POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: Informatyka (INF)
SPECJALNOŚĆ: Inżynieria internetowa (INT)

PRACA DYPLOMOWA INŻYNIERSKA

Aplikacja webowa udostępniająca informacje o podziałach administracyjnych Śląska

Web application that provides information about administrative division of Silesia

AUTOR:
Marcin Bajak

PROWADZĄCY PRACĘ:
dr inż. Roman Ptak
K30-W04-D03

WROCŁAW 2020

Chciałbym serdecznie podziękować mojemu Promotorowi, doktorowi Romanowi Ptakowi za wyrozumiałość oraz wsparcie merytoryczne podczas procesu tworzenia tej pracy.

Spis treści

Wstęp

Wprowadzenie

Niniejsza praca dyplomowa ma na celu przedstawienie procesu tworzenia aplikacji udostępniającej dane o podziałach administracyjnych Śląska.

Cel i zakres pracy

Celem pracy jest zaprojektowanie, implementacja i udokumentowanie bazy danych oraz aplikacji webowej udostępniającej możliwość odczytu i modyfikacji danych dotyczących podziału administracyjnego Śląska w latach 1946—2020.

Baza danych pozwala na przechowywanie informacji o jednostkach administracyjnych takich jak województwa, powiaty czy gminy, w okresie od czasów powojennych, aż po dni dzisiejsze. Ponadto realizuje to w taki sposób, aby jednostki oraz relacje pomiędzy nimi posiadały wymiar czasowy. Dzięki temu możliwe jest zwrócenie w aplikacji informacji o stanie podziału administracyjnego na dowolny punkt czasowy w obrębie przechowywanych danych. Źródłem danych dla bazy jest zbiór arkuszy kalkulacyjnych, zawierających stany podziału administracyjnego w określonych punktach czasowych. Drugim celem projektu jest napisanie aplikacji webowej pośredniczącej w odczycie i modyfikacji danych znajdujących się w bazie.

Prezentacja pracy

W pierwszym rozdziale omówiono jednostki podziału administracyjnego Polski oraz pojęcia z nimi związane, które są podstawą dla całego projektu. Następnie przedstawiono technologie zastosowane do realizacji tego projektu oraz krótkie podsumowanie założeń i celu projektu.

W rozdziale drugim omówiono procesy modelowania relacyjnej bazy danych, referencyjnego zbioru danych oraz samej aplikacji. Następnie przedstawiono przypadki użycia dla opracowanego modelu aplikacji.

Kolejny rozdział przedstawia szczegóły implementacji — zarówno w obszarze bazy danych, jak i przeprowadzonych transformacji na danych wejściowych, celem zasilenia bazy danych.

W kolejnym rozdziale omówiono szczegóły implementacyjne dotyczące aplikacji webowej udostępniającej dane, fazę testów oraz otrzymanych wyników przypadków użycia.

W ostatnim rozdziale przedstawiono podsumowanie projektu — omówiono napotkane trudności, wnioski wyciągnięte z procesu realizacji projektu oraz pokazano możliwości jego rozwoju.

Rozdział 1

Przedmiot projektu - historia zmian w układzie administracyjnym i opis wykorzystanych technologii

1.1 Historia podziału administracyjnego

1.1.1 Wprowadzenie

Podział administracyjny występuje w większości państw świata i służy usprawnieniu zarządzania państwem poprzez decentralizację, pozwalając na zarządzanie mniejszymi regionami państw. Mogą one posiadać specyficzne dla siebie własności, takie jak uwarunkowania geograficzne, potencjał gospodarczy, wydobywczy lub turystyczny.

W Polsce podział terytorialny oraz mnogość rodzajów jednostek ulegała wielu zmianom na przestrzeni lat. Przedmiotem projektu jest podział administracyjny na terenach Polski, a w szczególności krainy geograficznej Śląsk od roku 1946 do czasów obecnych.

1.1.2 Pierwszy podział administracyjny

Po zakończeniu II Wojny Światowej, w roku 1945 dokonano wstępnego podziału ziem niemieckich, które dołączono do Polski (tzw. Ziemie Odzyskane): Śląsk Opolski, Dolny Śląsk, Pomorze Zachodnie oraz Mazury. W dniu 28 czerwca 1946 roku przywrócono trójstopniowy podział administracyjny z czasów przedwojennych. Na obszarze „Ziem Odzyskanych" utworzono 3 nowe województwa: olsztyńskie, szczecińskie i wrocławskie.

Finalnie poskutkowało to następującą strukturą:

- największe jednostki podziału administracyjnego, czyli województwa, których w danym czasie było 14,
- powiaty, które były strukturami pośrednimi pomiędzy jednostkami najniższego szczebla a województwami, w liczbie 299,
- gminy, będące najmniejszymi jednostkami podziału administracyjnego, przynależąc do określonego powiatu, a więc pośrednio województwa. W danym czasie istniało 3006 gmin wiejskich oraz 703 miasta.

Istniały również miasta na prawach województwa - Łódź oraz Warszawa - wydzielone z otaczających je województw odpowiednio Łódzkiego i Warszawskiego.

Na rysunku 1.1 przedstawiono pierwszy podział administracyjny na województwa oraz miasta wydzielone Polski.

Rysunek 1.1 Podział administracyjny Polski – 1946 rok [6]

1.1.3 Istotne zmiany pomiędzy 1946 a 1976

Dnia 6 lipca 1950 wydzielono 3 nowe województwa (koszalińskie, opolskie, zielonogórskie), a łączna liczba województw wynosiła 17.

Z dniem 1 stycznia 1957 roku miasta Kraków, Poznań i Wrocław otrzymały rangę miast na prawach województwa.

Na rysunku 1.2 przedstawiono stan podziału administracyjnego na województwa i miasta wydzielone Polski na rok 1957.

Rysunek 1.2 Podział administracyjny Polski – 1957 rok [6]

1.1.4 Dwustopniowy podział administracyjny

Do roku 1975 podział terytorialny na wyższych szczeblach utrzymywał się prawie bez większych zmian. Na niższym poziomie w okresie 1954–1972 zniesiono gminy i wprowadzono gromady. Po niespełnieniu zakładanych celów reformy, tj. chęci zbliżenia „władzy ludowej" do mas chłopskich [1] gminy przywrócono z powrotem.

W dniu 1 czerwca 1975 roku podział administracyjny całkowicie zmieniono: trójstopniowy podział zastąpiono dwustopniowym, zlikwidowano powiaty, a w zamian wprowadzono znacznie większą liczbę województw, która urosła do 49. Nazwy województw bazowały na nazwach miast, będących siedzibami województw. Do listy miast na prawach województwa dodano Kraków.

Zmiana podziału administracyjnego województw w roku 1975 przedstawiono na rysunku 1.3.

Rysunek 1.3 Podział administracyjny Polski – 1975 rok [5]

1.1.5 Powrót do trójstopniowego podziału

Z dniem 1 stycznia 1999 przywrócono podział trójstopniowy, ponownie wprowadzając powiaty, ograniczając liczbę województw i tym samym powracając niemalże do stanu sprzed podziału dwustopniowego. Zlikwidowano miasta wydzielone. Nazwy i lokalizację nowo powołanych województw oparto na krainach oraz regionach historycznych i geograficznych Polski: Podlasie — województwo podlaskie, Małopolska — województwo małopolskie, Śląsk — województwo śląskie, Dolny Śląsk — województwo dolnośląskie itd. Obecna struktura jest bardzo podobna do tej, która występowała przed reformą 1975 roku w prawie niezmienionych granicach: utworzono 16 województw oraz 315 powiatów. Obecnie liczba powiatów wynosi 314, gmin — 2477, istnieje również 65 miast na prawach powiatu.

Obecnie panujący trójstopniowy podział Polski na województwa zaprezentowano na rysunku 1.4.

Rysunek 1.4 Podział administracyjny Polski – 1999 rok [19]

Informacja o współczesnym podziale jest dostępna na stronie internetowej Głównego Urzędu Statystycznego w postaci plików udostępnianych przez rządowy serwis TERYT [18].

1.2 Śląsk oraz tereny dawnego województwa Bielskiego

Dotychczas omówione zmiany w podziale administracyjnym naturalnie przekładają się na tereny Śląska, ponieważ prawie w całości znajduje się na terenie III Rzeczpospolitej [11].

Z uwagi na złożoność i mnogość jednostek administracyjnych na terenie Śląska oraz stosunkowo czasochłonny proces dodawania danych inicjalnych, inicjalny zestaw danych ograniczono do terenów dawnego, tj. istniejącego w latach 1975—1998, województwa bielskiego, stanowiącego zaledwie ułamek terenów Śląska. Ograniczono również wymiar czasu — początek stanowi moment wprowadzenia dwustopniowego podziału administracyjnego 1 czerwca 1976 roku, a koniec przypada na dzień 1 stycznia 2020 roku.

Pomimo ograniczenia stanu inicjalnego bazy danych, w projekcie założono możliwość rozszerzenia funkcjonalności na teren całej Polski oraz możliwość uwzględnienia zakresu czasowego od momentu wprowadzenia pierwszego podziału administracyjnego trzeciej Rzeczypospolitej.

1.3 Zakres projektu

Na podstawie dostarczonych przez Promotora referencyjnych danych o jednostkach podziału administracyjnego Śląska, w dalszej części pracy przedstawiono proces modelowania oraz szczegóły implementacji i fazy testów bazy danych oraz aplikacji webowej. Podstawowym celem aplikacji jest udostępnianie danych przechowywanych w bazie danych zarówno bezpośrednim użytkownikom aplikacji w formie dokumentów HTML, jak

i użytkownikom innych aplikacji, mogącym wykorzystać zwracane przez aplikacje dane zapisane w formacie JSON.

1.4 Przegląd wykorzystanych technologii

1.4.1 Dane referencyjne

Otrzymane od Promotora dane o podziałach administracyjnych Śląska na przestrzeni omówionych lat były zapisane w formie arkuszy kalkulacyjnych. Struktura danych w arkuszach dalece odbiegała od struktury danych w opracowanym modelu bazy danych. Przyjmowała formę podziału administracyjnego na konkretne punkty w czasie, w przeciwieństwie do zamodelowanej ciągłości wymiaru czasu przyjętej w modelu bazy danych. Aby móc w prawidłowy sposób przechować te dane w bazie danych, część transformacji na danych przeprowadzono przy pomocy formuł programu Microsoft Office Excel.

1.4.2 Baza danych

Definicja 1.1 (Baza danych) *Baza danych to zorganizowany zbiór ustrukturyzowanych informacji, czyli danych, zwykle przechowywany w systemie komputerowym w formie elektronicznej. Bazą danych steruje zwykle system zarządzania bazami danych (DBMS). [12]*

W projekcie do utworzenia i zarządzania serwerem i bazą danych oraz inicjalnego dodania zasobów zastosowano technologię Microsoft SQL Server. Z technologii tej skorzystano za pomocą programu Microsoft SQL Server Management Studio, pozwalającego — za pomocą języka T-SQL — na utworzenie instancji bazy danych na serwerze, obiektów w bazie danych oraz zasilenie tabel danymi inicjalnymi.

