GIS support

Materiały szkoleniowe

Bazy danych w QGIS
(poziom podstawowy)

Ministerstwo
Klimatu i Środowiska

Sfinansowano ze środków
Narodowego Funduszu
Ochrony Środowiska
i Gospodarki Wodnej

Spis treści:

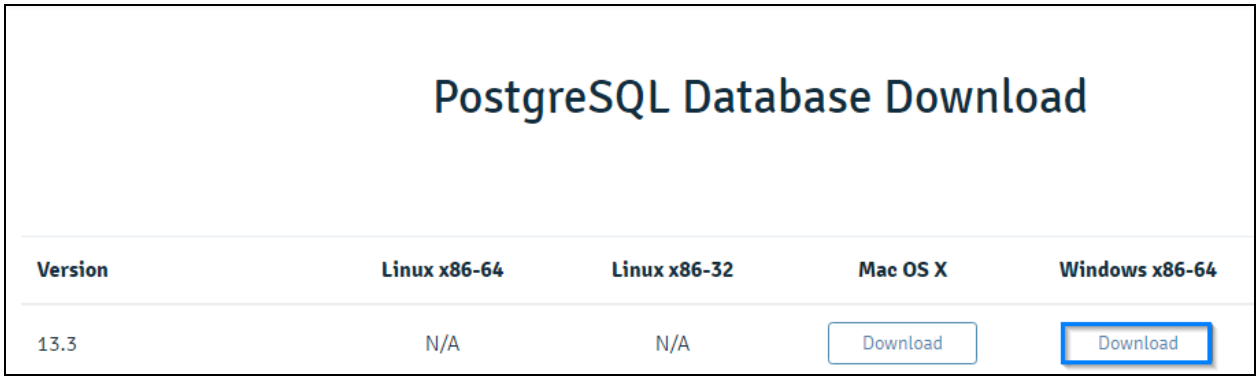
PostgreSQL - instalacja lokalnej bazy danych z rozszerzeniem PostGIS

Pobieramy plik instalacyjny ze strony
https://www.enterprisedb.com/downloads/postgres-postgresql-downloads
Wybieramy najnowszą wersję o numerze 13.4. Interesuje nas wersja instalatora Windows x86-64. W sytuacji, gdy korzystanie z najnowszej wersji nie jest możliwe, mogą Państwo wybrać wersję 12.8.

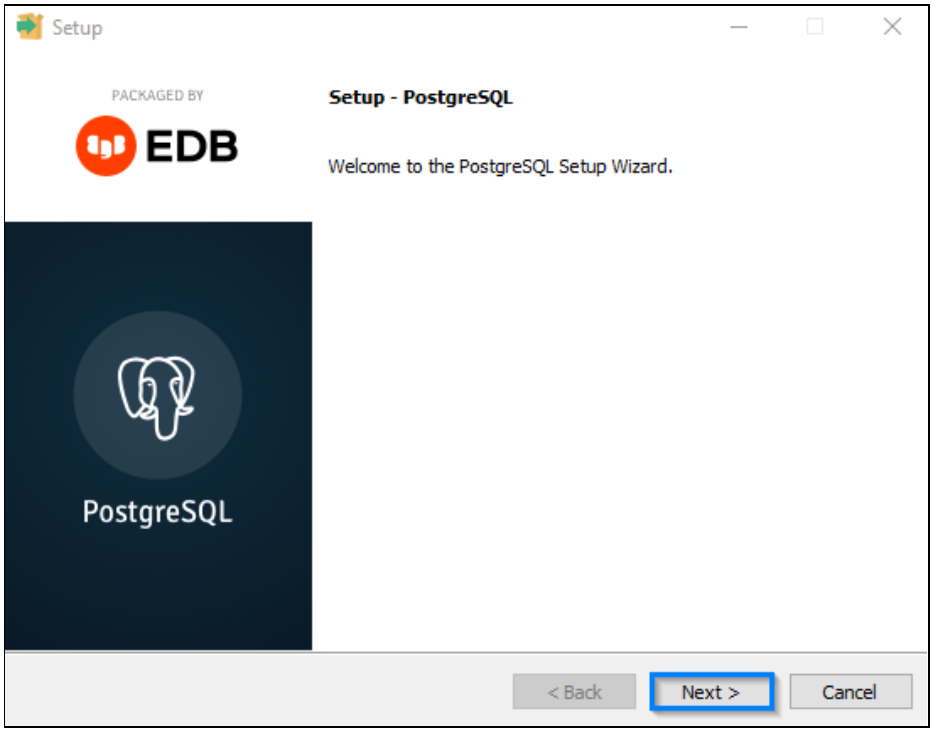
UWAGA! *Do poprawnego zainstalowania systemu PostgreSQL konieczne jest posiadania konta użytkownika z uprawnieniami administratora*.

Instalator systemu PostgreSQL:

- Serwer bazy danych
- Narzędzie pgAdmin do zarządzania oraz pracy na bazie danych
- StackBuilder - narzędzie do pobierania i instalowania nowych rozszerzeń

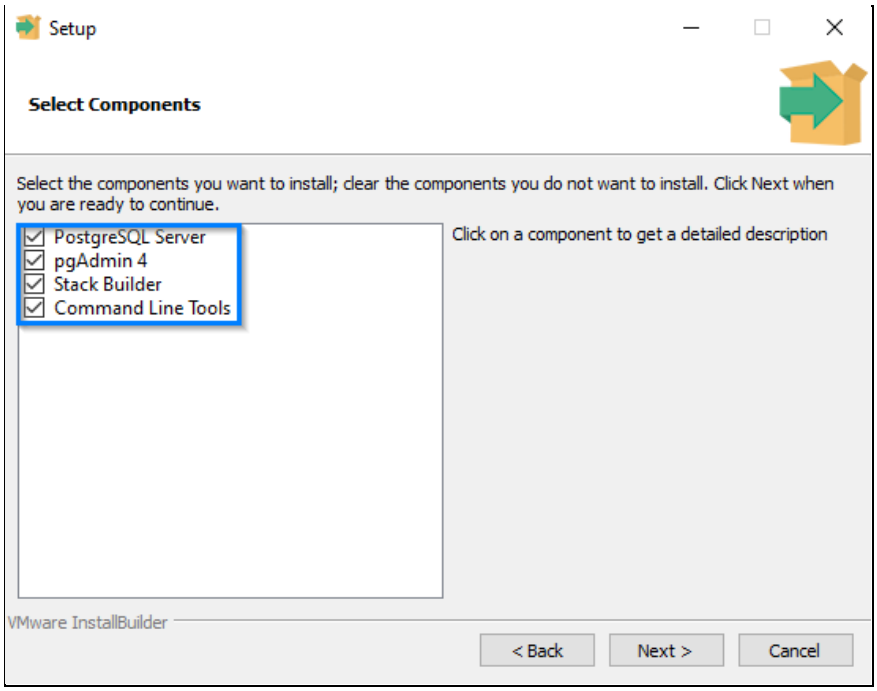


Po kliknięciu na przycisk *Download* pobieranie rozpocznie się automatycznie. Domyślnie plik instalacyjny zostanie zapisany w katalogu *Pobrane.* Po rozpakowaniu archiwum i uruchomieniu aplikacji instalacyjnej powita nas ekran:



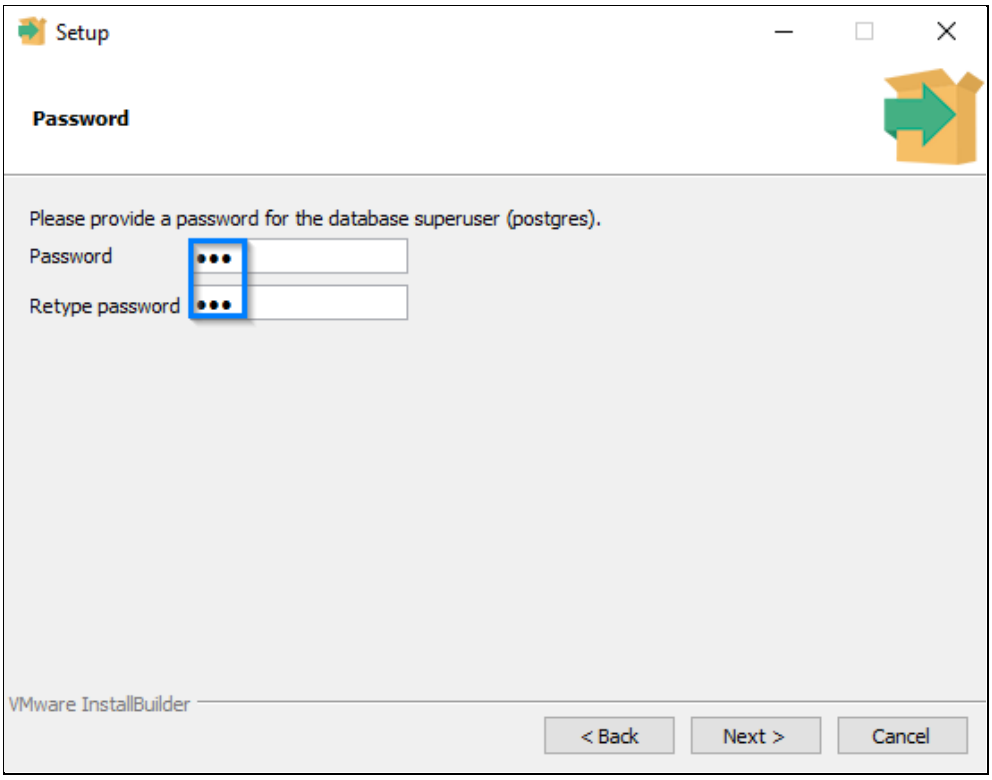
Aby przejść do kolejnego okna, klikamy na *Next.* Zostaniemy teraz poproszeni o wskazanie ścieżki do katalogu, w którym zostanie zainstalowana baza danych. Na potrzeby szkolenia pozostańmy przy ścieżce domyślnej, tj. *C:\Program Files\PostgreSQL\13.* Klikamy na *Next* i przechodzimy dalej.

W opcjach wyboru komponentów instalacji zaznaczamy wszystkie pozycje:

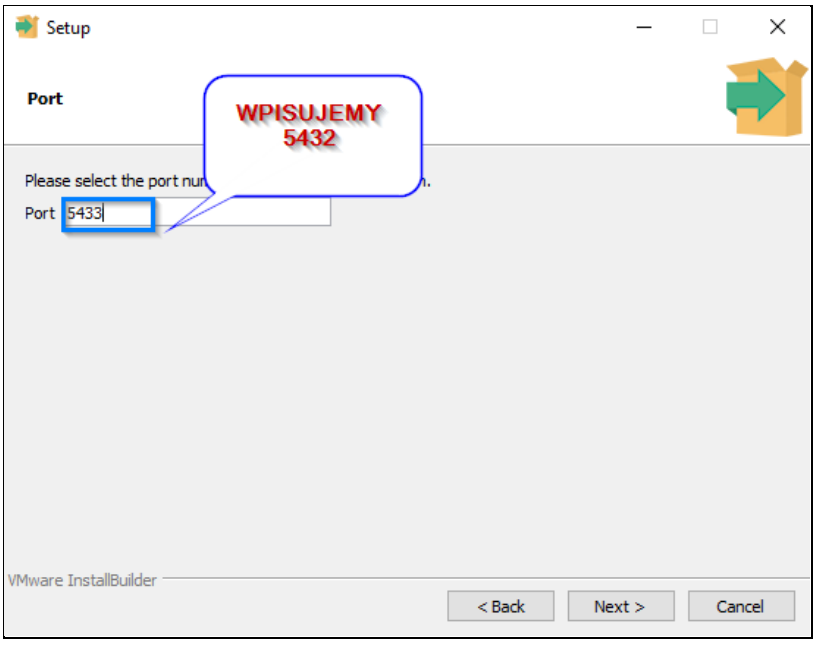


Następnie podajemy ścieżkę katalogu, w którym będą przechowywane dane. Również i w tym przypadku pozostajemy przy opcji domyślnej, tj. *C:\Program Files\PostgreSQL\13\data*.

