Implementacja aplikacji internetowej do organizacji wizyt duszpasterskich w parafiach

(Implementation of web app for organising pastoral visits in parishes)

Michał Chawar

Praca inżynierska

Promotor: dr Wiktor Zychla

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

12 stycznia 2026

Streszczenie

. . .

. . .

Spis treści

Rozdział 1.

Wprowadzenie

1.1. Problematyka

Doroczna wizyta duszpasterska (tzw. kolęda) jest tradycyjnym elementem życia religijnego w Polsce. Polega ona na odwiedzinach księdza w domach parafian, podczas których udziela on błogosławieństwa, rozmawia z mieszkańcami oraz zbiera ofiary na potrzeby parafii. Organizacja wizyt duszpasterskich może być jednak wyzwaniem logistycznym, zwłaszcza w większych parafiach. Często wymaga to koordynacji wielu osób, ustalania terminów oraz zarządzania informacjami o parafianach.

W dzisiejszych czasach w zależności od parafii wizyty duszpasterskie przeprowadzane są w dwóch modelach:

- **Kolęda tradycyjna**, podczas której duszpasterze starają się dotrzeć do wszystkich parafian.
- **Kolęda na zaproszenie**, podczas której duszpasterze odwiedzają tylko tych parafian, którzy wcześniej samodzielnie zgłosili chęć przyjęcia wizyty.

W niniejszej pracy skupiono się na opracowaniu aplikacji internetowej, wspierającej organizację wizyt duszpasterskich, przeprowadzanych w modelu *kolędy na zaproszenie*. Gdy dalej mowa o kolędzie, chodzi właśnie o ten model wizyty duszpasterskiej.

1.1.1. Kolęda tradycyjna

Model tradycyjny stosowany jest w większości parafii w Polsce. Funkcjonuje od dziesiątek lat i jest dobrze znany zarówno duszpasterzom, jak i parafianom. W tym modelu księża starają się odwiedzić wszystkich parafian w określonym czasie, zwyczajowo w okresie Bożego Narodzenia. Czynią to według ustalonego harmonogramu, zazwyczaj prostego i niewymagającego skomplikowanej logistyki.

Podczas gdy to podejście ma wiele zalet duszpasterskich, to jednak jest mocno czasochłonne i obciążające dla duszpasterzy oraz wiernych. Cierpi na nim często też jakoś wizyt, zarówno z powodu bardziej ścisłego ograniczenia czasowego na wizytę, jak i znacznego odsetka parafian, którzy wizytę przyjmują z obowiązku lub przyzwyczajenia, a nie z potrzeby duchowej. W większych parafiach taka forma dodatkowo angażuje ministrantów do dodatkowej pracy organizacyjnej — zapowiadania kolędy, czyli uprzedniego chodzenia od drzwi do drzwi i informowania o terminie wizyty oraz zbierania wstępnych deklaracji chęci przyjęcia wizyty. Jest to również główny sposób wczesnego prognozowania liczby wizyt danego dnia.

1.1.2. Kolęda na zaproszenie

Model ten zyskuje na popularności w ostatnich latach, zwłaszcza w większych miastach, gdzie parafianie mogą mieć bardziej zróżnicowane potrzeby, a odsetek odwiedzanych domów może być mniejszy. Znany jest również poza granicami Polski. W tej formie parafianie sami zgłaszają chęć przyjęcia wizyty duszpasterskiej poprzez odpowiedni formularz (papierowy lub elektroniczny).

Z perspektywy organizacyjnej, model ten jest znacznie bardziej efektywny. Pozwala dokładnie planować wizyty na każdy dzień, co zmniejsza obciążenie duszpasterzy, zwiększając jednocześnie kontrolę nad rozłożeniem wizyt w czasie oraz ich równomierność. Likwiduje dodatkowo potrzebę uprzedniego zapowiadania kolędy przez ministrantów, tworząc jednak nową odpowiedzialność za szczegółowe zaplanowanie porządku na dany dzień. Tym zajmuje się albo sam duszpasterz, albo wyznaczona do tego osoba (np. kościelny lub inny pracownik parafii) jako koordynator wizyty duszpasterskiej.

Kolęda na zaproszenie generuje jednak całkiem nową potrzebę — skutecznego zarządzania zgłoszeniami parafian. W większych parafiach liczba zgłoszeń może być znaczna, co wymaga odpowiednich narzędzi do ich rejestracji, przetwarzania i planowania wizyt. Wymaga to również odpowiedniej komunikacji z parafianami, aby zapewnić im jasne informacje o terminach wizyt oraz ewentualnych zmianach w harmonogramie. Dokładnie tym wymaganiom ma na celu sprostać opisywana w niniejszej pracy aplikacja internetowa.

1.2. Plan pracy

Rozdział 2.

Opis i analiza zagadnienia

Zagadnienie organizacji wizyty duszpasterskiej w modelu *kolędy na zaproszenie* jest dosyć skomplikowane i wymaga szeregu funkcjonujących i ściśle ze sobą współpracujących narzędzi (przede wszystkim do zbierania zgłoszeń, układania planów i informowania o nich). Dodatkowo potrzeba również personelu, który zajmie się zarówno wstępnym zaplanowaniem kolędy i ustaleniem harmonogramu, jak i późniejszą koordynacją przeprowadzania wizyty według planu.

Brak któregokolwiek z tych elementów znacznie utrudnia, a często nawet uniemożliwia, organizację kolędy na zaproszenie w sposób efektywny i zadowalający. Z kolei niewłaściwy dobór tych narzędzi lub nieścisła współpraca między nimi może prowadzić do licznych problemów organizacyjnych, błędów w planowaniu i komunikacji z parafianami.

Jakiś ładny dia
mik z potencja
nymi problema

Aby model kolędy na zaproszenie osiągał rzeczywistą przewagę nad modelem tradycyjnym, wszystkie te elementy muszą być starannie dobrane i nadzorowane. W wielu parafiach brakuje jednak odpowiednich narzędzi i zasobów. Nie ma też na rynku dedykowanych rozwiązań informatycznych, które kompleksowo wspierałyby ten model wizyty duszpasterskiej. Z tego powodu często korzysta się z rozwiązań prymitywnych, prowizorycznych lub niedostosowanych do specyficznych potrzeb kolędy na zaproszenie, co z kolei nie tylko prowadzi do licznych trudności organizacyjnych, ale dodatkowo powoduje frustrację zarówno wśród parafian, jak i koordynatorów kolędy.

Opisywana aplikacja wychodzi naprzeciw tym wyzwaniom, oferując kompleksowe narzędzie do zarządzania kolędą, które integruje wszystkie niezbędne funkcje w jednym miejscu. Dzięki temu parafie mogą skutecznie organizować wizyty duszpasterskie, minimalizując ryzyko błędów i usprawniając komunikację z parafianami.

2.1. Historyjki użytkownika

2.2. Wymagania ogólne

Jak wspomniano wcześniej, potrzebne narzędzia do organizacji kolędy muszą być ze sobą ściśle zintegrowane i współpracujące. Brak takich na rynku, powiązany z koniecznością przeprowadzenia kolędy, był główną motywacją do stworzenia i rozwoju opisywanej aplikacji. Już w początkowym etapie projektowania systemu wyklarowały się najważniejsze wymagania, które musiał spełniać, aby nie tylko skutecznie pomagać w organizacji kolędy, ale również mieć realną szansę na szerokie zastosowanie w różnych parafiach.

2.2.1. Jedność

Aplikacja musi integrować wszystkie kluczowe funkcje potrzebne do organizacji kolędy na zaproszenie w jednym miejscu. Oznacza to, że minimalnie powinna implementować funkcjonalności:

- zbierania zgłoszeń od parafian,
- zarządzania zebranymi zgłoszeniami oraz wprowadzania nowych,
- tworzenia i edycji harmonogramu.

Niemniej jednak biorąc pod uwagę współczesne możliwości technologiczne oraz dążenie do maksymalnej automatyzacji i uproszczenia procesu organizacji możemy na podstawie implementacji tych trzech punktów zbudować nowoczesny system, który dodatkowo będzie oferował wiele innych przydatnych funkcji:

- zautomatyzowane i zindywidualizowane powiadamianie mailowe,
- portal dla parafian, umożliwiający wgląd w dane swojego zgłoszenia, jego status, termin wizyty, a nawet przewidywane godziny,
- przeprowadzanie wizyty według planu,
- zarządzanie personelem zaangażowanym w kolędę,
- raportowanie i statystyki dotyczące przebiegu kolędy.

Powyższy zestaw funkcji zebrany w jednym miejscu pozwala na kompleksowe zarządzanie kolędą oraz zapewnia znaczną przewagę nad rozbitymi, prowizorycznymi narzędziami, które często są wykorzystywane w parafiach.

2.2.2. Intuicyjność

Aplikacja powinna być prosta i intuicyjna w obsłudze, tak aby nawet osoby nieposiadające zaawansowanych umiejętności technicznych mogły z niej skutecznie korzystać. Interfejs użytkownika musi być przejrzysty, zrozumiały i jednolity.

Należy jednak dołożyć wszelkich starań, aby nie upraszczać zbytnio funkcjonalności aplikacji kosztem jej użyteczności. Wdrożenie zbyt ograniczonego zestawu funkcji może sprawić, że aplikacja nie będzie w stanie spełnić wszystkie potrzeby parafii, co z kolei może prowadzić do jej odrzucenia na rzecz bardziej rozbudowanych, choć mniej zintegrowanych rozwiązań.

2.2.3. Poprawność

Aplikacja musi działać niezawodnie i bezbłędnie, zapewniając poprawne funkcjonowanie wszystkich swoich funkcji. Wszelkie błędy lub awarie mogą prowadzić do poważnych problemów organizacyjnych, a nawet do utraty zaufania parafian. Dlatego tak ważne jest, aby aplikacja była starannie przetestowana i regularnie aktualizowana, aby zapewnić jej stabilność i niezawodność.

Oprócz samej poprawności implementacji aplikacji należy wziąć pod uwagę, że dane wprowadzone do systemu muszą być również poprawne i spójne. W tym celu aplikacja powinna implementować mechanizmy walidacji danych, które zapewnią, że wprowadzone informacje są zgodne z określonymi standardami i nie zawierają błędów.