1.4.3 Aplikacja webowa

Definicja 1.2 (HTTP) *Protokół Przesyłania Danych Hipertekstowych (Hypertext Transfer Protocol, HTTP) to protokół warstwy aplikacji, odpowiedzialny za transmisję dokumentów hipermedialnych, jak np. HTML. Został stworzony do komunikacji pomiędzy przeglądarkami, a serwerami webowymi, ale może być używany również w innych celach. HTTP opiera się na klasycznym modelu klient-serwer, gdzie klient inicjuje połączenie poprzez wysłanie żądania, następnie czeka na odpowiedź. HTTP jest protokołem bezstanowym, co oznacza, że serwer nie przechowuje żadnych danych (stanów) pomiędzy oboma żądaniami [10].*

Napisaną na potrzeby projektu aplikację webową oparto na protokole HTTP. Aplikacja pełni rolę serwera webowego, do którego klient może wysłać żądanie i otrzymać na nie odpowiedź. Aplikację podzielono na dwie części: odpowiadającą za graficzny interfejs dla użytkownika (ang. *GUI - Graphical User Interface*), pozwalając na odesłanie dokumentu HTML zrozumiałego dla popularnych przeglądarek internetowych oraz interfejs programistyczny aplikacji (ang. *API - Application Programming Interface*) (definicja 2.4), udostępniającego jednolity sposób dostępu do bazy danych innym aplikacjom webowym. Do napisania aplikacji wykorzystano język programowania Python [17], a konkretnie microframework Flask [2], który posiada wiele gotowych rozwiązań i implementacji potrzebnych do napisania w pełni funkcjonalnej aplikacji webowej. W procesie tworzenia aplikacji wykorzystano dodatkowe narzędzia — moduły Pythona takie jak:

- pyodbc [15] do inicjacji połączenia pomiędzy aplikacją a bazą danych,
- os oraz pandas [13] do operacji na zwracanych przez aplikację dokumentach HTML,
- simplejson [16] do konwersji zwracanych przez aplikację danych do formatu JSON.

1.4.4 Testy

Na potrzeby weryfikacji działania systemu pod względem poprawności, jakości, powtarzalności oraz funkcjonalności przeprowadzono testy jednostkowe, funkcjonalne lub wydajnościowe z wykorzystaniem technologii:

- Microsoft Edge,
- Google Chrome,
- PostMan [14],
- SQL,
- Microsoft Server SQL Management Studio [9].

Pierwsze 3 technologie pozwalają na wysyłanie do aplikacji żądań HTTP oraz weryfikację poprawności zwracanych odpowiedzi na żądania, zarówno w przypadku interfejsu graficznego skierowanego dla użytkowników aplikacji poprzez przeglądarki internetowe, jak i interfejsu programistycznego aplikacji, skierowanego do innych aplikacji webowych i ich użytkowników. Bazę danych przetestowano dwoma ostatnimi technologiami, poprzez pisanie zapytań oraz wyświetlanie planów zapytań i licznika czasu wykonania zapytania w programie SSMS.

Rozdział 2

Modelowanie

Zdefiniowany w poprzednim rozdziale cel projektu można rozłożyć na następujące etapy projektowania:

- analiza zbioru danych inicjalnych,
- określenie przypadków użycia aplikacji,
- opracowanie modelu bazy danych,
- zamodelowanie aplikacji webowej,
- przedstawienie schematu testów aplikacji.

W poniższym rozdziale opisano strukturę otrzymanych danych inicjalnych oraz przedstawiono model bazy danych. Następnie pokazano proces projektowania aplikacji webowej oraz projekt etapu testów funkcjonalności.

2.1 Dane inicjalne

W tabeli 2.1 przedstawiono wybrane wiersze z otrzymanych arkuszy kalkulacyjnych. Dane ograniczono do województwa bielskiego, ponadto wybrano rekordy, które różniły się od siebie w znaczny sposób zapisem.

Każdy rekord odpowiada danej jednostce administracyjnej, w tym przypadku gminom, które dla przypomnienia w latach 1975-1998 należały bezpośrednio do województw. Kolumny to kolejno liczba porządkowa gminy z arkusza, województwo do której przynależała gmina, pierwszy punkt czasowy, uwagi dotyczące gmin oraz 6 kolejnych punktów w czasie. Ostatnia kolumna w tabeli mówi o rodzaju gminy.

W kolumnach z punktami czasowymi, wartością z nazwą gminy oznaczano fakt, że gmina w danym punkcie czasowym należy do województwa. Pola puste oznaczają sytuacje przeciwną (z wykluczeniem rekordu z gminą Bielsko Biała). Wartość 'x' oznacza, że dana gmina od daty w nagłówku już nie istnieje. Moment przeniesienia siedziby z jednej gminy do drugiej oznaczono przez [gmina] ➞[gmina]. Dodatkowo w celu wizualizacji, komórki które oznaczały zmianę dotychczasowego stanu gminy oznaczane były kolorami: zielonym w przypadku powołania gminy, żółtym w sytuacji zmiany nazwy gminy oraz czerwonym w przypadku jej zniesienia bądź przeniesienia.

Tablica 2.1 Przykładowe dane dla województwa bielskiego w latach 1975–1998

lp.	Woj.	Gmina 1975	1975 w tym/Uwagi	Gmina 2 VII 1976	Gmina 1 I 1977	Gmina 1 IV 1990	Gmina 2 IV 1991	Gmina 1 I 1992	Gmina 31 XII 1997	Rodzaj
1	bielskie	Bielsko-Biała							Bielsko-Biała	miasto
6	bielskie	Kęty	Kęty	Kęty	Kęty	Kęty	Kęty	x		miasto
19	bielskie	Andrychów		Andrychów	Andrychów	Andrychów	Andrychów	Andrychów	Andrychów	wieś
21	bielskie	Brody	2 lipca 1976 roku siedziba gminy została przeniesiona z Brodów do miasta Kalwaria Zebrzydowska, z jednoczesną zmianą nazwy gminy na gmina Kalwaria Zebrzydowska	Brody → Kalwaria Zebrzydowska						wieś
23	bielskie	Buczkowice		Buczkowice →Szczyrk		Szczyrk → Buczkowice	Buczkowice	Buczkowice	Buczkowice	wieś
25	bielskie	Chełmek	m. Chełmek	Chełmek	Chełmek	Chełmek	Chełmek	Chełmek	Chełmek	wieś
29	bielskie	Gilowice	2 lipca 1976 r. gmina Gilowice-Ślemień utworzona przez połączenie gmin Gilowice i Ślemień oraz dołączenie zniesionej gminy Łękawica (a więc w praktyce utworzona z trzech zniesionych gmin)	x			Gilowice	Gilowice	Gilowice	wieś
30	bielskie		2 kwietnia 1991 gmina Gilowice-Ślemień została zlikwidowana, a z jej obszaru odtworzono dawne gminy Gilowice, Ślemień i Łękawica sprzed komasacji. Jedynie wieś Oczków ze znoszonej gminy przyłączono do Żywca	Gilowice-Ślemień	Gilowice-Ślemień	Gilowice-Ślemień	x			wieś
35	bielskie						Jaworze	Jaworze	Jaworze	wieś
36	bielskie	Jeleśnica		Jeleśnia	Jeleśnia	Jeleśnia	Jeleśnia	Jeleśnia	Jeleśnia	wieś
37	bielskie			Kalwaria Zebrzydowska	Kalwaria Zebrzydowska	Kalwaria Zebrzydowska	Kalwaria Zebrzydowska	Kalwaria Zebrzydowska	Kalwaria Zebrzydowska	wieś
39	bielskie	Komorowice	1 stycznia 1977 roku jednostka została zniesiona przez włączenie jej terenów do miasta Bielska-Białej	Komorowice	x					wieś
44	bielskie	Łękawica	2 lipca 1976 r. gmina Gilowice-Ślemień utworzona przez połączenie gmin Gilowice i Ślemień oraz dołączenie zniesionej gminy Łękawica (a więc w praktyce utworzona z trzech zniesionych gmin)	x			Łękawica	Łękawica	Łękawica	wieś
45	bielskie	Łodygowica		Łodygowice	Łodygowice	Łodygowice	Łodygowice	Łodygowice	Łodygowice	wieś
51	bielskie							Polanka Wielka	Polanka Wielka	wieś
58	bielskie	Stare Bielsko	1 stycznia 1977 roku gmina została zniesiona przez włączenie sołectwa Stare Bielsko do miasta Bielsko-Biała	Stare Bielsko	x					wieś
62	bielskie		1 kwietnia 1990 siedziba gminy została przeniesiona z powrotem do Buczkowic, z jednoczesną zmianą nazwy gminy na gmina Buczkowice	Szczyrk	Szczyrk	x				wieś
63	bielskie	Ślemień	2 lipca 1976 r. gmina Gilowice-Ślemień utworzona przez połączenie gmin Gilowice i Ślemień oraz dołączenie zniesionej gminy Łękawica (a więc w praktyce utworzona z trzech zniesionych gmin)	x			Ślemień	Ślemień	Ślemień	wieś
68	bielskie	Wapienica	1 stycznia 1977 roku została zniesiona przez włączenie sołectwa Wapienica do miasta Bielsko-Biała	Wapienica	x					wieś

Na podstawie powyższych przykładów gmin, zbadano i zauważono pewne prawidłowości:

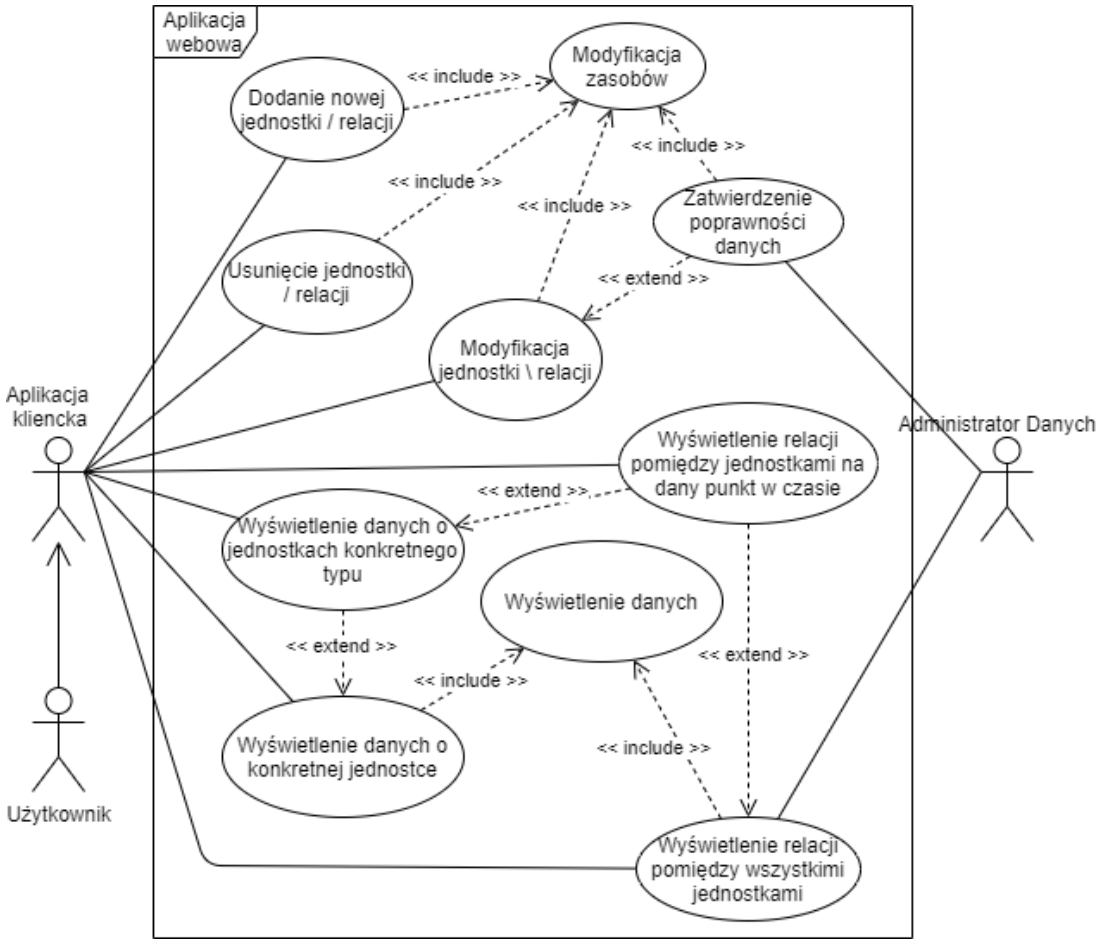
- liczba oraz nazwy gmin przynależących, w tym konkretnym przypadku, do województwa bielskiego ulegają zmianom w czasie,
- gmina może zostać powołana lub zniesiona w dowolnym punkcie w czasie istnienia województwa,
- gmina może przynależeć do wyższej jednostki przez pewien okres, następnie na pewien okres zostać zniesiona, a następnie znowu zostać przywrócona,
- możliwa jest sytuacja, w której w tym samym czasie istniało wiele gmin o tej samej nazwie, które powstałyby w tym samym punkcie w czasie,
- na dany punkt w czasie może istnieć więcej niż jedna gmina o tej samej nazwie, przynależąc do tej samej jednostki wyższego stopnia, różniąc się rodzajem (miasto a wieś / gmina miejska a wiejska),
- gminy wchodzą w relacje przynależności z jednostkami wyższego typu, ale mogą również być w relacji z innymi gminami (przykładowo poprzez przeniesienie siedziby lub przejęcie terenów jednej gminy przez drugą). Nie jest to krytycznie ważne dla zachowania spójności danych, ale stanowi informację, która dla użytkowników może okazać się istotna.

