



Akademia  
Techniczno-Humanistyczna  
w Bielsku-Białej

## PRACA DYPLOMOWA

WYDZIAŁ  
BUDOWY MASZYN I INFORMATYKI  
KIERUNEK: **Informatyka**  
SPECJALNOŚĆ: **Inżynieria oprogramowania**

**Piotr Florczak**

nr albumu: 55889

Praca inżynierska

### **TESTY AUTOMATYCZNE W APLIKACJACH INTERNETOWYCH**

Kategoria pracy: praca projektowa

Promotor: dr inż. Tomasz Gancarczyk  
Opiekun: prof. dr hab. inż. Mikołaj Karpiński

Bielsko-Biała, 2022/2023

Bielsko-Biała, dnia .....20..... r.

Florczak Piotr  
(nazwisko i imię)  
55889  
(nr albumu)  
Informatyka  
(kierunek studiów)  
Inżynieria oprogramowania  
(specjalność/specjalizacja)  
Niestacjonarne  
(forma studiów – stacjonarne/niestacjonarne)

## O Ś W I A D C Z E N I E

Oświadczam, że złożona praca końcowa

pt. „Testy automatyczne w aplikacjach internetowych”

1. została napisana przeze mnie samodzielnie
2. w swojej pracy korzystałem/łam z materiałów źródłowych w granicach dozwolonego użytku wymieniając autora, tytuł pozycji i źródło jej publikacji
3. zamieszczałem/łam krótkie fragmenty cudzych utworów w cudzysłowie, a w przypisie podałem/łam źródło tego cytatu. Dotyczy to cytatów zaczerpniętych z publikacji naukowych, takich jak książki, czasopisma, a także z wewnętrznych opracowań przedsiębiorstw, z instrukcji obsługi, prospektów reklamowych oraz z trwałych źródeł informacji w formie elektronicznej
4. praca nie ujawnia żadnych danych, informacji i materiałów, których publikacja nie jest prawnie dozwolona
5. praca nie była wcześniej podstawą żadnej innej procedury związanej z nadaniem stopni naukowych, dyplomów ani tytułów zawodowych
6. jestem świadoma/my, że przywłaszczenie sobie autorstwa albo wprowadzenie w błąd co do autorstwa całości lub części cudzego utworu jest przestępstwem – zagrożonym na podstawie ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych – odpowiedzialnością karną
7. nie zamieszczałem/łam w pracy fragmentów nietrwałych źródeł informacji. Przez nietrwałe źródła informacji rozumie się w szczególności informacje pozyskane za pomocą środków elektronicznych, które ze względu na swój modyfikowalny charakter, jak również brak ich przypisania do określonego wiarygodnego autora lub instytucji nie powinny stanowić rzetelnego źródła informacji stanowiącego podstawę dla realizacji pracy dyplomowej studenta.

Składając to oświadczenie, jestem świadomy/ma, że jeżeli moja praca narusza przepisy prawa, nie zostanie ona przez Uczelnię przyjęta. Ponadto już po obronie praca może być poddana kontroli następczej, która w przypadku naruszenia przepisów ustawy o ochronie praw autorskich i praw pokrewnych jak również przepisów szczególnych, prowadzić może do wszczęcia postępowania w przedmiocie cofnięcia jej autorowi uzyskanego tytułu zawodowego. Prawdziwość powyższego oświadczenia potwierdzam własnoręcznym podpisem.

.....  
(podpis studenta/studentki)

## Spis treści

1. Wstęp.....	6
2. Cel pracy.....	6
3. Zakres pracy.....	6
Część teoretyczna.....	8
4. Wstęp do testów w programowaniu.....	8
4.1. Testy manualne.....	8
4.1.1. Testy białej i czarnej skrzynki.....	9
4.1.2. Zalety i wady testowania manualnego.....	9
4.2. Testy automatyczne.....	9
4.2.1. Zalety i wady testowania automatycznego.....	10
4.2.2. Wzorzec AAA.....	10
5. Czym jest package.json.....	11
6. Ciągła integracja i ciągłe dostarczanie (CI/CD).....	12
7. Trzy główne obszary w Git.....	13
8. Konteneryzacja aplikacji.....	14
8.1. Architektura konteneryzacji.....	14
9. Znaczenie oraz sposoby testowania dostępności cyfrowej.....	16
9.1. Sposoby testowania dostępności cyfrowej.....	16
Część praktyczna.....	18
10. Przygotowanie projektu.....	18
10.1. Wymagania systemowe.....	18
10.2. Struktura projektu.....	18
10.3. Uruchomienie projektu.....	19
10.3.1. Przed pierwszym uruchomieniem.....	19
10.3.2. Dostępne komendy w projekcie.....	19
10.3.3. Uruchomienie projektu.....	20

11. Najprostsze rozwiązanie, czyli statyczna analiza kodu oraz jednolite formatowanie.....	22
11.1. Eslint.....	22
11.1.1. Konfiguracja.....	22
11.1.2. Prezentacja działania.....	25
11.2. Prettier.....	26
11.2.1. Konfiguracja.....	26
11.3. Lint-stage.....	28
11.3.1. Konfiguracja.....	28
11.3.2. Prezentacja działania.....	30
12. Testy jednostkowe i integracyjne.....	31
12.1. Konfiguracja.....	31
12.2. Testy jednostkowe.....	32
12.2.1. Prezentacja działania.....	34
12.3. Testy integracyjne.....	36
12.3.1. Prezentacja działania.....	38
13. Testy End to End.....	39
13.1. Konfiguracja.....	39
13.2. Prezentacja działania.....	40
14. Środowisko wdrożeniowe CI/CD.....	42
14.1. Własny hosting lub rozwiązanie chmurowe.....	43
14.2. Przygotowanie obrazu kontenera aplikacji.....	43
14.3. Konfiguracja serwera.....	44
14.4. Konfiguracja potoku CI/CD.....	45
14.4.1. Omówienie przygotowanego pliku „gitlab-ci.yml”.....	47
14.5. Prezentacja działania.....	52
15. Automatyczne testowania dostępności cyfrowej.....	54
16. Podsumowanie.....	56

Literatura.....	57
Spis kodu.....	57
Spis tabel.....	58
Spis rysunków.....	58

## 1. WSTĘP

Testowanie jest jednym z etapów inżynierii oprogramowania i zalicza się do procesów związanych z zapewnieniem jakości<sup>1</sup>. Testy mają na celu wykrycie błędów i anomalii w sprawdzanej aplikacji przed przekazaniem jej do ogólnego użytku.

Dana wersja programu zawiera w sobie skończoną ilość defektów. Dzięki znalezieniu potencjalnego błędu w systemie jesteśmy w stanie ocenić skalę problemu i odpowiednio zareagować.

Pozytywne przejście testów przez weryfikowane oprogramowanie buduje zaufanie do tworzonego programu i pozwala wyznaczyć punkty kontrolne, które mogą zostać wykorzystane do późniejszej weryfikacji zmian w programie. Przykładowo czy nowo dodana funkcjonalność nie wpłynęła negatywnie na działanie pozostałej części aplikacji.

Poziom wymaganej pewności w poprawnym działaniu oprogramowania jest zależny od docelowego zastosowania systemu, oczekiwań użytkownika, środowiska marketingowego, oraz możliwości wdrażania aktualizacji. Innymi słowy im bardziej krytyczne znaczenie ma tworzone oprogramowanie i trudniej jest je zaktualizować to tym większą uwagę powinno poświęcić się testom.

## 2. CEL PRACY

Celem pracy inżynierskiej było stworzenie projektu aplikacji internetowej, który zawierałby konfigurację narzędzi do statycznej analizy i formatowania kodu. Jak i również przygotowanie środowiska wdrożeniowego umożliwiającego tworzenie i uruchomienie testów automatycznych, oraz przeprowadzenia zautomatyzowanego procesu wydania finalnej wersji aplikacji.

## 3. ZAKRES PRACY

Praca w swoim zakresie obejmuje poniższe zagadnienia.

1. Przygotowanie i omówienie struktury projektu opartego o Next.js.
2. Konfigurację statycznej analizy kodu z użyciem Eslint.
3. Wymuszenie jednolitego formatowania kodu za pomocą narzędzia Prettier.

---

<sup>1</sup>Witryna internetowa: [https://pl.wikipedia.org/wiki/In%C5%Bcynieria\\_oprogramowania](https://pl.wikipedia.org/wiki/In%C5%Bcynieria_oprogramowania), 21.03.2023

4. Wykonanie testów jednostkowych i integracyjnych w React.js z użyciem Jest oraz React Testing Library.
5. Zastosowanie Cypress do stworzenia testów End to End.
6. Przygotowanie środowiska wdrożeniowego w GitLab.
7. Omówienie znaczenia i pokazanie sposobów testowania dostępności cyfrowej.

## CZĘŚĆ TEORETYCZNA

### 4. WSTĘP DO TESTÓW W PROGRAMOWANIU

Testowanie jest ważną częścią procesu wytwarzania oprogramowania. Polega na sprawdzeniu, czy program lub fragment kodu działa poprawnie, spełnia określone wymagania i specyfikacje.

Istnieje wiele rodzajów testów, między innymi są to:

1. testy jednostkowe – obejmują testowanie najmniejszych możliwych do sprawdzenia fragmentów kodu,<sup>2</sup>
2. testy integracyjne – rodzaj testów polegających na sprawdzeniu jak połączone fragmenty aplikacji współpracują ze sobą,
3. testy End to End – metoda testowania oprogramowania, która obejmuje sprawdzenie pracy całej aplikacji. Ten rodzaj testów ma na celu odtworzenie akcji jakie będą dostępne dla użytkownika końcowego, np. rejestracja, logowanie,<sup>3</sup>
4. testy akceptacyjne – rodzaj testów przeprowadzanych w celu ustalenia czy aplikacja jest dopuszczalna do wydania, np. alfa i beta testy,
5. testy wydajnościowe – obejmują testowanie aplikacji w celu sprawdzenia jak działa w różnych warunkach, np. duże obciążenie lub ograniczona ilość zasobów systemowych,<sup>4</sup>
6. testy penetracyjne – proces polegający na przeprowadzeniu kontrolowanego ataku na system. Mają one na celu określenie bieżącego stanu bezpieczeństwa sprawdzanego oprogramowania.<sup>5</sup>

#### 4.1. TESTY MANUALNE

Testy manualne przeprowadzane są przez testerów. Pozwalają one na analizę danej aplikacji według określonego scenariusza. Głównym celem tego typu testów jest znalezienie błędów, nieprawidłowości i niespójności z określoną specyfikacją programu. Jak i również sprawdzenia oprogramowania w sposób taki w jaki by z niego korzystał użytkownik końcowy.<sup>6</sup>

---

<sup>2</sup>Witryna internetowa: [https://en.wikipedia.org/wiki/Unit\\_testing](https://en.wikipedia.org/wiki/Unit_testing), 12.02.2023

<sup>3</sup>Witryna internetowa: <https://www.browserstack.com/guide/end-to-end-testing>, 12.02.2023

<sup>4</sup>Witryna internetowa: [https://pl.wikipedia.org/wiki/Testy\\_wydajno%C5%9Bciowe](https://pl.wikipedia.org/wiki/Testy_wydajno%C5%9Bciowe), 12.02.2023

<sup>5</sup>Witryna internetowa: [https://pl.wikipedia.org/wiki/Test\\_penetracyjny](https://pl.wikipedia.org/wiki/Test_penetracyjny), 12.02.2023

<sup>6</sup>Witryna internetowa: <https://www.ideo.pl/firma/o-nas/nasze-publicacje/testowanie-manualne-i-automatyczne,167.html>, 12.02.2023



#### **4.1.1. TESTY BIAŁEJ I CZARNEJ SKRZYNKI**

Istnieje wiele podejść do testowania manualnego między innymi są to:

1. testy białej skrzynki – tester ma wgląd do elementów budujących aplikację. Może mieć wiedzę odnośnie struktury przesyłanych danych, znać algorytmy odpowiedzialne za działanie danej funkcjonalności, posiadać wgląd do kodu źródłowego programu,<sup>7</sup>
2. testy czarnej skrzynki – tester weryfikuje aplikację nie mając wglądu w wewnętrzny sposób funkcjonowania systemu. Sprawdza program mając dostęp tylko do danych wejściowych i wyjściowych. Tester nie ma wglądu w kod źródłowy aplikacji.<sup>8</sup>

#### **4.1.2. ZALETY I WADY TESTOWANIA MANUALNEGO**

Zalety testów manualnych to:

1. niski próg wejścia,
2. proste we wdrożeniu,
3. nie wymagają wiedzy programistycznej,
4. odzwierciedlają interakcje użytkownika końcowego z aplikacją,
5. dobrze nadają się do testowania graficznych interfejsów.

