

W A R S Z A W S K A
WYŻSZA SZKOŁA INFORMATYKI

PRACA DYPLOMOWA
STUDIA PIERWSZEGO STOPNIA

Katarzyna Goźlińska

Numer albumu 8312

**Projekt i częściowa implementacja informatycznego systemu
obsługi linii lotniczej**

Promotor:

Mgr inż. Rosiek Zbigniew

Praca spełnia wymagania stawiane pracom dyplomowym na studiach pierwszego stopnia.

W A R S Z A W A 2020

Spis treści

1.	WSTĘP	5
1.1	Cel pracy	6
1.2.	Zakres pracy	6
2.	Przedstawienie i analiza istniejących rozwiązań	8
2.1.	Amadeus Altea Reservation Desktop Web	8
2.2.	SabreSonic CSS	10
2.3.	Merlot Aero.....	12
2.4.	Podsumowanie oraz wnioski	14
3.	Określenie wymagań systemu	17
3.1.	Wymagania funkcjonalne	17
3.1.1.	Aktorzy	18
3.1.2.	Opis funkcji systemu	18
3.1.3.	Diagram hierarchii funkcji	22
3.1.4.	Diagram przypadków użycia	23
3.1.5.	Diagram związków encji	25
3.2.	Wymagania pozafunkcjonalne.....	26
4.	Projekt systemu	30
4.1.	Architektura systemu	30
4.2.	Logika systemu	32
4.3.	Baza danych.....	55
4.3.1.	Diagram relacji bazy danych	55
4.3.2.	Opis tabel bazy danych.....	56
4.4.	Projekt interfejsów użytkownika	60
4.4.1.	Projekt interfejsów użytkownika niezalogowanego.....	62
4.4.2.	Projekt interfejsów użytkownika zalogowanego.....	63
4.4.3.	Projekt interfejsów dyspozytora.....	67
4.4.4.	Projekt komunikatów systemu.....	70
5.	Implementacja systemu	71
5.1.	Implementacja bazy danych.....	71
5.2.	Implementacja logiki systemu	77
5.3.	Implementacja interfejsu użytkownika	87
5.4.	Przebieg implementacji.....	93
6.	Testy systemu	95
6.1.	Testy jednostkowe	95
6.2.	Testy bezpieczeństwa	97
6.3.	Testy zgodności.....	99
7.	Podsumowanie.....	107
	WYKAZ LITERATURY	108
	Źródła literackie.....	108
	Źródła pozaliterackie	108
	Spis ilustracji	109
	Spis tabel	111
	ZAŁĄCZNIKI.....	112
	Instrukcja instalacji systemu obsługi linii lotniczej.....	112

WYKAZ UŻYTYCH W TEKŚCIE SKRÓTÓW

Lp.	Skrót	Opis
1	ULC	Urząd lotniska cywilnego
2	MVC	Z ang. Model-View-Controller. Wzorzec architektoniczny.

Tabela 1. Wykaz użytych w tekście skrótów.

1. WSTĘP

Lotnictwo jest jednym z tych rynków, które w Polsce z roku na rok rozwijają się coraz bardziej i cieszą się sporą popularnością. Taki wniosek można wysnuć patrząc na ilość pasażerów obsługiwanych przez polskie lotniska według statystyk ULC. W roku 2019 polskie lotniska obsługiwały prawie 49 milionów pasażerów – jest to wzrost o 7,1% w porównaniu z poprzednim rokiem. [1p, s. 1]. Wzrost popularności tanich przewoźników zdecydowanie zachęca polskich podróżujących do wylotów. Tacy przewoźnicy cieszą się bowiem dużą popularnością. W 2019 roku Ryanair oraz Wizz Air obsługiwały razem prawie połowę podróżujących korzystających z polskich lotnisk. [2p, s.1].

Pomimo tego, iż w Polsce a także i na całym świecie rynek lotnictwa rozrasta się nieustannie, raporty wskazują na to, że cyfryzacja w dalszym ciągu dopiero raczkuje w większości linii lotniczych. Według ankiety wykonanej przez PROS, mimo iż wiele firm rozpoczęło proces cyfryzacji, jedynie 8% zajmowało się tym przez ponad trzy lata. [3p, s.1] Duże linie lotnicze, które były w stanie przeznaczyć duże fundusze na rozwój cyfryzacji mogą zbierać już żniwo swoich starań. Badania wskazują na wiele pozytywnych efektów takich działań. Są to między innymi lepsze doświadczenia klienta, a co za tym idzie wzrost przychodów nawet o 15% u linii lotniczych, które personalizowały swoje oferty. [4p, s.1] Dodatkowo zauważyć można, że ucyfrowienie w sektorze zarządzania mogło by doprowadzić do wzrostu produktywności i poprawie efektywności poprzez zautomatyzowanie wielu procesów. [5p, s.40]

Co jakiś czas lotnictwo musi się zmierzyć z metaforycznym czarnym łabędziem - nieprzewidywalnym zdarzeniem o olbrzymim wpływie na rzeczywistość. Począwszy od I wojny w Zatoce Perskiej, ataku terrorystycznego z 11 września 2001 r. aż po kryzys finansowy lat 2007-2009 i wybuch wulkanu Eyjafjallajökull, który kompletnie sparaliżował lotnictwo w Europie, w roku 2020 rynek został zmuszony do walki ze skutkami epidemii koronawirusa, która wyrządziła szkody na niespotykaną dotychczas skalę. Poprzednio to właśnie atak terrorystyczny z 11 września był największym wydarzeniem zatrzymującym lotnictwo – minęło 6 lat zanim rynek odzyskał swoją sprawność, skutkiem czego nastąpiły nieodwracalne zmiany w sposobie przeprowadzania lotów i kwestiach bezpieczeństwa. [1, s. 39] W obecnym momencie wiadomo już, że koronawirus odbił się na liniach lotniczych o wiele mocniej, zmuszając je do zmniejszenia wydajności do zaledwie 10% i permanentnego uziemienia części samolotów. [1, s. 40] Pomimo tego, lotnictwo jest jednym z przemysłów, który raz za

razem udowadniał, że potrafi się podnieść nawet z największych kryzysów. Obecne założenia są dość pozytywne i zakładają powrót do normalnego funkcjonowania najwcześniej do 2023 r. [1, s. 43].

Poprzez obserwację środowiska lotniczego i potencjału jego cyfryzacji, powstał pomysł utworzenia systemu obsługi linii lotniczej, który mógłby pomóc w zarządzaniu i obsługiwaniu linii lotniczej.

1.1 Cel pracy

Celem niniejszej pracy jest przeprowadzenie analizy istniejących rozwiązań na rynku, wraz z zaprojektowaniem, implementacją oraz przetestowaniem aplikacji internetowej, która ułatwi zarządzanie zasobami małej linii lotniczej. Aplikacja zostanie stworzona w oparciu o technologię Microsoft .NET Core 3.1 z wykorzystaniem architektury klient-serwer. Do przechowywania danych zostanie użyta relacyjna baza danych MSSQL. Aby umilić patrzenie na interfejs graficzny, zastosuję framework Angular oraz TypeScript.

Podstawowymi wymaganiami funkcjonalnymi systemu będą: umożliwienie pasażerom łatwego przeglądania dostępnych lotów i zarezerwowania biletów, tworzenie i śledzenie lotów przez dyspozytorów, zarządzanie załogami a także umożliwienie załogom sprawdzenie swojego grafiku lotów.

Korzyści spodziewane z wdrożenia systemu obsługi linii lotniczej:

- Usystematyzowanie prowadzonych przelotów.
- Redukcja czasu, potrzebnego do zarezerwowania biletu.
- Możliwość śledzenia statusu zamówienia w czasie rzeczywistym.
- Automatyzacja rozliczania godzin
- Archiwizacja danych

1.2. Zakres pracy

Praca swoim zakresem obejmuje:

- Przedstawienie i analizę istniejących rozwiązań.
- Określenie wymagań funkcjonalnych i нефункциональных systemu.

- Określenie architektury systemu, opracowanie projektu logiki (diagramy i opis klas, obiektów i ich powiązań), projektu bazy danych oraz projektu interfejsu użytkownika.
- Implementację bazy danych, logiki oraz interfejsów użytkownika wybranych funkcji zdefiniowanego w projekcie oprogramowania.
- Testowanie aplikacji poprzez przygotowanie i przeprowadzenie testów jednostkowych, bezpieczeństwa oraz zgodności.
- Podsumowanie oraz wnioski.

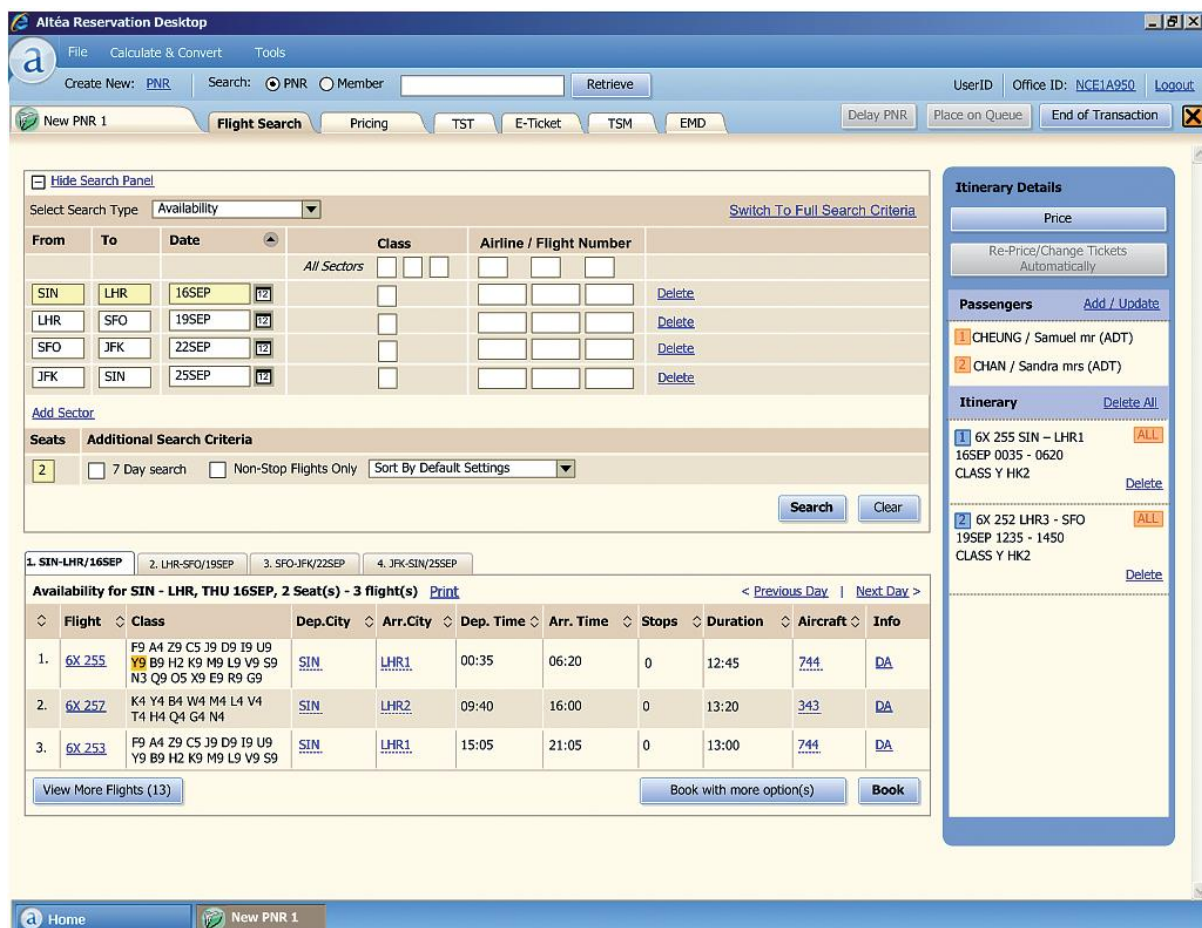
2. Przedstawienie i analiza istniejących rozwiązań

Jednym z etapów w procesie określania wymagań dla tworzenia nowych produktów informatycznych jest przegląd oraz zapoznanie się z istniejącymi już na rynku rozwiązaniami w celu ich możliwego rozszerzenia, bądź dostosowania do naszych potrzeb. [2, str. 57]

W rozdziale tym zostaną przedstawione trzy wybrane systemy, których cechy zostaną opisane wraz z dokonaniem analizy wad i zalet tych aplikacji. Na koniec podane zostaną wnioski oraz porównanie wszystkich rozwiązań.

2.1. Amadeus Altea Reservation Desktop Web

System Altea Reservation Desktop Web to aplikacja zaprojektowana przez firmę Amadeus IT Group. Jest to płatna aplikacja bazująca na subskrypcji, która umożliwia agentom tworzenie oraz zarządzanie lotami, a co za tym idzie ustalaniem cen biletów, co również może zostać zrobione automatycznie przez aplikację. Jedną z możliwości jest podgląd listy wszystkich lotów, listy pasażerów oraz dokumenty związane z płatnościami. Koszty są indywidualnie dostosowywane do danej firmy jednak najczęściej jest to opłata za każdą rezerwację obsługiwaną przez firmę Amadeus – około 4.30 euro za jedną transakcję. [6p, s.1]



Rysunek 1. Zrzut ekranu pokazujący stronę wyszukiwania rezerwacji systemu Altea Reservation

System cechuje prosty, przejrzysty interfejs o delikatnych kolorach. Skróty do podstawowych funkcji znajdują się w menu głównym umieszczonym w górnej części strony. Na dole znajduje się tabela z lotami spełniającymi kryteria wyszukiwania podanymi w środkowej części aplikacji.

Główne cechy programu to:

- Obsługa sprzedaży biletów,
- Automatyczna wycena biletów,
- Obsługa płatności za bilety,
- Obsługa zwrotów,
- Wgląd na dokumenty związane z płatnościami,
- Definiowanie bazy klientów,
- Podgląd na podstawowe dane pasażerów a także ich informacje kontaktowe,
- Możliwość tworzenia lotów,
- Wyświetlanie listy lotów,
- Wyświetlanie informacji o wybranym locie,

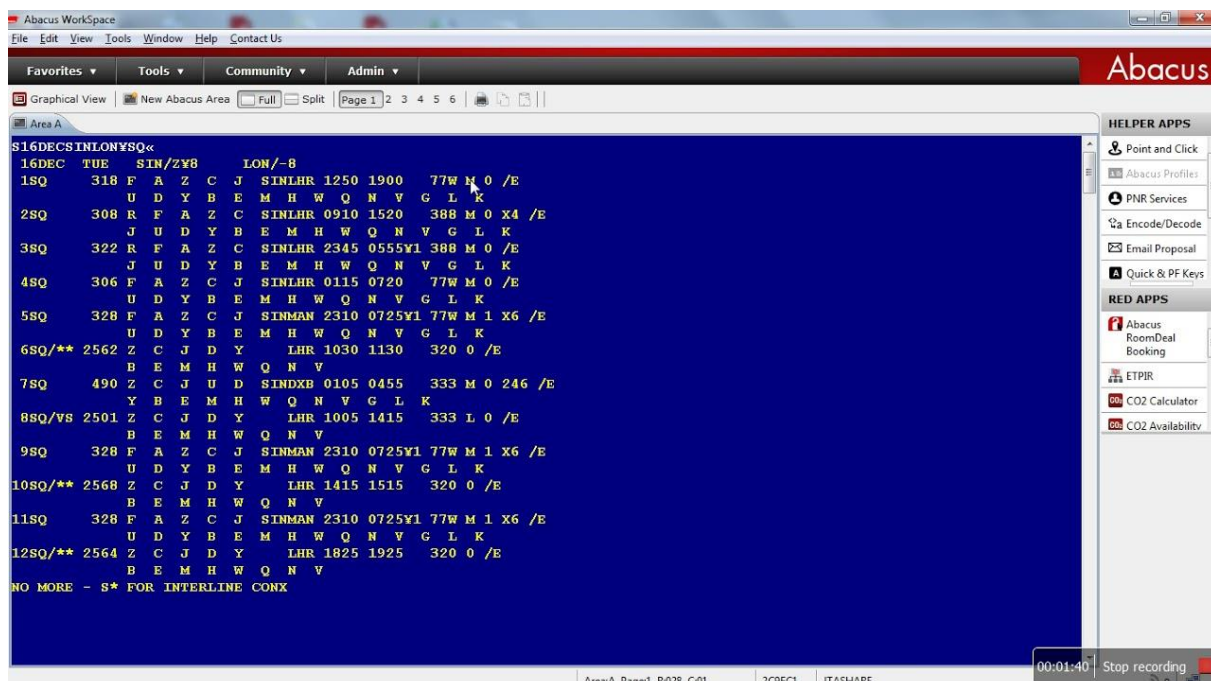
- Filtrowanie danych,
- Historia lotów,
- Tworzenie kont pracowników.

Zaletą tej aplikacji jest głównie jej niezawodność. Amadeus jest firmą, która powstała ponad 30 lat temu i obsługuje największych graczy na rynku – linie lotnicze takie jak KLM, Qantas czy Southwest Airlines. Stosują oni znane i skuteczne od lat metody zarządzania. Dodatkowym atutem jest możliwość modyfikacji oprogramowania w ramach potrzeby linii lotniczej – np. poprzez dodanie obsługi pasażerów często korzystających z usług danej firmy.

Wadą jest natomiast cena. Na chwilę obecną Amadeus jest największym graczem na rynku jeżeli chodzi o systemy zarządzania liniami lotniczymi - główną konkurencją są Sabre i Travelport. Z powodu posiadania tak dużej różnicy w siłach ma on możliwość dość swobodnego wyboru ceny za swoje oprogramowanie co skutkuje niezadowoleniem linii lotniczych, które muszą płacić duże pieniądze wiedząc, że ciężko jest wybrać alternatywę. Na chwilę obecną to głównie duże i starsze linie lotnicze korzystają z tego systemu. Dodatkowo interfejs graficzny systemu jest bardzo przestarzały co może zostać negatywnie odebrane przez klientów biorąc pod uwagę, że innowacja i nowoczesność są ważnymi elementami wyglądu aplikacji.

2.2. SabreSonic CSS

SabreSonic CSS to system stworzony w celu ułatwienia liniom lotniczym obsługi pasażerów. Zawiera on również funkcje zarządzania inwentarzem i kontrolą wylotów. Firma Sabre ma w planach przenieść całą infrastrukturę systemu do rozwiązań chmurowych do roku 2023. Podobnie jak w przypadku poprzednika, ceny za używanie systemu są indywidualnie dostosowywane do firmy planującej zakup, jednak w tym wypadku brak jest informacji publicznej o średnim koszcie lub formie płatności.



Rysunek 2. Zrzut ekranu pokazujący widok rozkładu lotów

System cechuje minimalistyczny interfejs wyposażony w konsolę używaną do wpisywania komend, znajdującą się w centralnej części okna. Na górze widoczne jest menu główne używane do przełączania się między różnymi narzędziami. Po prawej stronie znajduje się lista aplikacji pomocniczych typu drukowanie czy kalkulator, które można uruchomić poprzez kliknięcie na element listy.

Główne cechy programu to:

- Obsługa sprzedaży biletów,
- Automatyczna wycena biletów,
- Obsługa płatności za bilety,
- Obsługa zwrotów,
- Umożliwienie odprawy online,
- Definiowanie bazy klientów,
- Podgląd na podstawowe dane pasażerów a także ich informacje kontaktowe,
- Możliwość tworzenia lotów,
- Wyświetlanie listy lotów,
- Wyświetlanie informacji o wybranym locie,
- Filtrowanie danych,
- Zarządzanie inwentarzem,
- Tworzenie kont pracowników,

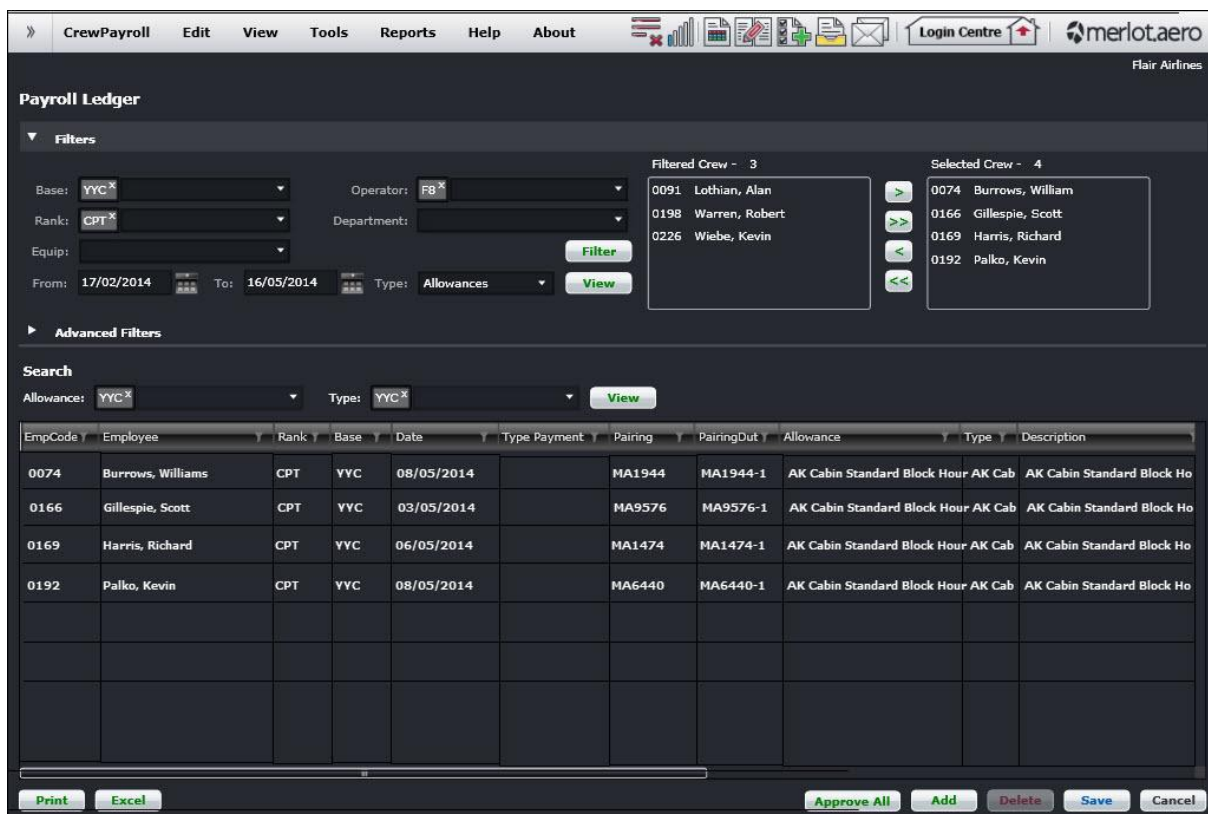
- Wysyłanie wiadomości

Zaletą aplikacji jest jej efektywność. W przypadku systemów wspomagających obsługę linii lotniczej, Sabre jest też liderem we wprowadzaniu nowych rozwiązań. Jako pierwsi zdecydowali się na przeniesienie całej architektury do chmury i planują zakończyć proces migracji w 2023 roku. Korzystają z bardzo wydajnych algorytmów przeliczających ceny lecz sam system jest również bardzo efektywny. Linie korzystające z tego systemu to np. Aeroflot, American Airlines. Dodatkowo również duże linie o niskich kosztach takie jak JetBlue czy WestJet korzystają z owego systemu.

Wadą systemu, która pojawia się w prawie każdej recenzji użytkowników, jest bardzo archaiczny interfejs, porównywany do systemów z lat 80. Wprowadzanie danych następuje poprzez konsolę co jest bardzo niewygodne i nieintuicyjne. Wymaga to szkolenia nowych pracowników w celu nauki bardzo specyficznej składni używanej w konsoli. Samodzielna nauka jest prawie niemożliwa. Dodatkowym problemem jest fakt, że system zawiesza się przy wolniejszych połączeniach internetowych.

2.3. Merlot Aero

Merlot Aero oferuje nowoczesny system zarządzania załogami i lotami. Umożliwia on tworzenie i zarządzanie załogami a także wgląd na ilość przepracowanych godzin. Informuje on o konfliktach w planie lotów a także o przekroczonym czasie w powietrzu – limit czasowy można ustawić w zależności od regulaminu danej linii lotniczej. Dodatkowo system umożliwia tworzenie lotów i wgląd na ich dane a także zapisuje historię poprzednich zleceń. Cena systemu jest dobierana indywidualnie dla linii lotniczej i jest ona oferowana w formie subskrypcji płatnej co miesiąc lub co rok. Dostępny jest również darmowy okres próbny.



Rysunek 3. Zrzut ekranu przedstawiający rejestr listy plac załóg

System posiada nowoczesny i przejrzysty interfejs, który jest bardzo prosty w obsłudze. Jego stylistyka również umila czas pracy poprzez zastosowanie nierzających kolorów. Możliwe jest filtrowanie poprzez różne kryteria: ranga pilota, utworzony skład, baza operacyjna itd.

Główne cechy programu to:

- Możliwość tworzenia lotów,
- Wyświetlanie listy lotów,
- Wyświetlanie informacji o wybranym locie,
- Filtrowanie danych,
- Historia lotów,
- Przeglądanie raportów z lotów,
- Tworzenie kont pracowników,
- Przypisywanie pracowników do danego lotu,
- Automatyczne obliczanie wypłat należnych za wykonanie pracy,
- Ostrzeganie przed kolidującymi lotami.

Zaletą systemu jest przyjazny dla użytkowników wygląd. Poza prostotą menu, interfejs ma ciemną szatę barw co jest o wiele przyjemniejsze dla oczu niż interfejs o

jaskrawych barwach. Merlot Aero to jedno z najnowocześniejszych rozwiązań, które jest stale rozwijane. Oferuje również ponad 30 rozszerzeń z których firmy mogą korzystać za dodatkową opłatą.

Wadą systemu jest jego awaryjność. Niestety pomimo innowacyjności systemu, użytkownicy mogą spotkać się z szeregiem błędów. W 2013 roku stały się one na tyle poważne, że linia lotnicza IslandAir pozwała Merlot Aero za kontynuujące problemy z systemem – między innymi zawieszanie się, problemy z przeliczaniem płac, brak dostępu do systemu z poza sieci linii lotniczej a także niedostateczną obsługę klienta w razie awarii.

2.4. Podsumowanie oraz wnioski

Podsumowanie cech funkcjonalnych prezentowanych wyżej systemów pomagających w obsłudze linii lotniczej przedstawiono w tabeli poniżej. W pierwszej kolumnie znajduje się nazwa danej cechy, a w kolejnych 3 za pomocą znaku „X” przedstawiono czy dany system posiadał wybraną cechę.

Funkcja	Amadeus Altea Reservation Desktop Web	SabreSonic CSS	Merlot Aero
Dostęp przez przeglądarkę internetową dla klienta	X	X	
Dostęp przez przeglądarkę internetową dla obsługi linii lotniczej	X		X
Obsługa sprzedaży biletów	X	X	
Obsługa płatności za bilety	X	X	
Odprawa Online		X	
Definiowanie bazy klientów	X	X	

Podgląd podstawowych danych pasażerów i ich informacji kontaktowych	X	X	
Możliwość tworzenia lotów	X	X	X
Wyświetlanie listy lotów	X	X	X
Wyświetlanie informacji o wybranym locie	X	X	X
Filtrowanie danych	X	X	X
Historia lotów	X		X
Przeglądanie raportów z lotów			X
Zarządzanie inwentarzem		X	
Tworzenie kont pracowników	X	X	X
Przypisywanie pracowników do danego lotu			X
Wysyłanie wiadomości		X	

Tabela 2 Podsumowanie funkcji oraz cen trzech wybranych systemów. Opracowanie własne.

Z przedstawionej analizy wynika, że system obsługi linii lotniczej powinien co najmniej spełniać wymagania funkcjonalne, takie jak:

- Dostęp przez przeglądarkę internetową dla klienta
- Dostęp przez przeglądarkę internetową dla obsługi linii lotniczej
- Obsługa sprzedaży biletów
- Obsługa płatności za bilety
- Definiowanie bazy klientów
- Podgląd podstawowych danych pasażerów i ich informacji kontaktowych
- Możliwość tworzenia lotów
- Wyświetlanie listy lotów
- Wyświetlanie informacji o wybranym locie

- Historia lotów
- Tworzenie kont pracowników

Dodatkowo, wyżej wymieniony system powinien spełniać następujące cechy niefunkcjonalne:

- Prosty interfejs graficzny – system musi być tak zbudowany, aby nie utrudniać użytkownikom korzystania z programu.
- Szybkość – system powinien reagować na żądania użytkownika w akceptowalnym czasie a także uaktualniać dane w możliwie jak najszybszym czasie.
- Bezpieczeństwo – ponieważ niektóre dane przechowywane w systemie mogą być sensytywne, muszą one być odpowiednio chronione.
- Dostępność – system powinien być dostępny w trybie 24h/7dni/365dni w roku z ewentualnymi przerwami moderacyjnymi.

