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

JDK(ang. *Java Development Kit*)

JVM (ang. *Java Virtual Machine*) – maszyna wirtualna Javy

JRE (ang. Java Runtime Environment)

REST (ang. *Representational state transfer*)

ORM (ang. *Object-Relational Mapping*)

SQL (ang. *Structured Query Language*)

JS (ang. *JavaScript*)

SPA (ang. *Single-page application*)

JSON (ang. *JavaScript Object Notation*)

XML (ang. *Extensible Markup Language*)

API (ang. *Application programming interface*)

JPA(ang. Java Persistence API)

CSS (ang. *Cascading Style Sheets*)

POM (ang. *Project Object Model*)

UML (ang. *Unified Modeling Language*)

ISO (ang. *International Standards Organization*)

IT (ang. *Information Technology*)

IDE (ang. Integrated Development Environment)

JDBC

IoC

AOP

EJB

JAR

URL

HTTP

EE

XSRF

JSX

HTML

PNG

DAO

1. Wstęp

1.1. Wprowadzenie

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Mauris id dapibus enim. Etiam lobortis pulvinar enim in maximus.

1.2. Cel i zakres pracy

Celem projektu jest stworzenie responsywnej aplikacji webowej umożliwiającej handel kwiatami oraz udostępniającej informacji na ich temat. Aplikacja zostanie stworzona w oparciu o trójwarstwową architekturę (warstwa prezentacji, logiki biznesowej, danych). System będzie podzielony na dwa 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

2. 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.

2.1. Opis aktorów

W systemie możemy wyróżnić trzech aktorów: Gościa, Klienta i Pracownika. Każdy z nich cechuje się funkcjami, jakimi powinien być w stanie dokonywać w systemie.

- **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 fizyczne realizuje zamówienia klientów i modyfikuje ich status w systemie.

2.2. Wymagania funkcjonalne

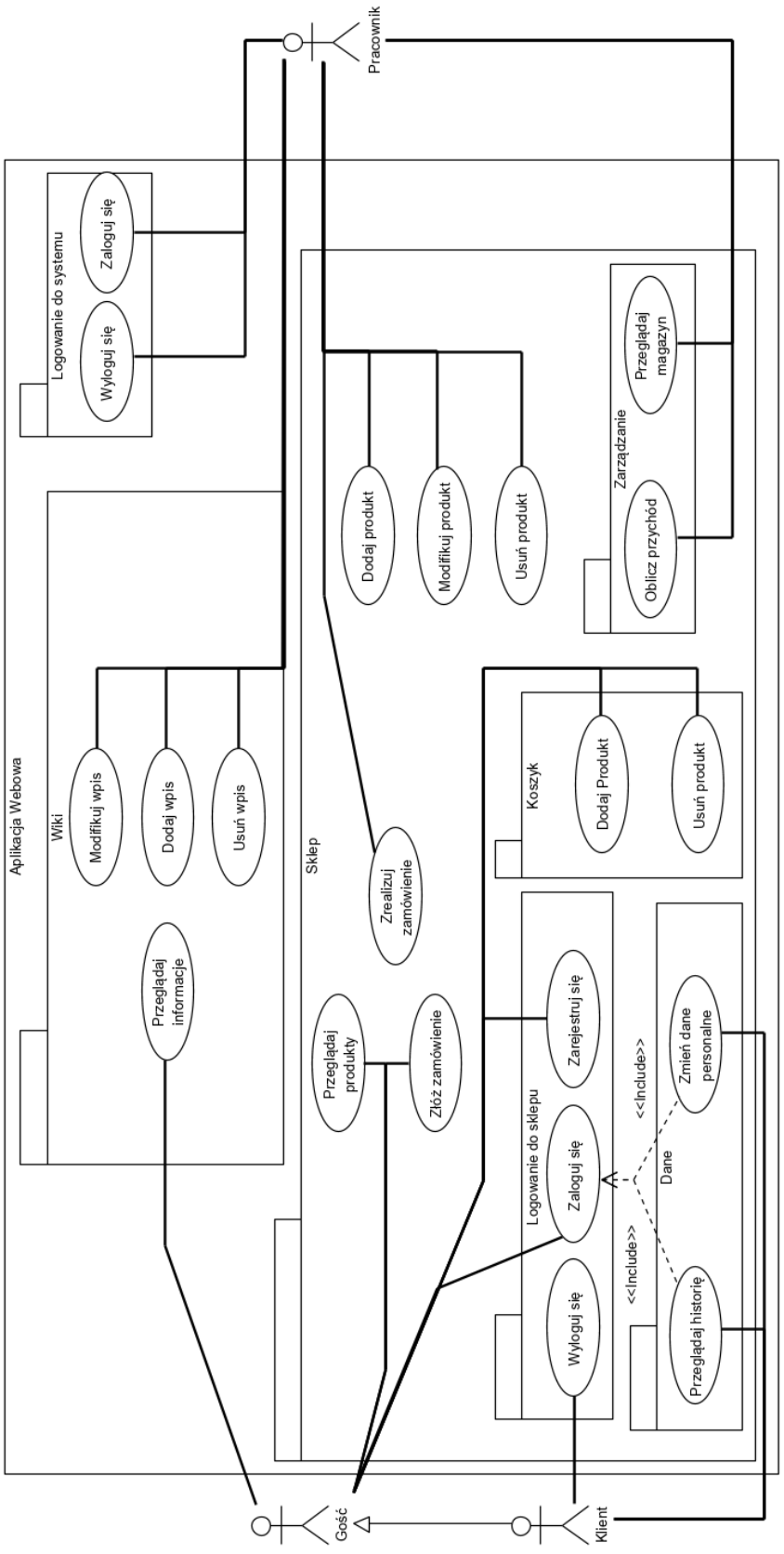
Tabela 1 Wymagania funkcjonalne aplikacji