2.2 Przypadki użycia

Posiadając wiedzę jak wyglądają dane i biorąc pod uwagę ich znaczenie, wyznaczono przypadki użycia aplikacji obsługującej te dane. Założeniem projektu jest udostępnienie użytkownikom funkcjonalności CRUD na zasobach w bazie danych. Skrót funkcjonalności CRUD można rozwinąć na: ***C****reate* — utworzenia, ***R****ead* — odczytu, ***U****pdate* — aktualizacji oraz ***D****elete* — usunięcia zasobów.

Zakres projektu zakłada dwa zbiory przypadków użycia. Pierwszy należy rozumieć jako aplikację REST API, która pozwala na uniwersalny dostęp do danych programistom innych aplikacji (i pośrednio ich użytkownikom), i jest głównym przedmiotem projektu.

Na potrzeby projektu utworzono również interfejs graficzny, który jest skierowany bezpośrednio do użytkowników, co stanowi osobny zbiór przypadków użycia.



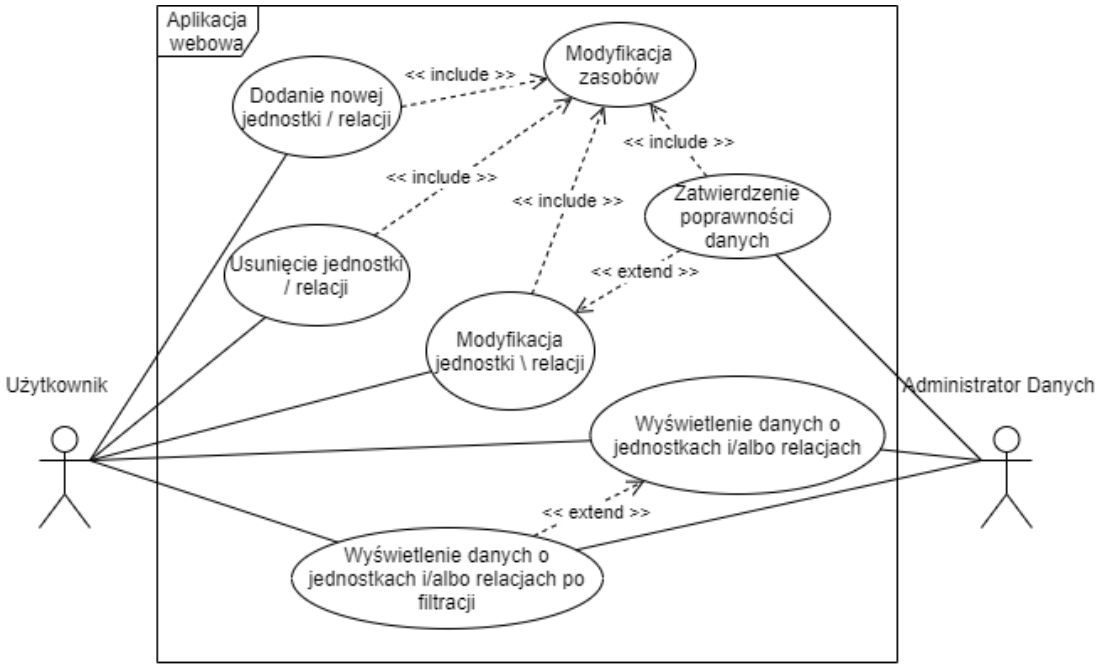
Rysunek 2.1 Diagram przypadków użycia REST API

Dla pierwszego zbioru przypadków użycia, diagram przypadków użycia przedstawiono na rysunku 2.1. Użytkownik, korzystając z aplikacji klienckiej wysyłającej żądania HTTP do aplikacji omawianej w tej pracy dyplomowej, może dokonać operacji CRUD na danych zawierających się w bazie danych.

Poprzez aktora nazwanego 'Aplikacja Kliencka', diagram podkreśla wykorzystanie aplikacji REST API będącej przedmiotem tego projektu, pełnienia roli serwera dla użytkowników pozostałych aplikacji pełniących role klienta w rozumieniu definicji protokołu HTTP (definicja 1.2).

W procesie projektowania, zdecydowano się na powołanie roli administratora danych, który może, w przypadku udostępnienia możliwości modyfikacji danych innym użytkownikom, potwierdzać wprowadzone przez pozostałych użytkowników dane.

Niezależnie od uprawnień do edycji, użytkownicy posiadają możliwość uzyskania zasobów na kilka różnych sposobów, jak chociażby pytając o konkretny rodzaj jednostki, konkretną jednostkę bądź o stan podziału administracyjnego na podany punkt w czasie.



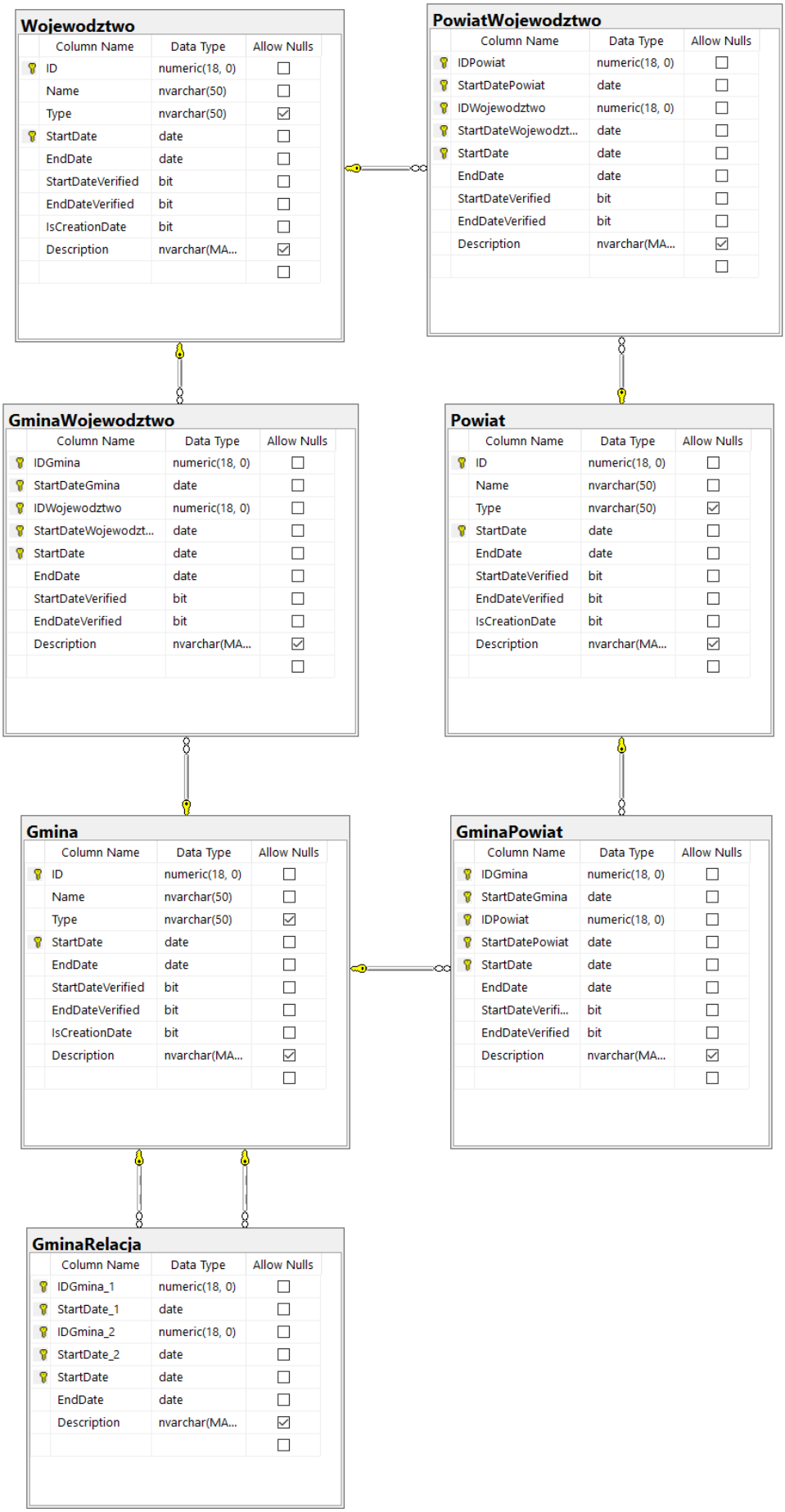
Rysunek 2.2 Diagram przypadków użycia interfejsu graficznego

Rysunek 2.2 obrazuje przypadki użycia w sytuacji korzystania z interfejsu graficznego przez użytkownika korzystające, w przeciwieństwie do poprzedniego przypadku, z przeglądarki internetowej. Użytkownik w tej sytuacji otrzymuje większą możliwość filtracji zwracanych przez aplikację danych poprzez określenie warunków równości wybranych kolumn z wprowadzonymi przez niego wartościami. Wciąż możliwym pozostaje odpytanie o zasoby na podany moment w czasie.

2.3 Model bazy danych

Definicja 2.1 (Temporalna baza danych) *Rodzaj bazy danych [definicja 1.1], która zawiera informacje o czasie ważności danych lub informację o czasie wprowadzenia bądź modyfikacji danych. Historyczna baza temporalna przechowuje tylko czas ważności danych, natomiast bitemporalna baza danych przechowuje zarówno czas ważności jak i informację o czasie wprowadzenia bądź modyfikacji danych. [4]*

Opracowany model bazy danych przedstawiony na rysunku 2.3 spełnia założenia historycznej bazy danych oraz posiada trzy rodzaje tabel, a ich struktura w obrębie rodzaju jest bliźniacza.



Rysunek 2.3 Model bazy danych

2.3.1 Encje

Tabele przechowujące encje (ang. *Entity*), czyli tabele:

- Gmina,
- Powiat,
- Wojewodztwo,

posiadają pola stanowiące klucz główny, będący kombinacją pól ID — identyfikator oraz StartDate — data rozpoczęcia obowiązywania rekordu.

Podstawowym założeniem bazy danych jest przechowywanie rozróżnialnych, jednoznacznie identyfikowalnych wpisów w tabelach przechowujących encje. Implikuje to konieczność utworzenia tzw. klucza sztucznego, identyfikatora, czyli kolumny przechowującej unikalną wartość numeryczną dla każdej z jednostek administracyjnych. Jednocześnie każda z jednostek administracyjnych może posiadać wiele rekordów w tabeli w rozłącznych okresach czasowych, chociażby w przypadku, gdy nazwa danej jednostki ulegnie zmianie. Pole ID odnosi się więc do konkretnej jednostki, a pole StartDate przechowuje informacje o początku obowiązywania wpisu w wymiarze czasu.

Następnie każda z jednostek posiada pole EndDate — data zakończenia obowiązywania danego rekordu, Name — nazwa, opcjonalną wartość Type — rodzaj, flagę IsCreationDate pozwalającą na określenie, czy wartość kolumny DataRozpoczecia w tym rekordzie, jest jednocześnie datą powstania samej jednostki administracyjnej. Dodatkowo tabele posiadają dwie flagi: StartDateVerified oraz EndDateVerified, co pozwala administratorowi danych na ręczną weryfikacje rekordów w przypadku upublicznienia możliwości dodawania kolejnych wpisów w bazie danych za pomocą aplikacji webowej szerszemu gronu użytkowników.