3. Określenie wymagań systemu

Podstawą do określenia wymagań systemu informatycznego są procesy związane z pozyskiwaniem informacji na temat danego projektu. Polegają one na przemianie celów klienta oraz interesariuszy na konkretne wymagania, które zapewnią osiągnięcie owych celów. Jest to prawdopodobnie jeden z cięższych etapów tworzenia programu, ponieważ klient niekoniecznie rozumie jak tworzy się oprogramowanie, wie jedynie to co chciałby zobaczyć jako rezultat. Ciężko jest im przewidzieć w jaki sposób system będzie faktycznie używany. Dodatkowo cele użytkowników mogą być sprzeczne. Ostateczne zdanie często należy jednak to zleceniodawcy, co nie zawsze skutkuje najlepszymi rezultatami. [7p, s.4]

Dobrze opracowana specyfikacja wymagań powinna kierować się różnymi dobrymi praktykami, które ułatwiają ten proces i zapobiegają popełnianiu błędów:

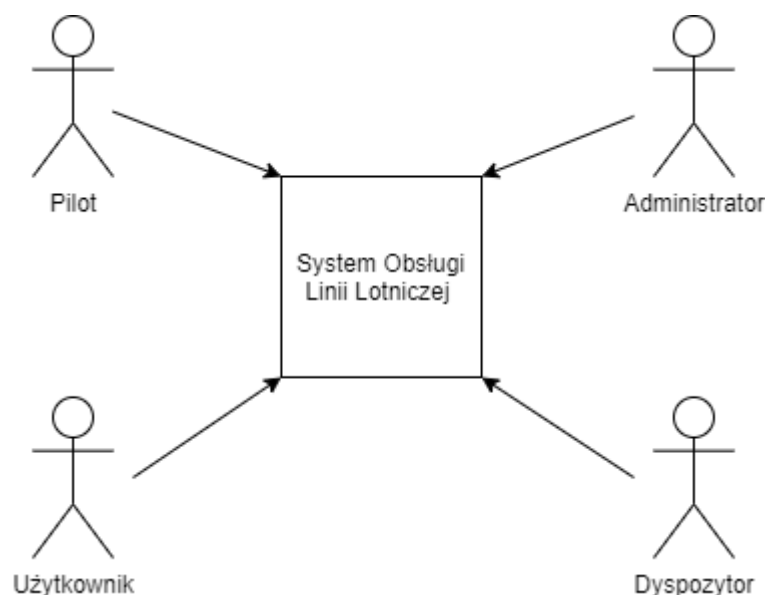
- Kompletność oraz brak sprzeczności z innymi dokumentami.
- Opisanie zewnętrznego zachowania się systemu w przeciwieństwie do sposobu jego realizacji.
- Branie pod uwagę możliwości zmiany wymagań wobec systemu – a co za tym idzie, możliwość łatwej modyfikacji struktury i stylu dokumentu, jednocześnie zachowując spójność.
- Przemyślenie ograniczeń przy jakich system będzie pracować, a także jego skrajne lub niepożądane zachowania.
- Zachowanie porozumienia między projektantami a użytkownikami systemu, jednocześnie nie lekceważąc klienta.
- Ułatwienie wyjaśniania wymagań poprzez referencje do istniejącego oprogramowania. [7p, s.6]

3.1. Wymagania funkcjonalne

Wymagania funkcjonalne opisują czynności, które system ma wykonywać. Obejmuje to czynności takie jak określenie rodzajów użytkowników korzystających z systemu, a także tych niezbędnych do działania systemu. Następnie należy określić w jaki sposób będą oni korzystać z systemu. Kolejnym krokiem jest określenie funkcji systemu a także systemów zewnętrznych. Ostatecznie należy zająć się ustaleniem struktur, przepisów prawnych, statutów, itd., które będą w mniejszym lub większym stopniu wpływać na funkcje w systemie.

3.1.1. Aktorzy

Poniższa ilustracja przedstawia zidentyfikowanych aktorów Systemu Obsługi Linii Lotniczej



Rysunek 4. Aktorzy systemu obsługi linii lotniczej. Opracowanie własne.

Nazwa	Opis
Użytkownik	Osoba rejestrująca się w systemie linii lotniczej, mająca możliwość przeglądania lotów i zarezerwowania miejsca w samolocie. Może również edytować swoje dane.
Administrator	Osoba administrująca systemem obsługi linii lotniczej, poprzez zarządzanie strukturą systemu, oraz wspieraniu innych aktorów w razie wystąpienia problemów.
Pilot	Osoba pracująca dla linii lotniczej, na stanowisku odpowiedzialnym za dostarczanie ładunków (w tym pasażerów) do celu podróży. Ma możliwość przeglądania przypisanych do niej lotów.
Dyspozytor	Osoba pracująca dla linii lotniczej, która zajmuje się organizacją lotów i przydzielaniem załóg do lotów. Ma dostęp do informacji na temat lotów oraz załóg, a co za tym idzie może tworzyć nowe załogi składające się z pilotów.

Tabela 3. Charakterystyka aktorów. Opracowanie własne.

3.1.2. Opis funkcji systemu

Poniżej przedstawię dostępne funkcje podzielone na role użytkowników, przedstawione w formie opisowej.

3.1.2.1. Funkcje Administratora

1. **Tworzenie użytkownika** – użytkownik, który jest administratorem, ma możliwość wstępu do panelu administratorów, z którego wybiera funkcję tworzenia użytkownika. Ma on możliwość podania wszystkich danych lub tylko ich części, w tym roli użytkownika, która determinuje poziom dostępu do systemu.
2. **Wyświetlanie danych użytkowników** – użytkownik, który jest administratorem, ma możliwość wstępu do panelu administratorów, w którym znajduje się lista ukazująca wszystkich zarejestrowanych użytkowników systemu. Lista ta umożliwia podgląd podstawowych i rozszerzonych danych użytkownika (z wyłączeniem hasła). Jednocześnie, lista ta umożliwia dostęp do innych funkcji, które zostały opisane poniżej.
3. **Edytowanie danych użytkownika** – użytkownik, który jest administratorem, ma możliwość wstępu do panelu administratorów, w którym znajduje się lista ukazująca wszystkich użytkowników systemu. Z tej samej listy ma on możliwość edytowania danych użytkownika. Nie ma on możliwości sprawdzenia hasła użytkowników ale jednocześnie nie jest zmuszony do jego zmiany.
4. **Usuwanie użytkowników** – użytkownik, który jest administratorem, ma możliwość wstępu do panelu administratorów, w którym znajduje się lista ukazująca wszystkich użytkowników system. Tak samo jak w przypadku edytowania użytkowników, ma on możliwość usunięcia danego użytkownika z poziomu listy.

3.1.2.2. Funkcje dyspozytora

1. **Tworzenie załóg** – użytkownik, który jest dyspozytorem, ma dostęp do panelu dyspozytora, z którego wybiera funkcję tworzenia załogi. Ma on możliwość utworzenia nowej załogi i przypisanie do niej pilotów.
2. **Wyświetlanie danych załóg** – użytkownik, który jest dyspozytorem, ma dostęp do panelu dyspozytora, w którym znajduje się lista zawierająca podstawowe i szczegółowe dane dotyczące wszystkich utworzonych załóg. Z poziomu tej listy, dyspozytor ma również możliwość dostępu do innych funkcji, które są opisane poniżej.

3. **Edytowanie danych załóg** – użytkownik, który jest dyspozytorem, ma dostęp do panelu dyspozytora, w którym znajduje się lista ukazująca wszystkie utworzone załogi. Z tej samej listy ma on możliwość wybrania załogi, która chce edytować i wprowadzenie tych zmian. Dyspozytor nie może zmienić danych osobistych pilotów zawartych w danej załodze, może jedynie zmienić, którzy piloci są do niej przypisani.
4. **Usuwanie załóg** – użytkownik, który jest dyspozytorem, ma dostęp do panelu dyspozytora, w którym znajduje się lista ukazująca wszystkie utworzone załogi. Z tej samej listy ma on możliwość usunięcia wybranej załogi. Usunięcie załogi nie powoduje usunięcia pilotów do niej przypisanych.
5. **Tworzenie lotów** – użytkownik, który jest dyspozytorem, ma dostęp do panelu dyspozytora, z którego wybiera funkcję tworzenia lotu. Dzięki tej funkcji może on stworzyć nowy lot i jednocześnie przypisać do niego załogę, która go zrealizuje.
6. **Wyświetlanie danych lotów** – użytkownik, który jest dyspozytorem, ma dostęp do panelu dyspozytora, w którym znajduje się lista zawierająca dane o wszystkich utworzonych lotach. Poza zwykłymi danymi lotu, dyspozytor ma również możliwość sprawdzenia ilości zajętych miejsc, a także podejrzenie danych kontaktowych użytkowników w razie potrzeby nawiązania kontaktu. Dodatkowo może on zobaczyć, która załoga jest przypisana do danego lotu.
7. **Edytowanie danych lotów** – użytkownik, który jest dyspozytorem, ma dostęp do panelu dyspozytora, w którym znajduje się lista ukazująca wszystkie utworzone loty. Z poziomu tej listy, ma on możliwość zmiany danych wybranego lotu, nie ma jednak możliwości modyfikacji danych klientów, którzy w danym locie się znajdują.
8. **Usuwanie lotów** – użytkownik, który jest dyspozytorem, ma dostęp do panelu dyspozytora, w którym znajduje się lista ukazująca wszystkie utworzone loty. Z poziomu tej listy jest on w stanie usunąć wybrany lot. Usunięcie lotu nie spowoduje usunięcia przypisanej do niego załogi bądź klientów którzy są do niego przypisani. Spowoduje jednak zmiany statusu rezerwacji w przypadku klientów, którzy byli do tego lotu przypisani.
9. **Wyświetlanie danych rezerwacji** – użytkownik, który jest dyspozytorem, ma dostęp do panelu dyspozytora, w którym znajduje się lista ukazująca

wszystkie dokonane rezerwacje. Mają oni też możliwość szybkiego podglądu danych użytkownika, który dokonał danej rezerwacji.

10. **Edycja danych rezerwacji** – użytkownik, który jest dyspozytorem, ma dostęp do panelu dyspozytora, w którym znajduje się lista ukazująca wszystkie dokonane rezerwacje. Z poziomu tej samej listy jest on w stanie edytować rezerwacje. Rezerwacje nie mogą zostać usunięte – ich status może być jednak zmieniony aby ukazać, że np. rezerwacja została anulowana.

3.1.2.3. Funkcje pilota

1. **Wyświetlanie danych lotów** – użytkownik, który jest pilotem, ma dostęp do panelu pilota, w którym znajduje się lista dostępnych lotów. Pilot ma możliwość szybkiego sprawdzenia, które loty są przypisane do niego. Ma on możliwość obejrzenia szczegółowych danych na temat lotu, w którym jest częścią załogi.
2. **Edycja danych lotów** – użytkownik, który jest pilotem, ma dostęp do panelu pilota, w którym znajduje się lista dostępnych lotów. Lista ta umożliwia szybką edycję lotu. Pilot, który jest przypisany do wybranego lotu ma możliwość zmiany daty wylotu i przylotu a także miejsca startu i miejsca docelowego w razie ewentualnych zmian względem planu lotu a faktycznym stanem rzeczy. Ma on również możliwość zmiany statusu lotu. Nie może on zmieniać danych lotów, do których nie jest przypisany.

3.1.2.4. Funkcje wspólne

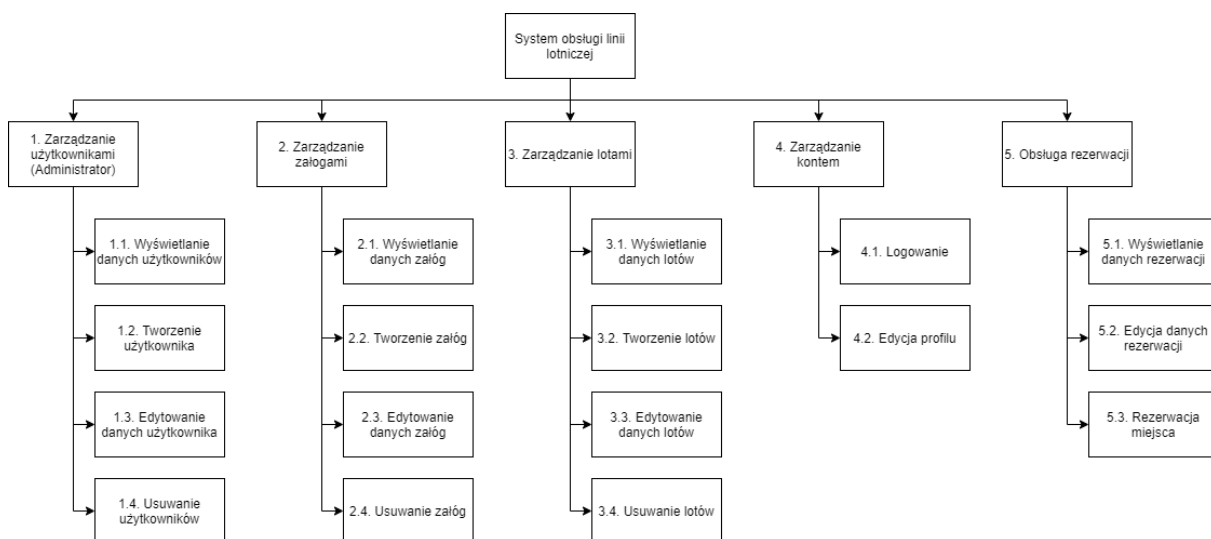
1. **Logowanie** – użytkownik podaje swój login i hasło na stronie logowania. Dane zostają zweryfikowane przez system. W przypadku poprawnej weryfikacji, użytkownik zostanie przekierowany do panelu odpowiadającego jego roli. W przypadku nieudanej próby logowania, użytkownik nie zostaje nigdzie przekierowany.
2. **Edycja profilu** – zalogowany użytkownik przechodzi do panelu profilowego, w którym ma możliwość edycji swojego profilu. Dane takie jak rola czy adres e-mail, mogą być zmienione tylko i wyłącznie przez użytkowników z rolą administratora.

3. **Wyświetlanie danych lotów** – zalogowany użytkownik ma dostęp do panelu klienta, w którym znajduje się lista lotów dla niego dostępnych. Użytkownik ma możliwość filtrowania lotów po miejscu startowym i docelowym a także po datach. Będzie on widzieć jedynie loty, które mają jeszcze wolne miejsca.
4. **Rezerwacja miejsca** – zalogowany użytkownik ma dostęp do panelu klienta, w którym znajduje się lista lotów. Z poziomu tej listy, użytkownik ma możliwość wyboru lotu a następnie zarezerwowanie w nim wolnego miejsca w samolocie.
5. **Wyświetlanie danych rezerwacji** – zalogowany użytkownik ma dostęp do panelu klienta, w którym znajduje się lista dokonanych przez niego rezerwacji. Z poziomu tej listy, użytkownik jest w stanie zobaczyć szczegółowe dane na temat rezerwacji a także lotu, którego ta rezerwacja dotyczy. Jedynie dyspozytorzy są w stanie zobaczyć rezerwacje dokonane przez innych użytkowników.

Należy dodać, że funkcje wspólne to w istocie funkcje klienta. System jednak pozwala administratorom, dyspozytorom i pilotom na bycie klientem.

3.1.3. Diagram hierarchii funkcji

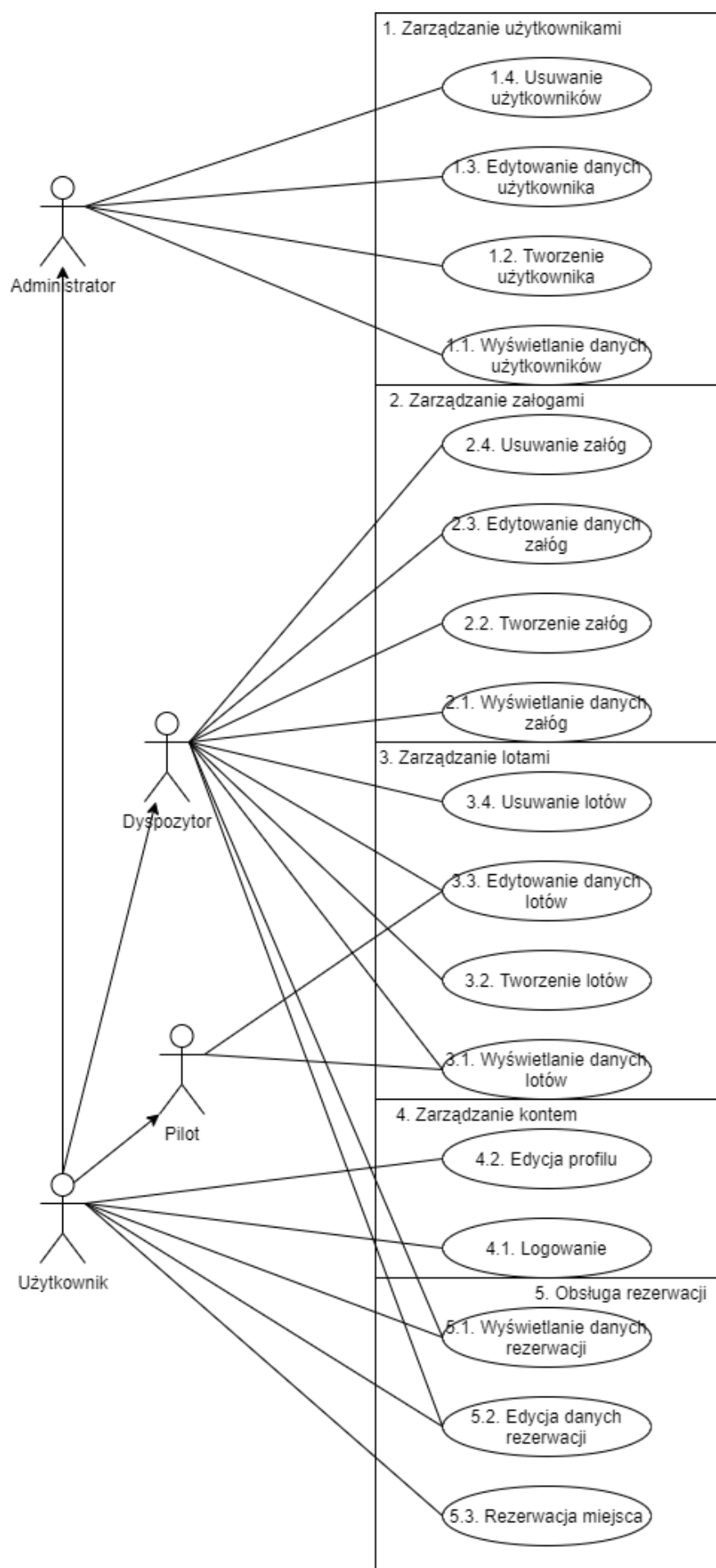
W celu zobrazowania i uporządkowania najważniejszych funkcji systemu, zostały one przedstawione poniżej w formie diagramu hierarchii funkcji.



Rysunek 5. Diagram hierarchii funkcji poziomów 1 - 2 najważniejszych funkcji systemu. Opracowanie własne

3.1.4. Diagram przypadków użycia

Niżej przedstawiony diagram przypadków użycia ma na celu przybliżenie nam planowanego działania systemu poprzez powiązanie funkcjonalności z użytkownikami docelowo korzystającymi z systemu, zwanymi również aktorami.

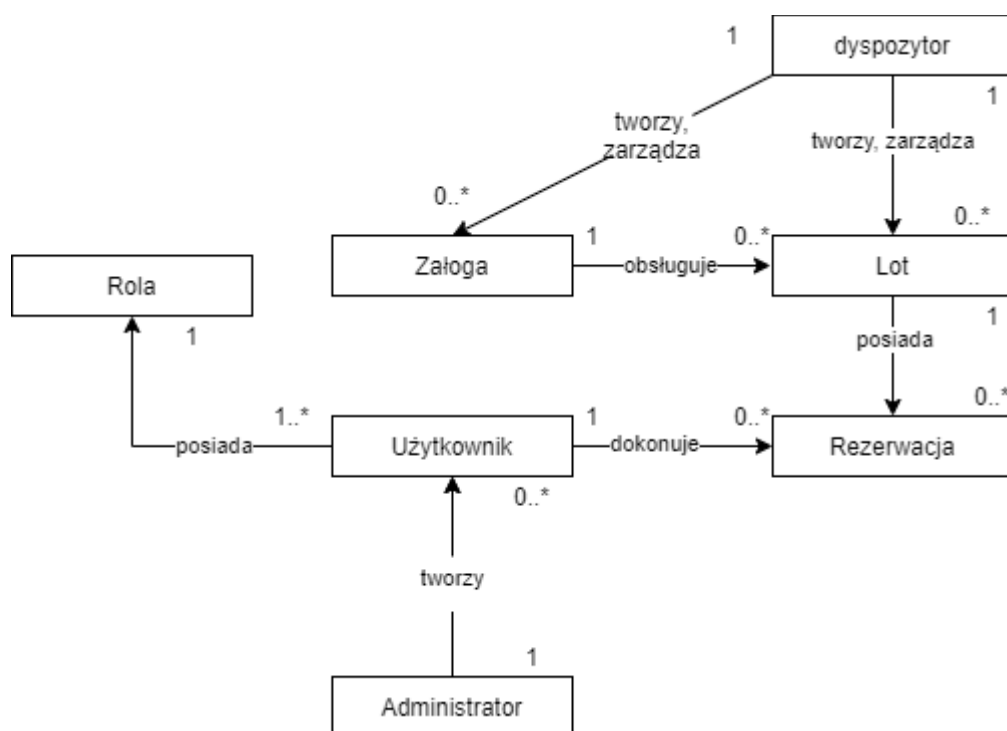


Rysunek 6. Diagram przypadków użycia. Opracowanie własne.

3.1.5. Diagram związków encji

Równie istotne jak funkcje są dane wywodzące się z działań funkcji systemu. W celu uporządkowania zgromadzonej wiedzy zastosowany został diagram encji [2, str. 93].

Poniżej przedstawiony został diagram związków encji w celu zrozumienia najważniejszej logiki projektowanego systemu. Encje nie zawierają atrybutów, bowiem zostaną one zdefiniowane w rozdziale projektu systemu.



Rysunek 7. Diagram encji. Opracowanie własne.

Opis encji oraz relacji między nimi został przedstawiony w poniższych tabelach:

Nr	Nazwa encji	Opis encji
1	Użytkownik	Użytkownik systemu, działający jako klient
2	Rola	Rola użytkownika.
3	Administrator	Administrator systemu.
4	Rezerwacja	Zarezerwowane miejsce w samolocie.
5	Lot	Lot obsługiwany przez linie lotniczą

6	Załoga	Grupa pilotów stworzona przez dyspozytorów, której celem jest obsługa lotu.
7	Dyspozytor	Pracownik linii lotniczej, który tworzy i zarządza nie tylko załogami, ale też lotami.

Tabela 4. Opis encji. Opracowanie własne.

Poniżej opis relacji.

Nr.	Encja I	Encja I	Typ relacji	Nazwa relacji	Opis
1	Rola	Użytkownik	1:N	Posiada	Jedną rolę posiada wielu użytkowników.
2	Administrator	Użytkownik	1:N	Tworzy, Zarządza	Jeden administrator tworzy wielu klientów.
3	Użytkownik	Rezerwacja	1:N	Dokonuje	Jeden użytkownik dokonuje wielu rezerwacji.
4	Lot	Rezerwacja	1:N	Posiada	Jeden lot posiada wiele rezerwacji.
5	Załoga	Lot	1:N	Obsługuje	Jedna załoga obsługuje wiele lotów.
6	Dyspozytor	Załoga	1:N	Tworzy, Zarządza	Jeden dyspozytor tworzy i zarządza wieloma załogami.
7	Dyspozytor	Lot	1:N	Tworzy, Zarządza	Jeden dyspozytor tworzy i zarządza wieloma lotami.

Tabela 5. Opis relacji. Opracowanie własne.

3.2. Wymagania pozafunkcjonalne

Wymagania poza funkcjonalne opisują ograniczenia i charakterystyki, przy których system powinien realizować swoje funkcje. Nie są związane ze stroną funkcjonalną. Opisują cechy takie jak, np.: ograniczenia biznesowe, ograniczenia związane z przepisami, itp.

W celu opracowania wymagań pozafunkcyjnych Systemu Obsługi Linii Lotniczej został użyty model standardu ISO/IEC 25010. Standard ten składa się z ośmiu kategorii, które określają właściwości oprogramowania.

Tymi kategoriami są:

1. Funkcjonalna odpowiedniość (ang. Functional Suitability) – funkcjonalna kompletność, poprawność i odpowiedniość.
2. Wydajność (ang. Performance Efficiency) – wydajność czasowa, zużycie zasobów, oczekiwana wydajność (ang. Capacity).
3. Kompatybilność (ang. Compatibility) – współistnienie, interoperacyjność.
4. Użyteczność (ang. Usability) – rozpoznawalność zastosowania, łatwość nauczania się, łatwość operowania, ochrona użytkownika przed błędami, estetyka interfejsu użytkownika, dostępność personalna.
5. Niezawodność (ang. Reliability) – dojrzałość, dostępność techniczna, odporność na wady, odtwarzalność.
6. Bezpieczeństwo (ang. Security) – poufność, integralność, niezaprzeczalność, identyfikowalność, autentyczność.
7. Łatwość utrzymania (ang. Maintainability) – modułowość, łatwość ponownego użycia, łatwość analizy, łatwość modyfikowania, łatwość testowania
8. Przenośność (ang. Portability) – łatwość adaptacji, łatwość instalacji, łatwość zamiany

Na podstawie wyżej wymienionego schematu stworzona została poniższa tabela, przedstawiająca wymagania poza funkcjonalne projektu.

- Identyfikator
- Opis
- Kategoria
- Priorytet
 - H – wysoki.
 - M – średni.
 - L – niski.

Identyfikator	Opis	Kategoria	Priorytet
1	System powinien podawać wartości liczbowe z dokładnością do drugiego miejsca po przecinku	Funkcjonalna odpowiedniość	H
2	Czas odpowiedzi nie powinien przekraczać 3 sekund.	Wydajność	L
3	Możliwość przeglądania lotów równocześnie przez 1000 klientów	Wydajność	M
4	Możliwość obsługi minimum 300 realizowanych jednocześnie zamówień	Wydajność	M
5	System musi obsługiwać archiwum lotów z okresu minimum 2 lat wstecz	Wydajność	H
6	System powinien blokować działania zewnętrznych źródeł danych przez wielokrotne wykonywanie zapytań.	Wydajność	M
7	System powinien działać w najpopularniejszych przeglądarkach: Chrome 86.0+, Firefox 68.0+, Edge 46.0+, Opera 61.0+	Przenośność	H
8	Skalowalność ilości danych – rozwiązanie powinno zapewnić skalowalność ze względu na regularne wprowadzanie nowych danych przez klientów i pracowników.	Przenośność	H
9	System powinien dostosowywać interfejs w zależności od używanego urządzenia oraz jego rozdzielczości.	Użyteczność	H
10	System powinien wyświetlać ostrzeżenia przed wykonywaniem funkcji administratorów w celu zmniejszenia błędów	Użyteczność	H
11	System powinien wyraźnie zaznaczać pola obowiązkowe w formularzu.	Użyteczność	M
12	System nie powinien wyświetlać tekstu napisanego czcionką mniejszą niż 12 pikseli.	Użyteczność	L
13	Prace systemowe, administracyjne i konserwacyjne powinny się odbywać	Niezawodność	M

	pierwszego dnia każdego miesiąca w godzinach 1.00-3.00		
14	Hasło użytkownika powinno składać się z co najmniej 8 znaków, w tym co najmniej jednej dużej litery, jednej cyfry oraz jednego znaku specjalnego.	Bezpieczeństwo	H
15	Hasła użytkowników powinny być zaszyfrowane. Administrator nie ma do nich dostępu.	Bezpieczeństwo	H

Tabela 6. Lista wymagań pozafunkcjonalnych systemu. Opracowanie własne.

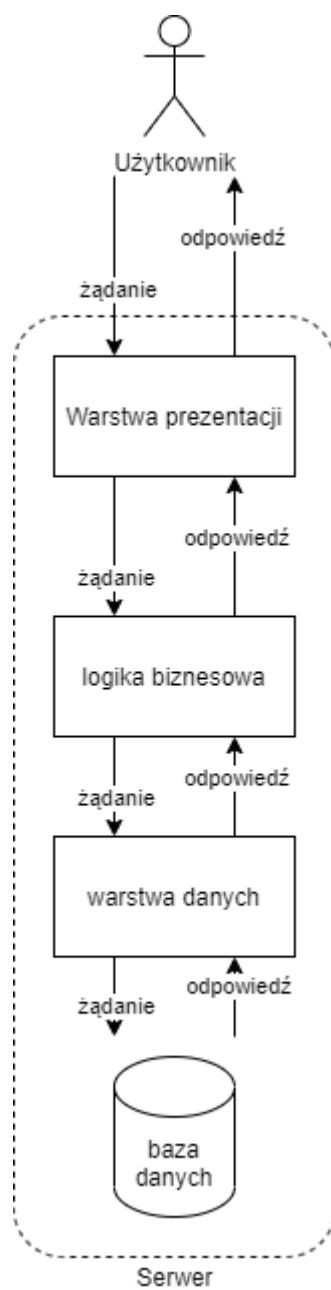
4. Projekt systemu

4.1. Architektura systemu

W zaprojektowanym systemie użyta zostanie architektura typu Klient-Serwer. A dokładniej – architektura trójwarstwowa, zwana czasami n-warstwową. Ten typ architektury zezwala na podział aplikacji na pewne warstwy – warstwę prezentacji, logiki biznesowej oraz warstwę danych. Wszystkie te warstwy współpracują ze sobą, aby udostępnić użytkownikowi usługi, do których może on wysyłać żądania.

Ten rodzaj architektury ma swoje zalety. Przede wszystkim dużym plusem jest przechowywanie danych w jednym miejscu na serwerze co umożliwia ich lepsze zabezpieczenie. Dodatkowo, dzięki podziałowi na różne warstwy i role, serwer może decydować, którzy użytkownicy mają dostęp do wybranej części funkcjonalności. [4, s. 100-101]. Warto wspomnieć również o fakcie, że w razie potrzeby rozbudowanie aplikacji i dodanie do niej nowych funkcji jest dość proste. Dzięki podziałowi na trzy warstwy, warstwa prezentacji nie modyfikuje bezpośrednio danych z bazy danych. Zamiast tego wprowadza jedynie zmianę na obiekcie w warstwie logiki biznesowej co pozwala na lepszą walidację poprawności danych przed zapisaniem ich w bazie danych.

