POLITECHNIKA WROCŁAWSKA

WYDZIAŁ ELEKTRONIKI

KIERUNEK: INFORMATYKA (INF)

SPECJALNOŚĆ: INŻYNIERIA SYSTEMÓW INFORMATYCZNYCH (INS)

PRACA DYPLOMOWA

INŻYNIERSKA

Wirtualna kwiaciarnia – interaktywny system sprzedaży kwiatów

Virtual florist – an interactive system for selling flowers

AUTOR:

Karol Maśluch

PROWADZĄCY PRACĘ:

Dr inż. Tomasz Babczyński, K9

OCENA PRACY:

WROCŁAW, 2019

**Spis treści**

[Skróty 5](#_Toc26802433)

[1. Wstęp 8](#_Toc26802434)

[1.1. Wprowadzenie 8](#_Toc26802435)

[1.2. Cel i zakres pracy 9](#_Toc26802436)

[2. Analiza wymagań systemu 10](#_Toc26802437)

[2.1. Opis aktorów 10](#_Toc26802438)

[2.2. Wymagania funkcjonalne 10](#_Toc26802439)

[2.3. Wymagania niefunkcjonalne 14](#_Toc26802440)

[2.4. Przyjęte założenia projektowe 14](#_Toc26802441)

[3. Wykorzystane technologie i narzędzia programistyczne 16](#_Toc26802442)

[3.1. JDK 16](#_Toc26802443)

[3.2. Eclipse 16](#_Toc26802444)

[3.3. Spring 16](#_Toc26802445)

[3.4. Spring Boot 17](#_Toc26802446)

[3.5. Maven 17](#_Toc26802447)

[3.6. Git 18](#_Toc26802448)

[3.7. Hibernate 18](#_Toc26802449)

[3.8. MySQL 18](#_Toc26802450)

[3.9. NPM 19](#_Toc26802451)

[3.10. React.js 19](#_Toc26802452)

[3.11. Visual Studio Code 20](#_Toc26802453)

[3.12. Redux 20](#_Toc26802454)

[3.13. Axios 20](#_Toc26802455)

[3.14. Material-UI 20](#_Toc26802456)

[3.15. MDBReact 21](#_Toc26802457)

[3.16. Pozostałe 21](#_Toc26802458)

[4. Projekt systemu 22](#_Toc26802459)

[4.1. Architektura aplikacji 22](#_Toc26802460)

[4.2. Projekt bazy danych 23](#_Toc26802461)

[4.2.1. Model konceptualny 23](#_Toc26802462)

[4.2.2. Model fizyczny z ograniczeniami integralności danych 26](#_Toc26802463)

[5. Implementacja systemu 29](#_Toc26802464)

[5.1. Back-End 29](#_Toc26802465)

[5.1.1. Zależności w pliku POM.xml 29](#_Toc26802466)

[5.1.2. Struktura projektu 29](#_Toc26802467)

[5.1.3. Encje 30](#_Toc26802468)

[5.1.4. DAO 31](#_Toc26802469)

[5.1.5. Serwisy 33](#_Toc26802470)

[5.1.6. Kontrolery 36](#_Toc26802471)

[5.2. Front-End 37](#_Toc26802472)

[5.2.1. Zależności w pliku package.json 37](#_Toc26802473)

[5.2.2. Struktura 38](#_Toc26802474)

[5.2.3. Axios 39](#_Toc26802475)

[5.2.4. Redux 39](#_Toc26802476)

[5.2.5. Pages 41](#_Toc26802477)

[5.2.6. UI 43](#_Toc26802478)

[5.2.7. App 43](#_Toc26802479)

[5.3. Projekt wybranych funkcji 44](#_Toc26802480)

[5.3.1. Funkcja kontrolera odpowiedzialna za zapisywanie zdjęć 44](#_Toc26802481)

[5.3.2. Funkcja serwisu odpowiedzialna za aktualizację wpisu na wiki 45](#_Toc26802482)

[5.3.3. Funkcja znajdująca magazyn 46](#_Toc26802483)

[5.4. Struktura interfejsu graficznego 47](#_Toc26802484)

[6. Testy 52](#_Toc26802485)

[6.1. Testy manualne 52](#_Toc26802486)

[6.2. Testy wydajnościowe 53](#_Toc26802487)

[6.3. Testy funkcjonalne 54](#_Toc26802488)

[6.4. Audyt Lighthouse 55](#_Toc26802489)

[7. Podsumowanie i wnioski 55](#_Toc26802490)

[Literatura 56](#_Toc26802491)

[Zawartość płyty 57](#_Toc26802492)

# 

# Skróty

**JDK** (ang. *Java Development Kit*) – środowisko deweloperskie Javy.

**JVM** (ang. *Java Virtual Machine*) – maszyna wirtualna Javy oraz środowisko zdolne do wykonywania kodu bajtowego Javy.

**JRE** (ang. *Java Runtime Environment*) – środowisko uruchomieniowe Javy zawierające maszynę wirtualną, oraz niezbędne biblioteki potrzebne do uruchomienia aplikacji napisanych w języku Java.

**REST** (ang. *Representational state transfer*) – styl architektury, który narzuca pewne zasady tworzenia serwisów webowych m.in. architekturę klient-serwer, bezstanowość, jednorodny interfejs, metadane.

**ORM** (ang. *Object-Relational Mapping*) - technika mapowania danych pomiędzy niekompatybilnymi formatami danych przy użyciu obiektowych języków programowania. Technika ta kreuje wirtualną bazę danych, której obiekty mogą być używane wewnątrz programu.

**SQL** (ang. *Structured Query Language*) – język programowania zaprojektowany do zarządzania danymi przechowywanymi w relacyjnych bazach danych.

**JS** (ang. *JavaScript*) – obiektowy język programowania wysokiego poziomu, głownie przeznaczony do tworzenia aplikacji webowych.

**SPA** (ang. *Single-page application*) – interaktywna aplikacja webowa, której zawartość jest ładowana podczas pierwszego wejścia na dany adres, podejście to pozwala tworzyć aplikacje webowe, które działają podobniej do aplikacji desktopowych, niż stron internetowych.

**JSON** (ang. *JavaScript Object Notation*) - otwarty standard formatu wymiany danych przeznaczony do łatwego odczytu przez człowieka. Format ten składa się z pól klucz-wartość.

**XML** (ang. *Extensible Markup Language*) – otwarty standard formatu danych, przeznaczony do łatwego odczytu przez człowieka i maszynę. Format ten oprócz samych wartości zawiera także tagi opisujące zawartość. W XML możemy definiować dodatkowe zasady składni, które muszą być przestrzegane w dokumencie.

**API** (ang. *Application programming interface*) – interfejs lub protokół komunikacji, przeznaczony do uproszczenia implementacji oraz łatwiejszego utrzymania wytworzonego oprogramowania.

**JPA** (ang. *Java Persistence API*) – specyfikacja, która opisuje zarządzanie relacyjnymi danymi w aplikacjach Javy.

**CSS** (ang. *Cascading Style Sheets*) – język stylu używany do opisu wyglądu dokumentów, głównie HTML.

**POM** (ang. *Project Object Model*) – obiekt modelu projektu, zapewnia wszystkie niezbędne informacje konfiguracyjne na temat projektu m.in. nazwę, wersje, autora, zależności, użyte wtyczki.

**IDE** (ang. *Integrated Development Environment*) – zintegrowane środowisko deweloperskie.

**JDBC** (ang. Java Database Connectivity) – API definiujące w jaki sposób połączyć się z bazą danych w aplikacjach Javy.

**IoC** (ang. *Inversion of control*) – reguła programowania, która odwraca standardowy przepływ sterowania. W inwersji kontroli to ogólne frameworki wywołują specyficzny kod, w porównaniu ze standardowym przepływem gdzie, konkretny kod wywołuje generyczne biblioteki.

**AOP** (ang. *Aspect-oriented programming*) – paradygmat programowania, którego celem jest zwiększenie modularności. Pozwala na definiowanie punktów przecięcia, a następnie wywoływanie specjalnych akcji w tych punktach. Paradygmat ten pozwala oddzielić od siebie kod z części systemu, których logika nie jest ściśle powiązana ze sobą.

**EJB** (ang. *Enterprise JavaBeans*) – jedno z API Javy definiujące modularne komponenty zawierające logikę biznesową.

**JAR** (ang. *Java Archive*) – format pakowania danych, przeznaczony dla klas Javy ich powiązanych metadanych i zasobów.

**URL** (ang. *Uniform Resource Locator*) – adres do lokalizowania zasobów sieciowych.

**HTTP** (ang. *Hypertext Transfer Protocol*) – protokół warstwy aplikacji przeznaczony do przesyłania dokumentów hipertekstowych, protokół ten jest podstawą komunikacji danych w World Wide Web.

**XSRF** (ang. *Cross-site request forgery*) – typ ataku, w którym nieautoryzowane zapytania są przesyłane przez zautoryzowanego użytkownika.

**JSX** (ang. *JavaScript XML*) – rozszerzenie języka JavaScript, w swojej budowie przypomina HTML.

**HTML** (ang. *Hypertext Markup Language*) – język znaczników opisujący dokumenty przeznaczone do wyświetlania w przeglądarce.

**PNG** (ang. *Portable Network Graphics*) – format danych graficznych.

**DAO** (ang. *Data access object*) – wzorzec projektowy.

**TBV** (ang. *Tulip breaking virus*) – wirus atakujący tulipany.

1. Wstęp
   1. Wprowadzenie

W XVII-wiecznej Holandii kwiaty były bardzo popularnym tematem. Było to spowodowane olbrzymimi cenami za cebulki niektórych z gatunków tulipanów. Popyt został stworzony przez bogatą klasę średnią oraz arystokrację, które zaczęły ze sobą konkurować pokazując coraz to bardziej piękne rośliny. Tulipany do Holandii zostały zaimportowane w XVI wieku z Turcji. Ich kultywacja była możliwa dzięki francuskiemu lekarzowi i botanikowi Charlesowi de L’Ecluse, który skutecznie krzyżował tulipany wytwarzając w nich odporność na zimno. Szczególnym przypadkiem był gatunek *Semper Augusus[[1]](#footnote-1).* Gatunek ten został zaatakowany przez wirusa TBV, który spowodował wyjątkowy wygląd tulipana. Niestety choroba powodująca unikatowy wygląd rośliny była przenoszona wyłącznie przez cebulki. Te szybko zyskały wielką popularność, a ich cena to odzwierciedlała. Ocenia się, że pojedyncza cebulka mogła kosztować nawet tysiąc guldenów[[2]](#footnote-2), przy czym średni roczny dochód pojedynczej osoby wynosił 150 guldenów. Lokalni kupcy handlowali tymi roślinami tak jak obecnie maklerzy handlują akcjami.

W dzisiejszych czasach kwiaty pełnią bardziej rolę dekoracyjną i symboliczną. Kwiatami ozdabiamy wesela, uroczystości, kwiaty dajemy w ramach podziękowań i kondolencji. Są one dobrym podarunkiem dla ukochanej osoby. Głównym źródłem skąd bierzemy nasze kwiaty są kwiaciarnie. Często są to osiedlowe sklepy, które swoim zasięgiem obejmują niewielką populację. Niektóre z kwiaciarni posiadają swoje strony internetowe, na których pokazują swoje towary lub nawet dają możliwość kupowania kwiatów przez Internet. Strona internetowa jest dobrym narzędziem pozwalającym nam na zwiększenie grona potencjalnych klientów.

Internetową sprzedażą kwiatów nie zajmują się wielkie serwisy takie jak Amazon[[3]](#footnote-3) czy Allegro[[4]](#footnote-4). Oznacza to, że kwiaciarnie muszą konkurować ze sobą używając własnych rozwiązań. Potencjalnie rozwiązanie, które dawało by większą wygodę i funkcjonalność niż konkurencja, powinno osiągnąć sukces. Należy też zauważyć pewien brak w istniejących serwisach sprzedaży kwiatów, a mianowicie to, że nie udostępniają one obszernych informacji na temat sprzedawanych roślin. Jest to dużym utrudnieniem dla potencjalnych klientów, którzy chcą poprawnie pielęgnować swoje rośliny.

Moją propozycją na rozwiązanie tego problemu jest stworzenie responsywnej aplikacji webowej, która umożliwiłaby dostęp do obszernych informacji na temat kwiatów oraz wygodny i szybki sposób na ich zakup. Serwis taki dawałby wiele korzyści klientom i pracownikom. Potencjalnie mógłby pomnożyć zyski ze sprzedaży.

* 1. Cel i zakres pracy

Celem projektu jest stworzenie responsywnej aplikacji webowej umożliwiającej handel kwiatami oraz udostępniającej informacje na ich temat. Tworzony produkt powinien umożliwiać pracownikowi dodawanie/usuwanie/modyfikowanie kwiatów i wpisów na wiki, a klientowi przeglądanie, kupowanie kwiatów. Aplikacja zostanie stworzona w oparciu o trójwarstwową architekturę (warstwa prezentacji, logiki biznesowej, danych). System będzie podzielony na dwie części:

* część transakcyjna – umożliwiająca handel kwiatami,
* część informacyjna (wiki) – udostępniająca szczegółowe informacje na temat kwiatów.