2.3.2 Tabele relacyjne

Drugi rodzaj tabel zastosowanych w modelu to tabele relacyjne (ang. *Cross Reference Table*), czyli tabele:

- GminaWojewodztwo,
- GminaPowiat,
- PowiatWojewodztwo,

przechowujące informacje o relacjach pomiędzy encjami różnych typów.

Biorąc pod uwagę specyfikę danych, czyli brak sytuacji, w której w tym samym momencie jakaś jednostka jest w wielu relacjach z konkretną jednostką innego typu, relacje pomiędzy różnymi encjami są również unikalne i rozróżnialne. Osiągnięto to poprzez ustanowienie klucza głównego tabeli kombinacją kluczy obcych do tabel z encjami oraz pola StartDate, które w tym przypadku przechowuje punkt w czasie stanowiący początek relacji.

Jak same nazwy wskazują, przykładowo tabela GminaWojewodztwo pozwala na ukazanie relacji 'wiele do wielu' gmin oraz województw, innymi słowy, sytuacji gdy wiele gmin może wchodzić w relacje z jednym województwem, a jedna gmina może wchodzić w relację z wieloma województwami. Klucze główne tabel relacyjnych składają się z kluczy obcych do tabelach przechowujących encje, oraz z pola StartDate, które pozwala

na zamodelowanie sytuacji, gdy np. jedna gmina przez pewien czas zawiera się w danym województwie, następnie gmina jest znoszona — a więc relacja nie zachodzi, następnie gmina zostaje przywrócona i posiada ponowną relację do województwa. Pozostałe pola przedstawiają analogiczną zawartość jak w przypadku tabel podstawowych.

Trzecim rodzajem tabel są tabele relacyjne modelujące relację pomiędzy jednostkami tego samego typu (ang. *Self Many to Many Relation*). W przedstawionym modelu jedyną tabelą tego rodzaju jest tabela GminaRelacja. Klucz główny tej tabeli to dwa różne od siebie klucze główne tabeli podstawowej Gmina oraz pole StartDate, pozwalające na jednoznaczną identyfikacje rekordów oraz określenie relacji w wymiarze czasu. Tabela pozwala przechować informacje, o tym, że przykładowo gmina 'A' powstała z rozbicia gminy 'B' na dwie gminy: gminę 'A' oraz gminę 'C'.

2.3.3 Założenia bazy danych

Każda tabela posiada pola StartDate i EndDate, przechowujące informacje o przedziale czasowym obowiązywania rekordu — zarówno dla encji jak i dla relacji — z rozdzielczością do dni, spełniając tym samym założenia historycznej bazy danych. Implikuje to konieczność tzw. domykania rekordów poprzez uzupełnianie pola EndDate. Tabele są więc tabelami statycznymi, przechowującymi dane wolnozmienne.

Aby ułatwić rozwój aplikacji, w bazie danych utworzono widoki, czyli zapisane zapytania w języku SQL, zawierające logikę relacji zachodzące pomiędzy tabelami, co pozwala na wygodny odczyt danych z bazy. Logikę połączeń pomiędzy tabelami w widokach zapisano przy użyciu kluczy obcych w tabelach relacyjnych, wiążących dane pomiędzy tabelami bazowymi w wymiarze czasowym. Szczegóły implementacji widoków przedstawiono w rozdziale trzecim.

2.4 Aplikacja webowa

Bazując na podrozdziale 1.4.3, opisującym wykorzystanie technologii HTTP jako protokołu komunikacji pomiędzy użytkownikiem a aplikacją webową, w tej części pracy przedstawiono proces modelowania aplikacji webowej.

Definicja 2.2 (URI) *URI (z ang. Uniform Resource Identifier) jest standardem umożliwiającym łatwą identyfikację zasobów w sieci. Zapisany jest zazwyczaj łańcuchem znaków, zgodnym ze składnią określoną w standardzie. Łańcuch ten określa nazwę (URN) lub adres (URL) zasobu, identyfikowanego przez dany URI. [3]*

Definicja 2.3 (REST) *REST (z ang. Representational State Transfer) jest to styl architektury oprogramowania opierający się o zbiór wcześniej określonych reguł opisujących jak definiowane są zasoby, a także umożliwiających dostęp do nich. Został zaprezentowany przez Roya Fieldinga w 2000 roku. [7]*

Definicja 2.4 (API) *API (z ang. Application Programming Interface) udostępnia usługi lub dane dostarczane przez aplikację za pośrednictwem zestawu predefiniowanych zasobów, takich jak metody, obiekty lub identyfikatory URI. Korzystając z tych zasobów inne aplikacje mogą uzyskać dostęp do danych lub usług bez konieczności implementacji podstawowych obiektów i procedur. [8]*

Na podstawie przytoczonych definicji można stwierdzić, że REST to styl architektury definiujący sposób udostępniania zasobów przez aplikację, natomiast API są regułami określającymi sposoby komunikacji przez użytkownika (programistę) z aplikacją.

Aplikacja webowa, posiadając możliwość nawiązania połączenia z bazą danych, przechowuje zbiór predefiniowanych metod odpowiedzialnych za komunikację użytkownika z bazą danych. Metody te przechowują logikę związaną z funkcjonalnościami CRUD opisaną w poprzednim punkcie na zasobach. W przypadku odczytu, aplikacja formatuje odczytane z bazy dane do jednego z dwóch formatów: języka HTML, zrozumiałego dla przeglądarek internetowych a więc dla bezpośrednich użytkowników aplikacji, oraz do formatu JSON w przypadku komunikacji pomiędzy aplikacjami webowymi.

W przypadku zapisu do bazy danych, sytuacja wygląda odwrotnie: dane przesłane do aplikacji (wpisane przez użytkownika w graficznym interfejsie lub wysłane przez inną aplikację) są formatowane na instrukcje bazodanowe, celem wykonania ich w bazie danych, modyfikując tym samym dane w bazie.

Podczas tworzenia i użytkowania aplikacji REST API, do komunikacji wykorzystuje się metody protokołu HTTP (def. 1.2) oraz definiuje się składowe żądania, czyli wiadomości wysłanej za pomocą protokołu takie jak adres, nagłówek czy treść zapytania.

Do spełnienia założeń funkcjonalności CRUD wykorzystano 4 poniższe metody protokołu HTTP:

- **GET** – służy do pobierania danych, czyli odczytu zasobów, a więc będzie spełniał funkcjonalność ***R**ead*. W większości przypadków do obsługi zapytania wystarczy adres URL zasobu.
- **POST** – służy tworzeniu zasobów i przesyłaniu nowych danych, a więc ***C**reate*. W tym przypadku konieczne jest już stworzenie ciała (ang. *body*) żądania HTTP, w którym można będzie przekazać dane do REST API.
- **PUT** – również służy przesyłaniu danych, lecz najczęściej w celu aktualizacji zasobów, a więc ***U**pdate*. Tutaj również wymagane jest przesłanie danych w ciele żądania.
- **DELETE** – metoda służąca do usuwania zasobów. Spełnia funkcjonalność ***D**elete*, a do obsługi, tak jak w przypadku metody GET, zazwyczaj wystarczy adres URL zasobu.

Wykorzystując wyżej wymienione metody żądań HTTP oraz znając zdefiniowany zbiór adresów URL (potocznie endpoint'ów) funkcjonalności oferowanych przez aplikację, użytkownik posiada możliwość uzyskania dostępu do funkcjonalności CRUD na zasobach w bazie danych.

2.5 Testy

W projekcie, celem weryfikacji poprawności działania całego systemu, przewidziano fazę testów napisanego oprogramowania.

2.5.1 Transformacje danych referencyjnych oraz baza danych

Spójność danych referencyjnych w bazie, do której trafiły po transformacjach, zweryfikowano za pomocą języka SQL, pozwalającego na wyświetlanie w środowisku SSMS danych

w tabelach. Spójność danych przetestowano zarówno w przypadku encji jak i wiążących ich relacji. Ponadto przeprowadzono testy wydajnościowe sprawdzające responsywność, czyli czas wykonania zapytań na bazie danych.

2.5.2 API

Część projektu poświęcona aplikacji stanowiącej interfejs programistyczny przetestowano za pomocą narzędzia PostMan, pozwalającego wysyłać na podany adres URL żądania HTTP, o określonych metodach, nagłówkach oraz ciałach. Zbadano czasy odpowiedzi aplikacji oraz poprawność wykonywanych metod w programie. Na potrzeby projektu utworzono zbiory zapisanych konfiguracji żądań HTTP testujących aplikację. Zbiory zapisane w formacie JSON zostały załączone do projektu.

2.5.3 GUI

Część systemu poświęcona interfejsowi graficznemu przetestowano za pomocą popularnych przeglądarek internetowych takich jak Microsoft Edge czy Google Chrome. Zweryfikowano tak jak w przypadku API czasy odpowiedzi oraz poprawność zwracanych na żądanie danych.

Rozdział 3

Szczegóły implementacji bazy danych

3.1 Dane referencyjne

Na potrzeby zasilenia bazy danych inicjalnymi danymi wykorzystano podstawowe formuły dostępne w programie Microsoft Office Excel. Korzystając z formuły ZłączTeksty wygenerowano instrukcje SQL insert, które po ręcznych weryfikacjach i modyfikacjach pozwalały na zasilenie bazy danych zbiorem gmin, ograniczonym do terenów województwa bielskiego z lat 1975-1998.

3.2 Baza danych

3.2.1 Struktury

Proces implementacji modelu bazy danych opierał się w głównej mierze na narzędziu służącym do tworzenia diagramów ERD (z ang. *Entity Relationship Diagram*). Środowisko SSMS oferuje możliwość wygenerowania i wykonania kodu odpowiedzialnego za tworzenie obiektów zamodelowanych w procesie tworzenia diagramu ERD. Tym samym część implementacji związanej z tworzeniem tabel, kolumn czy tzw. constrainów (z ang. przymus) takich jak klucze obce oraz główne ograniczono do minimum.

Przykładowe instrukcje SQL insert wygenerowane przy pomocy formuł programu Microsoft Office Excel znajdują się w fragmencie kodu 3.1.

Fragment kodu 3.1 Instrukcje odpowiedzialne za inicjalne dane tabeli Gmina — *Gminy Miejskie 1975−1998*

```
insert into gmina(id, Name, Type, startDate, enddate,
StartDateVerified, EndDateVerified, IsCreationDate)
values (1, 'Bielsko−Biała', 'Gmina␣Miejska', '1900−01−01',
'1998−12−31', 0, 1, 0);
insert into gmina(id, Name, Type, startDate, enddate,
StartDateVerified, EndDateVerified, IsCreationDate)
values (2, 'Andrychów', 'Gmina␣Miejska', '1900−01−01',
'1991−12−31', 0, 1, 0);
```

Celem ułatwienia operacji wykonywanych z poziomu REST API, na bazie danych utworzono możliwie uniwersalne widoki realizujące logikę złączeń tabel. Aplikacja (oraz administrator danych z poziomu środowiska SSMS) mogą korzystać z widoków, przesyłając ewentualne warunki filtracji tabel.

Widok realizujący logikę relacji dla wybranego typu encji przedstawiono w fragmencie kodu 3.2, natomiast widok realizujący logikę połączeń pomiędzy trzema tabelami głównymi pokazano w fragmencie kodu 3.3. Drugi z wspomnianych widoków napisano w celu przedstawienia wszystkich istniejących encji oraz relacji je wiążących — z wyłączeniem relacji gmin z innymi gminami.