Architektura ta nie jest jednak bez wad. Największym problemem stworzonym przez ten rodzaj architektury, jest zwiększenie poziomu złożoności aplikacji. Mimo, że dodawanie nowych funkcji jest proste, skutkuje ono dodaniem wielu nowych komponentów do aplikacji, co sprawia, że trzeba zarządzać coraz to większym kawałkiem kodu. [3, s. 134-136]



Rysunek 8. Diagram architektury systemu. Opracowanie własne.

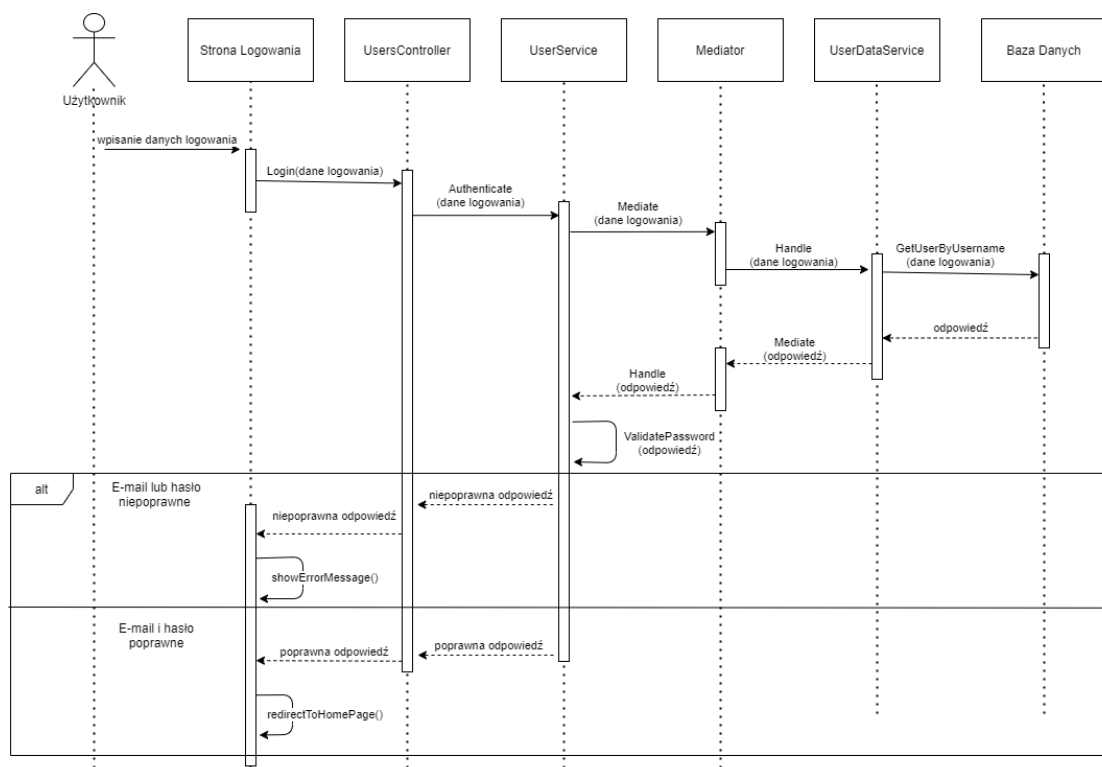
4.2. Logika systemu

W tym dziale projekt logiki systemu zostanie przedstawiony za pomocą kilku rodzajów diagramów – sekwencji, czynności i klas. Dodatkowo dołączone zostaną tabele dostarczające dodatkowe opisy.

Diagramy sekwencji to diagramy, które ukazują jak wykonywane są dane operacje. Ukazują interakcje między różnymi obiektami w kontekście ich współpracy. Pokazują też czas wykonywania danych operacji, a co za tym idzie, ich dokładną kolejność wykonywania. Pionowa oś przedstawia przepływ czasu, a pozioma ukazuje dokładną podróż wysłanego żądania.

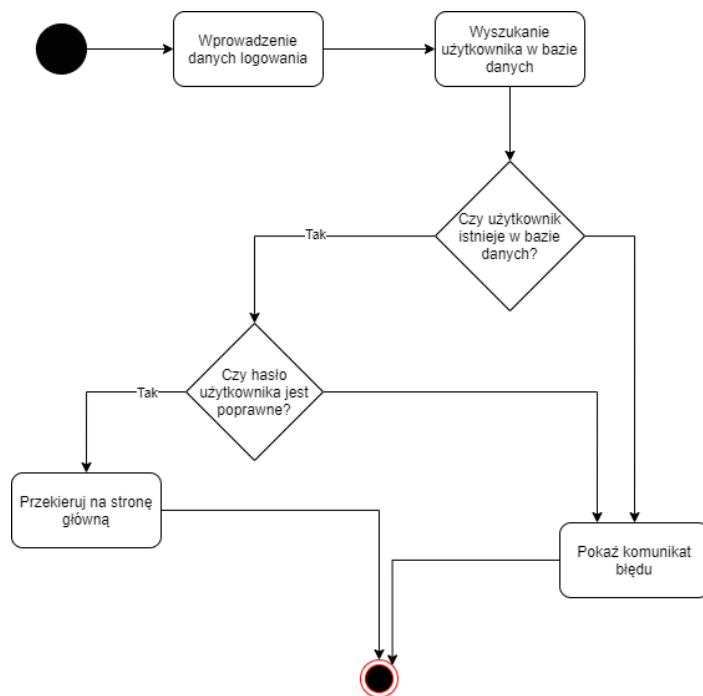
Ze względu na stopień złożoności aplikacji spowodowany przez jej rodzaj architektury, przedstawione zostaną jedynie najważniejsze funkcje – funkcja logowania, tworzenia załogi i edytowania lotu.

Niżej przedstawiony diagram sekwencji pokazuje przypadek logowania użytkownika do systemu.



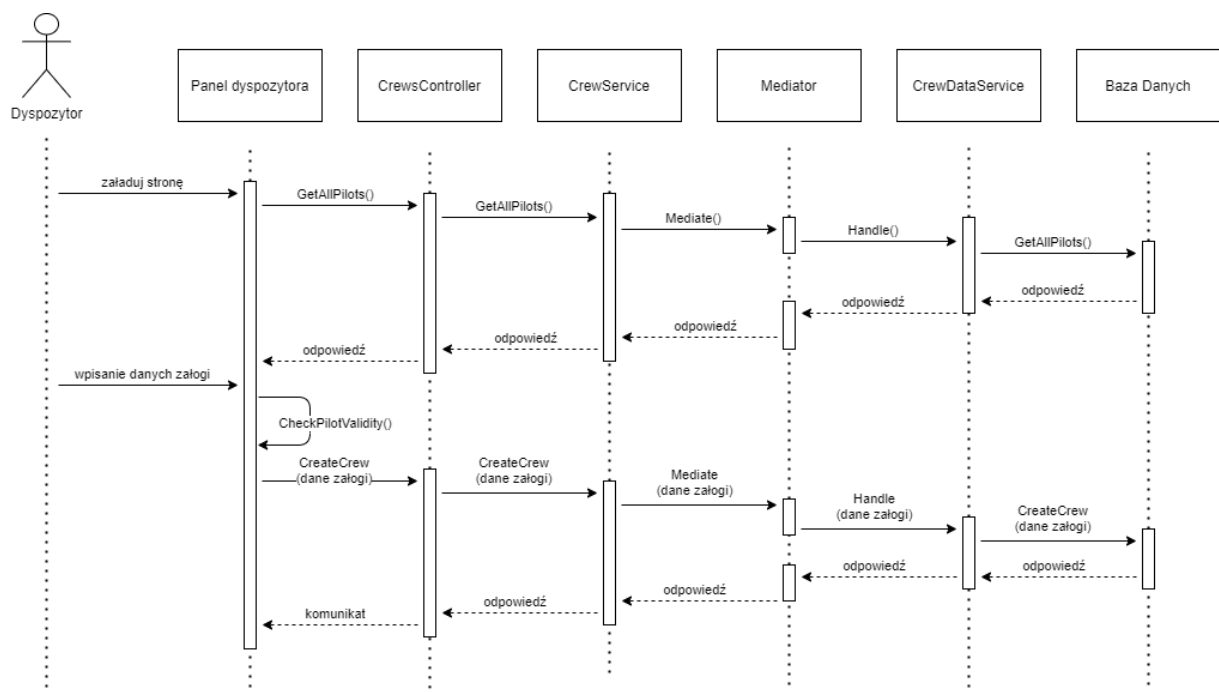
Rysunek 9. Diagram sekwencji - logowanie użytkownika. Opracowanie własne

Niżej przedstawiony diagram czynności pokazuje przypadek logowania użytkownika do systemu.



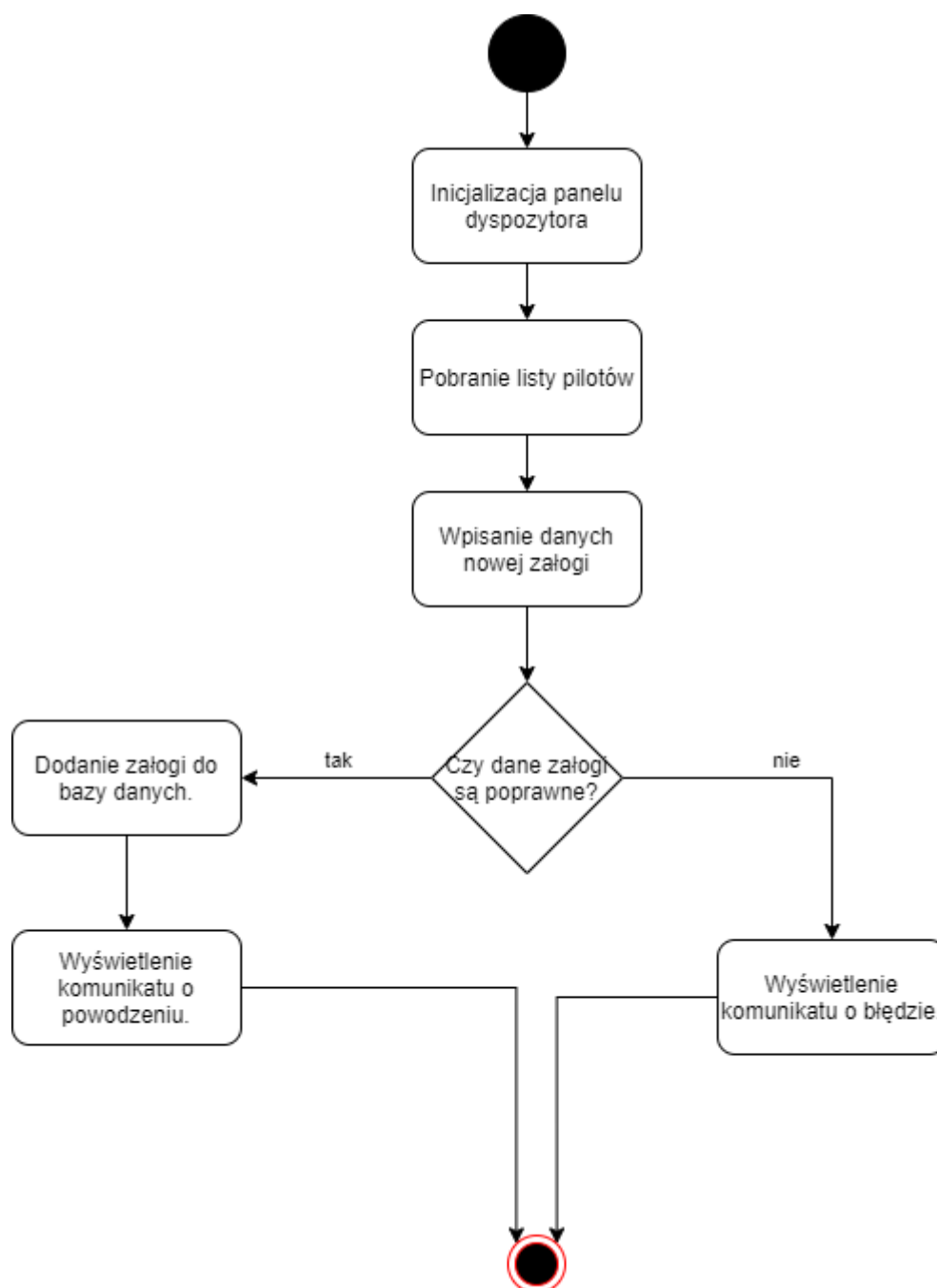
Rysunek 10. Diagram czynności - logowanie. Opracowanie własne.

Niżej przedstawiony diagram sekwencji pokazuje przypadek tworzenia załogi przez dyspozytora.



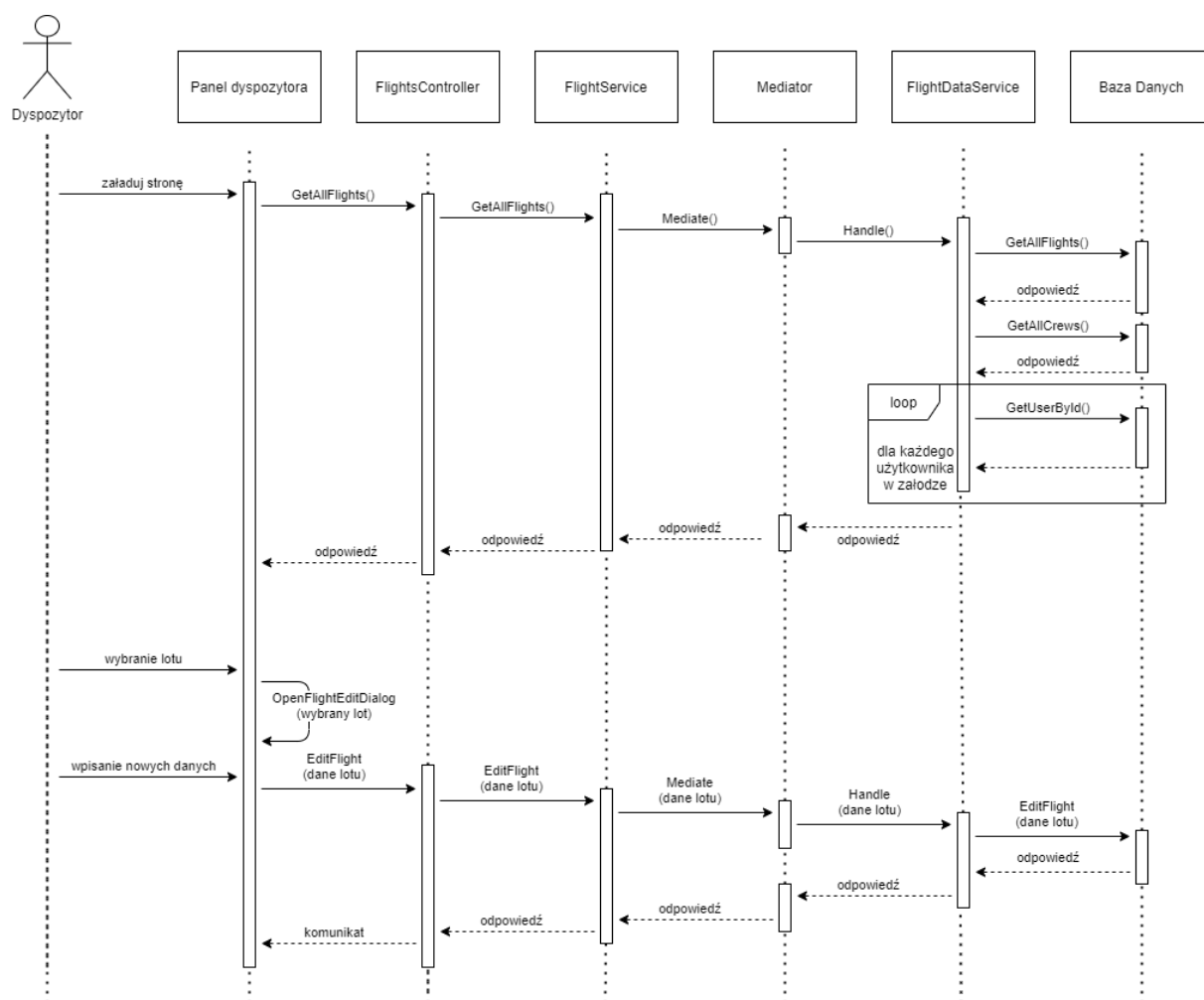
Rysunek 11. Diagram sekwencji - utworzenie załogi przez dyspozytora. Opracowanie własne.

Niżej przedstawiony diagram czynności pokazuje przypadek tworzenia załogi przez dyspozytora.



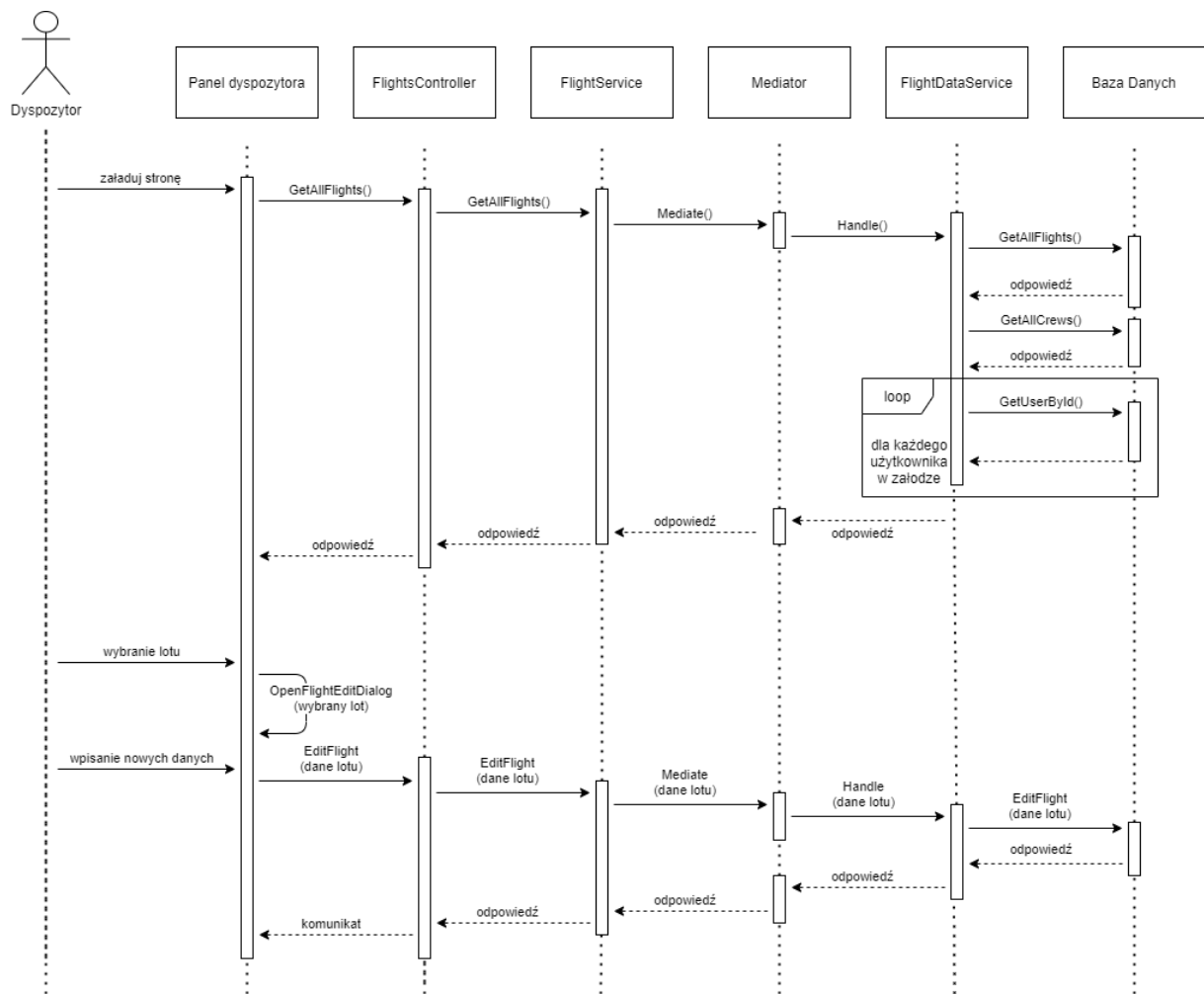
Rysunek 12. Diagram czynności - utworzenie załogi przez dyspozytora. Opracowanie własne.

Niżej przedstawiony diagram sekwencji pokazuje przypadek edytowania lotu przez dyspozytora.



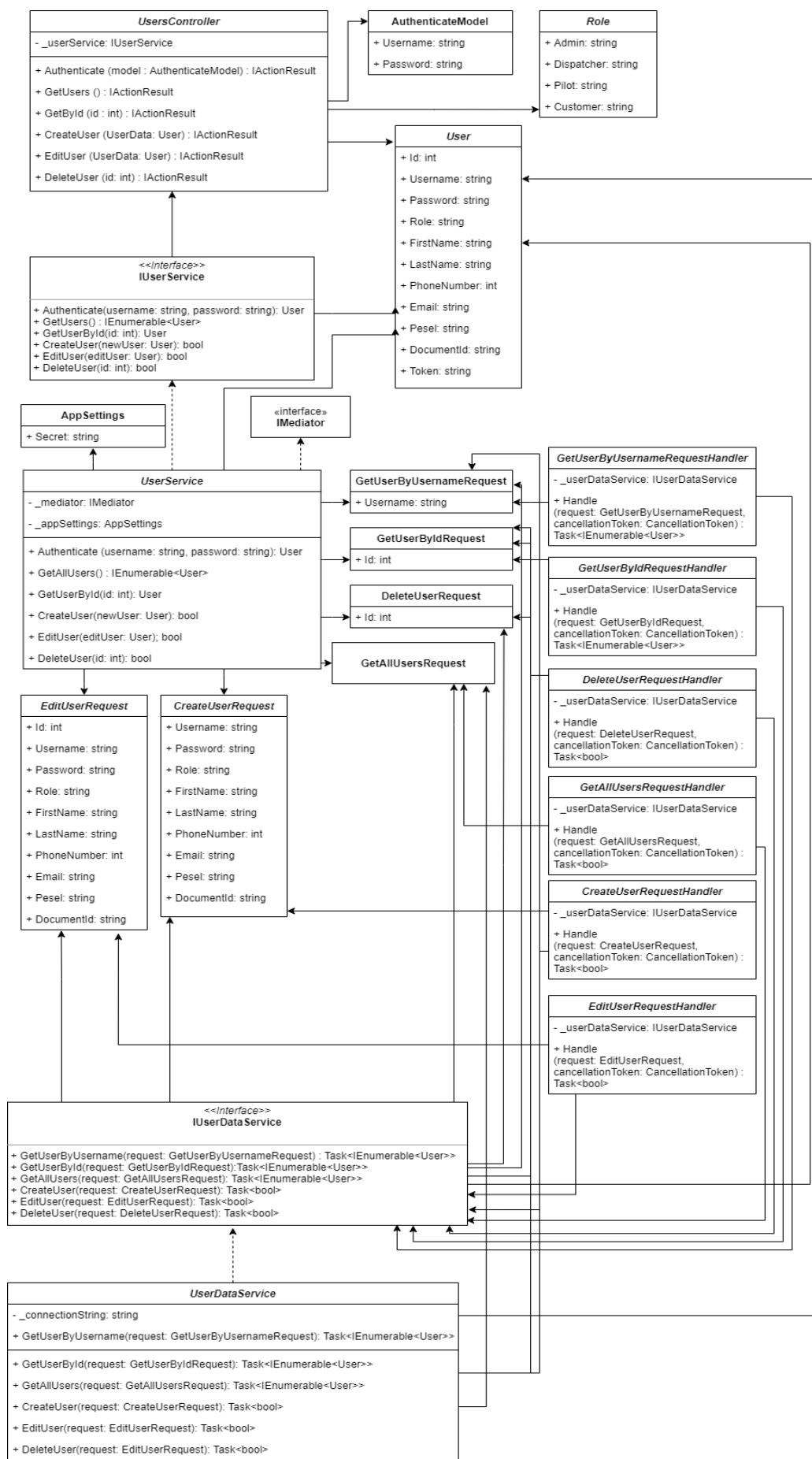
Rysunek 13. Diagram sekwencji - edytowanie lotu przez dyspozytora. Opracowanie własne.

Niżej przedstawiony diagram czynności pokazuje przypadek edytowania lotu przez dyspozytora.



Rysunek 14. Diagram czynności - edytowanie lotu przez dyspozytora. Opracowanie własne.

Poniżej zostanie przedstawiona część diagramu klas projektowanego systemu – części odpowiadającej za obsługiwane danych użytkowników. Wybrana architektura, a także plan użycia wzorca mediatora spowodowały, że liczba klas w systemie jest bardzo duża – planowane jest utworzenie około 83 klas. Wstawienie tak dużego diagramu na ograniczoną przestrzeń byłoby bardzo ciężkie i niepotrzebne – nie taki jest cel pracy. Cały diagram zostanie jednak dodany jako załącznik dla osób ciekawych.



Rysunek 15. Diagram klas części projektowanego systemu - część odpowiadająca za zarządzanie użytkownikami. Opracowanie własne.

Poniżej wstawiona tabela przedstawia wszystkie klasy projektu, a także ich metody z wyłączeniem ich argumentów. Wybrane klasy zostaną później opisane w bardziej szczegółowy sposób wraz z ich metodami.

Nazwa Klasy	Opis	Lista metod
UsersController	Klasa kontrolera interfejsu API służąca do wykonywania wszystkich czynności na użytkownikach .	Authenticate() GetAllUsers() GetUserById() CreateUser() EditUser() DeleteUser()
AuthenticateModel	Klasa reprezentująca model danych, służąca do tymczasowego przechowywania danych przekazanych przy logowaniu.	-
Role	Klasa reprezentująca model danych, dotyczy encji „Rola”.	-
User	Klasa reprezentująca model danych, dotyczy encji „Użytkownik”.	-
IUserService	Interfejs serwisu odpowiadającego za wszystkie czynności związane z zarządzaniem użytkownikami.	Authenticate() GetAllUsers() GetUserById() CreateUser() EditUser() DeleteUser()
UserService	Implementacji wyżej wymienionego interfejsu.	Authenticate() GetAllUsers() GetUserById() CreateUser() EditUser() DeleteUser()

IMediatorServices	Interfejs używany przez bibliotekę MediatR będący szyną komunikacyjną między klasami serwisów (np. UserService) a klasami obsługującymi dane (np. UserDataService).	-
AppSettings	Klasa reprezentująca model danych, zawiera wszystkie dane pobierane z pliku appsettings.json.	-
EditUserRequest	Klasa reprezentująca komunikat o edycji użytkownika, który będzie wysyłane do IMediator w celu jego zrealizowania.	-
EditUserRequestHandler	Klasa reprezentująca to co ma zostać zwrócone po otrzymaniu komunikatu o edytowaniu użytkownika.	Handle()
CreateUserRequest	Klasa reprezentująca komunikat o tworzeniu użytkownika, który będzie wysyłany do IMediator w celu jego zrealizowania.	-
CreateUserRequestHandler	Klasa reprezentująca to co ma zostać zwrócone po otrzymaniu komunikatu o	Handle()

	tworzeniu użytkownika.	
GetAllUsersRequest	Klasa reprezentująca komunikat o pobraniu wszystkich użytkowników, który będzie wysyłany do IMediator w celu jego zrealizowania.	-
GetAllUsersRequestHandler	Klasa reprezentująca to co ma zostać zwrócone po otrzymaniu komunikatu o pobraniu wszystkich użytkowników.	Handle()
DeleteUserRequest	Klasa reprezentująca komunikat o usunięciu użytkownika, który będzie wysyłany do IMediator w celu jego zrealizowania.	-
DeleteUserRequestHandler	Klasa reprezentująca to co ma zostać zwrócone po otrzymaniu komunikatu o usuwaniu użytkownika.	Handle()
GetUserByIdRequest	Klasa reprezentująca komunikat o pobraniu użytkownika o określonym Id, który będzie wysyłany do IMediator w celu jego zrealizowania.	-
GetUserByIdRequestHandler	Klasa reprezentująca to co ma zostać zwrócone	Handle()

	po otrzymaniu komunikatu o pobraniu użytkownika o określonym Id.	
GetUserByUsernameRequest	Klasa reprezentująca komunikat o pobraniu użytkownika o określonej nazwie użytkownika, który będzie wysyłany do IMediator w celu jego zrealizowania.	-
GetUserByUsernameRequestHandler	Klasa reprezentująca to co ma zostać zwrócone po otrzymaniu komunikatu o pobraniu użytkownika o określonej nazwie użytkownika.	Handle()
IUserDataService	Interfejs usługi odpowiedzialnej za komunikację z bazą danych a aplikacją w celu wykonywania wszystkich czynności związanych z zarządzaniem użytkownikami.	GetUserByUsername() GerUserById() GetAllUsers() CreateUser() EditUser() DeleteUser()
UserDataService	Implementacja wyżej wymienionego interfejsu.	GetUserByUsername() GerUserById() GetAllUsers() CreateUser() EditUser()