Wady testowania manualnego to:

1. czasochłonność,
2. zależność od wiedzy i doświadczeń testera,
3. podatność na błąd ludzki przy powtarzalnych testach,
4. monotonność testów może być nużąca.<sup>9</sup>

#### **4.2. TESTY AUTOMATYCZNE**

Testy automatyczne polegają na weryfikacji poprawności działania oprogramowania z użyciem narzędzi do tego przeznaczonych.<sup>10</sup> Można podzielić testy automatyczne na dwa typy:

---

<sup>7</sup>Smilgin, R. „Czarna skrzynka i biała skrzynka” w (2016) *Zawód tester*, Wyd. 1, Naukowe PWN, Warszawa

<sup>8</sup>Witryna internetowa: <https://www.testmonitor.com/blog/7-manual-testing-types-explained>, 03.04.2023

<sup>9</sup>Witryna internetowa: <https://www.simplilearn.com/manual-testing-article>, 30.03.2023

<sup>10</sup>Witryna internetowa: <https://www.techtarget.com/searchsoftwarequality/definition/automated-software-testing>, 03.04.2023

1. analizę statyczną – czyli sprawdzenie kodu źródłowego aplikacji pod kątem zdefiniowanych i ustandaryzowanych reguł poprawności,
2. testowanie automatyczne – uruchomienie i wykonanie przez środowisko testowe wcześniej przygotowanych testów przez programistę w wybranym języku programowania.<sup>11</sup>

#### **4.2.1. ZALETY I WADY TESTOWANIA AUTOMATYCZNEGO**

Zalety testów automatycznych to:

1. zapewniają powtarzalność,
2. ograniczają czas oczekiwania na otrzymanie wyniku po testach,
3. umożliwiają automatyzację monotonnych testów manualnych.

Wady testów automatycznych to:

1. wymagają przygotowania kodu testującego,
2. wymagają środowiska uruchamiającego testy,
3. wymagają utrzymania,
4. konieczna jest wiedza programistyczna.

#### **4.2.2. WZORZEC AAA**

Wzorzec *Arrange, Act, Assert* (zorganizować, działać, sprawdzać) określa strukturę testów jednostkowych z podziałem na trzy części:

1. *arrange* – przygotowanie danych wejściowych potrzebnych w części *act*,
2. *act* – wykonanie operacji na testowanym fragmencie funkcjonalności,
3. *assert* – zweryfikowanie czy otrzymane wartości po wykonaniu *act* zgadzają się z oczekiwaniami.<sup>12</sup>

---

<sup>11</sup>Smilgin, R. „Testy automatyczne” w (2016) *Zawód tester*, Wyd. 1, Naukowe PWN, Warszawa

<sup>12</sup>Witryna internetowe: <https://dariuszwozniak.net/posts/kurs-tdd-3-struktura-test-czyli-arrange-act-assert>, 09.03.2023

## 5. CZYM JEST PACKAGE.JSON

Package.json jest to pliku o rozszerzeniu JSON (JavaScript Object Notation), który zawiera metadane istotne dla projektu. Służy do zarządzania zależnościami i ich wersjami. Dane zawarte w tym pliku są przedstawione w formie czytelnej dla człowieka. Wymagane właściwości w „package.json” to *name* (nazwa) i *version* (wersja). Nazwa musi być pisana małymi literami, jednym słowem oraz może zawierać łączniki i podkreślenia. Wersja musi mieć postać x.x.x, np. 1.0.0.<sup>13</sup>

Dodatkowo istotnymi właściwościami, na które powinno się zwrócić uwagę to:

1. *license* (licencja) – rodzaj licencji, np. MIT czy UNLICENSED,
2. *scripts* (skrypty) – zawierają komendy wywołujące skrypty, które używane są w różnych cyklach życia projektu, np. build, test, publish,
3. *dependencies* (zależności) – prosty obiekt, który informuje jakie pakiety są zawarte w opublikowanej wersji projektu. Klucz obiektu zawiera nazwę pakietu, a wartości akceptowaną wersję,
4. *peerDependencies* (zależności równorzędne) – obiekt, którego struktura jest identyczna jak dla zależności. Ta właściwość mówi o tym jakie paczki muszą zostać zainstalowane aby projekt mógł działać poprawnie.

---

<sup>13</sup>Witryna internetowa: <https://docs.npmjs.com/cli/v9/configuring-npm/package-json>, 12.02.2023

## 6. CIĄGŁA INTEGRACJA I CIĄGŁE DOSTARCZANIE (CI/CD)

Ciągła integracja i ciągłe dostarczanie jest to zbiór zasad i wytycznych dotyczących pracy nad projektami informatycznymi. Dzięki nim jest możliwość częstego dostarczania sprawdzonej funkcjonalności, implementacja tych praktyk nazywana jest CI/CD pipeline (potok CI/CD).<sup>14</sup>



**Rys. 1: Schemat obrazujący proces ciągłej integracji, ciągłego dostarczania i ciągłego publikowania [źródło: <https://www.redhat.com/en/topics/devops/what-cicd-pipeline>, 2023]**

1. *Continuous integration* (ciągła integracja) – automatyczny proces, którego celem jest sprawdzenie zmian przed włączeniem ich do repozytorium. Można w ciągłej integracji wyróżnić trzy etapy takie jak budowa, testowanie oraz zatwierdzenie zmian.<sup>15</sup>
2. *Continuous Delivery* (ciągłe dostarczanie) – kontynuacja po procesie ciągłej integracji. Przygotowanie pod wdrożenie funkcjonalności i automatyczne opublikowanie zmian w wybranych środowiskach, np. deweloperskim, testowym.<sup>16</sup>
3. *Continuous deployment* (ciągłe publikowanie) – ciąg dalszy procesu ciągłego dostarczania. Automatyczne opublikowanie zmian w środowisku produkcyjnym.<sup>17</sup>

<sup>14</sup>Witryna internetowa: <https://fullstackadmin.pl/wyjasniamy-co-to-jest-ci-cd>, 03.03.2023

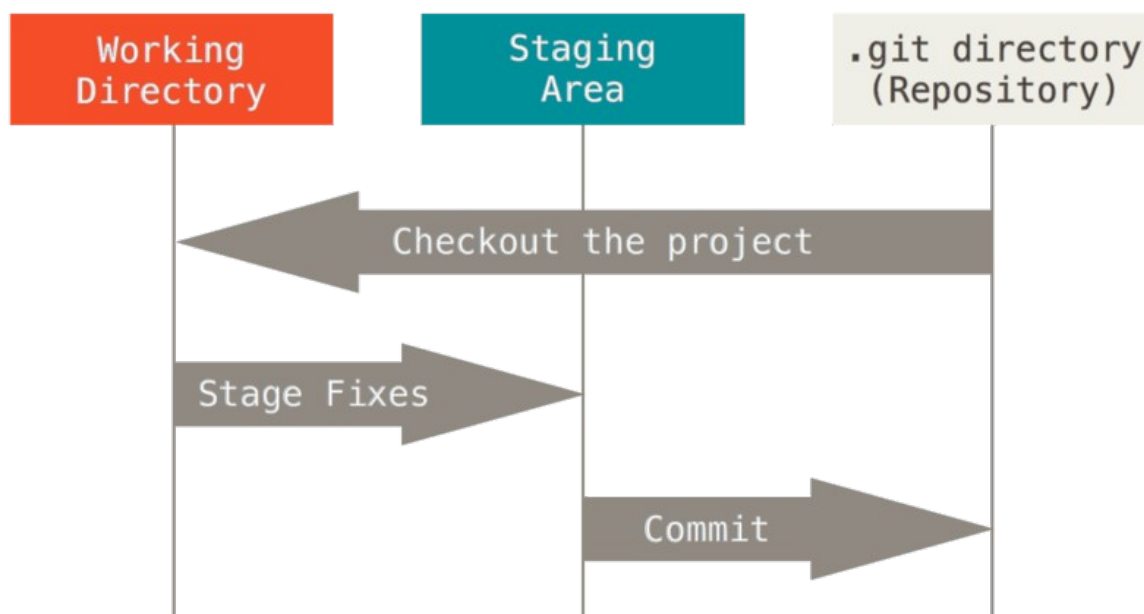
<sup>15</sup>Witryna internetowa: [https://en.wikipedia.org/wiki/Continuous\\_integration](https://en.wikipedia.org/wiki/Continuous_integration), 03.03.2023

<sup>16</sup>Witryna internetowa: <https://www.ibm.com/topics/continuous-integration>, 03.03.2023

<sup>17</sup>Witryna internetowa: <https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment>, 03.03.2023

## 7. TRZY GŁÓWNE OBSZARY W GIT

Git jest to darmowy i otwarcie źródłowy system do kontroli wersji. Pozwala on stworzyć repozytorium plików i zarządzać historią zmian w katalogu, w którym narzędzie to zostało zainicjalizowane.<sup>18</sup>



**Rys. 2: Trzy wyróżnione obszary w narzędziu Git [źródło:  
<https://git-scm.com/about/staging-area>, 2023]**

Na rysunku 2 pokazano trzy główne obszary z narzędzia Git, są to:

1. *working directory* (katalog roboczy) – jest to katalog udostępniający do modyfikacji aktualną wersję plików projektu,
2. *staging area* (obszar pomostowy) – jest to plik inaczej zwany indeksem, zawiera informacje o wykonanych modyfikacjach w katalogu roboczym. Po zatwierdzeniu zmian zostają one zapisane w repozytorium,
3. *repository* (repozytorium) – jest to katalog o nazwie „.git”, który zawiera informację o wszystkich zatwierdzonych zmianach z katalogu roboczego.<sup>19</sup>

<sup>18</sup>Witryna internetowa: <https://git-scm.com/>, 07.04.2023

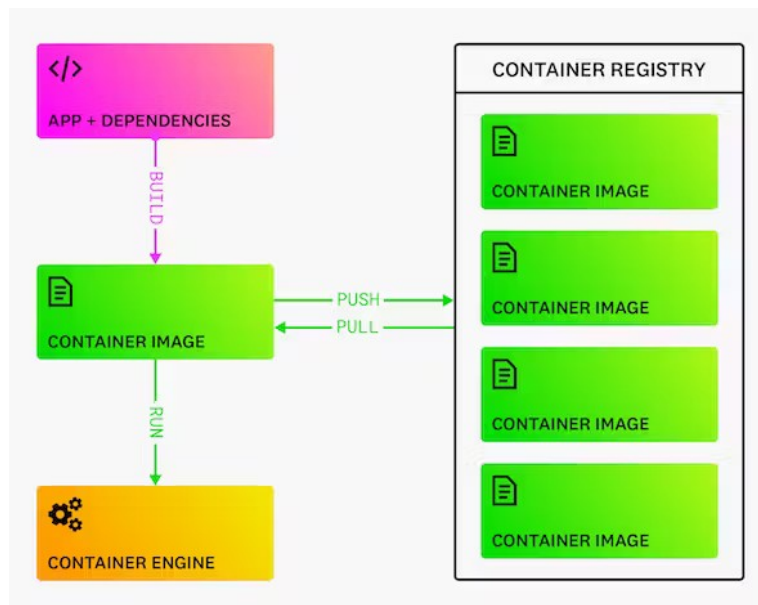
<sup>19</sup>Witryna internetowa: <https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F>, 07.04.2023

## 8. KONTENERYZACJA APLIKACJI

Konteneryzacja aplikacji jest to spakowanie kodu oprogramowania z wszystkimi potrzebnymi zależnościami i wymaganymi bibliotekami systemowymi do jednostki zwanej kontenerem. Ten proces umożliwia odizolowanie aplikacji od pozostałej części systemu co sprawia, że program staje się przenośny, możliwy do uruchomienia na różnych platformach.

Konteneryzację można porównać do wirtualizacji, celem obu technologii jest izolacja aplikacji. Wirtualizacja oddziela program z użyciem wirtualnych maszyn, które wymagają pełnej kopii systemu operacyjnego i zasobów sprzętowych. Podczas gdy konteneryzacja izoluje aplikacje korzystając z istniejącego już na maszynie systemu operacyjnego, udostępniając go i jego zasoby między działającymi kontenerami.

### 8.1. ARCHITEKTURA KONTENERYZACJI



**Rys. 3: Schemat pokazujący budowę, zapisanie i uruchomienie skonteneryzowanej aplikacji [źródło: <https://www.datadoghq.com/knowledge-center/containerize-d-applications/>, 2023]**

Na rysunku 3 zaprezentowano architekturę konteneryzacji składającą się z etapu „*build*” (budowy), czyli spakowania aplikacji z wymaganymi zależnościami do „*container image*” (obrazu kontenera). Następnie „*push*” (wysłania) zbudowanego obrazu do „*container registry*” (repozytorium kontenerów). Po czym w kolejnym etapie wykonania „*pull*” (pobrania) i „*run*” (uruchomienia) obrazu z użyciem „*container engine*” (silnik kontenerów).

