POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

**Wydział Informatyki**

**Katedra Metod Programowania**

Programowanie aplikacji biznesowych

**Daniel Gut**
Nr albumu s7452

# System wspomagania działu wsparcia technicznego w firmie technologicznej

Praca inżynierska

dr Krzysztof Barteczko

Warszawa, lipiec, 2021

## Streszczenie

Celem pracy inżynierskiej było stworzenie systemu wspomagającego pracę działu wsparcia technicznego w firmie będącej producentem sprzętu sieciowego. Głównym zadaniem systemu jest rejestracja klientów w bazie danych, powiązanie produktów z klientami na podstawie numeru seryjnego oraz obsługa zgłoszeń przypisanych do konkretnego produktu. System miał być prosty, ale zapewniać możliwość rozbudowy o nowe funkcjonalności lub dodatkową aplikację mobilną.

System został oparty na architekturze REST z wykorzystaniem frameworków Spring oraz Angular, języka Java oraz TypeScript, Hibernate, Bootstrap oraz bazy danych MySQL.



## Słowa kluczowe

Java, Angular, Hibername, HTML, Java Script, TypeScript, CSS, MySQL, REST API, Postman, zarządzanie klientami, zarządzanie produktami, zarządzenia zgłoszeniami, wsparcie techniczne, aplikacja webowa, front-end, back-end, zasób, endpoint

# Spis treści

**STRESZCZENIE** ..... **2**

**SŁOWA KLUCZOWE** ..... **2**

**SPIS TREŚCI** ..... **3**

**1. WSTĘP** ..... **5**

1.1. CEL PRACY ..... 5
1.2. REZULTAT PRACY ..... 5
1.3. ORGANIZACJA PRACY ..... 6

**2. ISTNIEJĄCE SYSTEMY WSPOMAGAJĄCE PRACĘ DZIAŁU WPARCIA TECHNICZNEGO** ..... **6**

2.1. TEAMSUPPORT ..... 6
2.2. FOCALSCOPE ..... 7
2.3. WIX ANSWERS ..... 8
2.4. SUPPORTBEE ..... 9
2.5. PODSUMOWANIE ISTNIEJĄCYCH ROZWIĄZAŃ ..... 10

**3. KONCEPCJA I FUNKCJONALNOŚĆ SYSTEMU** ..... **11**

3.1. ZAŁOŻENIA ..... 11
3.2. DEFINICJE ..... 11
3.3. UŻYTKOWNICY SYSTEMU ..... 12
*3.3.1. Użytkownik* ..... *12*
*3.3.2. Moderator* ..... *12*
*3.3.3. Administrator* ..... *12*
3.4. PRZYPADKI UŻYCIA ..... 12
*3.4.1. Przykładowy scenariusz przypadku użycia na podstawie dodawania zgłoszenia* ..... *16*
3.5. DIAGRAM KLAS ..... 16
3.6. WYMAGANIA FUNKCJONALNE ..... 18
3.7. WYMAGANIA NIEFUNKCJONALNE ..... 21

**4. WYBÓR ROZWIĄZAŃ TECHNOLOGICZNYCH** ..... **22**

4.1. SPRING FRAMEWORK ..... 22
4.2. GRADLE ..... 24
4.3. JAVA ..... 25
4.4. ANGULAR ..... 26
4.5. TYPESCRIPT ..... 27
4.6. BOOTSTRAP ..... 28
4.7. MYSQL ..... 29
4.8. ARCHITEKTURA REST ..... 30

**5. ZAGADNIENIA IMPLEMENTACYJNE** ..... **31**

5.1. ARCHITEKTURA ..... 31
5.2. APLIKACJA SERWEROWA ..... 32
*5.2.1. Kontrolery* ..... *33*
*5.2.2. Modele* ..... *35*
*5.2.3. Repozytoria* ..... *37*

*5.2.4. Serwisy* ..... *38*
*5.2.5. Payloads* ..... *39*
*5.2.6. Security* ..... *40*
5.3. APLIKACJA WEBOWA ..... 47
*5.3.1. Komponenty* ..... *48*
*5.3.2. Serwisy* ..... *53*
*5.3.3. Pipes* ..... *54*
*5.3.4. Bezpieczeństwo* ..... *55*
**6. FAZA TESTOWANIA ..... 57**
**7. PODSUMOWANIE ..... 59**
7.1. ZALETY SYSTEMU ..... 59
7.2. WADY SYSTEMU ..... 60
7.3. PROPOZYCJA ROZWOJU SYSTEMU ..... 60
**BIBLIOGRAFIA ..... 61**
**SPIS LISTINGÓW ..... 63**
**SPIS RYSUNKÓW ..... 65**
**SPIS TABEL ..... 66**

# 1. Wstęp

Dział wsparcia technicznego stanowi bardzo ważny element struktury organizacji świadczących usługi i oferujących rozwiązania sprzętowe. Jest naturalnym mostem pomiędzy użytkownikiem (zarówno klientem końcowym, jak i partnerem) a organizacją, a jego praca wymaga dobrej organizacji i właściwych narzędzi, aby odpowiednio szybko odpowiadać na zgłoszenia użytkowników i rozwiązywać sygnalizowane problemy, przy równoczesnym skrupulatnym dokumentowaniu przebiegu prac. Od jakości obsługi zależy bowiem opinia klienta, lub partnera biznesowego, jego późniejsza rekomendacja i decyzja o ponownym zakupie bądź współpracy. Do podstawowych zadań działu wsparcia technicznego należy:

- rejestrowanie zapytań klientów i partnerów, a w przypadku obsługi wewnątrz organizacji, także pracowników;
- badanie, diagnozowanie i rozwiązywanie problemów sprzętowych lub programowych;
- przeprowadzanie testów i ocen wykorzystywanych technologii;
- przestrzeganie firmowych procedur ws. delegowania bądź przekazywania zgłaszanych problemów do innych zespołów wewnątrz organizacji;
- przeprowadzanie kontroli bezpieczeństwa;
- odpowiadanie na wezwania w odpowiednim czasie.

Wydajność i właściwa organizacja pracy działu wsparcia technicznego zależy od wykorzystywanych na co dzień narzędzi, w tym aplikacji biznesowych.

## 1.1. Cel pracy

Celem niniejszej pracy inżynierskiej jest stworzenie aplikacji biznesowej na potrzeby działów wsparcia technicznego, współpracujących zarówno z użytkownikami końcowymi (model współpracy B2C; grupa produktów SOHO – z ang. small office, home office – małe, przydomowe biura), jak i z biznesem (model współpracy B2B; grupa produktów SMB – z ang. small and medium business – małe i średnie przedsiębiorstwa) oraz dokumentacja procesu jej powstawania w oparciu o nowoczesne technologie wykorzystywane obecnie w programowaniu aplikacji. Projekt zrealizowano w ramach specjalizacji z zakresu Programowania aplikacji biznesowych na Polsko-Japońskiej Akademii Technik Komputerowych.

## 1.2. Rezultat pracy

Rezultatem pracy jest prototyp systemu, którego propozycja powstała na podstawie przeprowadzonych analiz oraz prezentacja przebiegu jego implementacji udokumentowanej w niniejszym opracowaniu. Prototyp został stworzony z wykorzystaniem nowoczesnych technologii i ma on pomagać w rejestracji oraz śledzeniu zgłoszeń problemów technicznych przez różnego rodzaju działy techniczne w firmach.

## 1.3. Organizacja pracy

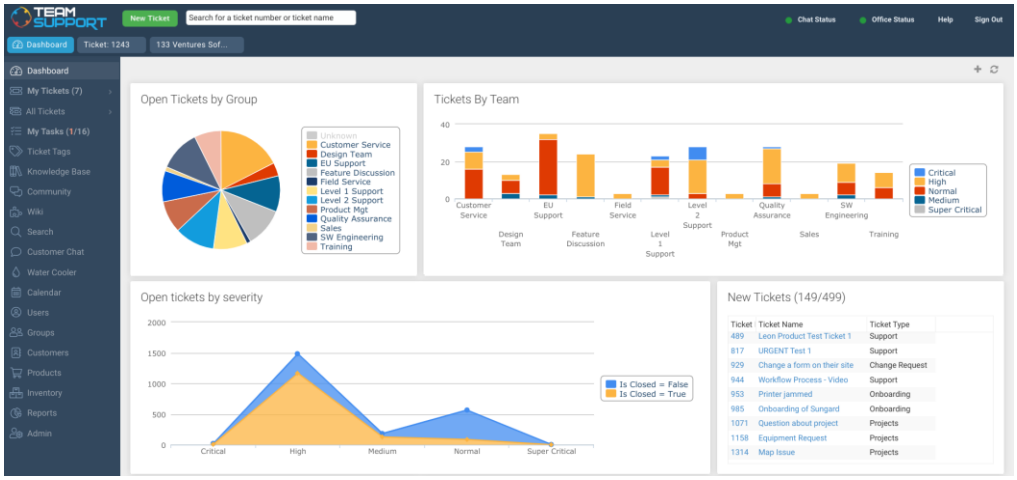
Praca składa się z siedmiu głównych rozdziałów. Pierwszy rozdział stanowi wstęp oraz wyznacza cel pracy. Drugi rozdział przedstawia istniejące podobne rozwiązania oraz ich analizę pod kątem wad oraz zalet. Trzeci rozdział opisuje koncepcję oraz funkcjonalność systemu. Przedstawia wstępne założenia, definicje, planowaną strukturę systemu, wymagania funkcjonalne oraz niefunkcjonalne. Czwarty rozdział przedstawia najważniejsze technologie użyte do implementacji prototypu systemu. Piąty rozdział przedstawia zagadnienia implementacyjne. Szósty rozdział prezentuje fazę testowania proponowanego rozwiązania. Ostatni, siódmy rozdział stanowi podsumowanie pracy z uwzględnieniem zalet i wad proponowanego rozwiązania oraz propozycją dalszego rozwoju systemu.

# 2. Istniejące systemy wspomagające pracę działu wparcia technicznego

Niniejszy rozdział poświęcony jest przedstawieniu przykładowych rozwiązań istniejących na rynku oraz prezentacji ich mocnych i słabych stron.

## 2.1. TeamSupport

Team Support (Team Support, 2021) jest kompletnym pakietem służącym do obsługi klienta. Umożliwia skuteczną komunikację miedzy różnymi działami w firmie, zarządzaniem zasobami oraz klientami. Oprócz podstawowych narzędzi do zarządzania zgłoszeniami od klientów aplikacja posiada wiele zaawansowanych funkcjonalności, takich jak integracja ze skrzynką e-mail, czat na żywo, integracja z popularnymi systemami CRM oraz zaawansowane raporty. Na uwagę zasługuje również rozbudowana dokumentacja wszystkich dostępnych funkcji. Platforma posiada rozwiązanie chmurowe, a jego wersja mobilna wpiera większość popularnych platform takich jak iOS, Android, BlackBerry. W celu zapoznania się z interfejsem należy wcześniej zaaplikować do producenta o demonstracyjne rozwiązanie. TeamSupport jest sprzedawany w modelu subskrypcji i jej cena uzależniona jest od liczby użytkowników. System kierowany jest do małych i średnich przedsiębiorstw, natomiast model biznesowy może być zniechęcający dla tych firm, które nie chcą ponosić stałych, miesięcznych kosztów oraz mają mniejsze wymagania co do funkcjonalności. Interfejs aplikacji webowej zaprezentowany jest na Rysunku 1.




*Rysunek 1. Interfejs aplikacji TeamSupport. Źródło: opracowanie własne*

**Zalety:**

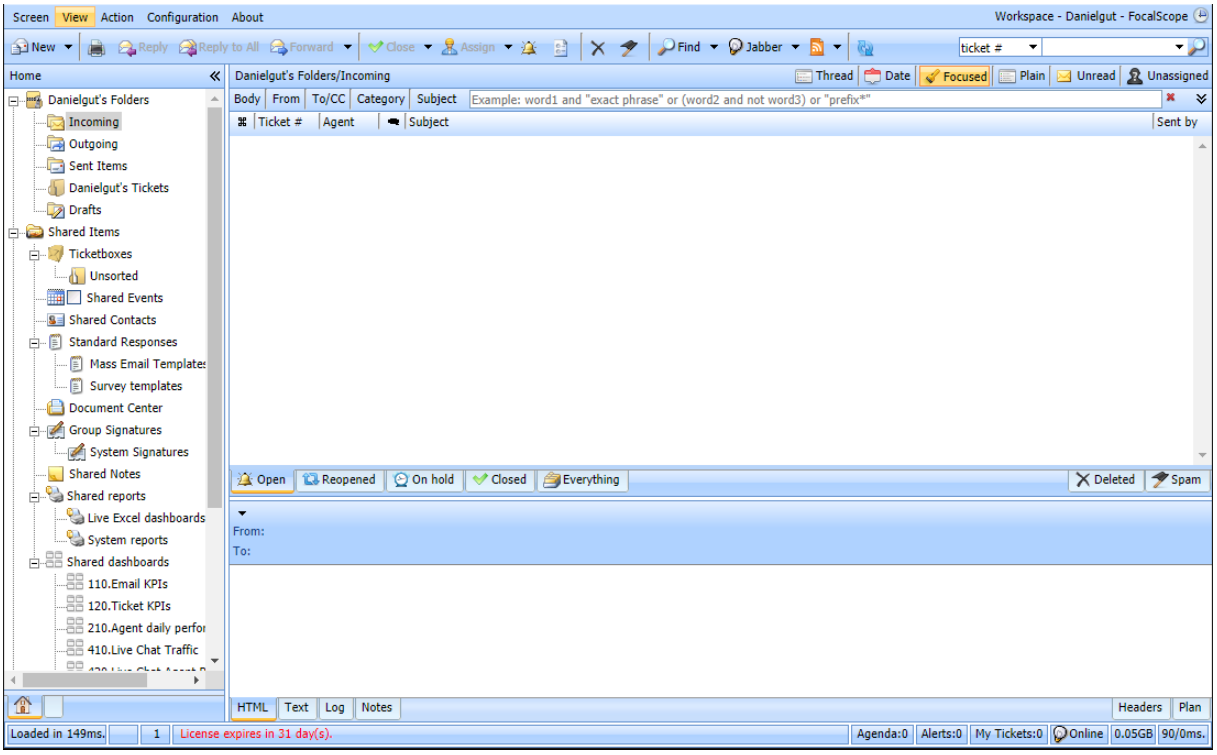
- Zaawansowane funkcje, takie jak czat na żywo, integracja ze skrzynką e-mail,
- Przejrzysty interfejs,
- Dobra dokumentacja.

**Wady:**

- Wersja demonstracyjna dostępna po zgłoszeniu się do producenta oprogramowania,
- Model subskrypcji płatnej miesięcznie, zależny od liczby użytkowników.

## 2.2. FocalScope

FocalScope (FocalScope, 2021) powstał w 2005 roku i jak twierdzi producent, oprogramowanie szybko stało się synonimem doskonałości. Rozwiązanie zostało oparte na chmurze i jest dostępne w płatnej subskrypcji. Podobnie jak w przypadku rozwiązania TeamSupport cena uzależniona jest od dostępnych funkcji oraz liczby użytkowników. Sprawy są zgłaszane za pomocą e-maila lub połączenia telefonicznego przez klientów, co automatycznie tworzy zadanie w systemie. Niestety nie ma możliwości tworzenia zadań niepowiązanych z wiadomością e-mail lub zgłoszeniem telefonicznym. Dodatkowo aplikacja umożliwia generowanie raportów według gotowych szablonów. Interfejs użytkownika jest czytelny, lecz dosyć przestarzały jak na dzisiejsze standardy. Na pochwałę zasługuje obszerna dokumentacja dla administratora systemu oraz użytkowników. Oprócz wersji webowej aplikacji producent oferuje również wersję mobilną na urządzenia z systemem iOS oraz Android. System przeznaczony jest zarówno dla małych, jak i dużych firm. Rysunek 2. przedstawia interfejs aplikacji.




*Rysunek 2. Interfejs aplikacji FocalScope. Źródło: opracowanie własne*

**Zalety:**

- Rozwiązanie oparte na chmurze,
- Aplikacją webowa oraz mobilna,
- Dobra dokumentacja dla administratora oraz użytkownika.

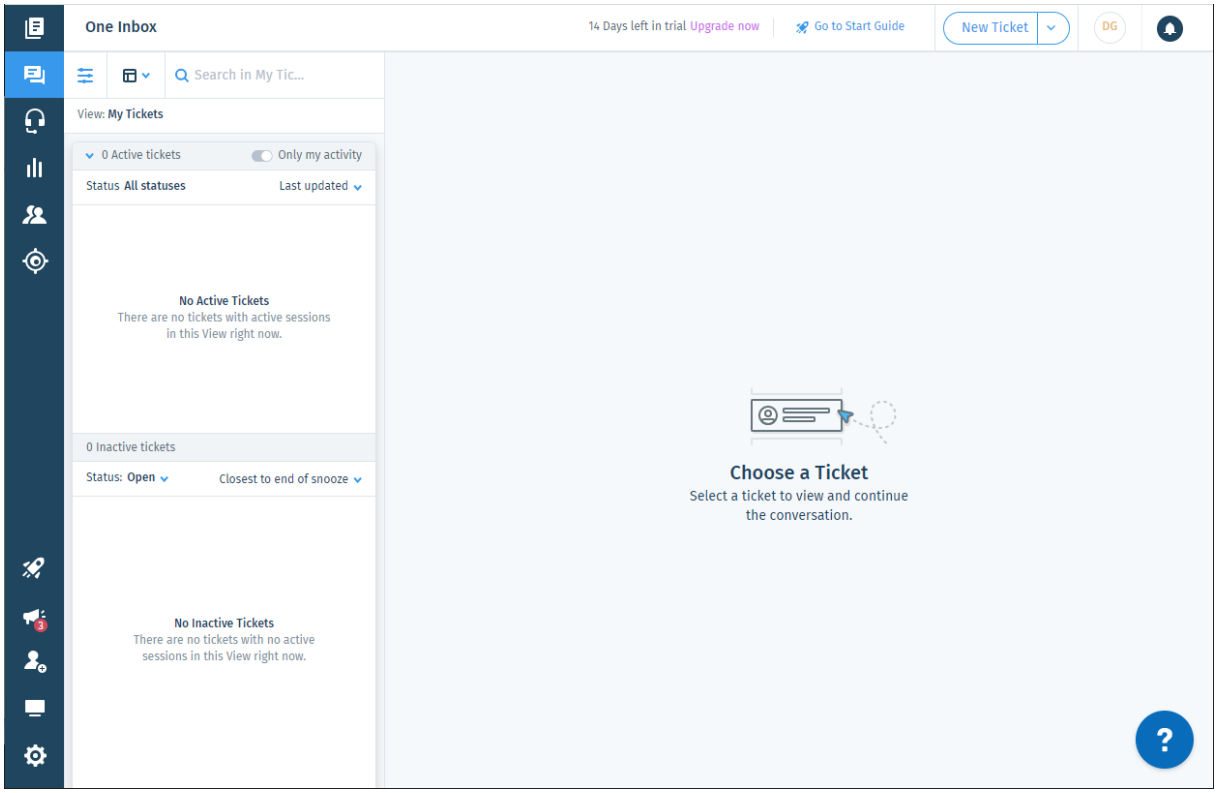
**Wady:**

- Przestarzały interfejs użytkownika,
- Plan sprzedażowy w postaci miesięcznej subskrypcji zależnej od liczby użytkowników,
- Najtańszy plan pozwala na podpięcie tylko dwóch adresów e-mail.

## 2.3. Wix Answers

Wix Answers (Wix Answers, 2021) jest rozwiązaniem zaproponowanym przez firmę Wix.com Ltd. Izraelski producent oprogramowania najbardziej słynie ze swojego narzędzia do budowania stron internetowych metodą „przeciągnij i upuść”, dzięki której użytkownicy nie mający podstaw programowania mogą w kilka minut stworzyć swoją własną witrynę internetową. Dokładnie taka sama filozofia przyświeca Wix Answers. Oprócz podstawowych wymaganych funkcji jak automatyczne powiązywanie spraw klientów, na przykład na postawie rozpoczętej konwersacji przez klienta, ciekawym rozszerzeniem oferty jest możliwość utworzenia portalu z bazą wiedzy. Wygląd portalu można w łatwy sposób personalizować metodą „przeciągnij i upuść”. Do wspomnianej strony można dodawać artykuł z gotowymi rozwiązaniami problemów oraz informacje z danymi kontaktowymi do działu wsparcia. Podobnie jak w poprzednich rozwiązaniach, aby w pełni korzystać z

dostępnych funkcji należy wykupić subskrypcję. Rysunek 3. przedstawia interfejs aplikacji webowej.




