    };

    for (const viewPort of this.#viewport) {
      for (const colorScheme of this.#colorSchemes) {
        for (const variant of testCases) {
          testFunction(
            this.#getTestDescription(viewPort, colorScheme, variant),
            async ({ page }) => {
              await this.#mockApiCall(page);
              await this.#setViewportFor(viewPort, page);
              await this.#setColorScheme(colorScheme, page);
              await page.goto(this.#pageRoute);
              await testState.pageInteraction?.(page);

              await expect(page).toHaveScreenshot(
                this.#getReferenceFileFor(viewPort, colorScheme,
                  ↪ variant),
                { fullPage: true }
              );
            }
          );
        }
      }
    }
  }
}
```

```

    }
  }

  const usedVariantName = variantName || 'main';
  this.#variantsState[usedVariantName] = this.exportState();

  this.#resetState();
  return this;
}

```

Opis działania:

Metoda `test(variantName)` pełni kluczową rolę w generowaniu i wykonywaniu właściwych testów w oparciu o Playwright. Każdorazowe wywołanie:

- Sprawdza, czy określono `this.#pageRoute` (adres strony do odwiedzenia).
- Rozróżnia, czy test ma zostać uruchomiony normalnie, czy w trybie CLI (`isCli`).
- Dla zadanych kombinacji `viewPort`, `colorSchemes` oraz dla konkretnej nazwy wariantu (`variantName` lub `null`) tworzy wewnątrz pętlę:
 - `testFunction(...)` to tak naprawdę wywołanie `test(...)` lub `test.only(...)` z Playwright (`t.only`).
 - Wywołuje `#mockApiCall`, by **podmieni**ć ewentualne wywołania sieciowe.
 - Ustawia rozdzielczość (`#setViewportFor`) oraz schemat kolorów (`#setColorScheme`).
 - Otwiera stronę `this.#pageRoute` (`page.goto(...)`) i wykonuje *interakcję* (`this.#pageInteraction`).
 - Kończy się asercją `expect(page).toHaveScreenshot(...)`, co generuje i porównuje zrzut ekranu.
- Na zakończenie zapisuje stan konfiguracji (`#variantsState`) i czyści `#pageInteraction` (`#resetState`).

W efekcie programista może wywołać `.test(...)` dla różnych wariantów (np. 'booking'), a każdy z nich spowoduje wykonanie jednej lub wielu kombinacji testowych (różne przeglądarki, rozdzielczości, kolory).

Funkcja `inferSchemaFromJSON(jsonData)`

```

async function inferSchemaFromJSON(jsonData) {
  const jsonInput = jsonInputForTargetLanguage('schema');
  await jsonInput.addSource({
    name: 'GeneratedSchema',
    samples: [JSON.stringify(jsonData)],
  });

  const inputData = new InputData();
  inputData.addInput(jsonInput);
}

```

```

const { lines } = await quicktype({
  inputData,
  lang: 'schema',
});

return lines.join('\n');
}

```

Opis działania:

Funkcja `inferSchemaFromJSON` (obecna w `mockTest.js`) korzysta z biblioteki *quicktype-core* w celu automatycznego wyprowadzenia schematu (JSON Schema) na podstawie przykładowych danych JSON. W skrócie:

1. Tworzony jest obiekt `jsonInputForTargetLanguage('schema')`, do którego *dodaje się* przykładowe dane `jsonData`.
2. Następnie `inputData` (klasa `InputData` z *quicktype*) zostaje wypełniona tym źródłem.
3. Wywołanie `quicktype(...)` generuje linie (tekst) zawierające schemat w formacie JSON Schema.
4. Funkcja zwraca całość w formie spójnego łańcucha znaków (`lines.join('')`).

Ten schemat jest dalej wykorzystywany do walidacji *mocków* w `mockTest`, aby sprawdzić, czy struktura `mockApi` **odpowiada** realnemu formatowi z produkcyjnego API.

Funkcja `codegenAndExtract(pageUrl)`