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 produktu 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 zalogowanie 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 na 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 mu status.
FU20	Oblicz przychody	Pracownik ma podgląd na przychody w danym miesiącu.
FU21	Przeglądaj magazyn	Pracownik ma wgląd w liczność produktów.

Na Rysunek 1 przedstawiony został diagram przypadków użycia. Na diagramie przedstawiono trzech aktorów oraz odpowiednie 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 1. Diagram przypadków użycia

2.3. Wymagania niefunkcjonalne

Tabela 2 Wymagania niefunkcjonalne aplikacji

Wymagania niefunkcjonalne		
Id	**Nazwa**	**Opis**
NFU01	Obsługa wielu użytkowników	Aplikacja powinna być w stanie obsłużyć co najmniej 50 jednoczesnych 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.

2.4. 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[1]. 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 powinna wyglądać dobrze w przeglądarce na ekranie komputera jak i telefonu.

[1] REST API – zbiór reguł definiujący format wysyłanych zasobów przesyłanych protokołem HTTP

3. Wykorzystane technologie i narzędzia programistyczne

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

3.1. 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.

3.2. 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. Eclipse 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.

3.3. Spring

Spring Framework[2][3] 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.

3.4. Spring Boot

Spring Boot[4] 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.

3.5. Maven

Apache Maven[5] jest narzędziem ułatwiającym tworzenie projektów w Javie. Udostępnia on jednolity system budowy projektów, jak 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

3.6. Git

Git[6] 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.

3.7. Hibernate

Hibernate[7] jest frameworkiem pozwalającym na automatyczne mapowanie obiekty 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.

3.8. MySQL

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

3.9. 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.

3.10. React.js

React.js[8] biblioteka do języka JavaScript stworzona przez programistę Facebooka. Główną cechą React jest to, że korzysta z wirtualnego obiektowego modelu dokumentu[2] (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 pisać 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

[2] Obiektowy model dokumentu – drzewiasta reprezentacja dokumentu(XML lub HTML), gdzie każdy węzeł reprezentuje fragment dokumentu. Każda gałąź kończy się węzłem

3.11. Visual Studio Code

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

3.12. Redux

Redux[9] 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.