Ostatecznie aplikacja będzie wykorzystywana przez koordynatorów kolędy, którzy mogą nie mieć zaawansowanych umiejętności technicznych. Z tego powodu należy przyłożyć szczególną uwagę do upewnienia się, że operacje dostępne z poziomu interfejsu użytkownika są dobrze przemyślane i nie prowadzą do niezamierzonych konsekwencji, a także dobrze opisane, aby użytkownicy mogli z nich korzystać bez ryzyka popełnienia błędów.

2.2.4. Reużywalność

Aplikacja powinna być zaprojektowana w sposób umożliwiający jej łatwe dostosowanie i ponowne wykorzystanie w różnych parafiach. Oznacza to, że powinna być elastyczna i konfigurowalna, a także umożliwiać coroczne przeprowadzanie wizyt kolędowych bez konieczności manualnej rekonfiguracji lub ponownego wdrażania. Dzięki temu parafie będą mogły korzystać z aplikacji przez wiele lat, co znacznie zwiększy jej wartość i użyteczność.

Sam fakt wykorzystania w różnych parafiach wymusza również konieczność uwzględnienia tej możliwości w systemie, aby różne parafie mogły korzystać z aplikacji jednocześnie, bez konieczności tworzenia odrębnych instancji lub wersji aplikacji

dla każdej z nich.

2.3. Wymagania funkcjonalne

Podsumowując powyższe rozważania, aplikacja powinna spełniać następujące wymagania funkcjonalne:

- umożliwiać parafianom zgłaszanie chęci uczestnictwa w kolędzie na zaproszenie poprzez formularz online,
- pozwalać koordynatorom kolędy na przeglądanie, edytowanie i zarządzanie zebranymi zgłoszeniami,
- umożliwiać tworzenie i edycję harmonogramu wizyt duszpasterskich na podstawie zebranych zgłoszeń,
- automatycznie wysyłać powiadomienia mailowe do parafian o potwierdzeniu przyjęcia zgłoszenia,
- umożliwiać manualne wysyłanie powiadomień mailowych z informacjami o zmianie danych zgłoszenia, zaplanowaniu zgłoszenia, itp.,
- oferować portal dla parafian, gdzie mogą oni sprawdzać status swojego zgłoszenia, termin wizyty oraz przewidywane godziny,
- wspierać koordynację personelu zaangażowanego w kolędę, umożliwiając przypisywanie zadań i monitorowanie postępów,
- generować raporty i statystyki dotyczące przebiegu kolędy, takie jak liczba zgłoszeń, liczba przeprowadzonych wizyt itp.,
- umożliwiać konfigurację i dostosowanie aplikacji do specyficznych potrzeb różnych parafii,
- zapewniać możliwość corocznego przeprowadzania kolędy bez konieczności manualnej rekonfiguracji lub ponownego wdrażania aplikacji,
- umożliwiać jednoczesne korzystanie z aplikacji przez różne parafie, bez konieczności tworzenia odrębnych instancji lub wersji aplikacji dla każdej z nich.

2.4. Wymagania niefunkcjonalne

Oprócz wymagań funkcjonalnych, aplikacja powinna również spełniać następujące wymagania niefunkcjonalne:

- być dostępna online, aby parafie i parafianie mogli z niej korzystać z dowolnego miejsca i o dowolnym czasie,
- być skalowalna, aby mogła obsługiwać rosnącą liczbę użytkowników i zgłoszeń bez utraty wydajności,
- być bezpieczna, aby chronić dane osobowe parafian i zapewnić poufność informacji,
- być zgodna z obowiązującymi przepisami dotyczącymi ochrony danych osobowych, takimi jak RODO,
- być łatwa w utrzymaniu i aktualizacji, aby zapewnić jej długotrwałe funkcjonowanie i możliwość dostosowywania do zmieniających się potrzeb parafii.

Rozdział 3.

Porównanie z innymi implementacjami

(może być również częścią wprowadzenia przed planem pracy)

Rozdział 4.

Implementacja

4.1. Technologie i narzędzia

Aplikacja została zaimplementowana w technologii ASP.NET Core MVC w środowisku .NET 8. Do zarządzania bazą danych wykorzystano Microsoft SQL Server 2022. Interfejs użytkownika został zbudowany przy użyciu TailwindCSS, zapewniającego nowoczesny i responsywny wygląd, oraz Vue.js do obsługi interaktywnych komponentów i wieloetapowych procedur. Całość projektu została objęta konteneryzacją przy użyciu narzędzia Docker, co umożliwia łatwe wdrożenie i utrzymanie aplikacji.

4.1.1. Strona kliencka

Część frontendowa aplikacji wykonana jest w dwóch metodykach. Pierwsza z nich przeznaczona jest do tworzenia statycznych stron HTML, które są renderowane po stronie serwera w sposób typowy dla wzorca MVC. Drugie podejście łączy wstępną generację HTML z dynamicznym uzupełnianiem treści po stronie klienta przy użyciu biblioteki Vue.js. Takie podejście zostało zastosowane w miejscach, gdzie wymagana jest większa interaktywność, na przykład w formularzach wieloetapowych czy dynamicznych listach.

Razor

Do tworzenia statycznych stron HTML wykorzystano silnik szablonów Razor, który jest integralną częścią frameworka ASP.NET Core MVC. Razor umożliwia łączenie kodu C# z HTML w sposób czytelny i efektywny, co pozwala na dynamiczne generowanie treści stron na serwerze przed ich wysłaniem do przeglądarki klienta.

Z perspektywy klienta, strony wygenerowane za pomocą Razor są tradycyjnymi stronami HTML, które mogą zawierać osadzone skrypty JavaScript i style

CSS. Dzięki temu możliwe jest tworzenie responsywnych i interaktywnych interfejsów użytkownika, nawet jeśli główna logika renderowania stron odbywa się po stronie serwera.

Vue.js

Niektóre części aplikacji wymagają większej interaktywności i dynamicznego zarządzania stanem interfejsu użytkownika (w celu zachowania wymogu intuicyjności). Do ich implementacji wykorzystano bibliotekę Vue.js.

Strony te są początkowo generowane na serwerze przy użyciu Razor, a następnie po załadowaniu w przeglądarce klienta, Vue.js przejmuje kontrolę nad interaktywnymi elementami interfejsu użytkownika. Dzięki temu możliwe jest dynamiczne aktualizowanie treści, obsługa zdarzeń użytkownika oraz zarządzanie stanem aplikacji bez konieczności ponownego ładowania całej strony.

Aplikacje Vue.js są implementowane bezpośrednio w tagach *script* w niektórych plikach widoków. Ładowany jest wówczas globalny plik biblioteki, który udostępnia wszelkie funkcjonalności jako właściwości globalnego obiektu Vue. Następnie za jego pomocą tworzone są instancje aplikacji Vue, które są przypisywane do określonych elementów DOM na stronie. Ten proces następuje w przeglądarce użytkownika, co pozwala na płynne przejście od statycznego renderowania do dynamicznej interaktywności.

W opisywanym projekcie aplikacje Vue.js używane są na dwa sposoby:

- do zwiększania interaktywności prostych bądź zaawansowanych elementów interfejsu po stronie klienta bez potrzeby komunikacji z serwerem,
- do zarządzania bardziej złożonymi komponentami interfejsu, które wymagają komunikacji z serwerem w celu pobierania lub wysyłania danych.

W pierwszym przypadku często zachowywana jest logika renderowania po stronie serwera, włącznie z tworzeniem formularzy za pomocą pomocników HTML ASP.NET Core przy użyciu modeli widoków. Vue.js jest wtedy wykorzystywany do obsługi interakcji użytkownika, takich jak walidacja danych wprowadzanych w formularzach, dynamiczne dodawanie lub usuwanie elementów listy, czy aktualizacja widoku na podstawie działań użytkownika bez konieczności ponownego ładowania strony. Czasem jednak dane osadzane są bezpośrednio w zmiennej JavaScript w widoku (po przekonwertowaniu do JSON), co pozwala na pełną kontrolę nad renderowaniem interfejsu po stronie klienta przez samą aplikację Vue.

W drugim przypadku Vue.js zarządza bardziej złożonymi komponentami, które wymagają komunikacji z serwerem. Wówczas aplikacja przypomina bardziej wzorzec SPA (Single Page Application), gdzie Vue.js odpowiada za renderowanie interfejsu

użytkownika, a komunikacja z serwerem odbywa się za pomocą asynchronicznych żądań HTTP (przy użyciu Fetch API). Dane są pobierane z serwera w formacie JSON, a następnie wykorzystywane do aktualizacji widoku w czasie rzeczywistym. Podobnie, dane wprowadzone przez użytkownika są wysyłane z powrotem na serwer w formacie JSON, gdzie są przetwarzane i zapisywane w bazie danych. Nie jest to jednak pełna aplikacja SPA, ponieważ nawigacja między różnymi stronami nadal odbywa się poprzez tradycyjne przeładowanie strony.

JQuery i JavaScript

Do obsługi prostych interakcji na stronach wykorzystano również bibliotekę jQuery oraz czysty JavaScript. W miejscach, gdzie nie jest wymagana pełna funkcjonalność Vue.js, jQuery pozwala na szybkie i efektywne manipulowanie elementami DOM oraz obsługę zdarzeń. Dodatkowo odpowiada za walidację formularzy po stronie klienta, co poprawia doświadczenie użytkownika poprzez natychmiastowe informowanie o błędach przed wysłaniem danych na serwer.

JavaScript jest również używany do implementacji powszechnych funkcji dla aplikacji, znajdujących się w plikach zewnętrznych, które są dołączane do odpowiednich widoków.

TailwindCSS

Do stylizacji interfejsu użytkownika wykorzystano framework CSS o nazwie TailwindCSS. Jest to narzędzie oparte na podejściu utility-first, które umożliwia szybkie tworzenie responsywnych i estetycznych interfejsów użytkownika poprzez stosowanie gotowych klas CSS bez konieczności pisania własnych stylów od podstaw.

Głównymi zaletami TailwindCSS są jego elastyczność i możliwość tworzenia responsywnych projektów. Framework oferuje szeroki zestaw klas, które pozwalają na precyzyjne kontrolowanie wyglądu elementów interfejsu, takich jak marginesy, wypełnienia, kolory, typografia i układ. Dzięki temu aplikacja została w prosty sposób dostosowana do różnych rozmiarów ekranów, zapewniając optymalne doświadczenie użytkownika na urządzeniach mobilnych, tabletach i komputerach stacjonarnych.