```

export async function codegenAndExtract(pageUrl) {
  const inputFile = path.join(__dirname, 'input.js');

  try {
    console.log(`Running playwright codegen for URL: ${pageUrl}`);
    await executeCommand(
      `npx playwright codegen ${pageUrl} --output=${inputFile}`
    );
    console.log('Playwright codegen completed.');

    const extractedCode = await parseInputFile(inputFile);

    await fs.promises.unlink(inputFile);
    console.log(`Removed temporary file: ${inputFile}`);

    return extractedCode;
  } catch (err) {
    console.error('An error occurred during codegen:', err);
    throw err;
  }
}

```


Opis działania:

Funkcja `codegenAndExtract` (z pliku `codegen.js`) stanowi interfejs do wbudowanego w Playwright `codegen`, który nagrywa akcje wykonywane w przeglądarce:

1. Uruchamia zewnętrzny proces `npx playwright codegen` dla podanego `pageUrl`, wskazując `--output` na tymczasowy plik `input.js`.
2. Po zakończeniu rejestracji `codegen` wytwarza kod (zwykle zestaw wywołań `page.click(...)`, `page.fill(...)` itp.).
3. Następnie `parseInputFile` (wewnętrzna funkcja) wyodrębnia istotne fragmenty ze wspomnianego `input.js`.
4. Plik tymczasowy jest usuwany, a funkcja zwraca gotowy kod (*string*) – który może być wstawiony np. do `setPageInteraction` przez `testManager`.

Dzięki temu podejściu można *nagrywać* kroki wykonywane na stronie, a następnie z poziomu narzędzia CLI `testManager` wstrzykiwać ten kod (interakcje) *bezpośrednio* do plików `.spec.js`, co przyspiesza i upraszcza proces tworzenia testów end-to-end.

Rozdział 7

Przykłady użycia aplikacji

W niniejszym rozdziale zaprezentowano, w jaki sposób można wykorzystać opisywane narzędzie w praktyce na przykładzie prostej aplikacji webowej. Przedstawiono rzeczywiste zrzuty ekranu z uruchomień testów w konsoli, raportów błędów w *Playwright*, a także proces walidacji plików `.mock.js` w odniesieniu do prawdziwego API. Dodatkowo ukazano przykładowy *screenshot* strony testowej (`home-desktop-light-booking-chromium.png`) oraz fragmenty kodu testów (`home.spec.js`, `home.mock.js`).

7.1 Uruchamianie narzędzia w trybie interaktywnym

W typowym scenariuszu deweloper uruchamia testy za pomocą komendy:

```
npm run test:e2e
```

co powoduje wywołanie testów w trybie konsolowym (*Playwright*). W przypadku chęci skorzystania z *UI Playwright* (interfejs graficzny testów) używana jest komenda:

```
npm run test:e2e:ui
```

W dalszej części pokazano przykłady logów i raportów z takich uruchomień:

```

16 failed
[chromium] > ../screenshotTestBuilder.js:115:13 > home in @desktop viewport with @light color scheme
[chromium] > ../screenshotTestBuilder.js:115:13 > home in @desktop viewport with @dark color scheme
[chromium] > ../screenshotTestBuilder.js:115:13 > home in @mobile viewport with @light color scheme
[chromium] > ../screenshotTestBuilder.js:115:13 > home in @mobile viewport with @dark color scheme
[chromium] > ../screenshotTestBuilder.js:115:13 > home in @desktop viewport with @light color scheme, @
@booking variant
[chromium] > ../screenshotTestBuilder.js:115:13 > home in @desktop viewport with @dark color scheme, @
booking variant
[chromium] > ../screenshotTestBuilder.js:115:13 > home in @mobile viewport with @light color scheme, @
booking variant
[chromium] > ../screenshotTestBuilder.js:115:13 > home in @mobile viewport with @dark color scheme, @b
ooking variant
[chromium] > ../screenshotTestBuilder.js:115:13 > home in @desktop viewport with @light color scheme, @
@anotherVariant variant
[chromium] > ../screenshotTestBuilder.js:115:13 > home in @desktop viewport with @dark color scheme, @
anotherVariant variant
[chromium] > ../screenshotTestBuilder.js:115:13 > home in @mobile viewport with @light color scheme, @
anotherVariant variant
[chromium] > ../screenshotTestBuilder.js:115:13 > home in @mobile viewport with @dark color scheme, @a
notherVariant variant
[chromium] > ../screenshotTestBuilder.js:115:13 > home in @desktop viewport with @light color scheme, @
@newInteraction variant
[chromium] > ../screenshotTestBuilder.js:115:13 > home in @desktop viewport with @dark color scheme, @
newInteraction variant
[chromium] > ../screenshotTestBuilder.js:115:13 > home in @mobile viewport with @light color scheme, @
newInteraction variant
[chromium] > ../screenshotTestBuilder.js:115:13 > home in @mobile viewport with @dark color scheme, @n
ewInteraction variant
4 passed (2.7m)