Projekt zakłada stworzenie trzech elementów aplikacji:

* baza danych,
* Back-End – część aplikacji działająca jako serwer obsługujący logikę biznesową,
* Front-End – część prezentacyjna uruchamiana w przeglądarce.

1. Analiza wymagań systemu

Wymagania tworzonej aplikacji zostały sformułowane na podstawie analizy istniejących produktów, ich zalet oraz braków. Głównym celem formułowania tych wymagań powinno być osiągniecie jak największej wygody użytkowania produktu. Można to osiągnąć, poprzez wygląd aplikacji oraz zapewnianą funkcjonalność.

* 1. Opis aktorów

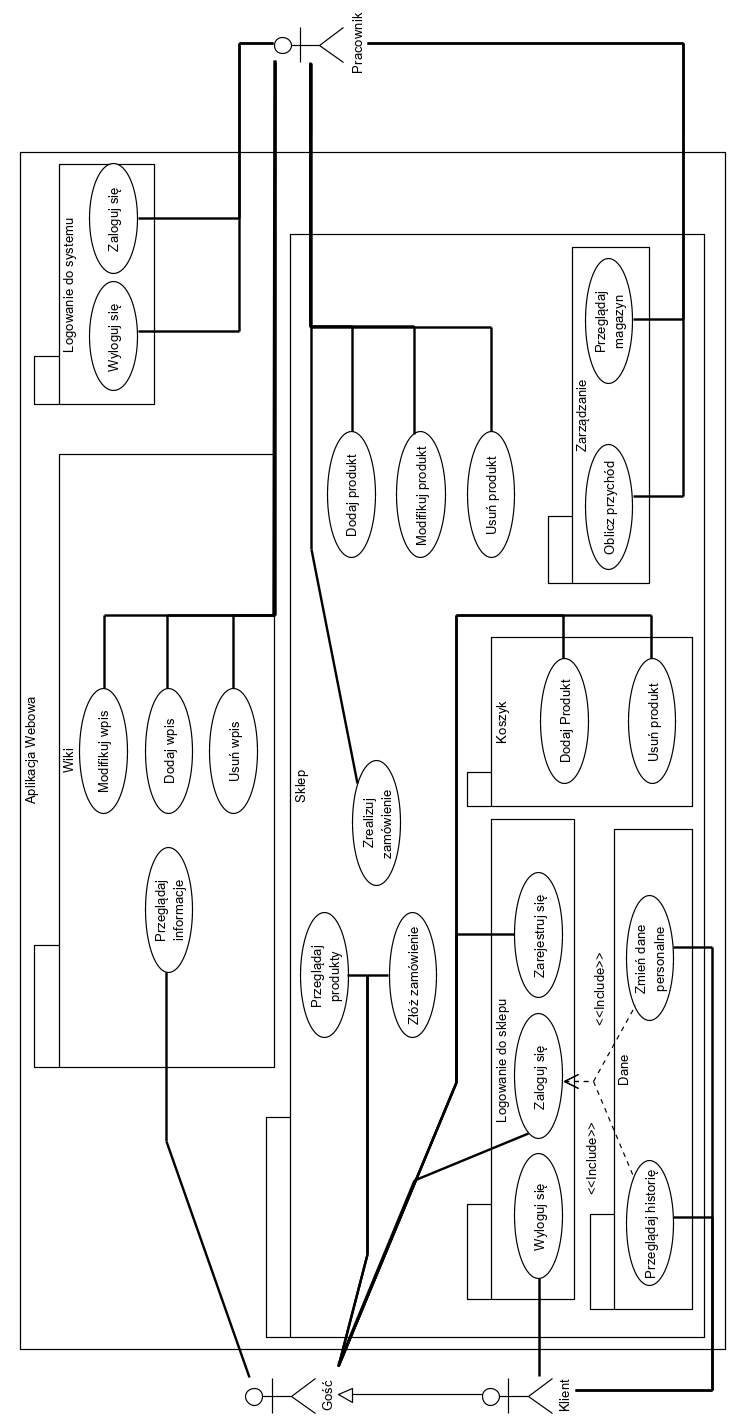
W systemie możemy wyróżnić trzech aktorów: Gościa, Klienta i Pracownika. Każdy z nich posiada swoje wymagania funkcjonalne.

* **Gość** – Jest niezalogowanym użytkownikiem systemu, posiada on możliwość oglądania produktów, dodawania ich do koszyka, kupowania produktów, po wprowadzeniu poprawnych informacji. Dodatkowo każdy gość ma dostęp do informacji znajdujących się w wiki. Gość ma możliwość rejestracji, po udanej rejestracji i logowaniu gość staje się klientem.
* **Klient** – Klient jest zalogowaną osobą posiadającą konto. Posiada on wszystkie możliwości Gościa oraz rozszerza niektóre z nich. Klient ma dostęp do historii kupionych produktów. Aktualny koszyk Klienta jest zapisywany na serwerze, przez co dostępny jest on z wielu urządzeń.
* **Pracownik** – Jest osobą która dodaje/modyfikuje/usuwa produkty oraz informacje dostępne w systemie. Jest osobą, która fizycznie realizuje zamówienia klientów i modyfikuje ich status w systemie.
  1. Wymagania funkcjonalne

Tabela . Wymagania funkcjonalne aplikacji

|  |  |  |
| --- | --- | --- |
| **Wymagania funkcjonalne** | | |
| **Id** | **Nazwa** | **Opis** |
| **Gość** | | |
| FU01 | Dostęp do informacji o kwiatach | Gość ma dostęp do szczegółowych informacji o kwiatach znajdujących się w wiki. |
| FU02 | Dostęp do produktów | Gość ma dostęp do produktów sprzedawanych w aplikacji. |
| FU03 | Dodaj produkt do koszyka | Gość ma możliwość dodania produktów do koszyka. |
| FU04 | Usuń produkt z koszyka | Gość ma możliwość usunięcia produktu z koszyka. |
| FU05 | Złóż zamówienie | Gość może złożyć zamówienie na produkty znajdujące się w koszyku, po poprawnym wypełnieniu wymaganych informacji. |
| FU06 | Zarejestruj się | Gość ma możliwość utworzenia konta. |
| FU07 | Zaloguj się | Gość ma możliwość zalogowania się do systemu sprzedaży pod warunkiem, że posiada on zarejestrowane konto. Gość po udanym zalogowaniu staje się Klientem. |
| **Klient** | | |
| FU08 | Dostęp do historii | Klient ma dostęp do historii swoich wcześniejszych zakupów. |
| FU09 | Modyfikuj dane personalne | Klient ma możliwość modyfikacji danych personalnych wprowadzonych podczas procesu rejestracji. |
| FU10 | Wyloguj się | Klient ma możliwość wylogowania się z systemu sprzedaży, po poprawnym wylogowaniu staje się Gościem. |
| **Pracownik** | | |
| FU11 | Zaloguj się do systemu | Pracownik ma możliwość zalogowania się do systemu. |
| FU12 | Wyloguj się z systemu | Pracownik ma możliwość wylogowania się z systemu. |
| FU13 | Dodaj wpis | Pracownik ma możliwość dodania wpisu do wiki. |
| FU14 | Modyfikuj wpis | Pracownik może modyfikować istniejący wpis w wiki. |
| FU15 | Usuń wpis | Pracownik ma możliwość usunięcia wpisu z wiki. |
| FU16 | Dodaj produkt | Pracownik może dodać nowy produkt. |
| FU17 | Modyfikuj produkt | Pracownik może modyfikować istniejący produkt. |
| FU18 | Usuń produkt | Pracownik może usunąć produkt. |
| FU19 | Zrealizuj zamówienie | Pracownik może zrealizować zamówienie, jednocześnie zmieniając jego status. |
| FU20 | Oblicz przychody | Pracownik ma podgląd na przychody w danym miesiącu. |
| FU21 | Przeglądaj magazyn | Pracownik ma wgląd w liczność produktów. |

Na rysunku 1 przedstawiony został diagram przypadków użycia. Na diagramie przedstawiono trzech aktorów oraz dostepne im przypadki użycia. System aplikacja webowa jest podzielony na trzy części: Wiki, Sklep, Logowanie do systemu. Pakiet Wiki przedstawia przypadki użycia, jakich użytkownicy mogą dokonywać w części informacyjnej aplikacji. Pakiet Sklep pokazuje przypadki użycia, jakich użytkownicy są w stanie dokonywać w części transakcyjnej aplikacji.



Rysunek . Diagram przypadków użycia

* 1. Wymagania niefunkcjonalne

Tabela . Wymagania niefunkcjonalne aplikacji

|  |  |  |
| --- | --- | --- |
| **Wymagania niefunkcjonalne** | | |
| **Id** | **Nazwa** | **Opis** |
| NFU01 | Obsługa wielu użytkowników | Aplikacja powinna być w stanie obsłużyć jednocześnie co najmniej 50 użytkowników. |
| NFU02 | Przejrzysty interfejs użytkownika | Aplikacja powinna posiadać jednolity, intuicyjny interfejs użytkownika. |
| NFU03 | Aplikacja powinna być dostępna z poziomu przeglądarki | Aplikacja powinna wspierać najpopularniejsze przeglądarki, mobilne i desktopowe. |
| NFU04 | Aplikacja powinna być rozszerzalna | Implementacja aplikacji powinna pozwalać na dodawanie nowych funkcji. |
| NFU05 | Użytkownik nie łączy się bezpośrednio z bazą danych | Do bazy danych aplikacji dostęp ma wyłącznie serwer, użytkownik aplikacji nie łączy się bezpośrednio z bazą danych. |
| NFU06 | Serwer wymaga stałego połączenia z Internetem | Aby aplikacja była dostępna w Internecie serwer musi być połączony z Internetem, dodatkowo na routerze trzeba ustawić przekierowanie odpowiednich portów. |
| NFU07 | Użytkownik aplikacji musi posiadać połączenie z Internetem | Aby aplikacja działa poprawnie użytkownik musi być połączony z Internetem lub znajdować się w tej samej sieci co serwer aplikacji. |
| NFU08 | Aplikacja powinna być odporna na ataki | Warstwa transakcyjna aplikacji powinna być odporna na wszelkiego rodzaju niepoprawne zapytania. |
| NFU09 | Obsługa JavaScriptu | Przeglądarka użytkownika musi mieć uruchomioną obsługę JavaScriptu. |

* 1. Przyjęte założenia projektowe

Tworzona aplikacja webowa będzie korzystała wyłącznie z jednego egzemplarza bazy danych oraz pojedynczego serwera. Tylko serwer aplikacji będzie miał dostęp do bazy danych. Dodatkowo na serwerze będą zapisywane zdjęcia, wrzucane przez pracownika kwiaciarni poprzez aplikację. Połączenie pomiędzy Front-Endem, a Back-Endem jest realizowane na podstawie REST API[[5]](#footnote-5). Aplikacja nie przewiduje obsługi płatności. Interfejs użytkownika aplikacji jest wyświetlany w języku angielskim. Na serwerze nie mogą być uruchomione żadne inne usługi korzystające z portów 3000 oraz 8080. Aplikacja jest responsywna, oznacza to, że powinna wyglądać dobrze w przeglądarce na ekranie komputera i telefonu.

1. Wykorzystane technologie i narzędzia programistyczne

Ten rozdział pokazuje oraz opisuje technologie i narzędzia, które były użyte podczas tworzenia aplikacji.

* 1. JDK

JDK jest paczką darmowego oprogramowania firmy Sun Microsytems (obecnie firma należy do Oracle Corporation), w której skład wchodzą trzy elementy: JVM, JRE, narzędzia programistyczne Javy (kompilator, debugger, generator dokumentacji itd.). Produkt ten jest skierowany głównie do programistów tworzących oprogramowanie w języku Java[1]. Java korzysta z maszyny wirtualnej, oznacza to, że oprogramowanie napisane w tym języku może zostać uruchomione na każdym urządzeniu wspierającym maszynę wirtualną, bez potrzeby modyfikowania, ani ponownej kompilacji kodu. W tworzeniu aplikacji została użyta wersja JDK 1.8.

* 1. Eclipse

Jest to darmowe IDE firmy Eclipse Foundation dla profesjonalnych deweloperów tworzących oprogramowanie. Zintegrowane środowiska deweloperskie są świetnym narzędziem w kontekście wytwarzania oprogramowania. Zawierają one w sobie środowiska uruchomieniowe, edytory kodu oraz podpowiadanie składni. W znaczny sposób przyspieszają oraz ułatwiają pracę programisty. Dla platformy Java Eclipse[2] wydawany jest w dwóch wersjach: standardowej oraz EE. Wersja EE zawiera w sobie kilka dodatków takich jak: integracja z gitem, obsługa Mavena, edytory XML, JPA i wiele innych.