*Rysunek 3. Interfejs aplikacji Wix Answers. Źródło: opracowanie własne*

**Zalety:**

- Przyjazny interfejs,
- Personalizowany portal z artykułami na temat rozwiązanych problemów,
- Czat na żywo,
- Statystyki,
- Szybka i prosta konfiguracja.

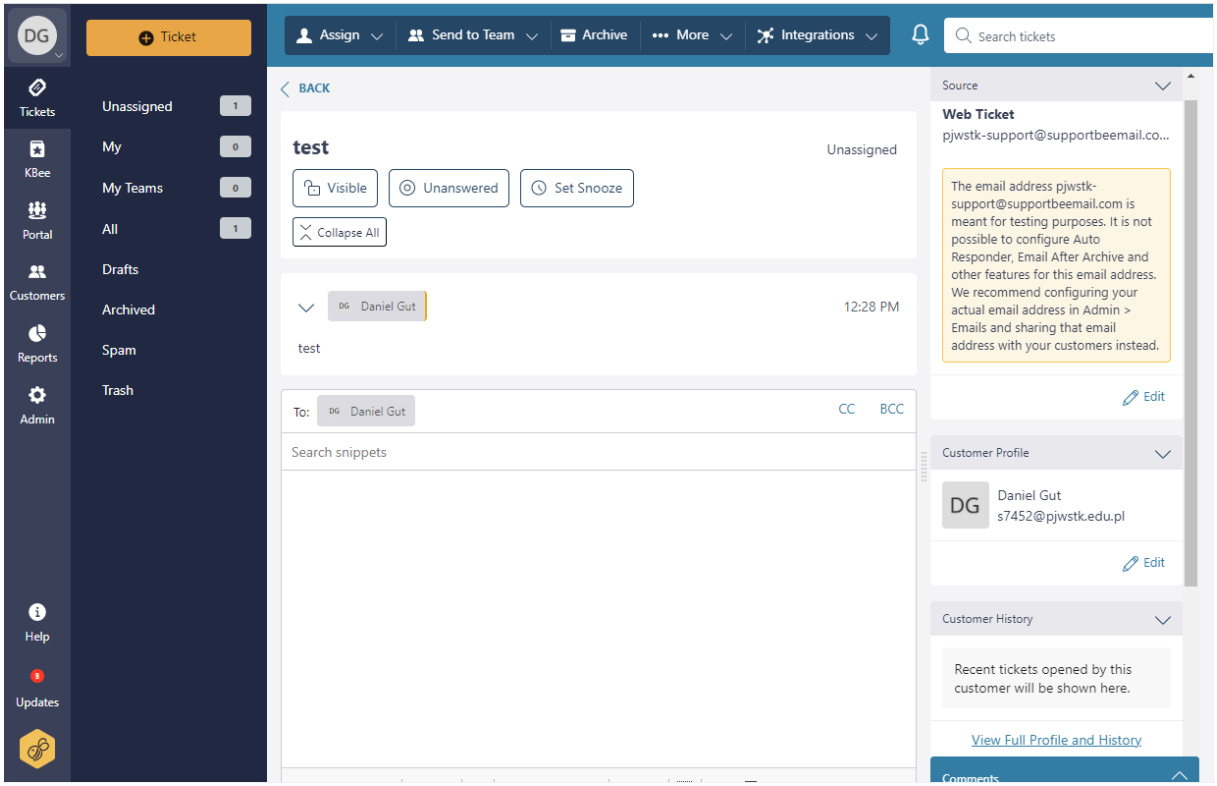
**Wady:**

- Miesięczna subskrypcja w cenie zależnej od liczby użytkowników.

## 2.4. SupportBee

Rozwiązanie SupportBee (SupportBee, 2021) jest podobne do opisywanego wcześniej Wix Answers. W pakiecie otrzymujemy narzędzie do tworzenia portalu, na którym klient może poznać aktualny status swojego zgłoszenia oraz prowadzić konwersację z pracownikiem działu wsparcia. Jest jednak rozwiązaniem prostszym gdyż nie posiada takich funkcji, jak czat na żywo, czy też integracja z centralą telefoniczną. Interfejs jest czytelny i zgodny z dzisiejszymi standardami. Podobnie jak pozostałe aplikacje, SupportBee jest sprzedawany w modelu płatnej miesięcznej subskrypcji w zależności od liczby użytkowników, natomiast jest znacznie tańszy od swojej konkurencji zważywszy na fakt, że jest rozwiązaniem o mniejszych możliwościach. Aplikacja kierowana jest do małych, jak i dużych firm. Szczególne zainteresowanie powinno wzbudzić u mniejszych przedsiębiorców,

którzy szukają prostego systemu z przystępnej cenie. Na Rysunku 4. przedstawiony jest interfejs aplikacji.




*Rysunek 4. Interfejs aplikacji SupportBee. Źródło: opracowanie własne*

**Zalety:**

- Intuicyjny interfejs,
- Portal dla klientów,
- Raporty,
- Dobra propozycja dla klientów poszukujących mniej rozbudowanego rozwiązania.

**Wady:**

- Miesięczna subskrypcja w cenie zależnej od liczby użytkowników,
- Mało rozbudowana na tle konkurencji.

## 2.5. Podsumowanie istniejących rozwiązań

Poprzedni rozdział omawiał krótko wybrane rozwiązania dostępne na rynku. Każde z nich ma swoje wady oraz zalety. Część z nich była bardziej lub mniej rozbudowana pod kątem oferowanych funkcji, natomiast cechą wspólną wszystkich czterech było automatyczne powiązywanie przysyłanych zgłoszeń z wiadomością e-mail. Wszystkie poza jedną charakteryzowały się nowoczesnym interfejsem użytkownika. Przeanalizowane, dostępne na rynku aplikacje są uniwersalnym rozwiązaniem dla każdego przedsiębiorstwa, co nie musi być zaletą w przypadku firm szukających zindywidualizowanych funkcjonalności.

# 3. Koncepcja i funkcjonalność systemu

Niniejszy rozdział przedstawia koncepcję autorskiego systemu na podstawie analizy istniejących rozwiązań. Zaprezentowane zostaną założenia, wymagania funkcjonalne oraz niefunkcjonalne systemu. Ponadto koncepcja została uzupełniona o diagram klas oraz przypadki użycia.

## 3.1. Założenia

Tech Support ma być prostym narzędziem wspierającym pracę działu wsparcia technicznego. Celem nie jest stworzenie rozwiązania idealnego, a przedstawienie własnego podejścia do tematu ułatwienia pracy działu wparcia technicznego. System ma zapewniać podstawowe funkcje, takie jak wprowadzanie danych klienta, powiązanie klienta z konkretnym produktem na podstawie numeru seryjnego, czego brakowało konkurencyjnym rozwiązaniom. Dodatkowo ma umożliwiać tworzenie zgłoszeń powiązanych z konkretnym produktem. Istotne jest wygodne filtrowanie klientów, produktów oraz zgłoszeń po wybranych parametrach w celu szybszego ich wyszukania. Użytkownicy systemu powinni posiadać różne uprawnienia. Zwykły użytkownik powinien móc dodać klienta, produkt, zgłoszenie oraz notatki i załączniki do zgłoszenia, natomiast nie powinien mieć możliwości usuwania klientów, produktów oraz zgłoszeń, a jedynie możliwość ich edycji. Dodatkowo niezbędny jest podgląd dla każdego z użytkowników, aby w szybki sposób mogli zobaczyć wszystkie wprowadzone przez siebie zgłoszenia. Moderator oraz Administrator powinni móc dodatkowo usuwać klientów, produkty oraz zgłoszenia. Powinni również mieć pełne uprawnienia dodawania, edycji oraz usuwania urządzeń, kategorii i typów. Administrator w przyszłości powinien mieć możliwość zarządzania użytkownikami systemu, między innymi zarządzaniem ich rolą w systemie.

Proponowany system nie jest wersją ostateczną. Powinien być elastyczny do celów dalszej rozbudowy o kolejne funkcjonalności. W wersji podstawowej ma dostarczać interfejs webowy, a w przyszłości powinien mieć możliwość rozbudowy o aplikację mobilną, raporty, zaawansowane statystyki oraz portal dla klientów. W obecnych czasach interfejs webowy jest podstawowym narzędziem do interakcji z użytkownikiem.

## 3.2. Definicje

**Klient** – osoba zgłaszająca problem techniczny,
**Pracownik wsparcia technicznego** – osoba, która wprowadza klientów do systemu oraz rejestruje zgłoszenia,
**Użytkownik** – pracownik, moderator lub administrator wprowadzony do systemu,
**Rola** – określa poziom uprawnień do zasobów w systemie,
**Kategoria** – określa przynależność urządzenia do pewnej grupy np. Router, Switch,
**Typ** – określa przynależność urządzenia do grupy ze względu na miejsce zastosowania np. SMB (Small and Medium Bussines), ISP (Internet Service Provider),
**Produkt** – konkretne urządzenie posiadające właściciela (klienta) oraz numer seryjny,
**Urządzenie** – przedmiot określony kategorią oraz typem,

**Administrator** – użytkownik o najwyższych uprawnieniach oraz osoba odpowiedzialna za wdrożenie systemu.

## 3.3. Użytkownicy systemu

System ma dostarczać różny poziom uprawnień użytkownika bazujący na rolach. Trzy podstawowe dostępne role to użytkownik, moderator oraz administrator. Kolejno użytkownik to rola o najmniejszych uprawnieniach, a administrator to rola o uprawnieniach najwyższych.

Wprowadzenie ról jest niezbędne do zachowania porządku w systemie. Zapobiega błędnym lub niechcianym działaniom, które w konsekwencji mogą prowadzić do nieporozumień. Domyślną rolą zaraz po zarejestrowaniu nowego konta jest użytkownik.

### 3.3.1. Użytkownik

Rola będzie zazwyczaj przypisana do pracownika wsparcia technicznego. Funkcjonalności dostępne dla roli to:

- Dodawanie klienta,
- Edycja klienta,
- Dodawanie produktu,
- Dodawanie zgłoszenia,
- Edycja zgłoszenia,
- Dodawanie notatek do zgłoszenia,
- Dodawanie załączników do zgłoszenia,
- Przeglądanie zgłoszeń, klientów oraz produktów.

### 3.3.2. Moderator

Osobą z tymi uprawnieniami będzie zazwyczaj koordynator zespołu lub zastępca kierownika. Oprócz wszystkich uprawnień, które posiada zwykły użytkownik, moderator może:

- Usuwać klientów,
- Usuwać produkty,
- Usuwać urządzenia, kategorie oraz typy,
- Dodawać urządzenia, kategorie oraz typy.

### 3.3.3. Administrator

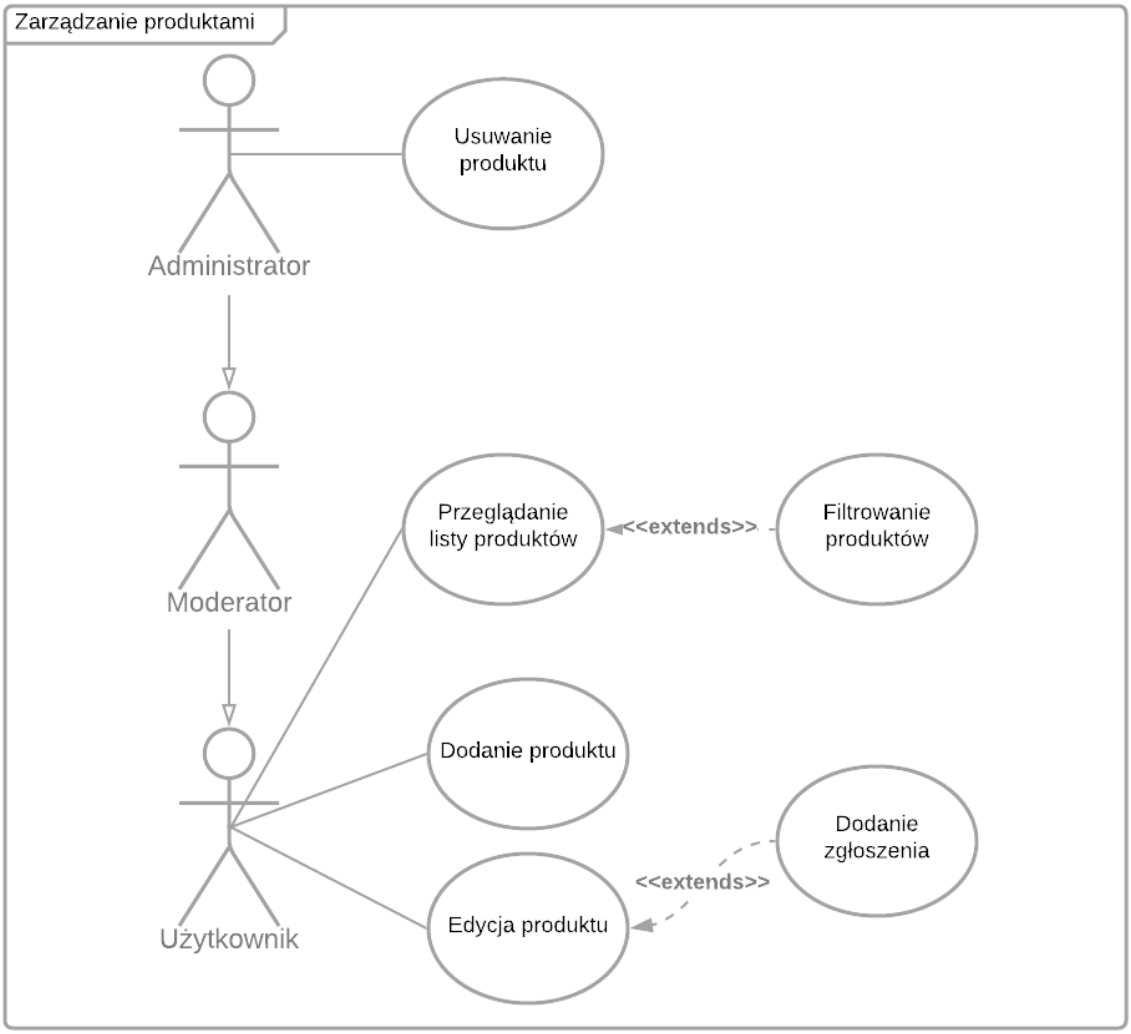
Tę rolę będą posiadali kierownik oraz osoba odpowiedzialna za wdrożenie systemu. Oprócz uprawnień dostępnych dla zwykłego użytkownika oraz moderatora, administrator może:

- Usuwać użytkowników,
- Edytować role użytkowników.

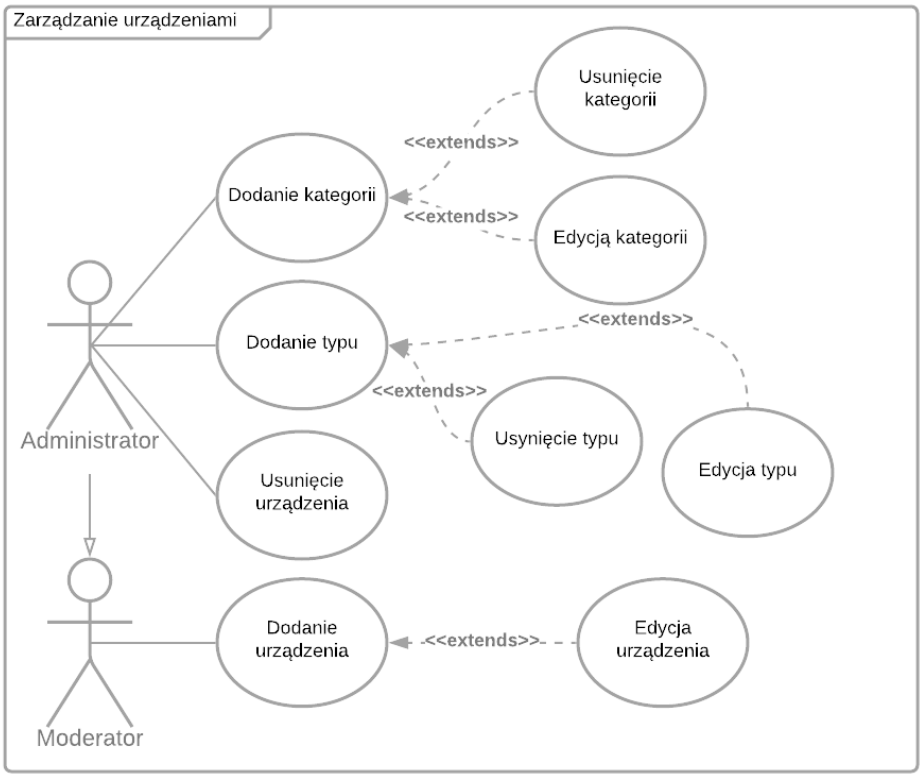
## 3.4. Przypadki użycia

Prototyp aplikacji przewiduje wiele różnych przypadków użycia w obszarach zarządzania klientami, produktami, urządzeniami, zgłoszeniami klientów oraz użytkownikami systemu.

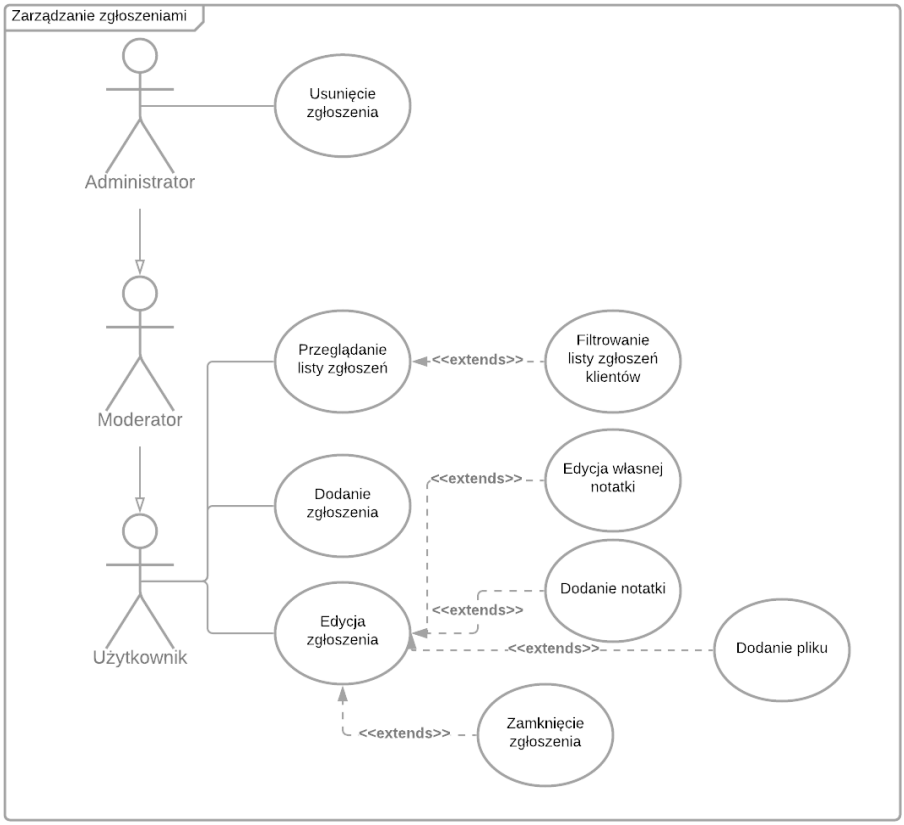
Poniżej, na Rysunkach od 5. do 9. zaprezentowane są diagramy przypadków użycia dla poszczególnych modułów, będące graficznym przedstawieniem interakcji między użytkownikami a systemem.