Serving HTML report at http://localhost:9323. Press Ctrl+C to quit.

```

Rysunek 7.1: Fragment logu uruchomienia testów E2E w trybie konsolowym. Widoczne są m.in. błędy nieudanego testu zrzutu ekranu (16 testów niepowodzeń i 4 zakończone sukcesem).

Na rysunku 7.1 widzimy przykładowy log z linii poleceń, gdzie *Playwright* raportuje niezgodności względem oczekiwanego zrzutu ekranu (`toHaveScreenshot`). Przyczyny błędów mogą wynikać np. ze zmian w układzie strony lub dodania nowych elementów, co sprawia, że generowany obraz różni się od wzorca.

Raport nieudanego testu wizualnego

Po zakończeniu testów w trybie *UI* bądź w trybie *console* (*headless*), możliwe jest obejrzanie raportów w formacie HTML. Rysunek 7.2 ilustruje fragment takiego raportu, w którym *Playwright* wskazuje różnice między oczekiwanym a aktualnym zrzutem ekranu.

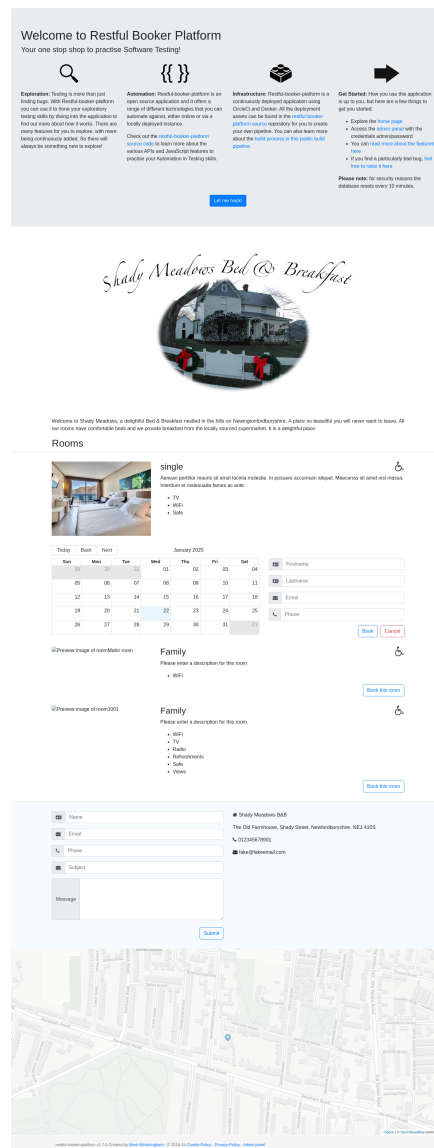
Z raportu (rys. 7.2) wynika m.in. że oczekiwano obrazu o wymiarach 1396px x 3367px, lecz w trakcie testu uzyskano 1396px x 2868px. Tego typu różnica może wystąpić w sytuacji, gdy nie załadowała się dana sekcja strony albo zmieniono styl CSS powodujący inny rozmiar.

Przykładowy *screenshot* testowanej aplikacji

Poniżej zamieszczono przykładowy zrzut ekranu testowanej aplikacji (home-desktop-light-booking-chromium.png). Ten plik jest generowany w folderze `tests/home/` podczas uruchamiania testów `home.spec.js`. Zgodnie z konwencją nazewnictwa:

home-desktop-light-booking-chromium.png

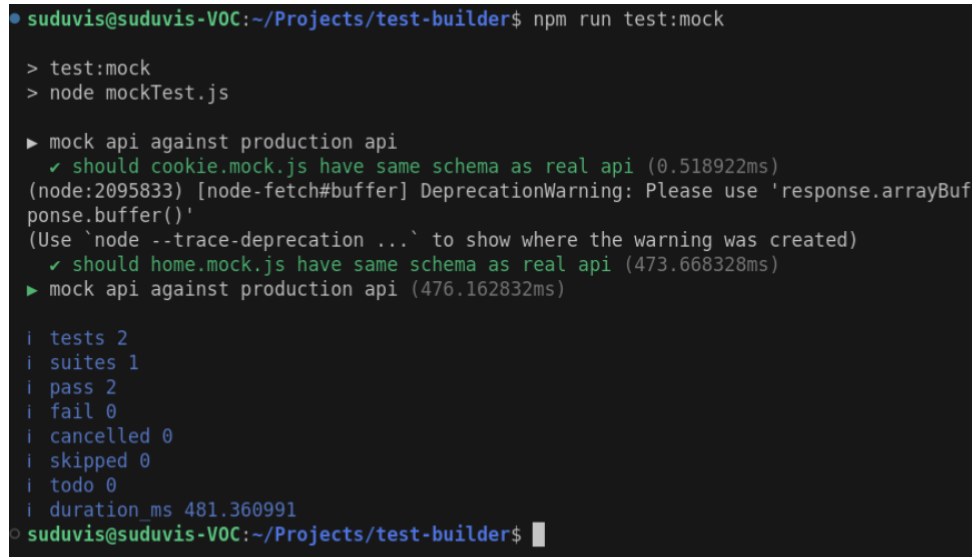
oznacza, że jest to widok `home` w trybie *desktop*, *light* i wariancie *booking*, uruchomionym w przeglądarce *chromium*.



Rysunek 7.3: Przykładowy screenshot aplikacji home (desktop, jasny motyw, wariant *booking*).

Testy mockApi – przykłady sukcesu i błędu

Poza samymi testami *E2E* ważną częścią narzędzia jest weryfikacja plików `.mock.js` względem rzeczywistego API (komenda `npm run test:mock`). Rysunek 7.4 przedstawia przykład poprawnych testów mock, a rysunek 7.5 pokazuje sytuację, gdy `mockApi` nie jest zgodne z aktualnym schematem serwera.