Fragment kodu 3.2 Instrukcja tworząca widok realizujący logikę połączeń tabel Powiat oraz Wojewodztwo z tabelą Gmina

```
create or alter view [dbo].[WojewodztwoRelacje] as
select w.ID as Wojewodztwo_ID, w.Name as Wojewodztwo_Name,
       w.StartDate as Wojewodztwo_StartDate, w.EndDate
       as Wojewodztwo_EndDate, g.Description as DescriptionUnit,
       g.ID as IDUnit, g.Name as NameUnit, g.StartDate
       as StartDateUnit, g.EndDate as EndDateUnit, gw.StartDate,
       gw.EndDate
 from wojewodztwo w
 join GminaWojewodztwo gw on w.ID = gw.IDWojewodztwo
  and w.StartDate = gw.StartDateWojewodztwo
 join gmina g on g.id = gw.IDGmina
  and g.StartDate = gw.StartDateGmina
union
select w.ID as Wojewodztwo_ID, w.Name as Wojewodztwo_Name,
       w.StartDate as Wojewodztwo_StartDate, w.EndDate
       as Wojewodztwo_EndDate, p.Description as DescriptionUnit,
       p.ID as IDUnit, p.Name as NameUnit, p.StartDate
       as StartDateUnit, p.EndDate as EndDateUnit, pw.StartDate,
       pw.EndDate
 from wojewodztwo w
 join PowiatWojewodztwo pw on w.ID = pw.IDWojewodztwo
  and w.StartDate = pw.StartDateWojewodztwo
 join powiat p on p.id = pw.IDPowiat
  and p.StartDate = pw.StartDatePowiat;
```

Fragment kodu 3.3 Instrukcja tworząca widok realizujący logikę złączeń głównych tabel przechowujących encje

```
select  /*(...)*/
from Wojewodztwo w
left join PowiatWojewodztwo pw on w.ID = pw.IDWojewodztwo
        and w.StartDate = pw.StartDateWojewodztwo
left join Powiat p on p.id = pw.IDPowiat
        and p.StartDate = pw.StartDatePowiat
```

```
left join GminaWojewodztwo gw on w.ID = gw.IDWojewodztwo
        and w.StartDate = gw.StartDateWojewodztwo
left join GminaPowiat gp on p.ID = gp.IDPowiat
        and p.StartDate = gp.StartDatePowiat
join gmina g on g.ID = gw.IDGmina and g.StartDate =
        gw.StartDateGmina or g.ID = gp.IDGmina and
        g.StartDate = gp.StartDateGmina
```

Na potrzeby ułatwienia komunikacji aplikacji z bazą zaimplementowano również widok realizujący logikę relacji gmin z innymi gminami. Implementację widoku przedstawiono w fragmencie kodu 3.4.

Fragment kodu 3.4 Instrukcja tworząca widok realizujący logikę połączeń dla relacji gmin z innymi gminami

```
create or alter view [dbo].[GminaGmina] as
select g.id as IDGmina1, g.Name as Name1, g.Description
        as Description1, g.StartDate as StartDate1,
        g.Description as Opis1,
        g2.id as IDGmina2, g2.Name as Name2, g2.Description
        as Description2, g2.StartDate as StartDate2,
        g2.Description as Opis2,
        gr.StartDate, gr.EndDate, gr.Description
from GminaRelacja gr
join gmina g on g.ID = gr.IDGmina_1
 and g.StartDate = gr.Start_Date_1
join gmina g2 on g2.ID = gr.IDGmina_2
 and g2.StartDate = gr.Start_Date_2;
```

3.2.2 Dane

Instrukcje insert odpowiedzialne za wstawianie danych do tabeli Wojewodztwo oraz Powiat napisano ręcznie z racji na mały wolumen danych (jeden wpis do tabeli Wojewodztwo dla lat 1975-1998, dla lat 1999-2020 dwa wpisy do tabeli Wojewodztwo i sześć wpisów dla tabeli Powiat).

Z powodu specyfiki zagadnienia, instrukcje insert odpowiedzialne za inicjalne dane w tabeli przechowującej relacje pomiędzy gminami również napisano ręcznie, tj. przy pomocy najprostszej instrukcji SQL insert.

Następnie do zasilenia danymi tabeli relacyjnej GminaWojewodztwo, przechowującej dane o relacjach pomiędzy gminami a województwami użyto instrukcji SQL merge, która pozwala m.in. na dodanie danych do tabeli w sytuacji, gdy spełniony zostanie określony warunek. Dzięki jej wykorzystaniu, możliwym było operowanie na wielu jednostkach administracyjnych jednocześnie.

Przykładowa instrukcja merge wykorzystana przy dodaniu inicjalnych danych bazy przedstawiono w fragmencie kodu 3.5.

Fragment kodu 3.5 Instrukcja merge wykorzystana do dodania inicjalnych danych do tabeli relacyjnej dla gmin i województw

```
merge into gminawojewodztwo gw
using (Select g.enddate, g.ID as IDGminy,
              g.startDate as StartDateGminy,
              g.startDateVerified, g.endDateVerified,
              w.ID as IDwojewodztwa,
              w.startDate as startDateWojewodztwa
        from gmina g, wojewodztwo w) g
          on g.idGminy=gw.idGmina
when not matched then
insert (IDGmina, StartDateGmina, IDWojewodztwo,
        StartDateWojewodztwo, StartDate, EndDate,
        StartDateVerified, EndDateVerified, Description)
values (g.IDGminy, g.StartDateGminy, g.IDWojewodztwa,
        g.StartDateWojewodztwa,
        case
            when g.StartDateGminy = '1900-01-01'
            then '1975-06-01'
            else g.StartDateGminy
        end, g.endDate, g.StartDateVerified,
        g.endDateVerified, null);
```

W populacji bazy danymi z przedziału czasowego 1999–2020 ponownie wykorzystano instrukcję SQL merge, operując wieloma jednostkami w tym samym czasie, co można zobaczyć w fragmencie kodu 3.6.

Fragment kodu 3.6 Dodanie danych do tabeli relacyjnej gmin i powiatów

```
merge into GminaPowiat gp
using ( Select g.ID as IDGminy, g.StartDate, p.ID
                from gmina g
                join powiat p on p.Name = 'Bielski'
                where g.Name in ('Szczyrk', 'Wilamowice',
                      'Buczkowice', 'Jasienica', 'Jaworze',
                       'Kozy', 'Porąbka', 'Wilkowice')
                and g.endDate = '9999-12-31'
                ) p on p.ID = gp.IDPowiat
when not matched then
insert (IDGmina, StartDateGmina, IDPowiat,
        StartDatePowiat, StartDate, EndDate,
        StartDateVerified, EndDateVerified,
        Description)
values (p.IDGminy, p.StartDate, p.ID, '1999-01-01',
        '1999-01-01', '9999-12-31', 1, 0, null);
```

3.2.3 Bezpieczeństwo bazy danych

Na potrzeby zaprezentowanych w poprzednim rozdziale przypadków użycia, pomijając użytkownika z uprawnieniami systemowymi, powołano dwóch użytkowników bazy danych z różnymi prawami dostępu. Pierwszy z nich, nazwany DBREADER posiada prawa odczytu danych znajdujących się we wszystkich powstałych strukturach. Użytkownika posiadającego prawa do odczytu oraz modyfikacji danych nazwano APIJSON. Na potrzeby prezentacji, hasła ustanowiono na dokładnie takie same, jak loginy użytkowników, a więc odpowiednio DBREADER oraz APIJSON.

Rozdział 4

Dokumentacja REST API

Niezależnie od poziomu, z którego użytkownik chce uzyskać dostęp do bazy danych, kwestię bezpieczeństwa i praw dostępu zrealizowano za pomocą utworzonych użytkowników bazy danych. Użytkownik chcąc odczytać bądź zmodyfikować dane w bazie, musi znać hasło oraz login do wyżej wymienionych użytkowników. W przypadku aplikacji REST API, użytkownik musi podać login oraz hasło użytkownika w nagłówkach żądania HTTP przy pomocy pól: login oraz password. W przypadku interfejsu graficznego, dostęp do aplikacji jest możliwy dopiero gdy użytkownik poda prawidłowe poświadczenia dla użytkownika w formularzu HTML.

4.1 API

Aplikacja REST API napisano w microframeworku'u języka Python, flask'u. Aplikacja jest w stanie obsłużyć wysłane do niej żądania HTTP realizujące zamierzone funkcjonalności, czyli CRUD na zasobach bazy danych za pomocą funkcji odpowiednio: POST, GET, UPDATE, DELETE. Po wysłaniu żądania HTTP o odpowiedniej metodzie z ewentualnymi parametrami w ciele pod odpowiedni URL, aplikacja zwraca status wykonanej operacji oraz pole 'Record count' w nagłówku odpowiedzi, które należy interpretować następująco:

- **GET** – służące do odczytywania zasobów na bazie, zwraca status 200 w przypadku poprawnego wykonania zapytania na bazie, a pole 'Record count' mówi o liczbie zwróconych przez zapytanie rekordów. Do obsługi tej metody wystarcza adres URL zasobu.
- **POST** – służące tworzeniu zasobów, zwraca status 201 w przypadku poprawnego wykonania instrukcji insert SQL na bazie danych, a pole 'record count' wartość 1. W tym konkretnym przypadku uzyskanie statusu 201 jest równoznaczne z uzyskaniem wartości 1 w nagłówku odpowiedzi HTTP, oraz z rzeczywistym wykonaniem instrukcji SQL insert, ponieważ ta, w przeciwieństwie do pozostałych instrukcji SQL, nie posiada możliwości wykonania się, jednocześnie nie modyfikując stanu danych w tabeli. Do obsługi tej metody konieczne jest podanie kolumn tabeli i wartości tworzonego zasobu w ciele żądania HTTP, zapisanych przy pomocy formatu JSON.
- **PUT** – służące aktualizacji danych, zwraca status 200 w przypadku poprawnego wykonania instrukcji SQL update na bazie, a pole 'Record count' mówi o liczbie

zmodyfikowanych przez instrukcję rekordów. W tym przypadku również poza adresem zasobu, w ciele żądania HTTP należy podać kolumny oraz ich nowe wartości dla aktualizowanego zasobu w formacie JSON.

- **DELETE** – służące usuwaniu zasobów, zwraca status 200 w przypadku poprawnego wykonania instrukcji SQL delete na bazie, a pole 'Record count' mówi o liczbie usuniętych przez instrukcję rekordów. Metoda ta jest obsługiwana dzięki adresowi zasobu i nie jest wymaganym podawanie ciała żądania HTTP.

4.1.1 Metoda GET

Dostęp do zasobów jest możliwy dla użytkowników znających poświadczenia dla użytkownika bazodanowego. Ponadto użytkownik musi znać adres, metodę oraz ewentualną treść ciała żądania HTTP, aby z powodzeniem korzystać z API. Wszystkie z tych składowych zostały opisane w poniższym punkcie.

W celu jednoznacznego wydzielenia API od pozostałej części aplikacji, adresy URL wszystkich zasobów zostały poprzedzone łańcuchem znaków '/api'. W środowisku deweloperskim, czyli aplikacją działającą lokalnie, niezmienna część URL dla API prezentuje się następująco:

http://127.0.0.1:5000/api

W przypadku chęci odczytu zasobów, a więc użycia metody HTTP GET, adres pod który użytkownik powinien się udać, może zostać odczytany z tabeli 4.1 opisującej schemat budowania adresów poszczególnych zasobów. Znak ✔oznacza obsługę przypadków użycia zapisanych w wierszach dla tabel bądź widoków zapisanych w kolumnach. Kolorem żółtym zaznaczono wybrane zbiory endpoint'ów, dla których aplikacja jest wrażliwa na wielkość liter w adresie URL.