Nad to użyto również wtyczki DaisyUI, która rozszerza możliwości TailwindCSS o gotowe komponenty UI, takie jak przyciski, formularze, karty i nawigacje, tworzone za pomocą określonych klas. Dzięki temu proces tworzenia interfejsu użytkownika był szybszy i bardziej efektywny, pozwalając skupić się na funkcjonalności aplikacji zamiast na szczegółach stylizacji.

4.1.2. Strona serwerowa

Część backendowa aplikacji została zaimplementowana przy użyciu frameworka ASP.NET Core, opartego na platformie ṄET w wersji 8. Wykorzystano w nim różne biblioteki i narzędzia dostępne w ekosystemie ṄET, aby zapewnić wydajność, skalowalność i bezpieczeństwo aplikacji.

ASP.NET Core

Framework ASP.NET Core umożliwia tworzenie aplikacji webowych zgodnych z wzorcem Model-View-Controller (MVC), co pozwala na oddzielenie logiki biznesowej od warstwy prezentacji i danych. Nadal pozwala przy tym udostępniać interfejs API w obrębie tej samej aplikacji, co zostało wykorzystane w projekcie. ASP.NET Core oferuje wbudowane mechanizmy do obsługi routingu, autoryzacji, uwierzytelniania oraz zarządzania sesjami, co ułatwia tworzenie bezpiecznych i wydajnych aplikacji webowych. Dodatkowo, framework ten jest wysoce konfigurowalny i wspiera nowoczesne praktyki programiste, takie jak wstrzykiwanie zależności i middleware.

Entity Framework Core Do komunikacji z bazą danych wykorzystano Entity Framework Core (EF Core), który jest popularnym narzędziem ORM (Object-Relational Mapping) dla platformy ṄET. EF Core umożliwia programistom pracę z bazą danych za pomocą obiektów C#, eliminując potrzebę pisania bezpośrednich zapytań SQL. Dzięki temu proces tworzenia, odczytu, aktualizacji i usuwania danych (CRUD) staje się bardziej intuicyjny i zintegrowany z logiką aplikacji.

ASP.NET Core Identity Do zarządzania uwierzytelnianiem i autoryzacją użytkowników wykorzystano bibliotekę ASP.NET Core Identity. Jest to kompleksowe rozwiązanie, które umożliwia tworzenie i zarządzanie kontami użytkowników, obsługę ról oraz implementację mechanizmów bezpieczeństwa, takich jak resetowanie haseł czy weryfikacja dwuetapowa. ASP.NET Core Identity integruje się bezproblemowo z frameworkiem ASP.NET Core, co pozwala na łatwe dodanie funkcji logowania i zarządzania użytkownikami do aplikacji webowej oraz przechowywanie ich danych w bazie danych za pośrednictwem Entity Framework Core.

WebOptimizer Do optymalizacji dostarczania statycznych plików JavaScript zastosowano bibliotekę WebOptimizer. Narzędzie to umożliwia minifikację, łączenie i kompresję plików statycznych, co prowadzi do zmniejszenia rozmiaru przesyłanych zasobów i przyspieszenia ładowania stron internetowych. WebOptimizer automatycznie przetwarza pliki podczas uruchamiania aplikacji, co ułatwia zarządzanie zasobami i poprawia wydajność aplikacji webowej.

MailKit Do obsługi wysyłania wiadomości e-mail z aplikacji wykorzystano bibliotekę MailKit. Jest to nowoczesne i wydajne narzędzie do obsługi protokołów SMTP, POP3 i IMAP w środowisku ṄET. MailKit oferuje szeroki zakres funkcji, takich jak tworzenie i wysyłanie wiadomości e-mail, obsługa załączników, szyfrowanie oraz autoryzacja. Biblioteka ta jest znana ze swojej wydajności i niezawodności, co czyni ją idealnym wyborem do integracji funkcji e-mail w aplikacjach webowych.

DataProtection Do zapewnienia trwałości kluczy kryptograficznych wykorzystano bibliotekę ASP.NET Core Data Protection. Dzięki integracji z Entity Framework Core, klucze są przechowywane w bazie danych, co zapewnia ich persystencję między restartami aplikacji, umożliwiając zachowanie sesji użytkowników i likwidując wymóg ponownego logowania po restarcie serwera.

QuestPDF Do generowania dokumentów PDF w aplikacji wykorzystano bibliotekę QuestPDF. Jest to nowoczesne narzędzie do tworzenia wysokiej jakości dokumentów PDF w środowisku ṄET. QuestPDF oferuje prosty i intuicyjny interfejs programistyczny, który umożliwia definiowanie układu i stylu dokumentów za pomocą kodu C#. Biblioteka obsługuje różnorodne funkcje, takie jak dodawanie tekstu, obrazów, tabel i wykresów, co pozwala na tworzenie profesjonalnie wyglądających raportów i dokumentów bez konieczności korzystania z zewnętrznych narzędzi do edycji PDF.

Narzędzie QuestPDF zostało wykorzystane na licencji Community MIT, która pozwala na darmowe użycie biblioteki w projektach niekomercyjnych i komercyjnych, generujących dochód poniżej $1,000,000 rocznie.

Microsoft SQL Server 2022

Aplikacja korzysta z relacyjnej bazy danych Microsoft SQL Server 2022 do przechowywania wszystkich danych niezbędnych do jej funkcjonowania. Komunikacja pomiędzy aplikacją a bazą danych odbywa się za pośrednictwem opisanego wyżej Entity Framework Core, który umożliwia mapowanie obiektów C# na tabele i rekordy w bazie danych. Dodatkowo z bazą danych komunikują się narzędzia ASP.NET Core Identity oraz Data Protection (poprzez integrację z EF Core).

Traefik

W środowisku produkcyjnym do zarządzania ruchem sieciowym i obsługi certyfikatów SSL/TLS wykorzystano narzędzie Traefik. Jest to nowoczesny i wydajny reverse proxy oraz load balancer, który automatycznie wykrywa usługi i konfiguruje trasowanie ruchu na podstawie reguł zdefiniowanych przez użytkownika. Traefik zintegrowany jest z Dockerem, co umożliwia dynamiczne zarządzanie ruchem

sieciowym w środowiskach kontenerowych. Dodatkowo, Traefik oferuje wbudowaną obsługę Let's Encrypt, co pozwala na automatyczne generowanie i odnawianie certyfikatów SSL/TLS, zapewniając bezpieczną komunikację między klientami a serwerem.

4.1.3. Ciągła integracja i dostarczanie (CI/CD)

Główna część aplikacji została objęta konteneryzacją przy użyciu narzędzia Docker. Konteneryzacja pozwala na zapakowanie aplikacji wraz ze wszystkimi jej zależnościami w odizolowane środowisko, co umożliwia łatwe wdrożenie i dalsze utrzymanie. Dodatkowo za pomocą pliku Dockerfile zdefiniowano proces budowania obrazu kontenera od podstaw, co zapewnia spójność środowiska uruchomieniowego aplikacji niezależnie od miejsca jej wdrożenia.

Wraz z kontenerem aplikacji ASP.NET Core przy pomocy narzędzia orkiestracji kontenerów Docker Compose można w prosty sposób uruchomić również kontener bazy danych Microsoft SQL Server 2022 oraz (w środowisku produkcyjnym) kontener Traefik, jednocześnie konfigurując ich współpracę, sieć wewnętrzną oraz wolumeny do trwałego przechowywania danych.

4.1.4. Testy

ić testy XD

4.1.5. Zarządzanie kodem źródłowym

Kod źródłowy aplikacji jest przechowywany w repozytorium Git na platformie GitHub. Wykorzystano funkcje zarządzania wersjami, takie jak gałęzie i pull requesty, aby umożliwić współpracę zespołową (w przyszłości) oraz śledzenie zmian w kodzie. Dodatkowo, repozytorium zawiera dokumentację projektu, instrukcje dotyczące wdrożenia oraz konfiguracji środowiska deweloperskiego.

4.1.6. Narzędzia modelowania diagramów

Do tworzenia diagramów UML oraz innych wizualizacji na potrzeby tej pracy użyto dwóch narzędzi:

- Structurizr — do tworzenia diagramów architektury oprogramowania (C4 Model),
- Visual Paradigm (w wersji Community) — do tworzenia pozostałych diagramów (np. modelu danych).

4.2. Modele danych

Poniżej przedstawiono dwa diagramy. Pierwszy odpowiada modelowi dziedzinowemu, reprezentującemu główne pojęcia oraz ich relacje w kontekście logiki aplikacji. Drugi odnosi się do modelu obiektowego, który odwzorowuje strukturę encji EF Core, tym samym determinujących strukturę bazy danych.

4.2.1. Model pojęciowy

Zaprezentowany diagram modelu pojęciowego (rys. 4.1) ilustruje główne pojęcia oraz ich wzajemne relacje w kontekście logiki aplikacji. Model ten stanowi abstrakcyjną reprezentację struktur danych i zależności między nimi, niezależnie od konkretnej implementacji technicznej.

Całość modelu mieści się w logicznych granicach jednej **parafii** — wymienione pojęcia odnoszą się do zarządzania kolędą w obrębie konkretnej parafii, a tym samym do jednej domeny aplikacji.