```
● suduvis@suduvis-VOC:~/Projects/test-builder$ npm run test:mock

> test:mock
> node mockTest.js

▶ mock api against production api
  ✓ should cookie.mock.js have same schema as real api (0.518922ms)
(node:2095833) [node-fetch#buffer] DeprecationWarning: Please use 'response.arrayBuffer()'
  (Use 'node --trace-deprecation ...' to show where the warning was created)
  ✓ should home.mock.js have same schema as real api (473.668328ms)
▶ mock api against production api (476.162832ms)

i tests 2
i suites 1
i pass 2
i fail 0
i cancelled 0
i skipped 0
i todo 0
i duration_ms 481.360991
● suduvis@suduvis-VOC:~/Projects/test-builder$
```

Rysunek 7.4: Przykład udanego uruchomienia testów `mockTest.js`, w którym zweryfikowano poprawność 2 plików `.mock.js`.

```

--- Schema Validation Failed for https://automationintesting.online/branding/ ---
Error 1:
  Field: N/A
  Item Index: N/A
  Invalid Value: "N/A"
  Entire Object: "N/A"
  Message: must be number,null
  Schema Path: #/definitions/Map/properties/latitude/type
-----
* should home.mock.js have same schema as real api (356.416966ms)
  AssertionError [ERR_ASSERTION]: Schema validation failed for branding/
    at TestContext.<anonymous> (file:///home/suduvis/Projects/test-builder/mockT
    at process.processTicksAndRejections (node:internal/process/task_queues:95:5)
    at async Test.run (node:internal/test_runner/test:632:9)
    at async Suite.processPendingSubtests (node:internal/test_runner/test:374:7)
    generatedMessage: false,
    code: 'ERR_ASSERTION',
    actual: undefined,
    expected: undefined,
    operator: 'fail'
  }

▶ mock api against production api (361.768857ms)

i tests 2
i suites 1
i pass 1
i fail 1
i cancelled 0
i skipped 0
i todo 0
i duration_ms 366.870316

* failing tests:

test at file:/home/suduvis/Projects/test-builder/mockTest.js:31:5
* should home.mock.js have same schema as real api (356.416966ms)
  AssertionError [ERR_ASSERTION]: Schema validation failed for branding/
    at TestContext.<anonymous> (file:///home/suduvis/Projects/test-builder/mockTes
    at process.processTicksAndRejections (node:internal/process/task_queues:95:5)
    at async Test.run (node:internal/test_runner/test:632:9)
    at async Suite.processPendingSubtests (node:internal/test_runner/test:374:7) {
    generatedMessage: false,
    code: 'ERR_ASSERTION',
    actual: undefined,
    expected: undefined,
    operator: 'fail'
  }
}
suduvis@suduvis-VOC:~/Projects/test-builder$

```

Rysunek 7.5: Przykład błędu w `mockTest.js` – struktura zwracanego `branding/` nie zgadza się z oczekiwaną (pole `latitude` nie jest typu `number/null`).

Jak widać, dzięki `mockTest` można w porę wykryć nieaktualne mocki, które mogłyby wprowadzać w błąd testy *E2E*.

7.1.1 Przykładowy kod `home.spec.js`

Poniżej zamieszczono przykładowy plik testowy `home.spec.js`, w którym zdefiniowano trzy warianty testu (*main*, *booking*, *newInteraction*). W każdym z nich można ustawić nieco inne interakcje (`setPageInteraction`) wykonywane przed zrobieniem zrzutu ekranu.