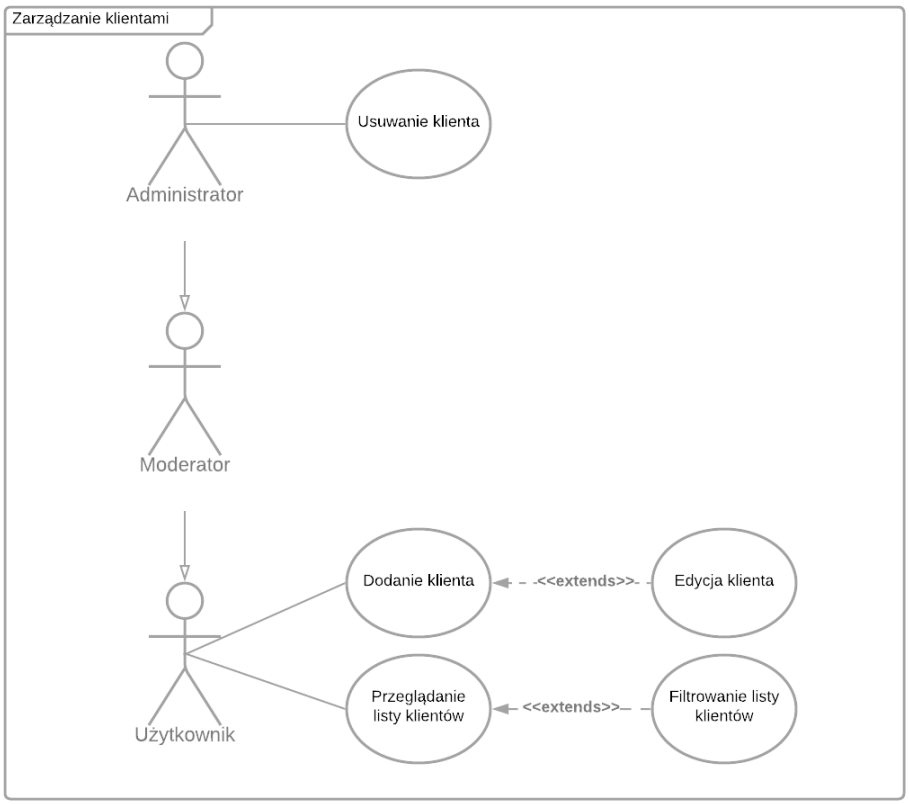

*Rysunek 5. Diagram przypadków użycia: zarządzanie produktami. Źródło: opracowanie własne*



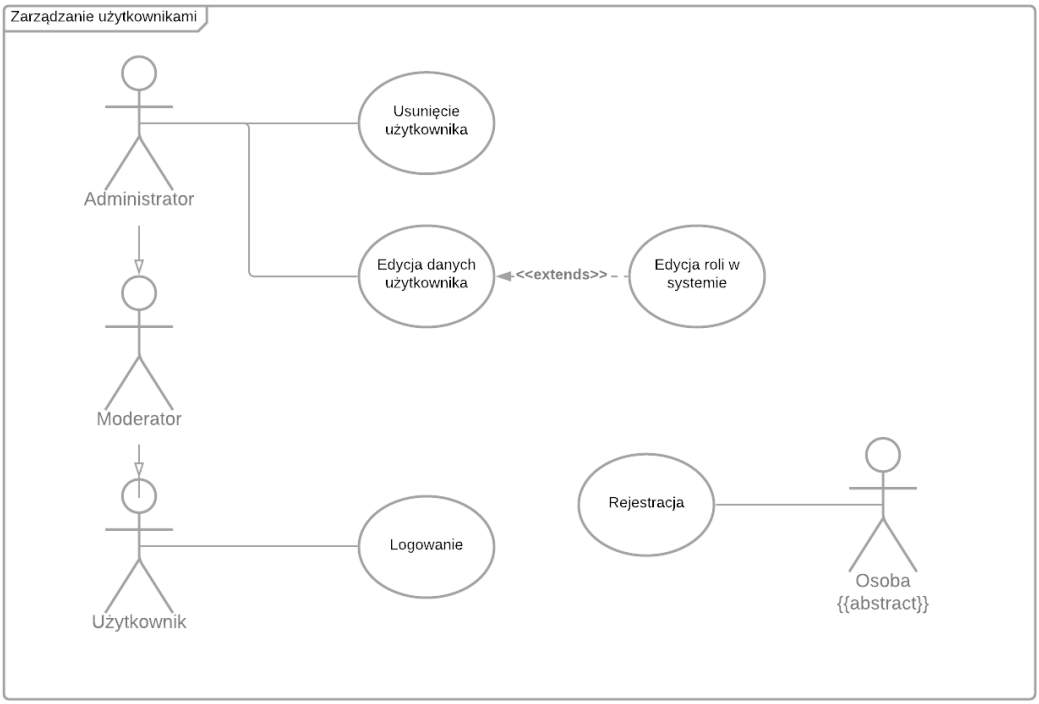

*Rysunek 6. Diagram przypadków użycia: zarządzanie urządzeniami. Źródło: opracowanie własne*




*Rysunek 7. Diagram przypadków użycia: zarządzanie zgłoszeniami. Źródło: opracowanie własne*




*Rysunek 8. Diagram przypadków użycia: zarządzanie klientami. Źródło: opracowanie własne*




*Rysunek 9. Diagram przypadków użycia: zarządzanie użytkownikami. Źródło: opracowanie własne*

### 3.4.1. Przykładowy scenariusz przypadku użycia na podstawie dodawania zgłoszenia

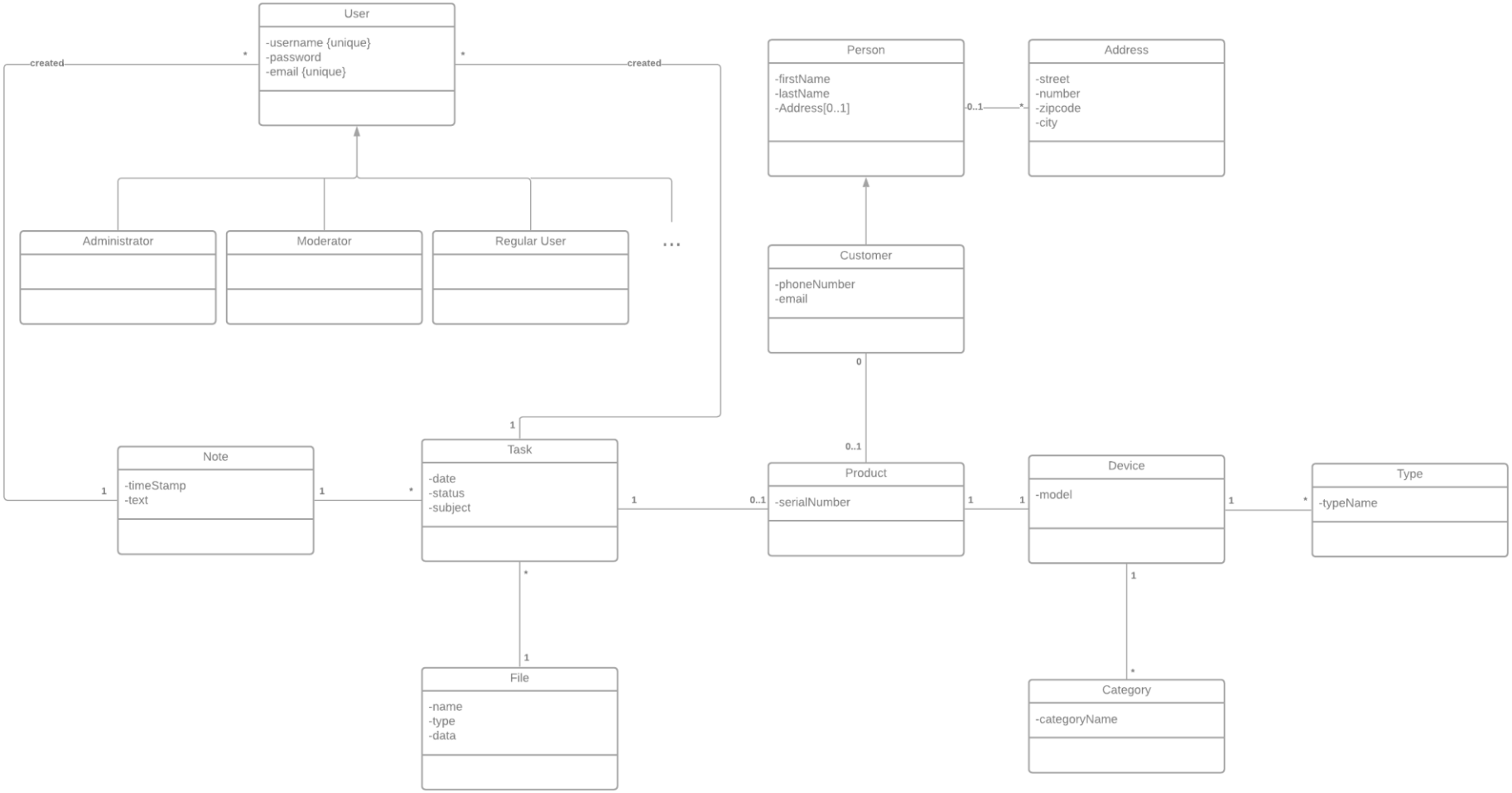
Przypadek użycia z dodawaniem zgłoszenia, którego scenariusz dla aktora Użytkownik zaprezentowany jest w Tabeli 1. jest jedną z kluczowych funkcji przewidzianych w ramach systemu. Poniżej zobaczymy główny oraz alternatywny przebieg zdarzenia.

*Tabela 1. Scenariusz przypadku użycia: Dodaj zgłoszenie. Źródło: opracowanie własne*

| **Dodaj zgłoszenie** | |
|---|---|
| Warunki początkowe | Użytkownik musi być zalogowany |
| Główny przebieg | 1. Użytkownik wyszukuje, czy produkt klienta jest dostępny w bazie,<br>2. Użytkownik przechodzi do edycji produktu,<br>3. System wyświetla informacje na temat produktu i właściciela,<br>4. Użytkownik przechodzi do formularza dodawania zgłoszenia,<br>5. System dodaje zgłoszenie, które pojawia się na liście zgłoszeń przypisanych do tego produktu,<br>6. Użytkownik przechodzi do zgłoszenia,<br>7. System wyświetla szczegóły dotyczące zgłoszenia,<br>8. Zgłoszenie dostaje status *pending (w toku)*, |
| Alternatywny przebieg | 1a. Jeżeli produkt nie jest dostępny w bazie to aktor przechodzi do edycji klienta, a następnie do utworzenia produktu,<br>1aa. Jeżeli klient nie istnieje w bazie, Aktor przechodzi do dodawania klienta,<br>1aaa Aktor przechodzi do dodania produktu,<br>1aaaa. System dodaje produkt i wyświetla go na liście produktów klienta,<br>1aaaaa. Aktor przechodzi do edycji produktu klienta,<br>8a. Jeżeli problem rozwiązany jest natychmiast, użytkownik zamyka sprawę i status zmienia się na *closed (zamknięta)*. |

## 3.5. Diagram klas

Rysunek 10. przedstawia diagram klas reprezentujący logikę aplikacji, relację między poszczególnymi klasami oraz atrybuty w nich zawarte.




*Rysunek 10. Diagram klas. Źródło: opracowanie własne*

## 3.6. Wymagania funkcjonalne

Na podstawie analizy istniejących systemów oraz własnych przemyśleń można oszacować, jakie funkcję będą kluczowe do stworzenia proponowanego prototypu. Efektem tej analizy jest Tabela 2.

*Tabela 2. Wymagania funkcjonalne. Źródło: opracowanie własne*

| Lp. | Nazwa wymagania | Opis | Aktorzy | Ograniczenia |
|---|---|---|---|---|
| 1.1 | Dodaj klienta | Użytkownik dodaje klienta (imię, nazwisko, telefon, email, adres) | Każdy użytkownik systemu | Brak |
| 1.2 | Edytuj dane klienta | Użytkownik edytuje dane klienta | Każdy użytkownik systemu | Klient znajduje się w bazie danych |
| 1.3 | Usuń klienta | Administrator usuwa danego klienta z bazy | Administrator | Klient znajduje się w bazie danych |
| 1.4 | Wyświetl listę klientów | Wyświetla listę wszystkich klientów | Każdy użytkownik systemu | Brak |
| 1.5 | Filtruj listę klientów | Filtruje listę wszystkich klientów po różnych atrybutach | Każdy użytkownik systemu | Brak |
| 2.1 | Dodaj produkt | Użytkownik dodaje produkt (numer seryjny oraz urządzenie), może powiązać produkt z klientem | Każdy użytkownik systemu | Brak |
| 2.2 | Edytuj produkt | Użytkownik edytuje dane produktu | Każdy użytkownik systemu | Produkt znajduje się w bazie danych |
| 2.3 | Usuń produkt | Administrator usuwa dane produktu | Administrator | Produkt znajduje się w bazie danych |
| 2.4 | Wyświetl listę produktów | Wyświetla listę wszystkich produktów | Każdy użytkownik systemu | Brak |

| | | | | |
|---|---|---|---|---|
| 2.5 | Filtruj listę produktów | Filtruje listę wszystkich produktów po różnych atrybutach | Każdy użytkownik systemu | Brak |
| 3.1 | Dodaj zgłoszenie | Użytkownik dodaje zgłoszenie (temat zgłoszenia), może powiązać zgłoszenie z klientem. Domyślny status zgłoszenia to *pending (w toku)* | Każdy użytkownik systemu | Brak |
| 3.2 | Edytuj zgłoszenie | Użytkownik edytuje dane zgłoszenia(temat zgłoszenia), dodatkowo edytowane zgłoszenie może zostań zamknięte | Każdy użytkownik systemu | Zgłoszenie znajduje się w bazie danych |
| 3.3 | Usuń zgłoszenie | Administrator usuwa dane zgłoszenie z bazy danych | Administrator | Zgłoszenie znajduje się w bazie danych |
| 3.4 | Wyświetl listę wszystkich zgłoszeń | Wyświetla listę wszystkich zgłoszeń w systemie | Każdy użytkownik systemu | Brak |
| 3.5 | Wyświetl listę zgłoszeń przez zalogowanego użytkownika | Wyświetla listę wszystkich zgłoszeń w systemie wprowadzonych przez zalogowanego użytkownika | Użytkownik | Brak |
| 3.6 | Filtruj listę zgłoszeń | Filtruje listę wszystkich zgłoszeń po różnych atrybutach | Każdy użytkownik systemu | Brak |
| 4.1 | Dodaj urządzenie | Administrator albo Moderator dodaje urządzenie (model) i przyporządkowuje go do typu oraz kategorii | Administrator Moderator | Brak |
| 4.2 | Edytuj urządzenie | Administrator albo Moderator edytuje dane urządzenia (model) | Administrator Moderator | Urządzenie znajduje się w bazie danych |

| | | | | |
|---|---|---|---|---|
| 4.3 | Usuń urządzenie | Administrator usuwa dane urządzenie z bazy danych | Administrator | Urządzenie znajduje się w bazie danych |
| 4.4 | Wyświetl listę urządzeń | Wyświetla listę wszystkich urządzeń | Administrator Moderator | Brak |
| 4.5 | Filtruj listę urządzeń | Filtruje listę wszystkich urządzeń po różnych atrybutach | Administrator Moderator | Brak |
| 5.1 | Dodaj kategorię | Administrator dodaje kategorię (nazwa kategorii) | Administrator | Brak |
| 5.2 | Edytuj kategorię | Administrator edytuje dane kategorii (nazwa kategorii) | Administrator | Kategoria znajduje się w bazie danych |
| 5.3 | Usuń kategorię | Administrator usuwa daną kategorię z bazy danych | Administrator | Kategoria znajduje się w bazie danych |
| 5.4 | Wyświetl listę kategorii | Wyświetla listę wszystkich kategorii | Administrator | Brak |
| 5.5 | Filtruj listę kategorii | Filtruje listę wszystkich kategorii po nazwie | Administrator | Brak |
| 6.1 | Dodaj typ | Administrator dodaje typ (nazwa typu) | Administrator | Brak |
| 6.2 | Edytuj typ | Administrator edytuje dane typu (nazwa typu) | Administrator | Typ znajduje się w bazie danych |
| 6.3 | Usuń typ | Administrator usuwa daną kategorię z bazy danych | Administrator | Typ znajduje się w bazie danych |
| 6.4 | Wyświetl listę typów | Wyświetla listę wszystkich typów | Administrator | Brak |
| 6.5 | Filtruj listę typów | Filtruje listę wszystkich typów po nazwie | Administrator | Brak |
| 7.1 | Zarejestruj | Pozwala na zarejestrowanie nowego użytkownika z domyślą rolą *user* (użytkownik) | Każdy użytkownik systemu | Brak |

| | | | | |
|---|---|---|---|---|
| 7.2 | Zaloguj | Pozwala na zalogowanie się użytkownikowi za pomocą loginu i hasła | Każdy użytkownik systemu | Użytkownik znajduje się w bazie danych |
| 7.3 | Wyświetl listę użytkowników | Wyświetla listę wszystkich użytkowników w systemie | Administrator | Brak |
| 7.4 | Edytuj użytkownika | Administrator edytuje dane użytkownika | Administrator | Użytkownik znajduje się w bazie danych |
| 7.5 | Edytuj role użytkownika | Pozwala Administratorowi na edycje uprawnień użytkownika | Administrator | Użytkownik znajduje się w bazie danych |

## 3.7. Wymagania niefunkcjonalne

Wymagania niefunkcjonalne stanowią zbiór cech opisujących takie aspekty oprogramowania jak niezawodność, bezpieczeństwo, czy efektywność. Mogą mieć wpływ na wybór technologii, biorąc pod uwagę na przykład liczbę użytkowników czy poziom zabezpieczenia. Tabela 3. przedstawia zbiór wymagań niefunkcjonalnych w ramach realizowanego projektu.

*Tabela 3. Wymagania niefunkcjonalne. Źródło: opracowanie własne*

| **Lp.** | **Opis wymagania niefunkcjonalnego** |
|---|---|
| 1 | System powinien dawać możliwość wdrożenia na systemach operacyjnych Linux, Windows, MacOs. |
| 2 | System powinien być dostępny dla użytkownika dopiero po zalogowaniu, uwierzytelniając się nazwą użytkownika i hasłem. |
| 3 | System powinien mieć kilka poziomów autoryzacji w celu kontroli dostępu do zasobów. |
| 4 | Interfejs powinien być prosty i intuicyjny. |
| 5 | Aplikacja webowa powinna działać na najpopularniejszych przeglądarkach internetowych (Google Chrome, Firefox, Safari, Edge). |
| 6 | System powinien zapewniać bezawaryjność przez 90 dni. |
| 7 | Użytkownicy powinni mieć dostęp do systemu w godzinach 8:00-21:00. |

| | |
|---|---|
| 8 | System powinien móc obsłużyć do 50 użytkowników jednocześnie. |
| 9 | Użytkownik powinien móc samodzielnie obsługiwać system po 1 godzinie szkolenia. |
| 10 | System powinien być elastyczny do dalszej rozbudowy funkcjonalności oraz obsługiwanych platform, na przykład w formie aplikacji mobilnej. |
| 11 | System powinien być wykonany w językach Java, JavaScript, SQL. |

# 4. Wybór rozwiązań technologicznych

Rozdział ten ma na celu omówienie głównych technologii wykorzystanych do stworzenia prototypu systemu.

## 4.1. Spring Framework

Spring Framework (Spring Framework, 2021) to jeden z najpopularniejszych frameworków języka Java. Jest to platforma typu Open Source używana do tworzenia aplikacji biznesowych, której pierwsza wersja produkcyjna została wydana 24 Marca 2004 roku. Bardzo szybko zyskał uznanie wśród programistów Enterprise Java Beans (EJB), między innymi dzięki swojemu rdzennemu wzorcowi Dependency Injection (DI) znanemu również jako Inversion of Control (IoC). Jest to wzorzec projektowy, który ma na celu usunięcie bezpośrednich powiązań między poszczególnymi komponentami metodą wstrzykiwania zależności. Jest to proces w którym zależności zdefiniowane są na przykład jako argumenty konstruktora i dowiązane zaraz po utworzeniu instancji obiektu. (Çalıs¸kan i Sevindik, 2015)

Spring Framework charakteryzuje się następującymi cechami:

- Główne technologie to wstrzykiwanie zależności, wydarzenia, zasoby, i18n, walidacja, wiązanie danych, konwersja typów, SpEL, programowanie aspektowe,
- Testowanie z wykorzystaniem atrap obiektów (mock object), Spring MVC Test, TestContext framework,
- Dostęp do danych wspiera transakcje, Java DataBase Connectivity (JDBC), Object-Relation Mapping (ORM), Data Access Object (DAO)
- Frameworki webowe Spring MVC oraz Spring WebFlux,
- Języki takie jak Java, Kotli czy Groovy.