Tablica 4.1 Możliwe operacje za pomocą aplikacji REST API

EndPoint	Nazwa tabeli/widoku					
	Gmina & Powiat & Wojewodztwo	GminaPowiat & GminaWojewodztwo & PowiatWojewodztwo	GminaRelacja	GminaGmina	GminaRelacje & PowiatRelacje & WojewodztwoRelacje	GminaPowiatWojewodztwo
/tabela lub widok	✔	✔	✔	✔	✔	✔
/ID	✔		✔	✔	✔	
/ArgDate	✔	✔	✔	✔	✔	✔
/ID/ArgDate	✔		✔	✔	✔	
/Typ/ID		✔				✔
/Typ/ID/ArgDate		✔				✔
/ID/DT/ID/DT/DT		✔	✔			

Nazwa tabeli lub widoku

Pierwszy zbiór endpoint'ów budowany jest poprzez dodanie do adresu aplikacji nazwy tabeli lub widoku. Każdy z niżej omówionych przypadków jest rozszerzeniem tego zbioru endpoint'ów, poprzez dodawanie do adresu URL opcjonalnych argumentów. Użytkownik przykładowo chcąc poznać całą zawartość tabeli Gmina, wysyła żądanie HTTP z metodą GET pod adres URL (na środowisku deweloperskim, serwerze lokalnym):

http://127.0.0.1:5000/api/gmina

Identyfikator encji

Drugi zbiór endpoint'ów ma zastosowanie gdy użytkownik chce zapoznać się z informacjami o konkretnej jednostce. W zależności od zakresu informacji interesujących użytkownika, na przykładzie gminy Bielsko-Biała, użytkownik znając jej identyfikator (przykładowo zwrócony w odpowiedzi na poprzednie żądanie), może wysłać żądanie z metodą GET pod jeden z poniższych adresów URL:

. . . /api/gmina/1, **. . . /api/GminaRelacja/1**, **. . . /api/GminaGmina/1**
lub **. . . /api/GminaRelacje/1**

Data jako argument

Użytkownik korzysta z trzeciego zbioru endpoint'ów, jeżeli chce się zapoznać z danymi obowiązującymi w tabeli bądź widoku na dany dzień. Użytkownik powinien dodać datę w formacie RRRR-MM-DD do adresu URL żądania, poprzedzoną nazwą obiektu w bazie, czyli na przykład:
. . . /api/gmina/2005-06-01,
. . . /api/GminaPowiatWojewodztwo/2005-06-01 itd.
Do obsługi tego zbioru adresów URL wykorzystano kolumny StartDate i EndDate zawarte w każdej ze struktur w bazie danych.

Identyfikator oraz data

Czwarty z pokazanych w tabeli 4.1 zbiorów adresów URL jest kombinacją dwóch poprzednich zbiorów, obsługując żądanie HTTP wysłane do aplikacji w celu poznania stanu danych dla konkretnej jednostki administracyjnej na podany punkt w czasie. Z uwagi na specyfikę danych, dla tabel przechowujących encje, zagwarantowane jest zwrócenie co najwyżej jednego zasobu. Przykładowe adresy:

. . . /api/gmina/1/2005-06-01, **. . . /api/GminaRelacje/1/2005-06-01**

Typ, identyfikator i opcjonalnie data

Piąty i szósty zbiór endpoint'ów ma zastosowanie w tabelach relacyjnych II rodzaju, a więc GminaPowiat, GminaWojewodztwo, PowiatWojewodztwo oraz widoku GminaPowiatWojewodztwo. Zbiory rozszerzają odpowiednio drugi i czwarty zbiór endpointów, poprzez określenie rodzaju jednostki interesującej użytkownika. Wysłanie żądania HTTP z metodą GET pod przykładowe adresy:

. . . /api/GminaWojewodztwo/Gmina/1,
. . . /api/GminaPowiatWojewodztwo/Gmina/1/2005-06-01
spowoduje odpowiedź serwera, odpowiednio zwracając wszystkie rekordy z informacjami o relacji gminy z identyfikatorem równym 1, oraz zwracając rekordy dla widoku ograniczonego do gminy z polem ID = 1, polem StartDate mniejszym lub równym od daty 1 czerwca 2005 roku i polem EndDate większym bądź równym od daty 1 czerwca 2005 roku.

Klucz główny tabeli relacyjnej

Ostatni z zaimplementowanych zbiorów endpoint'ów pozwala na wysłanie żądania do aplikacji celem uzyskania co najwyżej jednego rekordu z dowolnej tabeli relacyjnej, przykładowo:

.../api/GminaPowiat/1/2001-01-01/1/1999-01-01/2020-01-01
Aplikacja odeśle zasób jeżeli w dniu 1 stycznia 2020 roku zachodziła relacja pomiędzy gminą o identyfikatorze 1, istniejącą w dniu 1 stycznia 2001 roku a powiatem o identyfikatorze 1, istniejącym w dniu 1 stycznia 1999 roku.

Implementacja

W przypadku użycia metody HTTP typu GET, formatem zapisu zasobów zwracanych przez REST API jest JSON.

W fragmencie kodu 4.1 umieszczono kod przykładowej funkcji pozwalającej na obsługę żądania HTTP z metodą GET i nagłówkiem zawierającym dane do logowania wysłanym pod odpowiedni adres URL dla widoku GminaPowiatWojewodztwo.

Fragment kodu 4.1 Przykładowy kod funkcji odpowiedzialnej za obsługę metody GET dla widoku GminaPowiatWojewodztwo

```
@app.route('/api/widoki/GminaPowiatWojewodztwo/<table_name>/'
           '<arg>', methods=['GET'])
def json_widoki_GminaPowiatWojewodztwo_tablename_arg(
        table_name, arg):
    if str(arg).isnumeric():
        query = "SELECT * FROM GminaPowiatWojewodztwo where "
                + table_name + "_ID = " + arg + " order by "
                + table_name + "_nazwa"
    else:
        query = "SELECT * FROM GminaPowiatWojewodztwo where "
                + arg + " betweeen " + table_name +
                "_StartDate and " +
                table_name + "_EndDate order by " +
                table_name + "_nazwa"
    return api_respond_handler(query)
```

Kod funkcji odpowiedzialnej za wykonanie wyżej wygenerowanego zapytania SQL oraz sformatowanie zwróconych danych do formatu JSON znajduje się poniżej w fragmencie kodu 4.2.

Fragment kodu 4.2 Przykładowy kod funkcji odpowiedzialnej za wykonanie zapytania SQL na bazie danych i zwrócenie otrzymanej informacji w formacie JSON

```
def api_respond_handler(query):
    login = request.headers.get('login')
    password = request.headers.get('password')
    try:
        connection = pyodbc.connect(
            "Driver={SQL Server};Server=DESKTOP-BQPOPVS;"
            "PORT=1433;Database=Inzynier;UID=" + login
            + ";PWD=" + password
```

```
        )
        cursor = connection.cursor()
        cursor.execute(query)
        rows = cursor.fetchall()
        items = [dict(zip([key[0]
                           for key in cursor.description],
                          row)) for row in rows]
        resp_body = json.dumps(items, ensure_ascii=False)
        resp_head = dict()
        resp_head['Record_count'] = len(items)
        connection.close()
        response = make_response(resp_body, 200, resp_head)
        return response
    except:
        resp = Response(status=400)
        resp.headers['Record_count'] = 0
        return resp
```

Implementacja pozostałych metod jest zbliżona do przedstawionego przykładu dla metody GET.

4.1.2 Metoda POST

Chcąc utworzyć nowy zasób, użytkownik powinien wysłać żądanie HTTP z metodą POST pod adres URL odpowiadający nazwie tabeli, przykładowo
. . . /api/gmina, **. . . /api/GminaRelacja**
W ciele żądania musi jednak określić wartości, które ma przyjąć nowy zasób, zapisując to przy pomocy formatu JSON.

4.1.3 Metoda PUT

W przypadku potrzeby aktualizacji zasobu, użytkownik wysyła żądanie HTTP, używając metody PUT. Dla tabel przechowujących encje jest to możliwe za pomocą konstrukcji adresu URL, podając rodzaj jednostki jako nazwę tabeli, oraz klucz główny jednostki, czyli jej identyfikator oraz dotychczasową wartość pola StartDate, przykładowo:
. . . /api/gmina/1/1900-01-01.

Dla tabel relacyjnych użytkownik musi wprowadzić wszystkie pięć wartości klucza głównego (a więc kolumny stanowiące dwa klucze obce do encji i pole StartDate), przykładowo:
. . . /api/PowiatWojewodztwo/2/1999-01-01/3/1999-01-01/1999-01-01

Aktualizowane kolumny dla zasobów i ich nowe wartości, użytkownik podobnie jak w przypadku metody POST, podaje w ciele żądania HTTP.

4.1.4 Metoda DELETE

Celem usunięcia zasobów przy pomocy aplikacji, wysyłając żądanie HTTP należy użyć metody DELETE. Adres URL zasobu jest budowany analogicznie jak w przypadku użycia

wcześniejszych metod. Do obsługi tej funkcjonalności nie jest konieczne określanie ciała żądania HTTP.

4.2 GUI

Na potrzeby prezentacji przygotowano interfejs graficzny pozwalający na interakcje pomiędzy użytkownikiem a bazą danych. Zrealizowane to zostało przy pomocy języka HTML, możliwego do sformatowania dla przeglądarek internetowych. Interfejs graficzny pozwala na przeglądanie, dodawanie, modyfikację i usuwanie rekordów z bazy danych zalogowanym użytkownikom.

W fragmencie kodu 4.3 przedstawiono dwie przykładowe funkcje pozwalające na dodanie nowych zasobów w bazie danych.

Fragment kodu 4.3 Kod funkcji odpowiedzialnej za dodanie nowego rekordu do tabeli

```
@app.route('/insert<table_name>', methods=['GET', 'POST'])
def insert_table(table_name):
    if session.get("asd") is not None:
        login = session.get("login")
        password = session.get("password")
        connection = pyodbc.connect(
            "Driver={SQL Server};Server=DESKTOP-BQPOPVS;"
            "PORT=1433;Database=Inzynier;UID=" + login
            + ";PWD=" + password)
        cursor = connection.cursor()
        cursor.execute("SELECT column_name, data_type "
                       "FROM INFORMATION_SCHEMA.COLUMNS "
                       "where upper(table_name) = upper('"
                       + table_name + "') "
                       "order by ordinal_position asc")
        col_names = list()
        col_types = list()
        for row in cursor:
            col_names.append(row[0])
            col_types.append(row[1])
        col_vals = list()
        if request.method == 'POST':
            for x in range(len(col_names)):
                value = request.form[str(x + 1)]
                if value == '':
                    value = "null"
                elif col_types[x].lower() \
                        in ("date", "nvarchar"):
                    value = "'" + value + "'"
                col_vals.append(value)
            insert_all(table_name, col_names, col_vals,
                       GUIconnection)
        s, x = generate_table(table_name)
```

```
        nv = [str(x + 1) for x in range(len(col_names))]
        return render_template("insert.html",
                               table_name=table_name,
                               navigation=nv,
                               col_names=col_names,
                               table=s)
    return redirect(url_for('index'))


def insert_all(table_name, col_names, col_vals, connection):
    cursor = connection.cursor()
    s = "insert into " + table_name + "(" + str(col_names[0])
    for x in range(len(col_names) - 1):
        s = s + ", " + str(col_names[x + 1])
    s = s + ") values (" + str(col_vals[0])
    for x in range(len(col_vals) - 1):
        s = s + ", " + str(col_vals[x + 1])
    s = s + ")"
    cursor.execute(s)
    connection.commit()
    cnt = cursor.rowcount
    connection.close()
```

4.3 Testy i wyniki projektu

Aplikację można podzielić na trzy osobne, choć od siebie zależne komponenty. Komponenty aplikacji, czyli baza danych, API i GUI, zostały przetestowane różnymi narzędziami na wiele różnych sposobów.