```

===== tests/home/home.spec.js =====
//@ts-check
import ScreenshotTest from '../..screenshotTestBuilder.js';

```

```

new ScreenshotTest()
  .forPage('/', 'home')
  .only()
  .test()
  .setPageInteraction(async (page) => {
    await page.getByRole('button', { name: 'Book this room' }).click();
  })
  .test('booking')
  .test('anotherVariant')
  .setPageInteraction(async (page) => {
    await page.getByTestId('ContactName').fill('hihihi');
  })
  .test('newInteraction');

```

Każdorazowe wywołanie metody `test('nazwa')` generuje serię testów wizualnych w różnych kombinacjach rozdzielczości i wariantów (definiowanych w `screenshotTestBuilder`). W efekcie otrzymujemy m.in. zrzuty ekranu w *desktop/mobile*, trybie *light/dark* oraz – opcjonalnie – w różnych przeglądarkach (*chromium, firefox, webkit*).

7.1.2 Przykładowy kod `home.mock.js`

Poniżej pokazano część pliku `home.mock.js`, odpowiedzialnego za symulowanie odpowiedzi punktu `/branding/` oraz `/room/`. Zawiera on również definicję dwóch plików graficznych (bufor obrazu `room2.jpg` oraz `rbp-logo.jpg`), wczytywanych za pomocą `fs.readFileSync`.

```

===== mock-api/home/home.mock.js =====
import fs from 'fs';

const roomImage = fs.readFileSync(new URL('./room2.jpg',
  ↪ import.meta.url));
const rbpLogo = fs.readFileSync(new URL('./rbp-logo.jpg',
  ↪ import.meta.url));

const mockApi = {
  branding: {
    default: {
      endpoint: 'branding/',
      query: '',
      data: {
        name: 'Shady Meadows B&B',
        map: {
          latitude: 52.6351204,
          longitude: 1.2733774,
        },
        logoUrl: '/images/rbp-logo.jpg',
        description:

```



```

        'Welcome to Shady Meadows, a delightful Bed & Breakfast nestled
        ↪ ...',
    contact: {
        name: 'Shady Meadows B&B',
        address: 'The Old Farmhouse, Shady Street, Newfordburyshire,
        ↪ NE1 410S',
        phone: '012345678901',
        email: 'fake@fakeemail.com',
    },
},
},
},
room: {
    default: {
        endpoint: 'room/',
        query: '',
        data: {
            rooms: [
                {
                    roomid: 1,
                    roomName: '101',
                    type: 'single',
                    accessible: true,
                    image: '/images/room2.jpg',
                    description:
                        'Aenean porttitor mauris sit amet lacinia molestie...',
                    features: ['TV', 'WiFi', 'Safe'],
                    roomPrice: 100,
                },
            ],
        },
    },
},
},
images: {
    room: {
        endpoint: 'images/room2.jpg',
        query: '',
        contentType: 'image/jpeg',
        data: roomImage,
    },
    logo: {
        endpoint: 'images/rbp-logo.jpg',
        query: '',
        contentType: 'image/jpeg',
        data: rbpLogo,
    },
},
};

```

```
const mockApiPresets = {
  default: [
    mockApi['branding'].default,
    mockApi['room'].default,
    mockApi['images'].room,
  ],
};

export { mockApi, mockApiPresets };
```

Widzimy, że plik `home.mock.js` eksportuje `mockApiPresets.default`, dzięki czemu w `screenshotTestBuilder` można jednocześnie podmienić kilka endpointów (`/branding/`, `/room/`, `/images/room2.jpg`) w trakcie testu E2E, bez potrzeby odwoływania się do prawdziwej bazy danych czy usług backendowych.

7.2 Integracja z procesem CI/CD

W niniejszym podrozdziale przedstawiono przykładową konfigurację `.gitlab-ci.yml` w celu uruchamiania testów e2e oraz `mockTest` (zdefiniowanych w pliku `package.json`) w ramach pipeline w GitLab. Dzięki takiemu podejściu, przy każdym wprowadzeniu zmian w repozytorium, testy będą wykonywane automatycznie, co zapewni stałą kontrolę nad jakością aplikacji.