* 1. Spring

Spring Framework[3][4] wydany po raz pierwszy w roku 2002 miał na celu eliminować wiele problemów związanych z EJB, takich jak: narzucony model programowania, duży nakład kodu do osiągnięcia niewielkiego efektu, korzystanie tylko z niewielkiej części EJB. Spring nie narzuca żadnego modelu programowania. Głównymi cechami tego frameworka są inwersja kontroli (IoC) poprzez wstrzykiwanie zależności i programowanie aspektowe (AOP). Cechy te w znaczący sposób wpływają na przejrzystość tworzonego kodu oraz pozwalają go znacznie zredukować. Sam Spring możemy podzielić na:

* Dostęp do danych i integracja
* Usługi sieciowe i zdalne
* AOP
* Instrumentalizacja
* Podstawowy kontener Springa
* Testowanie

W dzisiejszych czasach Spring jest technologią bardzo popularną wśród wielu programistów.

* 1. Spring Boot

Spring Boot[5] jest paczką konfiguracji/oprogramowania, rodzajem udogodnienia bazującym na springu, wydawany jest on w oddzielnych wersjach niż spring. Łączy on w sobie kilka elementów:

* Wbudowany serwer – Spring Boot posiada wbudowany serwer (domyślnie Tomcat), na którym aplikacja jest automatycznie uruchamiana, nie wymaga to od programisty instalacji żadnego dodatkowego oprogramowania do uruchomienia aplikacji.
* Automatyczną konfigurację – Do uruchomienia aplikacji nie jest potrzebna żadna dodatkowa konfiguracją, nadal istnieje możliwość konfigurowania wielu elementów Springa.
* Szybkość – Spring Boot zawsze dostarcza kompatybilne ze sobą wersje Springa, pozwala to uniknąć problemów związanych z kompatybilnością pakietów. Dodatkowo w Spring Bootowych aplikacjach możemy znaleźć plik konfiguracyjny application.properties, w którym możemy modyfikować różne aspekty konfiguracji bez potrzeby używania XMLa.
  1. Maven

Apache Maven[6] jest narzędziem ułatwiającym tworzenie projektów w Javie. Udostępnia on jednolity system budowy projektów i narzuca dobre praktyki programowania. Maven posiada możliwości automatyzacji tworzenia oprogramowania. Budowa aplikacji jest oparta na informacjach zawartych w pliku POM.xml. Plik POM.xml zawiera:

* Informacje o projekcie – nazwa, id grupy, id projektu, wersję projektu, URL projektu
* Zależności – wymagane pliki jar, które możemy pobrać z repozytorium
* Pluginy – wtyczki np. failsave (uruchamia testy JUnit), install(instaluje zależności w lokalnym repozytorium), compiler (kompiluje pliki Javy), których czynności zostają wykonane podczas budowy projektu
  1. Git

Git[7] jest systemem kontroli wersji, służy on głównie do przechowywania oraz synchronizowania projektów. Git przechowuje projekty w centralnych repozytoriach, mogą one być lokalne bądź zdalne. Praca w gitcie jest oparta na komitach. Komit to stan plików(ich różnica w bajtach w stosunku do poprzedniego komita) wraz z komentarzem. Git pozwala nam na odtworzenie stanu naszego projektu do danego komita. Git składa się z gałęzi, początkową i główną jest gałąź master. Od istniejących gałęzi możemy dodawać dowolne nowe gałęzie i do nich przypisywać nasze komity. Gałęzie możemy ze sobą scalać łącząc ich zawartość.

* 1. Hibernate

Hibernate[8] jest frameworkiem pozwalającym na automatyczne mapowanie obiektów Javy na struktury danych bazy danych i struktury danych na obiekty. Hibernate spełnia specyfikację JPA. Posiada on wbudowaną obsługę wyjątków bazy danych. Dodatkowo Hibernate implementuje zabezpieczenia przed popularnymi atakami na bazy danych. Programista dzięki technologi Hibernate nie musi pisać zapytań SQL do bazy danych, wciąż jednak ma taką możliwość. Framework ten jest w stanie połączyć się z wieloma popularnymi serwerami baz danych.

* 1. MySQL

Oprogramowanie MySQL udostępnia wielowątkowy, niezawodny, wieloużytkownikowy serwer SQL. Serwer jest zaprojektowany do obsługi dużej ilości zapytań. Nadaje się do użycia w małych i dużych aplikacjach.

* 1. NPM

NPM (Node Package Manager) to zbiór narzędzi przydatnych w tworzeniu aplikacji w środowisku JavaScript. NPM w swojej zasadzie działania jest bardzo podobny do Mavena. Plikiem z informacjami o projekcie jest package.json, którego zawartość musi być napisana w konwencji JSON. Repozytorium NPM zawiera ponad 800,000 paczek z kodem.

* 1. React.js

React.js[9] biblioteka do języka JavaScript stworzona przez programistów Facebooka. Główną cechą React jest to, że korzysta z wirtualnego obiektowego modelu dokumentu[[6]](#footnote-6) (DOM). Zapisuje on wszystkie dane w pamięci przeglądarki, co pozwala na wydajne odświeżanie i modyfikowanie informacji. Technologia ta pozwala programiście pisać kod tak jakby cała strona się zmieniała, gdzie w rzeczywistości tylko zmiany są wyświetlane. React używa języka JSX, który pozwala na pisanie kodu HTML bezpośrednio w kodzie, zamiast jako ciągu znaków. Kod Reacta składa się z elementów nazwanych komponentami. Komponenty mogą być renderowane jako konkretny element(węzeł) DOM. Komponenty tworzą strukturę drzewiastą, gdzie w korzeniu znajduje się pojedynczy element. Komponenty mogą przekazywać swoim dzieciom parametry nazwane „props”, które mogą wpływać na ich stan oraz wyświetlaną zawartość.

Do głównych zalet Reacta należą:

* Jednolita struktura
* Duże wsparcie ze strony deweloperów oprogramowania
* Możliwość ponownego używania komponentów
* Wydajność

Do wad Reacta należą:

* Duże tempo wprowadzania nowych zmian
* Słaba dokumentacja
* Niemożliwość przekazywania stanu komponentów do ich rodziców
  1. Visual Studio Code

Visual Studio Code to IDE firmy Microsoft. IDE to nie zostało stworzone pod kątem konkretnego języka programowania, jednak świetnie się nadaje do edycji kodu JavaScriptu. Pluginy dostępne do Visual Studio Code pomagają programiście zachować czystość kodu i konwencje dzięki automatycznemu formatowaniu składni.

* 1. Redux

Redux[10] jest kontenerem stanu dla aplikacji napisanych w języku JavaScript. Pozwala na wyeliminowanie jednej z wad React.js, jaką jest niemożliwość przekazywania stanu do komponentu rodzica. Redux przechowuje stan w pojedynczym kontenerze nazwanym „store”. Stan możemy zmienić jedynie poprzez wywołanie akcji, w których zawarta jest informacja o zmianie. Akcje są przechwytywane przez reduktory, ich zawartość jest analizowana, a stan aplikacji jest zmieniany w zależności od danych zawartych w akcji.

* 1. Axios

Axios[11] jest biblioteką do JavaScriptu, która udostępnia klienta HTTP opartego na promisach[[7]](#footnote-7). Pozwala on w szybki sposób tworzyć zapytania HTTP, automatycznie formatuje dane na format JSON oraz zapewnia ochronę przed XSRF po stronie klienta.

* 1. Material-UI

Material-UI[12] to biblioteka darmowych komponentów do React.js. Komponenty te posiadają jednolity styl i śledzą nowoczesne trendy wyglądu interfejsu użytkownika. Z tego produktu korzystają znane firmy/organizacje: Nasa[[8]](#footnote-8), Capgemini[[9]](#footnote-9), Shutterstock[[10]](#footnote-10) i wiele innych.

* 1. MDBReact

MDBReact[13] podobnie do Material-UI jest biblioteką komponentów do React.js. Są to głównie proste elementu interfejsu użytkownika, takie jak przyciski, suwaki, nagłówki itp.

* 1. Pozostałe
* Xamp – dystrybucja Apache zawierająca serwer MySQL i graficzny edytor bazy
* Visio – program do tworzenia diagramów
* Visual paradigm – program do tworzenia diagramów
* Postman – program pozwalający wysyłać zapytania HTTP z dowolnie zmodyfikowanymi danymi
* Opera – przeglądarka internetowa
* Word – edytor tekstu
* Selenium – Wtyczka przeglądarkowa służąca do przeprowadzenia testów funkcjonalnych
* JMeter – narzędzie służące do przeprowadzania testów wydajnościowych aplikacji
* Lighthouse – automatyczne narzędzie do badania jakości stron internetowych

1. Projekt systemu

Rozdział ten przedstawia architekturę aplikacji wraz z projektem bazy danych.

* 1. Architektura aplikacji

Aplikacja została stworzona w trójwarstwowym modelu. Model został pokazany na rysunku 2. Poszczególne warstwy modelu to:

* Warstwa prezentacji (Front-End) – Najwyższa warstwa aplikacji. Warstwa prezentacji jest jedyną, do której użytkownik ma bezpośredni dostęp. Wyświetla ona informacje np. zawartość koszyka, dostępne produkty itd. Akcje dokonane przez użytkownika są przechwytywane i analizowane. Te związane z logiką biznesową są przesyłane do warstwy logiki biznesowej, gdzie są wykonywane, a odpowiedź jest wysyłana do wyższego poziomu. W przypadku tej aplikacji komunikacja pomiędzy warstwami odbywa się za pomocą REST API w formacie JSON, a funkcję wyświetlania zawartości i generowania akcji pełni przeglądarka internetowa.
* Warstwa logiki biznesowej (Back-End) – Warstwa ta jest odpowiedzialna za przetwarzanie akcji użytkowników, sprawdzanie ich poprawności i spójności z przechowanymi danymi oraz generowanie odpowiedzi. Back-End udostępnia punkty kończynowe (ang. Endpoint), na które możemy wysyłać zapytania HTTP. Dodatkowo Back-End odpowiedzialny jest za połączenie z bazą danych, które odbywa się za pomocą technologii Hibernate.
* Warstwa danych (Baza danych) – Warstwa danych jest odpowiedzialna za przechowywanie danych. W projekcie wybraną bazą danych jest serwer MySQL.

Dzięki przyjętej architekturze system jest podzielony na trzy niezależne od siebie elementy. Zmiana implementacji jednego z nich, nie wpływa na działanie innego elementu. Części te, łączą się ze sobą za pomocą protokołów sieciowych lub użytych technologii. Pomaga to osiągnąć rozszerzalność i skalowalność aplikacji. Dodatkowym atutem jest możliwość podziału pracy na fragmenty, z których każdy może być wykonywany jednocześnie. Wadą tej architektury jest nakład pracy potrzebny do wykonania trzech elementów. Dodatkową wadą jest propagacja błędów występujących w fazie projektowania, jednak często jest ona ograniczona tylko do poszczególnej warstwy.

Obraz zawierający mapa, zrzut ekranu

Opis wygenerowany automatycznie

Rysunek . Model trójwarstwowy aplikacji

* 1. Projekt bazy danych

Baza danych została zaprojektowana w oparciu o wymagania funkcjonalne i niefunkcjonalne postawione tworzonej aplikacji.

* + 1. Model konceptualny

Model konceptualny pokazuje encje, które stworzone zostały na podstawie opisów obiektów świata rzeczywistego oraz relacji występujących pomiędzy nimi. Model konceptualny posłuży jako baza do stworzenia modelu fizycznego bazy danych. W tabeli 3 przedstawiony został opis encji na podstawie elementów świata rzeczywistego. W tabeli 4 pokazany został opis relacji pomiędzy encjami.