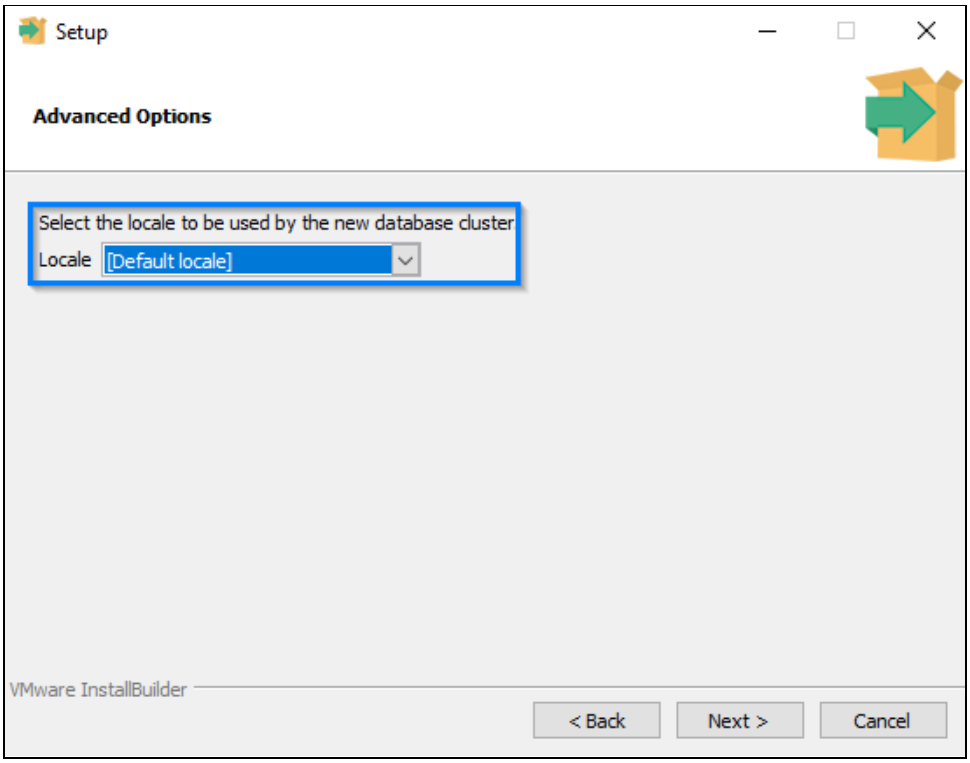
Przechodzimy do kolejnego okna. Tym razem musimy wprowadzić hasło do tworzonej bazy danych dla superusera o nazwie *postgres*. Zaleca się stosowanie konwencjonalnego hasła “gis”. Oczywiście mogą się Państwo zdecydować na inne, niemniej w sytuacji, gdy uleci ono z pamięci, konieczna będzie ponowna instalacja.



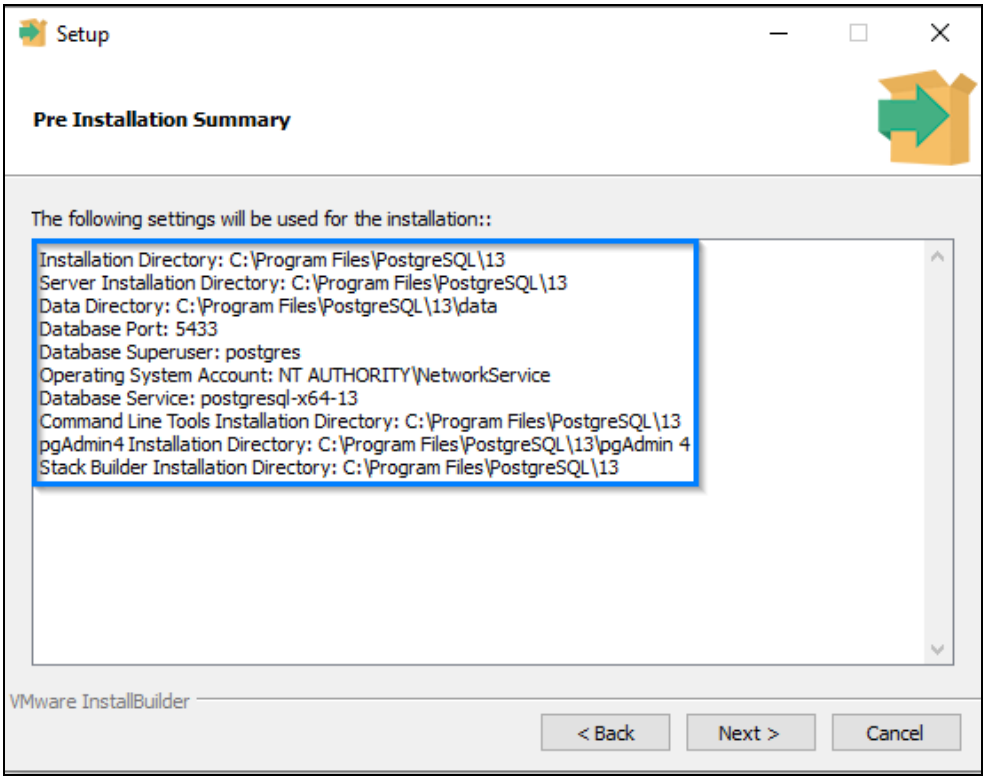
W następnym oknie należy wprowadzić numer portu bazy. Jeśli jest to pierwsza instalacja, program zasugeruje standardową wartość 5432. Jeśli natomiast port ten jest już zarezerwowany, wartość sugerowana zwiększy się o 1 (jak na ilustracji poniżej):



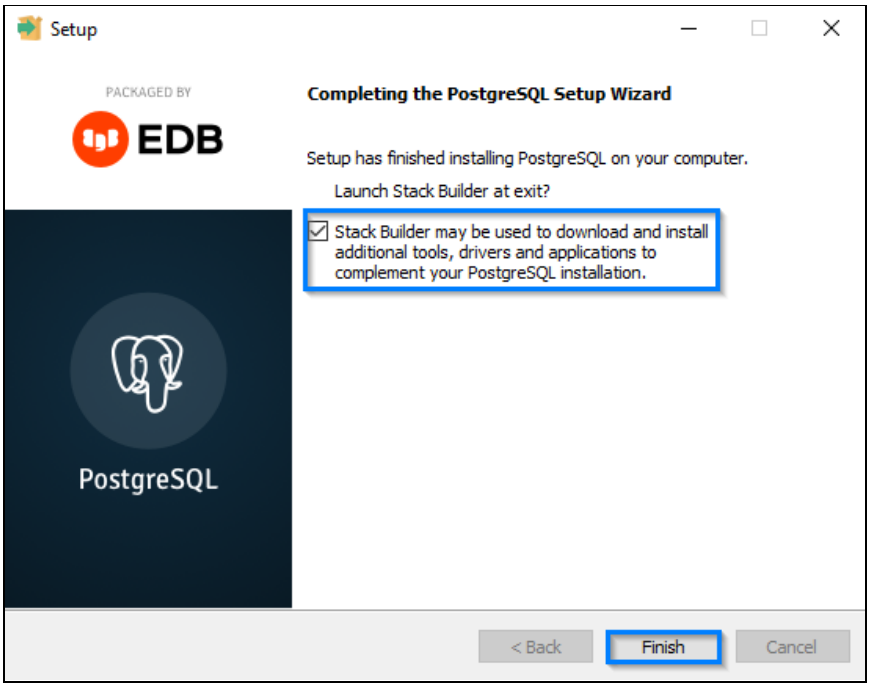
W kolejnym oknie definiujemy ustawienia lokalne systemu. Podobnie jak poprzednio, tak i tutaj pozostajemy przy wartości domyślnej:



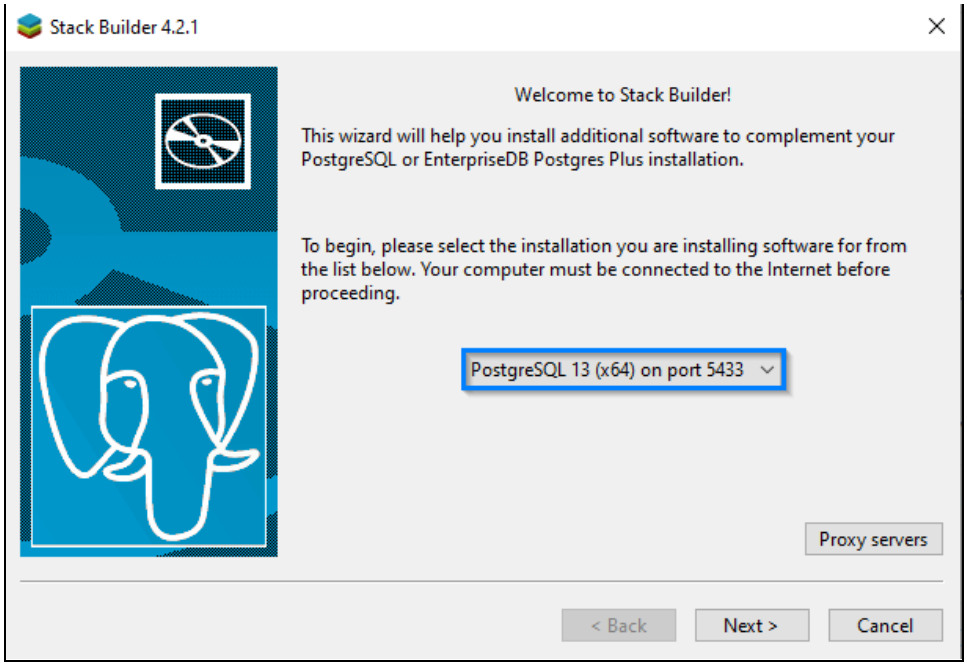
Na końcu procesu instalacji program instalator wyświetli ekran z informacjami podsumowującymi:



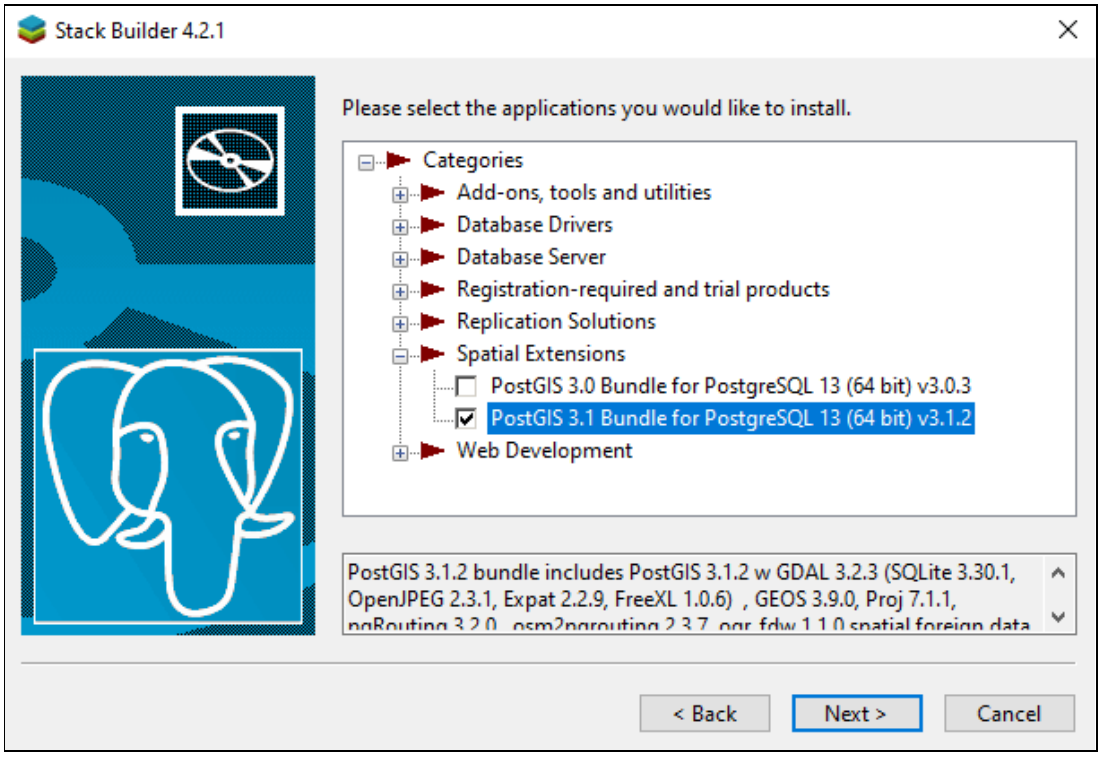
Po zakończeniu instalacji wyświetli się dodatkowe okno, z poziomu którego będziemy mogli uruchomić *Stack Buildera*, tj. narzędzie do instalacji dodatkowych rozszerzeń do bazy danych (jak chociażby POSTGIS).



Aby przejść dalej, musimy wskazać, dla której bazy instalowane będą dodatki. **W Państwa przypadku będzie to baza PostgreSQL w wersji 13 z portem 5432**.