3.13. Axios

Axios[10] jest biblioteką do JavaScriptu, która udostępnia klienta HTTP opartego na promisach[3]. Pozwala on w szybki sposób tworzyć zapytania HTTP, automatycznie formatuje dane na format JSON oraz zapewnia ochronę przed XSRF po stronie klienta.

3.14. Material-UI

Material-UI[11] 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[4], Capgemini[5], Shutterstock[6] i wiele innych.

3.15. MDBReact

MDBReact[12] podobnie do Material-UI jest biblioteką komponentów do React.js.

[3] Promise – jeden z mechanizmów języka JavaScript
[4] https://www.nasa.gov
[5] https://www.capgemini.com/pl-pl/
[6] https://www.shutterstock.com/pl/

3.16. 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

4. Projekt systemu

Rozdział ten przedstawia architekturę aplikacji wraz z projektem bazy danych.

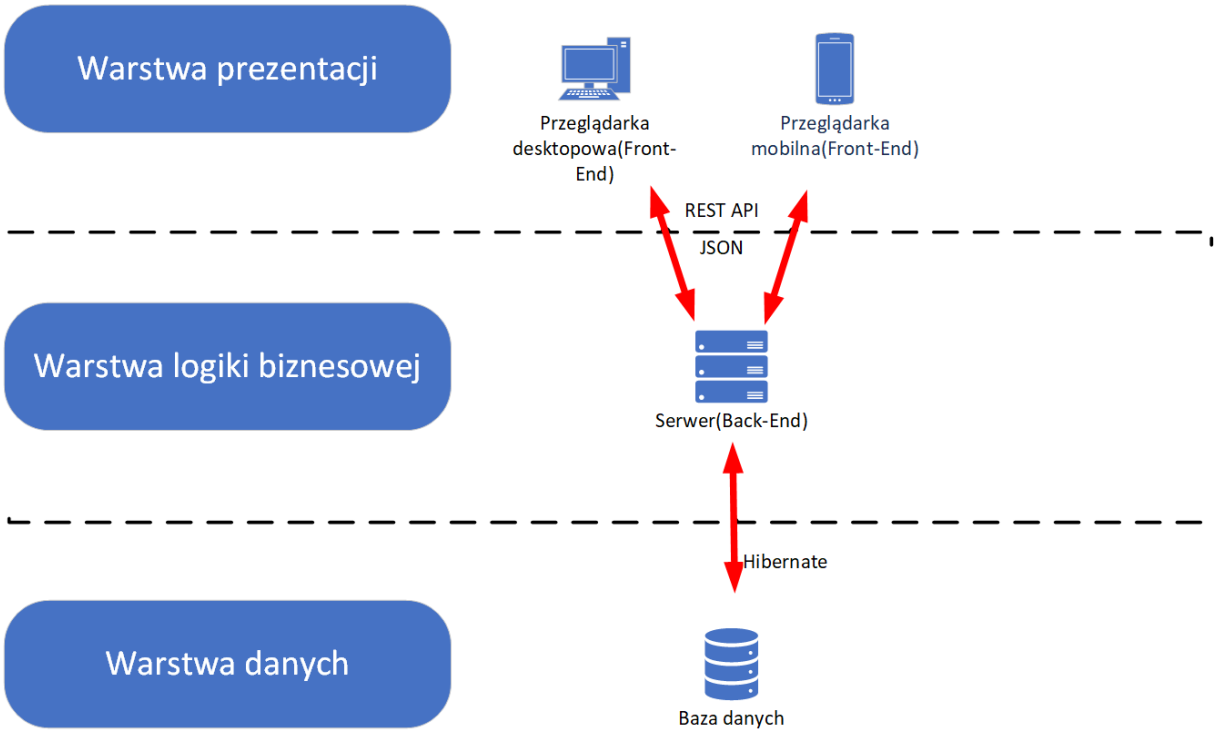
4.1. Architektura aplikacji

Aplikacja została stworzona w trójwarstwowym modelu. Model został pokazany na Rysunek 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 wyświetlana 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 jak i skalowalność aplikacji. Dodatkowym atutem jest możliwość podziału pracy na fragmenty, z których każdy może zostać wykonywany jednocześnie. Wadą tej architektury jest nakład pracy potrzebny do wykonania trzech elementów. Dodatkową wadą jest

propagacja błędów wykonanych w fazie projektowania, jednak często jest ona ograniczona tylko do poszczególnej warstwy.