Tabela . Opis encji na podstawie elementów świata rzeczywistego

|  |  |
| --- | --- |
| **Opis encji** | |
| **Obiekt świata rzeczywistego** | **Opis** |
| Użytkownik | Zarejestrowana osoba lub pracownik. Dodatkowo dla gościa, który dokona zakupu zostanie utworzone konto, na które nie będzie się jednak można zalogować. Użytkownik posiada takie informacje jak: imię, nazwisko, numer telefonu, email, hasło, rola oraz informacje, czy dane konto jest aktywne. Użytkownik połączony jest też z adresem, który charakteryzujemy jako inny obiekt. |
| Adres | Niezbędne informacje na temat lokacji wymagane przy dostawie. Składa się z takich informacji jak: kraj, miasto, ulica, numer lokalu, oraz kod pocztowy. |
| Produkt | Produkt przeznaczony do sprzedaży w kwiaciarni, zawiera informacje o cenie, opisie, typie produktu. Produkt może być połączony z wpisem na wiki. Pracownik jest w stanie dodawać, usuwać oraz modyfikować produkty. |
| Magazyn | Magazyn zawiera produkt wraz z informacją o jego ilości. Pracownik jest w stanie modyfikować stan danego magazynu. Może on też dodawać nowe magazyny dla produktów, które jeszcze nie posiadają magazynu. |
| Zdjęcie | To plik w formacie PNG oraz dodatkowe informacje, takie jak opis i typ. Pracownik potrafi dodawać zdjęcia oraz łączyć je z wpisami na wiki i produktami. |
| Wpis wiki | Wpis wiki jest to zbiór informacji na temat danego kwiatka. Pracownik jest w stanie dodawać modyfikować i usuwać wpisy na wiki. Użytkownik na podstawie nazwy pospolitej, łacińskiej lub zdjęcia jest w stanie zidentyfikować daną roślinę, a następnie uzyskać szczegółowe informacje na temat jej budowy oraz pielęgnacji. |
| Typ dostawy | Określa typ dostawy, jej koszt oraz metodę dostarczenia. Klient podczas finalizowania zamówienia będzie w stanie wybierać spośród wielu metod dostawy. |
| Zamówienie | Transakcja dokonana przez klienta. Zawiera wszystkie potrzebne informacje na temat dostawy oraz kupionych produktów. Pracownik kwiaciarni ma dostęp do listy zamówień, może on realizować zamówienia przygotowując towar i zmieniając ich status. |
| Koszyk | Koszyk zawiera produkty jakie użytkownik do niego dodał. Pozwala on zalogowanemu użytkownikowi na zapamiętywanie produktów dodanych do koszyka na serwerze, co zapewnia mu możliwość szybkiego powrotu do zakupów. |
| Miesięczny status | Aktualizowany co miesiąc status, który zawiera ilość sprzedanych produktów i zysk wynikający ze sprzedaży w danym miesiącu. Status ten jest dostępny wyłącznie dla zalogowanego pracownika kwiaciarni. |

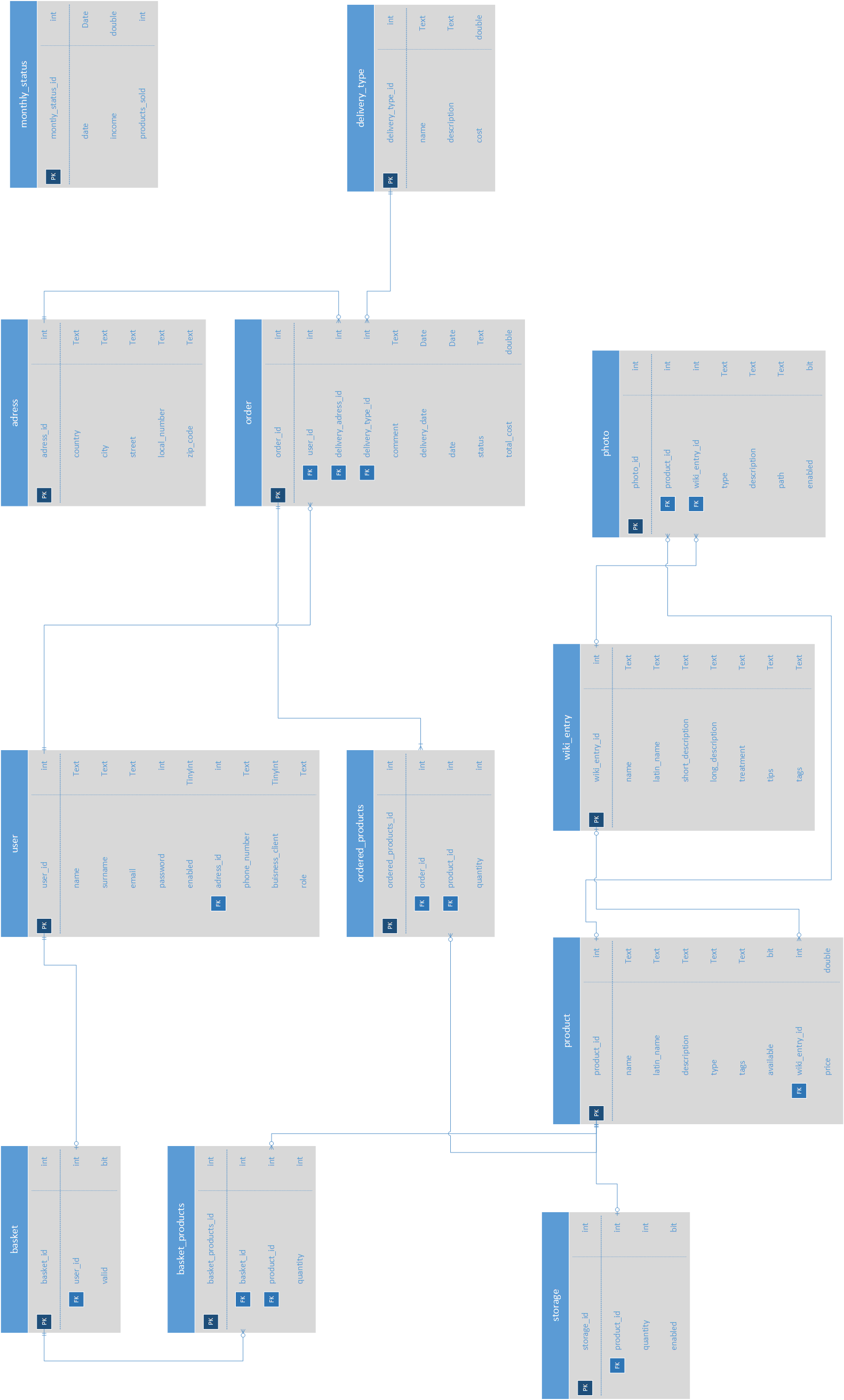
Tabela . Opis relacji pomiędzy encjami

|  |  |
| --- | --- |
| **Opis relacji** | |
| **Relacja pomiędzy encjami** | **Opis** |
| Użytkownik - Koszyk | Użytkownik może posiadać tylko jeden koszyk, koszyk musi być przypisany wyłącznie do jednego użytkownika |
| Użytkownik – Adres | Użytkownik musi posiadać wyłącznie jeden adres, jeden adres może być przypisany wielu użytkowników |
| Użytkownik - Zamówienie | Użytkownik może posiadać wiele zamówień, jedno zamówienie musi być przypisany do jednego użytkownika |
| Koszyk - Produkt | Koszyk może zawierać wiele produktów, produkt może być przypisany do wielu koszyków |
| Typ dostawy - Zamówienie | Typ dostawy może być przypisany do wielu zamówień, zamówienie musi posiadać jeden typ dostawy |
| Produkt - Magazyn | Produkt może być przypisany do jednego magazynu, magazyn musi posiadać tylko jeden produkt |
| Produkt - Zdjęcie | Produkt może posiadać wiele zdjęć, zdjęcie może być przypisane tylko do jednego produktu |
| Produkt – Wpis wiki | Produkt może posiadać tylko jeden wpis wiki, wpis wiki może być przypisany tylko do jednego produktu |
| Wpis wiki - Zdjęcie | Wpis wiki może posiadać wiele zdjęć, zdjęcie może być przypisany tylko do jednego wpisu wiki |
| Zamówienie - Produkt | Zamówienie musi posiadać co najmniej jeden produkt, produkt może być przypisany do wielu zamówień |
| Zamówienie – Adres | Zamówienie musi posiadać jeden adres, adres może być przypisany do wielu zamówień |

* + 1. Model fizyczny z ograniczeniami integralności danych

W tworzonej aplikacji model logiczny jest równoważny z modelem fizycznym. Model fizyczny powstał na podstawie modelu konceptualnego wraz z założonymi relacjami pomiędzy encjami. Jest to najważniejszy model z punktu widzenia programisty, ponieważ zawiera on typy oraz rzeczywiste nazwy atrybutów. Rysunek 3 przedstawia model fizyczny bazy danych zaimplementowany w aplikacji. Wszystkie tabele przedstawione są w trzeciej postaci normalnej oprócz tabeli *adress*, zrobiono to w celu osiągnięcia lepszej wydajności zapytań, poprzez zmniejszenie ilości kosztownych łączeń.

Dodatkowo do schematu bazy danych zostało dodane wydarzenie, które raz w miesiącu wstawia nowy rekord do tabeli *montly\_status*. Kod wydarzenia znajduje się na listingu 1.



Rysunek . Model fizyczny bazy danych

Listing . Kod wydarzenia na tabeli montly\_status

CREATE EVENT e\_monthly

ON SCHEDULE

EVERY 1 MONTH

COMMENT 'Create new month evaluation.'

DO

INSERT INTO `monthly\_status`( `date`, `income`, `products\_sold`) VALUES (CURRENT\_DATE,'0','0')

1. Implementacja systemu

Rozdział ten opisuje szczegóły implementacji systemu oraz zawiera prezentację interfejsu graficznego użytkownika.

* 1. Back-End

Cześć Back-Endowa aplikacji opiera się na stworzeniu oprogramowania uruchomianego jako serwer, którego zadaniem jest obsługiwanie zapytań wysyłanych z warstwy prezentacji. W tym celu wykorzystano Spring Boota w wersji 2.1.9, zawierał on w sobie zależności do Springa w wersji 5. Back-End realizował również połączenie z bazą danych poprzez technologię Hibernate.

* + 1. Zależności w pliku POM.xml

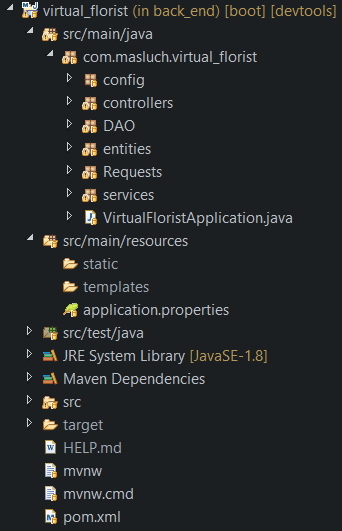
Na listingu 2 przedstawione zostały artefakty zależności używanych w projekcie. Maven na podstawie tych zależności pobiera odpowiednie pliki. Spring Boot daje nam pewność, że wszystkie pobrane pliki będą ze sobą kompatybilne. Uwagę należy też zwrócić na artefakt mysql-connector-java. Jest to implementacja konektora wykorzystywanego przez Hibernate w połączeniu z serwerem bazy danych MySQL.

Listing 2. Wycinek pliku POM.xml

<artifactId>spring-boot-starter-actuator</artifactId>  
<artifactId>spring-boot-starter-data-jpa</artifactId>  
<artifactId>spring-boot-starter-security</artifactId>  
<artifactId>spring-boot-starter-web</artifactId>  
<artifactId>mysql-connector-java</artifactId>  
<artifactId>spring-boot-devtools</artifactId>  
<artifactId>spring-boot-starter-test</artifactId>  
<artifactId>spring-security-test</artifactId>

* + 1. Struktura projektu

Na rysunku 4 przedstawiona została struktura projektu części Back-Endowej. W pakiecie *com.masluch.virtual\_florist* znajduje się sześć pakietów i klasa *VirtualFloristApplication.java*. Pakiet *config* zawiera klasy konfiguracyjne mechanizmów Springa dotyczących bezpieczeństwa. Pakiet *entities* zawiera klasy wraz z definicją mapowania, wykorzystywane przez framework Hibernate do automatycznego mapowania obiektów. Pakiet *DAO* zawiera interfejsy obiektów dostępu do danych oraz ich implementacje. Pakiet *services* zawiera interfejsy, wraz z implementacją serwisów. Pakiet *controllers* zawiera klasy kontrolerów.



Rysunek . Struktura projektu części Back-End

* + 1. Encje

Na listingu 3 przedstawiono fragment klasy encji *Product*. Wszystkie klasy zdefiniowane w pakiecie *entity* zawierają adnotacje JPA, co pozwala im być mapowane przez framework Hibernate. Za pomocą tych adnotacji Hibernate jest w stanie konwertować klasy Javy na struktury danych w bazie danych i na odwrót. Jest to mechanizm utrzymania spójności danych wykorzystany na Back-Endzie.

Wykorzystane adnotacje to:

@Entity – adnotacja wykorzystywana przez Hibernate w celu rozpoznania encji.

@Id – klucz prywatny.

@GeneratedValue – strategia generowania klucza prywatnego, w tym przypadku wykorzystano strategię użytą w schemacie bazy danych do generacji klucza prywatnego.

@Column – definiuje mapowanie kolumny w bazie na zmienną.

@OneToMany – definiuje typ relacji, kaskada oznacza jak niektóre operacje mają być kaskadowane na elementy złączone, w tym przypadku wszystkie akcje mają być kaskadowane, oprócz operacji usuwania.

@JoinColumn – definiuje klucz obcy, na podstawie którego odbywa się łączenie tabel.