		DeleteUser()
CrewsController	Klasa kontrolera interfejsu API służąca do wykonywania wszystkich czynności na załogach.	GetAllPilots() GetAllCrews() CreateCrew() EditCrew() DeleteCrew()
Crew	Klasa reprezentująca model danych, dotyczy encji „Załoga”.	
CrewResponse	Klasa reprezentująca model danych, są to dane zwracane pierwotnie zwracane przez bazę danych .	
ICrewService	Interfejs serwisu odpowiadającego za wszystkie czynności związane z zarządzaniem załogami.	GetAllPilots() GetAllCrews() DeleteCrew() EditCrew() CreateCrew()
CrewService	Implementacja wyżej wymienionego interfejsu.	GetAllPilots() GetAllCrews() DeleteCrew() EditCrew() CreateCrew()
CreateCrewRequest	Klasa reprezentująca komunikat o tworzeniu załogi, który będzie wysyłany do IMediator w celu jego zrealizowania.	
CreateCrewRequestHandler	Klasa reprezentująca to co ma zostać zwrócone po otrzymaniu	Handle()

	komunikatu o tworzeniu załogi.	
EditCrewRequest	Klasa reprezentująca komunikat o edycji danych załogi, który będzie wysyłany do IMediator w celu jego zrealizowania.	
EditCrewRequestHandler	Klasa reprezentująca to co ma zostać zwrócone po otrzymaniu komunikatu o edytowaniu danych załogi.	Handle()
DeleteCrewRequest	Klasa reprezentująca komunikat o usunięciu załogi, który będzie wysyłany do IMediator w celu jego zrealizowania.	
DeleteCrewRequestHandler	Klasa reprezentująca to co ma zostać zwrócone po otrzymaniu komunikatu o usunięciu załogi.	Handle()
GetAllCrewsRequest	Klasa reprezentująca komunikat o pobraniu danych wszystkich załóg, który będzie wysyłany do IMediator w celu jego zrealizowania.	
GetAllCrewsRequestHandler	Klasa reprezentująca to co ma zostać zwrócone po otrzymaniu	Handle()

	komunikatu o pobraniu danych wszystkich załóg.	
GetAllPilotsRequest	Klasa reprezentująca komunikat o pobraniu danych wszystkich pilotów, który będzie wysyłany do IMediator w celu jego zrealizowania.	
GetAllPilotsRequestHandler	Klasa reprezentująca to co ma zostać zwrócone po otrzymaniu komunikatu o pobraniu danych wszystkich pilotów.	Handle()
ICrewDataService	Interfejs serwisu odpowiadającego za komunikację między aplikacją a bazą danych w celu wykonywania wszystkich czynności związanych z zarządzaniem załogami.	GetAllPilots() GetAllCrews() CreateCrew() EditCrew() DeleteCrew()
CrewDataService	Implementacja wyżej wymienionego interfejsu.	GetAllPilots() GetAllCrews() CreateCrew() EditCrew() DeleteCrew()
FlightsController	Klasa kontrolera interfejsu API służąca do wykonywania wszystkich czynności na lotach.	CreateFlight() GetAllFlights() EditFlight() DeleteFlight() GetPilotFlights()

		EditFlightStatus() SearchFlights() GetFlight()
Flight	Klasa reprezentująca model danych, dotyczy encji „Lot”.	
FlightResponse	Klasa reprezentująca model danych, są to dane zwracane przez bazę danych.	
SearchParameters	Klasa reprezentująca model danych, są to parametry wyszukiwania wpisywane przez użytkowników którzy szukają lotu w którym chcą zarezerwować miejsce.	
IFlightService	Interfejs serwisu odpowiadającego za wszystkie czynności związane z zarządzaniem lotami.	GetAllFlights() DeleteFlight() EditFlight() CreateFlight() GetPilotFlights)_ EditFlightStatus() SearchFlights(0 GetFlight()
FlightService	Implementacja wyżej wymienionego serwisu.	GetAllFlights() DeleteFlight() EditFlight() CreateFlight() GetPilotFlights)_ EditFlightStatus() SearchFlights(0 GetFlight()

CreateFlightRequest	Klasa reprezentująca komunikat o utworzeniu lotu, który będzie wysyłany do IMediator w celu jego zrealizowania.	
CreateFlightRequestHandler	Klasa reprezentująca to co ma zostać zwrócone po otrzymaniu komunikatu o utworzeniu lotu.	Handle()
DeleteFlightRequest	Klasa reprezentująca komunikat o usunięciu lotu, który będzie wysyłany do IMediator w celu jego zrealizowania.	
DeleteFlightRequestHandler	Klasa reprezentująca to co ma zostać zwrócone po otrzymaniu komunikatu o usunięciu lotu.	Handle()
EditFlightRequest	Klasa reprezentująca komunikat o edytowaniu lotu, który będzie wysyłany do IMediator w celu jego zrealizowania.	
EditFlightRequestHandler	Klasa reprezentująca to co ma zostać zwrócone po otrzymaniu komunikatu o edytowaniu lotu.	Handle()
EditFlightStatusRequest	Klasa reprezentująca komunikat o	

	edytowaniu statusu lotu, który będzie wysyłany do IMediator w celu jego zrealizowania.	
EditFlightStatusRequestHandler	Klasa reprezentująca to co ma zostać zwrócone po otrzymaniu komunikatu o edytowaniu statusu lotu.	Handle()
GetAllFlightsRequest	Klasa reprezentująca komunikat o pobraniu wszystkich lotów, który będzie wysyłany do IMediator w celu jego zrealizowania.	
GetAllFlightsRequestHandler	Klasa reprezentująca to co ma zostać zwrócone po otrzymaniu komunikatu o pobraniu wszystkich lotów.	Handle()
GetFlightByIdRequest	Klasa reprezentująca komunikat o pobraniu lotu o określonym Id, który będzie wysyłany do IMediator w celu jego zrealizowania.	
GetFlightByIdRequestHandler	Klasa reprezentująca to co ma zostać zwrócone po otrzymaniu komunikatu o pobraniu lotu o określonym Id.	Handle()
GetPilotFlightsRequest	Klasa reprezentująca komunikat o pobraniu	

	lotów realizowanych przez wybranego pilota, który będzie wysyłany do IMediator w celu jego zrealizowania.	
GetPilotFlightsRequestHandler	Klasa reprezentująca to co ma zostać zwrócone po otrzymaniu komunikatu o pobraniu lotów obsługiwanych przez wybranego pilota.	Handle()
SearchFlightsRequest	Klasa reprezentująca komunikat o wyszukiwaniu lotów przez klienta, który będzie wysyłany do IMediator w celu jego zrealizowania	
SearchFlightsRequestHandler	Klasa reprezentująca to co ma zostać zwrócone po otrzymaniu komunikatu o wyszukiwaniu lotów przez klienta.	Handle()
IFlightDataService	Interfejs serwisu odpowiadającego za komunikację między aplikacją a bazą danych w celu wykonywania wszystkich czynności związanych z zarządzaniem lotami.	GetAllFlights() CreateFlight() EditFlight() DeleteFlight() GetPilotFlights() EditFlightStatus() SearchFlights() GetFlightById()
FlightDataService	Implementacja wyżej wymienionego interfejsu	GetAllFlights() CreateFlight() EditFlight()

		DeleteFlight() GetPilotFlights() EditFlightStatus() SearchFlights() GetFlightById()
ReservationsController	Klasa kontrolera interfejsu API służąca do wykonywania wszystkich czynności na rezerwacjach	CreateReservation() GetUserReservations() GetFlightReservations() GetTakenSeats()
IReservationService	Interfejs serwisu odpowiadającego za wszystkie czynności związane z zarządzaniem rezerwacjami.	CreateReservation() GetUserReservations() GetFlightReservations() GetTakenSeats() EditReservation()
ReservationService	Implementacja wyżej wymienionego serwisu.	CreateReservation() GetUserReservations() GetFlightReservations() GetTakenSeats() EditReservation()
Reservation	Klasa reprezentująca model danych, dotyczy encji „Rezerwacja”.	
ReservationResponse	Klasa reprezentująca model danych, są to nieprzetworzone dane zwracane przez bazę danych.	
ReservationUserResponse	Klasa reprezentująca model danych, są to nieprzetworzone dane zwracane przez bazę danych.	

CreateReservationRequest	Klasa reprezentująca komunikat o tworzeniu rezerwacji, który będzie wysyłany do IMediator w celu jego zrealizowania.	
CreateReservationRequestHandler	Klasa reprezentująca to co ma zostać zwrócone po otrzymaniu komunikatu o tworzeniu rezerwacji.	Handle()
GetFlightReservationsRequest	Klasa reprezentująca komunikat pobrania danych rezerwacji danego lotu, który będzie wysyłany do IMediator w celu jego zrealizowania.	
GetFlightReservationsRequestHandler	Klasa reprezentująca to co ma zostać zwrócone po otrzymaniu komunikatu o pobieraniu danych rezerwacji danego lotu.	Handle()
GetTakenSeatsRequest	Klasa reprezentująca komunikat o pobraniu danych o zajętych miejscach na danym locie, który będzie wysyłany do IMediator w celu jego zrealizowania.	
GetTakenSeatsRequestHandler	Klasa reprezentująca to co ma zostać zwrócone po otrzymaniu	Handle()

	komunikatu o pobraniu danych o zajętych miejscach na danym locie.	
GetUserReservationsRequest	Klasa reprezentująca komunikat o pobraniu rezerwacji danego użytkownika, który będzie wysyłany do IMediator w celu jego zrealizowania.	
GetUserReservationsRequestHandler	Klasa reprezentująca to co ma zostać zwrócone po otrzymaniu komunikatu o pobraniu danych o rezerwacjach danego użytkownika.	Handle()
EditReservationRequest	Klasa reprezentująca komunikat o edytowaniu rezerwacji, który będzie wysyłany do IMediator w celu jego zrealizowania.	
EditReservationRequestHandler	Klasa reprezentująca to co ma zostać zwrócone po otrzymaniu komunikatu o edytowaniu rezerwacji.	Handle()
IReservationDataService	Interfejs serwisu odpowiadającego za komunikację między aplikacją a bazą danych w celu wykonywania wszystkich czynności związanych z	CreateReservation() GetFlightReservations() GetUserReservations() GetTakenSeats() EditReservation()

	zarządzaniem rezerwacjami.	
ReservationDataService	Implementacja wyżej wymienionego interfejsu.	CreateReservation() GetFlightReservations() GetUserReservations() GetTakenSeats() EditReservation()
Startup	Klasa frameworku ASP.NET Core służąca do konfiguracji serwisów.	ConfigureServices() Configure()
Program	Klasa frameworku ASP.NET Core, która inicjalizuje hosta aplikacji.	Main() CreateWebHostBuilder()

Tabela 7. Opis klas projektowanego systemu. Opracowanie własne.

Klasy UserService oraz ReservationDataService zostały wybrane w celu szczegółowego przedstawienia z powodu bycia kluczowymi częściami systemu. Poniżej opiszę przeznaczenie ich metod, wymienię również typy zwracane oraz opiszę ich parametry.

Nazwa	Typ zwracany	Argumenty	Przeznaczenie
Authenticate	User	string username, string password	Metoda umożliwiająca autentykację użytkownika na podstawie wpisanego przez niego danych. Dane zostają pobrane za pomocą mediatora a następnie odszyfrowane za pomocą biblioteki BCrypt.
GetUsers	IEnumerable<User>	-	Metoda umożliwiająca pobranie listy wszystkich użytkowników za pomocą mediatora.

GetUserById	User	int id	Metoda umożliwiająca pobranie danych użytkownika o danym id za pomocą mediatora.
CreateUser	bool	User newUser	Metoda umożliwiająca utworzenie użytkownika o określonych danych za pomocą mediatora.
EditUser	bool	User editUser	Metoda umożliwiająca edycję użytkownika o określonych danych za pomocą mediatora.
DeleteUser	bool	int id	Metoda umożliwiająca usunięcie użytkownika o podanym Id za pomocą mediatora.

Tabela 8. Opis metod w klasie UserService. Opracowanie własne.

Nazwa	Typ Zwracany	Argumenty	Przeznaczenie
CreateReservation	Task<bool>	CreateReservationsRequest request	Metoda nawiązująca połączenie z bazą danych w celu utworzenia rezerwacji.
GetFlightsReservations	Task<IEnumerable<Reservations>>	GetFlightsReservationsRequest request	Metoda nawiązująca połączenie z bazą danych w celu pobrania danych rezerwacji danego lotu.
GetUserReservations	Task<IEnumerable<Reservations>>	GetUserReservationsRequest request	Metoda nawiązująca połączenie z bazą danych w celu pobrania danych o rezerwacjach

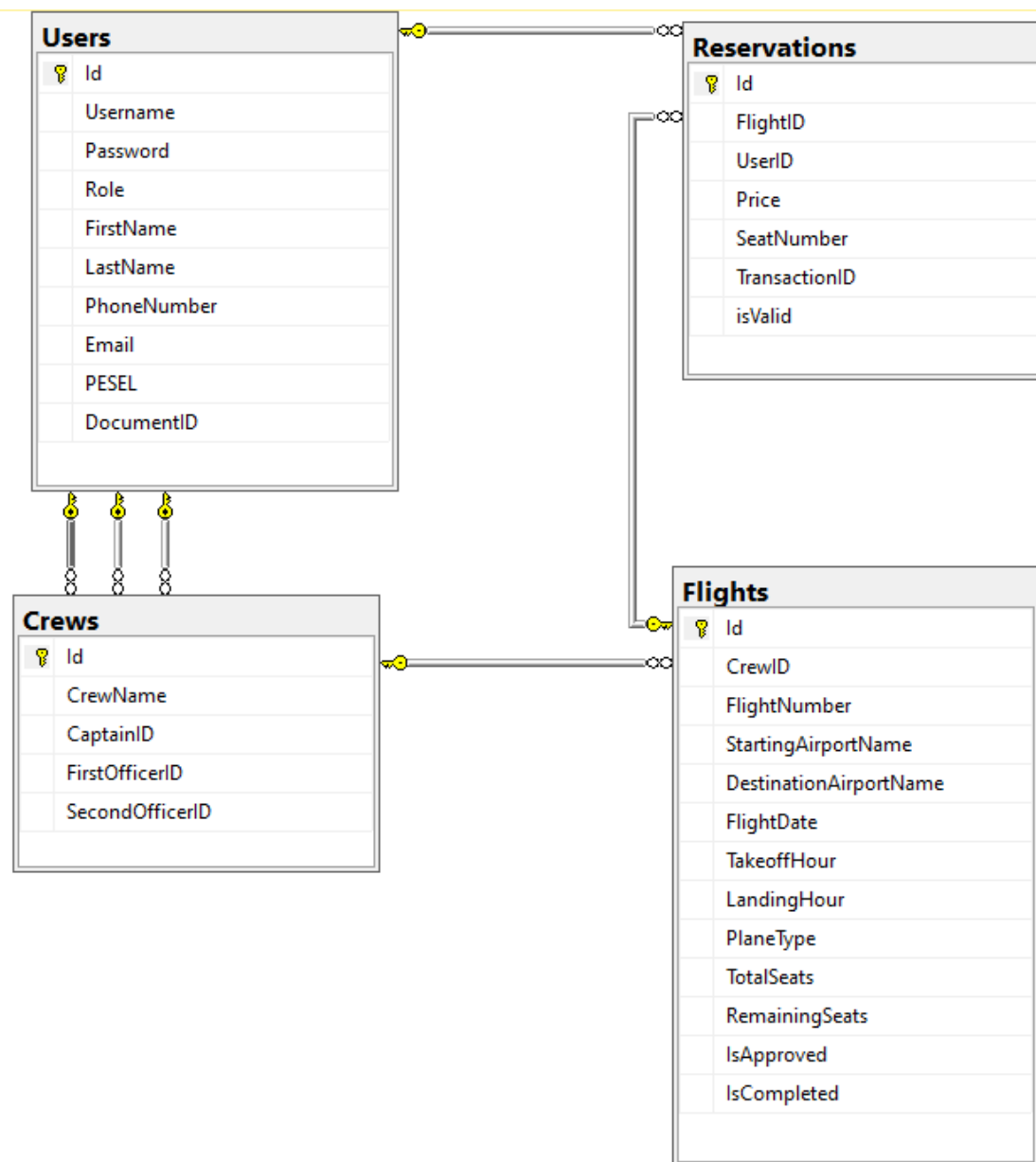
			danego użytkownika.
GetTakenSeats	Task<IEnumerable <string>>	GetTakenSeatsRequest request	Metoda nawiązująca połączenie z bazą danych w celu pobrania danych o miejscach zajętych w danym locie.
EditReservatio n	Task<bool>	EditReservationRequest request	Metoda nawiązująca połączenie z bazą danych w celu edycji danych danej rezerwacji.

Tabela 9. Opis klasy *ReservationDataService*. Opracowanie własne.

4.3. Baza danych

W tym rozdziale zostanie przedstawiony projekt bazy danych, który zostanie użyty w fazie implementacji. Do zarządzania bazą zostanie wykorzystany MS SQL oraz Microsoft SQL Server Management Studio 18.

4.3.1. Diagram relacji bazy danych



Rysunek 16. Projekt diagramu relacji bazy danych. Opracowanie własne.

4.3.2. Opis tabel bazy danych

Podczas tworzenia projektu, zdecydowałam się na wprowadzenie zmian odnośnie encji systemu. Dyspozytor, administrator, pilot oraz użytkownik zostali przedstawieni jako osobne encje. Ostatecznie jednak zdecydowałam się na ich połączenie – tabela „Users” będzie przedstawiać wszystkich użytkowników systemu. Ich rodzaj będzie rozróżniany na podstawie pola „Role”. Dzięki temu projekt bazy danych zostanie mocno uproszczony.

W poniższych tabelach zostanie przedstawiony wykaz tabel z projektowanej bazy danych.

Nazwa tabeli	Opis
Users	Tabela, która zawiera dane na temat użytkowników systemu
Crews	Tabela, która zawiera dane na temat załóg, składających się z użytkowników systemu.
Flights	Tabela, która zawiera dane na temat lotów.
Reservations	Tabela, która zawiera dane na temat rezerwacji dokonanych na danych lotach.

Tabela 10. Wykaz tabel projektowanej bazy danych. Opracowanie własne.

Następnie przedstawione zostaną tabele zawierające szczegółowe dane dotyczące pól wyżej wymienionych tabel. Zawierają one informacje takie jak: klucz, nazwa, typ, opcje oraz opis. Pola są domyślnie ustawione jako NOT NULL, chyba, że zostało to sprecyzowane inaczej w kolumnie opcje.

Klucz	Nazwa Pola	Typ	Opcje	Opis
PK	Id	int	IDENTITY(1,1)	Identyfikator użytkownika.
	Username	varchar(50)		Nazwa użytkownika.
	Password	varchar(250)		Zaszyfrowane hasło użytkownika.
	Role	varchar(15)		Rola użytkownika.
	FirstName	varchar(50)		Imię użytkownika.
	LastName	varchar(50)		Nazwisko użytkownika.
	PhoneNumber	int	NULL	Numer telefonu użytkownika.
	Email	varchar(50)	NULL	Adres E-Mail użytkownika.

	PESEL	char(11)	NULL	Numer Pesel użytkownika.
	DocumentID	char(9)	NULL	Numer dokumentu osobistego użytkownika.

Tabela 11. Opis tabeli bazy danych Users. Opracowanie własne.

Klucz	Nazwa Pola	Typ	Opcje	Opis
PK	Id	int	IDENTITY(1,1)	Identyfikator załogi.
	CrewName	nvarchar(50)		Nazwa załogi.
FK	CaptainID	int		Identyfikator kapitana.
FK	FirstOfficerID	int		Identyfikator pierwszego oficera,
FK	SecondOfficerID	int		Identyfikator drugiego oficera.

Tabela 12. Opis tabeli bazy danych Crews. Opracowanie własne.

Klucz	Nazwa Pola	Typ	Opcje	Opis
PK	Id	int	IDENTITY(1,1)	Identyfikator lotu
FK	CrewID	int		Identyfikator załogi, która obsługuje dany lot.
	FlightNumber	nvarchar(6)		Numer lotu.
	StartingAirportName	nvarchar(150)		Nazwa lotniska początkowego.
	DestinationAirportName	nvarchar(150)		Nazwa lotniska docelowego.
	FlightDate	date		Data wylotu.
	TakeoffHour	varchar(8)		Godzina wylotu.
	LandingHour	varchar(8)		Godzina przylotu.
	PlaneType	nvarchar(25)		Rodzaj samolotu (model np. A320)
	TotalSeats	int		Maksymalna ilość miejsc
	RemainingSeats	int		Pozostała ilość wolnych miejsc.
	IsApproved	bit		Określa, czy lot został zaakceptowany do obsłużenia.

	IsCompleted	bit		Określa, czy lot został ukończony.
--	-------------	-----	--	------------------------------------

Tabela 13. Opis tabeli bazy danych Flights. Opracowanie własne.

Klucz	Nazwa Pola	Typ	Opcje	Opis
PK	Id	int	IDENTITY(1,1)	Identyfikator rezerwacji.
FK	FlightID	int		Identyfikator lotu na którym została dokonana rezerwacja.
FK	UserID	int		Identyfikator użytkownika, który dokonał rezerwacji.
	Price	decimal(7,2)		Koszt dokonanej rezerwacji.
	SeatNumber	nvarchar(4)		Numer zarezerwowanego miejsca.
	TransactionID	nvarchar(100)		Identyfikator transakcji, dostarczany przez usługę, która obsługuje płatności w aplikacji.
	IsValid	bit		Określa, czy rezerwacja jest ważna.

Tabela 14. Opis tabeli bazy danych Reservations. Opracowanie własne.

4.4. Projekt interfejsów użytkownika

W celu zaprojektowania interfejsu użytkownika skorzystałam ze strony <https://app.moqups.com>. Pozwala ona na utworzenie makiet interfejsów za darmo i jest też dość prosta w obsłudze.

Strona będzie mieć dość jednolity układ w celu zapewnienia uporządkowanej struktury i ułatwienia implementacji. Ze względu na naturę implementacji interfejsów i kapryśnej natury języka CSS finalny wygląd strony może się zmienić w fazie implementacji.

Layout został podzielony na dwa główne elementy:

1. Pasek nawigacyjny, będący elementem, który wyświetla się na każdej stronie. W zależności od roli użytkownika udostępnia on przyciski do przekierowań na strony do których użytkownik danej roli ma dostęp. Dodatkowo będzie zawierać logo linii lotniczej używającej oprogramowania a także przycisk umożliwiający wylogowanie się.
2. Część główna, która z pomocą frameworku Angular będzie dynamicznie ładować wybraną przez użytkownika treść.



Rysunek 17. Układ interfejsu graficznego stron. Opracowanie własne.

Pasek nawigacji będzie wyświetlać inne ikony umożliwiające przekierowanie w zależności od roli użytkownika. W systemie możemy wyróżnić pięć głównych ról:

1. Użytkownik niezalogowany
2. Użytkownik zalogowany (klient)
3. Dyspozytor
4. Pilot
5. Administrator

Biorąc pod uwagę następujący podział, możliwe będzie wyświetlenie tych oto ikon:

1. Logo firmy – dla użytkownika niezalogowanego.
2. Logo firmy, ikona domku oznaczająca przekierowanie na stronę główną, ikona karty kredytowej pozwalająca na przejście na stronę zakupu biletu, ikona wyłączenia zasilania pozwalająca na wylogowanie – dla wszystkich użytkowników zalogowanych.
3. Wszystkie ikony użytkownika zalogowanego, a także ikona planszy do której przypisuje się zadania oznaczająca przekierowanie do strony panelu dyspozytora – dla użytkowników z rolą dyspozytora.
4. Wszystkie ikony użytkownika zalogowanego, a także ikona odlatującego samolotu pozwalająca na przekierowanie do strony panelu pilota – dla użytkowników z rolą pilota.
5. Wszystkie ikony użytkownika zalogowanego, a także ikona tarczy pozwalająca na przekierowanie na stronę panelu administratora – dla użytkowników z rolą administratora.

Ponieważ część główna będzie wyświetlać wybraną przez użytkownika stronę, w następnej kolejności przedstawię wybrane warianty interfejsu użytkownika – dla użytkownika niezalogowanego, zalogowanego i dyspozytora.

4.4.1. Projekt interfejsów użytkownika niezalogowanego.

Jedyną stroną do której będzie mieć dostęp użytkownik niezalogowany jest strona umożliwiająca logowanie. Poniżej przedstawiona została jej makiet.

The wireframe shows a login interface. At the top is a header bar with an airplane icon and the text "AIRLINE NAME". Below this is a red rectangular box containing the text "Ostrzeżenie o błędnych danych". Underneath the warning box is a white box with a grey header labeled "LOGIN". Inside this box are two labels, "Username:" and "Password:", each followed by a text input field. At the bottom of the box is a button labeled "Login".

Rysunek 18. Makiet strony logowania. Opracowanie własne.

Jest to pierwsze okno, które będzie się wyświetlać jako domyślna strona aplikacji.

Na górze widoczny jest pasek nawigacji, który zawiera logo a także nazwę linii lotniczej. Pionowa kreska obok napisu „AIRLINE NAME: oddziela nazwę firmy od ikon które będą pozwalać na poruszanie się po systemie.

Na środku strony znajduje się okienko pozwalające na wprowadzenie danych logowanie. Tekst wpisywany w pole wprowadzania hasła będzie zakryte kropkami.

Po naciśnięciu przycisku „Login” nastąpi weryfikacja danych użytkownika przez system a następnie w zależności od wyniku weryfikacji – nastąpi przekierowanie na stronę główną, lub domyślnie niewidoczne pole ostrzeżenia o błędnych danych stanie się widoczne i poinformuje użytkownika o błędnie wprowadzonych danych. Pole ostrzeżenia nie wyjawia które dane były niepoprawne w celu obfuskacji błędów.

4.4.2. Projekt interfejsów użytkownika zalogowanego.

Pierwszą stroną widoczną dla wszystkich użytkowników zalogowanych jest strona główna, inaczej zwana jako home page.

The wireframe shows a web application interface for a logged-in user. At the top is a navigation bar with a grey background. On the left, there is an airplane icon followed by the text 'AIRLINE NAME'. In the center, there are icons for a home page and a wallet. On the right, there is a power button icon. Below the navigation bar is a large white rectangular area representing the main content. Inside this area, there is a smaller white box with a grey header labeled 'HOME'. Below the header, the text 'Welcome, {{user}}!' is displayed. Underneath, there are four input fields arranged in two columns: 'Password:', 'Phone number:', 'Email:', and 'PESEL:'. Below the 'Password:' field is a button labeled 'Change Personal Info'. To the right of the 'PESEL:' field is another input field labeled 'Document ID:'.

Rysunek 19. Makieta strony głównej. Opracowanie własne.

Widoczny na górze pasek nawigacji powiększył się o dodatkowe ikony, posiadające funkcjonalności opisane w dziale 4.4.

Główną częścią tej strony jest okienko wyświetlane na jej środku, pokazujące nazwę zalogowanego użytkownika i pozwalające na zmianę danych osobowych. Pole wprowadzania hasła będzie zakryte kropkami.

Naciśnięcie przycisku „Change Personal Info” spowoduje wysłanie nowych danych do serwera. W zależności czy zmiana danych powiodła się czy nie, zostanie wyświetlony komunikat informujący użytkownika o tym. Wszystkie komunikaty zostaną opisane w późniejszym dziale.

Po naciśnięciu w ikonę karty kredytowej, użytkownik zostaje przeniesiony na stronę panelu użytkownika. Pozwala on zalogowanemu użytkownikowi na dokonanie rezerwacji a także wyświetlenie dokonanych już rezerwacji.

The wireframe shows a user interface for an airline's customer panel. At the top, there's a header bar with 'AIRLINE NAME' on the left, a home icon in the center, and a power icon on the right. Below the header, the main content area is titled 'CUSTOMER'. Inside this area, there's a search section with three input fields labeled 'Origin:', 'Destination:', and 'Date:', followed by a 'Search' button. Below the search section is a table with the following columns: 'Flight Number', 'Origin Airport Name', 'Destination Airport Name', 'Flight Date', 'Landing Date', and 'Actions'. The 'Actions' column contains four 'BUY' buttons. Below this table is a section titled 'View your reservations' which contains another table with the following columns: 'Reservation Id', 'Origin Airport Name', 'Destination Airport Name', 'Flight Date', 'Landing Date', and 'Status'.

Rysunek 20. Makieta panelu użytkownika. Opracowanie własne.

Przedstawiony na środku panel użytkownika składa się z kilku części:

Po pierwsze, wyszukiwarka lotów znajdująca się pod napisem 'CUSTOMER'. Pozwala ona na wprowadzenie przez użytkownika dowolnych danych po czym po kliknięciu na guzik „Search” wyszukanie lotów, które spełniają podane kryteria.

Po drugie, poniżej wyszukiwarki znajduje się lista ukazująca wyniki wyszukiwania. Domyślnie jest ona niewidoczna i pojawia się tylko wtedy gdy jakiegokolwiek dane zostaną znalezione. Lista zawiera w sobie dane takie jak: Numer lotu, lotnisko startowe, lotnisko docelowe, data wylotu, data lądowania a także kolumna z guzikami pozwalającymi na dokonanie rezerwacji.

Ostateczną częścią jest tabela „View your reservations”. Domyślnie użytkownik widzi jedynie pasek z nazwą tabeli. Dopiero po jego naciśnięciu otwiera się on jak akordeon i pokazuje tabelę. Tabela zawiera dane takie jak: Numer Id rezerwacji, lotnisko startowe, lotnisko docelowe, data wylotu, data lądowania a także status lotu – pokazuje on czy lot odbył się już czy nie.

Naciśnięcie w pole „BUY” w tabeli dokonywania rezerwacji przeniesie użytkownika do strony pozwalającej na dokonanie zakupu.

Row A	Row B	Row C	Row D	Row E	Row F
1A	1B	1C	1D	1E	1F
2A	2B	2C	2D	2E	2F
3A	3B	3C	3D	3E	3F
...

Rysunek 21. Makieta strony zakupu biletu. Opracowanie własne.

Przedstawiona na obrazku strona ukazuje panel umożliwiający dokonanie rezerwacji poprzez zakup biletu. Tekst na górze panelu informuje użytkownika o sprawdzeniu danych osobistych przed dokonaniem zakupu – użytkownik nie może dokonać rezerwacji jeżeli któreś z jego danych są puste, system do tego nie dopuści, jednak niemożliwe jest wykrycie literówek w numerach dokumentów lub adresu email. W celu szybkiego podglądu, dane użytkownika są wyświetlone pod tekstem ostrzegającym.