```
stages:
  - test

# Uruchamianie testów e2e za pomocą Playwright
test_e2e:
  stage: test
  image: mcr.microsoft.com/playwright:v1.49.1-jammy
  script:
    - npm ci
    - npm run test:e2e
  artifacts:
    when: always
    paths:
      - playwright-report
    expire_in: 1 day

# Uruchamianie mockTest w celu weryfikacji plików .mock.js
test_mock:
  stage: test
  image: node:18
  script:
    - npm ci
    - npm run test:mock
```

W powyższym przykładzie zdefiniowano jeden *stage* o nazwie `test`, w ramach którego działają dwa zadania (*jobs*):

- `test_e2e` – pobiera zależności (`npm ci`), a następnie uruchamia testy end-to-end (`npm run test:e2e`) z wykorzystaniem obrazu Dockera, który zawiera środowisko *Playwright*. Po zakończeniu testów artefakty (raporty i zrzuty ekranu) zostają zarchiwizowane (przechowywane przez 1 dzień) w zakładce *Jobs* w GitLab.
- `test_mock` – także pobiera zależności, a następnie wywołuje `npm run test:mock`, które sprawdza poprawność plików `.mock.js` względem realnego API (m.in. przy pomocy `mockTest.js`). Jeśli w `mockApi` występują niezgodności, zadanie zakończy się niepowodzeniem.

Efekt działania pipeline: Po zaktualizowaniu kodu w repozytorium GitLab automatycznie uruchamia oba zadania w kolejności zdefiniowanej w pliku `.gitlab-ci.yml`. W przypadku wykrycia błędów (zarówno w testach E2E, jak i w `mockTest`), pipeline zostaje oznaczony jako *failed*, a deweloperzy mogą przejrzeć logi w GitLab, aby dowiedzieć się, które testy nie przeszły pomyślnie. W efekcie zapewniona jest ciągła walidacja jakości aplikacji: testy *end-to-end* weryfikują rzeczywiste scenariusze działania w przeglądarce, zaś *mockTest* gwarantuje zgodność plików `.mock.js` ze strukturą aktualnego API.

Rozdział 8

Podsumowanie

W niniejszej pracy przedstawiono koncepcję i implementację narzędzia wspomagającego tworzenie testów automatycznych dla aplikacji webowych. Podczas realizacji celu skupiono się na kilku aspektach: generowaniu testów z użyciem podejścia *record-and-play* (*Playwright codegen*), elastycznej konfiguracji *mocków* API oraz integracji z pipeline’ami CI/CD. Projekt ma formę modułowego rozwiązania, w którym podstawową rolę odgrywa `screenshotTestBuilder` (klasa budująca testy wizualne) oraz `testManager` (narzędzie CLI do zarządzania plikami `.spec.js` i interakcjami testowymi).

8.1 Wnioski końcowe

Opracowane narzędzie pozwala tworzyć i utrzymywać testy w bardziej zorganizowany sposób, nawet w małych i średnich zespołach. Dzięki funkcji automatycznego nagrywania czynności (*Playwright codegen*) programista nie musi ręcznie pisać skryptów testowych — zamiast tego może „nagrać” interakcje w przeglądarce i w prosty sposób wstawić je do `.spec.js` za pomocą `testManager`. Wprowadzenie `mockTest` z kolei ułatwia dbanie o aktualność *mocków* względem faktycznego API, co zmniejsza ryzyko, że testy staną się nieadekwatne do stanu produkcyjnego.

Największym wyzwaniem okazało się zapewnienie spójności między wieloma komponentami (m.in. `screenshotTestBuilder`, `mockApi`, `testManager`, `codegen`) oraz umożliwienie ich rozszerzania w przyszłości. Istotnym wnioskiem jest też fakt, że jakość i stabilność testów wciąż w dużej mierze zależą od świadomego projektowania scenariuszy i ich regularnej aktualizacji — narzędzie automatyzuje wiele zadań, lecz nie zastępuje w pełni testera czy programisty.

8.2 Możliwości dalszego rozwoju