Listing 3. Fragment klasy encji *Product*

@Entity  
**public class** Product  
{  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 @Column(name = **"product\_id"**)  
 **private int productId**;  
   
 @Column(name = **"price"**)  
 **private** Double **price**;  
   
 @Column(name = **"name"**)  
 **private** String **name**;  
   
 @Column(name = **"latin\_name"**)  
 **private** String **latinName**;  
   
 @Column(name = **"description"**)  
 **private** String **description**;  
   
 @Column(name = **"type"**)  
 **private** String **type**;  
   
 @Column(name = **"tags"**)  
 **private** String **tags**;  
   
 @Column(name = **"available"**, columnDefinition = **"BOOLEAN"**)  
 **private boolean available**;  
   
 @OneToOne(optional = **true**, cascade = {CascadeType.DETACH, CascadeType.MERGE, CascadeType.PERSIST, CascadeType.REFRESH})  
 @JoinColumn(name = **"wiki\_entry\_id"**)  
 **private** WikiEntry **wikiEntry**;  
   
 @OneToMany(fetch = FetchType.LAZY, cascade = {CascadeType.DETACH, CascadeType.MERGE, CascadeType.PERSIST, CascadeType.REFRESH})  
 @JoinColumn(name = **"product\_id"**)  
 **private** List<Photo> **photos**;  
}

* + 1. DAO

DAO jest wzorcem programowania, który poprzez wykorzystanie abstrakcji pozwala oddzielić część transakcyjną od części spójności danych. W tym celu definiowany jest interfejs oraz jego implementacja. Interfejs i implementacja zostały przedstawione na listingu 4. Możemy zauważyć, wykorzystanie adnotacji **@Autowired**. Oznacza ona, że pozwalamy mechanizmom Springa na automatyczne wiązanie ziarenek. W tym przypadku Spring będzie poszukiwał ziarenka o nazwie *EntityManager* i automatycznie połączy je ze *zmienną entityManager*. Kod pokazuje wykorzystanie wbudowanych klas technologii Hibernate, aby zarządzać danymi.

Listing 4. Interfejs ProductDAO oraz jego implementacja

**public interface** ProductDAO  
{  
 **public** List<Product> findAll();  
 **public** Product findById(**int** productId);  
 **public** List<Product> findProducts(**int** numOfProducts);  
 **public** Product save(Product product);  
 **public void** update(Product product);  
 **public void** deleteById(**int** productId);  
 **public** Product findByWikiEntry(WikiEntry wikiEntry);  
}  
  
@Repository  
**public class** ProductDAOImpl **implements** ProductDAO  
{  
 @Autowired  
 **private** EntityManager **entityManager**;  
   
 @Override  
 **public** List<Product> findAll()  
 {  
 Session session = **entityManager**.unwrap(Session.**class**);  
   
 Query<Product> query = session.createQuery(**"FROM Product ORDER BY name"**, Product.**class**);  
 List<Product> result = query.getResultList();  
 **return** result;  
 }  
   
 @Override  
 **public** Product findById(**int** productId)  
 {  
 Session session = **entityManager**.unwrap(Session.**class**);  
 Product product = session.get(Product.**class**, productId);  
 **return** product;  
 }  
   
 @Override  
 **public** List<Product> findProducts(**int** numOfProducts)  
 {  
 Session session = **entityManager**.unwrap(Session.**class**);  
 Query<Product> query = session.createQuery(**"FROM Product p LIMIT :numOfProducts"**, Product.**class**);  
 query.setParameter(**"numOfProducts"**, numOfProducts);  
 List<Product> productList = query.getResultList();  
 **return** productList;  
 }  
   
 @Override  
 **public** Product save(Product product)  
 {  
 Session session = **entityManager**.unwrap(Session.**class**);  
 session.save(product);  
 **return** product;  
 }  
   
 @Override  
 **public void** update(Product product)  
 {  
 Session session = **entityManager**.unwrap(Session.**class**);  
 session.update(product);  
 }  
   
 @Override  
 **public void** deleteById(**int** productId)  
 {  
 Session session = **entityManager**.unwrap(Session.**class**);  
 Product product = session.get(Product.**class**, productId);  
 session.delete(product);  
 }  
   
 @Override  
 **public** Product findByWikiEntry(WikiEntry wikiEntry)  
 {  
 Session session = **entityManager**.unwrap(Session.**class**);  
 Query<Product> query = session.createQuery(**"FROM Product p WHERE p.wikiEntry=:wikiEntry"**, Product.**class**);  
 query.setParameter(**"wikiEntry"**, wikiEntry);  
 List<Product> productsList = query.getResultList();  
 **if** (productsList.isEmpty())  
 **return null**;  
 **return** productsList.get(0);  
 }  
}

* + 1. Serwisy

Na listingu 5 przedstawiono interfejs *ProductService* oraz fragment jego implementacji. W serwisie wykorzystano wzorzec projektowy fasada, w celu ujednolicenia dostępu do implementacji. Wszystkie serwisy w pakiecie *services* korzystają z tego wzorca. Serwis pełni funkcje logiki biznesowej, przyjmuje on dane od kontrolera, a następnie wykonuje na nich akcje. Możemy zauważyć, że większość metod generuje wartość zwrotną w postaci *ResponseEntity*, jest to forma odpowiedzi serwera na zapytanie wysłane z warstwy prezentacyjnej. W przypadku, gdy dane w zapytaniu są złe, odpowiedz posiada kod 400 oznaczający złe zapytanie. Odpowiedzi na poprawne zapytania posiadają status 200. Dodatkowo metody są opatrzone adnotacją ***@Transactional***, co oznacza, że operacje dokonywane na bazie w danej funkcji są wewnątrz transakcji.

Listing 5. Interfejs *ProductService* oraz fragment jego implementacji

**public interface** ProductService  
{  
 **public** List<Product> findAll();  
 **public** Product findById(**int** productId);  
 **public** Product save(Product product);  
 **public void** update(Product product);  
 **public** ResponseEntity<String> deleteById(**int** productId);  
 **public** ResponseEntity<Product> addNewProduct(Product newProduct);  
 **public** ResponseEntity<Product> addNewProductWithWiki(Product newProduct, String wikiEntryId);  
 **public** ResponseEntity<String> deleteProduct(String productId);  
 **public** ResponseEntity<Product> updateProduct(String productId, Product product, String wikiEntryId);  
}  
  
@Service  
**public class** ProductServiceImpl **implements** ProductService  
{  
   
 @Autowired  
 **private** ProductDAO **productDAO**;  
   
 @Autowired  
 **private** WikiEntryDAO **wikiEntryDAO**;  
   
 @Autowired  
 **private** PhotoDAO **photoDAO**;  
   
 @Override  
 @Transactional  
 **public** List<Product> findAll()  
 {  
 **return productDAO**.findAll();  
 }  
   
 @Override  
 @Transactional  
 **public** ResponseEntity<Product> addNewProduct(Product newProduct)  
 {  
 Product savedProduct = **productDAO**.save(newProduct);  
 ResponseEntity<Product> response = **new** ResponseEntity<Product>(savedProduct, HttpStatus.OK);  
 **return** response;  
 }  
   
 @Override  
 @Transactional  
 **public** ResponseEntity<Product> addNewProductWithWiki(Product newProduct, String wikiEntryId)  
 {  
 Integer entryId;  
 **try** {  
 entryId = Integer.decode(wikiEntryId);  
 }  
 **catch**(Exception ex) {  
 **return new** ResponseEntity<Product>(HttpStatus.BAD\_REQUEST);  
 }  
 **if**(entryId<1)  
 {  
 **return new** ResponseEntity<Product>(HttpStatus.BAD\_REQUEST);  
 }  
 WikiEntry wikiEntry = **wikiEntryDAO**.findById(entryId);  
 **if**(wikiEntry == **null**)  
 {  
 **return new** ResponseEntity<Product>(HttpStatus.BAD\_REQUEST);  
 }  
   
 newProduct.setWikiEntry(wikiEntry);  
 **productDAO**.save(newProduct);  
 **return new** ResponseEntity<Product>(newProduct, HttpStatus.OK);   
 }  
   
 @Override  
 @Transactional  
 **public** ResponseEntity<String> deleteProduct(String productId)  
 {  
 Integer id;  
 **try** {  
 id = Integer.decode(productId);  
 }  
 **catch**(Exception ex) {  
 **return new** ResponseEntity<String>(HttpStatus.BAD\_REQUEST);  
 }  
 **if**(id<1)  
 {  
 **return new** ResponseEntity<String>(HttpStatus.BAD\_REQUEST);  
 }  
   
 Product product = **productDAO**.findById(id);  
 **if**(product == **null**)  
 {  
 **return new** ResponseEntity<String>(HttpStatus.BAD\_REQUEST);  
 }  
   
 List<Photo> photosList = product.getPhotos();  
 **for**(Photo photo: photosList)  
 {  
 photo.setProductId(**null**);  
 **photoDAO**.update(photo);  
 }  
   
 **productDAO**.deleteById(product.getProductId());  
   
 **return new** ResponseEntity<String>(HttpStatus.OK);  
 }  
   
 @Override  
 @Transactional  
 **public** ResponseEntity<Product> updateProduct(String productId, Product product, String wikiEntryId)  
 {  
   
 Integer id = **null**;  
 Integer wikiId = **null**;  
 **try** {  
 id = Integer.decode(productId);  
 **if**(wikiEntryId!=**null**)  
 wikiId = Integer.decode(wikiEntryId);  
 }  
 **catch**(Exception ex) {  
   
 **return new** ResponseEntity<Product>(HttpStatus.BAD\_REQUEST);  
 }  
   
 Product productToUpdate = **productDAO**.findById(id);  
 **if**(productToUpdate == **null**)  
 {   
 **return new** ResponseEntity<Product>(HttpStatus.BAD\_REQUEST);  
 }  
   
 **if**(product.getPrice()==**null** || product.getPrice()<0.0)  
 {  
 **return new** ResponseEntity<Product>(HttpStatus.BAD\_REQUEST);  
 }  
   
 productToUpdate.setPrice(product.getPrice());  
 productToUpdate.setName(product.getName());  
 productToUpdate.setLatinName(product.getLatinName());  
 productToUpdate.setDescription(product.getDescription());  
 productToUpdate.setType(product.getType());  
 productToUpdate.setTags(product.getTags());  
 productToUpdate.setAvailable(product.isAvailable());  
   
 **if**(wikiId != **null**)  
 {  
 WikiEntry wikiEntry = **wikiEntryDAO**.findById(wikiId);  
 productToUpdate.setWikiEntry(wikiEntry);  
 }  
 **else** {  
 productToUpdate.setWikiEntry(**null**);  
 }  
   
 **productDAO**.update(productToUpdate);  
   
 **return new** ResponseEntity<Product>(productToUpdate, HttpStatus.OK);  
 }

* + 1. Kontrolery

Listing 6 przedstawia kontroler *Product*. Kontrolery są odpowiedzialne za przyjmowanie oraz odpowiadanie na zapytania wysyłane z Front-Endu. Udostępniają one punkty końcowe, na które można kierować zapytania. Wszystkie metody HTTP są filtrowane pod kątem ścieżki oraz rodzaju metody. Spring udostępnia nam adnotacje, które pozwalają nam odwoływać się do struktury wysyłanych zapytań i ich zawartości. Tymi adnotacjami sa:

@RequetsBody – odnosi się do ciała zapytania.

@RequestParam – odnosi się do parametru zapytania.

@PathVariable – odnosi się do fragmentu ścieżki

Listing 6. Kontroler *Product*

@RestController  
@RequestMapping(**"/product"**)  
@CrossOrigin  
**public class** ProductController  
{  
 @Autowired  
 **private** ProductService **productService**;  
   
 @GetMapping(path = **"/"**)  
 **public** List<Product> getAllProducts()  
 {  
 **return productService**.findAll();  
 }  
   
 @PutMapping(path =**"/newProduct"**)  
 **public** ResponseEntity<Product> addNewProduct(@RequestBody Product newProduct,@RequestParam(name = **"wikiEntryId"**,required = **false**) String wikiEntryId )  
 {  
 **if**(wikiEntryId == **null**)  
 **return productService**.addNewProduct(newProduct);  
 **else  
 return productService**.addNewProductWithWiki(newProduct, wikiEntryId);  
   
 }  
   
 @PostMapping(path=**"/{id}"**)  
 **public** ResponseEntity<Product> updateProduct(@PathVariable(name= **"id"**) String productId, @RequestBody Product product,@RequestParam(name = **"wikiEntryId"**,required = **false**) String wikiEntryId )  
 {  
 **return productService**.updateProduct(productId, product, wikiEntryId);  
 }  
   
 @DeleteMapping(path=**"/{id}"**)  
 **public** ResponseEntity<String> deleteProduct(@PathVariable(name = **"id"**) String productId )  
 {  
 **return productService**.deleteProduct(productId);  
 }  
   
   
}

* 1. Front-End

Front-End jest warstwą prezentacji aplikacji. Musi on być w stanie wyświetlić treści dla użytkowników i obsłużyć ich akcje. W tym celu wykorzystano bibliotekę React.js w wersji 16. Front-End łączy się z Back-Endem za pomocą REST API. Za wysyłanie zapytań i przyjmowanie odpowiedzi odpowiedzialna jest biblioteka Axios.