Na dole strony znajduje się tabela odzwierciedlająca miejsca w samolocie. Przykładowo, guzik „1A” oznacza miejsce w pierwszym rzędzie na siedzeniu oznakowanym literą A. Siedzenia oznakowane literą „A” i „F” to siedzenia znajdujące się przy oknie. Miejsca zaznaczone na czerwone odzwierciedlają zajęte już miejsca – kliknięcie na nie jest niemożliwe. Kliknięcie na jedno z dostępnych miejsc spowoduje otworenie okna dialogowego w celu finalizacji płatności.

CHECKOUT	
Flight Number:	Price:
Origin Airport Name:	Seat:
Destination Airport Name:	
Flight Date:	
Landing Date:	
<div>PAYPAL</div>	

Rysunek 22. Makieta okno dialogowego finalizacji płatności. Opracowanie własne.

Okno dialogowe pokazuje użytkownikowi dane wybranego lotu, cenę miejsca a także numer miejsca zanim podejmie się on dokonania płatności. Naciśnięcie przycisku „PAYPAL” spowoduje otworzenie nowego okna przeglądarki w celu obsłużenia płatności przez API serwisu Paypal Checkout. Zwrot informacji o pomyślnie przeprowadzonej płatności skutkuje zamknięciem powyższego okna i wyświetleniem komunikatu o powodzeniu, opisanym w późniejszym dziale.

4.4.3. Projekt interfejsów dyspozytora.

AIRLINE NAME

DISPATCHER

Create Crew

Crew Name

Captain

First Officer

Second Officer

Create

View All Crews

Filter

Filter

Id	Crew Name	Captain	First Officer	Second Officer	Edit	Delete
					EDIT	DELETE
					EDIT	DELETE
					EDIT	DELETE
					EDIT	DELETE

Create Flight

Flight Number

Crew

Origin Airport Name

Destination Airport Name

Takeoff Hour

Landing Hour

Plane Type

Total Seats

Flight date

CREATE

View All Flights

Filter

Filter

Id	Crew Name	Flight Number	Starting Airport	Destination Airport	Status	Edit	Delete	View Reservations
						EDIT	DELETE	Reservations
						EDIT	DELETE	Reservations
						EDIT	DELETE	Reservations
						EDIT	DELETE	Reservations

Rysunek 23. Makieta panelu dyspozytora. Opracowanie własne.

Użytkownik z rolą dyspozytora ma dostęp do strony panelu dyspozytora do którego można się dostać przez ikonę na pasku nawigacji widoczną tylko dla dyspozytorów – jest to ikona planszy zadań. Po jej naciśnięciu dyspozytor zostaje przekierowany do strony widocznej na powyższej makiecie.

Panel dyspozytora składa się z kilku elementów. Pierwszym z nich jest zakładka „Create Crew”. Domyślnie widoczny jest tylko jej tytuł – po jego naciśnięciu cały panel rozwija się jak akordeon. Po jego rozwinięciu, użytkownik ma możliwość stworzenia załogi poprzez wpisanie nazwy, wyboru pilotów z listy i zatwierdzenie wyboru przyciskiem „Create”. W zależności od powodzenia lub błędu pojawi się odpowiedni komunikat.

Kolejnym elementem jest zakładka „View All Crews”. Domyślnie widoczny jest tylko jej tytuł – po jego naciśnięciu cały panel rozwija się jak akordeon. Znajdująca się w nim tabela wyświetla dane wszystkich załóg. Użytkownik ma też możliwość filtrowania po nazwie załogi oraz imieniu i nazwisku każdego z pilotów. Naciśnięcie przycisku „Filter” spowoduje filtrowanie danych. W tabeli znajdują się przyciski „Edit” oraz „Delete”. Naciśnięcie na przycisk „Delete” sprawi, że załoga zostanie usunięta. Naciśnięcie przycisku „Edit” spowoduje ukazanie okna dialogowego, w którym pola są wypełnione danymi wybranej załogi.

Diagrama przedstawia okno dialogowe o tytule „Edit Crew”. Wewnątrz znajdują się cztery pola: „Crew Name” (pole tekstowe), „Captain” (menu rozwijane), „First Officer” (menu rozwijane) oraz „Second Officer” (menu rozwijane). Poniżej tych pól znajduje się przycisk „Edit”.

Rysunek 24. Makieta okna dialogowego edytowania załogi. Opracowanie własne.

Wypełnienie danych i naciśnięcie na przycisk „Edit” spowoduje wyświetlenie komunikatu informującego o tym czy lot został stworzony czy nie.

Następnym elementem jest zakładka „Create Flight”. Również działa ona na zasadzie akordeonu. Po jej otwarciu użytkownik może utworzyć nową załogę poprzez wpisanie odpowiednich danych. Żadne z danych nie mogą być puste. Pola „Takeoff Hour” i „Landing Hour” korzystają z time pickera – zegara z którego można wybrać godzinę. Można ją również wpisać ręcznie. Pole „Flight Date” jest polem typu data picker – po jego naciśnięciu ukazuje się mały kalendarz z którego można wybrać datę. Po potwierdzeniu danych i naciśnięciu przycisku „Create”, użytkownik otrzyma komunikat zwrotny czy lot został stworzony czy nie.

Ostatnim elementem panelu dyspozytora jest zakładka „View All Flights”. Jest to panel typu akordeon pozwalający na zobaczenie wszystkich utworzonych lotów. Możliwe jest też filtrowanie danych po numerze lotu, nazwie załogi i dacie wylotu. Podobnie jak w zakładce „View All Crews”, znajdują się tutaj przyciski „Edit”,

„Delete”, oraz nowy przycisk „Reservations”. Przycisk „Delete” pozwala na usunięcie lotu.. Naciśnięcie przycisku „Edit” skutkuje otwarciem się okna dialogowego, które wypełnione jest danymi wybranego lotu.

Edit Flight

Flight Number

Crew

Origin Airport Name

Destination Airport Name

Takeoff Hour

Landing Hour

Plane Type

Total Seats

Flight date

EDIT

☒ IsApproved

Rysunek 25. Makieta okna dialogowego edytowania lotu. Opracowanie własne.

W edycji lotu pojawia się poprzednio niewidoczne pole: „IsApproved”. Jest to checkbox, który dyspozytor może zaznaczyć lub odznaczyć w zależności czy lot został zaakceptowany i ma zostać obsłużony. W przypadku naciśnięcia przycisku „Edit”, dyspozytor otrzyma informację zwrotną o powodzeniu akcji lub jej porażce w komunikacie.

Naciśnięcie na przycisk „Reservations” spowoduje otworenie się okna dialogowego informującego o wszystkich rezerwacjach dokonanych w danym locie.

Reservations			
Seat Number	Name	Phone Number	Email

Rysunek 26. Makieta okna dialogowego podglądu rezerwacji. Opracowanie własne.

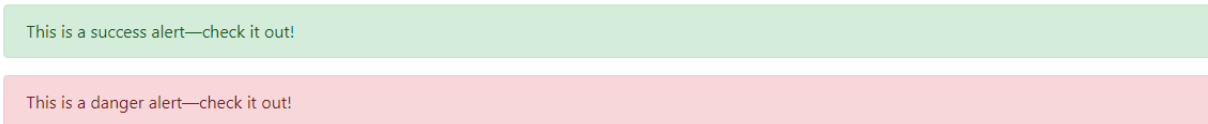
Z poziomu okna podglądu rezerwacji, dyspozytor może wyświetlić dane wszystkich zarezerwowanych miejsc – nr miejsca, imię i nazwisko osoby, która je zarezerwowała a także telefon i adres email ten osoby w razie konieczności kontaktu.

4.4.4. Projekt komunikatów systemu.

Wszystkie komunikaty zwrotne systemu zostaną stworzone z użyciem biblioteki Bootstrap 4. Biblioteka ta dostarcza wiele stylów, w tym style komunikatów. W systemie zostaną wykorzystane dwa rodzaje komunikatów – komunikat o powodzeniu i błędzie.

- Komunikat o powodzeniu będzie używany podczas informacji zwrotnej o powodzeniu wykonania jakiejś akcji.
- Komunikat o błędzie zostanie pokazany w razie wystąpienia jakiegokolwiek błędu – w tym błędów walidacji podczas wpisywania danych.

Poniżej przedstawiono wybrane komunikaty z biblioteki Bootstrap 4.



This is a success alert—check it out!

This is a danger alert—check it out!

Rysunek 27. Lista wybranych komunikatów z biblioteki Bootstrap 4. [6p].

Wszystkie komunikaty będą się pojawiać pod paskiem nawigacji – przykład jest widoczny w podpunkcie 4.4.1.

W przypadku błędów spowodowanych przez walidację danych, komunikat będzie widoczny bezpośrednio pod polem w którym błąd wystąpił.

5. Implementacja systemu

5.1. Implementacja bazy danych

Do zaimplementowania bazy danych nie został wykorzystany żaden ORM. Zamiast tego, posłużyłam się zbiorem bibliotek ADO.NET oraz biblioteką Dapper. Wybrałam to podejście z dwóch względów – niepoprawne napisanie zapytań linq w połączeniu z np. Entity Framework może skutkować ogromną ilością niepotrzebnie wykonywanych zapytań. Dodatkowo w celu uniknięcia pisania ‘surowych’ zapytań, skorzystałam z procedur składowanych. Wnoszą one do środowiska bazodanowego rzeczy takie jak tworzenie parametrów, przetwarzanie warunkowe i możliwości programistyczne. Dodatkowo zapewniają ochronę przed wstrzykiwaniem kodu SQL dzięki walidacji danych.

W celu uproszczenia pracy, w solucji utworzyłam nowy projekt, który przechowuje skrypty do tworzenia wszystkich tabel i procedur składowanych. W praktyce oznacza to, że utworzenie bazy danych oraz wprowadzanie zmian na kodzie SQL jest bardzo proste – po zapisaniu zmian wystarczy jedynie opublikować projekt bazy danych co doprowadza do nadpisania obecnej bazy danych, o ile nie zostały wprowadzone zmiany wymagające usunięcia części danych. Jeżeli baza danych nie istnieje, zostanie ona utworzona. Projekt generuje część skryptu samodzielnie dzięki czemu dostosowuje się do sytuacji.

Poniżej zamieściłam skrypty tworzące wszystkie tabele używane przez system.

```
CREATE TABLE [dbo].[Users]
(
    [Id] INT NOT NULL IDENTITY(1,1) PRIMARY KEY,
    [Username] VARCHAR(50) NOT NULL,
    [Password] VARCHAR(250) NOT NULL,
    [Role] VARCHAR(15) NOT NULL,
    [FirstName] VARCHAR(50) NOT NULL,
    [LastName] VARCHAR(50) NOT NULL,
    [PhoneNumber] INT NULL,
    [Email] VARCHAR(50) NULL,
    [PESEL] CHAR(11) NULL,
    [DocumentID] CHAR(9) NULL
)
```

Rysunek 28. Skrypt SQL tworzący tabelę "Users". Opracowanie własne.

```
CREATE TABLE [dbo].[Crews]
(
    [Id] INT NOT NULL IDENTITY(1,1) PRIMARY KEY,
    [CrewName] NVARCHAR(50) NOT NULL,
    [CaptainID] INT NOT NULL,
    [FirstOfficerID] INT NOT NULL,
    [SecondOfficerID] INT NOT NULL,
    CONSTRAINT [FK_CaptainID_UserID] FOREIGN KEY ([CaptainID]) REFERENCES [Users]([Id]),
    CONSTRAINT [FK_FirstOfficerID_UserID] FOREIGN KEY ([FirstOfficerID]) REFERENCES [Users]([Id]),
    CONSTRAINT [FK_SecondOfficerID_UserID] FOREIGN KEY ([SecondOfficerID]) REFERENCES [Users]([Id])
)
```

Rysunek 29. Skrypt SQL tworzący tabelę "Crews". Opracowanie własne.

```
CREATE TABLE [dbo].[Flights]
(
    [Id] INT NOT NULL IDENTITY(1,1) PRIMARY KEY,
    [CrewID] INT NOT NULL,
    [FlightNumber] NVARCHAR(6) NOT NULL,
    [StartingAirportName] NVARCHAR(150) NOT NULL,
    [DestinationAirportName] NVARCHAR(150) NOT NULL,
    [FlightDate] DATE NOT NULL,
    [TakeoffHour] VARCHAR(8) NOT NULL,
    [LandingHour] VARCHAR(8) NOT NULL,
    [PlaneType] NVARCHAR(25) NOT NULL,
    [TotalSeats] INT NOT NULL,
    [IsApproved] BIT NOT NULL,
    [IsCompleted] BIT NOT NULL,
    CONSTRAINT [FK_CrewID_CrewID] FOREIGN KEY ([CrewID]) REFERENCES [Crews]([Id])
)
```

Rysunek 30. Skrypt SQL tworzący tabelę "Flights". Opracowanie własne.

Warto wspomnieć, że w tabeli „Flights” nastąpiła pewna zmiana. Podczas projektu zdefiniowane zostało pole o nazwie „RemainingSeats”. Ostatecznie jednak nie zostało nigdzie wykorzystane w systemie, więc zostało usunięte.

```
CREATE TABLE [dbo].[Reservations]
(
    [Id] INT NOT NULL IDENTITY(1,1) PRIMARY KEY,
    [FlightID] INT NOT NULL,
    [UserID] INT NOT NULL,
    [Price] DECIMAL(7, 2) NOT NULL,
    [SeatNumber] NVARCHAR(4) NOT NULL,
    [TransactionID] NVARCHAR(100) NOT NULL,
    [IsValid] BIT NOT NULL,
    CONSTRAINT [FK_ReservationFlightID_ToFlightID] FOREIGN KEY ([FlightID]) REFERENCES [Flights]([Id]),
    CONSTRAINT [FK_ReservationsUserID_ToUserID] FOREIGN KEY ([UserID]) REFERENCES [Users]([Id])
)
```

Rysunek 31. Skrypt SQL tworzący tabelę "Reservations". Opracowanie własne.

Dodatkowo przedstawię kilka skryptów tworzących wybrane procedury składowane. W sumie utworzone zostało 26 procedur składowanych.

```
CREATE PROCEDURE [dbo].[CreateCrew]
    @CrewName nvarchar(50),
    @CaptainID int,
    @FirstOfficerID int,
    @SecondOfficerID int
AS
BEGIN
    INSERT INTO Crews
    (
        [CrewName],
        [CaptainID],
        [FirstOfficerID],
        [SecondOfficerID]
    )
    VALUES
    (
        @CrewName,
        @CaptainID,
        @FirstOfficerID,
        @SecondOfficerID
    )
END
```

Rysunek 32. Skrypt SQL tworzący procedurę składowaną "CreateCrew". Opracowanie własne.

```
CREATE PROCEDURE [dbo].[GetFlightsByPilotId]
    @Id int
AS
BEGIN
    SELECT *
    FROM Crews
    JOIN Flights
    ON Crews.Id=Flights.CrewID
    WHERE [CaptainID] = @Id OR
    [FirstOfficerID] = @Id OR
    [SecondOfficerID] = @Id
END
```

Rysunek 33. Skrypt SQL tworzący procedurę składowaną "GetFlightsByPilotId". Opracowanie własne.

```

CREATE PROCEDURE [dbo].[GetUserById]
    @Id int
AS
BEGIN
    SELECT
        [Id],
        [Username],
        [Password],
        [Role],
        [FirstName],
        [LastName],
        [PhoneNumber],
        [Email],
        [PESEL],
        [DocumentID]
    FROM Users
    WHERE [Id] = @Id
END

```

Rysunek 34. Skrypt SQL tworzący procedurę składowaną "GetUserById". Opracowanie własne.

W celu stworzenia połączenia między aplikacją a bazą danych poczyniłam dwie rzeczy:

W pierwszej kolejności, do pliku appsettings.json dodana została wartość odpowiadająca za connection string do bazy danych.

```

1 {
2   "ConnectionStrings": {
3     "Database": "Data Source=.;Initial Catalog=AirlineServiceSoftware.Database;Integrated Security=True"
4   },
5   "AppSettings": {
6     "Secret": "Bebzen i Toluen to najlepsi przyjaciele"
7   },
8   "Logging": {
9     "LogLevel": {
10      "Default": "Information",
11      "Microsoft": "Warning",
12      "Microsoft.Hosting.Lifetime": "Information"
13    }
14  },
15  "AllowedHosts": "*"
16 }

```

Rysunek 35. Zawartość pliku "appsettings.json". Opracowanie własne.

Następnie ta wartość jest używana w pliku Startup.cs podczas wstrzykiwania zależności wszystkich serwisów używanych przez aplikację. Pozwala to na wstrzyknięcie connection stringa bezpośrednio do konstruktora tych serwisów. Plusem takiego podejścia jest to, że można zdefiniować kilka connection stringów prowadzących do np. zapasowych baz danych lub po prostu gdy nasze dane są podzielone na kilka bazy danych. W moim wypadku wystarczyło jednak użyć tylko jednego connection stringa.


```
// dependency injection of all the services
services.AddScoped<IUserDataService, UserDataService>(ctor:IServiceProvider =>
    new UserDataService(Configuration.GetConnectionString(name: "Database")));
services.AddScoped<ICrewDataService, CrewDataService>(ctor:IServiceProvider =>
    new CrewDataService(Configuration.GetConnectionString(name: "Database")));
services.AddScoped<IFlightDataService, FlightDataService>(ctor:IServiceProvider =>
    new FlightDataService(Configuration.GetConnectionString(name: "Database")));
services.AddScoped<IReservationDataService, ReservationDataService>(ctor:IServiceProvider =>
    new ReservationDataService(Configuration.GetConnectionString(name: "Database")));
services.AddScoped<IUserService, UserService>();
services.AddScoped<ICrewService, CrewService>();
services.AddScoped<IFlightService, FlightService>();
services.AddScoped<IReservationService, ReservationService>();
```

Rysunek 36. Część pliku Startup.cs ukazująca wstrzykiwanie connection string do serwisów.


Opracowanie własne.

Po zdefiniowaniu połączenia z bazą danych i utworzeniu wszystkich tabel oraz procedur składowych, została utworzona baza danych o nazwie „AirlineServiceSoftware.Database”.


Poniżej pokazane zostały zrzuty ekranu dla wszystkich tabel w systemie.

	Name	Data Type	Allow Nulls
	Id	int	<input type="checkbox"/>
	Username	varchar(50)	<input type="checkbox"/>
	Password	varchar(250)	<input type="checkbox"/>
	Role	varchar(15)	<input type="checkbox"/>
	FirstName	varchar(50)	<input type="checkbox"/>
	LastName	varchar(50)	<input type="checkbox"/>
	PhoneNumber	int	<input checked="" type="checkbox"/>
	Email	varchar(50)	<input checked="" type="checkbox"/>
	PESEL	char(11)	<input checked="" type="checkbox"/>
	DocumentID	char(9)	<input checked="" type="checkbox"/>
			<input type="checkbox"/>


Rysunek 37. Tabela "Users". Opracowanie własne.

	Name	Data Type	Allow Nulls
	Id	int	<input type="checkbox"/>
	FlightID	int	<input type="checkbox"/>
	UserID	int	<input type="checkbox"/>
	Price	decimal(7,2)	<input type="checkbox"/>
	SeatNumber	nvarchar(4)	<input type="checkbox"/>
	TransactionID	nvarchar(100)	<input type="checkbox"/>
	IsValid	bit	<input type="checkbox"/>
			<input type="checkbox"/>

Rysunek 38. Tabela "Reservations". Opracowanie własne.

	Name	Data Type	Allow Nulls
	Id	int	<input type="checkbox"/>
	CrewID	int	<input type="checkbox"/>
	FlightNumber	nvarchar(6)	<input type="checkbox"/>
	StartingAirportName	nvarchar(150)	<input type="checkbox"/>
	DestinationAirportName	nvarchar(150)	<input type="checkbox"/>
	FlightDate	date	<input type="checkbox"/>
	TakeoffHour	varchar(8)	<input type="checkbox"/>
	LandingHour	varchar(8)	<input type="checkbox"/>
	PlaneType	nvarchar(25)	<input type="checkbox"/>
	TotalSeats	int	<input type="checkbox"/>
	IsApproved	bit	<input type="checkbox"/>
	IsCompleted	bit	<input type="checkbox"/>
			<input type="checkbox"/>

Rysunek 39. Tabela "Flights". Opracowanie własne.

	Name	Data Type	Allow Nulls
	Id	int	<input type="checkbox"/>
	CrewName	nvarchar(50)	<input type="checkbox"/>
	CaptainID	int	<input type="checkbox"/>
	FirstOfficerID	int	<input type="checkbox"/>
	SecondOfficerID	int	<input type="checkbox"/>
			<input type="checkbox"/>

Rysunek 40. Tabela "Crews". Opracowanie własne.

5.2. Implementacja logiki systemu

W poniższej tabeli zestawione zostały wszystkie klasy przedstawione w projekcie z klasami zaimplementowanymi

Nazwa klasy	Projekt	Implementacja
UsersController	+	+
AuthenticateModel	+	+
Role	+	+
User	+	+
IUserService	+	+
UserService	+	+
IMediatorServices	+	+
AppSettings	+	+
EditUserRequest	+	+
EditUserRequestHandler	+	+
CreateUserRequest	+	+
CreateUserRequestHandler	+	+
GetAllUsersRequest	+	+
GetAllUsersRequestHandler	+	+
DeleteUserRequest	+	+
DeleteUserRequestHandler	+	+
GetUserByIdRequest	+	+
GetUserByIdRequestHandler	+	+
GetUserByUsernameRequest	+	+
GetUserByUsernameRequestHandler	+	+
IUserDataService	+	+
UserDataService	+	+
CrewsController	+	+
Crew	+	+
CrewResponse	+	+
ICrewService	+	+
CrewService	+	+
CreateCrewRequest	+	+
CreateCrewRequestHandler	+	+
EditCrewRequest	+	+
EditCrewRequestHandler	+	+
DeleteCrewRequest	+	+
DeleteCrewRequestHandler	+	+
GetAllCrewsRequest	+	+
GetAllCrewsRequestHandler	+	+
GetAllPilotsRequest	+	+
GetAllPilotsRequestHandler	+	+
ICrewDataService	+	+
CrewDataService	+	+
FlightsController	+	+
Flight	+	+
FlightResponse	+	+
SearchParameters	+	+
IFlightService	+	+

FlightService	+	+
CreateFlightRequest	+	+
CreateFlightRequestHandler	+	+
DeleteFlightRequest	+	+
DeleteFlightRequestHandler	+	+
EditFlightRequest	+	+
EditFlightRequestHandler	+	+
EditFlightStatusRequest	+	+
EditFlightStatusRequestHandler	+	+
GetAllFlightsRequest	+	+
GetAllFlightsRequestHandler	+	+
GetFlightByIdRequest	+	+
GetFlightByIdRequestHandler	+	+
GetPilotFlightsRequest	+	+
GetPilotFlightsRequestHandler	+	+
SearchFlightsRequest	+	+
SearchFlightsRequestHandler	+	+
IFlightDataService	+	+
FlightDataService	+	+
ReservationsController	+	+
IReservationService	+	+
ReservationService	+	+
Reservation	+	+
ReservationResponse	+	+
ReservationUserResponse	+	+
CreateReservationRequest	+	+
CreateReservationRequestHandler	+	+
GetFlightReservationsRequest	+	+
GetFlightReservationsRequestHandler	+	+
GetTakenSeatsRequest	+	+
GetTakenSeatsRequestHandler	+	+
GetUserReservationsRequest	+	+
GetUserReservationsRequestHandler	+	+
EditReservationRequest	+	+
EditReservationRequestHandler	+	+
IReservationDataService	+	+
ReservationDataService	+	+
Startup	+	+
Program	+	+
ExtensionMethods	-	+

Tabela 15. Implementacja klas systemu. Opracowanie własne.

Podczas implementacji pojawiła się potrzeba stworzenia tylko jednej dodatkowej klasy. Jest to klasa o nazwie ExtensionMethods. Pozwala ona na rozszerzenie funkcjonalności niektórych klas. W poniższej tabeli znajdzie się jej dokładny opis.

Klasa	Opis	Metody
ExtensionMethods	Klasa rozszerzająca inne klasy o dodatkowe funkcjonalności.	WithoutPasswords(users: IEnumerable<User>): IEnumerable<User> WithoutPassword(user: User): User IsValidPESEL(input: string): bool CalculateControlSum(input: string, weights: int[], offset: int): int GetLetterValue(letter: char): int IsValidID(this id: string): bool IsValidPassword(password: string): bool

Tabela 16. Opis klas pominiętych w projekcie systemu. Opracowanie własne.

Następnie przedstawione zostaną metody dwóch wybranych klas systemu – klasy UsersController oraz ReservationDataService. Zostaną one porównane z ich projektem.

Nazwa Metody	Zwracana wartość	Projekt	Implementacja
Authenticate	ActionResult	+	+
Getusers	ActionResult	+	+
GetById	ActionResult	+	+
CreateUser	ActionResult	+	+
EditUser	ActionResult	+	+
DeleteUser	ActionResult	+	+

Tabela 17. Zaimplementowane metody klasy UsersController. Opracowanie własne.

Nazwa Metody	Zwracana wartość	Projekt	Implementacja
CreateReservation	Task<bool>	+	+
GetFlightReservations	Task<IEnumerable<Reservation>>	+	+
GetUserReservations	Task<IEnumerable<Reservation>>	+	+
GetTakenSeats	Task<IEnumerable<string>>	+	+
EditReservation	Task<bool>	+	+

Tabela 18. Zaimplementowane metody klasy *ReservationDataService*. Opracowanie własne.

W następnej kolejności przedstawię zrzuty ekranu wybranych zaimplementowanych klas.

```
1 using AirlineServiceSoftware.Entities;
2 using AirlineServiceSoftware.Helpers;
3 using AirlineServiceSoftware.Models;
4 using AirlineServiceSoftware.Services;
5 using Microsoft.AspNetCore.Authorization;
6 using Microsoft.AspNetCore.Mvc;
7
8 namespace AirlineServiceSoftware.Controllers
9 {
10     [Authorize]
11     [Route("[controller]")]
12     [ApiController]
13     public class UsersController : ControllerBase
14     {
15         private readonly IUserService _userService;
16
17         public UsersController(IUserService userService)
18         {
19             _userService = userService;
20         }
21
22         [AllowAnonymous]
23         [HttpPost("authenticate")]
24         public IActionResult Authenticate([FromBody] AuthenticateModel model)
25         {
26             var user = _userService.Authenticate(model.Username, model.Password);
27             if (user == null)
28             {
29                 return BadRequest(new { message = "Username or Password is incorrect" });
30             }
31             return Ok(user);
32         }
33
34         [Authorize(Roles = Role.Admin)]
35         [HttpGet("GetUsers")]
36         public IActionResult GetAllUsers()
37         {
38             var users = _userService.GetAllUsers();
39             return Ok(users);
40         }
41
42         [HttpGet("GetUsers/{id?}")]
43         public IActionResult GetUserById(int id)
44         {
```



```

45     var currentUserId = int.Parse(User.Identity.Name);
46     if (id != currentUserId && !User.IsInRole(Role.Admin))
47     {
48         return Forbid();
49     }
50
51     var user = _userService.GetUserById(id);
52     if (user == null)
53     {
54         return NotFound();
55     }
56     return Ok(user);
57
58
59     [Authorize(Roles = Role.Admin)]
60     [HttpPost("CreateUser")]
61     public IActionResult CreateUser([FromBody] User UserData)
62     {
63         var dataValidationResult = UserData.Password.IsValidPassword(); if (dataValidationResult == false) return BadRequest(new { message = "Invalid data." });
64         if (UserData.Pesel != null) dataValidationResult = UserData.Pesel.IsValidPESEL(); if (dataValidationResult == false) return BadRequest(new { message = "Invalid data." });
65         if (UserData.DocumentId != null) dataValidationResult = UserData.DocumentId.IsValidID(); if (dataValidationResult == false) return BadRequest(new { message = "Invalid data." });
66         var result = _userService.CreateUser(UserData);
67         if (!result)
68         {
69             return BadRequest(new { message = "User was not added." });
70         }
71
72         return Ok(result);
73     }
74
75     [Authorize(Roles = Role.Admin)]
76     [HttpPost("EditUser")]
77     public IActionResult EditUser([FromBody] User UserData)
78     {
79         var dataValidationResult = true;
80         if (UserData.Password != null) dataValidationResult = UserData.Password.IsValidPassword(); if (dataValidationResult == false) return BadRequest(new { message = "Invalid data." });
81         if (UserData.Pesel != null) dataValidationResult = UserData.Pesel.IsValidPESEL(); if (dataValidationResult == false) return BadRequest(new { message = "Invalid data." });
82         if (UserData.DocumentId != null) dataValidationResult = UserData.DocumentId.IsValidID(); if (dataValidationResult == false) return BadRequest(new { message = "Invalid data." });
83         var result = _userService.EditUser(UserData);
84         if (!result)
85         {
86             return BadRequest(new { message = "User was not modified." });
87         }
88         return Ok(result);
89     }
90
91     [Authorize(Roles = Role.Admin)]
92     [HttpDelete("DeleteUser/{Id}")]
93     public IActionResult DeleteUser(int Id)
94     {
95         var result = _userService.DeleteUser(Id);
96         if (!result)
97         {
98             return BadRequest(new { message = "User was not deleted." });
99         }
100         return Ok(result);
101     }
102 }
103
104

```

Rysunek 41. Zrzut ekranu klasy UsersController. Opracowanie własne.