1. Obraz kontenera – to plik powstały po skonteneryzowaniu aplikacji, zawierający kod źródłowy programu i wymagane zależności, używany jest do uruchomienia kontenera.
2. Repozytorium kontenerów – to miejsce, w którym przechowywane są wersjonowane obrazy kontenerów.
3. Silnik kontenerów – to narzędzie, które jest pośrednikiem pomiędzy systemem operacyjnym, a kontenerami. Udostępnia polecenia do zarządzania, budowania i uruchamiania kontenerów. Dostępne silniki kontenerów to, np. Docker, Containerd, Podman.<sup>20</sup>

---

<sup>20</sup>Witryna internetowa: <https://www.datadoghq.com/knowledge-center/containerized-applications/>, 21.04.2023

## 9. ZNACZENIE ORAZ SPOSOBY TESTOWANIA DOSTĘPNOŚCI CYFROWEJ

Dostępność cyfrowa to termin odnoszący się do stopnia, w jakim osoba mająca dostęp do technologii cyfrowej, np. strony internetowej, aplikacji mobilnej może z niej korzystać. Osoby mające utrudniony dostęp do technologii cyfrowych lub nie mogące z niej skorzystać z powodu niepełnosprawności lub innych czynników, mogą doświadczać wykluczenia cyfrowego.<sup>21</sup>

Dokument „*Web Content Accessibility Guidelines*” (Wytyczne dla dostępności treści internetowych) określa wspólny standard dotyczący dostępności treści internetowych dla osób indywidualnych, organizacji i rządów. WCAG wyróżnia trzy poziomy dostępności:

1. A – poziom podstawowy, jeżeli nie zostanie spełniony, technologie asystujące mogą nie być w stanie odczytać, zrozumieć lub w pełni obsługiwać strony lub widoki,
2. AA – poziom średni, wymagany na wielu stronach rządów i instytucji publicznych. Listę polityk rządowych dotyczących dostępności stron internetowych można znaleźć pod adresem „<https://www.w3.org/WAI/policies/>”.
3. AAA – poziom zaawansowany, zazwyczaj stosuje się ten poziom w przypadku części aplikacji internetowych, które obsługują specjalistyczną grupę odbiorców.<sup>22</sup>

### 9.1. SPOSOBY TESTOWANIA DOSTĘPNOŚCI CYFROWEJ

Odnosząc się do artykułu „*Efficiency in Accessibility Testing or, Why Usability Testing Should be Last*” (Skuteczność testowania dostępności lub dlaczego testy użyteczności powinny być ostatnie) opublikowanego przez Karl Groves<sup>23</sup>, można wyróżnić cztery rodzaje testowania dostępności cyfrowej:

1. automatyczne testy dostępności – polegają na użyciu, np. aplikacji komputerowych, usług internetowych, wtyczek do przeglądarek, które automatyzują proces testowania dostępności. Te narzędzia mogą zweryfikować około 60% najlepszych praktyk dotyczących dostępności, z której około 35% wymaga weryfikacji przez człowieka,
2. manualne testy dostępności – polegają na ręcznej weryfikacji aplikacji lub strony internetowej, mogą obejmować inspekcję kodu źródłowego, zmian ustawień

---

<sup>21</sup>Witryna internetowa: <https://www.artefakt.pl/blog/epr/dostepnosc-cyfrowa-accessibility>, 22.04.2023

<sup>22</sup>Witryna internetowa: <https://www.a11yproject.com/checklist/>, 22.04.2023

<sup>23</sup>Witryna internetowa: <https://karlgroves.com/efficiency-in-accessibility-testing-or-why-usability-testing-should-be-last/>, 23.04.2023



oprogramowania lub użycia technologii wspomagających osoby niepełnosprawne w celu sprawdzenia najlepszych praktyk dotyczących dostępności. Rzeczywista wartość tego rodzaju testów zależy w dużej mierze od umiejętności osób wykonujących testy,

3. testy przypadków użycia – polegają na sprawdzeniu jak system reaguje na interakcje z użytkownikiem. Dobrą praktyką jest wykonanie przypadku testowego z użyciem technologii wspomagających osoby niepełnosprawne, aby określić jak system zadziała, gdy jest używany przez osobę korzystającą właśnie z takiej technologii. Podobnie jak przy testowaniu manualnym jakość wyników jest uzależniona od wiedzy i umiejętności testera,
4. testowanie użyteczności – to proces oceny w jaki sposób użytkownicy końcowy jest w stanie osiągnąć określone cele podczas interakcji z produktem lub usługą. W przypadku testów użyteczności pod kątem dostępności użytkownikiem końcowym jest osoba z niepełnosprawnością.

## CZĘŚĆ PRAKTYCZNA

### 10. PRZYGOTOWANIE PROJEKTU

Projekt aplikacji internetowej został napisany w TypeScript, który jest językiem programowania zawierającym typowanie, klasy i interfejsy. Umożliwia sprawdzanie kodu programu pod względem potencjalnych błędów jeszcze przed jego uruchomieniem. Ostatecznie TypeScript jest kompilowany do JavaScript czyli języka skryptowego wykorzystywanego między innymi przez przeglądarki internetowe.<sup>24</sup>

Stworzona aplikacja wykorzystwała JavaScript-ową bibliotekę React.js, która umożliwia tworzenie stron internetowych z użyciem komponentów, czyli fragmentów kodu, które można wielokrotnie wykorzystywać w projekcie.<sup>25</sup>

Całość aplikacji została zbudowana przy użyciu platformy programistycznej Next.js, która ma dobrze udokumentowaną strukturę projektu, oraz zapewnia wsparcie dla uruchomienia kodu po stronie serwera i klienta.<sup>26</sup>

#### 10.1. WYMAGANIA SYSTEMOWE

1. Node.js 14.6.0 lub nowszy.
2. System operacyjny Linux, Windows lub MacOS.<sup>27</sup>

#### 10.2. STRUKTURA PROJEKTU

Projekt został zainicjalizowany komendą „npx create-next-app@13.2.1”<sup>28</sup>, która wygenerowała następujące pliki:

1. node\_modules – katalog zawierający zewnętrzne paczki,
2. public – katalog z ogólnodostępnymi zasobami, np. zdjęcia, czcionki,
3. src/app – katalog zawierający szablony i strony aplikacji<sup>29</sup>,

---

<sup>24</sup>Witryna internetowa: <https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html>, 12.02.2023

<sup>25</sup>Witryna internetowa: <https://www.taniarascia.com/getting-started-with-react/>, 12.02.2023

<sup>26</sup>Witryna internetowa: <https://nextjs.org/learn/basics/create-nextjs-app>, 12.02.2023

<sup>27</sup>Witryna internetowa: <https://nextjs.org/docs/getting-started#system-requirements>, 04.03.2023

<sup>28</sup>Witryna internetowa: <https://nextjs.org/docs#automatic-setup>, 06.04.2023

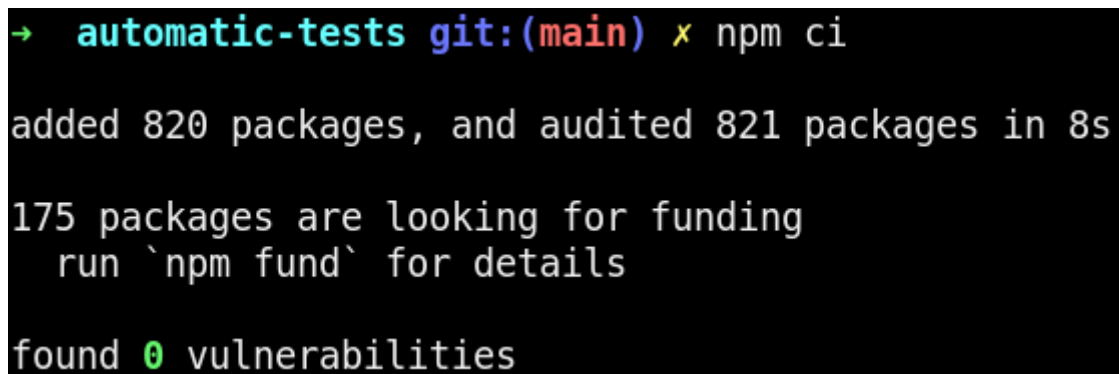
<sup>29</sup>Witryna internetowa: <https://nextjs.org/docs/advanced-features/custom-app>, 25.02.2023

4. .eslintrc.json – plik konfiguracyjny dla statycznej analizy kodu,
5. .gitignore – plik określający jakie pliki są ignorowane przez system kontroli wersji Git,
6. .next.config.js – plik konfiguracyjny Next.js,
7. package-lock.json – plik zawierający spis zainstalowanych paczek. Podczas instalacji pozwala pobrać konkretnie te wersje zależności, które zostały użyte w projekcie,
8. package.json – plik zawierający metadane projektu,
9. README.md – plik z informacjami od twórców,
10. tsconfig.json – plik konfiguracyjny TypeScript.

### 10.3. URUCHOMIENIE PROJEKTU

#### 10.3.1. PRZED PIERWSZYM URUCHOMIENIEM

Przed pierwszym uruchomieniem projektu należało w konsoli przejść do katalogu zawierającego aplikację komendą „cd automatic-tests/”. Następnie wykonać „npm ci” w celu zainstalowania wymaganych przez projekt pakietów.

A screenshot of a terminal window with a black background and white text. The first line shows a prompt '➔' followed by 'automatic-tests git:(main) x npm ci'. The subsequent lines show the output of the command: 'added 820 packages, and audited 821 packages in 8s', '175 packages are looking for funding', 'run `npm fund` for details', and 'found 0 vulnerabilities'.

```
➔ automatic-tests git:(main) x npm ci
added 820 packages, and audited 821 packages in 8s
175 packages are looking for funding
  run `npm fund` for details
found 0 vulnerabilities
```

Rys. 4: Wynik działania komendy „npm ci”

#### 10.3.2. DOSTĘPNE KOMENDY W PROJEKCIE

W projekcie dostępne komendy znajdują się w pliku „package.json” w właściwości „scripts”, można je użyć z poziomu katalogu aplikacji dopiero po wykonaniu punktu 10.3.1. Są to między innymi komendy takie jak:

1. npm run dev – uruchomienie aplikacji w trybie deweloperskim,

2. `npm run build` – zbudowanie produkcyjnej wersji aplikacji,
3. `npm run start` – uruchomienie serwera produkcyjnego aplikacji,
4. `npm run lint` – uruchomienie statycznej analizy kodu,
5. `npm run format:all` – uruchomienie formatowania kodu dla całego projektu,
6. `npm run test` – uruchomienie testów jednostkowych i integracyjnych,
7. `npm run test:e2e` – uruchomienie testów End to End,
8. `npm run cypress` – uruchomienie narzędzia Cypress,
9. `npm run check:quick` – uruchomienie statycznej analizy i formatowania kodu tylko na plikach znajdujących się w fazie pomostowej systemu kontroli wersji Git.

### 10.3.3. URUCHOMIENIE PROJEKTU

Projekt może zostać uruchomiony w dwóch trybach:

1. tryb deweloperski – przeznaczony do pracy programistycznej nad aplikacją. Zapewnia automatyczne odświeżanie strony w przeglądarce internetowej po dokonaniu zmian w kodzie projektu. W razie wystąpienia błędu w kodzie aplikacji, w konsoli zostanie wyświetlona informacja o problemie wraz ze ścieżką do kłopotliwego fragmentu. Tryb deweloperski uruchamia się komendą „`npm run dev`”.<sup>30</sup>
2. tryb produkcyjny – przeznaczony do uruchomienia serwera ze zbudowaną wersją aplikacji. W pierwszej kolejności aplikacja musi zostać zbudowana komendą „`npm run build`”, następnie tryb produkcyjny uruchamia się poleceniem „`npm run start`”.<sup>31</sup>

---

<sup>30</sup>Witryna internetowa: <https://nextjs.org/docs/api-reference/cli#development>, 06.04.2023

<sup>31</sup>Witryna internetowa: <https://nextjs.org/docs/api-reference/cli#production>, 06.04.2023

```
→ automatic-tests git:(main) ✖ npm run dev

> automatic-tests@0.1.0 dev
> next dev

ready - started server on 0.0.0.0:3000, url: http://localhost:3000
warn - You have enabled experimental feature (appDir) in next.config.js.
warn - Experimental features are not covered by semver, and may cause unexpected or
broken application behavior. Use at your own risk.

info - Thank you for testing `appDir` please leave your feedback at https://nextjs.l
ink/app-feedback
event - compiled client and server successfully in 5s (264 modules)
wait - compiling...
event - compiled client and server successfully in 1166 ms (264 modules)
```

Rys. 5: Wynik działania komendy „npm run dev” z konsoli

Po uruchomieniu komendy „npm run dev” został włączony serwer deweloperski z dostępną aplikacją pod adresem „http://localhost:3000”.



Rys. 6: Wyświetlona strona internetowa po wejściu z przeglądarki internetowej pod otrzymany adres z rysunku 5

Na rysunku 6 pokazano stronę z adresu „http://localhost:3000”, która została uruchomiona w rysunku 5.