* + 1. Zależności w pliku package.json

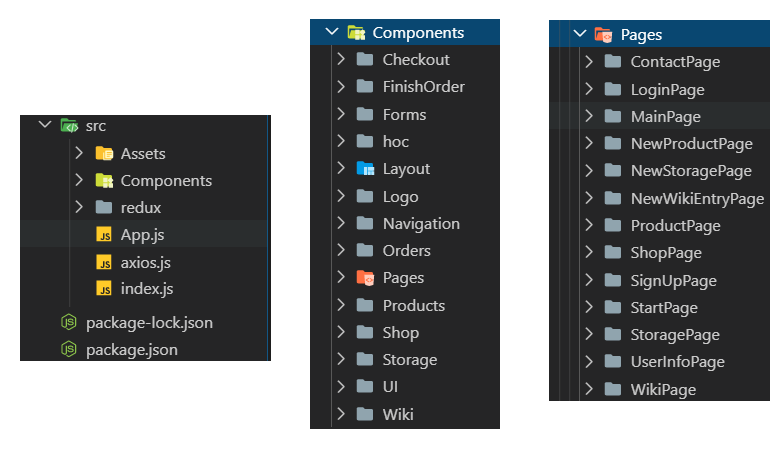
**Błąd! Nie można odnaleźć źródła odwołania.** jest fragmentem pliku packag.json. Zawiera on w sobie zależności jakie pobiera NPM. Obok zależności możemy zauważyć ich wymagane wersje. NPM na podstawie tych danych jest w stanie pobrać odpowiedni pliki z centralnego repozytorium.

Listing 7. Fragment pliku package.json

**"dependencies"**: {  
 **"@material-ui/core"**: **"^4.6.0"**,  
 **"@material-ui/icons"**: **"^4.5.1"**,  
 **"@material-ui/styles"**: **"^4.6.0"**,  
 **"@mdi/font"**: **"^4.5.95"**,  
 **"axios"**: **"^0.19.0"**,  
 **"bootstrap"**: **"^4.3.1"**,  
 **"mdbreact"**: **"^4.22.0"**,  
 **"react"**: **"^16.11.0"**,  
 **"react-bootstrap"**: **"^1.0.0-beta.14"**,  
 **"react-cookie-banner"**: **"^4.0.0"**,  
 **"react-dom"**: **"^16.11.0"**,  
 **"react-image-lightbox"**: **"^5.1.1"**,  
 **"react-lightbox-component"**: **"^1.2.1"**,  
 **"react-notifications-component"**: **"^2.2.3"**,  
 **"react-redux"**: **"^7.1.1"**,  
 **"react-router-dom"**: **"^5.1.2"**,  
 **"react-scripts"**: **"3.2.0"**,  
 **"react-select"**: **"^3.0.8"**,  
 **"react-swipeable-views"**: **"^0.13.3"**,  
 **"redux"**: **"^4.0.4"**,  
 **"redux-thunk"**: **"^2.3.0"** }

* + 1. Struktura

Na rysunku 5 przedstawiono strukturę projektu Front-Endu. Projekt jest podzielony na pliki konfiguracyjne, zasoby oraz komponenty Reacta. Korzenny komponent znajduje się w pliku *App.js*. W folderze *redux*, znajdują się reduktory i akcje. Katalog *Components* zawiera w sobie więcej katalogów, głównymi z nich są *UI*, *Layout* oraz *Pages*. Katalogi *UI* oraz *Layout* zawierają komponenty związane z układem i wyglądem aplikacji. Natomiast folder *Pages* zawiera w sobie widoki poszczególnych stron dostępnych w aplikacji. Zgodnie z filozofią Reacta wiele elementów było użyte kilkakrotnie.



Rysunek . Struktura Front-Endu

* + 1. Axios

Listing 8 pokazuje zawartość pliku *axios.js*. Kod ten jest odpowiedzialny za tworzenie instancji (obiektu) biblioteki Axios, który będzie pełnił funkcje tworzenia i wysyłania zapytań HTTP do Back-Endu.

Listing 8 Zawartość pliku *axios.js*

**import** axios from **"axios"**;  
**import** { config } from **"./config"**;  
   
 **const** instance = axios.create({  
 baseURL: config.serverURL  
 });  
export **default** instance;

* + 1. Redux

Folder redux zawiera w sobie reduktory i akcje. Akcje są odpowiedzialne za wysyłanie zapytań do Back-Endu oraz przetwarzanie odpowiedzi. Dodatkowo akcje wpływają na stan głównego kontenera aplikacji *store*. Na Listing 9 9 widzimy akcje *registerUser*, wysyła ona zapytanie HTTP metodą *PUT*, jako dane przesłany jest formularz, który funkcja otrzymuje poprzez argument. Po wykonaniu wyświetlona zostanie notyfikacja z informacją o powodzeniu, bądź porażce. Reduktory przechwytują akcje i odpowiednio zmieniają stan kontenera *store*. Na listingu 10 przedstawiony jest fragment reduktora *userReductor*, odpowiada on za przechwytywanie akcji *userActions*. Widzimy, że w zależności od akcji wywoływany jest różny przypadek, który modyfikuje kontener ze stanem.

Listing 9. Fragment pliku *UserActions.js*

**export const** registerUser = form => {  
 console.log(form);  
  
 **return** dispatch => {  
 axios  
 .put(**"/user/register"**, form)  
 .then(res => {  
 console.log(res);  
 store.addNotification({  
 ...notificationOk,  
 title: **"User"**,  
 message: **"Created account successfully"** });  
 })  
 .catch(err => {  
 console.log(err.response.data);  
 store.addNotification({  
 ...notificationError,  
 message: err.response.data  
 });  
 });  
 };  
};

Listing 10. Fragment pliku *userReducer.js*

**const** userReducer = (state = initialState, action) => {  
 **switch** (action.type) {  
 **case** actionTypes.SET\_USER: {  
 **return** {  
 ...state,  
 user: action.user  
 };  
 }  
 **case** actionTypes.LOGIN\_USER: {  
 **return** {  
 ...state,  
 user: { logged: **true** }  
 };  
 }  
  
 **case** actionTypes.LOG\_OUT: {  
 **return** {  
 ...state,  
 user: { ...initialState.user, logged: **false**, role: **"GUEST"** }  
 };  
 }  
 }  
  
 **return** state;  
};

* + 1. Pages

Folder *pages* zawiera widoki, dostępne w aplikacji. Przykładem takiego widoku może być ekran logowania. Na listingu widzimy komponent *LoginPage*, który odpowiedzialny jest za wyświetlanie formularza do logowania. Zmienna *state* przechowuje stan komponentu, a funkcja *render()* zwraca kod widoku formularza. Warto zwrócić uwagę na to, że tworzony komponent zwraca w sobie inne komponenty i przekazuje im różne parametry, jest to spójne z zasadami biblioteki React.

Listing 11. Fragment komponentu *LoginPage*

class LoginPage extends Component {  
state = {  
email: "",  
password: "",  
buttonEnabled: false  
};  
  
onEmailChangeHandler = event => {  
this.setState({ email: event.target.value });  
if (this.state.password !== "" && event.target.value !== "")  
this.setState({ buttonEnabled: true });  
else this.setState({ buttonEnabled: false });  
};  
  
onPasswordChangeHandler = event => {  
this.setState({ password: event.target.value });  
if (this.state.email !== "" && event.target.value !== "")  
this.setState({ buttonEnabled: true });  
else this.setState({ buttonEnabled: false });  
};  
  
onButtonClicked = () => {  
const form = {  
email: this.state.email,  
password: this.state.password  
};  
  
this.props.onUserLogin(form);  
this.props.history.push("/mainPage");  
};  
  
render() {  
{  
return (  
<div className={styleClass.All}>  
 <Container component="main" maxWidth="xs">  
 <div>  
 <Avatar>  
 <LockOutlinedIcon />  
 </Avatar>  
 <Typography component="h1" variant="h5">  
 Sign in  
 </Typography>  
  
 <TextField  
 variant="outlined"  
 margin="normal"  
 required  
 fullWidth  
 id="email"  
 label="Email Address"  
 name="email"  
 autoComplete="email"  
 autoFocus  
 onChange={event => this.onEmailChangeHandler(event)}  
 />  
 <TextField  
 variant="outlined"  
 margin="normal"  
 required  
 fullWidth  
 name="password"  
 label="Password"  
 type="password"  
 id="password"  
 autoComplete="current-password"  
 onChange={event => this.onPasswordChangeHandler(event)}  
 />  
 <FormControlLabel  
 control={<Checkbox value="remember" color="primary" />}  
 label="Remember me"  
 />  
 <Button  
 type="submit"  
 fullWidth  
 variant="contained"  
 color="primary"  
 disabled={!this.state.buttonEnabled}  
 onClick={this.onButtonClicked}  
 >  
 Sign In  
 </Button>  
 <Grid container>  
 <Grid item xs>  
 <Link href="/login" variant="body2">  
 Forgot password?  
 </Link>  
 </Grid>  
 <Grid item>  
 <NavLink to={"/signUp"} exact>  
 {"Don't have an account? Sign Up"}  
 </NavLink>  
 </Grid>  
 </Grid>  
 </div>  
 </Container>  
</div>  
);  
}  
}  
}  
  
export default connect(mapStateToProps, mapDispatchToProps)(LoginPage);

* + 1. UI

Katalog *UI* zawiera podstawowe elementy interfejsu użytkownika, takie jak: przyciski, pola tekstowe, nagłówki itp. Posiadają one jednolity styl. Dzięki temu, że w wielu miejscach aplikacji korzystamy z tej samej implementacji elementów interfejsu możemy łatwo zmienić styl naszej aplikacji bez potrzeby szukania poszczególnych odwołań do funkcji. Na listingu 12 możemy zobaczyć implementacje jednego z przycisków.

Listing 12. Komponent *ButtonPage*

import React, { Fragment } from "react";  
import { MDBBtn } from "mdbreact";  
  
const ButtonPage = props => {  
 return (  
 <Fragment>  
 <MDBBtn  
 disabled={props.isDisabled}  
 color="primary"  
 rounded  
 onClick={props.onClickAction}  
 >  
 {props.name}  
 {props.children}  
 </MDBBtn>  
 </Fragment>  
 );  
};  
  
export default ButtonPage;

* + 1. App

Plik App.js zawiera komponent korzeń. Na listingu 13 znajduje się implementacja tego komponentu. Warto zwrócić uwagę na to, że komponent Layoutokala inne komponenty,a w jego wnętrzu renderowane są poszczególne widoki stron. Komponenty te zostaną wyświetlone w momencie kiedy użytkownik przejdzie, lub zostanie przekierowany na ścieżkę podaną przez zmienną path.

Listing 13. Fragment kodu komponentu App

class App extends Component {  
 componentWillMount() {  
 this.props.onProductFetch();  
 this.props.onStorageFetch();  
 this.props.onWikiEntriesFetch();  
 }  
  
 render() {  
 return (  
 <div>  
 <Layout>  
 <Switch>  
 <Route path="/mainPage" component={MainPage} />  
 <Route exact path="/wiki" component={Wiki} />  
 <Route exact path="/wiki/:id" component={WikiPage} />  
 <Route exact path="/login" component={LoginPage} />  
 <Route exact path="/signUp" component={SignUpPage} />

…  
 </Switch>  
 </Layout>  
 </div>  
 );  
 }  
}

…  
  
export default connect(mapStateToProps, mapDispatchToProps)(App);

* 1. Projekt wybranych funkcji
     1. Funkcja kontrolera odpowiedzialna za zapisywanie zdjęć

Na listingu 14 został przedstawiony fragment funkcji odpowiedzialnej za zapisywanie zdjęć przesłanych z Front-Endu. Funkcja ta przyjmuje zdjęcie jako wieloczęściowy plik oraz dodatkowe informacje o zdjęciu. Na podstawie przesłanych danych tworzony jest obiekt *Photo*, który jest następnie zapisywany w bazie danych. Obiekt w momencie zapisywania otrzymuje swój unikatowy id, który służy jako nazwa pliku w następnym kroku. Otrzymany wieloczęściowy plik zapisywany jest jako format PNG. Jeżeli plik o danej nazwie istnieje jest, to on zastępowany nowym. Końcowo do bazy danych przesyłana jest ścieżka dostępu do pliku oraz generowana jest odpowiedź zwrotna zawierające utworzone zdjęcie.