Klasa UsersController to kontroler API, który obsługuje przychodzące żądania HTTP. Ten specyficzny kontroler obsługuje wszystkie żądania związane z obsługą danych użytkowników. Wszystkie kontrolery API w implementacji podążają za prostym schematem jeżeli chodzi o definicję ścieżek: „[nazwa kontrolera]/[nazwa akcji]”.

```

1 using System;
2 using System.Collections.Generic;
3 using System.Data;
4 using System.Data.SqlClient;
5 using System.Linq;
6 using System.Threading.Tasks;
7 using AirlineServiceSoftware.Entities;
8 using AirlineServiceSoftware.Helpers;
9 using AirlineServiceSoftware.Mediators.MediatorsRequests.Reservations;
10 using Dapper;
11
12 namespace AirlineServiceSoftware.DataAccess
13 {
14     public class ReservationDataService : IReservationDataService
15     {
16         private readonly string _connectionString;
17
18         public ReservationDataService(string connectionString)
19         {
20             if (string.IsNullOrEmpty(connectionString))
21             {
22                 throw new ArgumentException("message", nameof(connectionString));
23             }
24
25             _connectionString = connectionString;
26         }
27
28         public async Task<bool> CreateReservation(CreateReservationRequest request)
29         {
30             try
31             {
32                 await using (var conn = new SqlConnection(_connectionString))
33                 {
34                     conn.Open();
35
36                     var parameters = new DynamicParameters();
37                     parameters.Add("@FlightId", request.FlightId);
38                     parameters.Add("@UserId", request.UserId);
39                     parameters.Add("@Price", request.Price);
40                     parameters.Add("@SeatNumber", request.SeatNumber);
41                     parameters.Add("@TransactionId", request.TransactionId);
42                     parameters.Add("@IsValid", request.IsValid);
43
44                     conn.Query<bool>("CreateReservation", parameters, commandType: CommandType.StoredProcedure);
45                 }
46
47                 return true;
48             }
49             catch (Exception ex)
50             {
51                 Console.WriteLine(ex);
52                 return false;
53             }
54         }
55
56         public async Task<IEnumerable<ReservationUserResponse>> GetFlightReservations(GetFlightReservationsRequest request)
57         {
58             await using (var conn = new SqlConnection(_connectionString))
59             {
60                 conn.Open();
61
62                 var parameters = new DynamicParameters();
63                 parameters.Add("@Id", request.Id);
64
65                 var results = conn.Query<ReservationResponse>("GetFlightReservations", parameters, commandType: CommandType.StoredProcedure);
66                 List<ReservationUserResponse> list = new List<ReservationUserResponse>();
67                 foreach (var result in results)
68                 {
69                     User newUser = new User();
70                     var newParameters = new DynamicParameters();
71                     newParameters.Add("@Id", result.UserId);
72                     newUser = conn.Query<User>("GetUserById", newParameters, commandType: CommandType.StoredProcedure).FirstOrDefault();
73                     var newReservation = new ReservationUserResponse()
74                     {
75                         Id = result.Id,
76                         FlightId = result.FlightId,
77                         IsValid = result.IsValid,
78                         Price = result.Price,
79                         SeatNumber = result.SeatNumber,
80                         TransactionId = result.TransactionId,
81                         User = newUser.WithoutPassword()
82                     };
83                     list.Add(newReservation);
84                 }
85
86                 IEnumerable<ReservationUserResponse> reservations = list;
87                 return reservations;
88             }
89         }
90
91         public async Task<IEnumerable<Reservation>> GetUserReservations(GetUserReservationsRequest request)

```

```

92 |         {
93 |             await using (var conn = new SqlConnection(_connectionString))
94 |             {
95 |                 conn.Open();
96 |
97 |                 var parameters = new DynamicParameters();
98 |                 parameters.Add("@Id", request.Id);
99 |
100 |                 var results = conn.Query<ReservationResponse>("GetUserReservations", parameters, commandType: CommandType.StoredProcedure);
101 |                 List<Reservation>list = new List<Reservation>();
102 |
103 |                 foreach (var result in results)
104 |                 {
105 |                     Flight newFlight = new Flight();
106 |                     var newParameters = new DynamicParameters();
107 |                     newParameters.Add("@Id", result.FlightId);
108 |                     newFlight = conn.Query<Flight>("GetFlightById", parameters, commandType: CommandType.StoredProcedure).FirstOrDefault();
109 |                     var newReservation = new Reservation
110 |                     {
111 |                         Id = result.Id,
112 |                         Flight = newFlight,
113 |                         IsValid = result.IsValid,
114 |                         Price = result.Price,
115 |                         SeatNumber = result.SeatNumber,
116 |                         TransactionId = result.TransactionId,
117 |                         UserId = result.UserId
118 |                     };
119 |
120 |                     list.Add(newReservation);
121 |                 }
122 |
123 |                 IEnumerable<Reservation> reservations = list;
124 |                 return reservations;
125 |             }
126 |         }
127 |     }
128 |
129 |     2 references
130 |     public async Task<IEnumerable<string>> GetTakenSeats(GetTakenSeatsRequest request)
131 |     {
132 |         await using (var conn = new SqlConnection(_connectionString))
133 |         {
134 |             conn.Open();
135 |
136 |             var parameters = new DynamicParameters();
137 |             parameters.Add("@Id", request.Id);
138 |
139 |             var results = conn.Query<string>("GetTakenSeats", parameters, commandType: CommandType.StoredProcedure);
140 |             return results;
141 |         }
142 |     }
143 |
144 |     2 references
145 |     public async Task<bool> EditReservation(EditReservationRequest request)
146 |     {
147 |         try
148 |         {
149 |             await using (var conn = new SqlConnection(_connectionString))
150 |             {
151 |                 conn.Open();
152 |
153 |                 var parameters = new DynamicParameters();
154 |                 parameters.Add("@Id", request.Id);
155 |                 parameters.Add("@IsValid", request.IsValid);
156 |                 var result = conn.Query<bool>("EditReservation", parameters, commandType: CommandType.StoredProcedure);
157 |                 return true;
158 |             }
159 |         }
160 |         catch (Exception ex)
161 |         {
162 |             Console.WriteLine(ex);
163 |             return false;
164 |         }
165 |     }

```

Rysunek 42. Zrzuty ekranu dla klasy ReservationDataService. Opracowanie własne.

Klasa ReservationDataService to klasa, która odpowiada za połączenia z bazą danych. Obsługuje ona wszystkie połączenia z bazą dotyczące Rezerwacji. Wszystkie połączenia z bazą są zawarte w instrukcji using – takie rozwiązanie zapewnia, że po wykonaniu odpowiednich czynności połączenie zostanie zamknięte.

```

1 using AirlineServiceSoftware.Entities;
2 using AirlineServiceSoftware.Helpers;
3 using AirlineServiceSoftware.Mediators.MediatorsRequests.Crews;
4 using MediatR;
5 using System.Collections.Generic;
6
7 namespace AirlineServiceSoftware.Services.Crews
8 {
9     2 references
10    public class CrewService : ICrewService
11    {
12        private readonly IMediator _mediator;
13        0 references
14        public CrewService(IMediator mediator)
15        {
16            this._mediator = mediator;
17        }
18        2 references
19        public IEnumerable<User> GetAllPilots()
20        {
21            var pilots = _mediator.Send(new GetAllPilotsRequest()).Result;
22            pilots = pilots.WithoutPasswords();
23            return pilots;
24        }
25        2 references
26        public IEnumerable<Crew> GetAllCrews()
27        {
28            var crews = _mediator.Send(new GetAllCrewsRequest()).Result;
29            return crews;
30        }
31        2 references
32        public bool CreateCrew(Crew newCrew)
33        {
34            var result = _mediator.Send(new CreateCrewRequest()
35            {
36                CrewName = newCrew.CrewName,
37                Captain = newCrew.Captain,
38                FirstOfficer = newCrew.FirstOfficer,
39                SecondOfficer = newCrew.SecondOfficer
40            }).Result;
41            return result;
42        }
43        2 references
44        public bool DeleteCrew(int Id)

```

```

44 |
45 | {
46 |     var result = _mediator.Send(new DeleteCrewRequest()
47 |     {
48 |         Id = Id
49 |     }).Result;
50 |     return result;
51 | }
52 |
53 | public bool EditCrew(Crew editCrew)
54 | {
55 |     var result = _mediator.Send(new EditCrewRequest()
56 |     {
57 |         Id = editCrew.Id,
58 |         CrewName = editCrew.CrewName,
59 |         Captain = editCrew.Captain,
60 |         FirstOfficer = editCrew.FirstOfficer,
61 |         SecondOfficer = editCrew.SecondOfficer
62 |     }).Result;
63 |     return result;
64 | }
65 | }
66 |
67 | }
68 |

```

Rysunek 43. Zrzuty ekranu dla klasy CrewService. Opracowanie własne.

Klasa CrewService to klasa odpowiedzialna za tworzenie odpowiednich żądań, które będą wykorzystywane przez bibliotekę MediatR. Każda metoda, która tego wymaga, zasila dane żądanie potrzebnymi danymi, a następnie przekazuje to żądanie do instancji mediator-a, która została pobrana w konstruktorze.

```

1 using AirlineServiceSoftware.Entities;
2 using MediatR;
3 using System;
4
5 namespace AirlineServiceSoftware.Mediators.MediatorsRequests.Flights
6 {
7     public class CreateFlightRequest : IRequest<bool>
8     {
9         public Crew Crew { get; set; }
10         public string FlightNumber { get; set; }
11         public string StartingAirportName { get; set; }
12         public string DestinationAirportName { get; set; }
13         public DateTime FlightDate { get; set; }
14         public string TakeoffHour { get; set; }
15         public string LandingHour { get; set; }
16         public string PlaneType { get; set; }
17         public int TotalSeats { get; set; }
18         public int RemainingSeats { get; set; }
19         public bool IsApproved { get; set; }
20         public bool IsCompleted { get; set; }
21     }
22 }
23

```

Rysunek 44. Zrzut ekranu dla klasy *CreateFlightRequest*. Opracowanie własne.

Klasa *CreateFlightRequest* to klasa, będąca żądaniem wykorzystywanym przez bibliotekę MediatR. Jej działanie jest możliwe poprzez dziedziczenie klasy *IRequest*. Jej celem jest przechowanie parametrów żądania.

```

1 using AirlineServiceSoftware.DataAccess;
2 using AirlineServiceSoftware.Mediators.MediatorsRequests.Flights;
3 using Mediatr;
4 using System;
5 using System.Threading;
6 using System.Threading.Tasks;
7
8 namespace AirlineServiceSoftware.Mediators.MediatorsRequestsHandler.Flights
9 {
10     1 reference | Benzen, 17 days ago | 1 author, 1 change
11     public class CreateFlightRequestHandler : IRequestHandler<CreateFlightRequest, bool>
12     {
13         private readonly IFlightDataService _flightDataService;
14         0 references | Benzen, 17 days ago | 1 author, 1 change
15         public CreateFlightRequestHandler(IFlightDataService dataAccessService)
16         {
17             _flightDataService = dataAccessService ?? throw new ArgumentNullException(nameof(dataAccessService));
18             _flightDataService = dataAccessService;
19         }
20         23 references | Benzen, 17 days ago | 1 author, 1 change
21         public async Task<bool> Handle(CreateFlightRequest request, CancellationToken cancellationToken)
22         {
23             return await _flightDataService.CreateFlight(request);
24         }
25     }

```

Rysunek 45. Zrzut ekranu dla klasy *CreateFlightRequestHandler*. Opracowanie własne.

Klasa *CreateFlightRequestHandler*, to klasa, która obsługuje poprzednio opisane wydarzenie (*CreateFlightRequest*). Dziedziczy ona po *IRequestHandler<CreateFlightRequest, bool>* - właśnie ta deklaracja dziedziczenia ustala na które żądanie ta klasa ma odpowiadać. Drugi parametr to typ danych, który ma zostać zwrócony. Metoda *Handle* obsługuje żądanie poprzez skierowanie go do odpowiedniej metody w odpowiedniej klasie typu *DataService*. W tym wypadku jest to *FlightDataService*.

5.3. Implementacja interfejsu użytkownika

Do zaimplementowania interfejsów graficznych wykorzystany został framework Angular wraz z TypeScript. Angular przejmuje obowiązki, które kiedyś należały do ASP.NET MVC. Komponenty są głównym budulcem w frameworku Angular. Działają podobnie do elementów HTML i zawierają skrypty oraz pliki stylów. Żeby aplikacja działała poprawnie, musi zostać utworzony co najmniej jeden moduł. Moduły dzielą aplikację na pewne jednostki [5, s.53]. W implementacji stworzono kilka takich modułów, każdy odpowiada roli użytkownika co pozwala na łatwy podział obowiązków i dostępów a dodatkowo ułatwia organizację plików. Ponieważ tworzenie ładnego interfejsu od zera jest ciężkie i czasochłonne, posłużono się następującymi bibliotekami:

- Ngx-material-timepicker v5.5.3 [9p]
- Angular material v8.2.3 [10p]

- Bootstrap v4.5.3 [11p]
- jQuery v3.5.1 [12p]

Dodatkowo w celu przetworzenia płatności za rezerwację wykorzystany został PayPal SDK. Utworzenie własnej bramki płatności mogło by być oddzielnym tematem na pracę inżynierską z powodu złożoności tematu. Z tego powodu podjęta została decyzja o wykorzystaniu gotowego i bezpiecznego rozwiązania.

Utworzone zostało około 13 komponentów, które odpowiadają za przetwarzanie danych przysyłanych z back-endu i pokazanie je klientowi w czytelny sposób. Z tego powodu każdy z komponentów składa się z wielu metod, które pomagają osiągnąć ten cel.

Poniżej jako przykład pokażę komponent Checkout.

```

1  import { Component, OnInit } from '@angular/core';
2  import { MatDialog } from '@angular/material';
3  import { ActivatedRoute } from '@angular/router';
4  import { IFlight } from '../dispatcher/interfaces/iFlight';
5  import { IUser } from '../login/interfaces/iUser';
6  import { AuthenticationService } from '../services/authentication.service';
7  import { PaymentDialogComponent } from '../payment-dialog/payment-dialog.component';
8  import { CheckoutService } from '../services/checkout.service';
9  // tslint:disable: deprecation

10 @Component({
11   templateUrl: 'checkout.component.html',
12   styleUrls: ['./checkout.component.scss']
13 })
14
15 export class CheckoutComponent implements OnInit {
16   private sub: any;
17   flightId: number;
18   selectedFlight: IFlight;
19   user: IUser;
20   seats: string[] = [];
21   takenSeats: string[] = [];
22   configuration: boolean;
23
24   constructor(private route: ActivatedRoute,
25     private checkoutService: CheckoutService,
26     private authenticationService: AuthenticationService,
27     public dialog: MatDialog) {
28     this.selectedFlight = {} as IFlight;
29     this.seats = [] as string[];
30     this.takenSeats = [] as string[];
31     this.configuration = false;
32   }
33   ngOnInit(): void {
34     this.user = this.authenticationService.userValue;
35
36     this.sub = this.route.params.subscribe(params => {
37       this.flightId = +params['id'];
38     });
39
40     this.checkoutService.getFlight(this.flightId).subscribe(flight => {

```



```

41         this.selectedFlight = flight;
42         this.configureSeats();
43         this.getBoughtSeats();
44     },
45     error => {
46         console.log(error);
47     });
48 }
49
50 configureSeats(): void {
51     const rows = Math.floor(this.selectedFlight.totalSeats / 6);
52     console.log(rows);
53     for (let i = 0; i < rows; i++) {
54         this.seats[i] = (i + 1).toString();
55     }
56     this.configuration = true;
57 }
58
59 onClickProceedToPaypal(seat: string): void {
60     this.dialog.open(PaymentDialogComponent,
61     {
62         width: '50%',
63         data: {seat: seat, flight: this.selectedFlight, user: this.user}
64     });
65 }
66
67 getBoughtSeats() {
68     this.checkoutService.getTakenSeatNumbers(this.selectedFlight.id).subscribe(seats => {
69         this.takenSeats = seats;
70     },
71     error => {
72         console.log(error);
73     });
74 }
75
76 checkSeatValidity = function(seatNumber: string): boolean {
77     let result = this.takenSeats.find(element => element === seatNumber);
78     if (result === undefined) { return false; }
79     return true;
80 };
81 }

```

Rysunek 46. Zrzuty ekranu komponentu Checkout. Opracowanie własne.

Poniżej przedstawię ciąg interfejsów, które zobaczyłby dyspozytor chcący utworzyć nowy lot. Pokażę zrzuty ekranu interfejsów a także opiszę poszczególne kroki postępowania.

Po pierwsze, dyspozytor musi zalogować się w systemie. Z tego powodu udaje się on na stronę logowania serwisu.

Linia Lotnicza

Login

Username

Password

Login

Rysunek 47. Strona logowania. Opracowanie własne.

Dyspozytor wpisuje swoje dane logowania po czym zostaje przekierowany do strony dla dyspozytorów.

Dispatcher

Create Crew

View All Crews

Create Flight

View all flights

Rysunek 48. Zwinięty panel dyspozytora. Opracowanie własne.

Pierwotnie cały panel jest zwinięty. Dopiero po naciśnięciu wybranej pozycji rozwija się on do pełnej wersji i pozwala na wprowadzenie danych. W tym wypadku dyspozytor naciska na kartę z napisem „Create Flight”.

Dispatcher

Create Crew

View All Crews

Create Flight

Flight Number *

Crew *

Starting Airport Name *

Destination Airport name *

Flight Date *
3/17/2021

Takeoff Hour *

Landing Hour *

Plane Type *

Total Seats *

Create

View all flights

Rysunek 49. Panel dyspozytora - rozwinięta zakładka "Create Flight". Opracowanie własne.

Po rozwinięciu się panelu dyspozytor może wpisać dane lotu. Wszystkie dane są wymagane a więc jeżeli któreś z pól nie zostanie wypełnione podświetli się ono na czerwono. Przycisk „Create” będzie wtedy bezużyteczny i nie dopuści do przesłania przez użytkownika danych. Jeżeli jednak dane zostaną wpisane poprawnie, lot zostanie utworzony.

W przypadku gdy dyspozytor chce się upewnić czy aby na pewno wszystkie dane wpisał poprawnie, może otworzyć zakładkę „View All Flights”.

Dispatcher

Create Crew

View All Crews

Create Flight

View all flights

Filter

Filter

Id	Crew Name	Flight Number	Starting Airport Name	Destination Airport Name	Flight Date	Status	Edit	Delete	View Reservations
2	TestTest	LO5055	Warsaw Chopin Airport	JFK Airport	Mar 2, 2021	▶	Edit	Delete	Reservations
3	aaaaa	LO015	JFK Airport	Warsaw Chopin Airport	Mar 3, 2021		Edit	Delete	Reservations
4	TestTest	LO3924	Rzeszów International	Warsaw Chopin Airport	Mar 9, 2021		Edit	Delete	Reservations

Rysunek 50. Panel dyspozytora - rozwinięta zakładka "View All Flights". Opracowanie własne.

Dyspozytor ma możliwość wyświetlenia wszystkich lotów z wyżej wymienionej zakładki. W razie potrzeby rozszerza się ona żeby pokazać wszystkie loty. Jeżeli dyspozytor popełnił błąd, ma on możliwość edycji lotu poprzez naciśnięcie przycisku „Edit” w odpowiednim rzędzie danych.

Dispatcher

Create Crew

View All Crews

Create Flight

View s

Flight Number * LO5055

Crew * TestTest

Starting Airport Name * Warsaw Chopin Airport

Destination Airport name * JFK Airport

Flight Date * 3/2/2021

Takeoff Hour * 11:09

Landing Hour * 21:21

Plane Type * Airbus A320

Total Seats * 312

☒ Approve flight ☐ Complete flight

Edit

Id	Flight Number	Starting Airport Name	Destination Airport Name	Flight Date	Status	Edit
4	TestTest	LO3924	Rzeszów International	Warsaw Chopin Airport	Mar 9, 2021	II

Rysunek 51. Panel dyspozytora - okno dialogowe edycji lotu. Opracowanie własne.

Po naciśnięciu przycisku „Edit” otworzy się okno dialogowe z wypełnionymi danymi wybranego lotu. W tym momencie możliwa będzie edycja tych danych. Naciśnięcie w obszar poza oknem spowoduje jego zamknięcie. Jeżeli użytkownik zatwierdzi dokonanie zmian poprzez naciśnięcie przycisku „Edit”, dane zostaną zmienione a okno dialogowe zamknie się. W momencie gdy dyspozytor oznaczy lot checkboxem „Approve flight”, lot ten stanie się widoczny dla załogi, która ma go wykonać.

Pilot

View all Flights

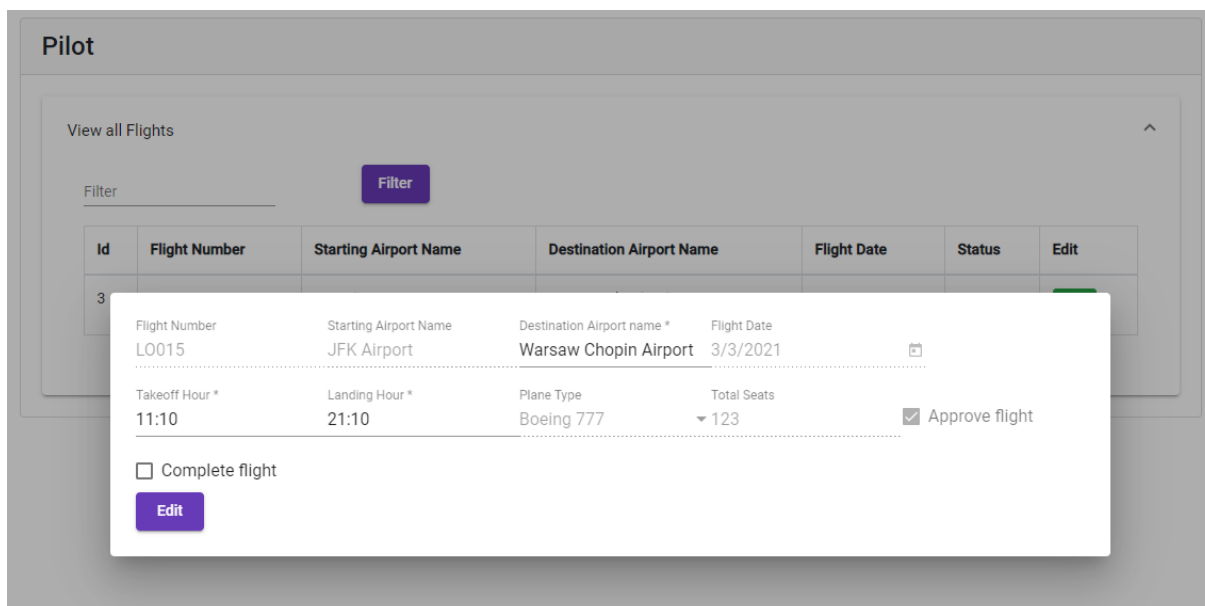
Filter

Filter

Id	Flight Number	Starting Airport Name	Destination Airport Name	Flight Date	Status	Edit
3	LO015	JFK Airport	Warsaw Chopin Airport	Mar 3, 2021	II	Edit

Rysunek 52. Panel pilota - rozwinięta zakładka "View All Flights". Opracowanie własne.

Pilot może ze swojego panelu zobaczyć wszystkie loty do których został przypisany. Ma on również ograniczoną możliwość edycji lotu. Może jedynie zmienić lotnisko docelowe, godziny wylotu i przylotu oraz zaznaczyć pole „Complete Flight”



Rysunek 53. Panel pilota - okno dialogowe edycji lotu. Opracowanie własne.

5.4. Przebieg implementacji

Implementację została rozpoczęta od stworzenia projektu na podstawie szablonu oferowane z programu Visual Studio 2019. Po utworzeniu szablonu został on dostosowany do własnych potrzeb. Pobrane zostały paczki NuGet, które uznane zostały za przydatne. W końcu zostało utworzone repozytorium na stronie GitHub w celu śledzenia postępu nad implementacją ale przede wszystkim w celu wersjonowania aplikacji. W razie popełnienia krytycznego błędu, który zniszczyłby aplikację, możliwe by było wrócenie do poprzedniej wersji. Na szczęście taka potrzeba nigdy nie zaszła. Następnie więc rozpoczęto pisanie kodu.

Na pierwszy ogień poszła próba zaimplementowania mechanizmu logowania z użyciem JWT Token. Był to pierwszy raz kiedy ten mechanizm był przeze mnie wykorzystywany, więc wymagało to rozczytania się w różnych artykułach. Mechanizm ten udało się zaimplementować. Framework Angular podczepia token w headerze do każdego zapytania kierowanego w stronę serwera ASP.NET Core, co pozwala na jego odczytanie i walidację. Implementacja tej części była na drugim miejscu pod względem czasochłonności ponieważ zależało mi na dobrym zrozumieniu wykorzystanego mechanizmu.

Kolejnym krokiem było utworzenie modelu danych wszystkich encji z projektu. Na ich podstawie utworzono tabele w bazie danych.

Następnie implementacja została podzielona na części według ról. Na tej podstawie powstały 4 moduły – moduł administratora, moduł dyspozytora, moduł pilota oraz moduł klienta. Były one po kolei implementowane właśnie w tej kolejności. Na ogół proces zaczynał się od utworzenia modułu w Angularze, wraz z komponentami oraz serwisami, które miały nawiązywać połączenie z back-endem. Następnie tworzony był kontroler odpowiadający typowi danych, których potrzebowałam. Kiedy kontroler był stworzony, trzeba było dodać odpowiedni dla niego serwis odpowiadający za logikę biznesową oraz serwis odpowiadający za pobieranie danych. Kolejnym krokiem było utworzenie requestów oraz request handlerów za pomocą biblioteki MediatR. Gdy i ten krok był ukończony, nadszedł czas na tworzenie procedur składowanych, które pobierałyby potrzebne dane z bazy. Cały ten proces był powtarzany aż do momentu w którym wszystkie moduły zostały zaimplementowane.

W całej implementacji najcięższym i najbardziej czasochłonnym zadaniem była implementacja obsługi płatności. Od początku nie było brane pod uwagę tworzenie własnego systemu płatności – byłoby to zbyt czasochłonne i technicznie wymagające. Zdecydowano się na wykorzystanie bramki płatności. Firmy które obsługują płatności na ogół udostępniają swoje SDK za darmo w celach testowych. Problemem okazał się wybór odpowiedniego rozwiązania. Jest ich dość sporo. Na pierwszy ogień poszła próba implementacji rozwiązania od firmy Stripe. Cały proces podłączania płatności do aplikacji okazał się jednak trudny i próba ta została uznana za porażkę. Ostatecznie zdecydowano się wykorzystać narzędzia od firmy PayPal. To rozwiązanie ostatecznie zadziałało i zostało w aplikacji.

Dodatkową przeszkodą okazało się implementowanie jakiegokolwiek funkcji związanej z datami. Przez długą część tworzenia programu nie został zauważony fakt, że daty wpisywane w miejscach typu ‘tworzenie lotu przez dyspozytora’ po prostu nie działały. W celu utworzenia obustronnego powiązywania danych w interfejsach tworzonych przez Angular, należy użyć tak zwanego „Banana w pudełku” czyli zapisu [(NgModel)]. Ja natomiast użyłam zapisu [value], który spowodował, że data była wyświetlana ale nienadpisywana.

Na sam koniec cały program został przejrany ponownie i doszlifowany pod względem wyglądu interfejsu na tyle na ile było to możliwe. Ten proces doprowadził do tego, że o wiele bardziej doceniam pracę osób, które zajmują się głównie projektowaniem wyglądu stron. Język CSS jest nieprzewidywalny i praca z nim była męcząca.

6. Testy systemu

W ramach sprawdzenia poprawności działania systemu obsługi linii lotniczej utworzone zostały różne rodzaje testów. Testy jednostkowe – mające na celu weryfikację działania pojedynczych jednostek tworzonego oprogramowania. Testy funkcjonalne – mające na celu weryfikację działania systemu z perspektywy użytkowników, oraz testy integracyjne – których celem jest weryfikacja działania komponentów systemu. Testy jednostkowe mogą być wykonywane wiele razy – na ogół są wykonywane podczas każdego deploymentu nowej wersji kodu. Pozwala to na wykrycie regresji – uszkodzenie funkcjonalności, które kiedyś działały, poprzez wprowadzenie nowych zmian.

6.1. Testy jednostkowe

Testy jednostkowe mogą być wykonywane wiele razy – na ogół są wykonywane podczas każdego deploymentu nowej wersji kodu. Pozwala to na wykrycie regresji – uszkodzenie funkcjonalności, które kiedyś działały, poprzez wprowadzenie nowych zmian.

W celu przeprowadzenia testów stworzony został nowy projekt typu MSTest Test Project w Visual Studio 2019. Wewnątrz tego projektu zostały napisane testy dla wybranych klas. W celu utworzenia sztucznych klas użyta została biblioteka Moq.

W tabeli poniżej umieszczone zostały opracowane testy jednostkowe dla wybranych funkcji systemu. Tabela pokazuje nazwę testu oraz jego opis.

Nazwa testu	Opis testu
ShouldCallGetUserByIdRequest	Sprawdzenie czy metoda GetUserById utworzy żądanie GetUserByIdRequest wykorzystywane przez MediatR.
ShouldCallGetAllUsersRequest	Sprawdzenie czy metoda GetAllUsers utworzy żądanie GetAllUsersRequest wykorzystywane przez MediatR.
ShouldCallCreateUserRequest	Sprawdzenie czy metoda CreateUser utworzy żądanie

	CreateUserRequest wykorzystywane przez MediatR.
ShouldCallEditUserRequest	Sprawdzenie czy metoda EditUser utworzy żądanie EditUserRequest wykorzystywane przez MediatR.
ShouldCallDeleteUserRequest	Sprawdzenie czy metoda DeleteUser utworzy żądanie DeleteUserRequest wykorzystywane przez MediatR.
ShouldNotCreateUserIfIncorrectPeselFormat	Sprawdzenie czy użytkownik nie zostanie stworzony jeżeli jego nr PESEL jest niepoprawny.
ShouldNotCreateUserIfIncorrectIdFormat	Sprawdzenie czy użytkownik nie zostanie stworzony jeżeli jego Id Dowodu osobistego jest niepoprawne.
ShouldNotCreateUserIfIncorrectPasswordFormat	Sprawdzenie czy użytkownik nie zostanie stworzony jeżeli jest Hasła nie spełnia wymogu.
ShouldNotCreateCrewIfIncorrectPilots	Sprawdzenie czy załoga nie zostanie stworzona jeżeli ten sam pilot będzie na dwóch takich samych pozycjach (np. kapitan i pierwszy oficer).