## 11. NAJPROSTSZE ROZWIĄZANIE, CZYLI STATYCZNA ANALIZA KODU ORAZ JEDNOLITE FORMATOWANIE

Tworzenie i utrzymanie testów zajmuje czas, co wiąże się z dodatkową pracą i kosztami. Na początku użyto rozwiązania, które małym nakładem pracy znacząco poprawiły jakość tworzonej funkcjonalności.

### 11.1. ESLINT

Eslint jest to konfigurowalne narzędzie do statycznej analizy kodu JavaScript. Sprawdza fragmenty kod pod względem określonych reguł. Plik konfiguracyjny znajduje się w głównej gałęzi projektu pod nazwą „.eslintrc.json”.<sup>32</sup>

Domyślna konfiguracja Eslint-a została rozszerzona o możliwość działania z TypeScript. Wymuszono sortowanie dołączanych do pliku zależności, oraz każdorazowe użycie funkcji „console.log” jest obarczone ostrzeżeniem.

#### 11.1.1. KONFIGURACJA

W wygenerowanym projekcie w punkcie 10.2 dołączona została domyślna konfiguracja narzędzie Eslint w pliku „.eslintrc.json” w głównej gałęzi projektu. Instalacja Eslint w innym projekcie jest możliwa przez uruchomienie komendy „npm install eslint --save-dev”.

```
{  
  "extends": "next/core-web-vitals"  
}
```

**Kod 1: Domyślna konfiguracja z pliku „.eslintrc.json”**

Żeby Eslint mógł działać poprawnie z TypeScript-em do projektu musiano zainstalować dodatkowe paczki:

---

<sup>32</sup>Witryna internetowa: <https://eslint.org/docs/latest/use/core-concepts>, 25.02.2023

1. `@typescript-eslint/parser` – tłumaczy kod źródłowy napisany w TypeScript na *Abstract Syntax Tree* (abstrakcyjne drzewo składni), kod zrozumiały przez Eslint,<sup>33</sup>
2. `@typescript-eslint/eslint-plugin` – zawiera reguły poprawności dla kodu zapisanego w TypeScript.<sup>34</sup>

Po zainstalowaniu paczek za pomocą komendy „`npm install @typescript-eslint/parser @typescript-eslint/eslint-plugin --save-dev`” należało zaktualizować plik konfiguracyjny Eslint o parser, plugin i rozszerzenie.

```
{
  "extends": [
    "next/core-web-vitals",
    "plugin:@typescript-eslint/recommended",
    "plugin:@typescript-eslint/recommended"
  ],
  "parser": "@typescript-eslint/parser",
  "plugins": ["@typescript-eslint"]
}
```

**Kod 2: Fragment pliku „.eslintrc.json” z dodanym nowym parserem i pluginem do obsługi TypeScript**

Dodano w konfiguracji Eslint regułę odpowiedzialną za wywołanie ostrzeżenia w momencie użycia funkcji „`console.log`”.

```
{
  [...]
  "rules": {
    "no-console": ["error", { "allow": ["warn", "error"] }]
  }
}
```

**Kod 3: Fragment pliku „.eslintrc.json” z dodaną regułą ostrzegającą o użyciu „`console.log`”**

<sup>33</sup>Witryna internetowa: <https://www.npmjs.com/package/@typescript-eslint/parser>, 12.02.2023

<sup>34</sup>Witryna internetowa: <https://www.npmjs.com/package/@typescript-eslint/eslint-plugin>, 12.02.2023

Umożliwiono automatyczne sortowanie importów za pomocą wtyczek „eslint-plugin-simple-import-sort” oraz „eslint-plugin-unused-imports”. Zainstalowano je komendą „npm install eslint-plugin-simple-import-sort eslint-plugin-unused-imports --save-dev”, oraz zaktualizowano konfigurację Eslint.<sup>35</sup>

```
{
  [...]
  "plugins": ["@typescript-eslint", "simple-import-sort", "unused-
imports"],
  "rules": {
    [...]
    "unused-imports/no-unused-imports": "error",
    "simple-import-sort/imports": "error",
  }
}
```

**Kod 4: Fragment pliku „.eslintrc.json” z dodanymi regułami wymuszającymi sortowanie importów**

Eslint udostępnia możliwość wykluczenia plików i całych katalogów ze statycznej analizy. Plik „.eslintignore” zawiera nazwy ignorowanych plików i katalogów.

Stworzono plik „.eslintignore” w głównej gałęzi projektu i wykluczono katalogi „.next” oraz „node\_modules”.

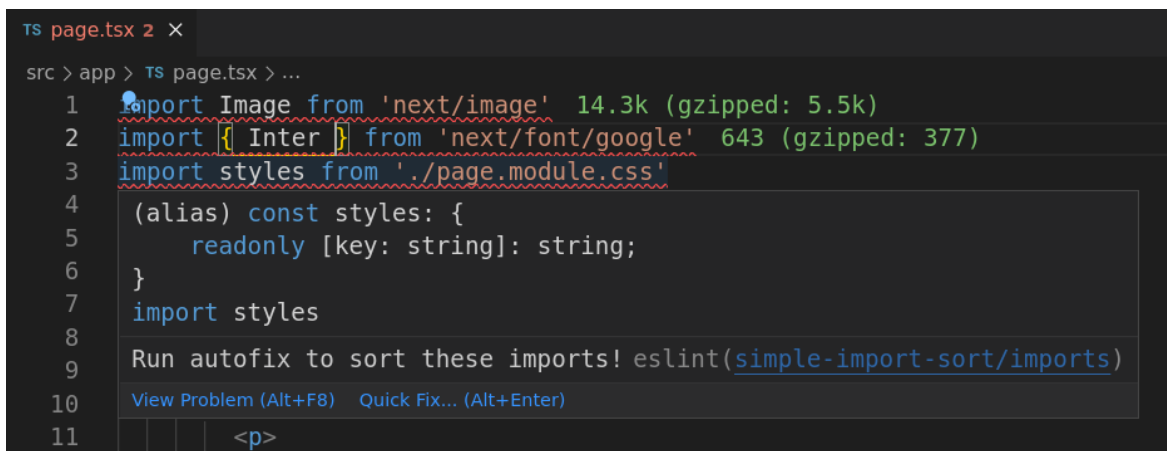
```
node_modules
.next
```

**Kod 5: Zawartość stworzonego pliku „.eslintignore”**

<sup>35</sup>Witryna internetowa: <https://github.com/lydell/eslint-plugin-simple-import-sort/#usage>, 25.02.2023

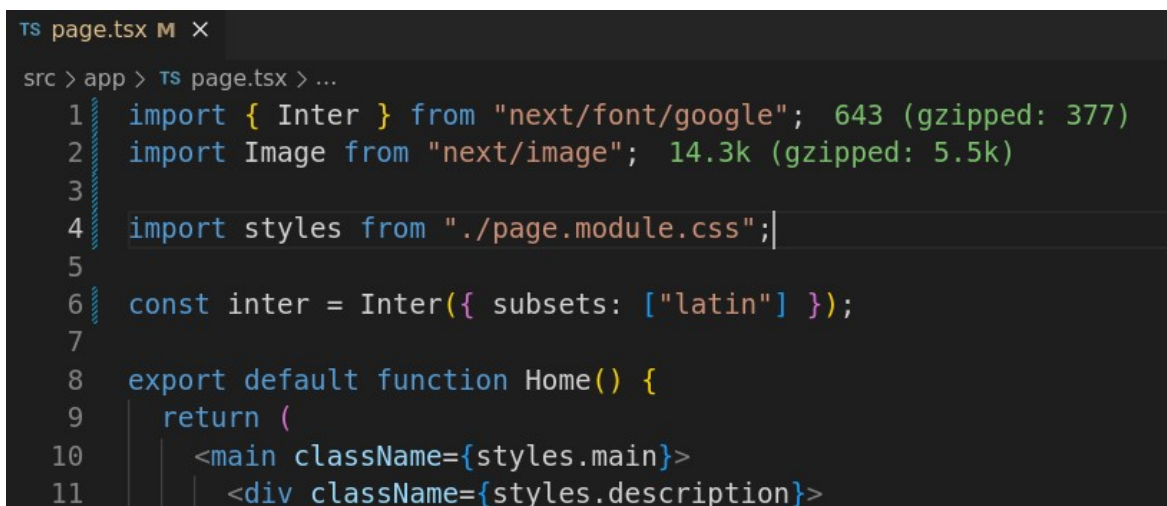


### 11.1.2. PREZENTACJA DZIAŁANIA



Rys. 7: Informacja o złamaniu reguła sortowania importów

Edytor tekstu poinformował użytkownika o potencjalnych błędach w kodzie. Na rysunku 7 pokazano błąd związany z sortowaniem importów. Problem został rozwiązany przy użyciu automatycznej opcji „Quick Fix” (szybka naprawa).



Rys. 8: Wynik użycia opcji „Quick Fix”

## 11.2. PRETTIER

Czytanie kodu jest czymś czego w programowaniu nie da się pominąć. Pisząc kod nie powinno się myśleć o konieczności jego formatowania. Późniejszą analizę kodu źródłowego ułatwia zachowanie jednolitego stylu, identycznych wcięć czy tych sam zakończenia linii.

Prettier to narzędzie do formatowania kodu, które zapewnia wsparcie przede wszystkim plików JavaScript, JSX, HTML, CSS, JSON, Markdown i innych. Usuwa oryginalne formatowanie i zapewnia, że kod wynikowy będzie spójny z ustalonymi stylami.<sup>36</sup>

### 11.2.1. KONFIGURACJA

Dodano Prettiera do projektu wykonując komendę „npm install eslint-config-prettier prettier --save-dev”. Następnie w „.eslintrc.json” dodano rozszerzenie „prettier”.

```
{  
  "extends": [  
    [...]  
    "prettier"  
  ],  
  ...  
}
```

**Kod 6: Fragment pliku „.eslintrc.json” z dodanym rozszerzeniem dla formatowania**

Później stworzono plik „.prettierrc” w głównej gałęzi projektu w celu ustawienia formatowanej szerokości tabów na 2 spacje.

---

<sup>36</sup>Witryna internetowa: <https://prettier.io/docs/en/>, 12.02.2023

```
{  
  "tabWidth": 2  
}
```

**Kod 7: Zawartość pliku „.prettierrc”**

Prettier pozwala wykluczyć pliki i całe katalogi z automatycznego formatowania za pomocą pliku „.prettierignore”, który może zawierać nazwy ignorowanych plików i katalogów.

Dodano plik „.prettierignore” w głównej gałęzi projektu, wykluczono w nim „package-lock” oraz katalogi „node\_modules” i „.next”.

```
node_modules  
.next  
package-lock
```

**Kod 8: Zawartość pliku „.prettierignore”**

Dodano do pliku „package.json” we właściwości scripts komendę „format:all” służącą do formatowania plików w całym projekcie.

```
{  
  [...]   
  "scripts": {  
    [...]   
    "format:all": "prettier --write ."  
  },  
  [...]   
}
```

**Kod 9: Fragment pliku „package.json” z dodaną komendą „format:all”**

### 11.3. LINT-STAGE

Domyślnie komendy „npm run lint” i „npm run format:all” działają w całym projekcie. Im więcej było plików to tym dłuższy był potrzebny czas na ich wykonanie.

```
→ projekt-1 git:(master) x time (npm run lint > /dev/null 2>&1)
( npm run lint > /dev/null 2>&1; ) 14,78s user 0,63s system 138% cpu 11,108 total
→ projekt-1 git:(master) x █
```

Rys. 9: Czas działania komendy „npm run lint” w katalogu projekt-1

```
→ projekt-1 git:(master) x time (npm run format:all > /dev/null 2>&1)
( npm run format:all > /dev/null 2>&1; ) 50,45s user 1,57s system 145% cpu 35,779 total
→ projekt-1 git:(master) x █
```

Rys. 10: Czas działania komendy „npm run format:all” w katalogu projekt-1

W katalogu „projekt-1” pierwsze uruchomienie Eslint-a komendą z rysunku 9 odbyło się na 992 plikach i zajęło to 14,8 sekund. Prettier w tym samym projekcie został uruchomiony komendą z rysunku 10 i sformatował 1475 plików, polecenie działało 50,5 sekund. Całościowy czas pracy tych dwóch komend wyniósł ponad minutę. Jest to niedopuszczalnie długi czas jak na wstępne testowanie kodu.

Czas działania Eslint i Prettier został ograniczony przy użyciu paczki Lint-stage. Lint-stage umożliwił uruchomienie komendy tylko na plikach, które znalazły się w obszarze pomostowym opisanym w punkcie 7.