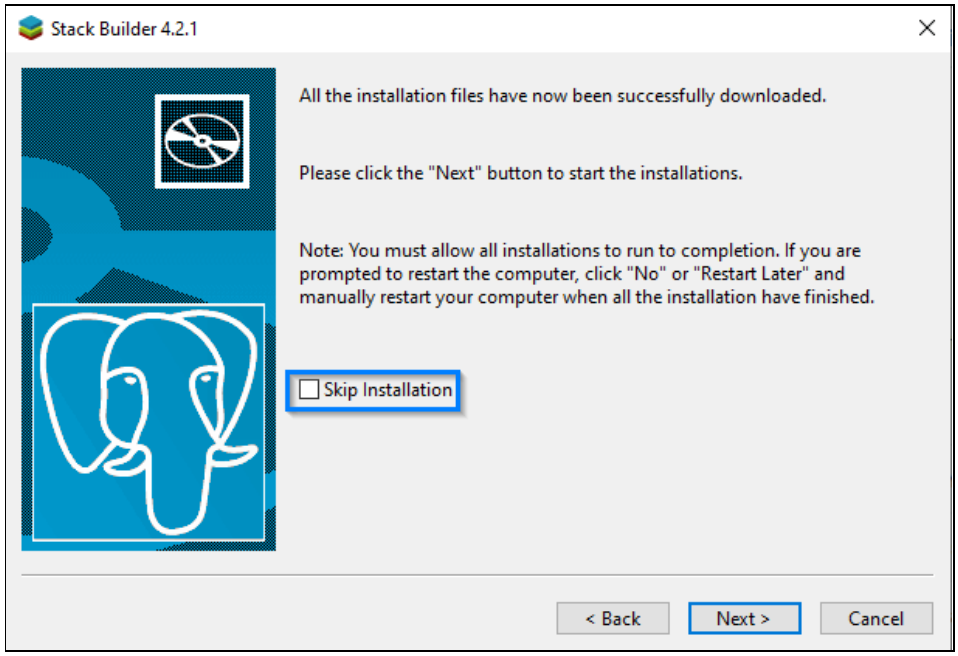


W kolejnym oknie interesuje nas przede wszystkim kategoria *Spatial Extensions*. Rozwijamy ją i wybieramy najświeższą wersję POSTGIS opatrzoną najwyższym numerem:

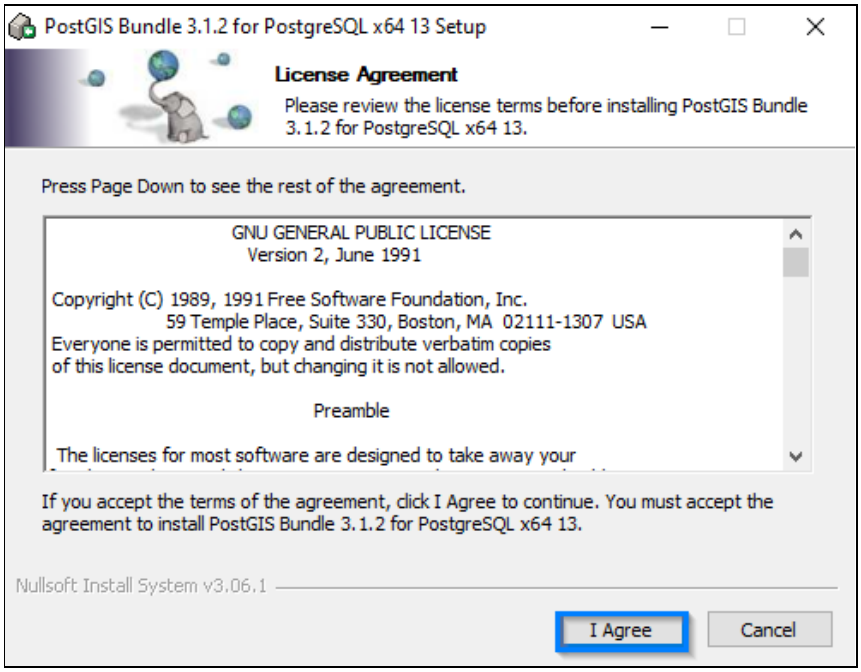


W ramach kolejnego ekranu zostanie wyświetlone podsumowanie. Ponadto mamy możliwość wskazania ścieżki katalogu, w którym zostaną zainstalowane rozszerzenia. Domyślnie jest to *C:\Users\nazwa użytkownika*.

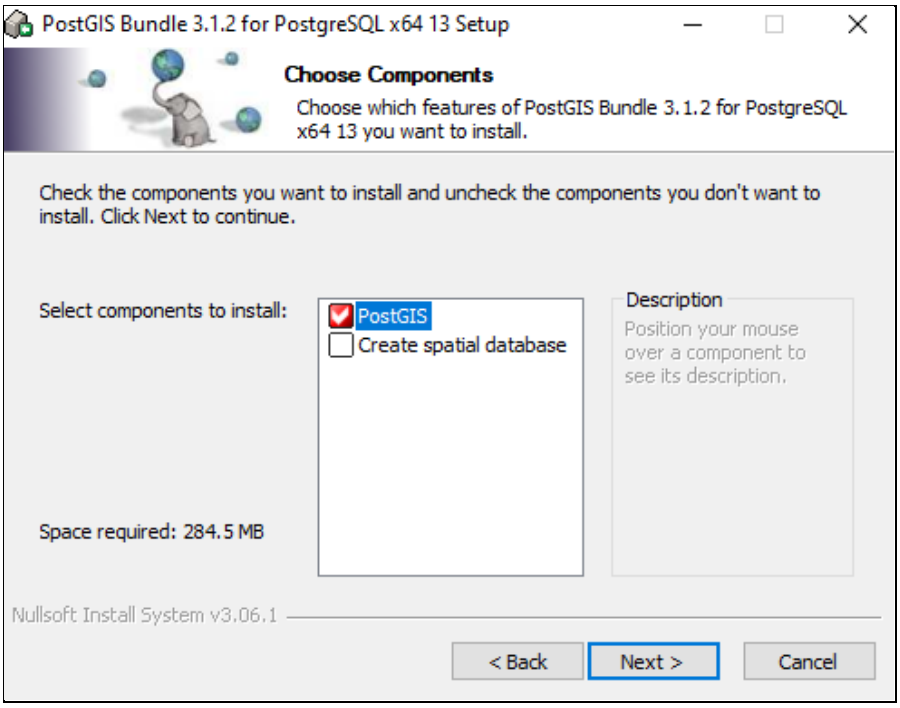
Proces konfiguracji instalacji zbliża się do końca. Musimy jeszcze tylko przebrnąć przez ostatni ekran i pamiętać o pozostawieniu pustego checkboxa przy opcji *Skip Installation*:



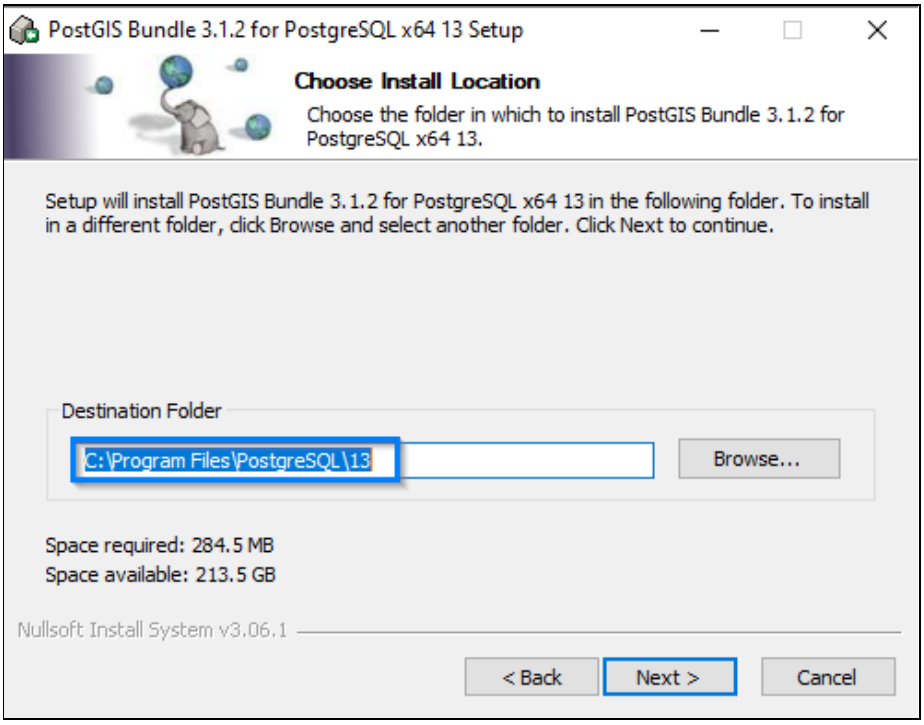
Przechodzimy do akceptacji warunków licencyjnych:



Dalej upewniamy się, że opcja instalacji POSTGIS jest zaznaczona i przechodzimy do następnego ekranu:

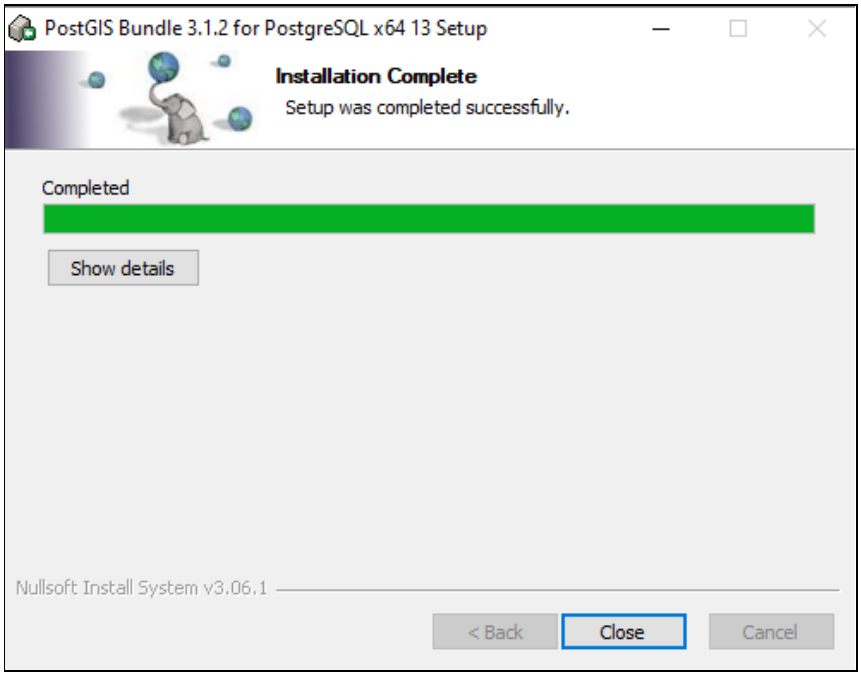


Wskazujemy ścieżkę instalacji komponentu (musi być zgodna ze ścieżką katalogu z bazą danych):



W trakcie instalacji będą pojawiać się dodatkowe ekrany z pytaniem o konfigurację zmiennych środowiskowych dla poszczególnych bibliotek. W każdym z przypadków klikamy na *OK*.

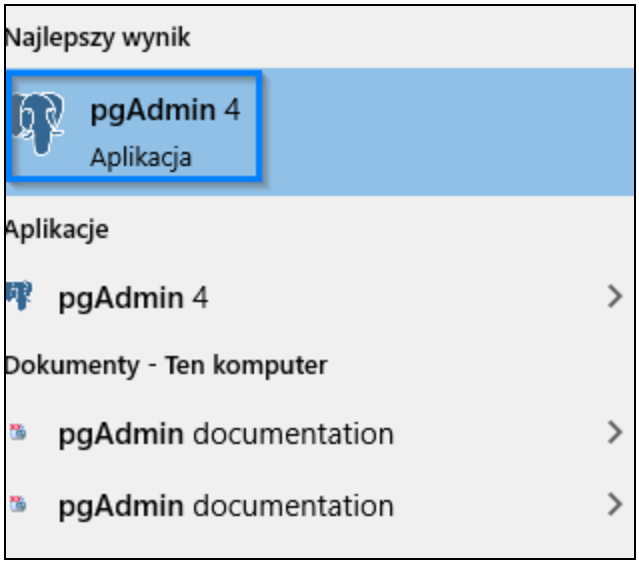
Po zakończeniu instalacji wyświetli się ekran końcowy z wypełnionym paskiem postępu i hasłem *Completed*. Aby ostatecznie zakończyć proces, klikamy na *Close*.



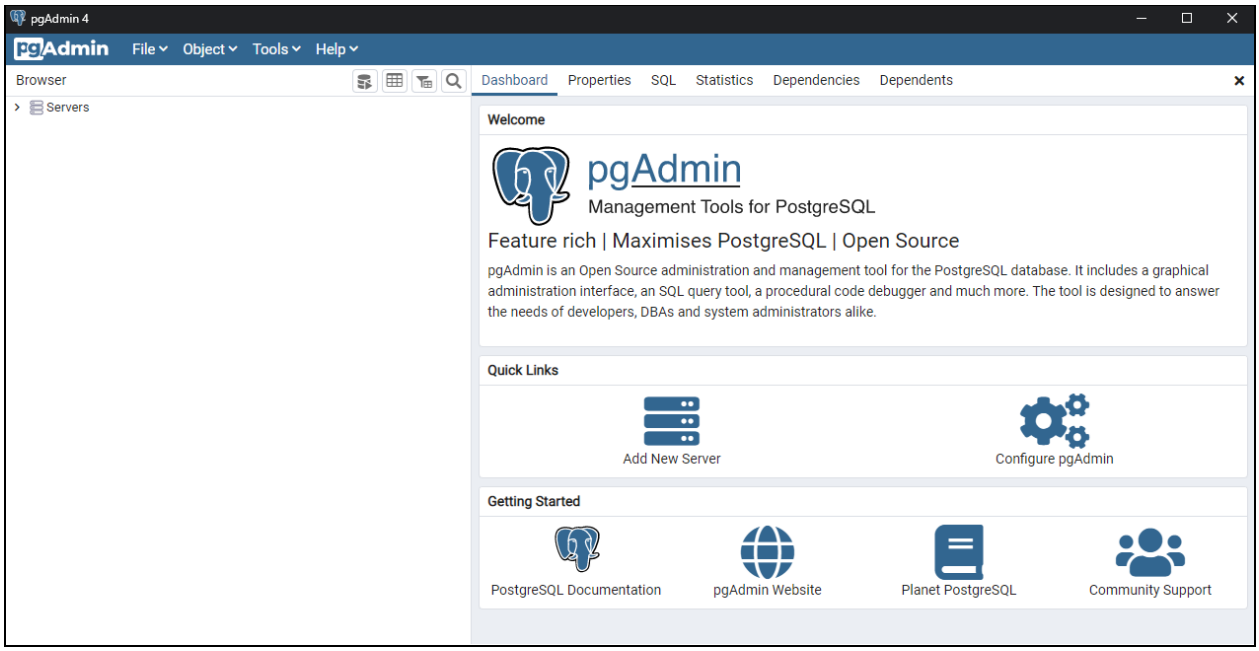
Konfiguracja aplikacji pgAdmin

Aplikacja pgAdmin to jeden z komponentów zainstalowanego pakietu, dedykowany zarządzaniu zawartością bazy danych. Zanim przystąpimy do pracy, musimy ją skonfigurować.