Listing 14. Funkcja *uplaodToLocalFileSystem(...)*

@PostMapping(path = "/upload", consumes = MediaType.MULTIPART\_FORM\_DATA\_VALUE)  
public ResponseEntity<Photo> uploadToLocalFileSystem(@RequestParam("file") MultipartFile file,  
 @RequestParam(name = "productId", required = false) Integer productId,  
 @RequestParam(name = "wikiEntryId", required = false) Integer wikiEntryId,  
 @RequestParam("type") String type, @RequestParam("description") String description,  
 @RequestParam("enabled") boolean enabled)  
{  
  
 Photo newPhoto = new Photo();  
  
 if (wikiEntryId != null && wikiEntryId > 0)  
 {  
 newPhoto.setWikiEntryId(wikiEntryId);  
 }  
  
 if (productId != null && productId > 0)  
 {  
 newPhoto.setProductId(productId);  
 }  
  
 newPhoto.setDescription(description);  
 newPhoto.setEnabled(enabled);  
 newPhoto.setType(type);  
 newPhoto.setPath("");  
  
 Photo savedPhoto = photoService.save(newPhoto);  
  
 String fileName = StringUtils.cleanPath(savedPhoto.getPhotoId() + ".png");  
 Path path = Paths.get("..//Photos//" + fileName);  
 try  
 {  
 Files.copy(file.getInputStream(), path, StandardCopyOption.REPLACE\_EXISTING);  
 }  
 catch (IOException e)  
 {  
 e.printStackTrace();  
 }  
  
 String fileDownloadUri = "/photo/download/" + fileName;  
  
 savedPhoto.setPath(fileDownloadUri);  
 photoService.update(savedPhoto);  
  
 return new ResponseEntity<>(savedPhoto, HttpStatus.OK);  
  
}

* + 1. Funkcja serwisu odpowiedzialna za aktualizację wpisu na wiki

Listing 15 zawiera implementację funkcji *updateWikiEntry()*, która jest odpowiedzialna za aktualizowanie wpisów wiki. Na początku sprawdzane są argumenty funkcji, w przypadku podania niepoprawnych danych generowana jest odpowiedź zwrotna z informacją o błędzie. Gdy podane argumenty są poprawne obiekt wiki jest aktualizowany w oparciu o nowe dane, a następnie dane są zapisywane w bazie danych. Po poprawnej operacji wysyłana jest informacja zwrotna zawierająca zaktualizowany wpis wiki.

Listing . Funkcja *updateWikiEntry(...)*

@Override  
@Transactional  
public ResponseEntity<WikiEntry> updateWikiEntry(String wikiEntryId, WikiEntry wikiEntry)  
{  
 Integer id = null;  
 try  
 {  
 id = Integer.decode(wikiEntryId);  
 }  
 catch (Exception ex)  
 {  
 return new ResponseEntity<WikiEntry>(HttpStatus.BAD\_REQUEST);  
 }  
  
 WikiEntry wikiEntryToUpdate = wikiEntryDAO.findById(id);  
  
 if (wikiEntryToUpdate == null)  
 {  
 return new ResponseEntity<WikiEntry>(HttpStatus.BAD\_REQUEST);  
 }  
  
 wikiEntryToUpdate.setName(wikiEntry.getName());  
 wikiEntryToUpdate.setLatinName(wikiEntry.getLatinName());  
 wikiEntryToUpdate.setShortDescription(wikiEntry.getShortDescription());  
 wikiEntryToUpdate.setLongDescription(wikiEntry.getLongDescription());  
 wikiEntryToUpdate.setTreatment(wikiEntry.getTreatment());  
 wikiEntryToUpdate.setTips(wikiEntry.getTips());  
 wikiEntryToUpdate.setTags(wikiEntry.getTags());  
  
 wikiEntryDAO.update(wikiEntryToUpdate);  
  
 return new ResponseEntity<WikiEntry>(wikiEntryToUpdate, HttpStatus.OK);  
  
}

* + 1. Funkcja znajdująca magazyn

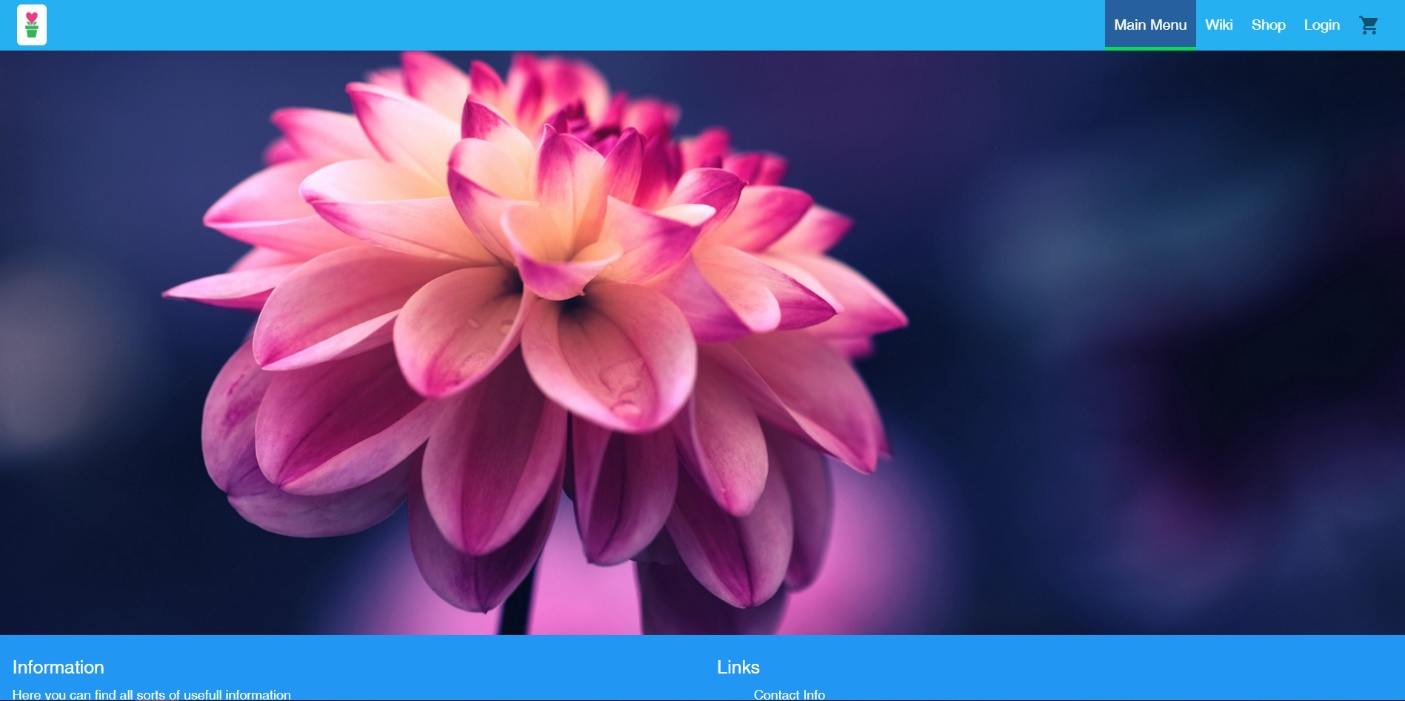
Listing 16 zawiera implementację funkcji *findByProduct(…)*, która wyszukuje magazyn na podstawie produktu. Na początku wypakowywana jest sesja połączenia z bazą danych. Sesja ta umożliwia tworzenie zapytań do bazy danych. Tworzone jest zapytanie z parametrem, które znajduje rekordy w bazie danych. Dodatkowo definiujemy, że odpowiedzią na zapytanie jest obiekt klasy *Storage*. Parametr jest ustawiany, a następnie zapytanie jest wykonywane. Wynik zapytania jest przypisywany do listy *storageList*, jeżeli lista jest pusta zwracana jest wartość *null*, w przeciwnym przypadku zwracana jest pierwsza wartość znajdująca się na liście *storageList*.

Listing 16. Funkcja findByProduct(...)

@Override  
public Storage findByProduct(Product product)  
{  
 Session session = entityManager.unwrap(Session.class);  
 Query<Storage> query = session.createQuery("FROM Storage s WHERE s.product=:product",Storage.class);  
 query.setParameter("product", product);  
 List<Storage> storageList = query.getResultList();  
 if(storageList.isEmpty())  
 return null;  
 return storageList.get(0);  
}

* 1. Struktura interfejsu graficznego

Poniżej znajdują się rysunki przestawiające wygląd aplikacji na ekranie komputera i telefonu. Motywem przewodnim w tworzonej aplikacji były kwiaty i kolor niebieski. W górnej części ekranu znajduje się pasek aplikacji z którego możemy przechodzić na poszczególne zakładki. W widoku klienta są to: ekran główny, wiki, sklep, opcja logowania oraz koszyk. Dla zalogowanego klienta opcja logowania zmienia się w ikonę użytkownika z dodatkowym menu. Widok pracownika jest rozbudowany o dodatkowe zakładki jak produkty i magazyn. W wersji na urządzenia mobilne górny pasek zmienia się w wysuwane boczne menu. Na spodzie aplikacji znajduje się stopka, która zawiera przeróżne linki z informacjami. Wygląd aplikacji jest dynamicznie skalowany w zależności od rozmiaru urządzenia.

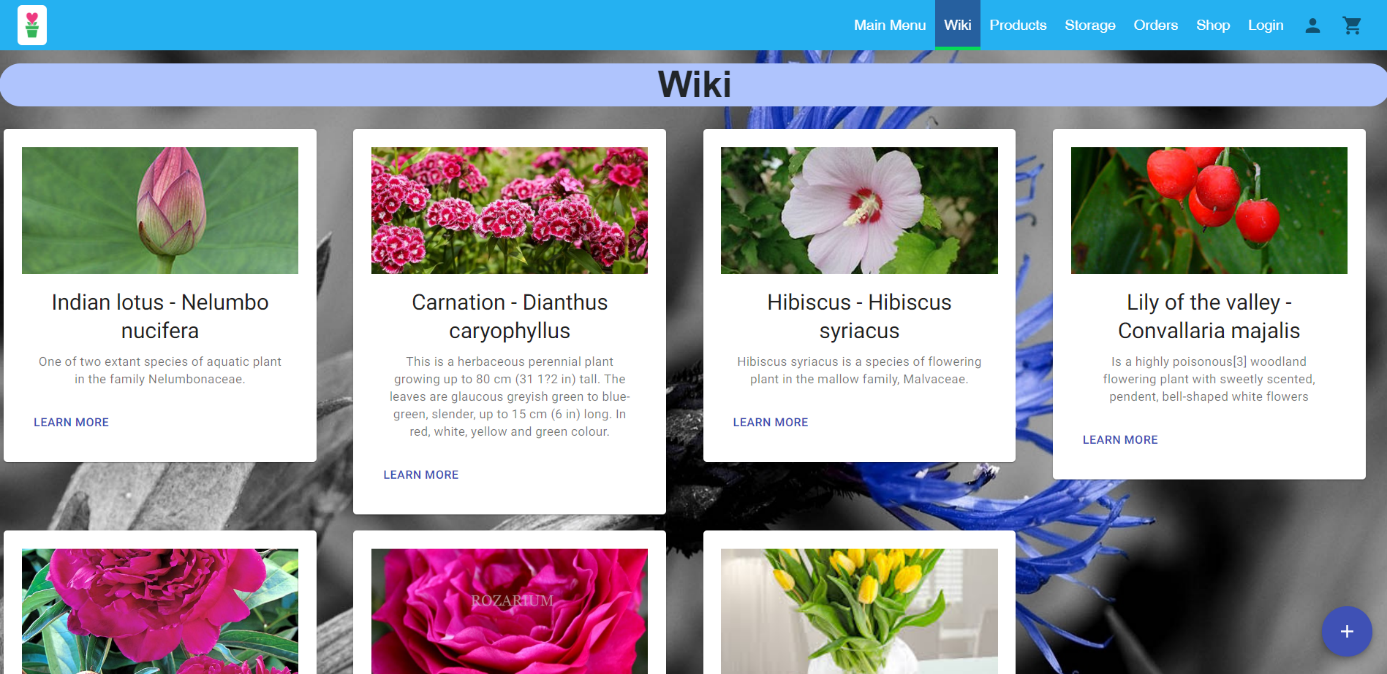


Rysunek . Ekran główny aplikacji (desktop)

Obraz zawierający roślina, kwiat

Opis wygenerowany automatycznie

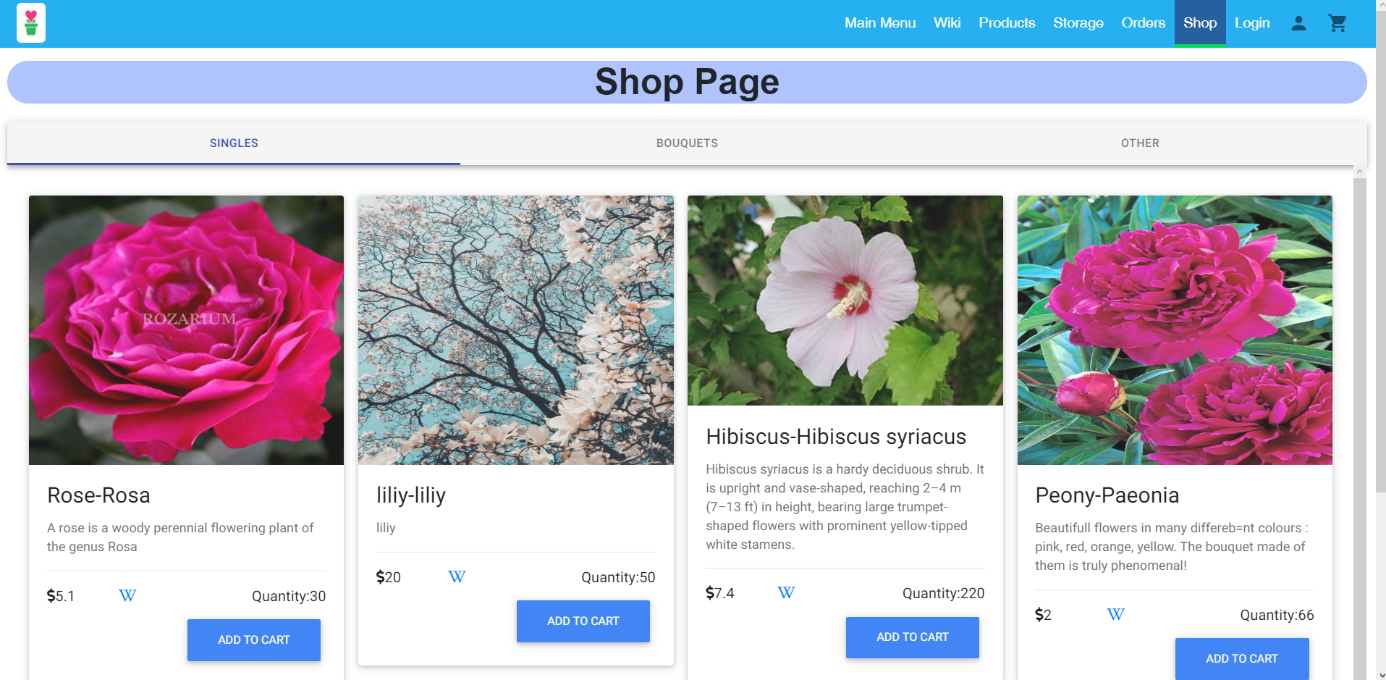
Rysunek . Ekran główny aplikacji (telefon)