Rysunek 2 Model trójwarstwowy aplikacji

4.2. Projekt bazy danych

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

4.2.1. 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 Tabela 3 przedstawiony został opis encji na podstawie elementów świata rzeczywistego. W Tabela 4 pokazany został opis relacji pomiędzy encjami.

Tabela 3 Opis encji na podstawie elementów świata rzeczywistego

Opis encji	
Obiekt świata rzeczywistego	**Opis**

Użytkownik	Zarejestrowana osoba lub pracownik. Dodatkowo dla gościa, który dokona zakupy zostanie utworzone konto. Na konto to 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 ze 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 to zbiór informacji na temat danego kwiatka. Pracownik jest w stanie dodawać modyfikować i usuwać wpisy na wiki. Użytkownik na podstawie nazwy pospolitej, łacińskiej oraz 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 wykonywać zamówienia przygotowując towar i zmieniając mu jego 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 na szybki powrót 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 4 Opis relacji pomiędzy encjami

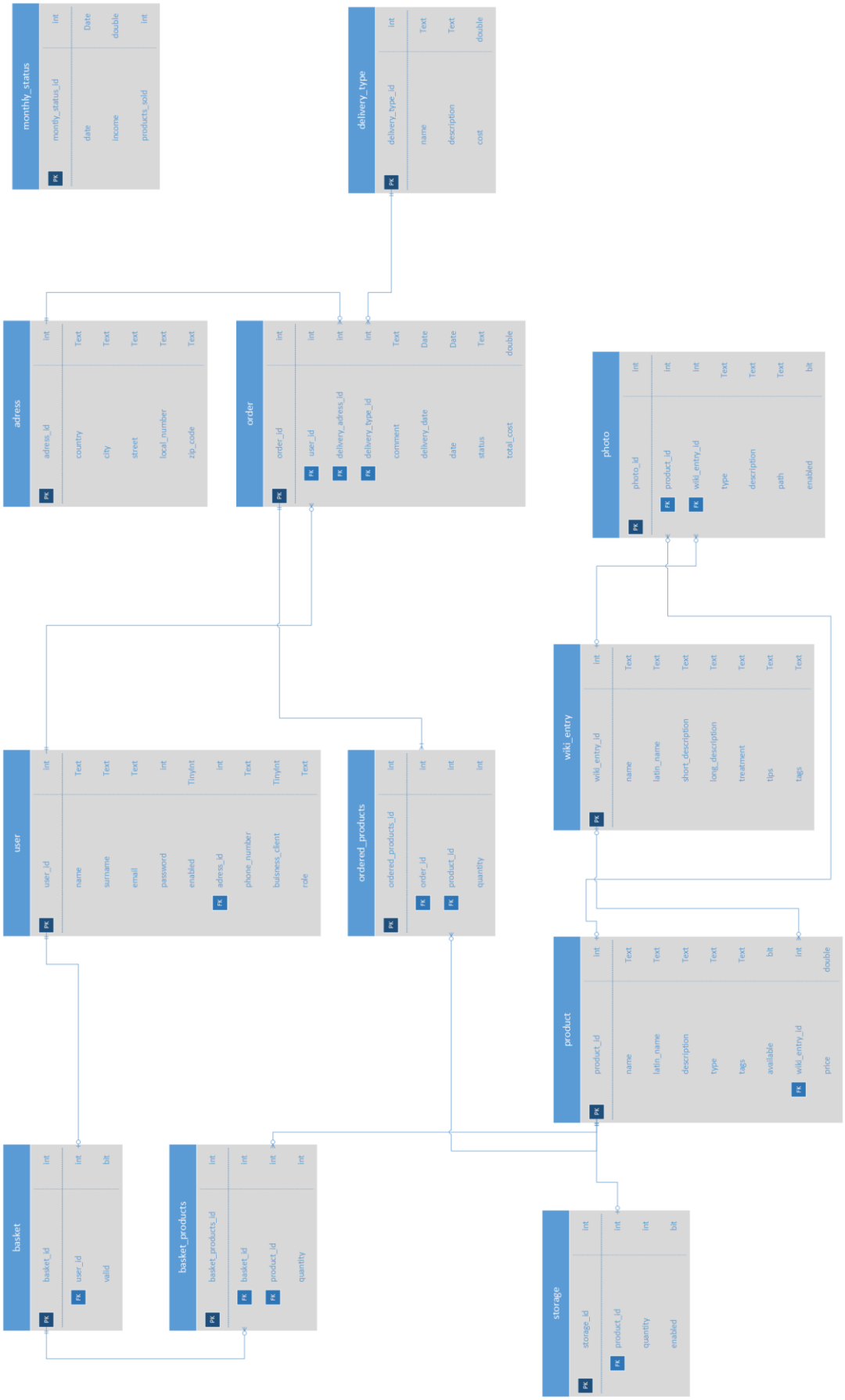
Opis relacji	
Relacja pomiędzy encjami	**Opis**
Użytkownik - Koszyk	Użytkownik może posiadać tylko jeden koszyk, koszyk musi należeć wyłącznie do jednego użytkownika
Użytkownik – Adres	Użytkownik musi posiadać wyłącznie jeden adres, jeden adres może należeć do wielu użytkowników
Użytkownik - Zamówienie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