Tabela 19. Wykaz testów jednostkowych. Opracowanie własne.

Poniżej przedstawione zostaną przykładowe testy.


```

[TestMethod]
public void ShouldNotCreateUserIfIncorrectPeselFormat()
{
    var controller = new UsersController(userService.Object);
    try
    {
        controller.CreateUser(this._invalidPeselUser);
    }
    catch (HttpResponseException ex)
    {
        Assert.AreEqual(ex.Response.StatusCode, HttpStatusCode.BadRequest, "Invalid PESEL.");
    }
}

```

Rysunek 54. Test jednostkowy "ShouldNotCreateUserIfIncorrectPeselFormat". Opracowanie własne.

```

[TestMethod]
public void ShouldCallGetUserByIdRequest()
{
    var service = new UserService(_appSettings.Object, mediator.Object);

    this.mediator
        .Setup(m => m.Send(It.IsAny<GetUserByIdRequest>(), It.IsAny<CancellationTokens>()))
        .Returns(Task.FromResult(this.userReturn));

    var result = service.GetUserById(1);
    result.Equals(this._correctUser);
    this.mediator.Verify(m => m.Send(It.IsAny<GetUserByIdRequest>(), It.IsAny<CancellationTokens>()), Times.Once());
}

```

Rysunek 55. Test jednostkowy "ShouldCallGetUserByIdRequest". Opracowanie własne.

Poniżej został zamieszczony wygenerowany przez Visual Studio 2019 wykaz wyników wszystkich testów jednostkowych.

Test	Duration	Traits	Error Message
▲ ✓ AirlineServiceSoftware.Tests (9)	116 ms		
▲ ✓ AirlineServiceSoftware.Tests (9)	116 ms		
▲ ✓ AirlineServiceSoftwareTests (9)	116 ms		
✓ ShouldCallCreateUserRequest	101 ms		
✓ ShouldCallDeleteUserRequest	1 ms		
✓ ShouldCallEditUserRequest	1 ms		
✓ ShouldCallGetAllUsersRequest	1 ms		
✓ ShouldCallGetUserByIdRequest	1 ms		
✓ ShouldNotCreateCrewIfIncorrec...	6 ms		
✓ ShouldNotCreateUserIfIncorrec...	5 ms		
✓ ShouldNotCreateUserIfIncorrec...	< 1 ms		
✓ ShouldNotCreateUserIfIncorrec...	< 1 ms		

Rysunek 56. Wykaz testów jednostkowych. Opracowanie własne.

6.2. Testy bezpieczeństwa

W celu spełnienia wymagań dotyczących bezpieczeństwa, zdecydowano się na opracowanie i przeprowadzenie testów bezpieczeństwa. Celem tych testów, jest weryfikacja poprawności wprowadzonych zabezpieczeń.

W aplikacji istnieje mechanizm autoryzacji użytkowników na podstawie ich roli. Dostęp do interfejsów oraz API opiera się właśnie na podstawie roli użytkownika (a także na tym, czy użytkownik jest w ogóle zalogowany). Dodatkowo należy przeprowadzić testy walidacji danych wprowadzanych przez użytkowników systemu. Nie powinni oni mieć możliwości wprowadzania niepoprawnych danych.

Poniżej zostaną przedstawione scenariusze testów, ich opis oraz wynik.

1. Test 1. - Walidacja danych w formularzu.

Opis – Administrator podczas tworzenia nowego użytkownika zapomniał wpisać jego hasło.

Wynik – Walidacja danych przebiegła poprawnie. Administrator otrzymał komunikat o treści „A Password is required.”

2. Test 2. - Walidacja formatu hasła.

Opis – Użytkownik podczas zmiany wpisał niewystarczająco silne hasło, w którym brakuje cyfr oraz znaku specjalnego.

Wynik – Walidacja danych przebiegła poprawnie. Pojawiają się dwa komunikaty – pierwszy „Password must be at least 8 characters long, have one uppercase letter, one lowercase, one numer and one special character.” Drugi, „Data could not be edited.”. Hasło użytkownika nie zostało zmienione.

3. Test 3 - Walidacja poprawności numeru PESEL.

Opis – Użytkownik podczas zmiany swojego numeru PESEL wpisał niepoprawny numer.

Wynik – Walidacja danych przebiegła poprawnie. Użytkownik otrzymał komunikat o treści „Data could not be edited.”. PESEL użytkownika nie został zmieniony.

4. Test 4 – Walidacja poprawności numeru Dokumentu Osobistego.

Opis – Użytkownik podczas zmiany swojego Dokumentu Osobistego wpisał niepoprawny numer.

Wynik – Walidacja danych przebiegła poprawnie. Użytkownik otrzymał komunikat o treści „Data could not be edited.”. Numer Dokumentu Osobistego użytkownika nie został zmieniony.

5. Test 5 – Błędne logowanie.

Opis – Użytkownik logujący się na stronę wpisał niepoprawne hasło.

Wynik – Walidacja danych przebiegła poprawnie. Użytkownik otrzymał komunikat o treści „Username or Password is incorrect.”

6. Test 6 – Brak dostępu do zasobów przez niezalogowanego użytkownika.

Opis – niezalogowany użytkownik w oknie przeglądarki wpisuje adres prowadzący do interfejsu administratora w celu uzyskania dostępu do zasobów.

Wynik – Walidacja danych przebiegła poprawnie. Niezalogowany użytkownik został przekierowany na stronę logowania.

7. Test 7 – Brak dostępu do zasobów przez użytkownika bez uprawnień.

Opis – zalogowany użytkownik wpisuje w oknie przeglądarki adres prowadzący do interfejsu administratora w celu uzyskania dostępu do zasobów.

Wynik – Walidacja danych przebiegła poprawnie. Zalogowany użytkownik został przekierowany na stronę główną.

6.3. Testy zgodności.

Interfejsem graficznym systemu jest przeglądarka, która działa po stronie klienta. Z tego powodu ważne jest zapewnienie, czy aplikacja działa poprawnie na różnych przeglądarkach.

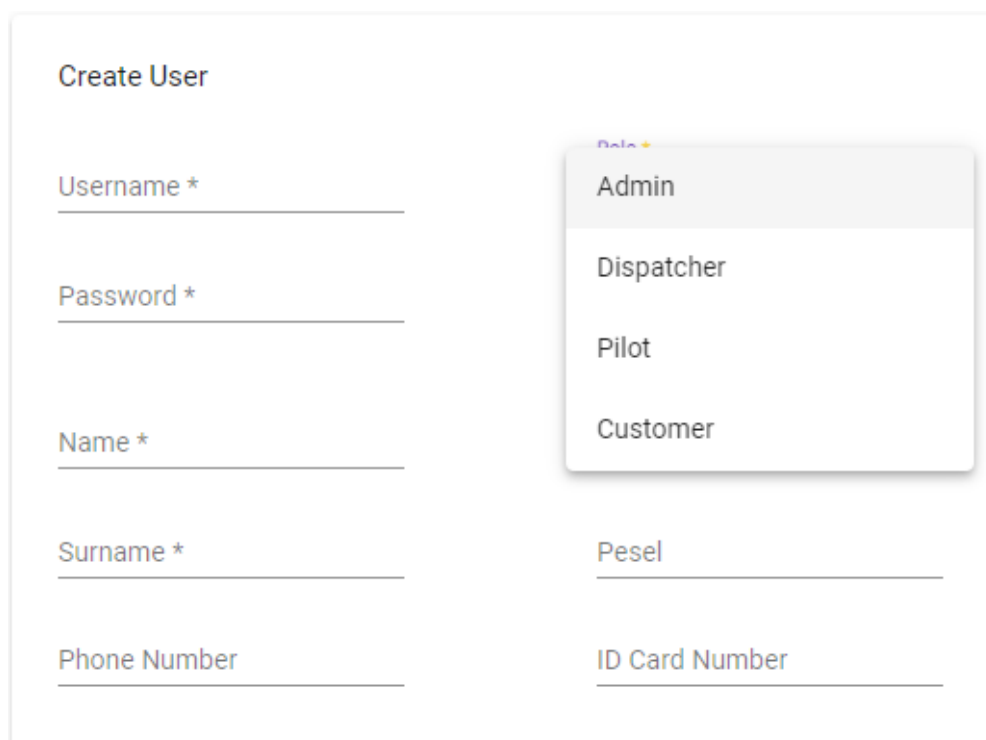
W wymaganiach poza funkcjonalnych wyznaczone zostało poprawne działanie systemu w przeglądarkach Opera, Chrome, Firefox oraz Edge.

Do przeprowadzenia testów wybrane zostały poniższe elementy interfejsu:

- Listy rozwijane.
- Pola edycji daty.
- Pola edycji godziny.
- Pola numeryczne.

Test 1. Sprawdzenie poprawności działania list rozwijanych.

Rezultat dla Opery:



Create User

Username *

Password *

Name *

Surname *

Phone Number

Admin

Dispatcher

Pilot

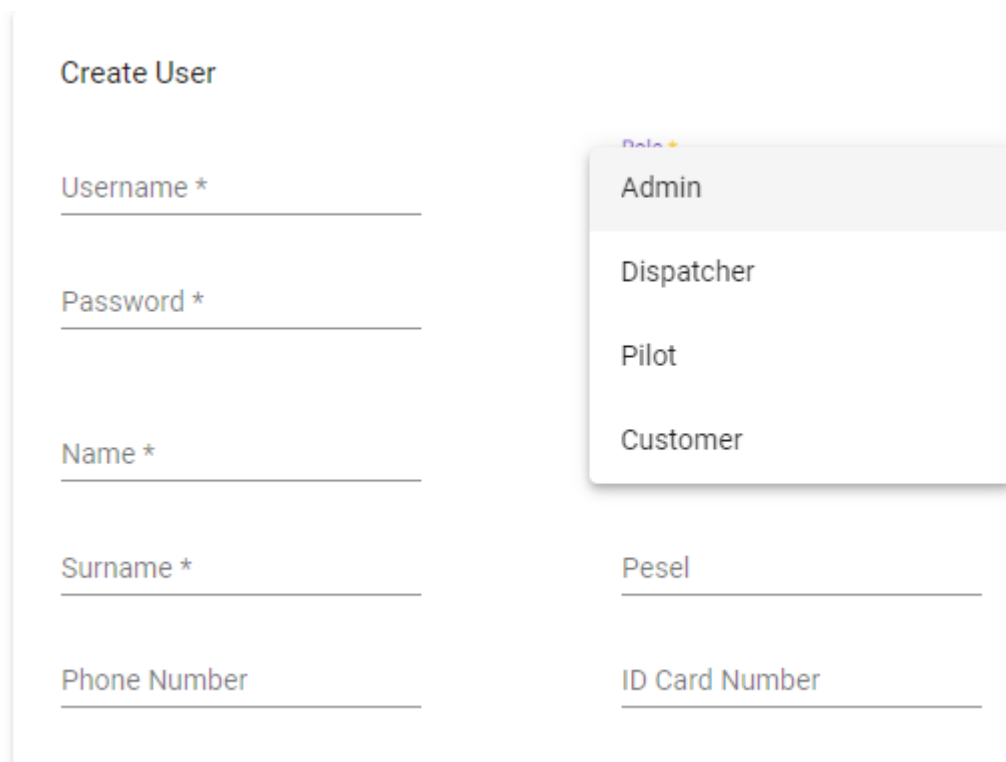
Customer

Pesel

ID Card Number

Rysunek 57. Test listy rozwijanej dla Opery. Opracowanie własne.

Rezultat dla Chrome:



Create User

Username *

Password *

Name *

Surname *

Phone Number

Admin

Dispatcher

Pilot

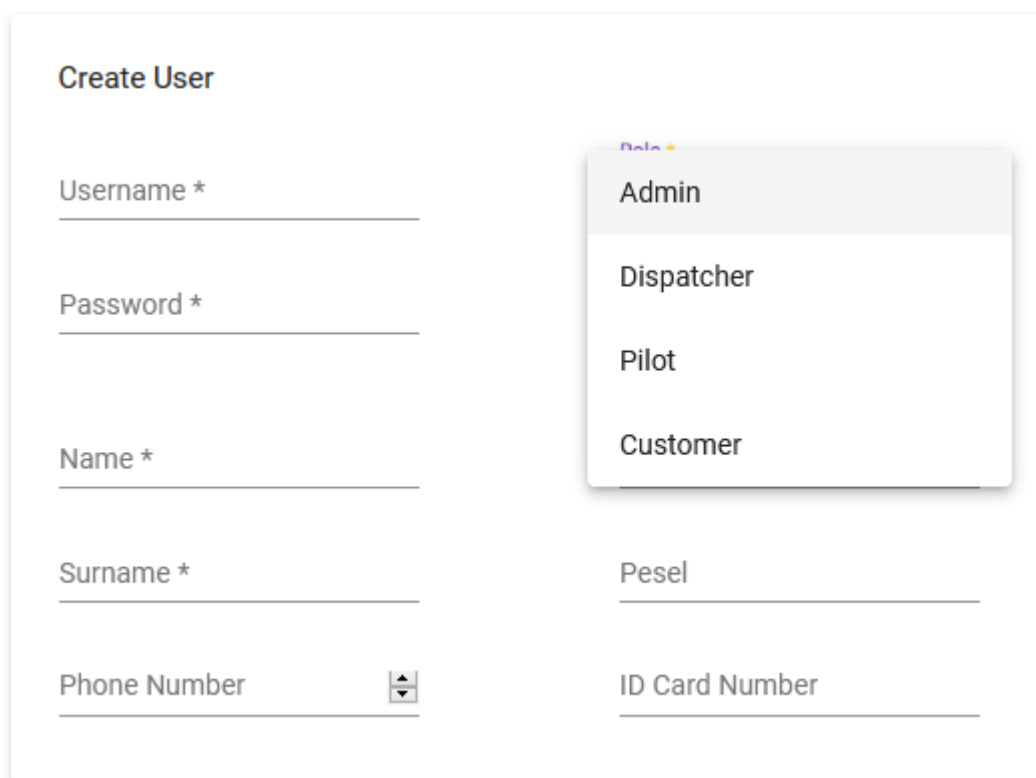
Customer

Pesel

ID Card Number

Rysunek 58. Testy listy rozwijanej dla Chrome. Opracowanie własne.

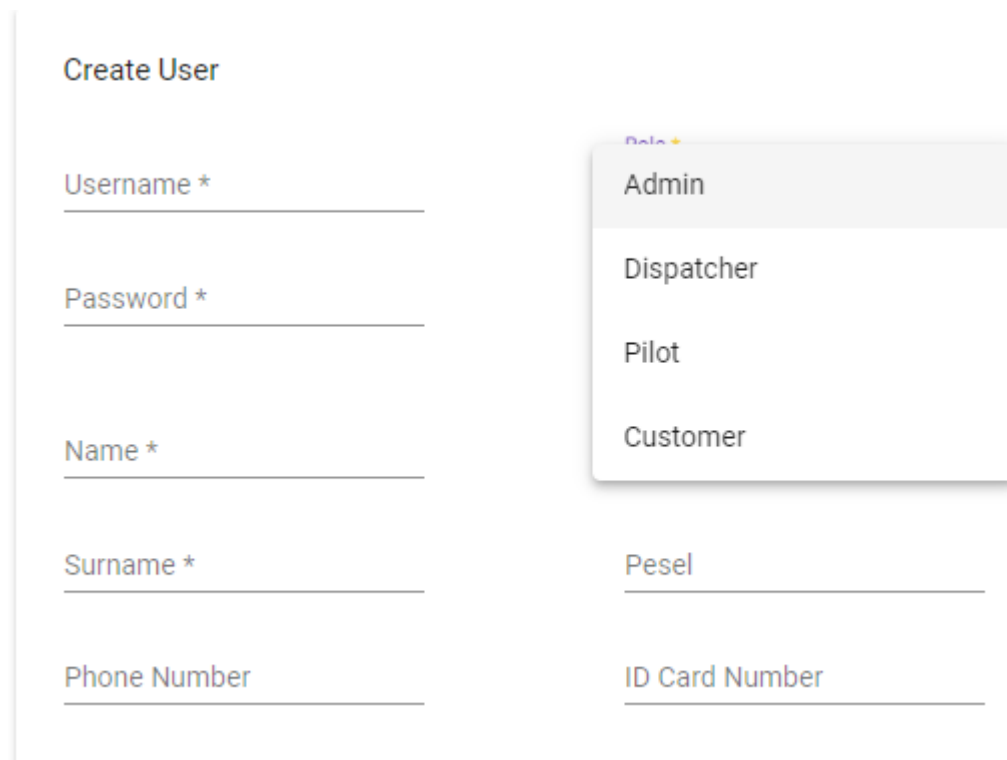
Rezultat dla Firefox:



The screenshot shows a web form titled "Create User" in a Firefox browser. The form contains several input fields: "Username *" (required), "Password *" (required), "Name *" (required), "Surname *" (required), "Phone Number" (with a small numeric keypad icon), "Pesel", and "ID Card Number". A dropdown menu is open next to the "Role" label, displaying four options: "Admin" (highlighted with a grey background), "Dispatcher", "Pilot", and "Customer". The browser's address bar shows a URL starting with "http://".

Rysunek 59. Test listy rozwijanej dla Firefox. Opracowanie własne.

Rezultat dla Edge:



The screenshot shows the same "Create User" form in an Edge browser. The layout and fields are identical to the Firefox version. The dropdown menu for the "Role" field is open, showing the same four options: "Admin" (highlighted), "Dispatcher", "Pilot", and "Customer". The browser's address bar shows a URL starting with "http://".

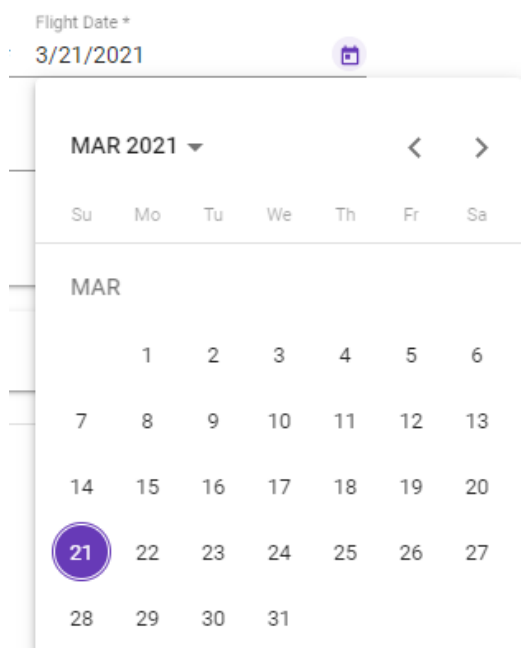
Rysunek 60. Test listy rozwijanej dla Edge. Opracowanie własne.

Wynik testu:

- Kontrolki wyglądają identycznie w Operze, Chrome oraz Edge.
- Kontrolki działają poprawnie na wszystkich środowiskach.

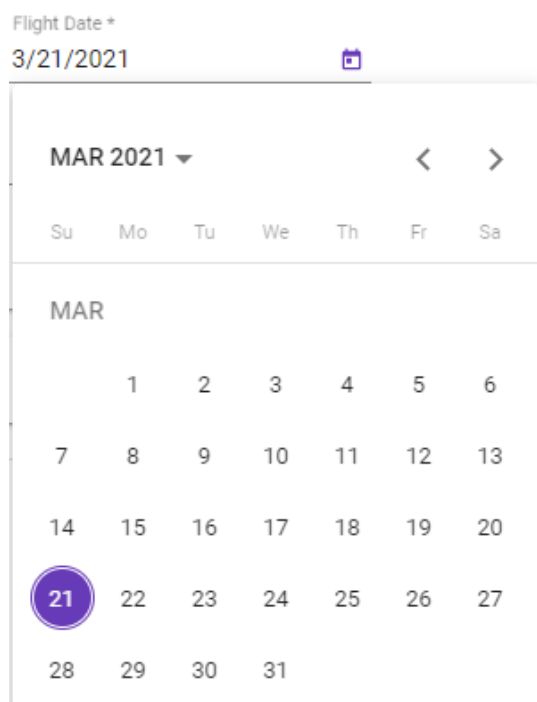
Test 2. Sprawdzenie poprawności działania pól edycji dat.

Rezultat dla Opery:



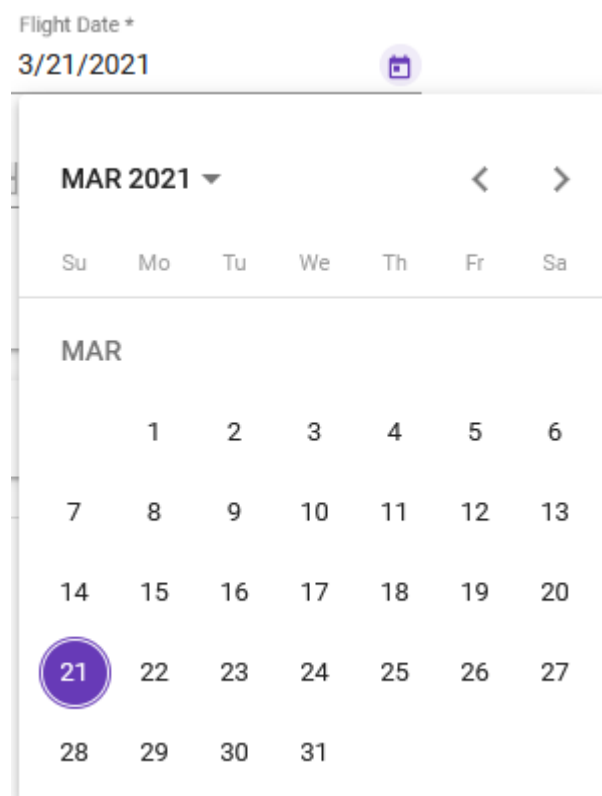
Rysunek 61. Test pola edycji daty dla Opery. Opracowanie własne.

Rezultat dla Chrome:



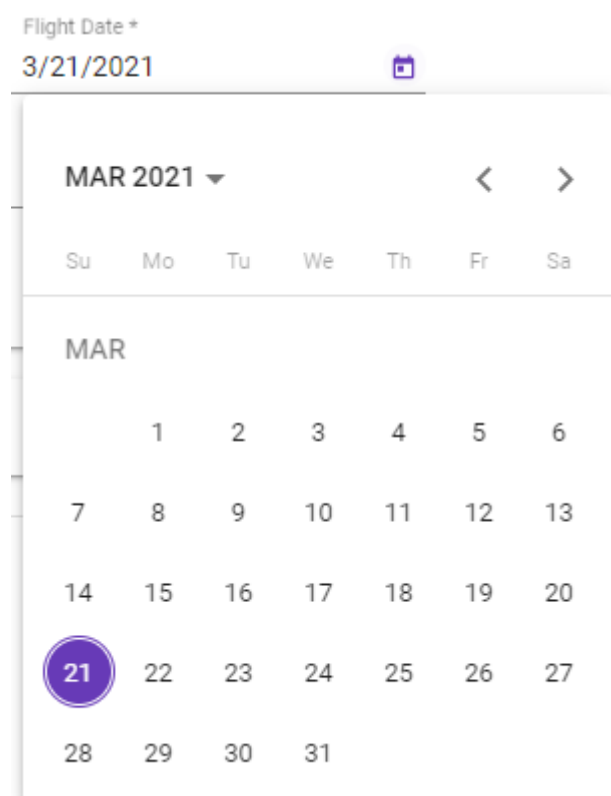
Rysunek 62. Test pola edycji daty dla Chrome. Opracowanie własne.

Rezultat dla Firefox:



Rysunek 63. Test pola edycji daty dla Firefox. Opracowanie własne.

Rezultat dla Edge:



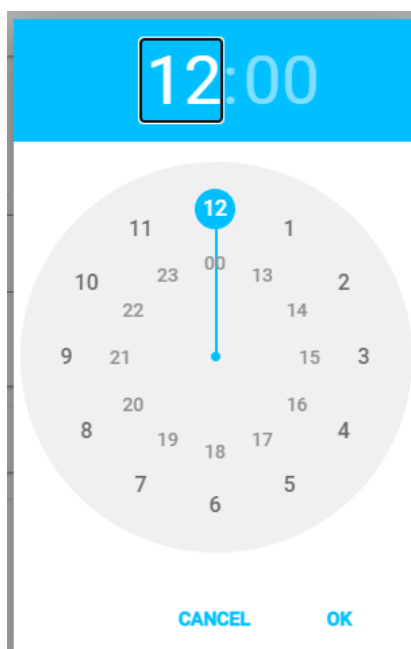
Rysunek 64. Test pola edycji daty dla Edge. Opracowanie własne.

Wynik testu:

- Wygląd kontrolki jest identyczny dla wszystkich środowisk.
- Kontrolki działają poprawnie na wszystkich środowiskach.

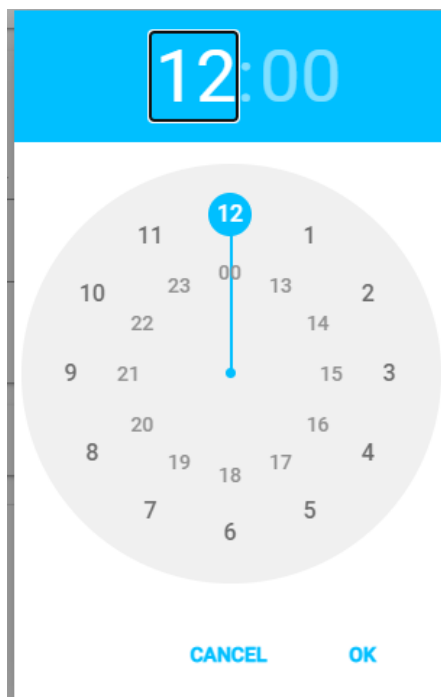
Test 3. Sprawdzenie poprawności działania pól edycji godziny.

Rezultat dla Operry:



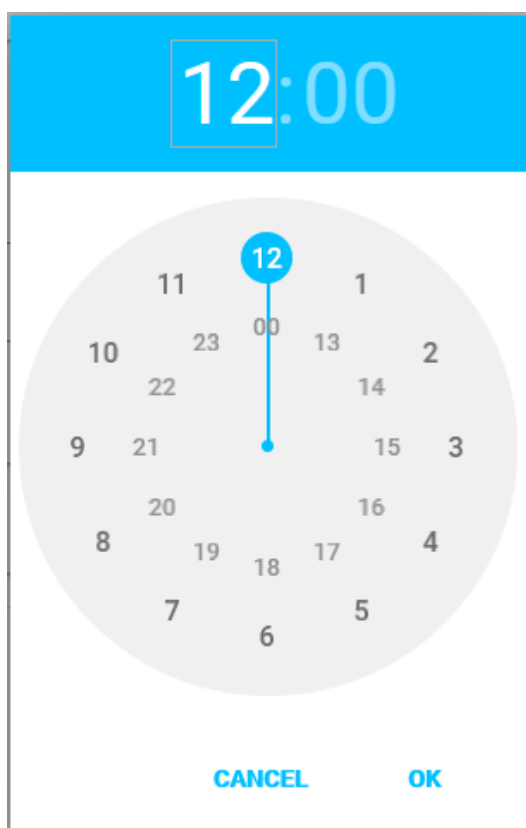
Rysunek 65. Test pola edycji godziny dla Operry. Opracowanie własne.

Rezultat dla Chrome:



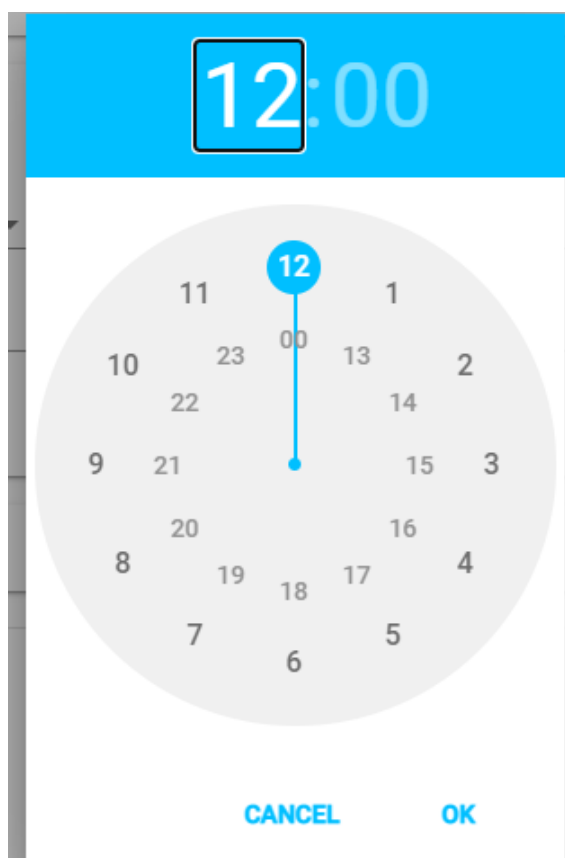
Rysunek 66. Test pola edycji godziny dla Chrome. Opracowanie własne.

Rezultat dla Firefox:



Rysunek 67. Test pola edycji godziny dla Firefox. Opracowanie własne.

Rezultat dla Edge:



Rysunek 68. Test pola edycji godziny dla Edge. Opracowanie własne.

Wynik testu:

- Kontrolki wyglądają identycznie na wszystkich środowiskach.
- Kontrolki działają poprawnie na wszystkich środowiskach.

Test 4. Sprawdzenie poprawności działania pól numerycznych.

Rezultat dla Opery:



Rysunek 69. Test pola numerycznego dla Opery. Opracowanie własne.

Rezultat dla Chrome:



Rysunek 70. Test pola numerycznego dla Chrome. Opracowanie własne.

Rezultat dla Firefox:



Rysunek 71. Test pola numerycznego dla Firefox. Opracowanie własne.