Personel

Personel parafialny składa się z różnych ról, które mają określone uprawnienia i obowiązki w kontekście zarządzania kolędą. Główne role to:

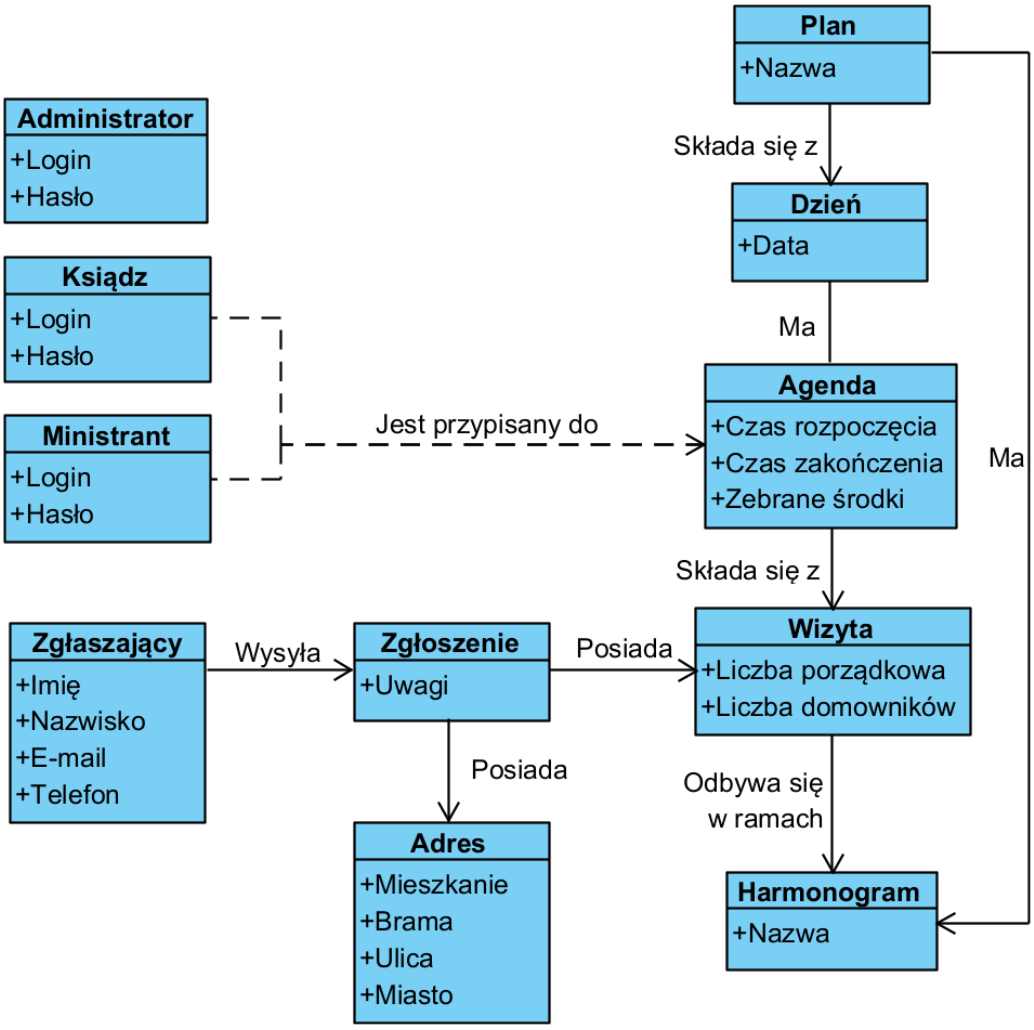
- **Administrator** — osoba odpowiedzialna za zarządzanie aplikacją, w tym tworzenie i modyfikowanie planów kolędy oraz zarządzanie użytkownikami.
- **Ksiądz** — duchowny, który odwiedza parafian podczas kolędy. Może zarządzać danym planem i planować wizyty.
- **Ministrant** — osoba, która towarzyszy księdzu podczas wizyt. Może być przypisana do konkretnych agend w planie kolędy i na bieżąco aktualizować status wizyt.

Plan

Najbardziej ogólnym pojęciem w zakresie planowania kolędy jest **Plan**. Reprezentuje on konkretny plan kolędy dla danego roku. Planów może być wiele, są zarządzane przez parafialnego administratora aplikacji.

Harmonogram

Każdy plan kolędy musi zawierać przynajmniej jeden **Harmonogram**. Harmonogramy umożliwiają podział zgłoszeń parafian na różne grupy, które z kolei



Rysunek 4.1: Model pojęciowy aplikacji

mogą służyć do filtrowania zgłoszeń, a także do tworzenia odrębnych planów wizyt dla różnych księży. Przykładowo można za ich pomocą rozdzielić kolędę w terminie zasadniczym od kolędy dodatkowej.

Dzień

Każdy plan składa się z wielu **Dni**, które reprezentują konkretne dni w trakcie trwania kolędy.

Agenda

Każdy dzień zawiera wiele **Agend**, które grupują i porządkują wizyty danego dnia. Każda agenda jest przypisana do konkretnego dnia i zawiera informacje o ministrantach towarzyszących podczas wizyt. Agendy są odpowiednikiem list wizyt,

które ksiądz i ministranci realizują w danym dniu kolędy. W najprostszym rozumieniu mogą służyć więc jako podział wizyt na księdza pierwszego, drugiego, itd.

Wizyta

Podstawowym elementem planu kolędy jest **Wizyta**. Reprezentuje ona konkretną wizytę księdza u parafianina. Wizyta może być przypisana do konkretnej agendy, a tym samym być zaplanowana na konkretny dzień. Kolejność wizyt w agendzie jest wyznaczana przez liczbę porządkową wizyty. Zasadniczo przypisana jest też do konkretnego harmonogramu.

Zgłoszenie, zgłaszający i adres

Każda wizyta jest powiązana z konkretnym **Zgłoszeniem** parafianina. Zgłoszenie zawiera wszystkie niezbędne informacje o parafianinie, takie jak dane kontaktowe oraz adres zamieszkania (w powiązanych pojęciach). Do zgłoszenia można dołączyć także dodatkowe uwagi (np. prośby dotyczące wizyty lub preferencje dotyczące terminu).

4.2.2. Model obiektowy

Poniżej zaprezentowano dwa diagramy modelu obiektowego aplikacji. Pierwszy z nich (rys. 4.2) odnosi się do kontekstu ogólnego (centralnego), tj. do logiki zarządzania użytkownikami (docelowo w różnych domenach — parafiach). Drugi (rys. 4.3) przedstawia model obiektowy funkcjonujący w obrębie konkretnej domeny (kontekst parafialny).

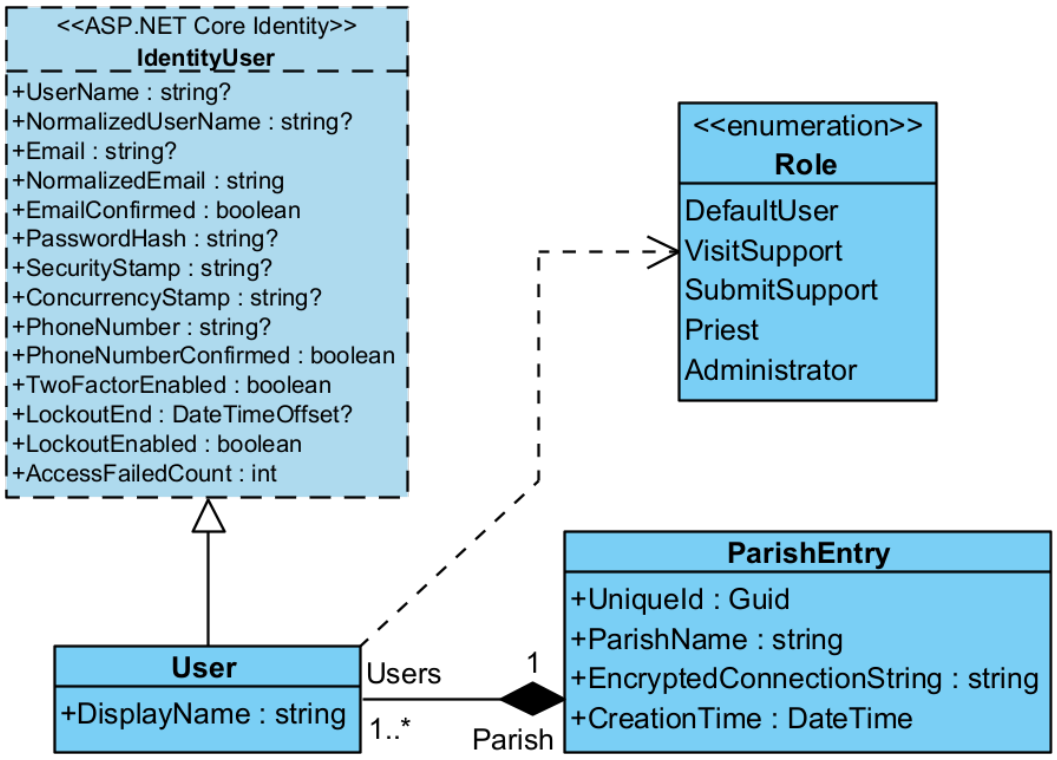
Kontekst centralny

W kontekście centralnym aplikacji zajęto się zarządzaniem użytkownikami oraz ich rolami w różnych domenach (parafiach). Główne klasy w tym modelu to **User**, **Role** oraz **ParishEntry**.

User Reprezentuje użytkownika aplikacji. Zawiera podstawowe informacje o nim, w większości dziedziczone z klasy `IdentityUser` z ASP.NET Core Identity. Jest przypisany do konkretnej parafii.

Najważniejsze pola klasy `User` to:

- `UserName` — login użytkownika,
- `Email` — adres e-mail użytkownika,



Rysunek 4.2: Model obiektowy aplikacji (kontekst centralny)

- `PasswordHash` — hash hasła użytkownika,
- `DisplayName` — nazwa użytkownika.

Role Reprezentuje role użytkowników w aplikacji. Każda rola definiuje zestaw uprawnień i obowiązków, które użytkownik posiada w kontekście zarządzania kolędą.

`Role` jest enumeracją o następujących wartościach:

- **DefaultUser** — uprawnia do podstawowego dostępu do aplikacji,
- **VisitSupport** — uprawnia do przeprowadzania wizyt kolędowych (odpowiednik ministranta),
- **SubmitSupport** — uprawnia do wprowadzania zgłoszeń i zarządzania nimi,
- **Priest** — uprawnia do wszystkich powyższych czynności oraz do planowania wizyt,
- **Administrator** — uprawnia do zarządzania aplikacją i wszystkimi jej funkcjami.

ParishEntry Reprezentuje rzeczywistą parafię, która używa aplikacji do zarządzania kolędą. Do każdej jest przypisany przynajmniej jeden użytkownik.

Najważniejsze pola klasy `ParishEntry` to:

- `UniqueId` — unikalny identyfikator parafii,
- `ParishName` — nazwa parafii,
- `EncryptedConnectionString` — zaszyfrowany łańcuch połączenia do bazy danych parafii,
- `CreationTime` — data utworzenia parafii w systemie.

Kontekst parafialny

Kontekst parafialny zbiera wszystkie klasy związane z zarządzaniem kolędą w obrębie konkretnej parafii. Główne klasy w tym modelu to **Plan**, **Schedule**, **Day**, **Agenda**, **Submission** oraz **Visit**.

W tym modelu wyróżniają się dwie najistotniejsze grupy encji (w podziale funkcjonalnym): encje służące do *Przyjmowania zgłoszeń* i do *Planowania wizyt*. Wszystkie opisane są poniżej.