Zamówienie – Adres	Zamówienie musi posiadać jeden adres, adres może należeć do wielu zamówień

4.2.2. 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 wydajność zapytań, poprzez zmniejszenie ilość 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 **Błąd! Nie można odnaleźć źródła odwołania.**.



Rysunek 3 Model fizyczny bazy danych

Listing 1 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')
```

5. Implementacja systemu

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

5.1. 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.

5.1.1. Zależności w pliku POM.xml

Na **Błąd! Nie można odnaleźć źródła odwołania.** widzimy artefakty zależności używanych w projekcie. Maven na podstawie tych zależności pobierze nam 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.

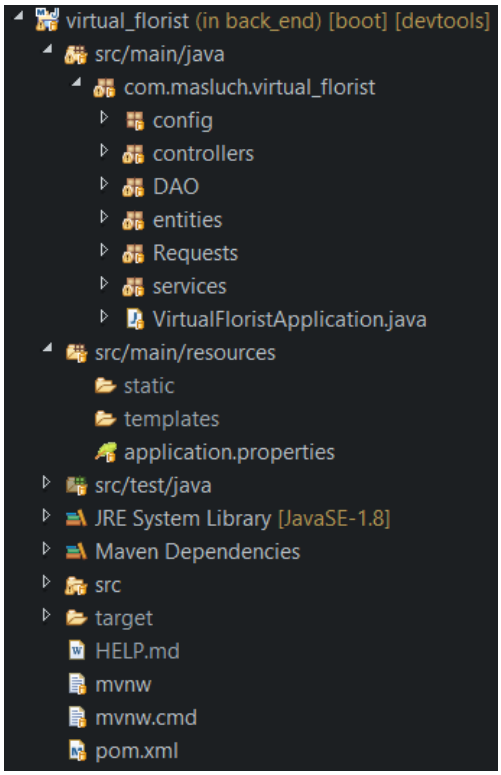
Listing 2 Wycinek pliku POM.xml