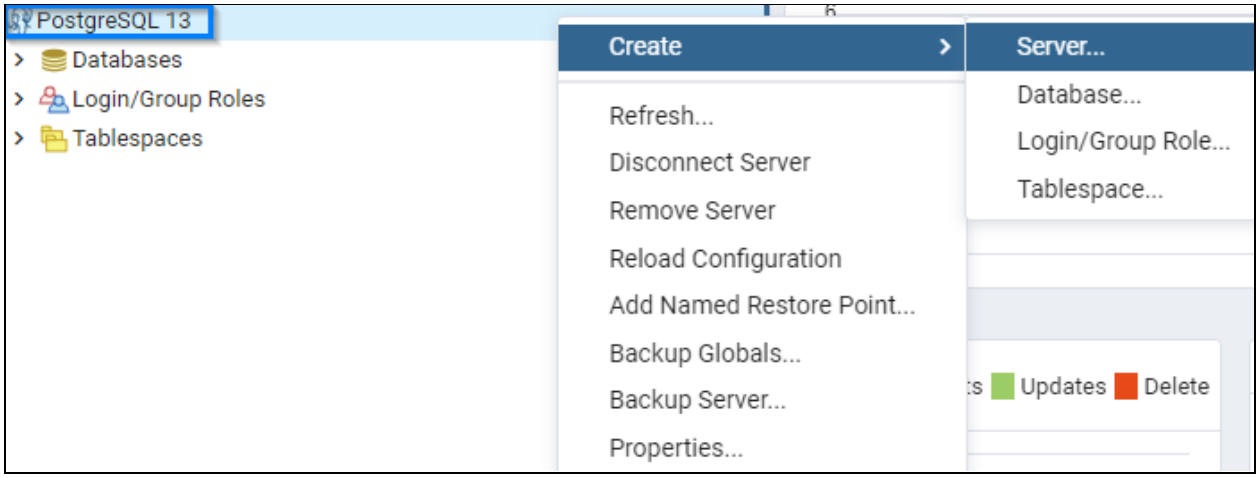
Najprostszym sposobem na uruchomienie aplikacji jest rozwinięcie menu *Start* systemu Windows (), wpisanie pgAdmin i wybór odpowiedniej pozycji z listy:



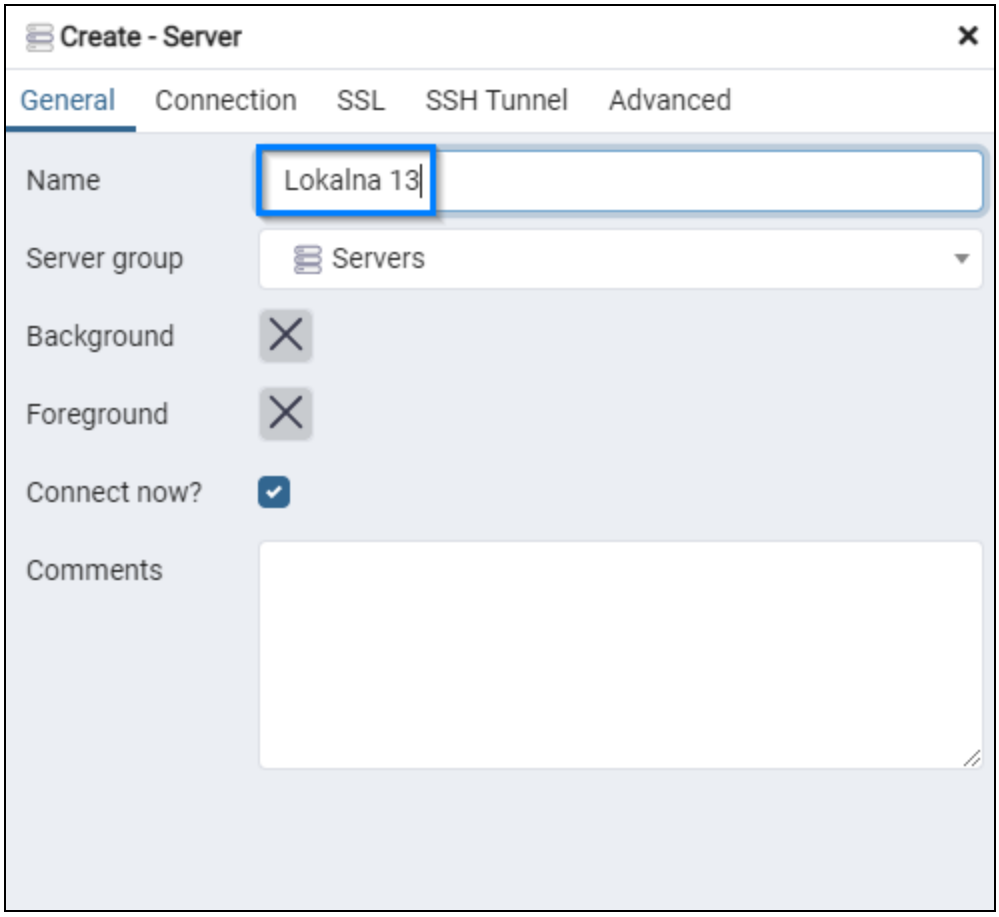
Po ponownym wpisaniu hasła zabezpieczającego (może być to ponownie fraza “gis”) otworzy się nowa karta przeglądarki, a w niej interfejs aplikacji:



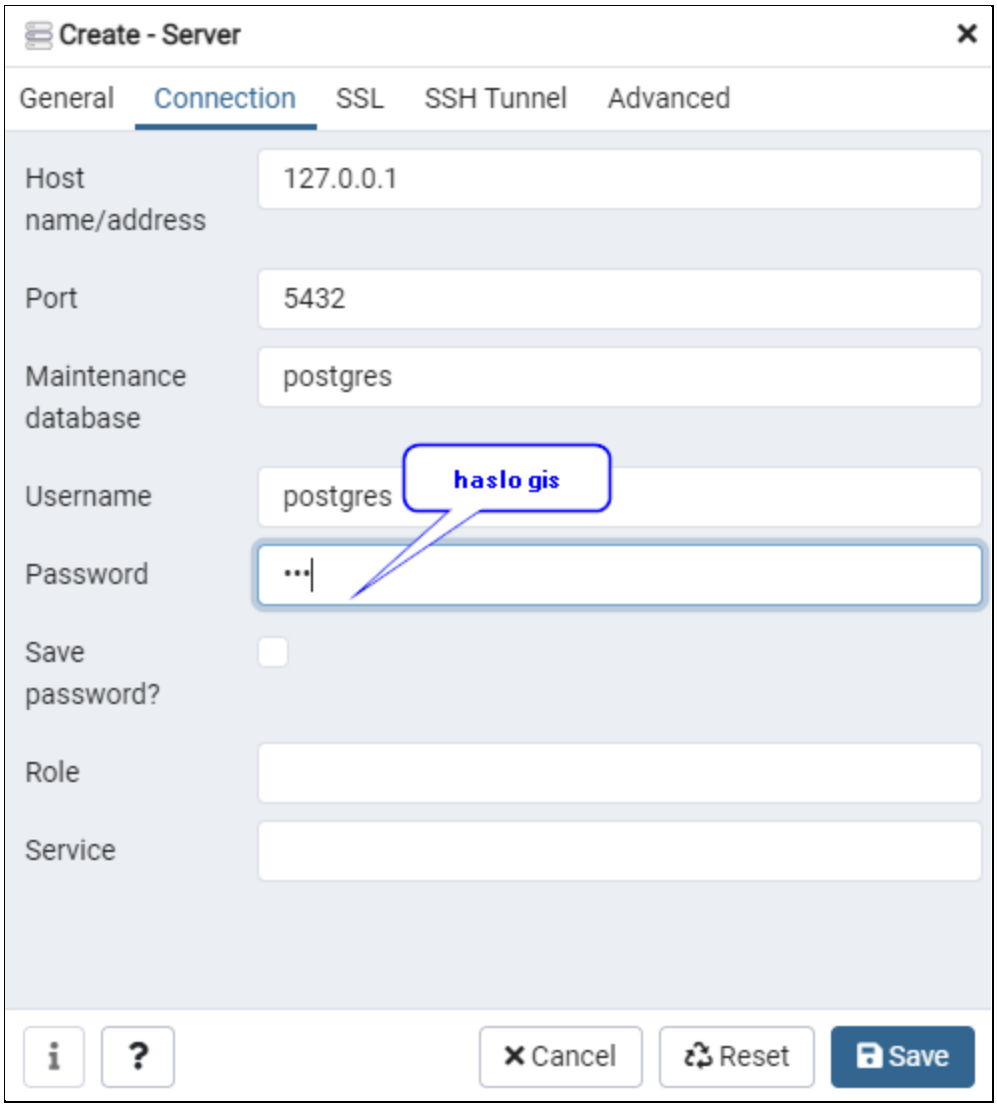
Nawiążemy teraz połączenie z wcześniej utworzoną bazą. W tym celu lokalizujemy naszą wersję bazy danych, umieszczoną w prawym górnym rogu, klikamy na nią prawym przyciskiem myszy i z rozwijanej listy wybieramy *Create -> Server*:



Pojawi się okno konfiguracji połączenia z serwerem. Zaczynamy od wypełnienia pól w domyślnie zaznaczonej zakładce *General*. Na wstępie musimy podać nazwę serwera – przykładowo ”Lokalna 13”:

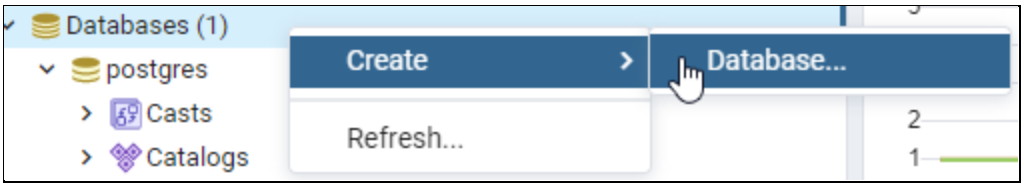


Następnie przechodzimy do zakładki *Connection*, gdzie należy wpisać nazwę/adres hosta (127.0.0.1), określić port (domyślnie 5432) oraz podać hasło (jeśli zastosowali się Państwo do wcześniejszych wytycznych, będzie to “gis”):

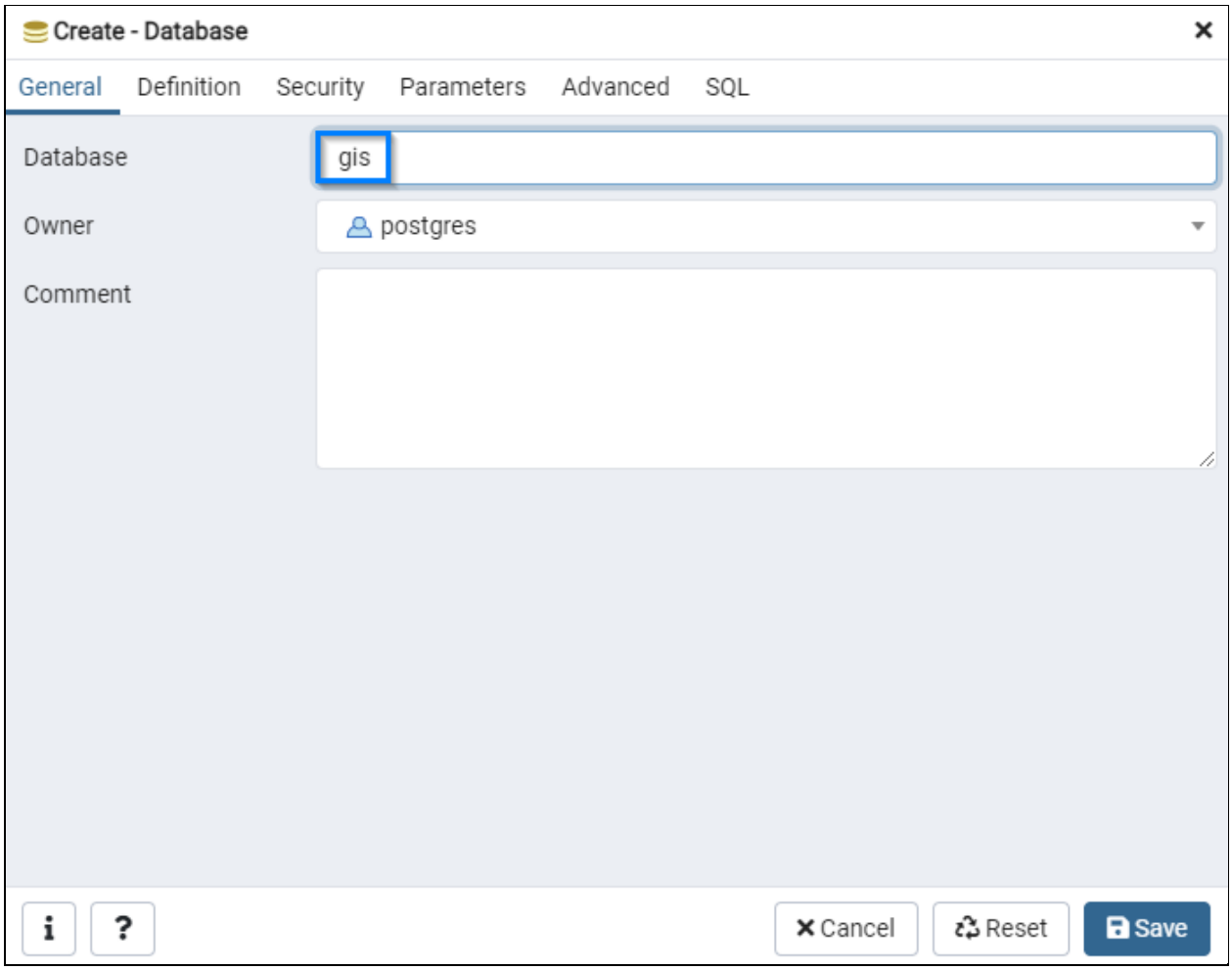


Po wykonaniu niezbędnych kroków klikamy na *Save*.