- **Integracja z innymi frameworkami:** Choć obecnie wykorzystano *Playwright*, narzędzie można wzbogacić o mechanizmy wspierające inne biblioteki (np. *Cypress*), zapewniając szerszy wybór w zależności od preferencji zespołu.
- **Rozszerzona edycja interakcji:** `testManager` mógłby oferować interfejs wizualny (np. w trybie webowym) do modyfikacji sekwencji kroków, z podglądem wszystkich akcji i możliwością ich przedstawiania.

- **Dodatkowe formy testów wizualnych:** Obecnie generowane są zrzuty ekranu pełnej strony. Cennym usprawnieniem może być porównywanie wybranych elementów interfejsu (np. wycinek z modalem) lub tworzenie animowanych *gifów* pokazujących interakcje krok po kroku.
- **Analiza regresji wizualnej w chmurze:** Można zintegrować narzędzie z zewnętrznymi usługami do porównywania obrazów w chmurze (np. *Applitools*), aby odciążyć infrastrukturę lokalną i usprawnić raportowanie różnic.
- **Weryfikacja dostępności (Accessibility):** Istnieje potencjał do rozszerzenia testów o automatyczną analizę dostępności (np. integracja z *@axe-core/playwright*), co umożliwi wychwytywanie problemów już na etapie codziennego *builda*.
- **Zaawansowane generowanie testów:** Oprócz prostego *record-and-play* można wprowadzić tryb *Model-Based Testing*, w którym narzędzie generowałoby wiele ścieżek przejścia w aplikacji w sposób systematyczny i pokrywający większą liczbę stanów.

Proponowane kierunki rozwoju pozwoliłyby jeszcze bardziej rozszerzyć możliwości narzędzia i uczynić je atrakcyjniejszym zarówno dla zespołów wdrażających podstawową automatyzację testów, jak i tych, które stawiają na kompleksowe, wielopoziomowe podejście do zapewniania jakości.

Spis rysunków

5.1	Diagram klas przedstawiający główne komponenty systemu	17
5.2	Diagram przypadków użycia z perspektywy użytkownika i systemu CI/CD	19
5.3	Główne menu <code>testManager</code> po uruchomieniu: użytkownik wybiera plik <code>.spec.js</code> lub opcję <i>Exit</i>	25
5.4	Menu wyboru wariantu w <code>testManager</code> . Można też wrócić do wyboru pliku <code>.spec.js</code>	26
5.5	Przykładowa tabelaryczna prezentacja stanu wariantu w <code>testManager</code> . .	27
5.6	Przykład równoczesnego działania CLI <code>testManager</code> i <i>Playwright codegen</i> (przeglądarka).	28
7.1	Fragment logu uruchomienia testów E2E w trybie konsolowym. Widoczne są m.in. błędy nieudanego testu zrzutu ekranu (16 testów niepowodzeń i 4 zakończone sukcesem).	42
7.2	Raport <i>Playwright</i> w przeglądarce – przykładowy błąd spowodowany różnicą w rozmiarach zrzutu ekranu.	43
7.3	Przykładowy screenshot aplikacji <code>home</code> (desktop, jasny motyw, wariant <i>booking</i>).	44
7.4	Przykład udanego uruchomienia testów <code>mockTest.js</code> , w którym zweryfikowano poprawność 2 plików <code>.mock.js</code>	45
7.5	Przykład błędu w <code>mockTest.js</code> – struktura zwracanego <code>branding/</code> nie zgadza się z oczekiwaną (pole <code>latitude</code> nie jest typu <code>number/null</code>). . . .	46

Bibliografia

- [1] Roman, A., & Zmitrow, K. (2024). *Testowanie oprogramowania w praktyce: studium przypadków 2.0*.
- [2] Oshero, R. (2024). *Testy jednostkowe: świat niezawodnych aplikacji*.
- [3] Roman, A., & Zmitrow, K. (2024). *Testowanie oprogramowania w praktyce: studium przypadków*.
- [4] Roman, A. (2024). *Testowanie i jakość oprogramowania: modele, techniki, narzędzia*.
- [5] CircleCI. (n.d.). What is End-to-End Testing? Pozyskano z <https://circleci.com/blog/what-is-end-to-end-testing/>
- [6] Microsoft Playwright. (n.d.). Introduction to Playwright. Pozyskano z <https://playwright.dev/docs/intro>