Przyjmowanie zgłoszeń

Submission Reprezentuje zgłoszenie na kolędę. Zawiera wszystkie niezbędne informacje o parafianinie, takie jak dane kontaktowe oraz adres zamieszkania (w powiązanych klasach).

Najważniejsze pola klasy `Submission` to:

- `SubmitterNotes` — dodatkowe uwagi (od zgłaszającego do administratora),
- `AdminMessage` — informacja systemowa (od administratora do zgłaszającego),
- `AdminNotes` — wewnętrzne notatki (widoczne tylko dla zalogowanych użytkowników),
- `NotesStatus` — status realizacji dodatkowych uwag (np. oczekujące, zrealizowane).

Submitter Reprezentuje parafianina, który składa zgłoszenie na kolędę. Jest bezpośrednio powiązany ze zgłoszeniem (lub wieloma).

Najważniejsze pola klasy `Submitter` to:

- `Name` — imię zgłaszającego,
- `Surname` — nazwisko zgłaszającego,
- `PhoneNumber` — numer telefonu zgłaszającego,
- `Email` — adres e-mail zgłaszającego.

Address Reprezentuje adres zamieszkania parafianina. Jest bezpośrednio powiązany ze zgłoszeniem (w relacji *1 do 1* z dokładnością do konkretnego planu kolędy). Jest on prawie w całości zatomizowany, aby ujednolicić format adresów, podawanych w formularzu zgłoszeniowym (by zachować jednoznaczność).

Jest powiązany z szeregiem klas pomocniczych, które reprezentują poszczególne elementy adresu (np. budynek (brama), ulica, miasto).

Najważniejsze pola klasy `Address` to:

- `ApartmentNumber` — numer mieszkania,
- `ApartmentLetter` — litera mieszkania (opcjonalnie, raczej rzadko),
- właściwości *cache* — kopia tekstowa właściwości z klas pomocniczych (dla przyspieszenia operacji bazodanowych),
- `FilterableString` — znormalizowany łańcuch znaków do celów filtrowania i wyszukiwania adresów (*computed* — obliczony przez bazę danych na podstawie pól *cache*).

Building Reprezentuje budynek (bramę) w adresie zamieszkania parafianina. Każdy budynek może mieć wiele adresów (mieszkań). Jest tworzony przez administratora parafii i wybierany z listy w formularzu zgłoszeniowym (i kilku innych miejscach).

Najważniejsze pola klasy `Building` to:

- `Number` — numer budynku,
- `Letter` — litera budynku (opcjonalnie),
- `FloorCount` — liczba pięter w budynku (opcjonalnie, do celów statystycznych),
- `ApartmentCount` — liczba mieszkań w budynku (opcjonalnie, do celów statystycznych),
- `HighestApartmentNumber` — najwyższy numer mieszkania w budynku (opcjonalnie, do celów statystycznych),

- `HasElevator` — czy budynek posiada windę (może mieć wpływ na planowanie wizyt),
- `AllowSelection` — czy budynek może być wybrany w formularzu zgłoszeniowym (w przeciwnym przypadku jest ukryty i może zostać wybrany tylko przez zalogowanego użytkownika).

Street Reprezentuje ulicę w adresie zamieszkania parafianina. Każda ulica może mieć wiele budynków. Podobnie jak budynek, jest tworzona przez administratora parafii i wybierany z listy w formularzu zgłoszeniowym (i kilku innych miejscach).

Najważniejsze pola klasy `Street` to:

- `Name` — nazwa ulicy (bez tytułów typu ulica, aleja, plac itd.),
- `PostalCode` — kod pocztowy ulicy (opcjonalnie).

StreetSpecifier Reprezentuje typ ulicy (np. ulica, aleja, plac itd.). Każda ulica jest powiązana z jednym `StreetSpecifier`, który określa jej typ. Podobnie jak budynek i ulica, jest tworzony przez administratora parafii.

Najważniejsze pola klasy `StreetSpecifier` to:

- `FullName` — pełna nazwa typu ulicy (np. ulica, aleja, plac itd.),
- `Abbreviation` — skrócona nazwa typu ulicy (np. ul., al., pl.).

City Reprezentuje miasto w adresie zamieszkania parafianina. Każde miasto może mieć wiele ulic. Podobnie jak powyższe klasy, jest tworzone przez administratora parafii.

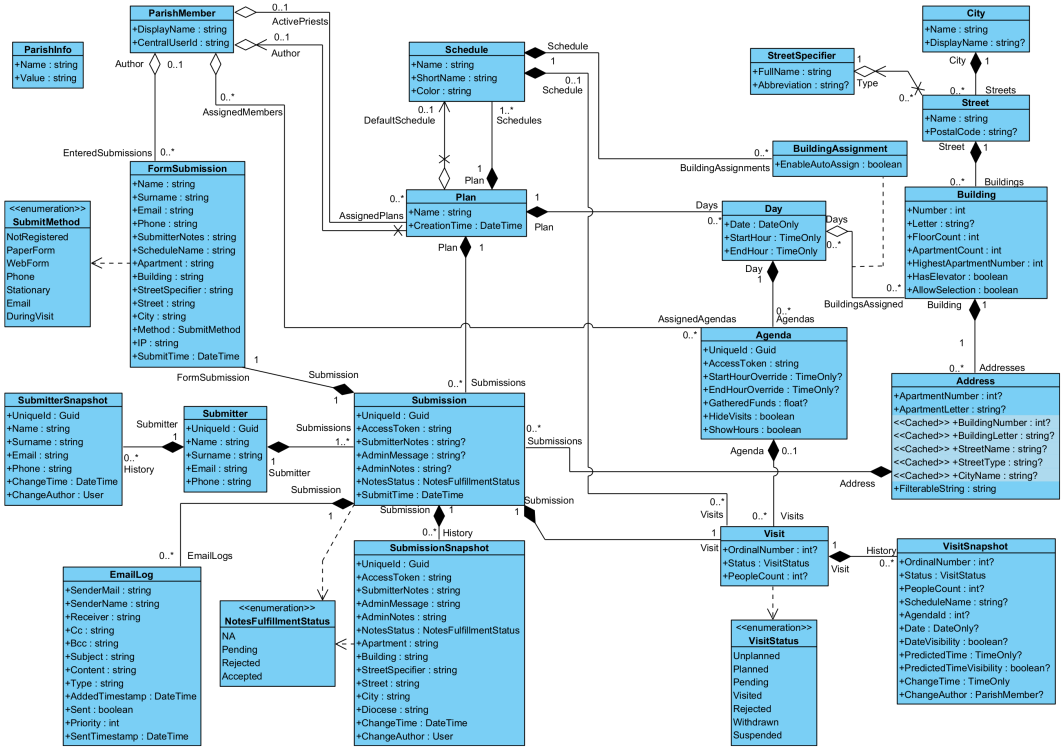
Najważniejsze pola klasy `City` to:

- `Name` — pełna nazwa miasta,
- `DisplayName` — wyświetlana nazwa miasta.

Planowanie wizyt

Visit Jest odpowiednikiem zgłoszenia w kontekście planowania wizyt. Reprezentuje konkretną wizytę księdza u parafianina. Jest powiązana z danym zgłoszeniem w relacji *1 do 1*, w ten sposób są ze sobą ściśle powiązane.

Najważniejsze pola klasy `Visit` to:



Rysunek 4.3: Model obiektowy aplikacji (kontekst parafialny)

- `OrdinalNumber` — liczba porządkowa wizyty w agendzie,
- `Status` — status wizyty (np. zaplanowana, zaakceptowana, odrzucona),
- `PeopleCount` — liczba domowników uczestniczących w wizycie (do celów statystycznych).

Plan Reprezentuje plan wizyt kolędowych (najczęściej na dany rok). Zawiera wiele dni, które z kolei zawierają agendy i wizyty. Jest zarządzany przez parafialnego administratora aplikacji. Można przypisać do niego księży, którzy będą w nim występować.

Najważniejsze pola klasy `Plan` to:

- `Name` — nazwa planu,
- `CreationTime` — czas utworzenia planu.

Schedule Reprezentuje harmonogram wizyt w planie kolędy. Służy do podziału zgłoszeń parafian na różne grupy, np. na kolędę w terminie i kolędę dodatkową. Każdy plan musi zawierać przynajmniej jeden harmonogram.

Najważniejsze pola klasy `Schedule` to:

- `Name` — nazwa harmonogramu,
- `ShortName` — skrócone oznaczenie harmonogramu,
- `Color` — kolor harmonogramu (zapisany w formacie szesnastkowym z poprzedzającym znakiem #).

Day Reprezentuje konkretny dzień w trakcie trwania kolędy. Zawiera wiele agend, które grupują wizyty danego dnia.

Najważniejsze pola klasy `Day` to:

- `Date` — data dnia,
- `StartHour` — godzina rozpoczęcia kolędy w danym dniu,
- `EndHour` — godzina zakończenia kolędy w danym dniu.

Agenda Reprezentuje listę wizyt w konkretnym dniu kolędy. Każda agenda jest przypisana do konkretnego dnia i można do niej przypisywać księdza oraz ministrantów. Porządek wizyt w agendzie jest wyznaczany przez liczbę porządkową wizyty.

Najważniejsze pola klasy `Agenda` to:

- `StartHourOverride` — godzina rozpoczęcia kolędy dla danej agendy (opcjonalnie, nadpisuje godzinę z dnia),
- `EndHourOverride` — godzina zakończenia kolędy dla danej agendy (opcjonalnie, nadpisuje godzinę z dnia),
- `GatheredFunds` — suma zebranych funduszy podczas wizyt w danej agendzie,
- `HideVisits` — czy ukryć wizyty w danej agendzie przed parafianami (np. przed publikacją planu, przy tworzeniu szkicu),
- `ShowHours` — czy pokazywać parafianom przewidywane godziny ich wizyty (dopiero gdy plan jest zatwierdzony).

BuildingAssignment Reprezentuje przypisanie konkretnego budynku do konkretnego dnia w planie kolędy (w ramach wybranego harmonogramu). Pozwala to na automatyczne sugerowanie terminów wizyt dla każdego zgłoszenia (na podstawie adresu oraz harmonogramu), a także automatyczny zapis przy rejestracji zgłoszenia.

Jedyne pole klasy `BuildingAssignment` to:

- `EnableAutoAssign` — czy włączyć automatyczne przypisywanie wizyt dla danego budynku w danym dniu (dla danego budynku w danym harmonogramie tylko jeden dzień może mieć włączony auto-zapis).