Na razie dysponujemy tylko jedną bazą danych o nazwie *postgres*. Na potrzeby szkolenia utworzymy teraz nową bazę, korzystając z interfejsu graficznego aplikacji pgAdmin. Klikamy prawym przyciskiem myszy na *Databases*, wybieramy *Create* i *Database*:

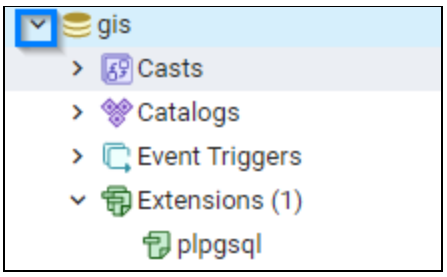


Na tym etapie możemy w zasadzie ograniczyć się do podania nazwy bazy - “gis”. Odpowiednie okno znajdziemy w domyślnie zaznaczonej zakładce *General*.



Tworzymy bazę, klikając na *Save*.

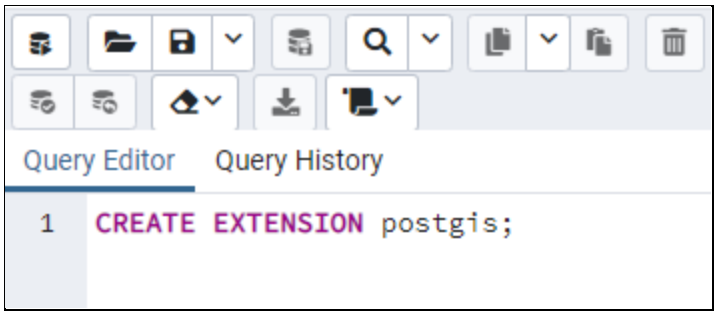
Baza danych *gis* powinna pojawić się na liście po lewej stronie. Aby rozwinąć jej zawartość, należy kliknąć lewym przyciskiem myszy na strzałce umieszczonej na lewo od nazwy bazy:



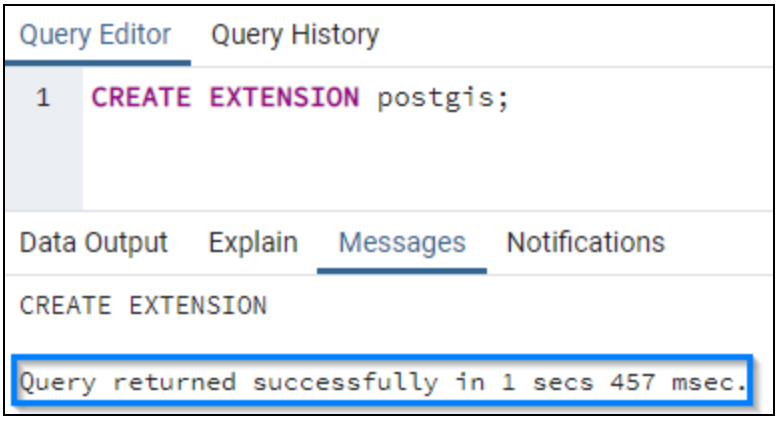
W dalszej części szkolenia będziemy wykonywali ćwiczenia na danych posiadających odniesienie przestrzenne. Rozszerzenie umożliwiające przeprowadzanie tego typu działań, tj. POSTGIS, możemy aktywować już teraz. Do tego celu wykorzystamy narzędzie o nazwie *Query*

Tool, dzięki któremu możemy zarządzać konfiguracją i zawartością bazy danych poprzez wpisywanie odpowiednich komend. Okno Query Tool otwieramy klikając na ikonkę:

W oknie narzędzia wpisujemy:



Instrukcja zostanie wykonana po wciśnięciu klawisza *F5*.



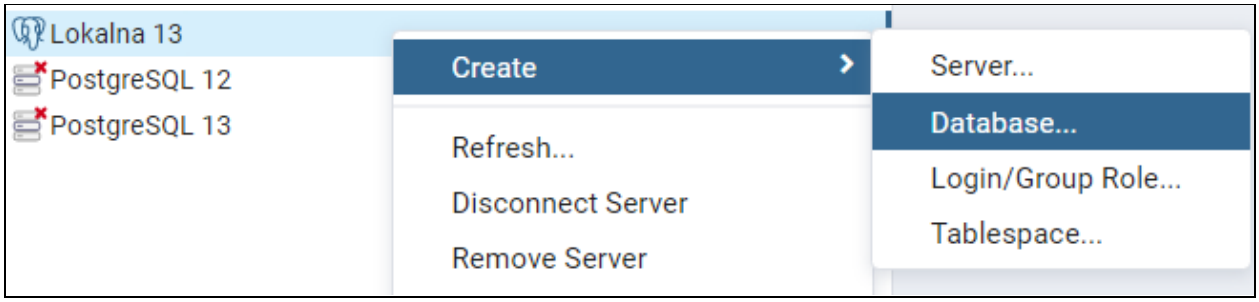
Omówienie funkcji zarządzania danymi z poziomu interfejsu pgAdmin4

Zasilenie bazy danymi z pliku zewnętrznego

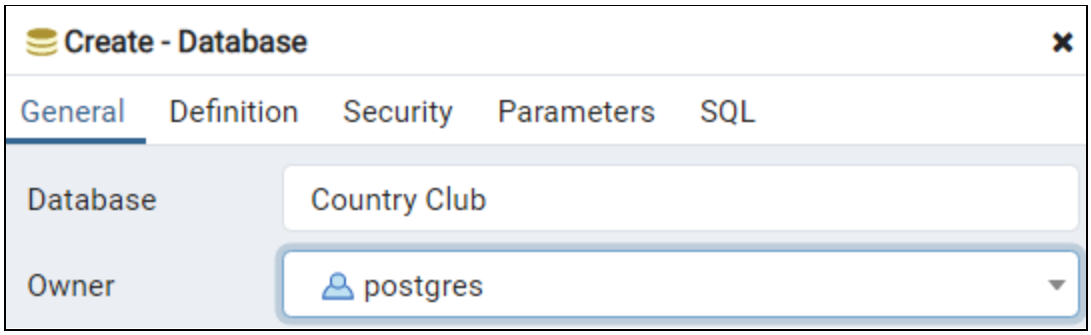
Zanim przejdziemy do wykonywania działań na danych przestrzennych, musimy poznać zasady konstruowania wyrażeń, za pomocą których będziemy wydobywać z bazy interesujące nas informacje. Potrzebujemy zatem zestawu danych treningowych.

W tym ćwiczeniu skorzystamy z jednego z gotowych zbiorów. Zawiera on informacje o członkach jednego z country clubów (organizacja o charakterze społeczno-sportowym, zwyczajowo zrzeszająca bogatych Amerykanów).

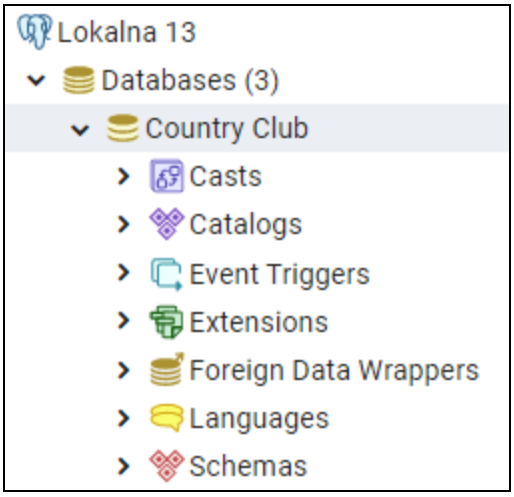
Plik SQL można pobrać stąd (**LINK** - **UWAGA! - ten sam zestaw danych znajdą Państwo w katalogu z materiałami szkoleniowymi**). Nosi on nazwę *clubdata* i zapisany jest w formacie .sql. Zawarte w nim dane zapiszemy w nowej bazie, którą nazwiemy *country club*. Klikamy prawym przyciskiem myszy na *Lokalna 13* i z podręcznego menu wybieramy *Create*, a następnie *Database*:



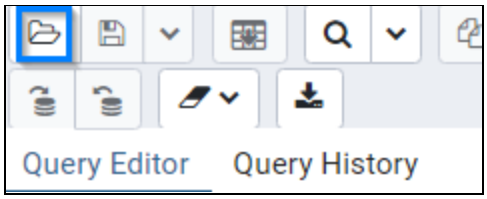
W nowym oknie wprowadzamy nazwę bazy, wskazujemy użytkownika - ownera i klikamy na *Save*:



Nowoutworzona baza powinna być już widoczna na liście dostępnych elementów:

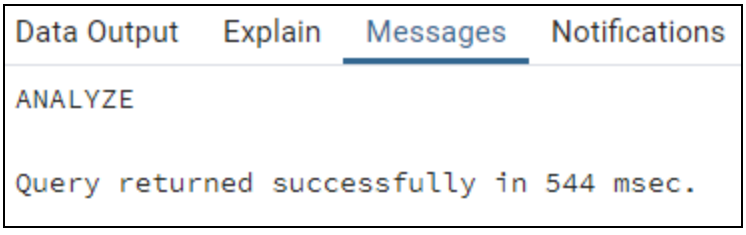


Aby zapełnić naszą bazę treścią, skorzystamy z gotowej instrukcji udostępnionej w pobranym wcześniej pliku .sql. Klikamy na ikonkę  i przechodzimy do okna narzędzia *Query Tool*. W górnej części panelu lokalizujemy i klikamy lewym przyciskiem myszy na ikonce *Open file*:

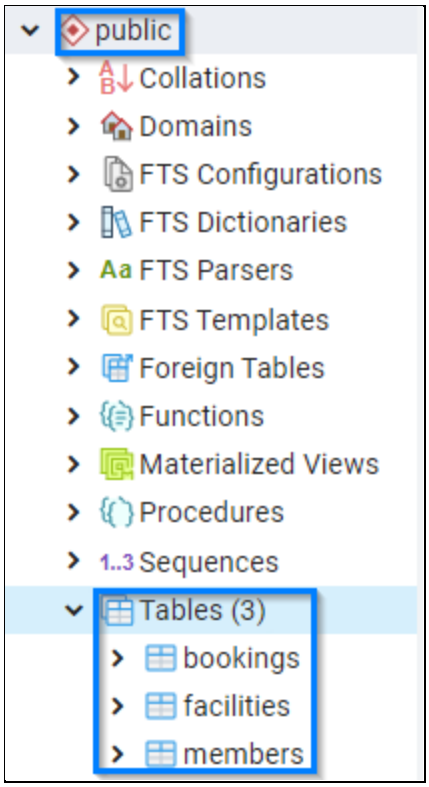


Przechodzimy do katalogu źródłowego, wskazujemy plik *club_data.sql* i klikamy na *Select* w prawym dolnym rogu:

Wciskamy klawisz F5, aby utworzyć nowy schemat. Po chwili w oknie pod polem zawierającym instrukcję pojawi się komunikat o pomyślnym zakończeniu działania:



Zestaw danych szkoleniowych powinien być już widoczny w formie tabel w SCHEMA *public* w bazie danych *Country Club*:



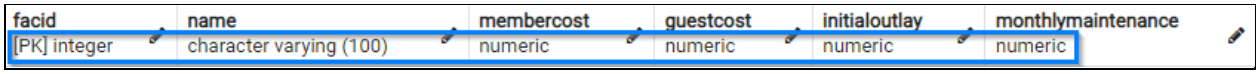
Struktura bazy danych w systemie PostgreSQL

Import ćwiczeniowej bazy danych do systemu PostgreSQL zakończył się powodzeniem. Jest to więc dobry moment, by przyjrzeć się bliżej jej strukturze. Sama baza danych stanowi izolowany magazyn informacji, umieszczony wewnątrz nadrzędnej formacji zwanej klastrem. Klaster może zawierać kilka baz danych. Określenie “izolowany” odnosi się natomiast do braku możliwości analizowania relacji między danymi umieszczonymi w odrębnych bazach.