#### 11.3.1. KONFIGURACJA

Zainstalowano Lint-stage w projekcie przy użyciu komendy „npm install lint-staged --save-dev”. Później stworzono plik konfiguracyjny dodanej paczki w głównej gałęzi projektu o nazwie „.lintstagedrc.js”.

```
const path = require("path");

[...]

const buildEslintCommand = (filenames) => {
  return `next lint --fix --file ${getStringOfFilenames({
    filenames,
  })} --max-warnings=0`;
};

module.exports = {
  ".*.{js,jsx,ts,tsx}": [buildEslintCommand, "prettier --write"],
};
```

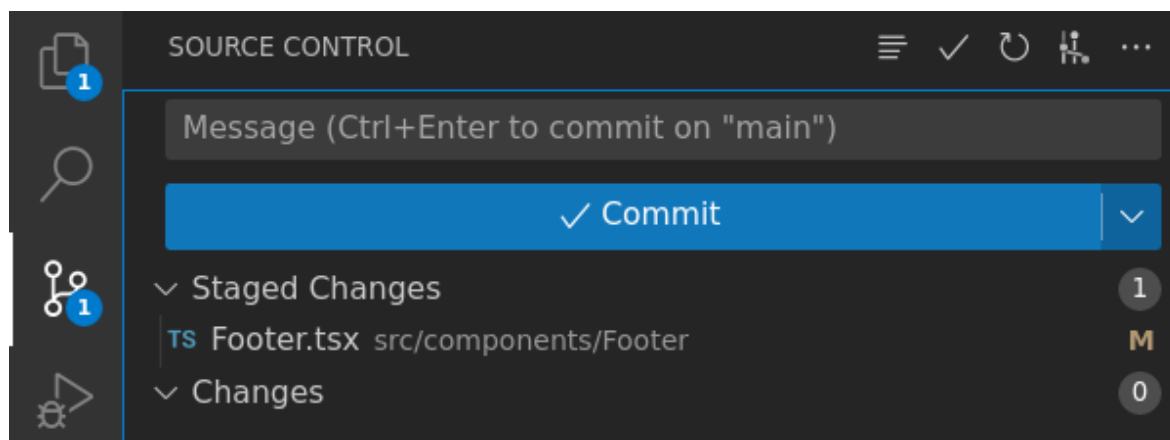
**Kod 10: Fragment pliku „lintstagedrc.js”**

Fragment kodu 10 zawarł w sobie funkcję odpowiedzialną za zbudowanie komend do uruchomienia narzędzi Eslint i Prettier tylko dla plików znajdujących się w obszarze pomostowym w repozytorium Git. Następnie dodano do pliku „package.json” polecenie „check:quick”, które pozwoliło na uruchomienia Lint-staged.

```
{
  [...]
  "scripts": {
    [...]
    "check:quick": "lint-staged"
  },
  ...
}
```

**Kod 11: Fragment pliku „package.json” z dodaną komendą „check:quick”**

### 11.3.2. PREZENTACJA DZIAŁANIA



Rys. 11: Zmiany które znajdowały się w sekcji staged

Na rysunku 11 pokazano, że w fazie pomostowej znalazł się tylko jeden plik „Footer.tsx”. Później uruchomiono komendę „npm run check:quick” i zmierzono czas potrzebny na jej wykonanie.

```
→ automatic-tests git:(main) x time (npm run check:quick > /dev/null 2>&1)
( npm run check:quick > /dev/null 2>&1; ) 2,76s user 0,47s system 102% cpu 3,153 total
→ automatic-tests git:(main) x
```

Rys. 12: Czas działania komendy „npm run check:quick”

Rysunek 12 zaprezentował, że komenda „npm run check:quick”, która działała tylko na jednym pliku zajęła 2,76 sekundy.

## 12. TESTY JEDNOSTKOWE I INTEGRACYJNE

Testy jednostkowe i integracyjne wymagają środowiska w którym można je uruchomić. Do tego celu wykorzystano platformę testową Jest oraz bibliotekę React Testing Library.

1. Jest – jest to framework do testowania aplikacji JavaScript. Zawiera wbudowane środowisko do uruchamiania testów, oraz narzędzia do śledzenia pokrycia kodu przez testy.<sup>37</sup>
2. React Testing Library – to biblioteka do testowania komponentów React w postaci zbliżonej tak jak robiłby to użytkownik końcowy.<sup>38</sup>

### 12.1. KONFIGURACJA

Należało zainstalować wymagane paczki komendą “`npm install --save-dev jest jest-environment-jsdom ts-jest @testing-library/react @testing-library/jest-dom @testing-library/user-event`”. Oraz w głównej gałęzi projektu dodać plik konfiguracyjny „`jest.config.js`”.<sup>39</sup>

---

<sup>37</sup>Witryna internetowa: <https://jestjs.io/docs/getting-started>, 04.03.2022

<sup>38</sup>Witryna internetowa: <https://testing-library.com/docs/react-testing-library/intro/>, 26.12.2022

<sup>39</sup>Witryna internetowa: <https://nextjs.org/docs/testing#setting-up-jest-with-the-rust-compiler>, 26.12.2022

```

const nextJest = require("next/jest");

const createJestConfig = nextJest({
  dir: "./",
});

/** @type {import('jest').Config} */
const customJestConfig = {
  moduleDirectories: ["node_modules", "<rootDir>/"],
  moduleNameMapper: {
    "^@/components/(.*)$": "<rootDir>/src/components/$1",
  },
  testEnvironment: "jest-environment-jsdom",
};

module.exports = createJestConfig(customJestConfig);

```

**Kod 12: Zawartość stworzonego pliku „jest.config.js”**

Dodano do „package.json” komendę pozwalającą na uruchomienie wszystkich testów jednostkowych i integracyjnych znajdujących się w projekcie.

```

{
  [...]
  "scripts": {
    [...]
    "test": "jest"
  },
  ...
}

```

**Kod 13: Fragment pliku „package.json” z dodaną komendą test**

## 12.2. TESTY JEDNOSTKOWE

Skupiono się w projekcie na przetestowaniu kluczowych w poprawnym działaniu komponentów, takich jak pole tekstowe, pole wyboru i pole wyboru z listy rozwijanej.



Każdy z testów został napisany zgodnie ze wzorcem AAA omówionym w punkcie 4.2.2. Testy są niezależne od siebie, oraz sprawdzana była w nich atomowa funkcjonalności.

```
[...]

describe("<Checkbox />", () => {
  it("displays label", async () => {
    [...]
  });

  it("displays value", async () => {
    [...]
  });

  it("executes onChange", async () => {
    [...]
  });

  it("executes onBlur", async () => {
    [...]
  });
});
```

**Kod 14: Fragment pliku „src/components/Inputs/Checkbox/Checkbox.test.tsx” z zgrupowanymi testami**

W kodzie 14 pokazano fragment pliku testującego pole wyboru. Funkcja „describe” została wykorzystana do grupowania testów. W pierwszym parametrze podano nazwę „<Checkbox />” co odnosiło się do testowanego pola wyboru. W drugim parametrze funkcji „describe” przekazano zgrupowane testy, w których funkcja „it” dla pierwszego parametru przyjęła nazwę testu, np. „displays label” (wyświetla etykietę). Nazwa „displays label” określiła, że w teście była sprawdzana etykieta pola. W drugim parametrze funkcja „it” zawarte zostały oczekiwania odnośnie testowanej funkcjonalności.

```
[...]

describe("<Checkbox />", () => {
  [...]
  it("executes onChange", async () => {
    // Podział testu na etapy zgodnie ze wzorcem AAA
    // Etap „arrange”
    const onChangeMock = jest.fn((e) => e);
    const { button } = setup({ ...requiredProps, onChange:
onChangeMock });

    // Etap „act”
    await userEvent.click(button);

    // Etap „assert”
    expect(onChangeMock.mock.results[0].value).toBeTruthy();
  });
  ...
});
```

**Kod 15: Fragment pliku „src/components/Inputs/Checkbox/Checkbox.test.tsx” z rozwinięciem testu „executes onChange”**

W kodzie 15 rozwinięto test „executes onChange”, który sprawdzał poprawność wywołania funkcji „onChange” po wywołaniu akcji zmiany w polu wyboru. Test został podzielony zgodnie ze wzorcem AAA na trzy etapy:

1. arrange – w tej części została przygotowana funkcja „onChangeMock”, którą później przekazano do pola w funkcji „setup”,
2. act – w tym etapie wykonano akcję kliknięcia w pole wyboru,
3. assert – ta część sprawdziła czy funkcja „onChangeMock” po wykonaniu akcji kliknięcia otrzymała wartość „true”.

### 12.2.1. PREZENTACJA DZIAŁANIA

Będąc w katalogu projektu uruchomiono test dla pola wyboru komendą „npm run test src/components/Inputs/Checkbox/Checkbox.test.tsx”. Gdy któryś z testów został wykonany z błędem to w konsoli widniał zwrócony wyjątek z informacją o problemie.

```

→ automatic-tests git:(main) x npm run test src/components/Inputs/Checkbox/Checkbox.test.tsx
> automatic-tests@0.1.0 test
> jest src/components/Inputs/Checkbox/Checkbox.test.tsx

warn - You have enabled experimental feature (appDir) in next.config.js.
warn - Experimental features are not covered by semver, and may cause unexpected or broken ap
info - Thank you for testing `appDir` please leave your feedback at https://nextjs.link/app-

FAIL src/components/Inputs/Checkbox/Checkbox.test.tsx
  <Checkbox />
    ✓ displays label (76 ms)
    ✓ displays value (20 ms)
    ✗ executes onChange (59 ms)
    ✓ executes onBlur (41 ms)

● <Checkbox /> > executes onChange

TypeError: Cannot read properties of undefined (reading 'value')

   56 |
   57 |     // Etap assert
>  58 |     expect(onChangeMock.mock.results[0].value).toBeTruthy();
      |                                           ^
   59 |   });
   60 |
   61 |   it("executes onBlur", async () => {

      at Object.value (src/components/Inputs/Checkbox/Checkbox.test.tsx:58:41)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 3 passed, 4 total
Snapshots:   0 total
Time:        1.128 s
Ran all test suites matching /src\/components\/Inputs\/Checkbox\/Checkbox.test.tsx/i.
→ automatic-tests git:(main) x

```

**Rys. 13: Test z pliku „src/components/Inputs/Checkbox/Checkbox.test.tsx” został wykonany z błędem**

Test dla pola wyboru został wykonany z błędem, na rysunku 13 przedstawiono wynik otrzymany z konsoli. Zawiodła część „executes onChange”, pozostałe testy „displays label”, „displays value”, „executes onBlur” zakończył się sukcesem.

```

→ automatic-tests git:(main) x npm run test src/components/Inputs/Checkbox/Checkbox.test.tsx
> automatic-tests@0.1.0 test
> jest src/components/Inputs/Checkbox/Checkbox.test.tsx

warn - You have enabled experimental feature (appDir) in next.config.js.
warn - Experimental features are not covered by semver, and may cause unexpected or broken ap
info - Thank you for testing `appDir` please leave your feedback at https://nextjs.link/app-f

PASS src/components/Inputs/Checkbox/Checkbox.test.tsx
  <Checkbox />
    ✓ display label (80 ms)
    ✓ display value (20 ms)
    ✓ executes onChange (62 ms)
    ✓ executes onBlur (40 ms)

Test Suites: 1 passed, 1 total
Tests: 4 passed, 4 total
Snapshots: 0 total
Time: 1.109 s
Ran all test suites matching /src\components\Inputs\Checkbox\Checkbox.test.tsx/i.
→ automatic-tests git:(main) x

```

**Rys. 14: Wynik ponownego uruchomienie testu z rysunku 13 po rozwiązaniu problemu**

Po rozwiązaniu problemu z rysunku 13 wszystkie testy zakończyły się powodzeniem, grupa testów „<Checkbox />” przeszła pomyślnie.

### 12.3. TESTY INTEGRACYJNE

Imię

Nazwisko

Adres e-mail

Opcjonalny zapis na warsztaty

Wybierz

Rejestrując się akceptujesz regulamin.

Zapisz się

**Rys. 15: Wygląd testowanego formularza do rejestracji**

W projekcie został przygotowany test integracyjny dla formularza rejestracyjnego pokazanego na rysunku 15. W teście wykonano sprawdzenie poprawności działania

połączenia pomiędzy systemem formularzy React Hook Form, a komponentami tworzącymi przyciski, pole tekstowe i pole wyboru z listy rozwijanej.

1. React Hook Form – jest to biblioteka do tworzenia formularzy, która dostarcza wbudowaną walidację wprowadzonych danych. Dodatkowo React Hook Form izoluje poszczególne pola co pozwala na odświeżanie tylko wybranych fragmentów bez konieczności ponownego przebudowania całego formularza.

```
[...]

describe("<RegistrationForm />", () => {
  it("displays all fields", async () => {
    [...]
  });

  it("displays required fields message", async () => {
    [...]
  });

  it("displays not valid email message", async () => {
    [...]
  });

  it("executes onSubmit", async () => {
    [...]
  });
});
```

**Kod 16: Fragment pliku**

**„src/components/RegistrationForm/RegistrationForm.test.tsx”**

Kod 16 przedstawił fragment pliku dla testu integracyjnego formularza rejestracyjnego. Zostały w pliku zawarte testy:

1. *displays all fields* (wyświetla wszystkie pola) – sprawdzono czy pola imię, nazwisko, adres e-mail, zapis na warsztaty widniały w formularzu,
2. *displays required fields message* (wyświetla wiadomość o wymaganych polach), *displays not valid email message* (wyświetla wiadomość o niepoprawnym adresie e-mail) – przetestowano działanie walidacji przy wprowadzeniu błędnych danych,

3. *executes onSubmit* (wykonuje *onSubmit*) – sprawdzono poprawności zwracanych danych po zatwierdzeniu formularza.