4.3.1 Baza danych

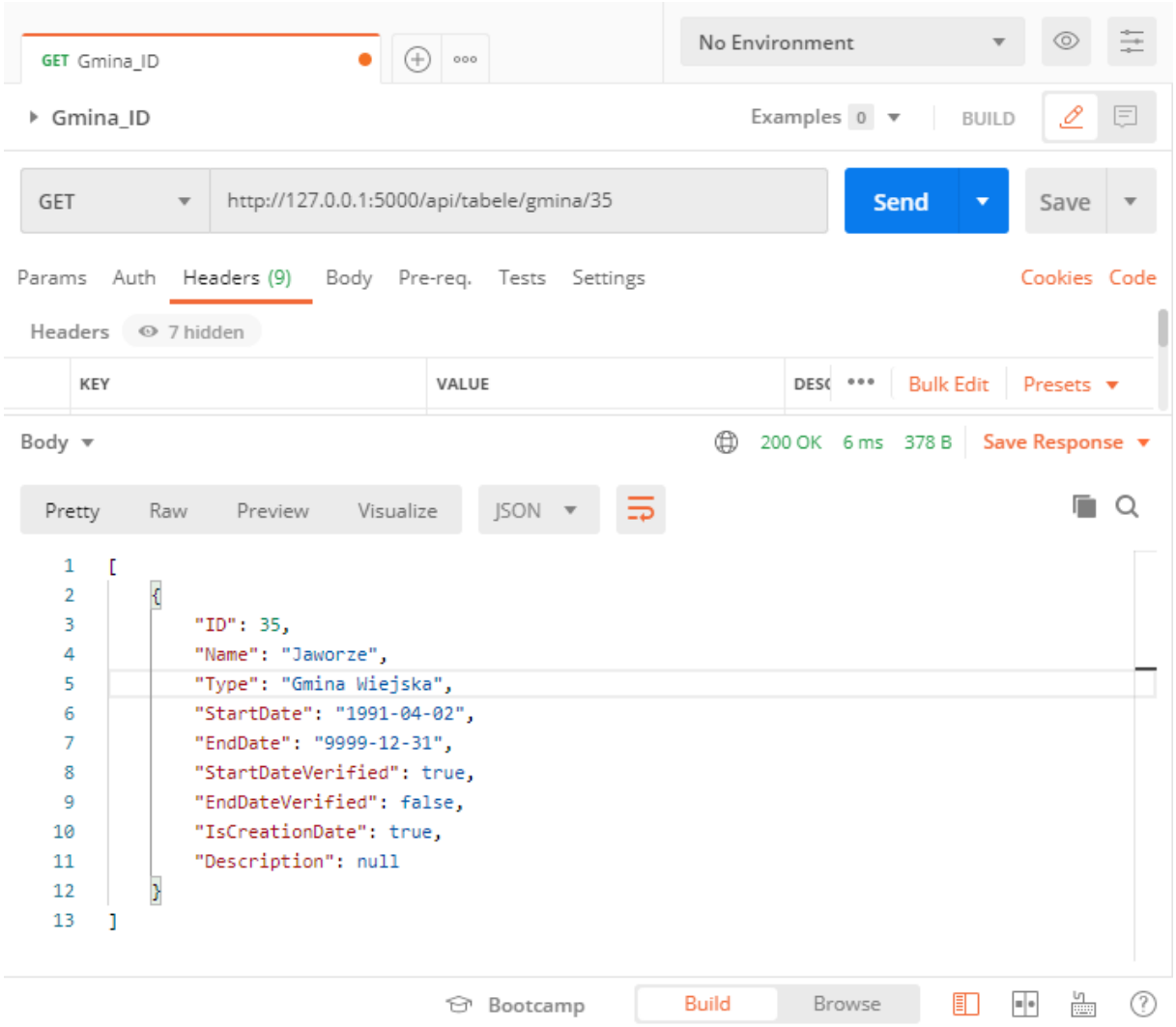
Testy bazy danych opierały się głównie na weryfikacji spójności danych, potraktowanej jako krytycznie ważny aspekt projektu. Testy odbywały się przy pomocy pisania i wykonywania zapytań w języku SQL na bazie danych. Przy pomocy instrukcji takich jak **join, left join, where, group by** sprawdzano poprawność zwracanych danych, porównując z oczekiwanymi wynikami. Duża część logiki wykorzystanej do pisania zapytań testujących bazę została wykorzystana podczas tworzenia widoków.

Jednocześnie weryfikowano czasy wykonania zapytań, jednak wobec potęgi automatycznych mechanizmów optymalizacji języka SQL, zbiór danych rzędu setek rekordów, nawet w środowisku deweloperskim, był za mały do wyciągnięcia z tych testów wartościowych wniosków. Analiza planów zapytań i ewentualne optymalizacje zapytań również okazały się bezcelowe. Nie miałoby to przełożenia dla większego zbioru przechowywanych danych, głównie z powodu stanu statystyk tabel, wykorzystywanych do wybierania optymalnego planu zapytania.

4.3.2 API

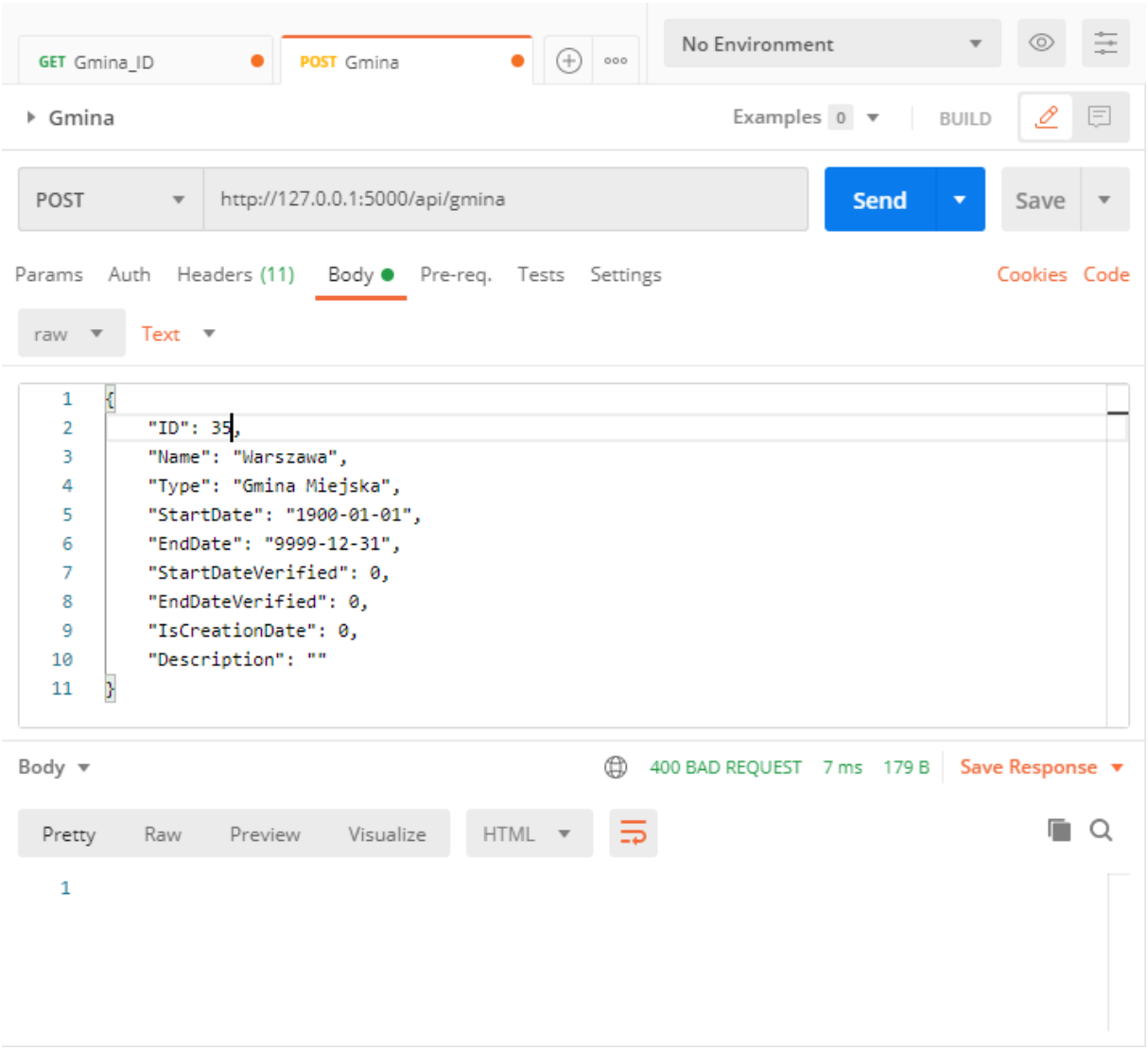
Zdecydowanie najwięcej czasu poświęcono testowaniu REST API. Do testów wykorzystano narzędzie PostMan, które pozwala na wysłanie żądania o określonej metodzie, nagłówku czy ciele pod wskazany adres URL. Narzędzie to pozwala również na odczytanie odpowiedzi odesłanej przez aplikację. Przeprowadzając testy jednostkowe, weryfikowano otrzymane wyniki, odsyłane przez aplikacje w odpowiedzi. Sprawdzano między innymi statusy, nagłówki oraz ciała odpowiedzi. Po wielokrotnych weryfikacjach czasu odpowiedzi serwera, zaobserwowano, że pierwsze po uruchomieniu aplikacji nawiązanie połączenia aplikacji z bazą danych zajmuje zdecydowanie najwięcej czasu. Obliczona średnia czasu oczekiwania na odpowiedź wynosi w przybliżeniu 4 sekundy. Biorąc pod uwagę fakt procesowania zarówno aplikacji jak i bazy danych, czasy odpowiedzi na kolejne wysyłane żądania były bardzo małe, liczone w tysięcznych częściach sekundy.

Na rysunku 4.1 pokazano jak wygląda przykładowa konfiguracja programu - zawierające się w nagłówku poświadczenia do użytkownika bazodanowego, metodę oraz adres URL w przypadku wysyłania żądania HTTP do aplikacji oraz przykładowy status i ciało odpowiedzi aplikacji.



Rysunek 4.1 Testowanie metody GET przy pomocy narzędzia PostMan

Dla kontrastu na rysunku 4.2 pokazano nieudaną — z powodu naruszenia klucza głównego — próbę dodania nowego zasobu do bazy danych. Kolumny i ich wartości dla nowo tworzonego zasobu zapisano w ciele żądania HTTP.



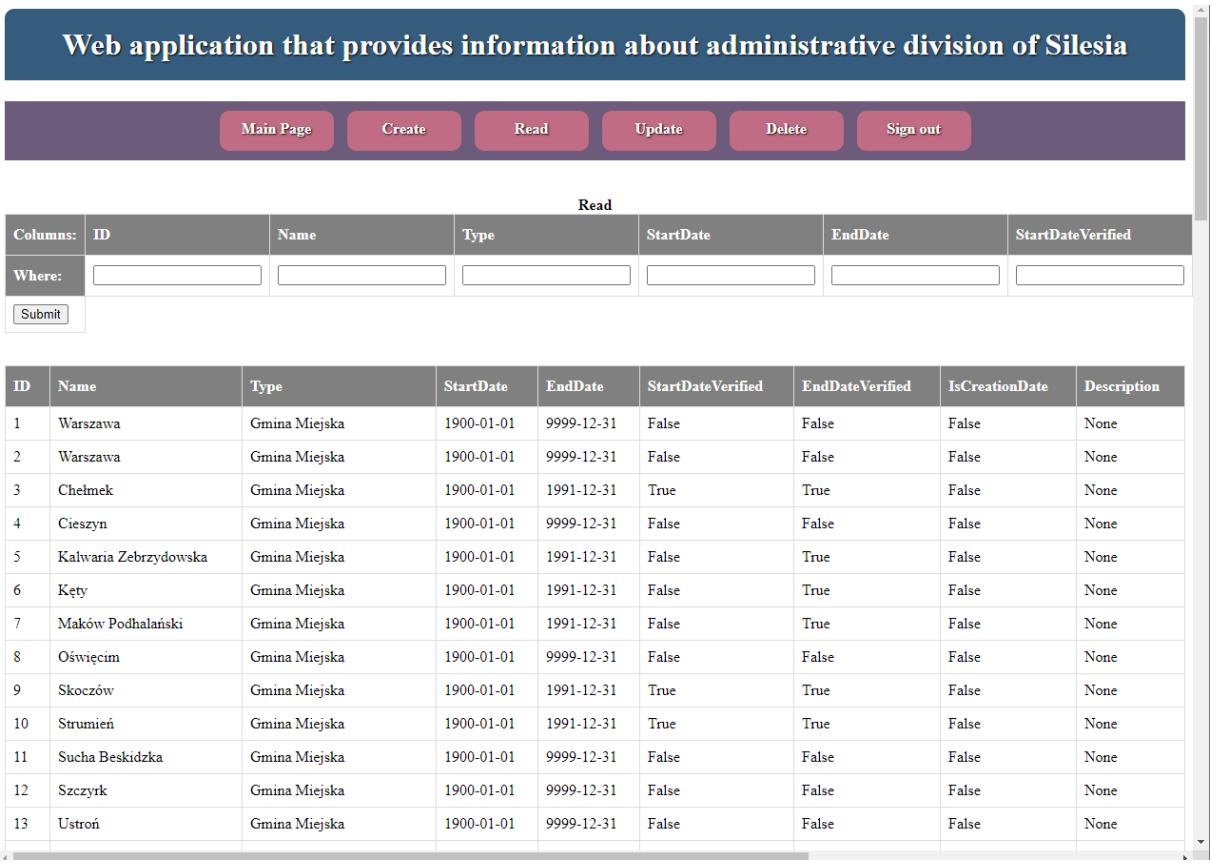
Rysunek 4.2 Testowanie metody POST przy pomocy narzędzia PostMan

Celem ułatwienia procesu testowania utworzono zbiory konfiguracji żądań, które zapisano w formacie JSON i dołączono do projektu. Zbiory obejmują po jednym przykładzie dla każdego możliwego przypadku użycia, czyli konfiguracje testów metod żądania HTTP: PUT, POST, DELETE dla 7 tabel oraz łącznie 52 konfiguracje testów dla metody GET.