Bazy danych zawierają jeden lub więcej SCHEMATÓW. Schemat to swoista “przegródka” w bazie danych, umożliwiająca grupowanie informacji. W przeciwieństwie do baz możliwe jest analizowanie relacji między zbiorami danych umieszczonymi w różnych schematach. Każda baza danych posiada domyślny schemat o nazwie *public*. Ponadto możliwe jest tworzenie nowych schematów i wraz definicją uprawnień w zakresie wyświetlania i edycji danych, przypisywanych poszczególnym użytkownikom.

Podstawową strukturą wchodzącą w skład schematu jest TABELA. Tabela to zbiór autonomicznych danych (a więc takich, które nie są wynikiem przetworzenia informacji przechowywanych w innych tabelach), trwale zapisywanych na twardym dysku.

Tabela ma charakterystyczną strukturę, na którą składają się wiersze i kolumny. Każda kolumna ma przypisany określony typ danych, który decyduje o tym, jakimi informacjami można, a jakimi nie można zasilić poszczególne komórki tabeli. Typ danych możemy odczytać m.in. z widoku tabeli wyświetlanego w aplikacji pgAdmin.



Do najczęściej wykorzystywanych typów danych należą:

Integer - liczby całkowite, przechowuje dane w zakresie od -2*10^9 do 2*10^9.
DateTime (Timestamp) - zawiera informację o dacie i czasie. Jeżeli czas nie został podany, jest automatycznie ustawiany na północ.
Date - tylko data, bez znacznika czasu
Varchar (character varying) - dane tekstowe. Liczba w nawiasie określa długość danego pola
Text - nie ma możliwości ograniczenia długości pola. Obowiązują zasady ogólne - wartość w komórce nie może przekroczyć 1gb.
Numeric - dane z liczbami dziesiętnymi. Zdefiniowana precyzja, liczba miejsc po przecinku jest stała.
Float (double precision) - przechowuje dane zmiennoprzecinkowe.

Tabele mogą dodatkowo zawierać ograniczenia, czyli **CONSTRAINTS**. Aby lepiej zrozumieć, na czym rzeczone ograniczenia polegają, kliknijmy lewym przyciskiem myszy na tabeli *bookings* i wybierzmy widok SQL:

Zwróćmy uwagę na część instrukcji zawierającą frazę **CONSTRAINT**:

```
CONSTRAINT bookings_pk PRIMARY KEY (bookid),
CONSTRAINT fk_bookings_facid FOREIGN KEY (facid)
```

Pierwsze z nałożonych na tabelę ograniczeń to tzw. **PRIMARY KEY**, czyli klucz główny. Jest to kolumna, której wartości jednoznacznie definiują każdy wiersz występujący w tabeli. Poprawny klucz główny wymusza pewne ograniczenia:

- Kolumna musi zawierać unikalną wartość dla każdego wiersza;

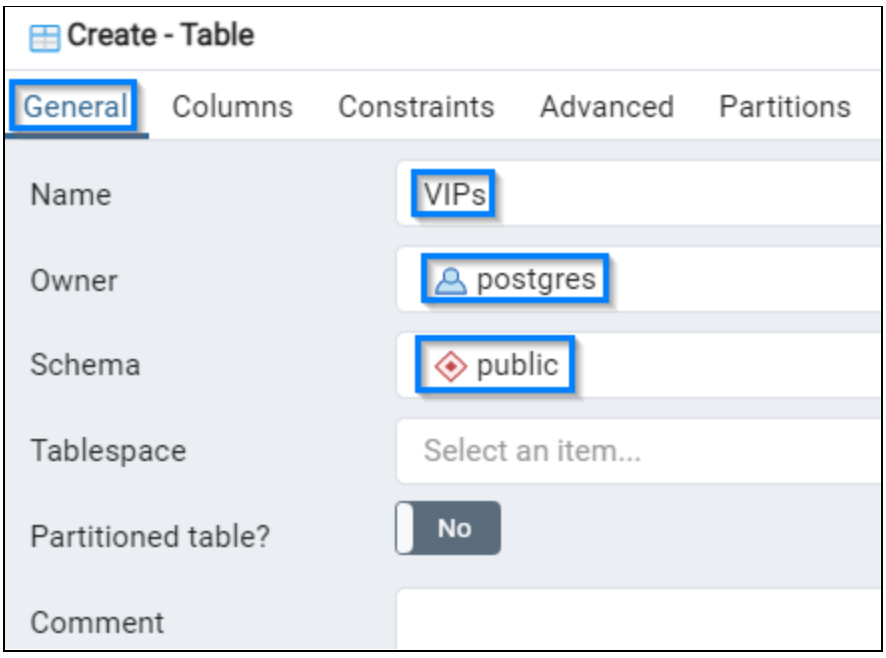
- Kolumna nie może zawierać brakujących wartości - innymi słowy, nie może pozostać pusta.

FOREIGN KEY, czyli klucz obcy, to kolumna której wartości odnoszą się do wartości z kolumny pełniącej rolę klucza głównego w innej tabeli. W przeciwieństwie do klucza głównego wiersze klucza obcego mogą być puste lub zawierać zduplikowane wartości.

Spróbujmy teraz w ramach ćwiczenia przygotować tabelę z danymi VIPów z *Country Clubu*, dla której utworzymy klucz główny i obcy.

Zaczynamy od kliknięcia prawym przyciskiem myszy na *Tables*, następnie wybieramy *Create* i *Table*:

Wyświetli się okno kreatora tabeli. Podajemy nazwę tworzonej tabeli, określamy jej właściciela i schemat:

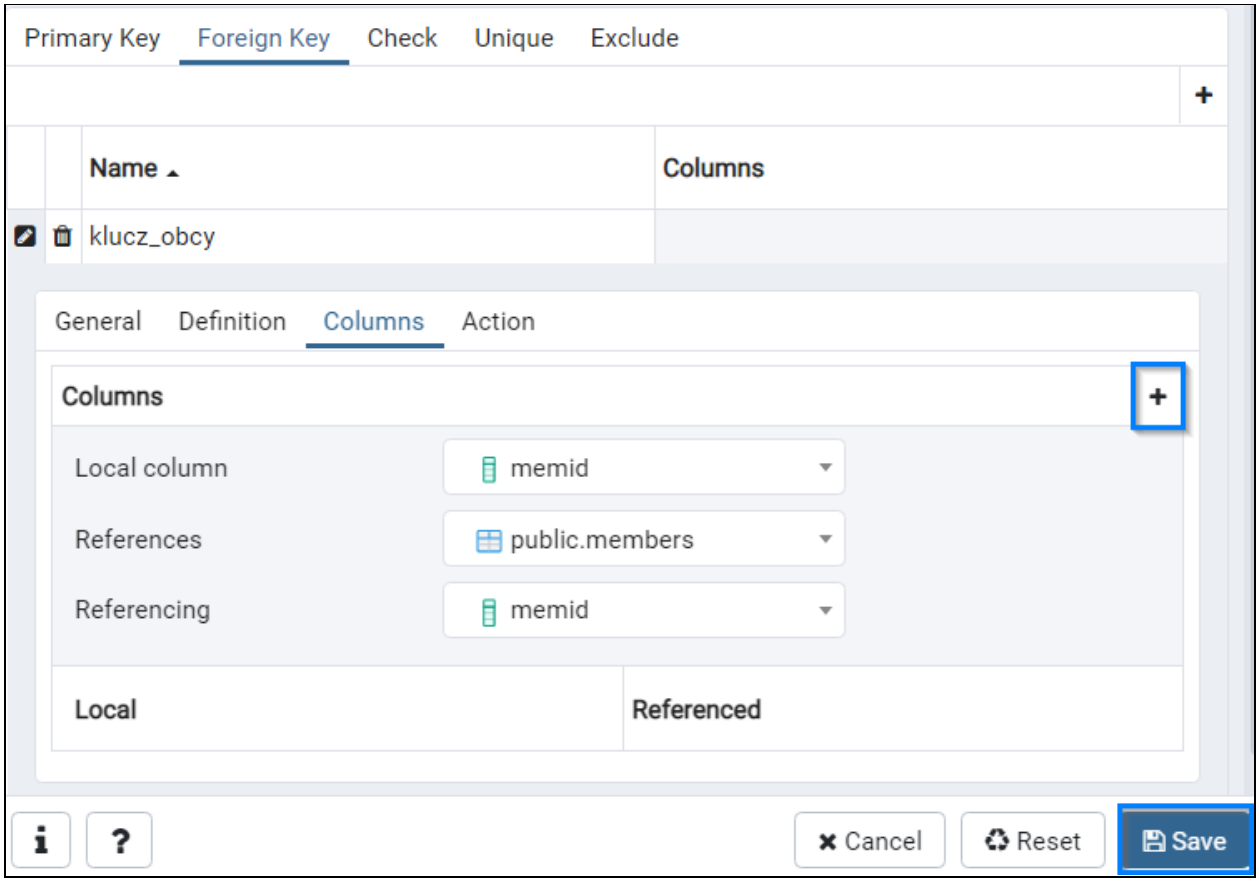


Przechodzimy do zakładki *Columns* w górnej części okna i przystępujemy do tworzenia kolumn.

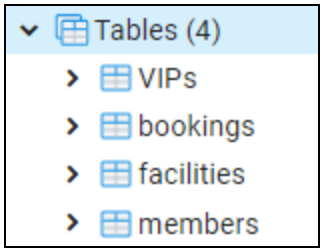
Aby dodać kolumnę do projektu, klikamy na + . Nadajemy jej nazwę “id”, typ danych określamy jako *serial* (pole liczbowe z autonumeracją). Zaznaczamy również, że kolumna nie może

zawierać wartości NULL i stanowić będzie klucz główny tabeli. Dodatkowo tworzymy dwie kolejne kolumny, zawierajace odpowiednio imię i nazwisko VIPa oraz jego unikalny numer identyfikacyjny (będzie to klucz obcy, nawiązujący do kolumny “memid” z tabeli *members*):

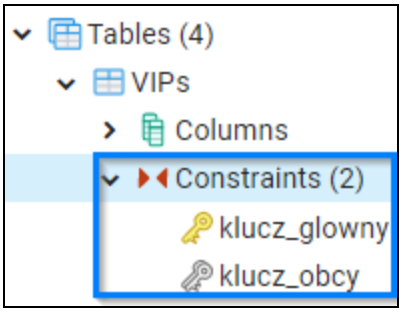
Teraz przechodzimy do zdefiniowania klucza obcego. Wybieramy zakładkę *Constraints* w górnej części okna i uzupełniamy poszczególne pola według poniższego wzoru:



Aby ostatecznie dodać kolumnę z kluczem obcym, klikamy na ikonkę “plusa” zaznaczoną na obrazku kolorem niebieskim. Tabelę tworzymy i zapisujemy, klikając na *Save*. Nowa tabela powinna pojawić się na rozwijanej liście po lewej stronie:



Utworzone ograniczenia w postaci kluczy można także podejrzeć z poziomu zakładki dostępnej z rozwijanej listy po lewej stronie:



Wyświetlanie i edycja zawartości tabel

Spróbujmy teraz wyświetlić zawartość tabeli *members*. Klikamy na jej nazwie prawym przyciskiem myszy i z podręcznego menu wybieramy *View/Edit Data*, a następnie *First 100 Rows*. Oczywiście nic nie stoi na przeszkodzie, by skorzystać z opcji *All Rows*, która zwróci nam wszystkie rekordy. Należy jednak mieć na uwadze, że w przypadku dużych baz operacja ta może trwać stosunkowo długo.