Tworzenie historii zmian Oprócz powyższych dwóch grup w modelu znajduje się również pięć klas pomocniczych, służących do rejestrowania historii zmian dla wybranych encji modelu. Trzy z nich to klasy typu *snapshot*: **SubmissionSnapshot**, **SubmitterSnapshot** oraz **VisitSnapshot**. Przechowują one kopię stanu odpowiadającej im encji w momencie dokonania zmiany wraz z informacją o autorze zmiany. Czwarta klasa to **FormSubmission**, która zachowuje kopię oryginalnych danych zgłoszenia (w momencie wprowadzenia do systemu). Pozwala na odtworzenie pierwotnej wersji zgłoszenia w przypadku nadużyć lub problemów. Ostatnia klasa to **EmailLog**, która rejestruje wysyłane wiadomości e-mail z aplikacji, wraz z ich zawartością i odbiorcami. Pozwala to na audyt i śledzenie komunikacji prowadzonej przez system, a także na ponawianie próby wysyłania wiadomości (gdy serwer pocztowy był niedostępny itp.).

Zarządzanie parafią W modelu znajdują się również dwie encje do zarządzania informacjami dot. parafii: **ParishMember** oraz **ParishInfo**.

Pierwsza z nich reprezentuje użytkownika systemu, odpowiadającego dokładnie jednemu użytkownikowi z kontekstu centralnego (klasa `User`) poprzez pole `CentralUserId`. Podczas gdy do celów zarządzania użytkownikami i pobierania ich informacji służy klasa `User` w kontekście centralnym, to klasa `ParishMember` pozwala na tworzenie relacji między użytkownikami (reprezentowanymi przez nią w kontekście parafii) a innymi encjami, np. planem kolędy (przypisanie księży do planu) lub agendą (przypisanie ministrantów do agendy).

Druga z nich jest prostym magazynem dla wszelakich informacji (w tym tych o parafii), które mogą być potrzebne w aplikacji, ale nie mają dedykowanej encji w modelu. Składa się z par klucz-wartość, gdzie klucz jest unikalnym identyfikatorem informacji, a wartość przechowuje jej treść.

Typy wyliczeniowe W modelu zastosowano również kilka typów wyliczeniowych (*enum*'ów), które służą do definiowania stałych wartości dla określonych właściwości encji.

SubmitMethod Określa metodę, za pomocą której parafianin złożył zgłoszenie na kolędę. Dostępne wartości to:

- **NotRegistered** — nie zarejestrowano (domyślnie),
- **PaperForm** — formularz papierowy,
- **WebForm** — formularz internetowy,
- **Phone** — telefonicznie,
- **Stationary** — osobiście (np. w kancelarii parafialnej),

- **Email** — mailowo,
- **DuringVisit** — osobiście podczas sąsiednich wizyt danego dnia.

NotesFulfillmentStatus Określa status realizacji dodatkowych uwag przez personel parafii. Dostępne wartości to:

- **NA** — nie dotyczy (domyślnie, gdy brak uwag),
- **Pending** — oczekujące (domyślnie gdy są uwagi),
- **Rejected** — niezrealizowane,
- **Accepted** — zrealizowane.

VisitStatus Określa status wizyty kolędowej. Dostępne wartości to:

- **Unplanned** — niezaplanowana (domyślnie),
- **Planned** — zaplanowana (w agendzie),
- **Pending** — trwająca,
- **Visited** — zrealizowana (odbyta),
- **Rejected** — nieodbyta (np. parafianin nie otworzył drzwi),
- **Withdrawn** — wycofana (np. parafianin wycofał zgłoszenie),
- **Suspended** — wstrzymana (w szczególnych okolicznościach, stan tymczasowy podczas przeprowadzania wizyt).

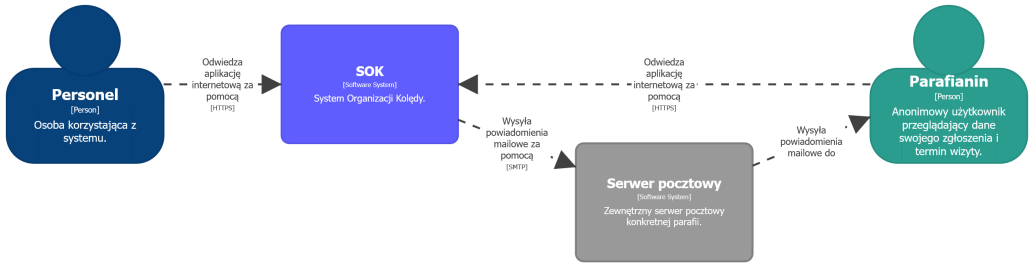
4.3. Architektura

4.3.1. Poziom I — diagram kontekstowy systemu

Architektura aplikacji została zaprezentowana poniżej z użyciem modelu C4. Jego najwyższy poziom (*System Context Diagram*, rys. 4.4) przedstawia ogólny widok na system oraz jego interakcje z użytkownikami i zewnętrznymi systemami.

W naszym przypadku głównym komponentem jest aplikacja webowa **SOK**. Umożliwia ona zarządzanie kolędą w parafiach poprzez interfejs użytkownika dostępny z poziomu przeglądarki internetowej. Aplikacja komunikuje się dodatkowo z serwerem pocztowym (do wysyłania powiadomień e-mail).

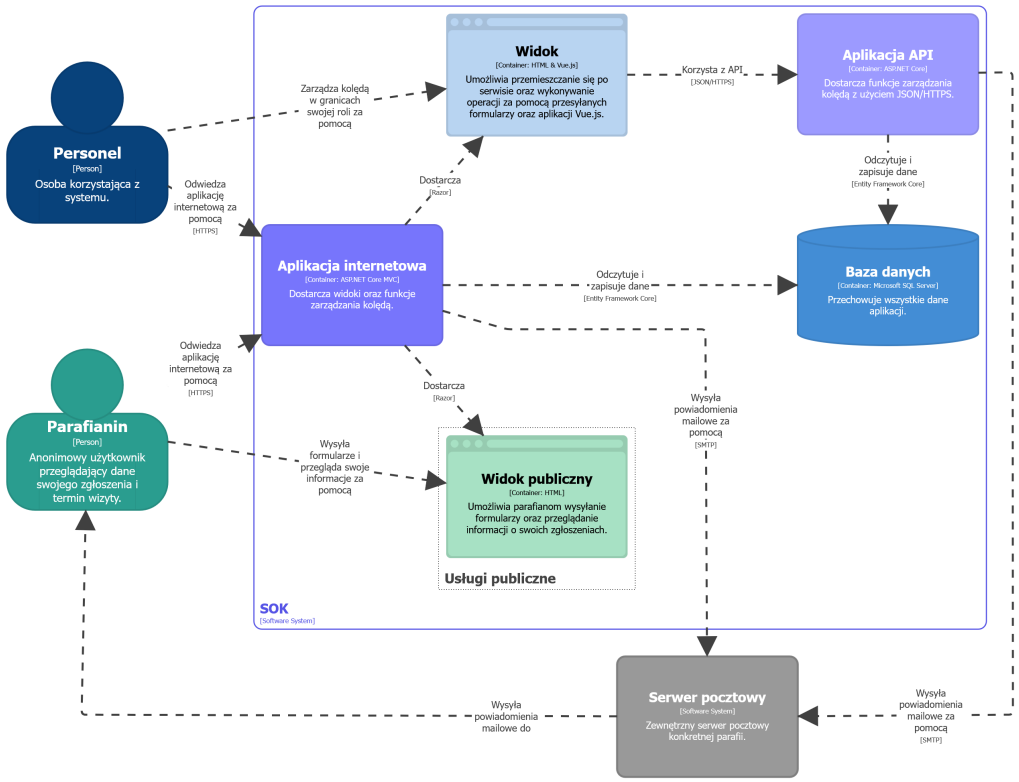
Na diagramie zaznaczono również dwóch aktorów: **Parafianina** oraz **Personel parafialny**. Parafianin może składać zgłoszenia na kolędę oraz przeglądać swój panel zgłoszenia (wszystko jako anonimowy użytkownik). Personel parafialny zarządza zgłoszeniami i planuje wizyty kolędowe.



Rysunek 4.4: Diagram kontekstowy systemu

4.3.2. Poziom II — diagram kontenerów

Drugi poziom modelu C4 (*Container Diagram*, rys. 4.5) przedstawia główne kontenery aplikacji oraz ich interakcje.



Rysunek 4.5: Diagram kontenerów aplikacji

Na tym poziomie widzimy rozróżnienie między personelem a parafianinami. Personel, odwiedzając aplikację webową, korzysta z pełnego interfejsu użytkownika, który udostępnia wszystkie funkcje aplikacji (w granicach roli). Parafianin natomiast korzysta z *widoków publicznych*, które są dostępne bez logowania. W dalszej części pracy odnosząc się do *widoku* będziemy mieli na myśli interfejs użytkownika zalogowanego.

Widok aplikacji webowej może również korzystać z API, które udostępnia wybrane funkcje aplikacji w formie usług sieciowych (por. rozdział 4.1.1.). Aplikacja API, tak samo jak główny komponent aplikacji, komunikuje się z bazą danych oraz z serwerem pocztowym.

4.3.3. Poziom III — diagram komponentów

Aplikacja webowa

Aplikacja webowa jest zaprogramowana w architekturze MVC (Model-View-Controller). Podział ten widoczny jest na diagramie (rys. 4.6), choć nie jest wyraźny przy pierwszej analizie z powodu dodatkowych komponentów, które zostały zaprezentowane ze względu na ich istotną rolę w aplikacji.

Kontrolery Pierwszym rodzajem komponentu jest **Kontroler**, który obsługuje żądania HTTP od użytkowników i koordynuje przepływ danych między modelem a widokiem. W aplikacji znajdują się dwa rodzaje kontrolerów: kontrolery aplikacji (dla personelu parafialnego) oraz kontrolery publiczne (dla parafian).

Pierwszy z nich obsługuje wszystkie funkcje aplikacji dostępne dla zalogowanych użytkowników (w granicach ich ról). Korzysta przy tym z komponentu do autoryzacji, aby sprawdzić uprawnienia użytkownika przed wykonaniem danej akcji. Drugi z nich obsługuje funkcje dostępne bez logowania, tj. składanie zgłoszeń na kolędę oraz przeglądanie panelu zgłoszenia.