Spring Boot, który wykorzystany został do stworzenia prototypu systemu jest niejako rozszerzeniem dla Spring Framework, który w znaczny sposób ułatwia i skraca proces konfiguracji. Dysponuje szeregiem bibliotek startowych, zmniejszających ilość zależności, które musimy dodać do pliku pom.xml lub build.gradle w zależności od tego, z którego narzędzia automatyzującego budowę aplikacji korzystamy. Poniżej przykład zależności w przypadku tworzenia aplikacji webowej w Spring Framework oraz Spring Boot dla porównania.

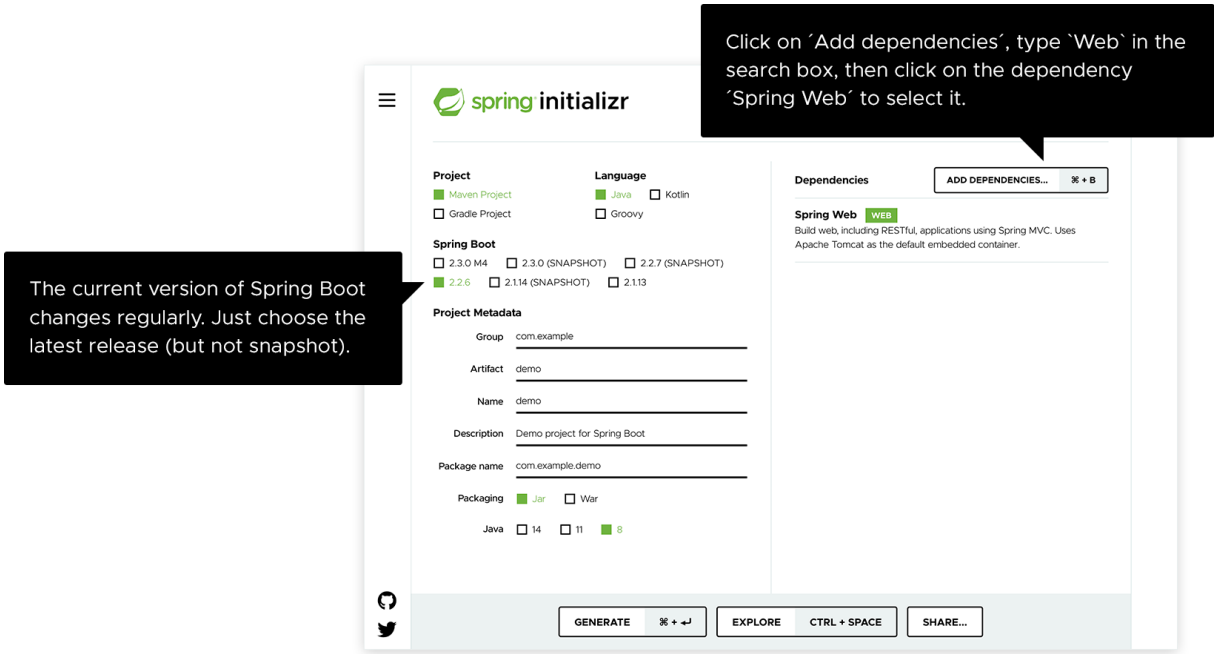
*Listing 1. Przykład podstawowych zależności w pliku pom.xml dla projektu Spring Framework*

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
    <version>5.3.7</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.3.7</version>
</dependency>
```

*Listing 2. Przykład podstawowych zależności w pliku pom.xml dla projektu Spring Boot*

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Dodatkowym atutem jest również wbudowany serwer Tomcat Apache oraz inicjalizator dostępny pod adresem http://start.spring.io, który dodatkowo pozwala pominąć czas poświęcony na ręczne konfigurowanie. Inicjalizator pozwala wybrać język w jakim będzie pisany projekt, wersję Spring Boot, narzędzie automatyzujące budowę oprogramowania (Maven lub Gradle), metadane oraz zależności. W efekcie otrzymujemy paczkę zip z wygenerowanym szkieletem projektu. Interfejs narzędzia Spring Initializr przedstawiony jest na Rysunku 11.




*Rysunek 11. Interfejs narzędzia Spring Initializr. Źródło: (Spring Quick Start, 2021)*

Punktem wyjściowym aplikacji Spring Boot jest klasa z adnotacją *@SpringBootApplication* i ma postać jak na Listingu 3.

*Listing 3. Główna klasa uruchamiająca aplikację Spring Boot*

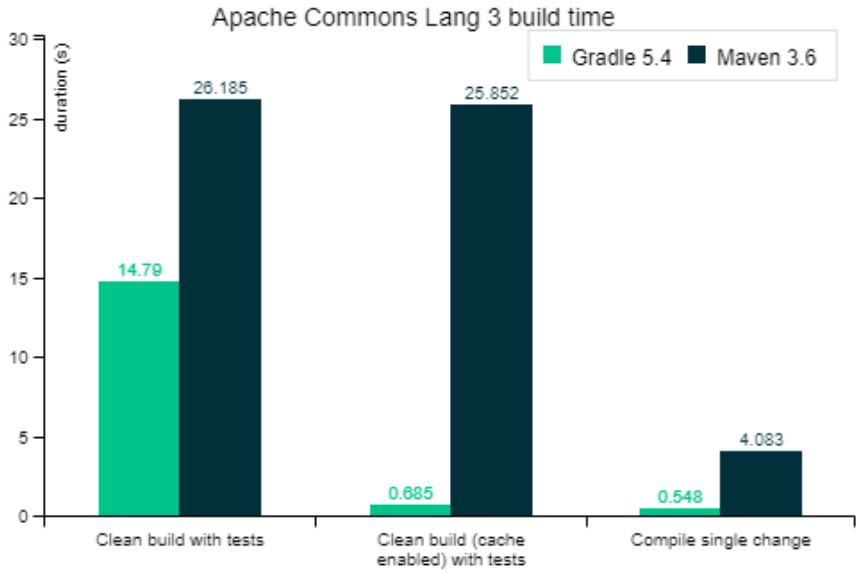
```
package com.example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

## 4.2. Gradle

Gradle (Gradle, 2021) jest drugim dostępnym oficjalnie wspieranym narzędziem do automatyzacji budowy aplikacji Spring Framework. W przeciwieństwie do narzędzia Maven, które jest prawdopodobnie częstszym wyborem programistów przy projektach w Spring Framework, ten nie korzysta z języka XML w tworzonym dokumencie określającym budowę aplikacji. Do tego celu wykorzystywany jest język DSL (domain-specific language), który charakteryzuje się zwięzłością oraz lepszą przejrzystością. Gradle operuje na zadaniach, które definiują listę operacji do przeprowadzenia. Przykładowym zadaniem może być uruchomienie testów. Dodatkowo możemy określić zależności między zadaniami, które definiują kolejność w jakiej mają być one wykonywane. Wszystkie te informacje wprowadzamy do pliku bulid.gradle.

Gradle może również pochwalić się krótszym czasem budowania aplikacji o czym mówią przeprowadzone testy udostępnione na stronie projektu. Właśnie te dwa główne kryteria, czyli składnia DSL, która subiektywnie jest czytelniejsza oraz szybszy czas budowania aplikacji zadecydowały o wyborze Gradle jako narzędzia do automatyzacji budowy oprogramowania.

Wyniki testów czasu budowania aplikacji przez Gradle w wersji 5.4 oraz Maven w wersji 3.6. znajdują się na Rysunku numer 12.




*Rysunek 12. Wyniki testów czasu budowania aplikacji przez Gradle oraz Maven. Źródło: (Gradle - Maven vs Gradle, 2021)*

Przykładowy plik build.gradle wygenerowany przy pomocy narzędzia Spring Initalizr przedstawiono na Listingu 4.

*Listing 4. Przykładowy plik build.gradle wygenerowany przy pomocy narzędzia Spring Initializr*

```
plugins {
    id 'org.springframework.boot' version '2.5.0'
    id 'io.spring.dependency-management' version '1.0.11.RELEASE'
    id 'java'
}

group = 'com.example'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '11'

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}

test {
    useJUnitPlatform()
}
```

## 4.3. Java

Język programowania Java (Oracle, 2021) od wielu lat znajduje się w czołówce najbardziej popularnych języków programowania. W serwisie społecznościowym, na którym programiści mogą zadawać pytania w szerokim kontekście tworzenia oprogramowania, zajmuje 5 miejsce pod względem najczęściej używanych języków programowania w roku 2020. (Stack Overflow, 2021)

Java to obiektowy język programowania ogólnego przeznaczenia. Twórcą jest James Gosling, który rozpoczął projekt w czerwcu 1991 r. wraz z grupą programistów w firmie Sun Microsystems.

Główny założeniem języka jest prostota oraz intuicyjność. Prosta składnia oraz w pewnym stopniu podobieństwo do takich języków jak C, C++ mają na celu ułatwić programistom migrację na platformę Java. Przykład najprostszego programu napisanego w języku Java, którego wynikiem jest wypisany na ekranie konsoli napis *Hello, World!* prezentuje Listing 5.

*Listing 5. Przykład aplikacji HelloWorld*

```
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Programy napisane w języku Java nie są zależne od platformy. Kompilowane są do kodu bajtowego, a następnie wykonywane przez maszynę wirtualną (JVM). Maszyna wirtualna posiada takie mechanizmy jak obsługa wyjątków, czy odśmiecanie pamięci (ang. garbage collection). Pozwala to na zdjęcie z programisty odpowiedzialności za zwalnianie przydzielonej pamięci. (Horstmann, 2016)

## 4.4. Angular

Angular (Angular Google, 2021) jest platformą programistyczną zbudowaną w oparciu o język TypeScript. Został stworzony przez firmę Google i zawiera następujące elementy:

- Oparty na komponentach framework do budowania skalowalnych aplikacji webowych,
- Kolekcje dobrze zintegrowanych bibliotek które pokrywają szeroki zakres funkcjonalności, takich jak routing, formularze do komunikacji klient-serwer i wiele więcej,
- Zestaw narzędzi pomocny przy tworzeniu, budowaniu czy też testowaniu aplikacji.

Główne idee stojące za Angularem to komponenty, szablony, wstrzykiwanie zależności. Komponenty to „cegiełki”, które składają się na budowę aplikacji. Komponent zawiera między innymi klasę TypeScript oznaczoną adnotacją *@Component()*, szablon HTML oraz arkusz stylu (na przykład CSS). Adnotacja *@Component()* określa takie informacje jak:

- Selektor CSS, który definiuje jak komponent jest używany w szablonie. Elementy HTML pasujące do selektora stają się instancją komponentu,
- Szablon HTML, który jest instrukcją jak ma zostać renderowany komponent,
- Opcjonalnie arkusz styli CSS.

Na Listingu 6. zaprezentowano przykładowy minimalistyczny komponent, który po wywołaniu w szablonie wyświetla napis *Example App*.

*Listing 6. Przykładowy minimalistyczny komponent we frameworku Angular. Źródło: opracowanie własne*

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: `
    <h2>Example App</h2>
    `,
})
export class MyAppComponent {

}
```

Aby użyć powyższego komponentu należy się do niego odwołać w szablonie w sposób przedstawiony na Listingu 7.

*Listing 7. Przykład użycia komponentu w szablonie. Źródło: opracowanie własne*

```
<my-app></my-app>
```

Szablony w Angularze są to rozszerzone o dodatkową składnię pliki HTML, które umożliwiają dynamiczne wstawianie wartości z komponentów. W ten sposób posiadając w komponencie tablicę obiektów, na przykład klientów, w łatwy sposób można wyświetlić ich listę. Przykład kodu, który umożliwia taką operację znajduje się w Listingu 8.

*Listing 8. Przykład pętli w szablonie Angular wypisującej imiona klientów w postaci listy. Źródło: opracowanie własne*

```
<ul>
    <li *ngFor="let c of customers">
      {{c.name}}
    </li>
</ul>
```

## 4.5. TypeScript

TypeScript (Microsoft TypeScript, 2021) został stworzony przez firmę Microsoft w roku 2012 i jest to język programowania będący nadzbiorem dla języka Java Script. Działający kod napisany w języku Java Script będzie również działał dla TypeScript. Na tle języka Java Script wyróżnia się między innymi opcjonalnym statycznym typowaniem. W Java Script mimo dostarczenia typów prymitywnych jak *string* czy *number* do zmiennej, mechanizmy języka nie sprawdzają konsekwentnie czy zostały one do niej przypisane. TypeScript natomiast daje taką możliwość, co wpływa na większą kontrolę nad tworzonym kodem źródłowym.

Chcąc stworzyć obiekt możemy to zrobić w sposób, jaki umożliwia nam język Java Script i jest przedstawiony na Listingu 9.

*Listing 9. Przykładowy kod JS tworzący obiekt car. Źródło: opracowanie własne*

```
const car = {
    id: 0,
    brand: "Audi",
    model: "A6"
}
```

Możemy również skorzystać z interfejsów, które dostarczane są wraz z językiem TypeScript, aby później użyć go przy tworzeniu nowego obiektu. Zapewni to sprawdzanie poprawności dostępnych zmiennych oraz ich typów. Przykład takiego kodu znajduje się w Listingu 10 oraz Listingu 11.

*Listing 10. Przykład interfejsu stworzonego w języku TypeScript. Źródło: opracowanie własne*

```
interface Car {
    id: number;
    brand: string;
    model: string;
}
```

*Listing 11. Przykładowy kod deklarujący obiekt wraz z jego typem w TypeScript. Źródło: opracowanie własne*

```
const car: Car = {
    id: 0,
    brand: "Audi",
    model: "A6"
}
```

W przypadku niespełnienia warunków odnośnie poprawnej nazwy zmiennej lub użycia nieprawidłowego typu, zostaniemy o tym poinformowani komunikatem błędu.

## 4.6. Bootstrap

Boostrap (Bootstrap, 2021) jest biblioteką CSS stworzoną przez programistów znanego serwisu społecznościowego Twitter. Jest to zbiór szablonów CSS oraz opcjonalnie Java Script, stylizacji typografii, formularzy, przycisków oraz wielu innych komponentów składających się na interfejs przedstawiany użytkownikowi. W znaczny sposób upraszcza i przyspiesza proces tworzenia wizualnego aspektu aplikacji webowych. Zapewnia responsywność dla mobilnych wersji stron internetowych oraz rozwiązuje problem kompatybilności pomiędzy najpopularniejszymi przeglądarkami internetowymi. Odpowiednie cechy różnym elementom w dokumencie HTML nadajemy na przykład poprzez dodawanie do atrybutu *class* odpowiednich selektorów dostarczonych przez bibliotekę Bootstrap. Na Rysunku 13. zaprezentowana jest przykładowa lista z odznakami stylizowanymi przy pomocy Bootstrap, natomiast w Listingu 12. kod, który za to odpowiada.


*Rysunek 13. Przykład listy z odznakami. Źródło: (Bootstrap list-group, 2021)*

*Listing 12. Przykładowy kod tworzący listę z odznakami. Źródło: (Bootstrap list-group, 2021)*

```
<ul class="list-group">
    <li class="list-group-item d-flex justify-content-between align-items-center">
        A list item
        <span class="badge bg-primary rounded-pill">14</span>
    </li>
    <li class="list-group-item d-flex justify-content-between align-items-center">
        A second list item
        <span class="badge bg-primary rounded-pill">2</span>
    </li>
    <li class="list-group-item d-flex justify-content-between align-items-center">
      A third list item
      <span class="badge bg-primary rounded-pill">1</span>
    </li>
</ul>
```

## 4.7. MySQL

MySQL (MySQL Documentation, 2021) jest systemem zarządzania relacyjnymi bazami danych typu Open Source. Od roku 2010 jest w posiadaniu firmy Oracle, która odpowiada za jego dalszy rozwój. MySQL można opisać wymieniając jego kilka podstawowych cech:

- MySQL jest systemem bazy danych – baza danych jest pewnym zbiorem ustrukturyzowanych danych. Chcąc uzyskać dostęp do tych danych, dodać je lub usunąć, potrzebujemy systemu zarządzania bazą danych. Takim system jest właśnie MySQL Server,
- Bazy danych MySQL są relacyjne – taki model bazy danych pozwala na przechowywanie oraz dostęp do powiązanych ze sobą danych. Dane przedstawiane są w tabelach. Każdy wiersz w tabeli to rekord, który posiada unikatowy identyfikator. Kolumny natomiast to atrybuty,
- Open Source – oznacza, że każdy może pobrać oprogramowanie bezpłatnie ze strony producenta. Kod źródłowy jest otwarty, wiec można go analizować oraz modyfikować,
- MySQL Server jest szybki, niezawodny, skalowalny oraz łatwy w użyciu,
- Serwer MySQL działa w systemach klient – serwer lub systemach wbudowanych.

SQL (ang. Structured Query Language) to strukturalny język zapytań używany w systemie MySQL. Podstawowe operacje wykonywane na danych to:

- **INSERT** – polecenie dodaje dane do bazy danych,
- **UPDATE** – polecenie modyfikuje dane w bazie danych,
- **DELETE** – polecenie usuwa dane z bazy danych.

Przykład użycia jednej z powyższych funkcji znajduje się w Listingu 13.

*Listing 13. Przykład instrukcji INSERT.*

```
INSERT INTO Customers (firstName, lastName) VALUES ('Jan', 'Nowak');
```

Podstawowe operacje które możemy wykonać na strukturach danych (np. tabela) to:

- **CREATE –** polecenie tworzy nową strukturę,
- **DROP** – polecenie usuwa określoną strukturę,
- **ALTER** – polecenie modyfikuje daną strukturę.

Przykładowe polecenie dodające nową tabelę do bazy danych przedstawione jest w Listingu 14.

*Listing 14. Przykładowa instrukcja CREATE*

```
CREATE TABLE Customers (FirstName varchar(255), LastName varchar(255));
```

Jeżeli zamiast korzystania z instrukcji SQL wolimy interakcję z bazą danych za pomocą graficznego interfejsu, w pakiecie dostarczone jest oprogramowanie MySQL Workbench.

## 4.8. Architektura REST

REST (ang. Representational State Transfer) (Wstęp do REST API, 2021) jest stylem architektury oprogramowania używającym podzbioru protokołu HTTP. Pierwszy raz został zdefiniowany przed Roya Fieldinga w 2000 roku. REST mimo korzystania z takich technologii jak HTTP, URI czy JSON, sam w sobie nie jest protokołem, ani nie jest standardem. Definiuje natomiast kilka podstawowych zasad, które powinna spełniać aplikacja, aby móc ją nazwać aplikacją REST. (Bretet, 2016)

- Jednolity interfejs – poszczególne zasoby identyfikowane są poprzez żądania, np. poprzez URI. Zasoby po stronie serwera oddzielone są od reprezentacji wysyłanej do klienta, na przykład w postaci JSON. Po stronie serwera reprezentowane są w inny sposób. Klient mający reprezentację zasobu posiada wystarczającą ilość informacji, aby móc go usunąć lub zmodyfikować.
- Klient-Serwer – istnieje wyraźne rozdzielenie aplikacji działającej po stronie klienta oraz po stronie serwera, dzięki czemu każda z części architektury może rozwijać się niezależnie od drugiej.
- Bezstanowość – polega na tym, że serwer nie przechowuje informacji na temat stanu klienta, natomiast klient za każdym razem, gdy wysyła zapytanie do serwera dostarcza pakiet informacji, na podstawie których serwer może określić, czy klient ma dostęp do zasobów.
- Cachable – jeżeli jest to możliwe, klient ma prawo zapisania pewnych danych w pamięci podręcznej w celu późniejszego ich wykorzystania. Pomaga to w zwiększeniu wydajności.
- System warstwowy – system powinien być zaprojektowany w taki sposób, aby klient nie musiał wiedzieć, co znajduje się po drugiej stronie. Jeżeli między klientem a

serwerem znajduje się na przykład serwer proxy, to nie powinno mieć to wpływu na komunikację.