Widok tabeli wyświetli się w oknie po prawej stronie:

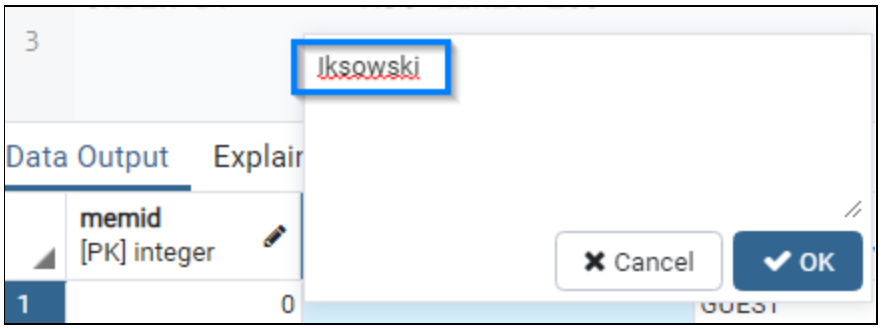
Query Editor | Query History | Scratch Pad

```
1  SELECT * FROM public.members
2  ORDER BY memid ASC LIMIT 100
3
```

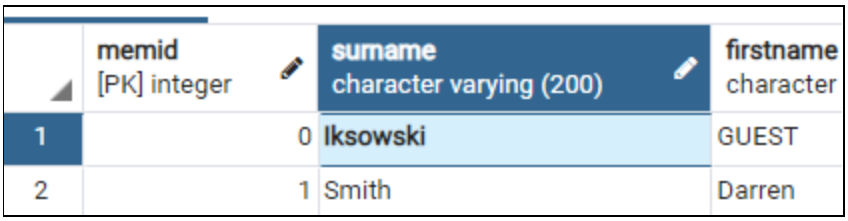
Data Output | Explain | Messages | Notifications

	memid [PK] integer	surname character varying (200)	firstname character varying (200)	address character varying (300)	zipcode integer	telephone character varying (20)
1	0	GUEST	GUEST	GUEST	0	(000) 000-0000
2	1	Smith	Darren	8 Bloomsbury Close, Boston	4321	555-555-5555
3	2	Smith	Tracy	8 Bloomsbury Close, New York	4321	555-555-5555
4	3	Rownam	Tim	23 Highway Way, Boston	23423	(844) 693-0723
5	4	Joplette	Janice	20 Crossing Road, New York	234	(833) 942-4710
6	5	Butters	Gerald	1065 Huntingdon Avenue, Bost...	56754	(844) 078-4130

Pierwszy rekord nie zawiera danych osobowych. Jest to więc dobra okazja, by zademonstrować Państwu, w jaki sposób edytować zawartość poszczególnych rekordów. Kliknijmy dwukrotnie lewym przyciskiem myszy na polu w kolumnie **surname**. Wyświetli się nowe okno, w którym można wprowadzić nową wartość - niech będzie to nazwisko “Iksowski”:



Po wprowadzeniu zmian klikamy na OK. Pogrubienie nowej wartości wskazuje, że zmiany w tabeli nie zostały zapisane:



Aby zapisać zmiany, klikamy lewym przyciskiem myszy na ikonkę *Save Data Changes* (funkcję tę można również wywołać wciskając klawisz F6):

W ramach praktyki uzupełnijmy pozostałe rubryki i zapiszmy zmiany (firstname: Max, address: Strawberry Hill 567, zipcode: 5432.

Z kolei aby dodać kolejny rekord do już istniejącej tabeli należy wyświetlić jej pełen widok. W tym celu z podręcznego menu (które, jak pamiętamy, wywołuje się klikając prawym przyciskiem myszy na nazwie tabeli) wybieramy opcję

Używając suwaka po prawej stronie przesuwamy widok tabeli w dół i odszukujemy ostatni rekord. Klikamy na przestrzeń pod ostatnim wierszem, “aktywując” zawartość należących do niego komórek:

Zmiany w tabeli zapisujemy analogicznie jak w poprzednim przykładzie (w tym przypadku nie jest to jednak konieczne).

Struktura i zastosowanie instrukcji SQL na przykładzie zbioru danych bez odniesienia przestrzennego

Do tej pory (za wyjątkiem aktywacji rozszerzenia POSTGIS) komunikowaliśmy się z bazą danych za pośrednictwem GUI aplikacji pgAdmin4. Dużo większe możliwości daje wykorzystanie zapytań pisanych w języku SQL.

Przykładowo, omówioną wcześniej aktualizację danych w tabeli można przeprowadzić także poprzez uruchomienie kodu sformatowanego według poniższego schematu:

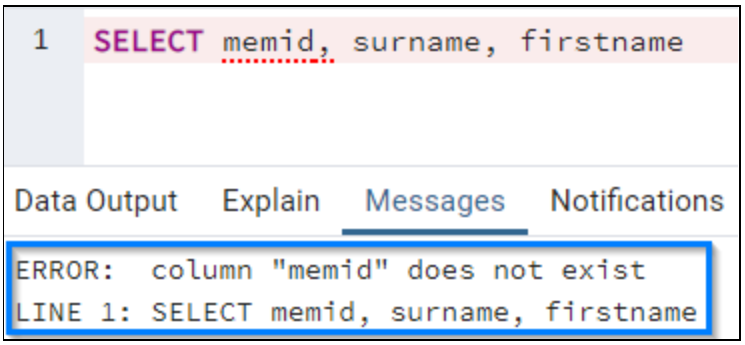
UPDATE nazwa_tabeli
SET nazwa_kolumny = 'nowa_wartość'
WHERE nr_porządkowy = 2
RETURNING *;

Kilka słów wyjaśnienia: klauzula **UPDATE** wskazuje, że celem kwerendy jest aktualizacja zawartości tabeli. Stosując komendę **SET** definiujemy nazwę kolumny, w której ma nastąpić zmiana oraz nową wartość. Jeśli zależy nam na aktualizacji konkretnego wiersza, możemy zastosować klauzulę **WHERE** i zdefiniować warunek. W tym konkretnym przypadku dążymy do zmiany w rekordzie o numerze porządkowym 2.
Komenda **RETURNING *** zwraca natomiast wszystkie zaktualizowane wiersze.

Zanim jednak przejdziemy dalej, odpowiedzmy sobie jasno, co kryje się za kodem. W dużym uproszczeniu można przyjąć, iż zawiera on ustrukturyzowane instrukcje, przechowujące informacje o tym z jakich kolumn i tabel będą pozyskiwane dane. Dodatkowo w zapytaniu możemy umieścić sformułowania wprowadzające bardziej szczegółowe warunki selekcji (np. liczby w określonym przedziale), ograniczyć liczbę zwracanych rekordów, lub grupować dane (np. suma liczby ludności z wszystkich gmin należących do tego samego powiatu). Jedną z ważniejszych funkcjonalności, jakie oferuje język SQL, jest łączenie danych z różnych tabel. Szczegółowe informacje na temat poszczególnych elementów zapytania SQL zostaną przedstawione w kolejnych częściach skryptu. Zanim jednak do tego przejdziemy, przyjrzyjmy się podstawowej konstrukcji takiego zapytania:

SELECT nazwa_kolumny_1, nazwa_kolumny_2
FROM nazwa_tabeli

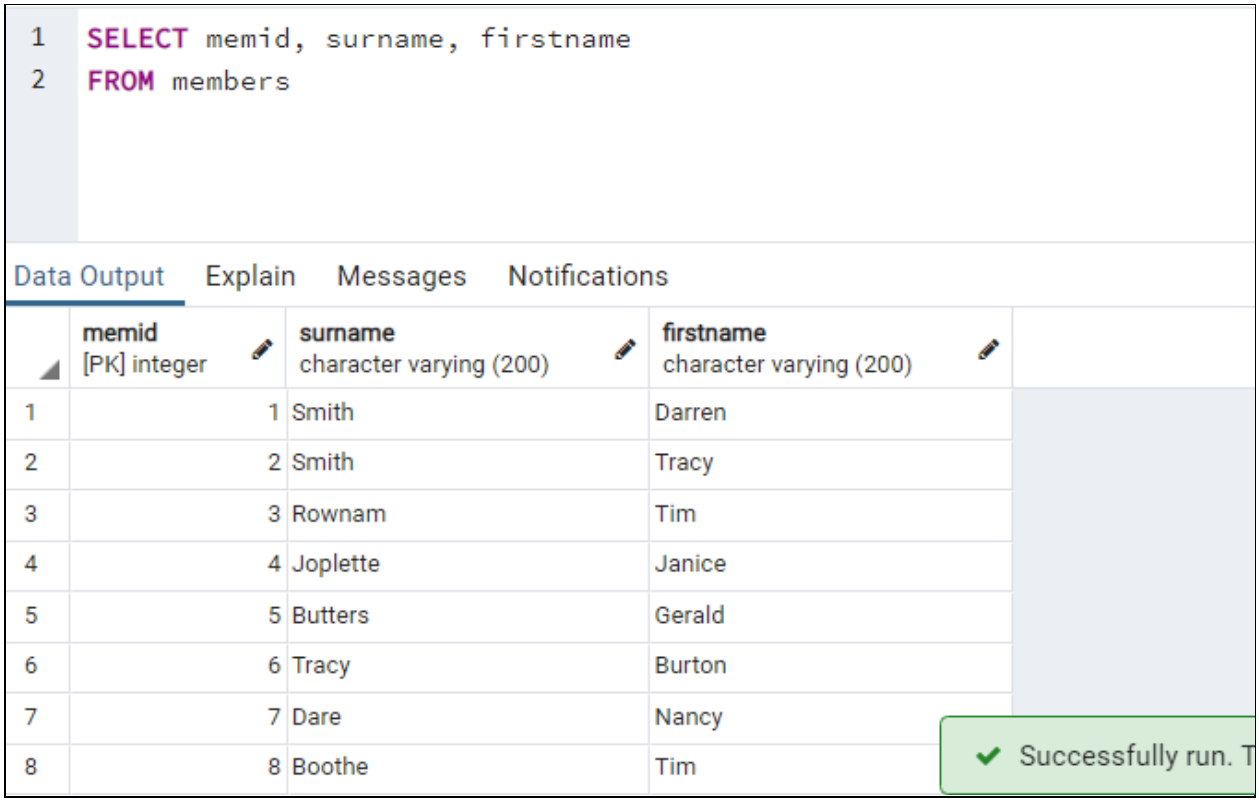
Klauzule **SELECT** i **FROM** stanowią niezbędną część każdego zapytania. Bez jednej bądź drugiej instrukcja nie zostanie wykonana:



Ilustracja powyżej przedstawia niepełną kwerendę, pozbawioną klauzuli FROM. Nawet jeśli nazwy kolumn, z których chcemy pozyskać dane, zapisane są poprawnie, to ze względu na brak bezpośredniego wskazania tabeli, instrukcja nie może zostać wykonana. Spróbujmy w takim razie usprawnić naszą kwerendę, dodając do niej brakującą informację w postaci nazwy tabeli - należy ją wpisać po klauzuli FROM:

```
SELECT memid, surname, firstname
FROM members
```

Spróbujmy wykonać powyższe zapytanie z poziomu okna narzędzia *Query Tool* (dla przypomnienia - ikonka ). Tym razem zapytanie zwróci oczekiwany wynik:



Tak jak wspomniano wcześniej, klauzule **SELECT** i **FROM** są niezbędne do wykonania zapytania, niemniej wachlarz dostępnych opcji jest zdecydowanie szerszy. Zanim jednak przejdziemy do ich prezentacji, rozważmy jeszcze jeden ciekawy przypadek. Załóżmy mianowicie, że chcemy wywołać pełen widok wybranej tabeli - tj. wszystkie wiersze i kolumny - bez korzystania z rozwijanego, podręcznego menu. We wcześniejszym przykładzie “wyciągnęliśmy” wszystkie rekordy z trzech wybranych kolumn jednej tabeli. Jak można się domyślić, wpisywanie nazw wszystkich kolumn z interesującej nas tabeli nie stanowi optymalnego rozwiązania problemu, choć z technicznego punktu widzenia jest to oczywiście

wykonalne. Niemniej w podobnych przypadkach dużo wygodniej zastąpić długi zapis nazw atrybutów znakiem *:

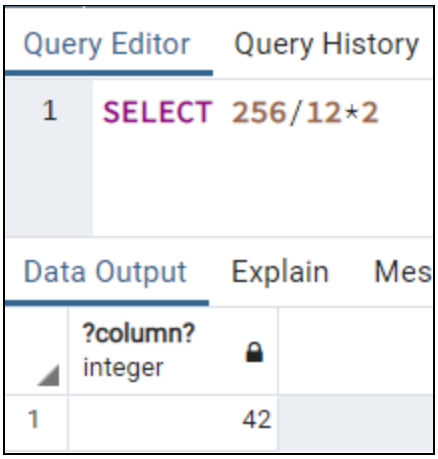
```
SELECT *
FROM members
```

W wyniku tak sformułowanego zapytania otrzymamy zestawienie wszystkich wierszy i kolumn tabeli o nazwie podanej po klauzuli **FROM**.