### 12.3.1. PREZENTACJA DZIAŁANIA

W katalogu projektu uruchomiono komendę „`npm run test src/components/RegistrationForm/RegistrationForm.test.tsx`”.

```
→ automatic-tests git:(main) x npm run test src/components/RegistrationForm/RegistrationForm.test.tsx
> automatic-tests@0.1.0 test
> jest src/components/RegistrationForm/RegistrationForm.test.tsx

warn - You have enabled experimental feature (appDir) in next.config.js.
warn - Experimental features are not covered by semver, and may cause unexpected or broken application
info - Thank you for testing `appDir` please leave your feedback at https://nextjs.link/app-feedback

PASS src/components/RegistrationForm/RegistrationForm.test.tsx
  <RegistrationForm />
    ✓ displays all fields (179 ms)
    ✓ displays required fields message (327 ms)
    ✓ displays not valid email message (323 ms)
    ✓ executes onSubmit (836 ms)

Test Suites: 1 passed, 1 total
Tests: 4 passed, 4 total
Snapshots: 0 total
Time: 4.693 s
Ran all test suites matching /src\/components\/RegistrationForm\/RegistrationForm.test.tsx/i.
→ automatic-tests git:(main) x
```

**Rys. 16: Wynik działania komendy „`npm run test src/components/RegistrationForm/RegistrationForm.test.tsx`”**

Wynik przeprowadzonego testu integracyjnego z rysunku 16 pokazał wykonanie czterech testów. Każdy z poszczególnych testów „*displays all fields*”, „*displays required fields message*”, „*displays not valid email message*” i „*executes onSubmit*” przeszedł pozytywnie.

## 13. TESTY END TO END

Do tworzenia testów End to End zostało wykorzystane narzędzie Cypress. Jest to darmowe otwarte źródłowe oprogramowanie służące do sprawdzania aplikacji internetowych. Cypress umożliwia inicjalizację, pisanie, uruchamianie oraz debugowanie testów. Dodatkowo to narzędzie zawiera funkcje takie jak:

1. zarządzanie ruchem sieciowym – pozwala na przechwycenie zapytania, wykonanie na nim operacji i podjęciu decyzji czy powinno być przesłane dalej,
2. nagrywanie ekranu – możliwość zobaczenia nagrań i zrzutów ekranu z testu, który został wykonany z błędem,
3. uruchomienie testów w przeglądarkach internetowych – testy mogą być uruchamiane w przeglądarkach z rodziny Chrome i Firefox.<sup>40</sup>

### 13.1. KONFIGURACJA

Zainstalowano Cypress w projekcie za pomocą „npm install cypress --save-dev”. Dodano do pliku „package.json” komendę „cypress” i „test:e2e”. Komenda „cypress” pozwoliła na uruchomienie narzędzia Cypress w formie graficznej. Komenda „test:e2e” umożliwiła wystartowanie wszystkich testów stworzonych w Cypress.

```
{
  [...]
  "scripts": {
    [...]
    "test:e2e": "cypress run",
    "cypress": "cypress open"
  },
  ...
}
```

**Kod 17: Fragment pliku „package.json” z dodaną komendą „cypress” i „test:e2e”**

<sup>40</sup>Witryna internetowa: <https://docs.cypress.io/guides/overview/why-cypress>, 04.03.2023

## 13.2. PREZENTACJA DZIAŁANIA

```
[...]

const REGISTRATION_URL = "/rejestracja";

context("Registration", () => {
  beforeEach(() => {
    cy.visit(REGISTRATION_URL);
  });

  it("has registration form", () => {
    [...]
  });

  it("can register", () => {
    [...]
  });

  it("displays required fields message", () => {
    [...]
  });

  it("displays not valid email message", () => {
    [...]
  });
});
```

**Kod 18: Fragment pliku „cypress/e2e/spec/registration.spec.cy.ts”**

W kodzie 18 pokazano fragment stworzonego test End to End. Test weryfikował proces rejestracji poprzez w pierwszej kolejności wejście na stronę „/rejestracja” i późniejsze wykonanie następujących sprawdzeń:

1. *has registration form* (posiada formularz rejestracyjny) – przetestowano czy w formularzu rejestracyjnym wyświetlają się wszystkie pola,
2. *can register* (można zarejestrować się) – zweryfikowano czy po wpisaniu danych do pól formularza rejestracyjnego można było się zarejestrować,



3. *displays required fields message* (wyświetla wiadomość o wymaganych polach) – sprawdzono czy wyświetlały się wiadomości o wymaganym polu po próbie zatwierdzenia formularza rejestracyjnego bez wprowadzenia oczekiwanych danych,
4. *displays not valid email message* (wyświetla wiadomość o niepoprawnym adresie e-mail) – przetestowano czy wyświetlała się wiadomość o niepoprawnym adresie e-mail po wpisaniu w pole z adresem e-mail niewłaściwej wartości.

```
→ automatic-tests git:(main) x npm run test:e2e
> automatic-tests@0.1.0 test:e2e
> cypress run

=====

(Run Starting)

Cypress:      12.7.0
Browser:      Electron 106 (headless)
Node Version: v16.19.0 (/home/floreq/.nvm/versions/node/v16.19.0/bin/node)
Specs:        1 found (registration.spec.cy.ts)
Searched:     cypress/e2e/**/*.cy.{js,jsx,ts,tsx}

Running: registration.spec.cy.ts (1 of 1)

Registration
  ✓ has registration form (1791ms)
  ✓ can register (2801ms)
  ✓ displays required fields message (1413ms)
  ✓ displays not valid email message (1564ms)
  ✓ can not register with not valid email (2494ms)

5 passing (10s)
```

**Rys. 17: Fragment wyniku zwróconego po wykonaniu komendy „npm run test:e2e”**

Na rysunek 17 pokazano otrzymany wynik po uruchomieniu komendy „npm run test:e2e”, Można było odczytać, że wykonana została jedna grupa testów o nazwie „Registration” (rejestracja) składająca się z pięciu testów.



**Rys. 18: Wynik wykonania testu „registration.spec.cy.ts” z poziomu interfejsu graficznego z narzędzia Cypress**

Na rysunek 18 powtórzono wykonanie grupy testów z rysunku 17, tym razem z użyciem interfejsu graficznego narzędzia Cypress.

## 14. ŚRODOWISKO WDROŻENIOWE CI/CD

Do przygotowania środowiska wdrożeniowego wykorzystano narzędzie GitLab oraz wirtualny prywatny serwer.

1. GitLab – jest to internetowe repozytorium Git, które pozwala na tworzenie prywatnych i publicznych repozytoriów. Platforma umożliwia śledzenie problemów, dodawanie dokumentacji projektu, wykonywanie recenzji kodu i zapewnia mechanizmy tworzenia potoków CI/CD.
2. Wirtualny prywatny serwer – jest to odizolowane środowisko utworzone na fizycznym serwerze z wykorzystaniem technologii wirtualizacji, któremu przydzielono określoną pulę zasobów.<sup>41</sup>

<sup>41</sup>Witryna internetowa: <https://www.ovhcloud.com/pl/vps/definition/>, 17.04.2023

### 14.1. WŁASNY HOSTING LUB ROZWIĄZANIE CHMUROWE

GitLab udostępnia wydanie społecznościowe, które jest aplikacją o otwartym kodzie źródłowym. Umożliwia uruchomienie repozytoriów Git na własnym serwerze. Daje to korzyści polegające na pełnej kontroli nad własnymi danymi, oraz nad tym, kto może mieć do nich dostęp.<sup>42</sup>

Rozwiązanie chmurowe zapewnia skonfigurowane i gotowe narzędzie do użycia po zalogowaniu. Nie wymagana jest wiedza techniczna jak stworzyć i utrzymać własny hosting, oraz nie trzeba pamiętać o aktualizacjach i kopiach zapasowych danych.

Rozwiązanie chmurowe w zupełności wystarczyło na potrzeby pracy inżynierskiej. Dlatego, że darmowy plan w GitLab w momencie tworzenia projektu udostępnił przede wszystkim 400 minut dla potoków CI/CD miesięcznie.<sup>43</sup>

### 14.2. PRZYGOTOWANIE OBRAZU KONTENERA APLIKACJI

Do przygotowania obrazu skonteneryzowanej aplikacji użyto udostępnionego rozwiązania przez Next.js, które pozwoliło na publikację aplikacji z użyciem silnika kontenerów Docker.<sup>44</sup>

W celu umożliwienia konteneryzacji aplikacji, dodano zgodnie z dokumentacją Next.js plik „Dockerfile” do głównej gałęzi projektu, oraz zmodyfikowano plik „.next.config.js” w celu włączenia trybu „standalone” (samodzielnego).

1. Tryb samodzielny – podczas budowy aplikacji dołącza do wynikowego kodu tylko wymagane pliki z katalogu „node\_modules”.<sup>45</sup>

---

<sup>42</sup>Witryna internetowa: <https://www.openproject.org/blog/why-self-hosting-software/>, 12.02.2023

<sup>43</sup>Witryna internetowa: <https://about.gitlab.com/pricing/>, 12.02.2023

<sup>44</sup>Witryna internetowa: <https://nextjs.org/docs/deployment#docker-image>, 21.04.2023

<sup>45</sup>Witryna internetowa: <https://nextjs.org/docs/advanced-features/output-file-tracing#automatically-copying-traced-files>, 21.04.2023

```
[...]
const nextConfig = {
[...],
output: "standalone",
};
...
```

**Kod 19: Fragment pliku „.next.config.js” zawierający opcję włączającą tryb samodzielny**

### 14.3. KONFIGURACJA SERWERA

Do opublikowania aplikacji internetowej opisanej w punkcie 10 wykorzystano wirtualny prywatny serwer z systemem operacyjnym Ubuntu 20.04 wykupionym w platformie chmurowej Microsoft Azure.

Na maszynie zostały otworzone dwa porty protokołu HTTP i SSH.

1. *Hypertext Transfer Protocol* (hipertekstowy protokół transferowy) – protokół przesyłania dokumentów hipertekstowych, na którym domyślna komunikacja pomiędzy klientem, a źródłem odbywa się na porcie 80,<sup>46</sup>
2. *Secure Shell* (bezpieczna powłoka) – standard protokołów komunikacyjnych używanych w sieciach komputerowych TCP/IP, których zadaniem jest zwykle umożliwienie bezpiecznego zalogowania się na serwer z poziomu wiersza poleceń. Domyślnie SSH działa na porcie 22.<sup>47</sup>

Następnie stworzono na serwerze użytkownika „deployer” któremu wygenerowano parę klucz poleceniem „ssh-keygen -b 4096”. Później dodano utworzony klucz publiczny do pliku „.ssh/authorized\_keys” w katalogu domowym użytkownika „deployer”.

1. *authorized\_keys* (autoryzowane klucze) – jest to plik pozwalający na zalogowanie się na danego użytkownika za pomocą protokołu SSH. Do pliku „authorized\_keys” dodaje się klucz publiczny, a przy logowaniu wykorzystuje się odpowiadający klucz prywatny.

Później w punkcie 14.4 klucz prywatny został wykorzystany do umożliwienia dostępu do serwera z poziomu GitLab.

<sup>46</sup>Witryna internetowa: [https://pl.wikipedia.org/wiki/Hypertext\\_Transfer\\_Protocol](https://pl.wikipedia.org/wiki/Hypertext_Transfer_Protocol), 17.04.2023

<sup>47</sup>Witryna internetowa: [https://pl.wikipedia.org/wiki/Secure\\_Shell](https://pl.wikipedia.org/wiki/Secure_Shell), 17.04.2023

Dodatkowo na serwerze zainstalowano oprogramowanie Docker, które zostało wykorzystane do uruchomienia aplikacji internetowej z punktu 10.

1. Docker – jest to otwarte źródłowe oprogramowanie służące do realizacji wirtualizacji z poziomu systemu operacyjnego, tak zwanej konteneryzacji. Narzędzie pozwala na zbudowanie obrazu aplikacji i późniejsze jej uruchomienie w postaci kontenera.<sup>48</sup>

Przypisano użytkownika „deployer” do grupy „docker” w celu umożliwienia dostępu do narzędzia Docker.

#### 14.4. KONFIGURACJA POTOKU CI/CD

Wzorując się poradnikiem „How To Set Up a Continuous Deployment Pipeline with GitLab CI/CD on Ubuntu 18.04” („Jak utworzyć potok ciągłego wdrożenia z GitLab CI/CD na Ubuntu 18.04”)<sup>49</sup> autora Mike Nöthiger na stronie „gitlab.com” stworzono projekt „automated-tests”, do którego przesłano repozytorium aplikacji omówionej w punkcie 10.