Widoki Kolejnym rodzajem komponentu są **Widoki**, które zawierają interfejs użytkownika aplikacji, prezentowany w przeglądarce. Widoki są tworzone przy użyciu technologii Razor, która umożliwia dynamiczne generowanie stron HTML na podstawie danych z modelu. Podobnie jak kontrolery, podzielone są na dwie kategorie: widoki dla zalogowanych użytkowników oraz widoki publiczne dla parafian.

Podczas gdy widoki publiczne są prostsze i bardziej statyczne (głównie formularze i strony informacyjne), widoki dla zalogowanych użytkowników są często bardziej rozbudowane i interaktywne, oferując zaawansowane funkcje zarządzania kolędą. W sekcji 4.1.1. opisano podejścia wykorzystane do tworzenia interfejsu użytkownika w widokach aplikacji webowej.

Modele W architekturze MVC modele są przekazywane z kontrolerów do widoków, aby prezentować dane użytkownikowi. W opisywanej aplikacji jest to każdorazowo realizowane za pomocą jednego ze specjalnych typów modeli:

- **ViewModels** (*modele widoków*) — specjalne modele zaprojektowane do reprezentowania danych w widokach. Mogą zawierać dodatkową logikę prezentacyjną lub formatowanie danych.
- **DTO (Data Transfer Objects)** — proste obiekty, pochodzące bezpośrednio z warstwy aplikacji (por. sekcja 4.4.2.), które często są wystarczające do prezentacji danych.
- **Encje EF Core** — bezpośrednie modele danych z warstwy domeny. Stosowane głównie w prostych widokach, gdzie nie jest wymagana dodatkowa logika prezentacyjna.

Logika biznesowa Logika biznesowa aplikacji jest zaimplementowana w warstwie aplikacji (por. sekcja 4.4.2.) i jest wykorzystywana przez kontrolery aplikacji przy użyciu **Serwisów aplikacji**, reprezentowanych na diagramie (rys. 4.6) przez komponent *Usługi*. Serwisy aplikacji zawierają metody do wykonywania operacji biznesowych, takich jak zarządzanie zgłoszeniami, planowanie wizyt czy wysyłanie powiadomień e-mail. Kontrolery aplikacji nigdy nie komunikują się bezpośrednio z warstwą domeny lub infrastrukturą, lecz zawsze poprzez serwisy aplikacji.

Dostęp do danych Dostęp do danych w aplikacji jest realizowany za pomocą wzorca **Repozytoriów**, które są reprezentowane na diagramie przez komponent *Repozytoria*. Repozytoria zapewniają abstrakcję nad warstwą dostępu do danych, umożliwiając kontrolerom i serwisom aplikacji interakcję z bazą danych bez konieczności bezpośredniego korzystania z ORM (Entity Framework Core). Repozytoria zawierają metody do wykonywania operacji CRUD (tworzenie, odczyt, aktualizacja, usuwanie) na encjach domeny.

Repozytoria również są zebrane w jednym komponencie. W tym przypadku jednak ma to dodatkowy wymiar, ponieważ wszystkie one są dostępne poprzez jeden obiekt `UnitOfWork`, który dodatkowo koordynuje transakcje i zapewnia spójność danych.

Aplikacja API

Aplikacja API działa w podobny sposób jak aplikacja webowa, jednak jej głównym celem jest udostępnianie funkcji aplikacji w formie usług sieciowych (dostępnych poprzez protokół HTTP). Diagram komponentów aplikacji API został zaprezentowany na rys. 4.7.

Podobnie jak w aplikacji webowej, głównymi komponentami są tutaj **Kontrolery**, **Serwisy aplikacji** oraz **Repozytoria**. Kontrolery API obsługują żądania HTTP od klientów API (widoków aplikacji) i wykonują operacje biznesowe za pomocą serwisów aplikacji. Serwisy aplikacji i repozytoria działają identycznie jak w

aplikacji webowej, zapewniając abstrakcję nad logiką biznesową i dostępem do danych. Jedyną istotną różnicą jest brak części publicznej, ponieważ aplikacja API jest przeznaczona wyłącznie do użytku przez widoki aplikacji webowej dla zalogowanych użytkowników.

Aplikacja API udostępnia jedynie wybrane funkcje aplikacji, które bezpośrednio wspierają interfejs użytkownika. Z tego powodu nie wszystkie serwisy aplikacji i repozytoria są dostępne poprzez API. Również punkty końcowe często są dostępne tylko poprzez niektóre metody HTTP (spośród GET, POST, PUT, PATCH, DELETE).

4.4. DDD i warstwy systemu

Aplikację zaprojektowano i zaimplementowano w podejściu Domain-Driven Design (DDD), tworząc warstwową architekturę, która składa się z następujących poziomów:

- Warstwa domeny (Domain Layer)
- Warstwa aplikacji (Application Layer)
- Warstwa infrastruktury (Infrastructure Layer)
- Warstwa interfejsu użytkownika (UI Layer)

Każdy z nich jest odpowiedzialny za określone aspekty aplikacji i komunikuje się z innymi warstwami w sposób jasno zdefiniowany.

4.4.1. Warstwa domeny

Najniższą warstwą w architekturze jest **Warstwa domeny**, która zawiera wszystkie elementy związane z modelem domeny aplikacji. Umieszczone zostały tutaj głównie encje domenowe (por. 4.2.2.). Warstwa domeny jest niezależna od innych warstw i nie zawiera żadnych odwołań do technologii zewnętrznych (np. baz danych, frameworków webowych itp.).

4.4.2. Warstwa aplikacji

Warstwa aplikacji znajduje się powyżej warstwy domeny i jest odpowiedzialna za implementację logiki biznesowej aplikacji. Zawiera serwisy aplikacji, które realizują operacje biznesowe, korzystając z encji domeny. Warstwa aplikacji komunikuje się z warstwą domeny poprzez bezpośrednie odwołania do encji oraz z warstwą infrastruktury poprzez repozytoria. Jest wykorzystywana przez warstwę interfejsu

użytkownika i najczęściej przekazuje do niej dane w formie DTO (Data Transfer Objects).

4.4.3. Warstwa infrastruktury

Warstwa infrastruktury zapewnia implementację techniczną dla aplikacji. Zawiera realizację repozytoriów, które umożliwiają dostęp do bazy danych oraz inne komponenty infrastrukturalne, w tym komponent do wysyłania wiadomości e-mail czy komponent autoryzacji. Główną odpowiedzialnością warstwy infrastruktury jest izolowanie pozostałych warstw od szczegółów technicznych, takich jak konkretna baza danych czy protokoły komunikacyjne.

4.4.4. Warstwa UI

Najwyższym poziomem w architekturze jest **Warstwa interfejsu użytkownika**, która zawiera komponenty odpowiedzialne za interakcję z użytkownikami. W naszym przypadku są to aplikacja webowa oraz aplikacja API. Warstwa UI komunikuje się z warstwą aplikacji, korzystając z serwisów aplikacji i przekazuje dane do prezentacji użytkownikowi do widoków.

4.5. Architektura wdrażania

Jak wspomniano w sekcji 4.1.3. aplikacja została zaopatrzona w stosowny plik `Dockerfile`, który pozwala na zbudowanie obrazu aplikacji. Na jego podstawie można następnie tworzyć kontenery Dockera.

W projekcie umieszczono również plik `docker-compose.yml` (wraz z kilkoma odmianami), który pozwala na uruchomienie całego środowiska aplikacji (z aplikacją, bazą danych i menadżerem ruchu) za pomocą jednego polecenia z uwzględnieniem opcji konfiguracyjnych za pomocą orkiestratora Docker Compose.

Na diagramie wdrażania (rys. 4.8) zaprezentowano strukturę środowiska aplikacji. Zaznaczone są na nim trzy wyżej wymienione usługi: aplikacja webowa, baza danych oraz menadżer ruchu (reverse proxy).

Konteneryzacja

Wszystkie trzy usługi są uruchamiane w oddzielnych kontenerach Dockera, co pozwala na ich izolację i łatwe zarządzanie. Konteneryzacja umożliwia również łatwe skalowanie aplikacji oraz przenoszenie jej między różnymi środowiskami (deweloperskim, testowym, produkcyjnym).

Przy zmianach w kodzie aplikacji lub jej konfiguracji wystarczy więc zbudować nowy obraz Dockera i uruchomić nowy kontener. Zachowuje się w ten sposób izolację od bazy danych, która nie odnotowuje żadnych przerw w działaniu.

Zarządzanie siecią

W środowisku aplikacji zastosowano również wirtualną sieć Dockera, która pozwala na zachowanie bezpośredniej komunikacji między kontenerami przy pełnej ich izolacji od sieci zewnętrznej. Kontenery mogą się ze sobą komunikować za pomocą nazw usług (np. `database` dla bazy danych), co upraszcza konfigurację połączeń.

Jedynym punktem dostępu do aplikacji z zewnątrz jest menadżer ruchu, który przekazuje żądania do aplikacji webowej. Pozwala to na dodatkowe zabezpieczenie aplikacji oraz na łatwe zarządzanie ruchem sieciowym (np. poprzez konfigurację certyfikatów SSL/TLS).