Rysunek . Widok wiki (desktop)



Rysunek . Widok wpisu na wiki (desktop)



Rysunek . Widok sklepu (desktop)

Obraz zawierający zrzut ekranu

Opis wygenerowany automatycznie

Rysunek . Menu boczne oraz widok na wiki (telefon)

Obraz zawierający zrzut ekranu

Opis wygenerowany automatycznie

Rysunek . Modal ze zdjęciem oraz widok sklepu (telefon)

1. Testy

Rozdział ten jest poświęcony testowaniu wytworzonego oprogramowania.

* 1. Testy manualne

Aplikacja była poddana rygorystycznym testom manualnych, sprawdzane były przypadki przesyłania poprawnych i niepoprawnych danych. Testom manualnym podlegała głownie część Back-Endowa aplikacji, ponieważ to ona zajmowała się logiką biznesową. Narzędzie Postman idealnie nadawało się do przeprowadzania tego rodzaju testów, ponieważ pozwalało dowolnie modyfikować dane oraz metody wysyłania danych. Dodatkowo narzędzie to udostępniało historię już wysłanych zapytań. Poniżej przedstawiony został scenariusz jednego z testów.

**Test:** Rejestracja użytkownika z poprawnymi danymi.

**Warunki początkowe:**

W bazie danych nie istnieje użytkownik z takim samym emailem jak ten, który chcemy zarejestrować.

**Dane:**

Poprawne dane osobowe użytkownika i unikatowy email oraz poprawne dane adresowe.

**Przebieg:**

Dane są sprawdzane pod kątem poprawności, następnie z bazy danych pobierane są rekordy z podanym emailem, znalezienie rekordu oznacza porażkę. Dane adresowe są sprawdzane pod kątem tego czy istnieją w bazie. Jeżeli konkretny adres istnieje, to jest on przypisywany do użytkownika, jeżeli nie istnieje, nowy adres jest tworzony i przypisywany do użytkownika. Tworzony jest nowy użytkownik na podstawie przesłanych danych, generowana jest odpowiedz 200 oznaczająca, że wszystko jest w porządku. Następnie sprawdzane są stworzone dane w bazie danych, a mianowicie to czy ściśle odpowiadają danym rejestrowanego użytkownika. Przypadek braku jednolitości danych oznacza porażkę. Poprawne wykonanie wszystkich kroków oznacza sukces.

**Oczekiwany wynik:**

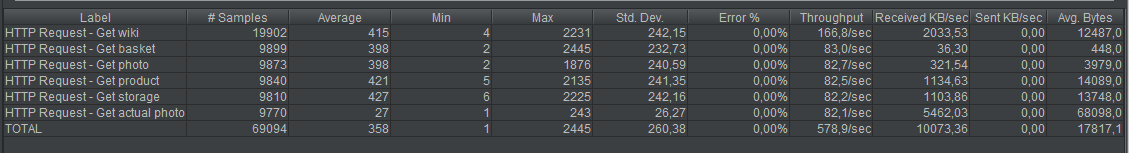
Sukces.

**Otrzymany wynik:**

Sukces.

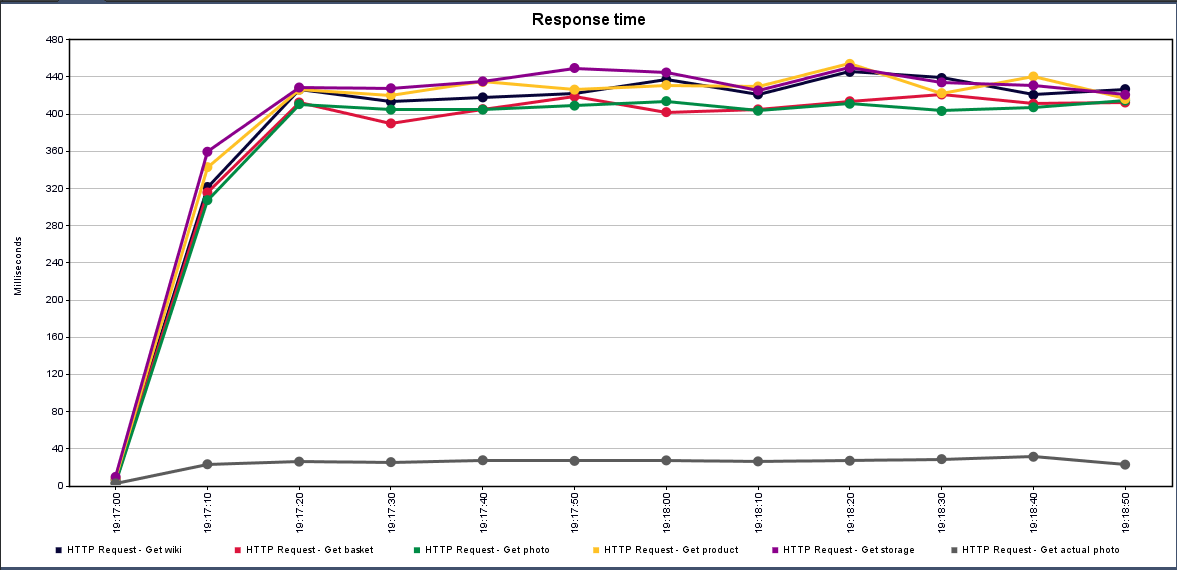
* 1. Testy wydajnościowe

Powstała aplikacja została przetestowana pod kątem wydajności. Programem użytym do wykonania testu był JMeter. W konfiguracji początkowej ustawione zostało trzysta równoległych wątków, które miały symulować trzystu jednoczesnych użytkowników serwisu. Do badania zostało wyznaczonych sześć punktów końcowych, które są najczęściej używane w aplikacji. Badanie trwało 120 sekund wraz z 5 sekundowym czasem rozgrzania i ostudzenia. Łącznie zostało wykonanych ok 70000 zapytań, co się przekłada na ok 600 zapytań na sekundę. Na rysunku 13 możemy zauważyć, że łączny procent błędów to 0, dodatkowo możemy zaobserwować, że aplikacja wysyłała około 10 MB danych na sekundę.



Rysunek Wyniki testu wydajnościowego

Rysunek 14 przedstawia średni czas odpowiedzi na zapytania. Wahał się on w granicy 450 ms. Jest to świetny wynik. Warto zauważyć, że w tworzonej aplikacji większość zapytań będzie wysłana przy pierwszym ładowaniu aplikacji, więc ewentualne opóźnienia w dostarczaniu danych nie będą mocno wyczuwalne podczas używania serwisu.



Rysunek Czas odpowiedzi na zapytania

* 1. Testy funkcjonalne

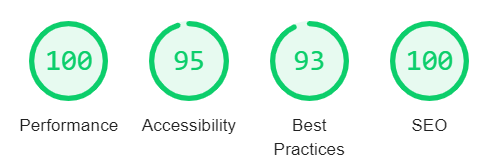
Tworzona aplikacja została przetestowana pod kątem funkcjonalnym, przy użyciu narzędzia Selenium. Selenium pozwala nam zdefiniować scenariusze testów. Testowane przypadki to:

* przechodzenie pomiędzy zakładkami serwisu,
* rejestracja klienta,
* logowanie się na konto utworzonego klienta,
* dodanie produktu do koszyka klienta,
* zakup produktu,
* wylogowanie się z systemu,
* zalogowanie się na konto pracownika,
* dodanie nowego produktu,
* usunięcie dodanego produktu,
* dodanie nowego wpisu na wiki,
* usuniecie dodanego wpisu na wiki.

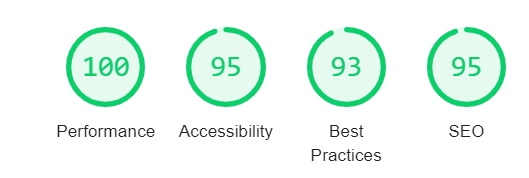
Wszystkie testowe scenariusze były zakończone sukcesem.

* 1. Audyt Lighthouse

Narzędzie Lighthouse pozwala na szybkie przeprowadzenie audytu aplikacji na platformie desktopowej i mobilnej. Audyt przeprowadzany był pod kątem: wydajności, dostępności, użycia dobrych praktyk programowania oraz potencjalnego pozycjonowania strony w wyszukiwarce. Rysunek 15 przedstawia audyt dla aplikacji wyświetlanej w przeglądarce desktopowej. Wszystkie osiągnięte wyniki są powyżej 90%. Rysunek 16 przedstawia wyniki audytu aplikacji dla przeglądarki mobilnej.



Rysunek Wynik audytu dla przeglądarki desktopowej



Rysunek wynik audytu do urządzania mobilnego

1. Podsumowanie i wnioski

# Literatura

[1] Dokumentacja Java 8: https://docs.oracle.com/javase/8/docs/ [dostęp 29.11.2019].

[2] Dokumentacja Eclipse: https://www.eclipse.org/documentation/ [dostęp 06.12.2019].

[3] Dokumentacja Springa w wersji 5.2.1: https://spring.io/projects/spring-framework [dostęp 29.11.2019].

[4] Craig Walls. Spring w akcji. Wydanie V. Helion 2019.

[5] Dokumentacja Springa Boot w wersji 2.2.1: https://spring.io/projects/spring-boot [dostęp 29.11.2019].

[6] Dokumentacja Apache Maven: https://maven.apache.org/guides/index.html [dostęp 29.12.2019].

[7] Richard E. Silverman. Leksykon kieszonkowy Git. Helion 2014

[8] Christian Bauer, Gavin King. Hibernate w akcji. Helion 2007

[9] Dokumentacja React.js: https://pl.reactjs.org/docs/getting-started.html [dostęp 29.11.2019]

[10] Dokumentacja Redux: https://redux.js.org/introduction/getting-started [dostęp 29.11.2019]

[11] Dokumentacja Axios: https://github.com/axios/axios [dostęp 29.11.2019]

[12] Dokumentacja Material-UI: https://material-ui.com [dostęp 29.11.2019]

[13] Dokumentacja MDBReact: https://mdbootstrap.com/docs/react/ [dostęp 29.11.2019]

# Zawartość płyty

Do niniejszej pracy została dołączona płyta, na której znajduje się:

* kod źródłowy programu,
* elektroniczna wersja pracy w formacie pdf,
* instrukcja instalacji aplikacji,
* instrukcja użytkownika.

1. https://penelope.uchicago.edu/~grout/encyclopaedia\_romana/aconite/semperaugustus.html [↑](#footnote-ref-1)
2. Gulden holenderski – waluta Holandii używana od XVII wieku do 2002 roku [↑](#footnote-ref-2)
3. https://www.amazon.com [↑](#footnote-ref-3)
4. http://www.allegro.pl/ [↑](#footnote-ref-4)
5. REST API – zbiór reguł definiujący format wysyłanych zasobów przesyłanych protokołem HTTP [↑](#footnote-ref-5)
6. Obiektowy model dokumentu – drzewiasta reprezentacja dokumentu(XML lub HTML), gdzie każdy węzeł reprezentuje fragment dokumentu. Każda gałąź kończy się węzłem [↑](#footnote-ref-6)
7. Promise – jeden z mechanizmów języka JavaScript, jest to wynik funkcji asynchronicznej wraz z automatycznym łapaniem wyjątków. [↑](#footnote-ref-7)
8. https://www.nasa.gov [↑](#footnote-ref-8)
9. https://www.capgemini.com/pl-pl/ [↑](#footnote-ref-9)
10. https://www.shutterstock.com/pl/ [↑](#footnote-ref-10)