Następnie w ustawieniach projektu GitLab w zakładce „CI/CD” i sekcji „Variables” dodano trzy zmienne opisane w tabeli 1.

**Tabela 1. Dodane wewnętrzne zmienne projektu Gitlab**

Key (Klucz)	Value (Wartość)	Type (Typ)	Flags (Flagi)
ID_RSA	Klucz prywatny wygenerowany w punkcie 14.3. <b>Ważne, wprowadzoną wartość zakończono znakiem końca linii (enterem)</b>	File (Plik)	Expand variable reference (Rozwiń odwołanie do zmiennej)
SERVER_IP	Publiczny adres IP serwera	Variable	Mask variable (zamaskowana zmienna), Expand variable reference
SERVER_USER	Nazwa użytkownika stworzonego	Variable	Mask variable,

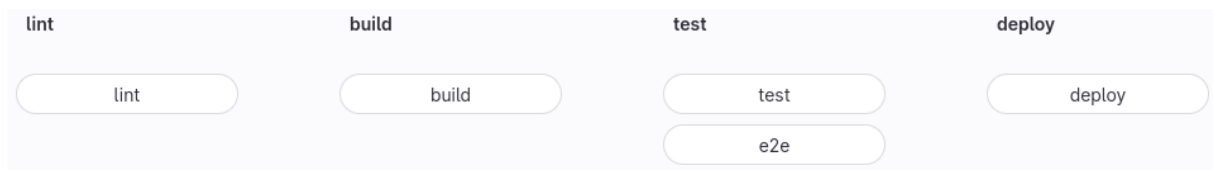
<sup>48</sup>Witryna internetowa: [https://pl.wikipedia.org/wiki/Docker\\_\(oprogramowanie\)](https://pl.wikipedia.org/wiki/Docker_(oprogramowanie)), 17.04.2023

<sup>49</sup>Witryna internetowa: <https://www.digitalocean.com/community/tutorials/how-to-set-up-a-continuous-deployment-pipeline-with-gitlab-ci-cd-on-ubuntu-18-04>, 17.04.2023

	w punkcie 14.3		Expand variable reference
--	----------------	--	---------------------------

Później stworzono w głównej gałęzi projektu plik „.gitlab-ci.yml” z przygotowaną konfiguracją potoku CI/CD.

1. .gitlab-ci.yml – jest to plik o rozszerzeniu YAML, który służy do konfiguracji środowisk i zadań uruchamianych w potoku CI/CD projektu GitLab.



**Rys. 19: Wizualna reprezentacja stworzonego pliku „.gitlab-ci.yml”**

Na rysunku 19 pokazano wizualną reprezentację przygotowanego potoku CI/CD, wydzielono w nim cztery etapy:

1. lint – etap odpowiedzialny za uruchomienie komend „npm run lint” i „npm run format:check” przygotowanych w punktach 11.1 i 11.2. Następuje w nim sprawdzenie kodu źródłowego pod względem poprawności reguł statycznej analizy kodu i ustalonego stylu formatowani,
2. build – etap wykonujący budowę aplikacji przy użyciu pliku „Dockerfile” opisanego w punkcie 14.2,
3. test – etap uruchamiający dwa niezależne zadania „test” i „e2e”, pierwsze zadanie wykonuje polecenie „npm run test” z punktu 12, a drugie „npm run test:e2e” z punktu 13,
4. deploy – etap przeprowadzający publikację zbudowanej aplikacji z etapu „build” na serwerze skonfigurowanym w punkcie 14.3.

#### 14.4.1. OMÓWIENIE PRZYGOTOWANEGO PLIKU „.GITLAB-CI.YML”

```
image: node:16.19.0-alpine

cache:
  key: $CI_COMMIT_REF_SLUG
  paths:
    - .npm/

stages:
  - lint
  - build
  - test
  - deploy
```

**Kod 20:** Fragment pliku „.gitlab-ci.yml” określający dostępne etapy procesu CI/CD

W pierwszej linii pliku „.gitlab-ci.yml” z kodu 20 umieszczono słowo kluczowe *image* (obraz) o wartości „node:16.19.0-alpine”, które zdefiniowało domyślne środowisko uruchomieniowe późniejszych etapów potoku CI/CD.

Następnie dodano sekcję „*cache*” (pamięć podręczna), która pozwoliła na wykorzystanie wbudowanego mechanizmu pamięci podręcznej w menadżerze paczek „npm” w celu usprawnienia późniejszego instalowania modułów projektowych przez komendę „npm ci --cache .npm --prefer-offline”.

Kolejno wprowadzono klucz *stages* (etapy), który określił dostępne etapy i kolejność ich wykonywania, zaczynając od „lint” przechodząc przez „build” i „test”, a kończąc na „deploy”.

```
[...]
```

```
lint:
```

```
  stage: lint
```

```
  script:
```

- npm ci --cache .npm --prefer-offline
- npm run lint && npm run format:check

```
  rules:
```

- if: \$CI\_PIPELINE\_SOURCE == "merge\_request\_event"
- if: \$CI\_COMMIT\_BRANCH == "main"

```
...
```

**Kod 21: Fragment pliku „.gitlab-ci.yml” zawierający etap „lint”**

Kod 21 zawarł konfigurację pierwszego zadania o nazwie „lint” i przynależącego do etapu „lint”. Klucz „*script*” (skrypt) określił komendy, które zostaną uruchomione podczas wykonywania tego zadania.

Dodatkowo dołączono klucz „*rules*” (reguły) definiujące momenty wywołania zadania. W tym wypadku zadanie zostanie uruchomione tylko gdy utworzono zdarzenie związane z prośbą o scalenie kodu lub wysłano zmiany do gałęzi „main”. „\$CI\_PIPELINE\_SOURCE” i „\$CI\_COMMIT\_BRANCH” są globalnymi zmiennymi ustawionymi przez GitLab i dostępnymi z poziomu potoku CI/CD. Pełną listę predefiniowanych zmiennych można znaleźć na stronie [https://docs.gitlab.com/ee/ci/variables/predefined\\_variables.html](https://docs.gitlab.com/ee/ci/variables/predefined_variables.html).



[...]

build:

image: docker:23.0.3

services:

- docker:23.0.3-dind

stage: build

script:

```
- docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD
$CI_REGISTRY
- docker build -t $CI_REGISTRY_IMAGE:$CI_COMMIT_SHORT_SHA .
- docker push $CI_REGISTRY_IMAGE:$CI_COMMIT_SHORT_SHA
```

...

**Kod 22: Fragment kodu „.gitlab-ci.yml” zawierający etap „build”**

W kodzie 22 pokazano fragment konfiguracji zadania „build” przynależącego do etapu „build”. Słowo kluczowe „image” nadpisało domyślnie ustawiony obraz z fragmentu kodu 20 na „docker:23.0.3” tym sposobem zmieniając środowisko uruchomieniowe z Node.js na zawierające narzędzie Docker. W celu udostępnienia komend bazujących na Docker CLI, np. „docker login”, „docker build” w „script” użyto klucza „services” (serwisy) z dodanym jednym obrazem „docker:23.0.3-dind”<sup>50</sup>. Następnie w „script” umieszczono trzy komendy:

1. docker login ... – komenda umożliwiająca zalogowanie się do prywatnego repozytorium kontenerów projektu, które jest dostępne z poziomu projektu GitLab z zakładki „*Packages and registries*” (Pakiery i repozytoria) i sekcji „*Container Registry*” (Repozytorium kontenerów),
2. docker build ... – komenda uruchamiająca proces budowy obrazu przygotowanego w punkcie 14.2,
3. docker push ... – komenda odpowiedzialna za przesłanie zbudowanego obrazu do prywatnego repozytorium kontenerów.

<sup>50</sup>Witryna internetowa: [https://docs.gitlab.com/ee/ci/docker/using\\_docker\\_build.html#use-docker-in-docker](https://docs.gitlab.com/ee/ci/docker/using_docker_build.html#use-docker-in-docker), 20.04.2023

```
[...]
test:
  stage: test
  script:
    - npm ci --cache .npm --prefer-offline
    - npm run test
[...]

e2e:
  services:
    - name: $CI_REGISTRY_IMAGE:$CI_COMMIT_SHORT_SHA
      alias: frontend
  image: cypress/browsers:node-16.18.1-chrome-110.0.5481.96-1-ff-109.0-edge-110.0.1587.41-1
  stage: test
  script:
    - npm ci --cache .npm --prefer-offline
    - CYPRESS_BASE_URL=http://frontend:3000 npm run test:e2e
...
```

**Kod 23: Fragment pliku „gitlab-ci.yml” zawierający etap „test”**

Kod 23 zawarł w sobie fragment dwóch równolegle wykonywanych zadań „test” i „e2e” przypisanych do etapu „test”. Pierwsze zadanie uruchamia testy jednostkowe i integracyjne stworzone w punkcie 12, a zadanie drugie realizuje testy End to End przygotowane w punkcie 13. Warto zaznaczyć, że zadanie „e2e” wykorzystuje „services”, w którym uruchamiany jest obraz kontenera ze zbudowaną stroną podczas etapu „build”.

```

[...]  

deploy:  

  stage: deploy  

  script:  

    - chmod og= $ID_RSA  

    - apk update && apk add openssh-client  

    - >  

      ssh -i $ID_RSA -o StrictHostKeyChecking=no  

$SERVER_USER@$SERVER_IP "  

  (docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD  

$CI_REGISTRY) &&  

  (docker pull $CI_REGISTRY_IMAGE:$CI_COMMIT_SHORT_SHA) &&  

  (docker container rm -f frontend || true) &&  

  (docker run -d -p 80:3000 --name frontend $CI_REGISTRY_IMAGE:  

$CI_COMMIT_SHORT_SHA) &&  

  (docker tag $CI_REGISTRY_IMAGE:$CI_COMMIT_SHORT_SHA  

$CI_REGISTRY_IMAGE:latest) &&  

  (docker push $CI_REGISTRY_IMAGE:latest)  

  "  

only:  

  - main

```

**Kod 24: Fragment pliku „.gitlab-ci.yml” zawierający etap „deploy”**

Kod 24 zawarł konfigurację zadania „deploy” należącego do etapu „deploy”. W kluczu „script” użyto zmienne „\$ID\_RSA”, „\$SERVER\_USER” i „\$SERVER\_IP” dodane do projektu GitLab w punkcie 14.4.

Komenda „ssh -i \$ID\_RSA -o StrictHostKeyChecking=no \$SERVER\_USER@\$SERVER\_IP [...]” umożliwiła na połączenie się z serwerem skonfigurowanym w punkcie 14.3 i wykonania następujących poleceń:

1. docker login ... – komenda omówiona podczas analizy kodu 22,
2. docker pull ... – polecenie pobierające zbudowany obraz z etapu „build”,
3. docker container ... – komenda zatrzymująca i usuwająca wcześniejszą wersję obrazu aplikacji,
4. docker run ... – polecenie uruchamiające pobrany obraz aplikacji z przypisaniem portu 80 serwera do portu 3000 uruchomionego kontenera,

5. `docker push [...]:latest` – komenda wysyłająca pobrany obraz z powrotem do prywatnego repozytorium kontenerów z ustawioną etykietą na „*latest*” (najnowszy).

## 14.5. PREZENTACJA DZIAŁANIA

Status	Pipeline	Triggerer	Stages	
<div>running</div> <div>In progress</div>	Removed manual <a href="#">#843818481</a> main -> 12821969 latest			<div> </div>
<div>passed</div> <div>00:07:44 21 hours ago</div>	Added manual <a href="#">#842651644</a> main -> 54002abd latest			<div> </div>

**Rys. 20: Prezentacja aktualnie działającego i pozytywnie zakończonego potoku CI/CD**

Na rysunku 20 zaprezentowano dwa potoki CI/CD, pierwszy o statusie „*running*” (działający) i drugi „*passed*” (zakończony pomyślnie).