- Kod na żądanie – zasada opcjonalna, która polega na tym, że serwer może tymczasowo rozszerzyć funkcjonalność klienta poprzez przesłanie fragmentu kodu (np. Java Script) w celu wykonania go po stronie klienta.

Podstawowe metody HTTP używane podczas budowy API, wystarczające do operacji CRUD (ang. create, read, update, delete) to:

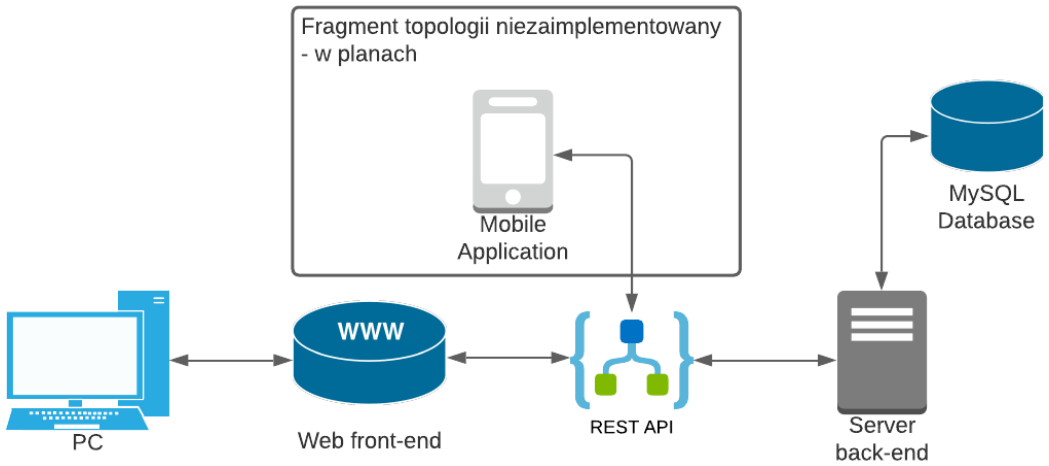
- **GET** – służy do pobierania reprezentacji zasobu,
- **POST** – pozwala zasobowi na przetworzenie reprezentacji zawartej w żądaniu,
- **PUT** – ustawia stan zasobu na stan zdefiniowany w reprezentacji zasobu w żądaniu,
- **DELETE** – usuwa stan zasobu.

# 5. Zagadnienia implementacyjne

W niniejszym rozdziale została opisana logika, architektura systemu oraz przebieg implementacji.

## 5.1. Architektura

Głównym założeniem podczas projektowania systemu była potrzeba ukierunkowania go na dalszy rozwój w kontekście nowych funkcjonalności oraz platform, na których ma działać. Rozwój poszczególnych części systemu nie mógł być zależny od pozostałych. Właśnie dlatego wybór padł na popularną dziś architekturę REST. Na rysunku 14. znajduje się graficzna reprezentacja logiki projektowanego systemu.



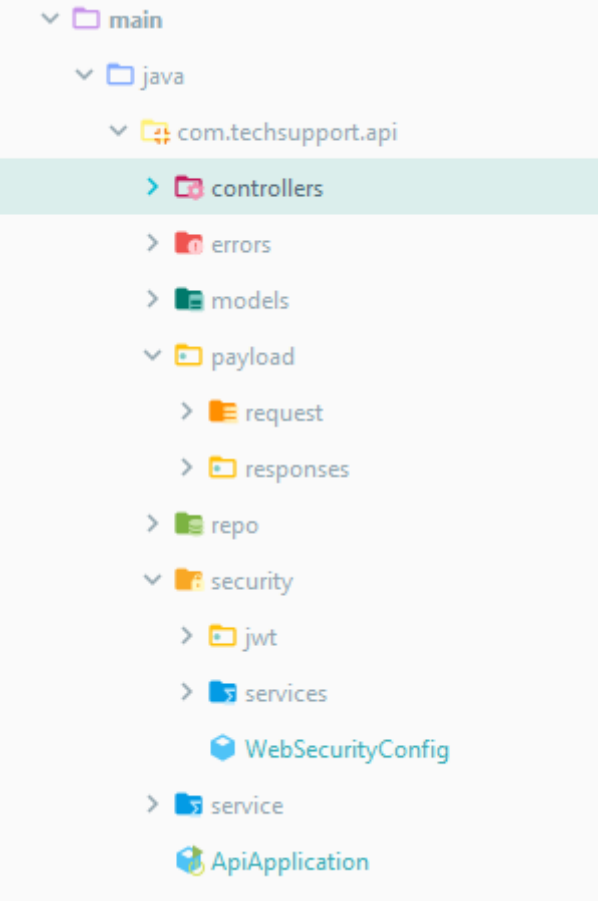

*Rysunek 14. Graficzna reprezentacja logiki systemu. Źródło: opracowanie własne*

Głównym rdzeniem aplikacji jest Back-end po stronie serwera, który definiuje całe API. Odpowiada za komunikację z bazą danych, definiuje dostępne endpointy, zależności pomiędzy klasami oraz bezpieczeństwo aplikacji. Drugim ważnym elementem aplikacji jest serwer, na którym znajduje się front-end, czyli warstwa systemu odpowiedzialna za interfejs

graficzny i komunikację z API. Jest to warstwa pośrednicząca między osobą obsługującą system a serwerem. W kolejnych rozdziałach zostaną szczegółowo opisane aspekty implementacji z podziałem na część związaną z back-endem oraz cześć związaną z front-endem.

## 5.2. Aplikacja serwerowa

Aplikacja serwerowa została napisana z wykorzystaniem frameworku Spring (Spring Framework, 2021) w języku Java. Ze względu na swoją długą obecność na rynku oferuje mnogość bibliotek, stabilność i bezpieczeństwo. Tworzenie aplikacji w takim frameworku znacznie przyspiesza cały proces powstawania prototypu. Dodatkowo tworzony jest przez zespół doświadczonych programistów, dzięki czemu mamy większą pewność co do bezpieczeństwa implementowanych w nim systemów. Rysunek 15. przedstawia strukturę katalogów w projekcie.




*Rysunek 15. Struktura katalogów w projekcie*

Podczas tworzenia aplikacji, bardzo ważne jest aby zachować pewną logikę oraz porządek. Umieszczając klasy o podobnym znaczeniu i działaniu do odpowiednich paczek sprawia, że w łatwy i intuicyjny sposób możemy poruszać się po strukturze projektu. W myśl tej zasady, klasy odpowiedzialne za komunikację między klientem a serwerem zostały umieszczone w folderze controllers, klasy odpowiadające zasobom w folderze models, klasy będące reprezentacją zapytań i odpowiedzi w folderze payload, klasy odpowiedzialne za

bezpieczeństwo w folderze security oraz klasy i interfejsy odpowiedzialne za komunikację z bazą danych w folderach repo i service. W kolejnych podrozdziałach zostały szerzej omówione poszczególne klasy oraz pakiety.

### 5.2.1. Kontrolery

W folderze controllers znajdują się klasy, tak zwane kontrolery, które odpowiadają za komunikację po protokole HTTP, między klientem a serwerem. To właśnie w tych klasach definiujemy, jak będą wyglądały poszczególne końcówki (endpointy) oraz za co będą odpowiadały. Kontroler, który odpowiada za akcje na zasobach klientów (Customer) przedstawiony jest na Listingu 15.

*Listing 15. Fragment klasy CustomerController.java. Źródło: opracowanie własne*

```
@RestController
@CrossOrigin(origins = "http://localhost:4200")
public class CustomerController {

    @Autowired
    private CustomerService service;

    @RequestMapping(value = "/customers", method = RequestMethod.GET)
    public List<Customer> findAll(){ return service.getCustomers();}

    @RequestMapping(value = "/customers/{id}", method = RequestMethod.GET)
    public Customer findById(@PathVariable("id") long id){
        return service.getCustomer(id);
    }

    @RequestMapping(value = "/customers", method = RequestMethod.POST)
    public Customer create(@RequestBody Customer customer){
        service.saveCustomer(customer);
        return customer;
    }

    @RequestMapping(value = "/customers", method = RequestMethod.PUT)
    public Customer update(@RequestBody Customer customer){
        service.updateCustomer(customer);
        return customer;
    }

    @RequestMapping(value = "/customers/{id}", method = RequestMethod.DELETE)
    @PreAuthorize("hasRole('ADMIN')")
    public void delete(@PathVariable("id") long id){
        service.deleteCustomer(id);
    }
}
```

Klasa CustomerController rozpoczyna się adnotacją *@RestController*, która jest swego rodzaju skrótem dwóch adnotacji. Pierwszą z nich jest adnotacja *@Controller*, która wskazuje na to, że mamy do czynienia z klasą będącą kontrolerem. Druga adnotacja to *@ResponseBody*, którą musielibyśmy dodawać do każdej metody obsługującej żądania w naszym kontrolerze.

Kolejną adnotacją jest *@CrossOrigin(origins = http://localhost:4200)* do obsługi mechanizmu CROS (ang. Cross-origin Resource Sharing) (@CrossOrigin Annotation Example, 2021). Odpowiada za bezpieczną komunikację między przeglądarką a serwerem. Wskazując w parametrze *origins* na adress http://localhost:4200 pozwalamy na komunikację właśnie z tym adresem.

Następnie użyto automatycznego dowiązania obiektu klasy CustomerService, które odpowiada za operację na bazie danych. Jego implementacja zostanie przedstawiona w dalszych podrozdziałach.

Nad każdą z metod odpowiadających za akcję na zasobie zauważymy adnotację @RequestMapping. Parametr *value* określa ścieżkę, pod którą dostępna jest akcja, natomiast *method* wskazuje na metodę HTTP, która musi zostać użyta. Zatem, aby pobrać reprezentację zasobów klientów należy skorzystać z metody HTTP GET, podając ścieżkę http://localhost:8080/api/customers, gdzie http://localhost:8080/api/ jest główną ścieżką naszej aplikacji. Odwołując się do metody *getCustomers();* w klasie CustomerService zostaną zwróceni klienci w postaci listy.

*Listing 16. Metoda findById zwracająca klienta o konkretnym numerze id. Źródło: opracowania własne*

```
@RequestMapping(value = "/customers/{id}", method = RequestMethod.GET)
public Customer findById(@PathVariable("id") long id){
    return service.getCustomer(id);
}
```

Metoda z Listingu 16 zawiera dodatkowo adnotację *@PathVariable*, pozwala ona na wyciągnięcie zmiennej ze ścieżki zapytania. W tym przypadku jest to parametr id, jako typ zmiennej *long***.** W ten sposób z łatwością możemy wyszukać konkretnego klienta, korzystając z metody getCustomer, przyjmującej jako parametr zmienną typu *long*.

Korzystając z metody POST z protokołu HTTP, możemy przekazać pewien pakiet informacji w postaci JSON. W tym celu korzystamy z metody *create* w naszym kontrolerze, co pozwala utworzyć nowego klienta. Potrzebna jest nam do tego adnotacja *@RequestBody*, którą umieszczamy przed parametrem w metodzie. Reprezentacja zasobu zostanie przekonwertowana do obiektu typu Customer, po czym zapisana przy pomocy metody *saveCustomer*, tak jak przedstawiono na Listingu 17. Podobnie wygląda klasa update.

*Listing 17. Metoda create w kontrolerze odpowiadająca za utworzenie nowego klienta*

```
@RequestMapping(value = "/customers", method = RequestMethod.POST)
public Customer create(@RequestBody Customer customer){
    service.saveCustomer(customer);
    return customer;
}
```

W metodzie odpowiadającej za usuwanie zasobu została użyta dodatkowa adnotacja *@PreAuthorize("hasRole('ADMIN')")*. Jest to adnotacja ze Spring Security i polega ona na sprawdzaniu autoryzacji do metody przed jej wywołaniem. W tym przypadku jedynie użytkownik z rolą ADMIN może wykonać akcję usuwania klienta.

### 5.2.2. Modele

Kolejnym ważnym miejscem w strukturze projektu jest folder *models*, w którym znajdują się klasy. Są to klasy będące reprezentacją zasobów po stronie serwera back-end. Reprezentują one tabele w bazie danych. Określamy w niej parametry oraz relacje z obiektami innych klas. W celu zdefiniowania takiej klasy jako model musimy użyć adnotacji *@Entity*. Przykład takiej klasy znajduje się w Listingu 18.

*Listing 18. Przykład klasy typu Entity na podstawie CustomerProduct*

```
@Setter
@Getter
@Entity
public class CustomerProduct {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long customerProductId;

    @NotBlank
    private String serialNumber;

    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "customerId")
    @JsonIgnoreProperties(value= {"customerProductList"}, allowSetters = true)
    private Customer customer;

    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "deviceId")
    @JsonIgnoreProperties(value = {"customerProductList"}, allowSetters = true)
    private Device device;

    @OneToMany(cascade = CascadeType.ALL, mappedBy = "customerProduct" ,
            fetch = FetchType.EAGER)
    @JsonIgnoreProperties(value = {"customerProduct"}, allowSetters = true)
    private List<Task> taskList;

    public CustomerProduct() {
    }
}
```

Na samej górze klasy CustomerProduct znajdują się dwie dodatkowe adnotacje *@Setter* oraz *@Getter*. Pochodzą one z biblioteki Lombok. Stosując te dwie adnotacje automatycznie zostaną utworzone metody typu setter oraz getter, co skraca ilość kodu w klasie. Następnie w ciele klasy możemy zobaczyć *@Id* oraz *@GeneratedValue(strategy = GenerationType.IDENTITY*. Pierwsza z nich sprawia, że zmienna *customerProductId*, staje się kluczem głównym, natomiast druga odpowiada za automatyczne generowanie wartości tego klucza na postawie wybranej strategii. Do dyspozycji mamy cztery strategie AUTO, TABLE, SEQUENCE albo IDENTITY.

W klasie tej określamy również relacje między obiektami. Biblioteka Spring Data JPA dostarcza nam cztery podstawowe adnotacje służące do określenia relacji. Są nimi:

- *@ManyToMany* – oznacza relację wiele do wielu,
- *@ManyToOne* – oznacza relację wiele do jednego,
- *@OneToMany* – oznacza relację jeden do wielu,

- *@OneToOne* – oznacza relację jeden do jednego.

W klasie przedstawionej na Listingu 18. zostały użyte dwie z wyżej wymienionych adnotacji. Pierwszą z nich jest *@ManyToOne*. W Listingu 19. dotyczy ona zmiennej typu *Customer* i mówi nam o tym, że obiekt typu *CustomerProduct* może zostać powiązany z jednym obiektem typu *Customer*. Po drugiej stronie natomiast, w klasie *Customer* znajduje się kolekcja na przykład typu *List*, stanowiąca relację w drugą stronę, określającą relację w której to obiekt *Customer* może zostać powiązany z wieloma obiektami typu *CustomerProduct.*

*Listing 19. Relacja @ManyToOne w klasie CustomerProduct*

```
@ManyToOne(fetch = FetchType.EAGER)
@JoinColumn(name = "customerId")
@JsonIgnoreProperties(value= {"customerProductList"}, allowSetters = true)
private Customer customer;
```

Dodatkowo przy pomocy *@JoinColumn* wskazujemy na kolumnę, w której ma być reprezentowana ta relacja. Na końcu została użyta adnotacja *@JsonIgnoreProperties*, która pozwala na ignorowanie pewnych właściwości w zwracanym formacie JSON. Eliminuje to pewne problemy w momencie odczytu zasobów, między którymi istnieją relacje zwrotne.

W Listingu 20. znajduje się fragment kodu określający relację jeden do wielu. Wskazuje ona na listę w tym przypadku obiektów typu *Task.* Oznacza to, że jeden objekt typu *CustomerProduct* może zostać powiązany z wieloma obiektami typu *Task*. Oprócz wskazania typu relacji możemy określić dodatkowe parametry. Wartość *CascadeType.ALL* dla parametru cascade. określa, jakie akcje mogą zostać wykonane kaskadowo na powiązanym objekcie lub kolekcji obiektów w przypadku wykonania danej operacji na obiekcie, który nas interesuje. W przypadku *CascadeType.ALL* jest to każda z dostępnych operacji. To znaczy, że w momencie usunięcia obiektu typu *CustomerProduct* zostaną usunięte również obiekty *Task* z nim powiązane. Możemy również wybrać pojedyncze operacje, które chcemy, aby były wykonane. Do dyspozycji mamy następujące operacje:

- CascadeType.ALL,
- CascadeType.PERSIST,
- CascadeType.MERGE,
- CascadeType.REMOVE,
- CascadeType.REFRESH,
- CascadeType.DETACH.

*Listing 20. Relacja @OneToMany w klasie CustomerProduct*

```
@OneToMany(cascade = CascadeType.ALL, mappedBy = "customerProduct" ,
      fetch = FetchType.EAGER)
@JsonIgnoreProperties(value = {"customerProduct"}, allowSetters = true)
private List<Task> taskList;
```

Atrybut *mappedBy = „customerProduct”* umożliwia utworzenie relacji dwukierunkowej między obiektami, poprzez mapowanie. Robimy to wskazując na zmienną,

pod którą występuje relacja w klasie powiązanej. W przypadku klasy *Task* jest to zmienna *customerProduct*.

### 5.2.3. Repozytoria

W folderze *repo* znajdują się interfejsy rozszerzające interfejs JpaRepository, który jest częścią biblioteki Spring Data JPA. Do mapowania obiektowo-relacyjnego wybrano Spring Data JPA, które stanowi wygodną alternatywą dla takich narzędzi jak Hibernate. Jest prostsze w użyciu oraz wymaga mniejszej ilości kodu do wykonania tych samych czynności. Interfejs JpaRepository dostarcza kilka podstawowych metod CRUD. Na Listingu 21. znajduje się przykładowe repozytorium.

*Listing 21. Interfejs repozytorium na przykładzie UserRepository*

```
@Repository
public interface UserRopository extends JpaRepository<User, Long> {
    Optional<User> findByUsername(String username);
    Boolean existsByUsername(String username);
    Boolean existsByEmail(String email);
}
```

W celu określenia interfejsu jako repozytorium musimy dodać adnotację *@Repository*. W podanym przykładzie interfejs UserRepository został rozszerzony o interfejs JpaRepository, gdzie wskazujemy na model, którego repozytorium tworzymy oraz typ dla klucza głównego. Już samo to pozwala nam na korzystanie z podstawowych metod, jak na przykład *findAll()*, która zwróci nam listę wszystkich obiektów danej klasy.

Oprócz tego możemy zdefiniować własne metody. Istnieje klika zasad dotyczących tworzenia własnych metod. Zawierając odpowiednie słowa kluczowe w nazwie metody oraz wskazując na konkretny parametr z modelu, którego tyczy się repozytorium, jesteśmy w stanie uzyskać metodę z wygenerowanym odpowiednim zapytaniem w JPQL (Java Persistence Query Language) do bazy danych. W tabeli 4. możemy zobaczyć przykład metod przetłumaczonych na JPQL.

*Tabela 4. Przykłady metod tłumaczonych na JPQL w Spring Data JPA*

| Słowo kluczowe | Nazwa metody | JPQL |
|---|---|---|
| And | findByLastnameAndFirstname | … where x.lastname = ?1 and x.firstname = ?2 |
| Is, Equals | findByFirstname,<br>findByFirstnameIs,<br>findByFirstnameEquals | … where x.firstname = ?1 |

Oczywiście słów kluczowych oraz możliwości jakie oferuje nam Spring Data JPA jest więcej (Query Creation, 2021), natomiast mając ogólną wiedzę na temat tego, w jaki sposób zapytania są tworzone, z łatwością możemy domyślić się, jak będą wyglądały zapytania z Listingu 21. oraz wydedukować, jak powinny wyglądać metody utworzone w przyszłości, rozszerzające możliwości interfejsu UserRepository.

### 5.2.4. Serwisy

W folderze *services* przechowywane są klasy z logiką biznesową. Ilość klas w tym folderze odpowiada ilości repozytoriów oraz modeli. W wybranej klasie zostaje wstrzyknięty interfejs repozytorium przy pomocy adnotacji *@Autowired*. Poszczególne metody odpowiadają czynnościom jakie chcemy przeprowadzić oraz które mogą zostać wykonane. W celu utrwalenia efektów tych operacji w bazie danych odwołujemy się do metod z repozytorium. Listing 22. przedstawia jedną z takich klas.