Rezultat dla Edge:



Rysunek 72. Test pola numerycznego dla Edge. Opracowanie własne.

Wynik testu:

- Kontrolki mają różny wygląd przycisków zmieniających wartość pola
- Kontrolki działają poprawnie na wszystkich środowiskach.

7. Podsumowanie

Celem niniejszej pracy było opisanie i stworzenie aplikacji internetowej wspomagającej obsługę linii lotniczej, pomagającej nie tylko pracownikom linii lotniczej ale także jej potencjalnym klientom. Pierwszym krokiem w osiągnięciu tego celu było przeanalizowanie dostępnych już na rynku rozwiązań. Na ich podstawie określone zostały najważniejsze cechy takiego systemu w formie wymagań funkcjonalnych oraz pozafunkcyjnych. W implementacji zostały wykorzystane framework-i ASP.NET Core oraz Angular ze względu na ich nowoczesne rozwiązania, świetną współpracę ze sobą a także ze względu na możliwość utworzenia przenośnego i uniwersalnego systemu. Poprzez użycie biblioteki MediatR oraz architektury N-warstwowej, system ma bardzo łatwo zrozumiałą i logiczną konstrukcję, która pozwala na jego łatwe rozbudowanie.

Oczywiście każdy system można ulepszać i rozwijać – ten system nie jest wyjątkiem. Dobrym rozwiązaniem byłoby dodanie modułu dla mechaników. Ze względu na wymóg utrzymania samolotów w jak najlepszej kondycji, wiele linii lotniczych zatrudnia swoich własnych mechaników, którzy serwisują samoloty. Każda część samolotu ma pewien „termin przydatności”, po którym musi zostać dokładnie sprawdzona pod kątem jakichkolwiek uszkodzeń i w razie czego wymieniona. Moduł taki mógłby pozwolić na utworzenie obecnych w załodze samolotów a następnie przypomnienie mechanikom o nadchodzących wymaganych przeglądach. Dodatkową funkcjonalnością mogło by być sprawdzanie stanu magazynu. Dzięki temu mechanicy wiedzieliby czy mają narzędzia i części których potrzebują. Innym ulepszeniem mogło by być dodanie funkcji grafiku dla pilotów. Mógłby on w graficzny sposób przedstawić ich rozkład lotów i lepiej zwizualizować ich pracę na np. przyszły tydzień.

WYKAZ LITERATURY

Źródła literackie

1. K. Sehl, Aftershocks, „APEX Experience”, 2020, 10.3, s. 38 – 43
2. K. Sacha, Inżynieria oprogramowania, Warszawa 2010, Wydawnictwo Naukowe PWN SA.
3. G. Reese, Database Programming with JDBC & Java: Developing Multi-Tier Applications, Sebastopol 2000, O'Reilly Media.
4. J. Rogulski, Wspomaganie procesów zarządzania działaniami w straży pożarnej, Józefów 2016, Wydawnictwo CNBOB-PIB.
5. Jeremy Wilken, Angular w akcji, Gliwice 2019, Wydawnictwo HELION SA.

Źródła pozaliterackie

- 1p. https://www.ulc.gov.pl/download/regulacja_ryнку/statystyki/2019/wg_porto_w_lotniczych_4kw2019.pdf
- 2p. https://www.ulc.gov.pl/download/regulacja_ryнку/statystyki/2019/wg_przew_regularne_4kw2019.pdf
- 3p. <https://www.phocuswire.com/PROS-research>
- 4p. <https://amadeus.com/en/insights/blog/deep-dive-airlines-personalization>
- 5p. <http://pracenaukowe.wwszip.pl/prace/prace-naukowe-43.pdf>
- 6p. <https://skift.com/2017/12/07/airlines-rebel-against-amadeus-fees-but-investors-arent-worried/>
- 7p. https://k.bartecki.po.opole.pl/io/io_wyklad4.pdf
- 8p. <https://getbootstrap.com/docs/4.0/components/alerts/>
- 9p. <https://www.npmjs.com/package/ngx-material-timepicker>
- 10p. <https://material.angular.io>
- 11p. <https://getbootstrap.com>
- 12p. <https://jquery.com>

Spis ilustracji

Rysunek 1. Zrzut ekranu pokazujący stronę wyszukiwania rezerwacji systemu Altea Reservation	9
Rysunek 2. Zrzut ekranu pokazujący widok rozkładu lotów	11
Rysunek 3. Zrzut ekranu przedstawiający rejestr listy płac załóg.....	13
Rysunek 4. Aktorzy systemu obsługi linii lotniczej. Opracowanie własne.....	18
Rysunek 5. Diagram hierarchii funkcji poziomów 1 - 2 najważniejszych funkcji systemu. Opracowanie własne.....	22
Rysunek 6. Diagram przypadków użycia. Opracowanie własne.....	24
Rysunek 7. Diagram encji. Opracowanie własne.	25
Rysunek 8. Diagram architektury systemu. Opracowanie własne.	31
Rysunek 9. Diagram sekwencji - logowanie użytkownika. Opracowanie własne	32
Rysunek 10. Diagram czynności - logowanie. Opracowanie własne.....	33
Rysunek 11. Diagram sekwencji - utworzenie załogi przez dyspozytora. Opracowanie własne.	33
Rysunek 12. Diagram czynności - utworzenie załogi przez dyspozytora. Opracowanie własne.	34
Rysunek 13. Diagram sekwencji - edytowanie lotu przez dyspozytora. Opracowanie własne.	35
Rysunek 14. Diagram czynności - edytowanie lotu przez dyspozytora. Opracowanie własne.....	36
Rysunek 15. Diagram klas części projektowanego systemu - część odpowiadająca za zarządzanie użytkownikami. Opracowanie własne.	37
Rysunek 16. Projekt diagramu relacji bazy danych. Opracowanie własne.	55
Rysunek 17. Układ interfejsu graficznego stron. Opracowanie własne.	60
Rysunek 18. Makieta strony logowania. Opracowanie własne.	62
Rysunek 19. Makieta strony głównej. Opracowanie własne.....	63
Rysunek 20. Makieta panelu użytkownika. Opracowanie własne.	64
Rysunek 21. Makieta strony zakupu biletu. Opracowanie własne.	65
Rysunek 22. Makieta okna dialogowego finalizacji płatności. Opracowanie własne.....	66
Rysunek 23. Makieta panelu dyspozytora. Opracowanie własne.	67
Rysunek 24. Makieta okna dialogowego edytowania załogi. Opracowanie własne.	68
Rysunek 25. Makieta okna dialogowego edytowania lotu. Opracowanie własne.	69
Rysunek 26. Makieta okna dialogowego podglądu rezerwacji. Opracowanie własne.....	69
Rysunek 27. Lista wybranych komunikatów z biblioteki Bootstrap 4. [6p].	70
Rysunek 28. Skrypt SQL tworzący tabelę "Users". Opracowanie własne.	71
Rysunek 29. Skrypt SQL tworzący tabelę "Crews".Opracowanie własne.....	72
Rysunek 30. Skrypt SQL tworzący tabelę "Flights". Opracowanie własne.	72
Rysunek 31. Skrypt SQL tworzący tabelę "Reservations". Opracowanie własne.	72
Rysunek 32. Skrypt SQL tworzący procedurę składowaną "CreateCrew". Opracowanie własne.	73
Rysunek 33. Skrypt SQL tworzący procedurę składowaną "GetFlightsByPilotId". Opracowanie własne.....	73
Rysunek 34. Skrypt SQL tworzący procedurę składowaną "GetUserById". Opracowanie własne.	74
Rysunek 35. Zawartość pliku "appsettings.json". Opracowanie własne.	74
Rysunek 36. Część pliku Startup.cs ukazująca wstrzykiwanie connection string do serwisów. Opracowanie własne.....	75
Rysunek 37. Tabela "Users". Opracowanie własne.	75
Rysunek 38. Tabela "Reservations". Opracowanie własne.	75
Rysunek 39. Tabela "Flights". Opracowanie własne.	76
Rysunek 40. Tabela "Crews". Opracowanie własne.	76
Rysunek 41. Zrzut ekranu klasy UsersController. Opracowanie własne.	81

Rysunek 42. Zrzuty ekranu dla klasy ReservationDataService. Opracowanie własne.	83
Rysunek 43. Zrzuty ekranu dla klasy CrewService. Opracowanie własne.	85
Rysunek 44. Zrzut ekranu dla klasy CreateFlightRequest. Opracowanie własne.	86
Rysunek 45. Zrzut ekranu dla klasy CreateFlightRequestHandler. Opracowanie własne.	87
Rysunek 46. Zrzuty ekranu komponentu Checkout. Opracowanie własne.	89
Rysunek 47. Strona logowania. Opracowanie własne.	90
Rysunek 48. Zwinięty panel dyspozytora. Opracowanie własne.	90
Rysunek 49. Panel dyspozytora - rozwinięta zakładka "Create Flight". Opracowanie własne.	91
Rysunek 50. Panel dyspozytora - rozwinięta zakładka "View All Flights". Opracowanie własne.	91
Rysunek 51. Panel dyspozytora - okno dialogowe edycji lotu. Opracowanie własne.	92
Rysunek 52. Panel pilota - rozwinięta zakładka "View All Flights". Opracowanie własne.	92
Rysunek 53. Panel pilota - okno dialogowe edycji lotu. Opracowanie własne.	93
Rysunek 54. Test jednostkowy "ShouldNotCreateUserIfIncorrectPeselFormat". Opracowanie własne.	97
Rysunek 55. Test jednostkowy "ShouldCallGetUserByIdRequest". Opracowanie własne.	97
Rysunek 56. Wykaz testów jednostkowych. Opracowanie własne.	97
Rysunek 57. Test listy rozwijanej dla Opery. Opracowanie własne.	100
Rysunek 58. Testy listy rozwijanej dla Chrome. Opracowanie własne.	100
Rysunek 59. Test listy rozwijanej dla Firefox. Opracowanie własne.	101
Rysunek 60. Test listy rozwijanej dla Edge. Opracowanie własne.	101
Rysunek 61. Test pola edycji daty dla Opery. Opracowanie własne.	102
Rysunek 62. Test pola edycji daty dla Chrome. Opracowanie własne.	102
Rysunek 63. Test pola edycji daty dla Firefox. Opracowanie własne.	103
Rysunek 64. Test pola edycji daty dla Edge. Opracowanie własne.	103
Rysunek 65. Test pola edycji godziny dla Opery. Opracowanie własne.	104
Rysunek 66. Test pola edycji godziny dla Chrome. Opracowanie własne.	104
Rysunek 67. Test pola edycji godziny dla Firefox. Opracowanie własne.	105
Rysunek 68. Test pola edycji godziny dla Edge. Opracowanie własne.	105
Rysunek 69. Test pola numerycznego dla Opery. Opracowanie własne.	106
Rysunek 70. Test pola numerycznego dla Chrome. Opracowanie własne.	106
Rysunek 71. Test pola numerycznego dla Firefox. Opracowanie własne.	106
Rysunek 72. Test pola numerycznego dla Edge. Opracowanie własne.	106

Spis tabel

Tabela 1. Wykaz użytych w tekście skrótów.	4
Tabela 2 Podsumowanie funkcji oraz cen trzech wybranych systemów. Opracowanie własne.	15
Tabela 3. Charakterystyka aktorów. Opracowanie własne.....	18
Tabela 4. Opis encji. Opracowanie własne.....	26
Tabela 5. Opis relacji. Opracowanie własne.	26
Tabela 6. Lista wymagań pozafunkcjonalnych systemu. Opracowanie własne.....	29
Tabela 7. Opis klas projektowanego systemu. Opracowanie własne.	52
Tabela 8. Opis metod w klasie UserService. Opracowanie własne.....	53
Tabela 9. Opis klasy ReservationDataService. Opracowanie własne.	54
Tabela 10. Wykaz tabel projektowanej bazy danych. Opracowanie własne.....	56
Tabela 11. Opis tabeli bazy danych Users. Opracowanie własne.	57
Tabela 12. Opis tabeli bazy danych Crews. Opracowanie własne.	58
Tabela 13. Opis tabeli bazy danych Flights. Opracowanie własne.	59
Tabela 14. Opis tabeli bazy danych Reservations. Opracowanie własne.....	59
Tabela 15. Implementacja klas systemu. Opracowanie własne.....	78
Tabela 16. Opis klas pominiętych w projekcie systemu. Opracowanie własne.	79
Tabela 17. Zaimplementowane metody klasy UsersController. Opracowanie własne.	79
Tabela 18. Zaimplementowane metody klasy ReservationDataService. Opracowanie własne.	80
Tabela 19. Wykaz testów jednostkowych. Opracowanie własne.	96

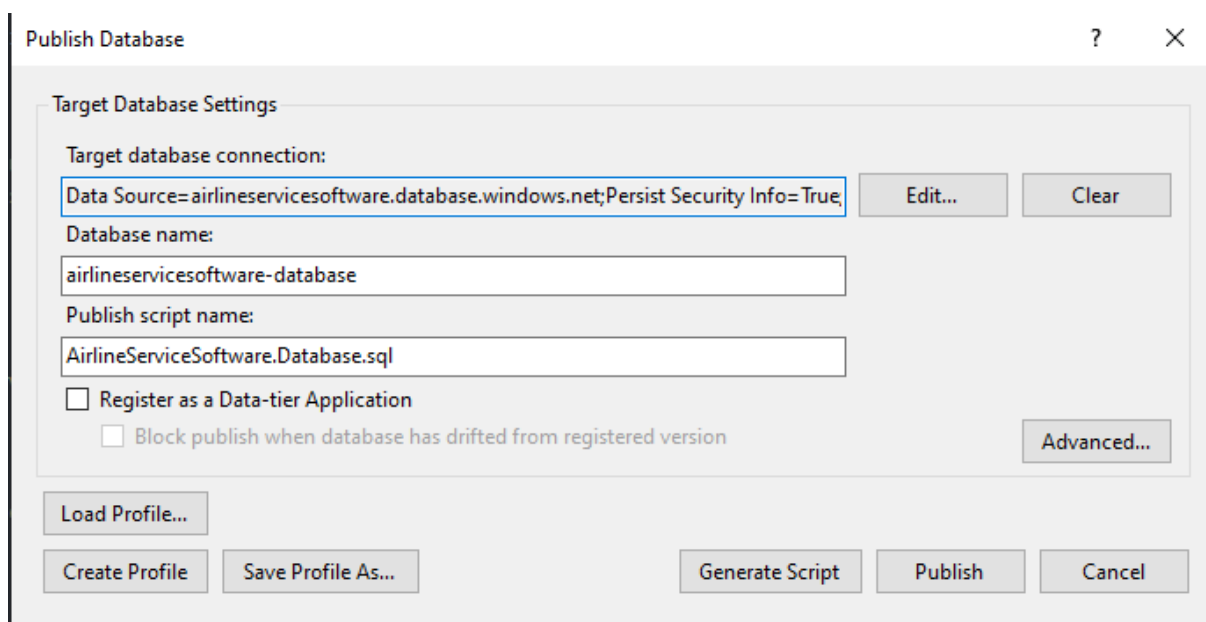
ZAŁĄCZNIKI

Instrukcja instalacji systemu obsługi linii lotniczej.

System obsługi linii lotniczej jest aplikacją web-ową, która wymaga jednorazowej instalacji i konfiguracji na serwerze hostingowym. Dodatkowo należy także zapewnić serwer bazy danych MS SQL Server.

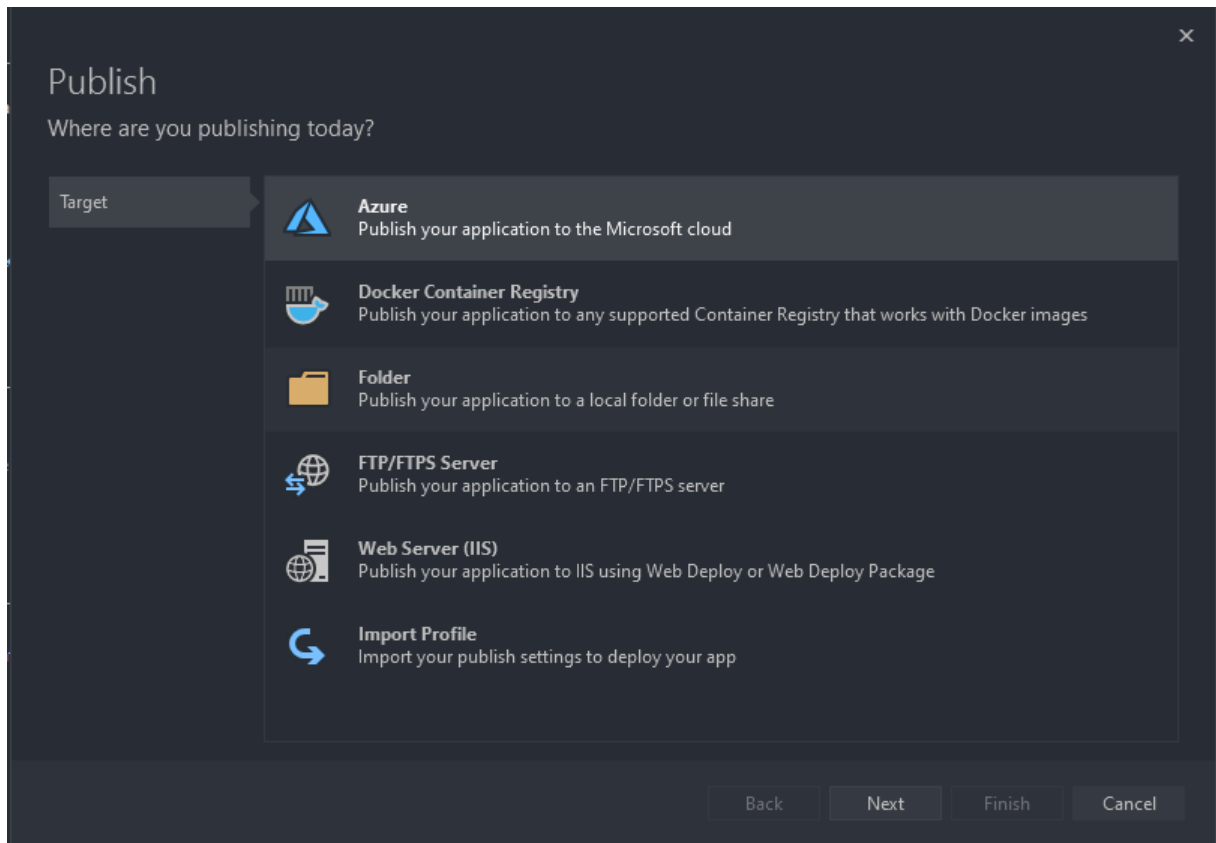
W celu instalacji systemu obsługi linii lotniczej należy:

1. Utworzyć nową bazę danych.
2. Zapisać tymczasowo adres serwera, nazwę użytkownika oraz hasło do połączenia z bazą danych.
3. Uruchomić plik o nazwie „AirlineServiceSoftware.sln” w programie Visual Studio 2019. Jest to plik solucji aplikacji, który zawiera projekt aplikacji a także projekt bazy danych.
4. Najeżdżamy myszką na projekt „AirlineServiceSoftware.Database” i prawym przyciskiem myszy wywołujemy menu opcji. Następnie wybieramy opcję „Publish”.

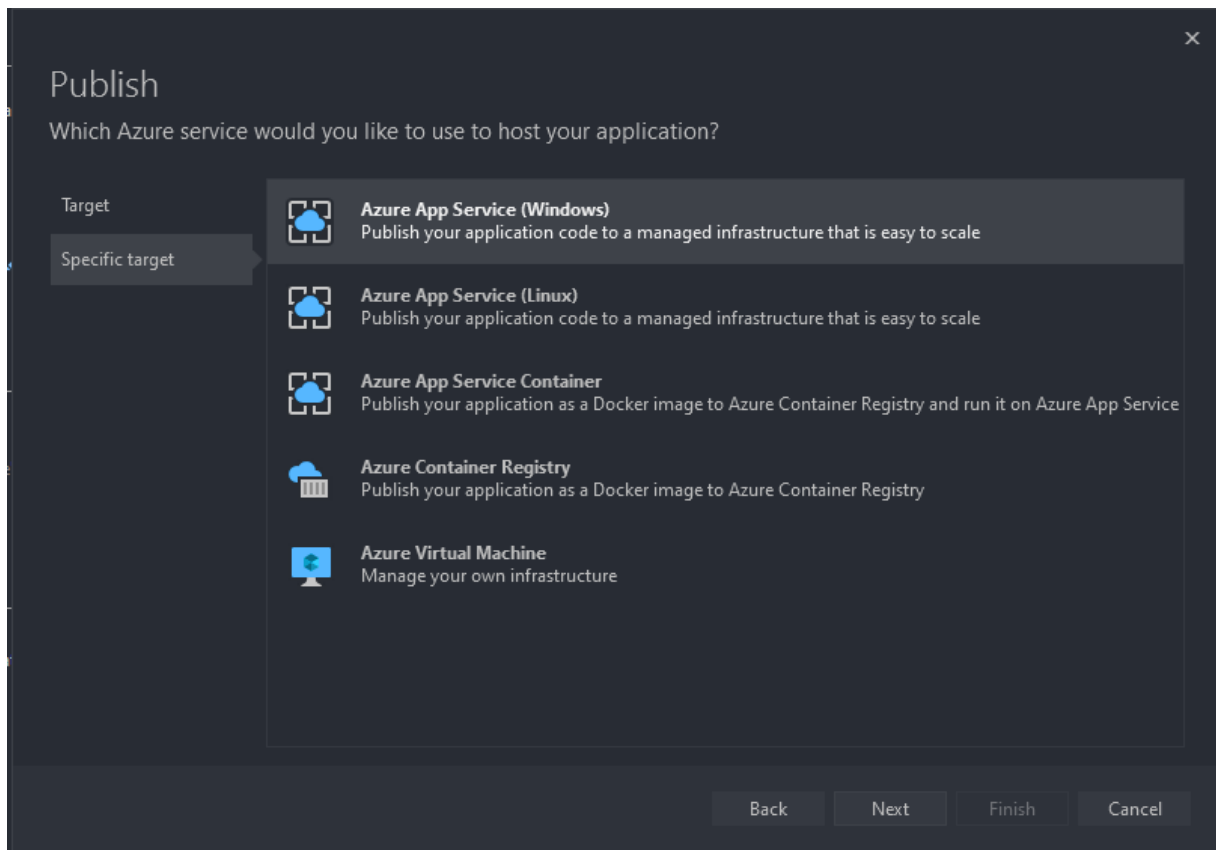


5. Pojawia się okienko „Publish Database”. Należy wypełnić dane bazy danych po czym nacisnąć przycisk „Publish”. Tabele oraz procedury składowane zostaną wtedy opublikowane do bazy danych.
6. Włączamy procedurę składowaną o nazwie „CreateFirstAdmin”. Stworzy to pierwsze konto administratora o loginie „admin” i hasło „admin”. Należy zmienić jego dane jak najprędzej.

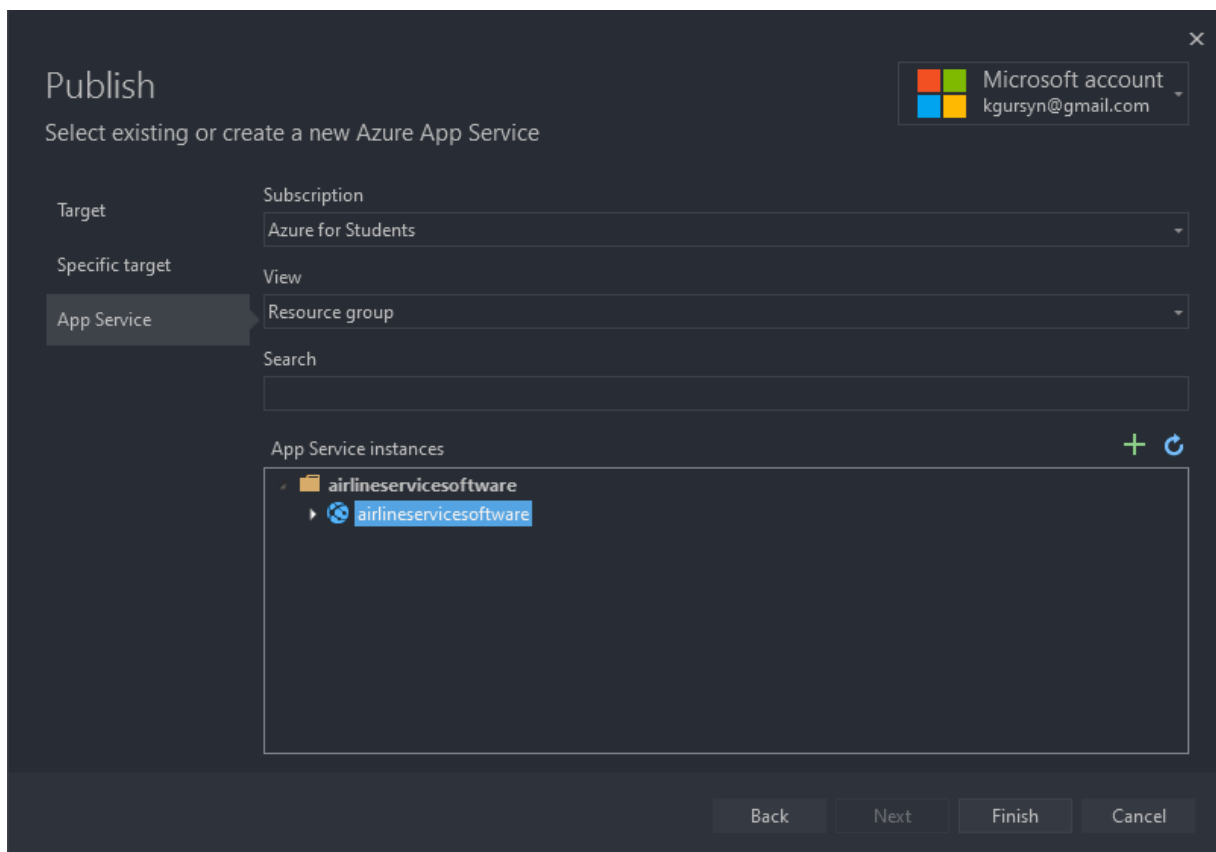
7. Kiedy baza danych zostanie już utworzona, w pliku „appsettings.json” należy podmienić string „Database” na poprawny.
8. W pliku „environment.prod.ts” zmieniamy wartość „apiUrl” na adres strony.
9. Prawym przyciskiem myszy naciskami na projekt „AirlineServiceSoftware” i wybieramy opcję „Publish.”



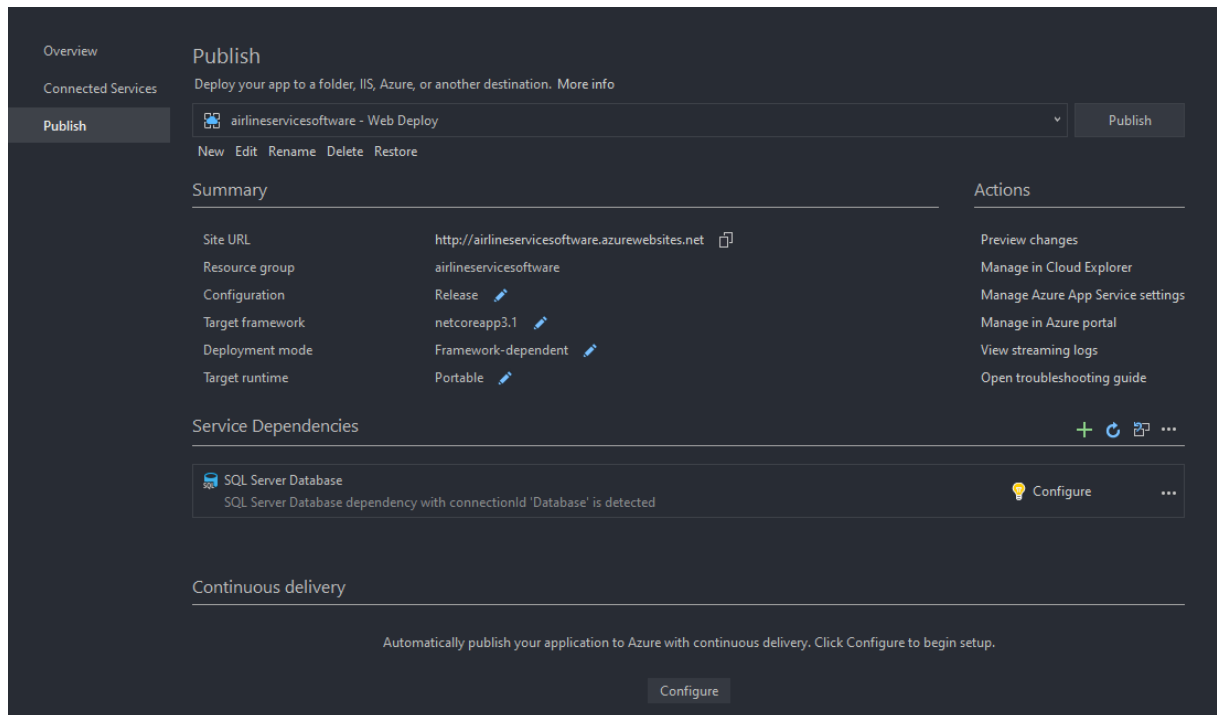
10. Pojawia się okno wyboru miejsca docelowego deploymentu.
Wybieramy odpowiednią platformę, np. Azure i klikamy na „Next”.



11. Następnie wybieramy rodzaj Azure Service na którym chcemy hostować aplikację, np. Azure App Service(Windows) i klikamy na „Next”.



12. W polu subskrypcja wybieramy rodzaj naszej subskrypcji. Następnie wybieramy Resource group do którego deployment ma zostać wykonany i naciskamy na przycisk „Finish”.



13. Pojawia się okno publikacji projektu. Naciskamy na przycisk „Publish.”