Status	Pipeline	Triggerer	Stages	
<div>failed</div> <div>00:07:35 20 hours ago</div>	Refactored Checkbox <a href="#">#843829886</a> main -> a3e556e2		<div> <b>Stage: test</b>  <div>  test  </div> <div>  e2e  </div> </div>	<div> </div>

**Rys. 21: Prezentacja zatrzymanego potoku CI/CD z powodu błędu**

Rysunek 21 przedstawił potok CI/CD „*Refactored Checkbox*” (przebudowany checkbox), który zakończył się niepowodzeniem. Po kliknięciu na trzeci etap zauważono, że zadanie „*test*” zostało wykonane z błędem.

```
38 PASS src/components/RegistrationForm/RegistrationForm.test.tsx
39 PASS src/components/Inputs/Select/Select.test.tsx
40 FAIL src/components/Inputs/Checkbox/Checkbox.test.tsx
41   • <Checkbox /> > executes onChange
42     expect(received).toBeTruthy()
43     Received: undefined
44       56 |
45       57 |     // Etap assert
46   > 58 |     expect(onChangeMock.mock.results[0].value).toBeTruthy();
47         |                                                ^
48       59 |   });
49       60 |
50       61 |   it("executes onBlur", async () => {
51         at Object.toBeTruthy (src/components/Inputs/Checkbox/Checkbox.test.tsx:58:48)
52 PASS src/components/Inputs/InputText/InputText.test.tsx
53 PASS src/components/Inputs/FieldWrapper/FieldWrapper.test.tsx
54 Test Suites: 1 failed, 4 passed, 5 total
55 Tests:      1 failed, 17 passed, 18 total
56 Snapshots:  0 total
57 Time:       5.601 s
58 Ran all test suites.
✓ 60 Cleaning up project directory and file based variables
62 ERROR: Job failed: exit code 1
```

**Rys. 22: Szczegóły błędnie wykonanego zadania „test”**

Na rysunku 22 zademonstrowano logi konsoli z błędnie wykonanego zadania „test” z potoku CI/CD z rysunku 21. Pokazano, że test „executes onChange” z grupy „<Checkbox />” zakończył się niepowodzeniem.

## 15. AUTOMATYCZNE TESTOWANIA DOSTĘPNOŚCI CYFROWEJ

Do przeprowadzenia automatycznego testowania dostępności cyfrowej wykorzystano narzędzie Cypress dodane w punkcie 13, oraz paczki Axe-core i Cypress-axe zainstalowane z użyciem komendy „npm install cypress-axe axe-core --save-dev”.

1. Axe-core – to silnik testujący dostępność dla stron internetowych oraz innych interfejsów użytkownika opartych na języku HTML. Axe-core zostało zbudowane w taki sposób, aby można było je zintegrować z istniejącym środowiskiem testowym.<sup>51</sup>
2. Cypress-axe - to biblioteka narzędziowa, która pozwala na automatyczne testowanie dostępności stron internetowych integrując Axe-core z Cypress.<sup>52</sup>

```
[...]
import "cypress-axe";
...
```

**Kod 25: Fragment pliku „cypress/support/e2e.ts” z dołączoną paczką „cypress-axe”**

W celu dołączenia do narzędzia Cypress możliwości testowania dostępności w kodzie 25 do pliku „cypress/support/e2e.ts” dodano odwołanie się do paczki Cypress-axe.

```
[...]
context("Homepage", () => {
  [...]

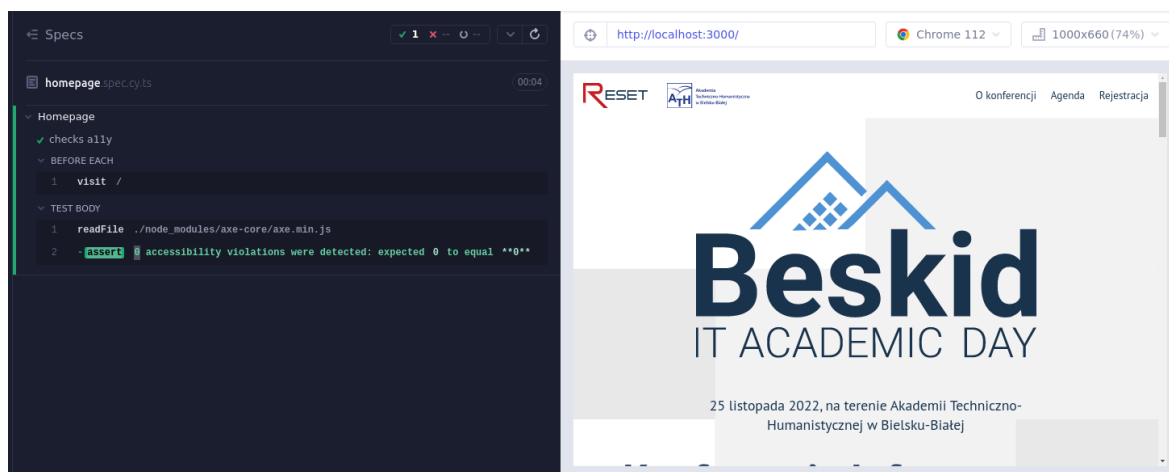
  it("checks a11y", () => {
    cy.injectAxe();
    cy.checkA11y();
  });
});
```

**Kod 26: Fragment pliku „homepage.spec.cy.ts” testującego stronę główną**

<sup>51</sup>Witryna internetowa: <https://github.com/dequelabs/axe-core>, 23.04.2023

<sup>52</sup>Witryna internetowa: <https://github.com/component-driven/cypress-axe>, 23.04.2023

W kodzie 26 pokazano test „checks a11y” (sprawdź a11y) odpowiedzialny za testowanie dostępności strony głównej, skrót „a11y” oznacza „*accessibility*” (dostępność). Użyto dwóch funkcji dostarczonych przez paczkę Cypress-axe, pierwsza „injectAxe”, zainicjalizowała funkcjonalność Axe-core, następnie „checkA11y” wykonało weryfikację dostępności na stronie głównej.



**Rys. 23: Wynik działania testu dostępności na stronie głównej**

Na rysunku 23 pokazano wynik działania testu dostępności strony głównej. Test nie znalazł problemów związanych z dostępnością.

## 16. PODSUMOWANIE

Proces automatycznego testowania aplikacji jest ważnym etapem w tworzeniu oprogramowania. Błędy mogą wystąpić z powodu każdego rodzaju zmian. Począwszy od dodawania nowej funkcjonalności, nanoszenia poprawek czy optymalizacji kodu. Testy automatyczne powinny zostać uruchomione po każdej ze zmian w aplikacji.

Zaniedbania w temacie testowania prowadzą do braku pewności w działaniu programu po dokonaniu zmian. Problemатyczne jest określenie wtedy, czy wprowadzona modyfikacja nie wpłynęła negatywnie na działającą wcześniej funkcjonalność.

Należy zaznaczyć, że automatyczne testy nie dają gwarancji, że wdrożone zmiany są wolne od błędów. Pozwalają one sprawdzić czy kod pokryty testami działał tak jak przed dodanymi zmianami.

Ręczna weryfikacja działania aplikacji nie jest łatwo powtarzalna, mogą podczas niej wystąpić błędy ludzkie. Dlatego konieczna jest automatyzacja monottonnych i powtarzalnych czynności, tak aby odciążyć testerów i pozwolić im skupić się na innych aspektach aplikacji.

Środowisko, na którym wykonywane są testy ma znaczenie dla ich skuteczności. Zaleca się, aby testy automatyczne były przeprowadzane w środowisku jak najbardziej zbliżonym do takiego, na którym oprogramowanie będzie ostatecznie działać.

Ważne jest zaznaczenie, że testy automatyczne nie zastąpią testów manualnych, ale stanowią istotne uzupełnienie procesu testowania, poprawiając jakość i skuteczność procesu wytwarzania oprogramowania.



## LITERATURA

- Krug, S. (2014). *Nie każ mi myśleć!*, Wyd. 3, Helion, Gliwice.
- Smilgin, R. (2016). *Zawód tester*, Wyd. 1, Wydawnictwo Naukowe PWN, Warszawa.
- Sommerville, I. (2020). *Inżynieria oprogramowania*, Wyd. 10, Wydawnictwo Naukowe PWN, Warszawa.

## SPIS KODU

Kod 1: Domyślna konfiguracja z pliku „.eslintrc.json” .....	22
Kod 2: Fragment pliku „.eslintrc.json” z dodanym nowym parserem i pluginem do obsługi TypeScript .....	23
Kod 3: Fragment pliku „.eslintrc.json” z dodaną regułą ostrzegającą o użyciu „console.log” .....	23
Kod 4: Fragment pliku „.eslintrc.json” z dodanymi regułami wymuszającymi sortowanie importów .....	24
Kod 5: Zawartość stworzonego pliku „.eslintignore” .....	24
Kod 6: Fragment pliku „.eslintrc.json” z dodanym rozszerzeniem dla formatowania .....	26
Kod 7: Zawartość pliku „.prettierrc” .....	27
Kod 8: Zawartość pliku „.prettierignore” .....	27
Kod 9: Fragment pliku „package.json” z dodaną komendą „format:all” .....	27
Kod 10: Fragment pliku „.lintstagedrc.js” .....	29
Kod 11: Fragment pliku „package.json” z dodaną komendą „check:quick” .....	29
Kod 12: Zawartość stworzonego pliku „jest.config.js” .....	32
Kod 13: Fragment pliku „package.json” z dodaną komendą test .....	32
Kod 14: Fragment pliku „src/components/Inputs/Checkbox/Checkbox.test.tsx” z zgrupowanymi testami .....	33
Kod 15: Fragment pliku „src/components/Inputs/Checkbox/Checkbox.test.tsx” z rozwinięciem testu „executes onChange” .....	34
Kod 16: Fragment pliku „src/components/RegistrationForm/RegistrationForm.test.tsx” .....	37
Kod 17: Fragment pliku „package.json” z dodaną komendą „cypress” i „test:e2e” .....	39
Kod 18: Fragment pliku „cypress/e2e/spec/registration.spec.cy.ts” .....	40
Kod 19: Fragment pliku „.next.config.js” zawierający opcję włączającą tryb samodzielny .....	44
Kod 20: Fragment pliku „.gitlab-ci.yml” określający dostępne etapy procesu CI/CD .....	47

Kod 21: Fragment pliku „gitlab-ci.yml” zawierający etap „lint” .....	48
Kod 22: Fragment kodu „gitlab-ci.yml” zawierający etap „build” .....	49
Kod 23: Fragment pliku „gitlab-ci.yml” zawierający etap „test” .....	50
Kod 24: Fragment pliku „gitlab-ci.yml” zawierający etap „deploy” .....	51
Kod 25: Fragment pliku „cypress/support/e2e.ts” z dołączoną paczką „cypress-axe” .....	54
Kod 26: Fragment pliku „homepage.spec.cy.ts” testującego stronę główną.....	54

## SPIS TABEL

Tabela 1. Dodane wewnętrzne zmienne projektu Gitlab.....	45
--	----

## SPIS RYSUNKÓW

Rys. 1: Schemat obrazujący proces ciągłej integracji, ciągłego dostarczania i ciągłego publikowania [źródło: <a href="https://www.redhat.com/en/topics/devops/what-cicd-pipeline">https://www.redhat.com/en/topics/devops/what-cicd-pipeline</a> , 2023] .....	12
Rys. 2: Trzy wyróżnione obszary w narzędziu Git [źródło: <a href="https://git-scm.com/about/staging-area">https://git-scm.com/about/staging-area</a> , 2023].....	13
Rys. 3: Schemat pokazujący budowę, zapisanie i uruchomienie skonteneryzowanej aplikacji [źródło: <a href="https://www.datadoghq.com/knowledge-center/containerized-applications/">https://www.datadoghq.com/knowledge-center/containerized-applications/</a> , 2023].....	14
Rys. 4: Wynik działania komendy „npm ci” .....	19
Rys. 5: Wynik działania komendy „npm run dev” z konsoli.....	21
Rys. 6: Wyświetlona strona internetowa po wejściu z przeglądarki internetowej pod otrzymany adres z rysunku 5.....	21
Rys. 7: Informacja o złamaniu reguła sortowania importów.....	25
Rys. 8: Wynik użycia opcji „Quick Fix” .....	25
Rys. 9: Czas działania komendy „npm run lint” w katalogu projekt-1.....	28
Rys. 10: Czas działania komendy „npm run format:all” w katalogu projekt-1.....	28
Rys. 11: Zmiany które znajdowały się w sekcji staged.....	30
Rys. 12: Czas działania komendy „npm run check:quick” .....	30
Rys. 13: Test z pliku „src/components/Inputs/Checkbox/Checkbox.test.tsx” został wykonany z błędem.....	35
Rys. 14: Wynik ponownego uruchomienie testu z rysunku 13 po rozwiązaniu problemu.....	36
Rys. 15: Wygląd testowanego formularza do rejestracji.....	36

Rys. 16: Wynik działania komendy „npm run test src/components/RegistrationForm/RegistrationForm.test.tsx” .....	38
Rys. 17: Fragment wyniku zwróconego po wykonaniu komendy „npm run test:e2e” .....	41
Rys. 18: Wynik wykonania testu „registration.spec.cy.ts” z poziomu interfejsu graficznego z narzędzia Cypress.....	42
Rys. 19: Wizualna reprezentacja stworzonego pliku „.gitlab-ci.yml” .....	46
Rys. 20: Prezentacja aktualnie działającego i pozytywnie zakończonego potoku CI/CD.....	52
Rys. 21: Prezentacja zatrzymanego potoku CI/CD z powodu błędu.....	52
Rys. 22: Szczegóły błędnie wykonanego zadania „test” .....	53
Rys. 23: Wynik działania testu dostępności na stronie głównej.....	55