4.3.3 GUI

Testy interfejsu graficznego aplikacji przeprowadzono za pomocą przeglądarki internetowej Google Chrome oraz Microsoft Edge. Przeprowadzono testy analogiczne do omówionych wyżej testów przy pomocy aplikacji PostMan. Zmierzono podobne czasy odpowiedzi aplikacji na wysyłane żądania, również w przypadku nawiązania pierwszego połączenia aplikacji z bazą danych. Poddano weryfikacji zachowanie się aplikacji przy nietypowych zachowaniach użytkownika, takich jak utworzenie wielu połączeń jednocześnie z poziomu

różnych przeglądarek. Pozwalało to również na przetestowanie sytuacji, w której z aplikacji korzysta wielu użytkowników jednocześnie. Stwierdzono poprawność działania całego systemu.



Rysunek 4.3 GUI aplikacji

Podsumowanie

Wnioski

Spełniono zdecydowaną większość założeń projektu, poprzez zamodelowanie oraz utworzenie bazy danych i zasilenie jej inicjalnymi danymi. Zaprojektowano i zamodelowano w aplikacji wszystkie oczekiwane, podstawowe funkcjonalności CRUD dla danych. Wszystkie części aplikacyjne projektu z osobna udokumentowano oraz przetestowano, co potwierdza poprawność ich działania.

W trakcie realizacji projektu zapoznałem się z nowymi technologiami oraz podejściami do różnych zagadnień, zarówno w odniesieniu do części związanej z projektowaniem aplikacji, jej implementacją jak i testowaniem.

Nie zaprojektowano procesu automatyzującego zasilanie bazy danymi z arkuszy kalkulacyjnych ze względu na poziom skomplikowania i czas potrzebny do jego implementacji. Poskutkowało to zmianą założeń i ograniczeniem wolumenu do danych dotyczących województwa bielskiego z okresu lat 1975—2020. Niemożliwym okazało się wykorzystanie w projekcie technologii Swagger, oferującej możliwość zautomatyzowanego generowania dokumentacji API. Zostało to spowodowane przyjętym modelem aplikacji REST API, co w żaden sposób nie wpływa na oferowane przez nią funkcjonalności.

Napotkane trudności

Pomijając początkowe trudności w zrozumieniu rzeczywistego celu projektu, faza transformacji danych referencyjnych okazała się najtrudniejsza, głównie z powodu wstępnych założeń o dodaniu do bazy wszystkich otrzymanych od promotora danych o jednostkach administracyjnych. Wolumen danych z początku był przytłaczający, fakt zróżnicowanych formatów zapisu danych spowodował poświęcenie zdecydowanie zbyt dużej ilości czasu na te zagadnienie.

Podczas procesu modelowania, implementacji oraz testowania części projektu związanej z bazą danych nie napotkałem poważniejszych trudności, co nie zmienia faktu, że nauczyłem się nowych technologii oraz zmierzyłem się z nowymi wyzwaniami.

Faza projektowania i implementowania aplikacji wymagała poświęcenia dużej ilości czasu celem poznania i zrozumienia wielu nieznanych dotychczas zagadnień informatyki, zarówno podczas modelowania, jak i w fazie implementacji różnych technologii.

Faza testów również stanowiła duże wyzwanie. Dzięki wytężonym wysiłkom i wsparciu merytorycznemu Promotora udało się ją ukończyć z powodzeniem, spełniając wymogi jakościowe projektu.

Jestem zdecydowanie zadowolony z finalnych efektów projektu, w szczególności z potraktowanego priorytetowo procesu modelowania bazy danych, oraz zrozumienia zagadnień związanymi z możliwościami dostępu do tej bazy danych poprzez aplikację typu REST API, gdyż planuję rozwój zawodowy właśnie w tym kierunku.

Możliwości Rozwoju

Główna możliwość rozwoju projektu zdecydowanie opiera się na zwiększeniu wolumenu danych, manualnym bądź automatycznym, pozwalając na zwiększenie użyteczności projektu. Rozważanym rozwiązaniem było utworzenie pakietu w bazie danych, który na podstawie tabeli konfiguracyjnej z identyfikatorami jednostek byłby w stanie automatycznie zasilać struktury bazodanowe. Zrezygnowano z tej funkcjonalności ze względu na dużą ilość scenariuszy, które należałoby uwzględnić.

Ponadto aplikacja REST API po odpowiednich restrukturyzacjach, może zostać rozszerzona o technologię Swagger, celem automatyzacji generowania dokumentacji REST API.

Pole do rozwoju jest również widoczne w części projektu odpowiedzialnej za graficzny interfejs użytkownika. Ta część projektu jest opcjonalnym dodatkiem, została potraktowana drugorzędnie i posiada jedynie podstawowe funkcjonalności.

Z uwagi na charakter udostępnianych danych i niskie zagrożenie ataku systemu, kwestie związane z bezpieczeństwem zostały również potraktowane drugorzędnie. Celem projektu było zapewnienie możliwości udostępniania danych. Możliwość modyfikacji zasobów poprzez aplikację wynika głównie z niepełnego stanu danych w bazie. Tym samym w przypadku ewentualnej potrzeby umożliwienia użytkownikom modyfikacji zasobów w środowisku produkcyjnym, koniecznym będzie zwiększenie poziomu bezpieczeństwa aplikacji. Można to zrealizować przykładowo poprzez wykorzystanie technologii HTTPS (z ang. *Hypertext Transfer Protocol Secure*). Wiązałoby się to z dodaniem funkcjonalności nadawania indywidualnych tokenów dostępowych dla użytkowników. Tokeny te stanowiłyby wówczas unikalne poświadczenia użytkownika do korzystania z wybranych funkcjonalności aplikacji. Nic nie stoi na przeszkodzie, aby podobne podejście zastosować w przypadku potrzeby ograniczenia odbiorców danych. Celem zabezpieczenia struktur oraz danych w bazie można również utworzyć tzw. backup'y (z ang. *kopia zapasowa*) bazy danych na osobnym serwerze.

Bibliografia

[1] T. Dziki. Podziały administracyjne Polski w latach 1944–1998. Z badań nad ustrojem ziem polskich w XIX i XX w. http://www.gwsh.gda.pl/uploads/oryginal/3/3/a3c67_433-450_Dziki.pdf [dostęp: 24 sierpnia 2020 roku].

[2] Flask. Dokumentacja Flask. https://flask.palletsprojects.com/en/1.1.x/ [dostęp: 24 sierpnia 2020 roku].

[3] Google. Słownik Google. https://support.google.com/google-ads/answer/14095?hl=pl [dostęp: 24 sierpnia 2020 roku].

[4] P. Głuchowski. Logika temporalna i automaty czasowe. http://staff.iiar.pwr.wroc.pl/pawel.gluchowski/wp-content/uploads/miasi/logika_w10.pdf Politechnika Wrocławska wersja 2.3 [dostęp: 24 sierpnia 2020 roku].

[5] junak.org. Podział administracyjny polski. https://junak.org/wp-content/uploads/Podzia%C5%82-adm.-1975.png [dostęp: 24 sierpnia 2020 roku].

[6] D. Knuth. Podział administracyjny polski. https://www.geografia24.eu/geo_prezentacje_rozsz_3/383_2_ludnosc_urbanizacja/r3_2_01a.pdf?fbclid=IwAR1dKScUVjHon_JbNLy4jQph6aboVD9aPv5rxhA_E3cKU3xR9RXRKGI-ckc [dostęp: 24 sierpnia 2020 roku].

[7] K. Lange. The little book on rest services. https://www.kennethlange.com/books/The-Little-Book-on-REST-Services.pdf [dostęp: 24 sierpnia 2020 roku].

[8] M. Meng, S. Steinhardt, A. Schubert. Application programming interface documentation: What do software developers want? *Journal of Technical Writing and Communication*, 48:295–330, 07 2018.

[9] Microsoft. Dokumentacja Sql Server Management Studio. https://docs.microsoft.com/en-us/sql/ssms/sql-server-management-studio-ssms?view=sql-server-ver15 [dostęp: 24 sierpnia 2020 roku].

[10] Mozilla. Dokumentacja Mozilla. HTTP. https://developer.mozilla.org/pl/docs/Web/HTTP [dostęp: 24 sierpnia 2020 roku].

[11] W. P. Nechay. Śląsk jako regjon geograficzny, 1935. http://rcin.org.pl/Content/13666/WA058_3447_K1004_Slask-rej-geo-Nechay.pdf [dostęp: 24 sierpnia 2020 roku].

[12] Oracle. Dokumentacja oracle. Co to jest baza danych? https://www.oracle.com/pl/database/what-is-database.html [dostęp: 24 sierpnia 2020 roku].

[13] Pandas. Python, dokumentacja biblioteki pandas. https://pandas.pydata.org/pandas-docs/version/0.25.3/ [dostęp: 24 sierpnia 2020 roku].

[14] PostMan. Dokumentacja PostMan. https://learning.postman.com/docs/getting-started/introduction/ [dostęp: 24 sierpnia 2020 roku].

[15] PyPI. Python, dokumentacja biblioteki pyodbc. https://pypi.org/project/pyodbc/ [dostęp: 24 sierpnia 2020 roku].

[16] PyPI. Python, dokumentacja biblioteki simplejson. https://pypi.org/project/simplejson/ [dostęp: 24 sierpnia 2020 roku].

[17] Python. Dokumentacja Python 3.7. https://docs.python.org/3.7/ [dostęp: 24 sierpnia 2020 roku].

[18] G. U. Statystyczny. eTERYT. http://eteryt.stat.gov.pl/eteryt/raporty/WebRaportZestawienie.aspx [dostęp: 24 sierpnia 2020 roku].

[19] Wikipedia. Podział administracyjny polski. https://pl.wikipedia.org/wiki/Podzia%C5%82_administracyjny_Polski [dostęp: 24 sierpnia 2020 roku].

Spis rysunków

Spis tablic

Spis fragmentów kodu

Dodatek A

Zawartość płyty CD/DVD

Zawartość płyty CD/DVD dołączonej do pracy przedstawia się następująco:

- **W04_225984_2020_praca_inżynierska.pdf** — dokumentacja projektu, czyli niniejszy dokument w wersji elektronicznej,
- **podzial_administracyjny_Slask.zip** — archiwum zawierające dane referencyjne otrzymane od Promotora,
- **Database.zip** — archiwum zawierające wszystkie napisane lub wygenerowane przy pomocy narzędzia SSMS kody odpowiadające za funkcjonowanie bazy danych — kolejność uruchamiania alfabetyczna po nazwie pliku,
- **requirements.txt** — plik służący konfiguracji środowiska wirtualnego dla języka Python, przechowujący wersje używanych bibliotek,
- **app.zip** — archiwum zawierające kod dla aplikacji REST API oraz niezbędne pliki pomocnicze,
- **Postman.zip** — archiwum zawierające zapisane i możliwe do importu konfiguracje testów aplikacji w programie PostMan.