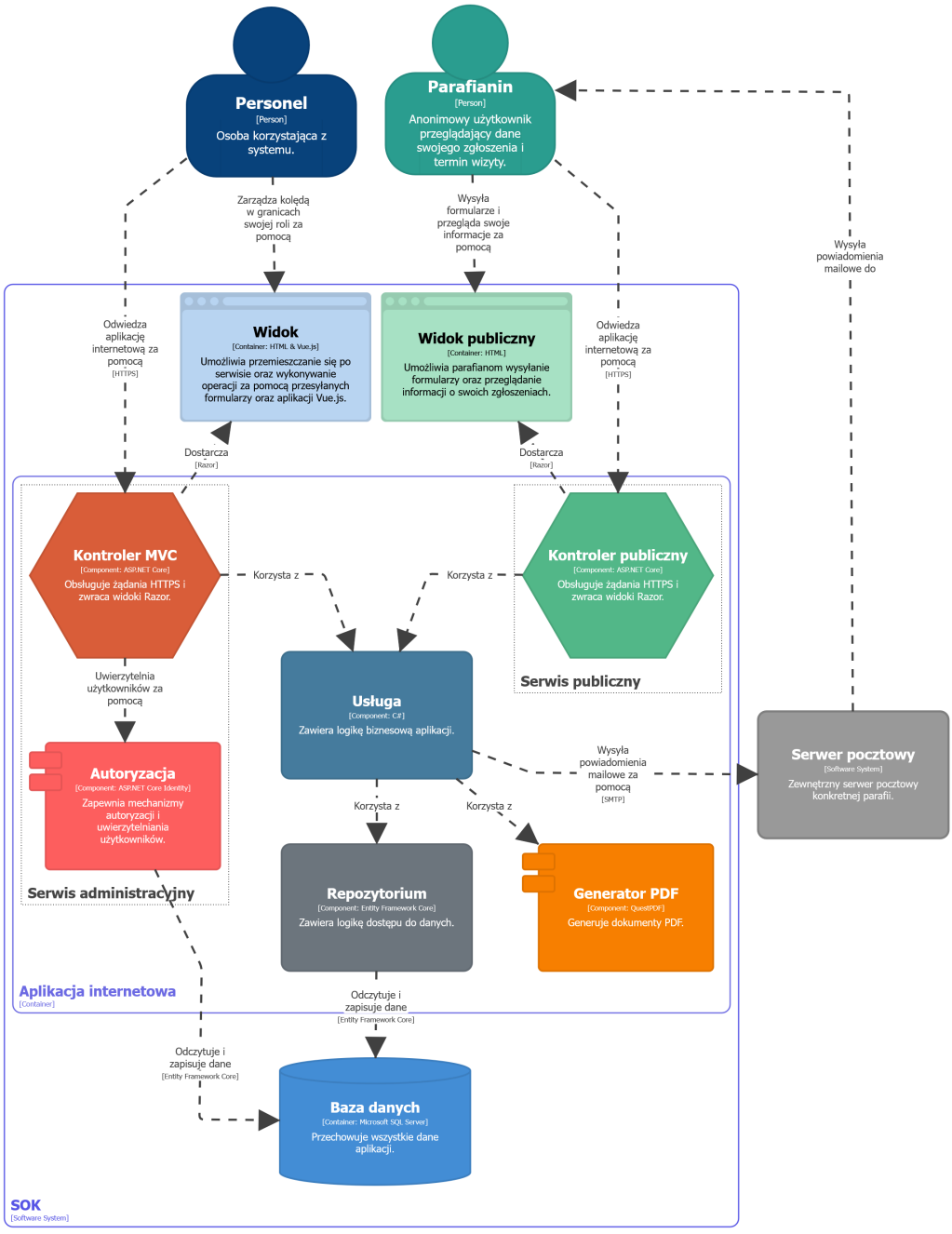
Trwałość danych

Baza danych jest skonfigurowana z użyciem wolumenu Dockera, który zapewnia trwałość danych pomiędzy restartami kontenera. Oznacza to, że dane przechowywane w bazie danych nie zostaną utracone w przypadku ponownego uruchomienia kontenera lub aktualizacji aplikacji. Wolumen jest mapowany na lokalny katalog na hoście, co pozwala na łatwe tworzenie kopii zapasowych i zarządzanie danymi bazy.

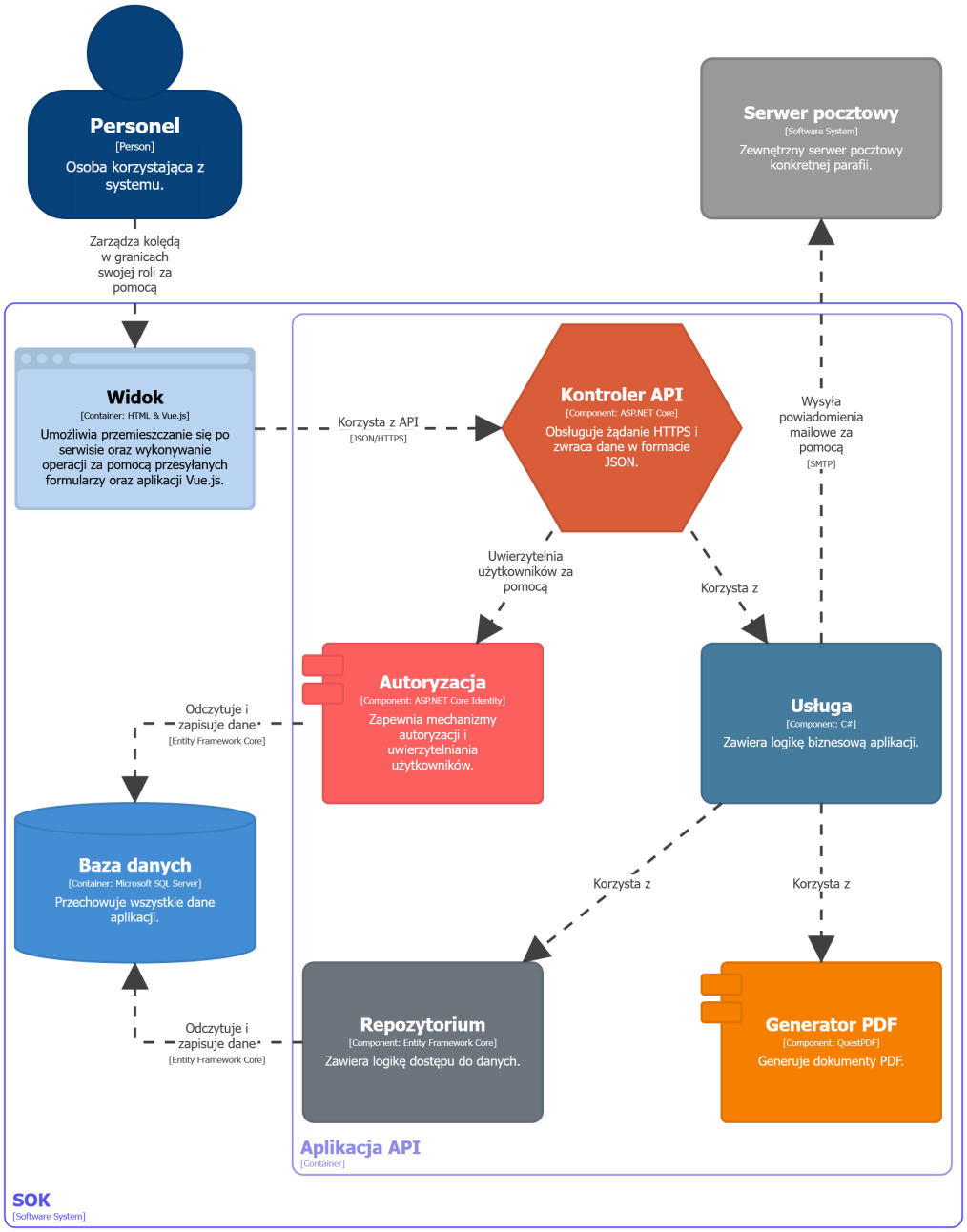
Środowisko produkcyjne

Dzięki zastosowaniu powyższego zestawu technologii i podejść architektonicznych aplikacja jest łatwa do wdrożenia i utrzymania na dowolnym serwerze obsługującym Dockera, w szczególności na popularnych platformach chmurowych. Nie jest jednak do nich ograniczone — cały stos technologiczny może być uruchomiony na dowolnym serwerze fizycznym lub wirtualnym, do którego użytkownik ma dostęp, za pomocą jednego polecenia (bez konieczności manualnego konfigurowania środowiska — instalowania oprogramowania i jego zależności).

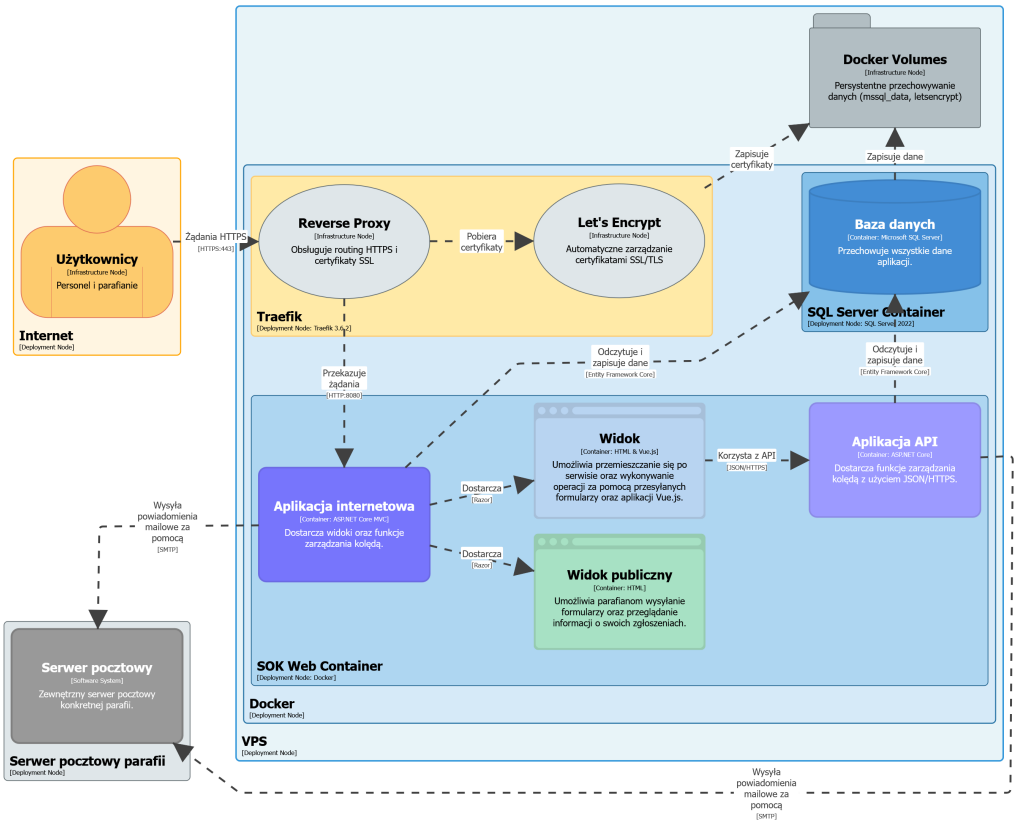
4.6. Opis rozwiazania wybranych problemów



Rysunek 4.6: Diagram komponentów aplikacji webowej



Rysunek 4.7: Diagram komponentów aplikacji API



Rysunek 4.8: Diagram wdrażania aplikacji

Rozdział 5.

Instrukcja użytkownika (+ zrzuty ekranów)

Rozdział 6.

Dla programistów

6.1. Instrukcja instalacji

6.2. Statystyki, testy jednostkowe

Rozdział 7.

Podsumowanie i dalszy rozwój