```
<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>
```

5.1.2. Struktura projektu

Na Rysunek 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 4 Struktura projektu części Back-End

5.1.3. Encje

Na **Błąd! Nie można odnaleźć źródła odwołania.** 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, Cas-
cadeType.MERGE, CascadeType.PERSIST, CascadeType.REFRESH})
    @JoinColumn(name = "wiki_entry_id")
    private WikiEntry wikiEntry;

    @OneToMany(fetch = FetchType.LAZY, cascade = {CascadeType.DETACH, Cas-
cadeType.MERGE, CascadeType.PERSIST, CascadeType.REFRESH})
    @JoinColumn(name = "product_id")
    private List<Photo> photos;
}
```

5.1.4. 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 jak i implementacja zostały przedstawione na **Błąd! Nie**

można odnaleźć źródła odwołania.. Możemy zauważyć, że korzystamy z 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 go ze zmienną entityManager. Widzimy też wykorzystanie wbudowanych klasy 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);
    }
}
```

5.1.5. Serwisy

Na **Błąd! Nie można odnaleźć źródła odwołania.** przedstawiono interfejs ProductService oraz fragment jego implementacji. W serwisie wykorzystano wzorzec projektowy fasada w celu ujednolicenia dostępu do implementacji w innych lokacjach kodu. Wszystkie serwisy w pakiecie service 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 ResonseEntity, 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, oznacza ona to, że operacje dokonywane na bazie w danej funkcji są wewnątrz transakcji.

Listing 5 Interjest 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 newProd-
uct, 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<Prod-
uct>(savedProduct, HttpStatus.OK);
        return response;
    }

    @Override
    @Transactional
    public ResponseEntity<Product> addNewProductWithWiki(Product newProd-
uct, 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);
    }
```

5.1.6. Kontrolery

Błąd! Nie można odnaleźć źródła odwołania. 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 jak i zawartości zapytań.

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 new-
Product,@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);
    }


}
```

5.2. Front-End

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

5.2.1. Zależności w pliku package.json

Listing 7 jest fragmentem pliku packag.json. Zawiera on w sobie zależności jakie pobiera NPM.

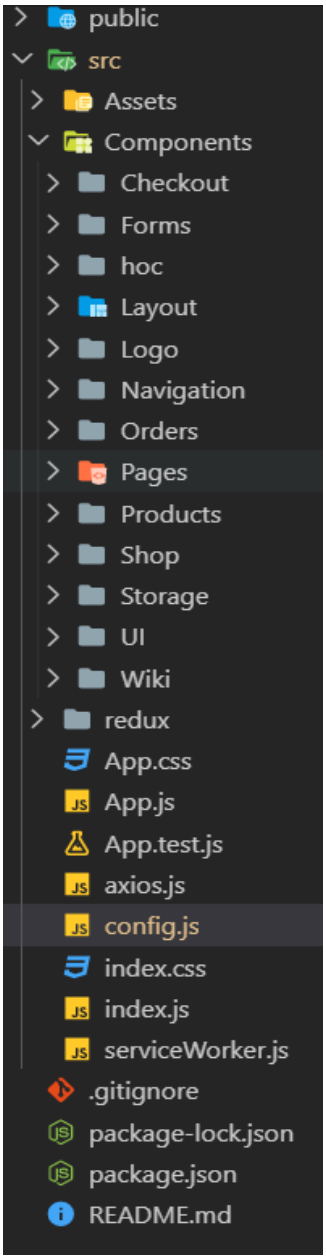
```
"dependencies": {
        "@material-ui/core": "^4.6.0",
        "@material-ui/icons": "^4.5.1",
        "@material-ui/styles": "^4.6.0",
        "@mdi/font": "^4.5.95",
        "axios": "^0.19.0",
        "bootstrap": "^4.3.1",
```

```
"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"
}
```

Listing 7 Wycinek pliku package.json

5.2.2. Struktura

Na Rysunek 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 UI oraz Layout zawierają komponenty związane z układem i wyglądem aplikacji. Zgodnie z filozofią Reacta wiele elementów było używane kilkukrotnie. Folder Pages zawiera w sobie widoki poszczególnych stron dostępnych w aplikacji.



Rysunek 5 Struktura Front-Endu

Literatura

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

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

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

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

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

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

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

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

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

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

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

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

Dodatek A