*Listing 22. Przykład klasy Service na przykładzie CustomerService*

```
@Service
@Transactional
public class CustomerService {

    @Autowired
    private CustomerRepo repository;

    @Autowired CustomerProductService customerProductService;

    public List<Customer> getCustomers(){
        return repository.findAll();
    }

    public Customer getCustomer(long id){
        return repository.findById(id).orElse(null);
    }

    public Customer getCustomerByFirsName(String firstName){
        return repository.findByFirstName(firstName);
    }

    public Customer getCustomerByPhonenumber(String phonenumber) {
         return repository.findByPhonenumber(phonenumber);
    }

    public Customer saveCustomer(Customer customer){
        return repository.save(customer);
    }

    public List<Customer> saveCustomers(List<Customer> customers){
        return repository.saveAll(customers);
    }

    public Customer updateCustomer(Customer customer){
        Customer existingCustomer = repository.findById(customer.getPersonId()).
orElse(null);
        existingCustomer.setEmail(customer.getEmail());
        existingCustomer.setCustomerProductList(customer.getCustomerProductList(
));
        existingCustomer.setPhonenumber(customer.getPhonenumber());
        existingCustomer.setFirstName(customer.getFirstName());
        existingCustomer.setLastName(customer.getLastName());
        existingCustomer.setAddress(customer.getAddress());
        return repository.save(existingCustomer);
    }
```

```
    public void deleteCustomer(long id){
        Customer existingCustomer = repository.findById(id).get();
        if(existingCustomer.getCustomerProductList() != null){
            for (CustomerProduct customerProduct: existingCustomer.getCustomerProd
uctList()) {
                customerProduct.setCustomer(null);
            }
            existingCustomer.setCustomerProductList(null);
        }
        repository.deleteById(id);
    }
}
```

Klasa staje się serwisem po użyciu adnotacji *@Service*. Pierwsze zaimplementowane metody komunikują się z bazą danych przy pomocy dowiązanego interfejsu repozytorium. Nieco bardziej rozbudowane natomiast są metody aktualizujące oraz usuwające dany zasób.

W metodzie *updateCustomer*, której jako parametr dostarczamy obiekt typu *Customer*, tworzymy zmienną tymczasową *existingCustomer*. Wartość zmiennej pobieramy z bazy danych, o ile istnieje. Jeżeli nie jest dostępna, zostaje ustawiona wartość *null*. Następnie za pomocą zmiennych typu setter oraz getter ustawiane są wartości dla nowo utworzonego obiektu *existingCustomer*, który finalnie zapisywany jest przy pomocy *repository.save(existingCustomer)*.

Podobna w działaniu jest metoda *deleteCustomer* odpowiedzialna za usuwanie klienta z bazy danych, której parametr to identyfikator typu *long*. Po znalezieniu klienta uruchamiana jest pętla usuwająca powiązanie klienta we wszystkie obiekty typu CustomerProduct. Dopiero po tej operacji usuwany jest klient. W efekcie usuwając klienta zatrzymujemy informacje na temat samego produktu oraz historię zgłoszeń, co może być przydatne przy analizie przyszłych problemów zgłaszanych przez klientów.

### 5.2.5. Payloads

Kolejny folder *payload* zawiera w sobie dwa podfoldery, *Requests* oraz *Responses*, w których dodane są klasy będące reprezentacją pewnych specyficznych dopuszczalnych zapytań oraz odpowiedzi od serwera. W folderze *Requests*, można znaleźć na przykład klasę LoginRequest, którego kod widoczny jest w Listingu 23.

*Listing 23. Przykład klasy LoginRequest*

```
@Setter
@Getter
public class LoginRequest {

    @NotBlank
    private String username;

    @NotBlank
    private String password;
}
```

Jest to prosta klasa, która posiada dwa parametry *username* i *password*. Następnie, po zdefiniowaniu takiej klasy możemy ją umieścić w parametrze metody w kontrolerze, dzięki

czemu metoda kontrolera będzie spodziewała się zapytania w ustalonej przez nas formie. W tym przypadku mowa o AuthenticationController, czyli kontrolerze który odpowiada za uwierzytelnienie. Przed typem argumentu w metodzie należy dodać adnotację *@RequestBody*, tak jak przedstawiono w Listingu 24.

*Listing 24. Metoda odpowiedzialna za logowanie w klasie AuthenticationController*

```
@PostMapping("/signin")
public ResponseEntity<?> authenticateUser
(@Valid @RequestBody LoginRequest loginRequest) {
    ...
}
```

Dodatkowo metoda authenticateUser jest typu ResponseEntity, dzięki czemu możemy manipulować całą odpowiedzią http od strony serwera. Jeżeli po udanym logowaniu chcemy zwrócić odpowiedź o statusie 200 OK, który informuje o powodzeniu, korzystamy z funkcji *ResponseEntity.ok()*. Jako argument funkcji w naszym przypadku wskazujemy klasę JwtResponse zdefiniowaną wcześniej w folderze *payload/responses*, w której zwracane są informacje na temat zalogowanego użytkownika. Przykład wykorzystanie tej funkcji możemy zaobserwować w Listingu 25.

*Listing 25. Przykład wykorzystania funkcji ResponseEntity*

```
return ResponseEntity.ok(new JwtResponse(
    jwt,
    userDetails.getId(),
    userDetails.getUsername(),
    userDetails.getEmail(),
    roles,
    userDetails.getTasks(),
    userDetails.getNotes()
));
```

### 5.2.6. Security

Implementacja bezpieczeństwa w aplikacjach jest złożonym zagadnieniem, dlatego najlepszym rozwiązaniem jest korzystanie z gotowych bibliotek stworzonych przez doświadczonych programistów oraz inspirowanie się istniejącymi wdrożeniami. Implementacja bezpieczeństwa w prototypie aplikacji Tech Support wykorzystuje gotowe biblioteki ze Spring Security oraz bazuje na różnych sprawdzonych implementacjach znalezionych w internecie (Java JWT: JSON Web Token for Java and Android, 2021) (Spring JWT Token, 2021) (Spring Security JPA authentication, 2021).

Główna logika związana z systemem uwierzytelniania oraz autoryzacji znajduje się w folderze *security*, ale klasy z nią powiązane znajdują się również w folderach *controllers*, *models* oraz *repo*.

W aplikacji zostało wykorzystane uwierzytelnienie za pomocą bezstanowego standardu JWT Web Token z użyciem biblioteki pomocniczej *jjwt* dostępnej na stronie github.com.

W folderze *security* znajduje się główna klasa odpowiedzialna za bezpieczeństwo aplikacji o nazwie *WebSecurityConfig*, dodatkowo dwa podfoldery *jwt* oraz *services*. Implementacja klasy *WebSecurityConfig* znajduje się w Listingu 26.

Klasa staje się konfiguracyjną po dodaniu adnotacji *@Configuration*. Dodatkowo klasa *WebSecurityConfig* zawiera adnotację *@EnableWebSecurity*, która sprawia, że dana klasa staje się klasą konfiguracyjną dla Spring Security. Jeżeli utworzymy klasy z tą adnotacją to Spring Security korzysta z konfiguracji domyślnej. Dzięki *@EnableGlobalMethodSecurity* dostarczamy dwie adnotacje *@PreAuthorize* oraz *@PostAuthorize*, które mogą być użyte w kontrolerze dla wybranych metod. Mając do dyspozycji taką adnotację możemy sprawdzić na poziomie metody, czy użytkownik posiada odpowiednią rolę, aby skorzystać z danego endpointu.

Klasa *WebSecuirtyConfig* rozszerza interfejs *WebSecurityConfigurerAdapter*, przez co od razu zmuszeni jesteśmy nadpisać metodę *configure(HttpSecurity http)*. Określamy w niej takie aspekty jak CORS oraz CSRF, warunkujemy czy użytkownik musi być zalogowany, aby móc wykonywać operację na zasobach, filtry oraz obsługę błędów.

Na początku klasy dowiązany jest serwis *UserDetailsServiceImpl* odpowiedzialny za wczytanie szczegółów na temat użytkownika. Ten serwis zostaje podany jako parametr w funkcji *userDetailsService* z klasy *AuthenticationManagerBuilder*. Ponadto, aby nie przetwarzać hasła tekstem odkrytym, używamy metody *passwordEncoder()*, w którym korzystamy z funkcji hashującej BCrypt.

*Listing 26. Klasa WebSecurityConfig*

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(
        prePostEnabled = true
)
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    UserDetailsServiceImpl userDetailsService;

    @Autowired
    private AuthEntryPointJwt unauthorizedHandler;

    @Bean
    AuthTokenFilter authenticationJwtTokenFilter(){
        return new AuthTokenFilter();
    }

    @Override
    public void configure(AuthenticationManagerBuilder authenticationManagerBuilde
r) throws Exception {
        authenticationManagerBuilder.userDetailsService(userDetailsService).passwo
rdEncoder(encoder());
    }
```

```
@Bean
    @Override
    public AuthenticationManager authenticationManagerBean() throws Exception {
        return super.authenticationManagerBean();
    }

    @Bean
    public PasswordEncoder encoder() {
        return new BCryptPasswordEncoder();
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.cors().and().csrf().disable();
        http.exceptionHandling().authenticationEntryPoint(unauthorizedHandler);
        http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATE
LESS);
        http.authorizeRequests()
                .antMatchers("/auth/**").permitAll()
                .anyRequest().authenticated();
        http.addFilterBefore(authenticationJwtTokenFilter(), UsernamePasswordAuthe
nticationFilter.class);
    }
}
```

W folderze *security/services* znajduje się klasa *UserDetailsImpl* implementująca interfejs UserDatails. Utworzenie takiej klasy pozwala nam na rozbudowanie podstawowej implementacji UserDetails o możliwość zwracania dodatkowych informacji, takich jak adres email, id czy też w naszym przypadku listę *tasków* oraz *notatek* użytkownika. Fragment klasy *UserDetailsImpl* znajduje się w Listingu 27.

*Listing 27. Fragement klasy UserDetailsImpl*

```
public class UserDetailsImpl implements UserDetails {
    private static final long serialVersionUID = 1L;

    private Long id;

    private String username;

    private String email;

    @JsonIgnore
    private String password;

    private Collection<? extends GrantedAuthority> authorities;

    private Set<Task> taskList;
    private Set<Note> notesList;
```

```
    public static UserDetailsImpl build(User user) {
        List<GrantedAuthority> authorities = user.getRoles().stream()
                .map(role -> new SimpleGrantedAuthority(role.getName().name()))
                .collect(Collectors.toList());
        return new UserDetailsImpl(
                user.getUserId(),
                user.getUsername(),
                user.getEmail(),
                user.getPassword(),
                authorities,
                user.getTaskList(),
                user.getNotesList()
        );
    }

    ...

     public Long getId() {
        return id;
    }

    public String getEmail() {
        return email;
    }
}
```

W folderze *security/jwt* znajdują się trzy klasy. Pierwszą z nich jest *AuthEntryPointJwt.java*, która implementuje interfejs *AuthenticationEntryPoint.* Klasa ta nadpisuje metodę *commence()*, obsługuje ona błąd *AuthenticationExeption* i zwraca komunikat błędu, kiedy nieautoryzowany użytkownik wysyła żądanie do zabezpieczonych zasobów. Fragment klasy znajduje się w Listingu 28.

*Listing 28. Fragement klasy AuthEntryPointJwt*

```
public class AuthEntryPointJwt implements AuthenticationEntryPoint {
    ...
    @Override
    public void commence(HttpServletRequest request, HttpServletResponse response,
 AuthenticationException authException) throws IOException, ServletException {
        logger.error("Unauthorized error: {}", authException.getMessage());
        response.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Error: Unauthoriz
ed");
    }
}
```

Drugą klasą jest *AuthTokenFilter.java*, rozszerzająca *OncePerRequestFilter*. Jest to filtr, który został dodany w klasie *WebSecurityConfig.java* przy pomocy metody *addFilterBefore().* Nadpisana metoda *doFilterInternal()* odpowiada za pobranie, przetworzenie oraz sprawdzenie tokenu z nagłówku żądania, pobranie użytkownika i uwierzytelnienie. Metoda *doFilterInternal* znajduje się na Listingu 29.

*Listing 29. Metoda doFilterInternal z klasy AuthTokenFilter*

```
protected void doFilterInternal(HttpServletRequest request, HttpServletResponse re
sponse, FilterChain filterChain) throws ServletException, IOException {
        try{
            String jwt = parseJwt(request);
            if(jwt != null && jwtUtils.validateJwtToken(jwt)) {
                String username = jwtUtils.getUserNameFromJwtToken(jwt);

                UserDetails userDetails = userDetailsService
                    .loadUserByUsername(username);
                UsernamePasswordAuthenticationToken authentication =
                    new UsernamePasswordAuthenticationToken(
                        userDetails, null, userDetails.getAuthorities());
                authentication.setDetails(new WebAuthenticationDetailsSource()
                    .buildDetails(request));
                SecurityContextHolder.getContext()
                    .setAuthentication(authentication);
            }
        } catch (Exception e) {
            logger.error("Cannot set user authentication: {}", e);
        }
        filterChain.doFilter(request, response);
}
```

Ostania klasa z folderu *security/jwt* to klasa *JwtUtils.java*. Jej zadaniem jest dostarczyć narzędzia do generowania tokenów na podstawie nazwy użytkownika, daty, czasu trwania tokenu oraz ciągu znaków *secret*. Dwie ostanie wartości definiowane są w pliku application.properties. Oprócz tego dostarcza mechanizm sprawdzania poprawności tokenu oraz funkcję zwracającą nazwę użytkownika na podstawie tokenu. Implementacja znajduje się w Listingu 30.

*Listing 30. Klasa JwtUtils*

```
@Component
public class JwtUtils {
    @Value("${api.jwt.secret}")
    private String secret;

    @Value("${api.jwt.expiration}")
    private int jwtExpiration;

    public String generateJwtToken(Authentication authentication) {
        UserDetailsImpl userPrincipal = (UserDetailsImpl) authentication.getPrinci
pal();

        return Jwts.builder()
                .setSubject((userPrincipal.getUsername()))
                .setIssuedAt(new Date())
                .setExpiration(new Date((new Date()).getTime() + jwtExpiration))
                .signWith(getSigningKey())
                .compact();
    }
```

```
    public String getUserNameFromJwtToken(String token){
        return Jwts.parserBuilder().setSigningKey(secret).build().parseClaimsJws(t
oken).getBody().getSubject();
    }

public boolean validateJwtToken(String authToken){
        try{
            Jwts.parserBuilder().setSigningKey(secret).build().parseClaimsJws(auth
Token);
            return true;
        } catch (SignatureException e) {
            logger.error("Invalid JWT signature: {}", e.getMessage());
        } catch (MalformedJwtException e) {
            logger.error("Invalid JWT token: {}", e.getMessage());
        } catch (ExpiredJwtException e) {
            logger.error("JWT token is expired: {}", e.getMessage());
        } catch (UnsupportedJwtException e) {
            logger.error("JWT token is unsupported: {}", e.getMessage());
        } catch (IllegalArgumentException e) {
            logger.error("JWT claims string is empty: {}", e.getMessage());
        }

        return false;
    }

    private Key getSigningKey() {
        byte[] keyBytes = Decoders.BASE64.decode(this.secret);
        return Keys.hmacShaKeyFor(keyBytes);
    }
}
```

Całą funkcjonalność uwierzytelnienia dopełnia kontroler *AuthenticationController*, zawierający endpointy, które obsługują logowanie oraz rejestrację użytkownika. Metoda odpowiadająca za logowanie była krótko omówiona przy okazji podrozdziału 5.3.5. Payloads.

Kod źródłowy odpowiedzialny za rejestrację użytkownika możemy zobaczyć w Listingu 31.

*Listing 31. Implementacja endpointu do rejestracji nowego użytkownika*

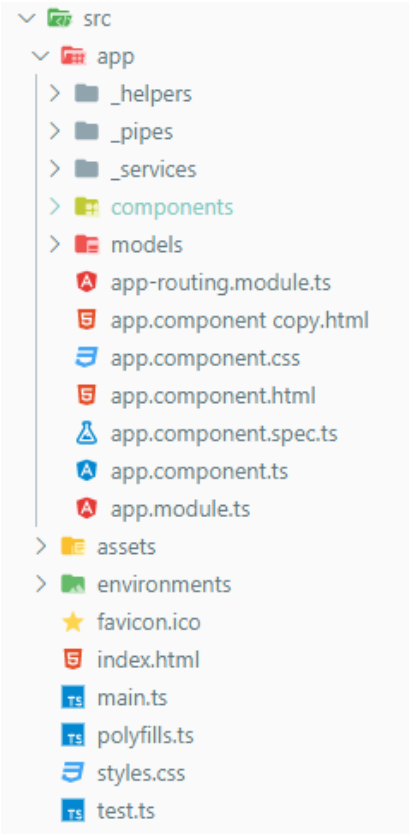
```
@PostMapping("/signup")
    public ResponseEntity<?> registerUser(@Valid @RequestBody SignupRequest signup
Request) {
        if (userRepository.existsByUsername(signupRequest.getUsername())) {
            return ResponseEntity
                    .badRequest()
                    .body(new MessageResponse("Error: Username is already taken!")
);
        }

        if (userRepository.existsByEmail(signupRequest.getEmail())) {
            return ResponseEntity
                    .badRequest()
                    .body(new MessageResponse("Error: Email is already in use!"));
        }
```

```
User user = new User(
                signupRequest.getUsername(),
                signupRequest.getEmail(),
                encoder.encode(signupRequest.getPassword()));

        Set<String> strRoles = signupRequest.getRole();
        Set<Role> roles = new HashSet<>();

        if (strRoles == null) {
            Role userRole = roleRepository.findByName(EnumRole.ROLE_USER)
                    .orElseThrow(() -
> new RuntimeException("Error:  Role is not found."));
            roles.add(userRole);
        } else {
            strRoles.forEach(role -> {
                switch (role) {
                    case "admin":
                        Role adminRole = roleRepository.findByName(EnumRole.ROLE_A
DMIN)
                                .orElseThrow(() -
> new RuntimeException("Error: Role is not found."));
                        roles.add(adminRole);
                        break;
                    case "mod":
                        Role modRole = roleRepository.findByName(EnumRole.ROLE_MOD
ERATOR)

                                .orElseThrow(() -
> new RuntimeException("Error: Role is not found."));
                        roles.add(modRole);
                        break;
                    default:
                        Role userRole = roleRepository.findByName(EnumRole.ROLE_US
ER)
                                .orElseThrow(() -
> new RuntimeException("Error: Role is not found."));
                        roles.add(userRole);
                }
            });
        }

        user.setRoles(roles);
        userRepository.save(user);

        return ResponseEntity.ok(new MessageResponse("User registered successfully
!"));
}
```

W pierwszej kolejności należy sprawdzić na podstawie użytkownika oraz maila, czy znajduje się on w bazie. Jeżeli nie, tworzony jest nowy użytkownik, następnie przypisywane są mu role. Domyślą rolą, w przypadku gdy w żądaniu nie została zdefiniowana żadna rola, jest zwykły użytkownik.

## 5.3. Aplikacja webowa

Aplikacja webowa została stworzona w technologii Angular (Angular Google, 2021) oraz Bootstrap. Angular odpowiada za interakcję z użytkownikiem i stanowi warstwę pośrednią między API, czyli aplikacją serwerową a użytkownikiem. Struktura aplikacji webowej przedstawiona jest na Rysunku 16.



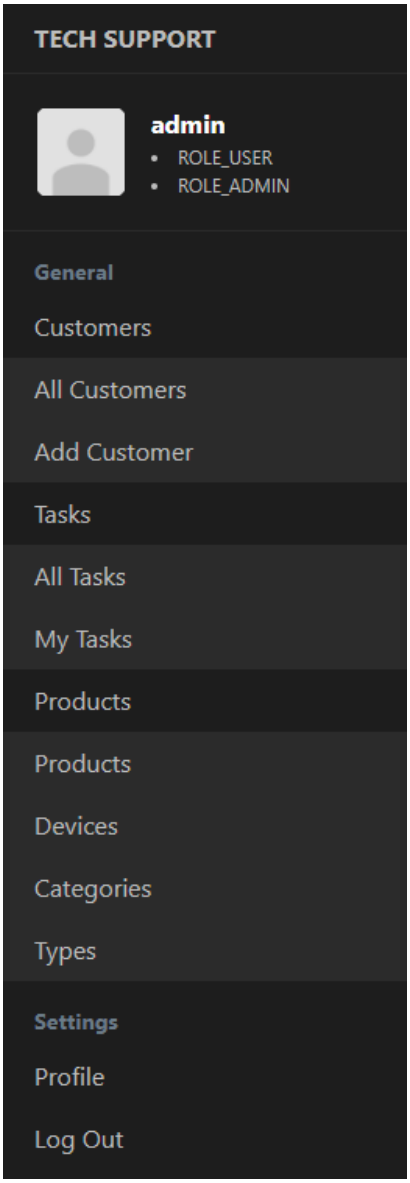

*Rysunek 16. Struktura aplikacji webowej we frameworku Angular*

W głównym folderze *src* znajduje się plik main.ts, który jest uruchamiany jako pierwszy. Odpowiada między innymi za uruchomienie modułów z pliku *app.module.ts.* W głównym folderze znajduje się również plik *index.html,* który zawiera główny szkielet html oraz tag *<app-root></app-root>*, odwołujący się do głównego komponentu jakim jest *app.component.ts*. Najważniejszymi folderami, które składają się na aplikację są:

- *Components* – zawiera podfoldery z poszczególnymi komponentami, jak *customer-list*, przykładowy komponent odpowiadający za wyświetlanie listy klientów,
- *Models* – klasy będące modelami, odpowiadają zasobom z aplikacji serwerowej,
- *_services* – serwisy odpowiadające za komunikację z aplikacją serwerową poprzez protokół http,
- *_helpers* oraz *_pipes* – klasy pomocnicze.

Menu aplikacji staje się dostępne po zalogowaniu się użytkownika. W systemie występują role i na ich podstawie dostępne są lub nie pewne zakładki. Administrator ma dostęp do wszystkich zakładek, natomiast zwykły użytkownik nie zobaczy na przykład

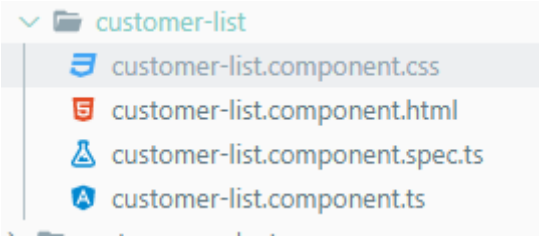
zakładek, w których można dodać lub edytować kategorie czy urządzenie. Widok menu z poziomu administratora zaprezentowany jest na Rysunku 17.




*Rysunek 17. Widok menu z poziomu administratora*

### 5.3.1. Komponenty

Komponenty (Angular Components Overview, 2021) to główne elementy, z których zbudowana jest aplikacja. Każdy komponent składa się z czterech plików: plik ze stylami CSS, dla każdego komponentu można zdefiniować dodatkowe style; plik html, który odpowiada za budowę naszego dokumentu; plik z testami jednostkowymi z rozszerzeniem *spec.ts* oraz sam komponent. Rysunek 18. przedstawia strukturę folderu komponentu *customer-list.*




*Rysunek 18. Struktura folderu komponentu customer-list*

W głównym pliku komponentu nad klasą znajduje się adnotacja *@Component*, która sprawia, że klasa staje się komponentem. W niej jako parametry podajemy *selector*, po którym możemy się odwołać do konkretnego komponentu. *TemplateUrl* oraz *styleUrl* wskazują odpowiednio na plik html oraz css. W przypadku omawianego komponentu *CustomerListComponent* pliki wskazane w adnotacji *@Component* to *customer-list.component.html* oraz *customer-list.component.css.*

U góry ciała klasy znajdują się dwie zmienne *customers* oraz *searchValue*. Pierwsza z nich to tablica obiektów typu *Customer*. Do tej tablicy zostanie przypisana lista obiektów pobranych z aplikacji serwerowej. Zmienna *searchValue* jest pomocnicza i do niej będzie przypisywany ciąg znaków stanowiący wartość potrzebną do filtrowania listy.

W konstruktorze podane zostały dwa parametry. Pierwszy to *customerService* typu *CustomerService.* Jest to serwis który odpowiada za komunikację z aplikacją serwera. Posiada funkcje odnoszące się do endpointów z adresem „http://localhost:8080/customers”. Drugi parametr to *router* Typu router. Pozwala on na przekierowanie na inny niż obecny adres URL.

Funkcja *ngOnInit* wywoływana jest podczas inicjalizacji komponentu. W tym przypadku wywołuje funkcję *retrieveCustomers().* Funkcja odwołuje się do serwisu *customerService* i wywołuje na nim funkcję *getCustomers* zwracającą typ *Observable*. Na otrzymanym objekcie typu *Observable* wywołujemy *subscribe*, a otrzymane dane przypisujemy do zmiennej *customers*.

Dodatkowo w kontrolerze została zdefiniowana funkcja *deleteCustomer*, działająca na takiej samej zasadzie jak funkcja pobierająca listę klientów, natomiast odwołująca się do endpointu *delete*, czyli usuwającej pewien zasób z bazy danych.

*Listing 32. Przykład komponentu na podstawie klasy CustomerListComponent*

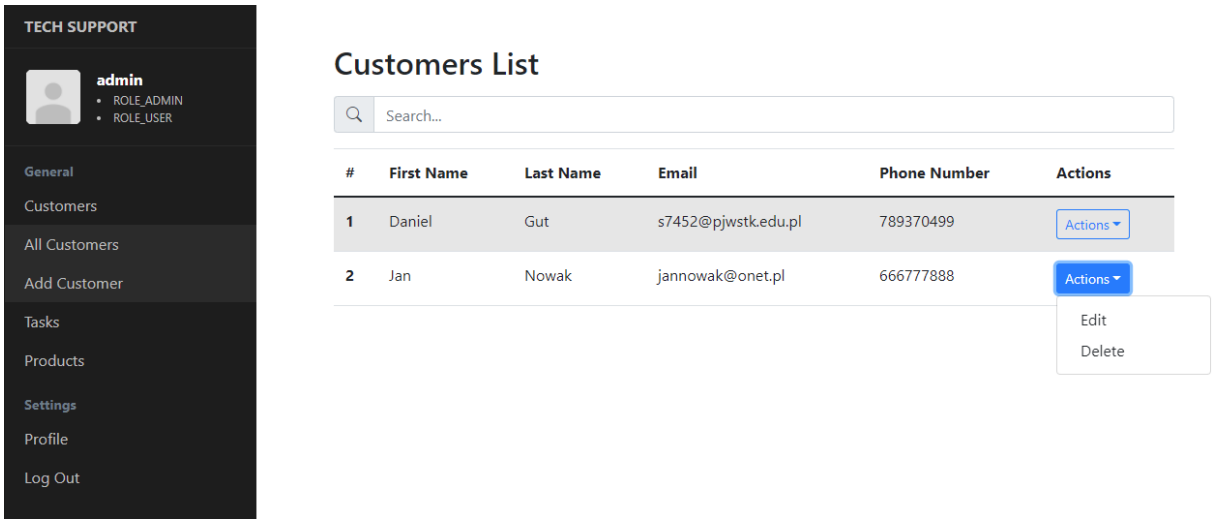
```
@Component({
  selector: 'app-customer-list',
  templateUrl: './customer-list.component.html',
  styleUrls: ['./customer-list.component.css']
})
export class CustomerListComponent implements OnInit {
  customers: Customer[] = [];
  searchValue = '';

  constructor(private customerService: CustomerService, private router: Router) {
  }

  ngOnInit(): void {
    this.retrieveCustomers();
  }
  retrieveCustomers() {
    this.customerService.getCustomers().subscribe(
      data => {
        this.customers = data;
        console.log(data);
      },
      err => {
        console.log(err);
      }
    );
  }
  deleteCustomer(customer: Customer): void {
    this.customerService.delete(customer.personId).subscribe(
      response => {
        console.log(response);
      },
      err => {
        console.log(err);
      }
    );
  }
}
```

Sposób reprezentacji danych definiujemy w pliku z rozszerzeniem html. Przedstawiony w Listingu 33. przykład wykorzystuje bibliotekę bootstrap oraz rozszerzenie ng-bootstrap. Przy ich pomocy można wygenerować w łatwy sposób tabelę. ngModel pozwala na przypisanie ciągu znaków z pola wyszukiwania do zmiennej *searchValue*. Tabela klientów generowana jest z wykorzystaniem pętli **ngFor*, w której dodatkowo wskazujemy na filtr *cusotmerFilter*. Dzięki znacznikowi *ngb-highlight* zapewniamy pogrubienie czcionki dla wyszukiwanych wartości w kolumnach tabeli. Kolumny na których ma być zastosowane podświetlanie oraz wyszukiwanie określamy przy użyciu *[term]*, w którym wskazujemy na zmienną *searchValue.*

*Listing 33. Przykład dokumentu html na podstawie customer-list.component.html*

```
<h2 class="mb-3">Customers List</h2>
<div class="input-group mb-3">
  <span class="input-group-text" id="basic-addon1"><i class="bi bi-
search"></i></span>
  <input type="text" class="form
control" placeholder="Search..." [(ngModel)]="searchValue">
</div>
<table class="table table-striped">
  <thead>
    <tr><th scope="col">#</th>
      <th scope="col">First Name</th>
      <th scope="col">Last Name</th>
      <th scope="col">Email</th>
      <th scope="col">Phone Number</th>
      <th scope="col">Actions</th></tr>
  </thead>
  <tbody>
    <tr *ngFor="let customer of customers | customerFilter:searchValue; index as i
">
      <th scope="row">{{ i + 1 }}</th>
      <td>
        <ngb-highlight [result]="customer.firstName" [term]="searchValue"></ngb-
highlight>
      </td>
      <td>
        <ngb-highlight [result]="customer.lastName" [term]="searchValue"></ngb-
highlight>
      </td>
      <td>
        <ngb-highlight [result]="customer.email" [term]="searchValue"></ngb-
highlight>
      </td>
      <td>
        <ngb-highlight [result]="customer.phonenumber" [term]="searchValue"></ngb-
highlight>
      </td>
      <td class="overflow-hidden">
        <div ngbDropdown container="body">
          <button class="btn btn-outline-primary btn-
sm" ngbDropdownToggle>Actions</button>
          <div ngbDropdownMenu>
            <button ngbDropdownItem [routerLink]="customer.personId">Edit</button>
            <button ngbDropdownItem (click)="deleteCustomer(customer)">Delete</but
ton>
          </div>
        </div>
      </td>
    </tr>
  </tbody>
</table>
```

Oprócz tego możemy zauważyć listę typu dropdown z dodatkowym znacznikiem *ngbDropdown* z rozszerzenia *ng-bootstrap*. Do dyspozycji są dwa przyciski. Pierwszy *Edit* wykorzystuje przekierowanie według id klienta, korzystając z funkcji *routerLink*. Drugi natomiast wywołuje funkcję *deleteCustomer* z kontrolera na danym kliencie. Widok z listy klientów zaprezentowano na Rysunku 19.




*Rysunek 19. Widok z zakładki All Customers*

Podczas implementacji niektórych funkcjonalności zaszła potrzeba przekazania pewnych informacji z jednego komponentu do drugiego. Jednym z takich miejsc jest widok edycji klienta, w którym oprócz informacji o samym kliencie wyświetlana jest lista powiązanych z nim produktów, z możliwością dodania nowego. W tym celu w pliku *customer-details.component.html*, przy pomocy selektora został wywołany komponent *AddCustomerproductComponent*, który odpowiada za dodawanie nowego produktu. Listing 34. przedstawia wywołanie komponentu *AddCustomerComponent*.

*Listing 34. Fragment kodu wywołujący komponent AddCustomerComponent*

```
<app-add-customerproduct (added)="refresh($event)" [customer]="currentCustomer">
</app-add-customerproduct>
```

W nawiasach kwadratowych umieszczona jest zmienna *customer*, do której zostaje przypisana wartość wskazanego obiektu *currentCustomer* typu *Customer*. Zmienna *customer* pochodzi z komponentu *AddCustomerproductComponent*, natomiast zmienna *currentCustomer* pochodzi z kontrolera *CustomerDetailsComponent*. Oprócz tego w komponencie *AddCustomerproductComponent* przed zmienną *customer*, należy dodać adnotację *Input()*. Podsumowując, przekazaliśmy obiekt z komponentu będącego „rodzicem” do komponentu będącego „dzieckiem”.

Chcąc przesłać pewną wartość z komponentu będącego „dzieckiem” do komponentu „rodzica” musimy posłużyć się *eventem*. W tym celu po stronie komponentu *AddCustomerproductComponent* została utworzona zmienna *added* poprzedzona adnotacją *@Output()*, jako nowy obiekt *EventEmitter()*. Fragment kodu *(added)=”refresh($event)”* znajdujący się w Listingu 34. odpowiada za wywołanie funkcji *refresh*, znajdującej się w komponencie *CustomerDetailsComponent*, w momencie gdy na zmiennej *added* zostanie wyemitowane zdarzenie. Fragmenty kodu odpowiedzialne za opisane powyżej mechanizmy znajdują się w Listingu 35., Listingu 36. oraz Listingu 37.

*Listing 35. Deklaracja zmiennych w komponencie AddCustomerproductComponent*

```
@Output() added = new EventEmitter();
@Input() customer?: Customer;
```

*Listing 36. Fragment funkcji submit() z komponentu AddCustomerproductComponent wywołujący emisję zdarzenia na zmiennej added*

```
submit(){
    …
    this.added.emit();
}
```

*Listing 37. Funkcja w komponencie CustomerDetailsComponent, która zostaje wywołana po wyemitowaniu zdarzenia z komponentu AddCustomerproductComponent*

```
refresh($event: any){
    window.location.reload();
}
```

### 5.3.2. Serwisy

Serwisy są klasami odpowiedzialnymi za komunikację z aplikacją serwerową. Komunikacja wykorzystuje protokół http. W tym celu w konstruktorze została zdefiniowana zmienna *http* typu *HttpClient*. Następnie na tej zmiennej możemy wykorzystać dostępne funkcje jak *get*, *post*, *put* czy *delete*. Funkcje zostały wykorzystane w taki sposób, aby odpowiadały endpointom zdefiniowanym po stronie aplikacji serwera.

Na samej górze klasy znajduje się stała typu string która wskazuje na URL endpointu. Poniżej dodana jest adnotacja *@Injectable*, która oznacza klasę jako gotową do dostarczenia i wstrzyknięcia jako zależność. Na Listingu 38. widzimy przykład implementacji serwisu na podstawie klasy *TaskService*.

*Listing 38. Przykład serwisu na podstawie klasy TaskService*

```
const API_URL = 'http://localhost:8080/api/tasks';

@Injectable({
  providedIn: 'root'
})
export class TaskService {

  constructor(private http: HttpClient) { }

  getTasks(): Observable<Task[]>{
    return this.http.get<Task[]>(API_URL);
  }

  getTasksByUser(data: any): Observable<Task[]>{
    return this.http.post<Task[]>(`${API_URL}/user/`, data);
  }

  get(id: any): Observable<Task>{
    return this.http.get(`${API_URL}/${id}`);
  }

  create(data: any): Observable<any> {
    return this.http.post(API_URL, data);
  }

  update(data:any): Observable<any> {
    return this.http.put(API_URL, data);
  }

  delete(id: any): Observable<any> {
    return this.http.delete(`${API_URL}/${id}`);
  }
}
```

### 5.3.3. Pipes

W folderze *_pipes* zostały zdefiniowane filtry (Angular 12 Custom Filter Search Pipe Example Tutorial, 2021). Filtry wykorzystywane są między innymi do wyszukiwania i filtrowania tabel wyświetlających listy obiektów w warstwie widoku. W celu wykorzystania danego filtru należy na przykład wskazać go w pętli **ngFor*, tak jak zostało to przedstawione w podrozdziale 5.3.1. Komponenty. Przykład takiego filtra znajduje się w Listingu 39.

*Listing 39. Przykład filtra typu Pipe*

```
@Pipe({
  name: 'customerFilter',
  pure: false
})
export class CustomerFilterPipe implements PipeTransform {

  transform(customers: Customer[], searchValue: string): Customer[] {
    if (!customers || !searchValue) {
      return customers;
    }
    return customers.filter(customer =>
      customer.firstName?.toLocaleLowerCase().includes(searchValue.toLowerCase())
||
      customer.lastName?.toLocaleLowerCase().includes(searchValue.toLowerCase()) |
|
      customer.email?.toLocaleLowerCase().includes(searchValue.toLowerCase()) ||
      customer.phonenumber?.toLocaleLowerCase().includes(searchValue.toLowerCase()
)
    );
  }
}
```

Nad klasą *CustomerFilterPipe*, znajduje się adnotacja *@Pipe*, w której definiujemy nazwę, po której możemy się odwołać w innych częściach aplikacji. Sama klasa implementuje interfejs *PipeTransform*, który dostarcza funkcję *transform*. W tej właśnie funkcji można zdefiniować logikę działania danego filtru. W przykładzie z Listingu 35. jako parametry zostały wskazane: tablica obiektów typu *Customer* oraz wartość typu *string*, na podstawie której odbywa się przeszukiwanie. Samo filtrowanie listy odbywa się przy pomocy funkcji *filter* dostarczanej w ramach języka TypeScript.

### 5.3.4. Bezpieczeństwo

Po stronie aplikacji webowej bezpieczeństwo realizowane jest za pomocą dwóch serwisów *auth.service* oraz *token-storage.service*, pomocniczej klasy *auth.interceptor*, komponentów do logowania oraz rejestracji. Dodatkowo w każdym komponencie mogą zostać sprawdzone role użytkownika i na tej podstawie mogą zostać wyświetlone lub nie pewne elementy widoku. Implementacja według pomysłu różnych autorów i internecie. (Angular JWT Authentication, 2021)

Zaczynając od serwisów, mają one za zadanie komunikację z endpointem aplikacji serwerowej służącej do rejestracji i uwierzytelniania użytkowników. Oprócz tego *token-storage.service* odpowiada za przechowywanie otrzymanego tokena. Na Listingu 40. oraz Listingu 41. przedstawiono implementację klas serwisowych.

*Listing 40. Klasa AuthService z aplikacji webowej*

```
const AUTH_API = 'http://localhost:8080/api/auth/';

const httpOptions = {
  headers: new HttpHeaders({ 'Content-Type': 'application/json' })
};

@Injectable({
  providedIn: 'root'
})
export class AuthService {
  constructor(private http: HttpClient) { }

  login(username: string, password: string): Observable<any> {
    return this.http.post(AUTH_API + 'signin', {
      username,
      password
    }, httpOptions);
  }

  register(username: string, email: string, password: string): Observable<any> {
    return this.http.post(AUTH_API + 'signup', {
      username,
      email,
      password
    }, httpOptions);
  }
}
```

*Listing 41. Klasa TokenStorageService z aplikacji webowej*

```
const TOKEN_KEY = 'auth-token';
const USER_KEY = 'auth-user';

@Injectable({
  providedIn: 'root'
})
export class TokenStorageService {
  constructor() { }

  signOut(): void {
    window.sessionStorage.clear();
  }

  public saveToken(token: string): void {
    window.sessionStorage.removeItem(TOKEN_KEY);
    window.sessionStorage.setItem(TOKEN_KEY, token);
  }
```

```
  public getToken(): string | null {
    return window.sessionStorage.getItem(TOKEN_KEY);
  }

  public saveUser(user: any): void {
    window.sessionStorage.removeItem(USER_KEY);
    window.sessionStorage.setItem(USER_KEY, JSON.stringify(user));
  }

  public getUser(): any {
    const user = window.sessionStorage.getItem(USER_KEY);
    if (user) {
      return JSON.parse(user);
    }

    return {};
  }
}
```

Klasa pomocnicza *AuthInterceptor* implementuje interfejs *HttpInterceptor*. Dostarcza funkcję *intercept*, dzięki której możemy sprawdzać oraz modyfikować żądanie http zanim zostanie ono wysłane do serwera. W przypadku aplikacji Tech Support w tym miejscu zostaje dodany header *Authorization* z odpowiednim tokenem. Przykład znajduje się w Listingu 42.

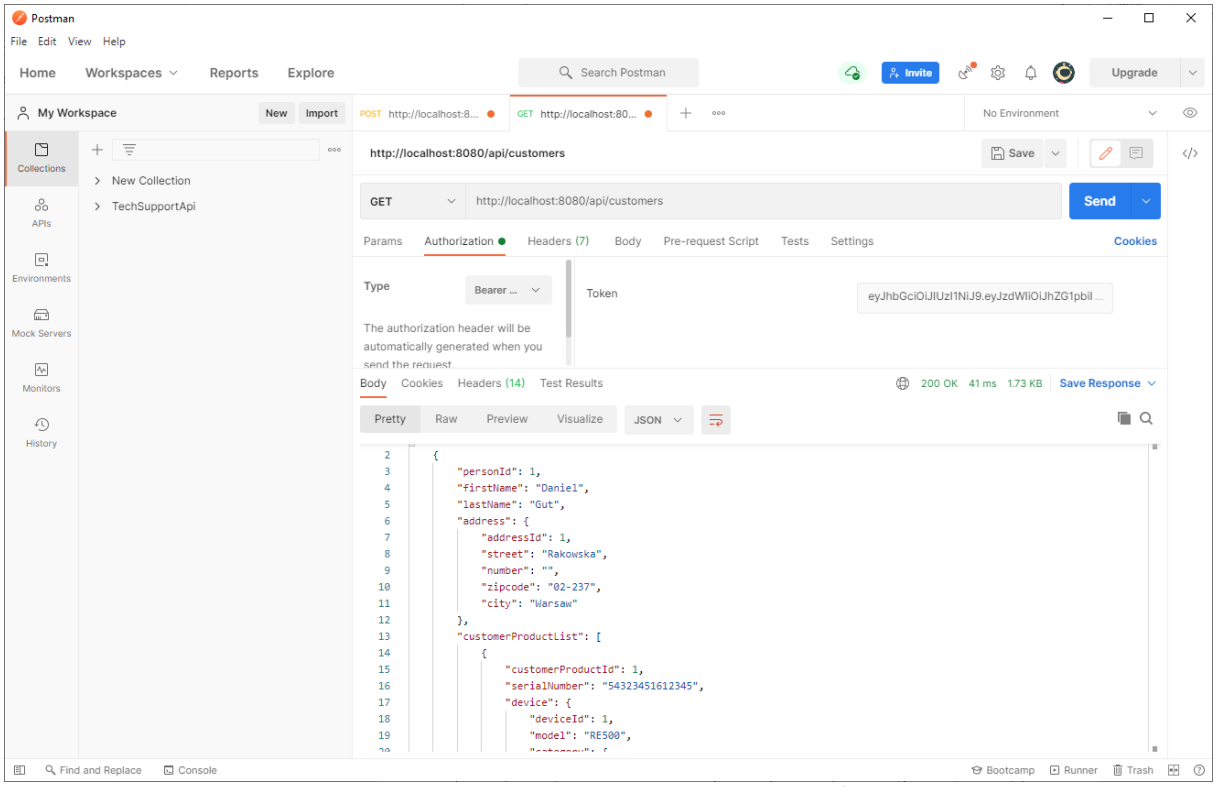
*Listing 42. Implementacja klasy AuthInterceptor z aplikacji webowej*

```
@Injectable()
export class AuthInterceptor implements HttpInterceptor {
  constructor(private token: TokenStorageService, private router: Router) { }
  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>>
{
    let authReq = req;
    const token = this.token.getToken();
    if (token != null) {
      authReq = req.clone({ headers: req.headers.set(TOKEN_HEADER_KEY, 'Bearer ' +
 token) });
    }
    return next.handle(authReq).pipe(
      tap(null,
        err => {
          if(err.status==401) {
            this.token.signOut();
            this.router.navigate(['/login']);
          }
        })
    );
  }
}
```

# 6. Faza testowania

Testowanie odbywało się manualnie z wykorzystaniem aplikacji Postman. Jest to bardzo popularne narzędzie, dzięki któremu w prosty sposób możemy przetestować endpointy z naszego API, nie posiadając jeszcze stworzonego interfejsu graficznego. Możemy wybrać

metodę http, taką jak na przykład GET czy POST oraz wskazać na adres URL. Dodatkowo mamy możliwość dodania nagłówków oraz ciała zapytania. Wszystko to w przystępnej formie interfejsu graficznego, tak jak przedstawia Rysunek 20.




*Rysunek 20. Przykład testowana z wykorzystaniem aplikacji Postman. Źródło: opracowanie własne*

Po stworzeniu aplikacji webowej, testy również wykonano manualnie, w tradycyjny sposób. Przechodząc przez kolejne zakładki aplikacji testy obejmowały logowanie, wylogowanie, dodawanie, edycję oraz usuwanie kolejnych zasobów.

Tabela 5 oraz Tabela 6 przedstawiają przykładowy scenariusz testowania, kolejno z wykorzystaniem aplikacji Postman oraz interfejsu aplikacji webowej.

*Tabela 5. Testowanie. Dodawanie klienta*

| **Dodanie klienta** | | |
|---|---|---|
| **Lp.** | **Wykonany krok** | **Oczekiwany wynik** |
| 1 | Odwołanie się do endpointu służącego do logowania, podając login i hasło użytkownika. Wykorzystanie metody POST. | Aplikacja serwerowa zwraca JWT Token. |

| | | |
|---|---|---|
| 2 | Dodanie otrzymanego tokenu jako metody uwierzytelnienia. Utworzenie reprezentacji zasobu klienta w JSON. Wysłanie za pomocą metody POST. | Nowy zasób zostaje utworzony oraz wyświetlony w formacie JSON. |
| 3 | Za pomocą endpoinu odwołującego się do konkretnego klienta na podstawie id podanego w URL, pobranie danych klienta przy pomocy metody GET. | Prawidłowo zwrócona reprezentacja zasobu danego klienta w formacie JSON. |

*Tabela 6. Testowanie. Usuwanie klienta*

| **Usunięcie klienta** | | |
|---|---|---|
| **Lp.** | **Wykonany krok** | **Oczekiwany wynik** |
| 1 | Zalogowanie się, podając login i hasło w formularzu odpowiadającym za logowanie. | Po udanym zalogowaniu następuje przekierowanie na stronę główną aplikacji webowej. |
| 2 | Przejście do zakładki Customer-> List, w celu wyświetlenia listy klientów. | Wyświetlenie listy klientów w formie tabeli. |
| 3 | Rozwinięcie listy typu drop-down przy wybranym kliencie i wciśnięcie przycisku Delete. | Klient zostaje usunięty po wciśnięciu przycisku, a widok zostaje odświeżony. Klient nie powinien znajdować się na liście. |

# 7. Podsumowanie

W przedstawionym prototypie udało się zaimplementować podstawowe działanie systemu: zostało umożliwione dodawanie oraz edycja klientów, urządzeń, kategorii, typów oraz produktów, które mogą być identyfikowane na podstawie numeru seryjnego. Dodatkowo udało się stworzyć system logowania użytkowników, autoryzację oraz możliwość stworzenia zgłoszeń powiązanych z produktem klienta oraz z użytkownikiem tworzącym zgłoszenie. W niniejszym rozdziale zostały przedstawione zalety oraz wady utworzonego systemu oraz przedstawiono propozycję dalszego rozwoju systemu o funkcjonalności, których nie udało się wprowadzić ze względu na ograniczone ramy czasowe.

## 7.1. Zalety systemu

Jako główną zaletę systemu należy przedstawić przede wszystkim jego prostotę oraz główną koncepcję. Spełnia wszystkie założenia dotyczące rejestracji zgłoszeń od klienta, a jednocześnie nie przytłacza swoją złożonością. Bardzo dobrze sprawdzi się w przedsiębiorstwach, które nie mają zapotrzebowania na skomplikowany system łączący się z serwerem e-mail oraz z centralą telefoniczną. Będą to głównie mniejsze firmy, które chcą uzyskać możliwość rejestracji zgłoszeń klienta oraz śledzenie ich historii.

Dodatkowo system jest przystosowany do dalszego rozwoju ze względu na zastosowaną architekturę REST. Funkcjonalności po stronie serwera oraz aplikacji webowej mogą rozwijać się niezależnie.

Prosta i intuicyjna szata graficzna aplikacji webowej zapewnia szybkie wdrożenie nowego użytkownika systemu.

## 7.2.Wady systemu

Do głównych wad systemu należy przede wszystkim fakt, iż jest on w fazie wczesnego prototypu, dlatego mimo wykonania podstawowych testów może się zdarzyć, że w aplikacji co jakiś czas pojawią się błędy. Wiele fragmentów kodu będzie musiała zostać przebudowana oraz zoptymalizowana.

To prowadzi do kolejnej wady systemu jakim są właśnie testy. W przyszłości należy poprawić ten element i zagwarantować testowanie lepszej jakości oraz częściowo zautomatyzowane.

Interfejs użytkownika mimo swojej prostoty nie zapewnia najlepszych wrażeń z perspektywy użytkownika. W przyszłości powinien wykorzystywać w pełni potencjał zastosowanych technologii.

## 7.3.Propozycja rozwoju systemu

Wielokrotnie w trakcie opisywania założeń proponowanego systemu poruszany był temat elastyczności całego systemu w kontekście przyszłego rozwoju. W międzyczasie pojawiały się pomysły, które ze względu na ograniczenia czasowe nie mogły zostać wdrożone.

Pierwszym proponowanym elementem w kierunku rozbudowy, jaki nasuwa się przy architekturze REST, jest stworzenie aplikacji mobilnej, która zapewni możliwość rejestrowania spraw dla wszystkich osób pracujących w dziale wsparcia technicznego, nie tylko w biurze, ale i w terenie. Do tego celu może zostać wykorzystany framework NativeScript do tworzenia aplikacji mobilnych, a który wspiera wykorzystany już w aplikacji webowej framework Angular.

Dodatkowo, tak jak w przypadku niektórych konkurencyjnych rozwiązań, ciekawym rozszerzeniem będzie portal dla klientów, na którym będą dostępne artykuły z wcześniej rozwiązanymi problemami, czy też możliwość podejrzenia statusu swojego zgłoszenia na podstawie numeru seryjnego urządzenia.

Wzorując się również na konkurencji, warto rozważyć wprowadzenie obsługi serwera pocztowego, w celu scentralizowania obsługi zgłoszeń poprzez wspólny adres e-mail dla całego działu wsparcia technicznego.

W dalszej perspektywie przydatnym rozszerzeniem będzie wdrożenie nowej roli oraz obsługi dla działu reklamacji, a nawet moduł umożliwiający podejrzenie stanów magazynowych, dostępności produktów.

Podsumowując, kierunków rozwoju jest wiele, a ich wdrożenie zależne będzie od faktycznego zapotrzebowania po stronie organizacji klienta.

# Bibliografia


@CrossOrigin Annotation Example. (2021). Pobrano z lokalizacji https://examples.javacodegeeks.com/enterprise-java/spring/boot/spring-boot-crossorigin-annotation-example/

Angular 12 Custom Filter Search Pipe Example Tutorial. (2021). Pobrano z lokalizacji https://www.remotestack.io/angular-custom-filter-search-pipe-example-tutorial/

Angular Components Overview. (2021). Pobrano z lokalizacji https://angular.io/guide/component-overview

Angular Google. (2021). Pobrano z lokalizacji https://angular.io/docs

Angular JWT Authentication. (2021). Pobrano z lokalizacji https://bezkoder.com/angular-jwt-authentication/

Bootstrap. (2021). Pobrano z lokalizacji https://getbootstrap.com/

Bootstrap list-group. (2021). Pobrano z lokalizacji Źródło:https://getbootstrap.com/docs/5.0/components/list-group/

Bretet, A. (2016). *Spring MVC Cookbook.* Birmingham: Packt Publishing Ltd.

Çalıs¸kan, M. i Sevindik, K. (2015). *Beginning Spring.* Indianapolis: John Wiley & Sons, Inc.

FocalScope. (2021). Pobrano z lokalizacji https://www.focalscope.com/

Gradle - Maven vs Gradle. (2021). Pobrano z lokalizacji https://gradle.org/maven-vs-gradle/

Gradle. (2021). Pobrano z lokalizacji https://gradle.org/

Horstmann, C. S. (2016). Core Java. W C. S. Horstmann, *Core Java.* Oracle.

Java JWT: JSON Web Token for Java and Android. (2021). Pobrano z lokalizacji https://github.com/jwtk/jjwt

Microsoft TypeScript. (2021). Pobrano z lokalizacji https://www.typescriptlang.org/

MySQL Documentation. (2021). Pobrano z lokalizacji https://dev.mysql.com/doc/refman/8.0/en/what-is-mysql.html

Oracle. (2021). Pobrano z lokalizacji https://www.oracle.com/pl/java/

Query Creation. (2021). Pobrano z lokalizacji https://docs.spring.io/spring-data/jpa/docs/2.5.1/reference/html/#jpa.query-methods.query-creation

Spring Framework. (2021). Pobrano z lokalizacji https://spring.io/projects/spring-framework

Spring JWT Token. (2021). Pobrano z lokalizacji https://bezkoder.com/

Spring Quick Start. (2021). Pobrano z lokalizacji https://spring.io/quickstart

Spring Security JPA authentication. (2021). Pobrano z lokalizacji https://www.websparrow.org/spring/spring-boot-spring-security-with-jpa-authentication-and-mysql

Stack Overflow. (2021). Pobrano z lokalizacji https://insights.stackoverflow.com/survey/2020

SupportBee. (2021). Pobrano z lokalizacji https://supportbee.com/

Team Support. (2021). Pobrano z lokalizacji https://www.teamsupport.com/

Wix Answers. (2021). Pobrano z lokalizacji https://www.wixanswers.com/

Wstęp do REST API. (2021). Pobrano z lokalizacji https://devszczepaniak.pl/wstep-do-rest-api/

# Spis listingów

Listing 1. Przykład podstawowych zależności w pliku pom.xml dla projektu Spring Framework ..... 23
Listing 2. Przykład podstawowych zależności w pliku pom.xml dla projektu Spring Boot.... 23
Listing 3. Główna klasa uruchamiająca aplikację Spring Boot ..... 24
Listing 4. Przykładowy plik build.gradle wygenerowany przy pomocy narzędzia Spring Initializr ..... 25
Listing 5. Przykład aplikacji HelloWorld ..... 26
Listing 6. Przykładowy minimalistyczny komponent we frameworku Angular. Źródło: opracowanie własne ..... 27
Listing 7. Przykład użycia komponentu w szablonie. Źródło: opracowanie własne ..... 27
Listing 8. Przykład pętli w szablonie Angular wypisującej imiona klientów w postaci listy. Źródło: opracowanie własne ..... 27
Listing 9. Przykładowy kod JS tworzący obiekt car. Źródło: opracowanie własne..... 28
Listing 10. Przykład interfejsu stworzonego w języku TypeScript. Źródło: opracowanie własne..... 28
Listing 11. Przykładowy kod deklarujący obiekt wraz z jego typem w TypeScript. Źródło: opracowanie własne ..... 28
Listing 12. Przykładowy kod tworzący listę z odznakami. Źródło: (Bootstrap list-group, 2021) ..... 29
Listing 13. Przykład instrukcji INSERT. ..... 29
Listing 14. Przykładowa instrukcja CREATE ..... 30
Listing 15. Fragment klasy CustomerController.java. Źródło: opracowanie własne..... 33
Listing 16. Metoda findById zwracająca klienta o konkretnym numerze id. Źródło: opracowania własne ..... 34
Listing 17. Metoda create w kontrolerze odpowiadająca za utworzenie nowego klienta..... 34
Listing 18. Przykład klasy typu Entity na podstawie CustomerProduct ..... 35
Listing 19. Relacja @ManyToOne w klasie CustomerProduct ..... 36
Listing 20. Relacja @OneToMany w klasie CustomerProduct ..... 36
Listing 21. Interfejs repozytorium na przykładzie UserRepository ..... 37
Listing 22. Przykład klasy Service na przykładzie CustomerService..... 38
Listing 23. Przykład klasy LoginRequest ..... 39
Listing 24. Metoda odpowiedzialna za logowanie w klasie AuthenticationController ..... 40
Listing 25. Przykład wykorzystania funkcji ResponseEntity..... 40
Listing 26. Klasa WebSecurityConfig ..... 41
Listing 27. Fragement klasy UserDetailsImpl..... 42
Listing 28. Fragement klasy AuthEntryPointJwt ..... 43
Listing 29. Metoda doFilterInternal z klasy AuthTokenFilter ..... 44
Listing 30. Klasa JwtUtils ..... 44
Listing 31. Implementacja endpointu do rejestracji nowego użytkownika..... 45
Listing 32. Przykład komponentu na podstawie klasy CustomerListComponent ..... 50

Listing 33. Przykład dokumentu html na podstawie customer-list.component.html ..... 51
Listing 34. Fragment kodu wywołujący komponent AddCustomerComponent..... 52
Listing 35. Deklaracja zmiennych w komponencie AddCustomerproductComponent..... 53
Listing 36. Fragment funkcji submit() z komponentu AddCustomerproductComponent wywołujący emisję zdarzenia na zmiennej added..... 53
Listing 37. Funkcja w komponencie CustomerDetailsComponent, która zostaje wywołana po wyemitowaniu zdarzenia z komponentu AddCustomerproductComponent..... 53
Listing 38. Przykład serwisu na podstawie klasy TaskService..... 54
Listing 39. Przykład filtra typu Pipe ..... 55
Listing 40. Klasa AuthService z aplikacji webowej ..... 56
Listing 41. Klasa TokenStorageService z aplikacji webowej ..... 56
Listing 42. Implementacja klasy AuthInterceptor z aplikacji webowej..... 57

# Spis rysunków

Rysunek 1. Interfejs aplikacji TeamSupport. Źródło: opracowanie własne ..... 7
Rysunek 2. Interfejs aplikacji FocalScope. Źródło: opracowanie własne..... 8
Rysunek 3. Interfejs aplikacji Wix Answers. Źródło: opracowanie własne ..... 9
Rysunek 4. Interfejs aplikacji SupportBee.Źródło: opracowanie własne ..... 10
Rysunek 5. Diagram przypadków użycia: zarządzanie produktami. Źródło: opracowanie własne..... 13
Rysunek 6. Diagram przypadków użycia: zarządzanie urządzeniami. Źródło: opracowanie własne..... 14
Rysunek 7. Diagram przypadków użycia: zarządzanie zgłoszeniami. Źródło: opracowanie własne..... 14
Rysunek 8. Diagram przypadków użycia: zarządzanie klientami. Źródło: opracowanie własne ..... 15
Rysunek 9. Diagram przypadków użycia: zarządzanie użytkownikami. Źródło: opracowanie własne..... 15
Rysunek 10. Diagram klas. Źródło: opracowanie własne..... 17
Rysunek 11. Interfejs narzędzia Spring Initializr. Źródło: (Spring Quick Start, 2021) ..... 23
Rysunek 12. Wyniki testów czasu budowania aplikacji przez Gradle oraz Maven. Źródło: (Gradle - Maven vs Gradle, 2021) ..... 25
Rysunek 13. Przykład listy z odznakami. Źródło: (Bootstrap list-group, 2021)..... 28
Rysunek 14. Graficzna reprezentacja logiki systemu. Źródło: opracowanie własne..... 31
Rysunek 15. Struktura katalogów w projekcie..... 32
Rysunek 16. Struktura aplikacji webowej we frameworku Angular ..... 47
Rysunek 17. Widok menu z poziomu administratora ..... 48
Rysunek 18. Struktura folderu komponentu customer-list..... 49
Rysunek 19. Widok z zakładki All Customers ..... 52
Rysunek 20. Przykład testowana z wykorzystaniem aplikacji Postman. Źródło: opracowanie własne..... 58

# Spis tabel

Tabela 1. Scenariusz przypadku użycia: Dodaj zgłoszenie, Źródło: opracowanie własne ..... 16
Tabela 2. Wymagania funkcjonalne. Źródło: opracowanie własne ..... 18
Tabela 3. Wymagania niefunkcjonalne. Źródło: opracowanie własne ..... 21
Tabela 4. Przykłady metod tłumaczonych na JPQL w Spring Data JPA..... 37
Tabela 5. Testowanie. Dodawanie klienta ..... 58
Tabela 6. Testowanie. Usuwanie klienta..... 59