Zapytanie **SELECT** służy więc do pobierania danych z pojedynczej lub kilku tabel (do tej operacji przejdziemy jednak nieco później). Zaznaczono również, że do poprawnego wykonania zapytania zapisanego w języku SQL oprócz klauzuli **SELECT** należy podać **FROM**. Nie jest to jednak bezwzględnie obowiązująca reguła, gdyż odnosi się do sytuacji, gdy chcemy pozyskać dane z tabeli. Możliwe jest jednak skonstruowanie zapytania imitującego działanie kalkulatora, w którego przypadku danymi wejściowymi będą dane liczbowe. Oto przykład:

```
SELECT 256/12*2
```

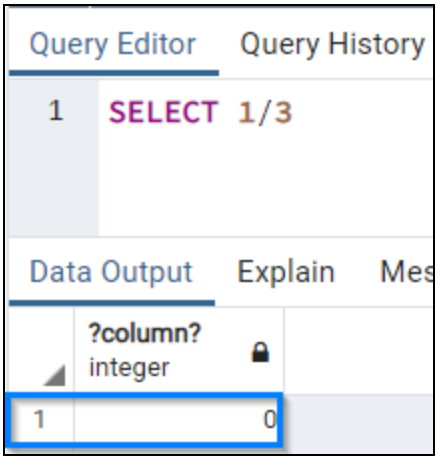
Po naciśnięciu klawisza F5 zostanie zwrócony następujący wynik:



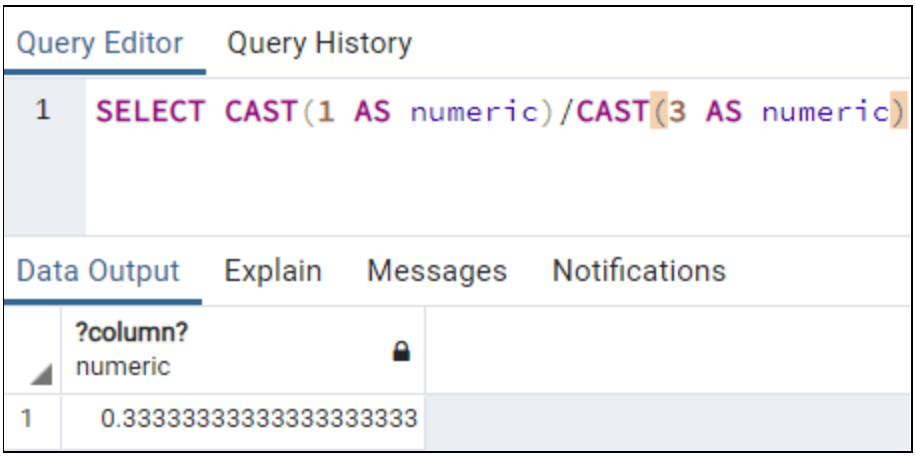
W efekcie otrzymaliśmy tabelę z jedną kolumną o niezdefiniowanej nazwie oraz wiersz z liczbą całkowitą, będącą wynikiem powyższego działania. Co jednak, jeśli wynikiem operacji arytmetycznej jest liczba dziesiętna? Sprawdźmy to na kolejnym przykładzie:

```
SELECT 1/3
```

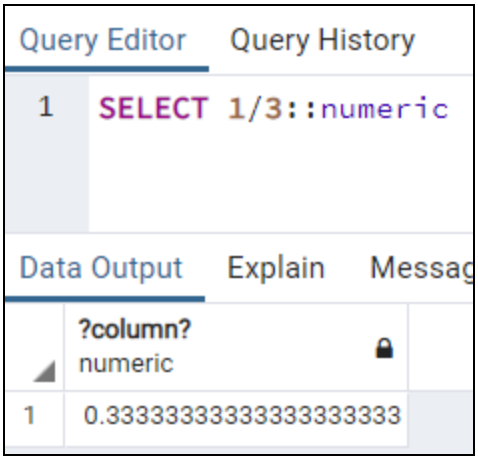
Tym razem wynik zdecydowanie odstaje od oczekiwań:



Instrukcja zwraca bowiem liczbę całkowitą, zaokrągloną dodatkowo do jednego miejsca po przecinku. Jeśli zależy nam na wyniku w postaci liczby dziesiętnej, musimy zastosować **RZUTOWANIE**, tj. ,konwersję typów. Oto przykład:



Ten sam efekt uzyskamy stosując zapis charakterystyczny dla dialektu PostgreSQL:



Filtrowanie danych przy użyciu klauzuli WHERE

Zastosowanie klauzuli **WHERE** w zapytaniu pozwala na doprecyzowanie kryteriów wyboru danych. Takie dopasowanie może odnosić się zarówno do określonej wartości, zakresu wartości lub wielu wartości określonych przez operator. Możliwe jest również wykluczenie określonych rekordów poprzez zastosowanie odpowiednio sformułowanego wyrażenia. W standardowej składni SQL (jak również w przypadku dialektu PostgreSQL) klauzula **WHERE** pojawia się po klauzuli **FROM**, np.:

SELECT typ_uslugi, abonament
FROM dane
WHERE abonament > 150

Przećwiczmy zastosowanie klauzuli **WHERE** na kilku przykładach. Zacznijmy od tabeli *facilities*. Spróbujmy wyświetlić tylko te rekordy, w których przypadku koszt skorzystania z usługi dla członków klubu jest większy niż 0.

Otwieramy narzędzie *Query Tool*, klikając na ikonę  . W treści polecenia nie podano dokładnie, jakie kolumny mają znaleźć się w zestawieniu wynikowym. Zakładamy więc, że chodzi o wszystkie atrybuty. Najwygodniej będzie więc skorzystać z symbolu ***** - jak pamiętamy, oznacza on bowiem zbiór wszystkich kolumn z danej tabeli.

SELECT *

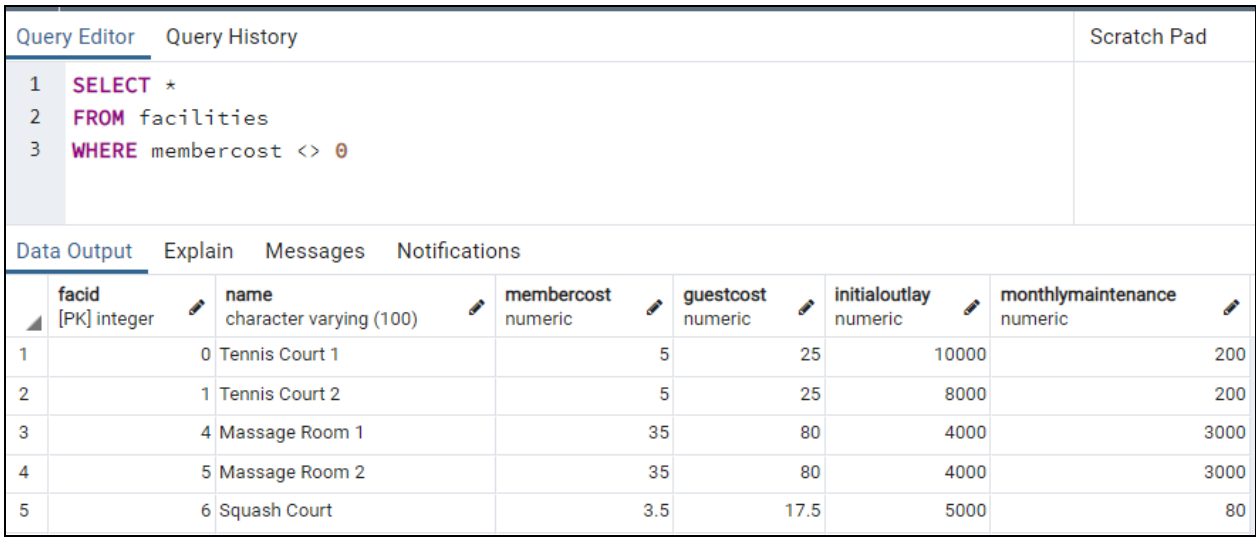
Następnie wprowadzamy klauzulę **FROM** wraz z nazwą tabeli, do której chcemy się odwołać:

SELECT *
FROM facilities

Na końcu zaś umieszczamy klauzulę **WHERE**. Zależy nam na wszystkich rekordach, które w kolumnie *membercost* przyjmują wartość inną niż 0. Warunek ten można zapisać na kilka sposobów, np. membercost <> 0, membercost != 0, lub po prostu membercost > 0:

```
SELECT *
FROM facilities
WHERE membercost <> 0
```

System powinien zwrócić 5 rekordów. Rzut oka na tabelę wynikową pozwoli zweryfikować, czy zapytanie zostało wykonane poprawnie:



Query Editor | Query History | Scratch Pad

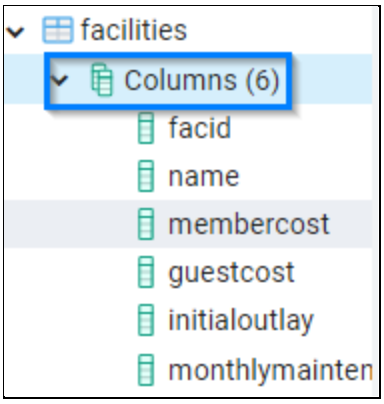
```
1  SELECT *
2  FROM facilities
3  WHERE membercost <> 0
```

Data Output | Explain | Messages | Notifications

	facid [PK] integer	name character varying (100)	membercost numeric	guestcost numeric	initialoutlay numeric	monthlymaintenance numeric
1	0	Tennis Court 1	5	25	10000	200
2	1	Tennis Court 2	5	25	8000	200
3	4	Massage Room 1	35	80	4000	3000
4	5	Massage Room 2	35	80	4000	3000
5	6	Squash Court	3.5	17.5	5000	80

W kolejnym ćwiczeniu z wykorzystania klauzuli **WHERE** porównamy wartości z dwóch odrębnych atrybutów. Tym razem chcemy wyświetlić tylko te rekordy, w których przypadku miesięczny abonament członkowski jest większy niż 0 i jednocześnie mniejszy niż 1/50 miesięcznego kosztu utrzymania obiektu sportowego. Dodatkowo w tabeli wynikowej powinny pojawić się cztery wybrane kolumny: id obiektu, nazwa obiektu, cena dla członków klubu oraz koszt miesięcznego utrzymania obiektu.

Tym razem po klauzuli **SELECT** musimy podać nazwy kolumn, które zostaną wyświetlone w tabeli wynikowej. Możemy podejrzeć nazwy kolumn w wybranej tabeli z poziomu jej zakładki, wyświetlającej się po lewej stronie okna aplikacji pgAdmin:



Pierwsza część zapytania powinna więc wyglądać następująco:

```
SELECT facid, name, membercost, monthlymaintenance
FROM facilities
```

Jak jednak zapisać zdefiniowany w treści zadania warunek, a więc koszt dla członków mniejszy niż 1/50 miesięcznego kosztu utrzymania obiektu? W tym przypadku można potraktować odpowiednie kolumny jako ekwiwalenty zmiennych, a więc:

```
SELECT facid, name, membercost, monthlymaintenance
FROM facilities
WHERE membercost < monthlymaintenance / 50
```

Pamiętamy również o dodatkowym warunku - wartość kolumny membercost nie może wynosić 0. Aby dodać dodatkowe kryterium, posłużymy się operatorem **AND**:

```
SELECT facid, name, membercost, monthlymaintenance
FROM facilities
WHERE membercost < monthlymaintenance / 50
AND membercost != 0
```

Zapytanie powinno zwrócić tabelę z dwoma wierszami:

Query Editor | Query History

```
1
2  SELECT facid, name, membercost, monthlymaintenance
3  FROM facilities
4  WHERE membercost < monthlymaintenance / 50
5  AND membercost != 0
```

Data Output | Explain | Messages | Notifications

	facid [PK] integer	name character varying (100)	membercost numeric	monthlymaintenance numeric
1	4	Massage Room 1	35	3000
2	5	Massage Room 2	35	3000

Klauzuli **WHERE** możemy używać również do filtrowania danych tekstowych. Przykładowo, chcąc uzyskać informacje o obiektach sportowych, w których nazwie zawiera się fraza “Tennis”, moglibyśmy zastosować następujący zapis:

SELECT *
FROM facilities
WHERE name **LIKE** ‘%Tennis%’

Zarówno sposób wykorzystania operatora **LIKE**, jak i zapis wartości z wykorzystaniem apostrofów i znaku %, przywodzą na myśl kreator filtrowania programu QGIS. Przypomnijmy, że operator **LIKE,** będący częścią standardu **ANSI SQL**, ***rozróżnia małe i wielkie litery***. Natomiast operator **ILIKE**, jako specyficzny dla dialektu **PostgreSQL**, ***takiego rozróżnienia nie uwzględnia***. Znak procentu (%) to ***symbol wieloznaczny, odpowiadający jednemu bądź większej liczbie znaków***. Niekiedy stosuje się również znak podkreślenia (_) - również jest to ***symbol wieloznaczny, odpowiada jednak tylko jednemu znakowi. Zastosowanie pojedynczych apostrofów wskazuje, że mamy do czynienia z wartością z atrybutu*** *name*. Jest to więc sposób zapisu analogiczny do tego, jaki zaimplementowano w programie QGIS.

Query Editor | Query History | Scratch Pad

```
1  SELECT *
2  FROM facilities
3  WHERE name LIKE '%Tennis%'
4
```

Data Output | Explain | Messages | Notifications

	facid [PK] integer	name character varying (100)	membercost numeric	guestcost numeric	initialoutlay numeric	monthlymaintenance numeric
1	0	Tennis Court 1	5	25	10000	200
2	1	Tennis Court 2	5	25	8000	200
3	3	Table Tennis	0	5	320	10

Zastosowanie słowa kluczowego **AND** znacząco zwiększa użyteczność operatorów porównania. Nie jest to jednak jedyna dostępna opcja. Zdarza się, że z puli danych chcemy wyfiltrować wiersze, które przyjmują nie jedną określoną, lecz kilka różnych wartości. W takim przypadku mamy do czynienia z alternatywą, oznaczaną za pomocą **OR.** Sprawdźmy, jak przytoczone rozwiązanie sprawdza się w praktyce. Z tabeli *members* wybierzmy tylko te rekordy, gdzie nazwisko członkini lub członka klubu zaczyna się na literę “B” lub “C”:

SELECT firstname, surname
FROM members
WHERE surname **LIKE ‘**B%’
OR surname **LIKE ‘**C%’

Wynik zapytania zaprezentowano na ilustracji poniżej:

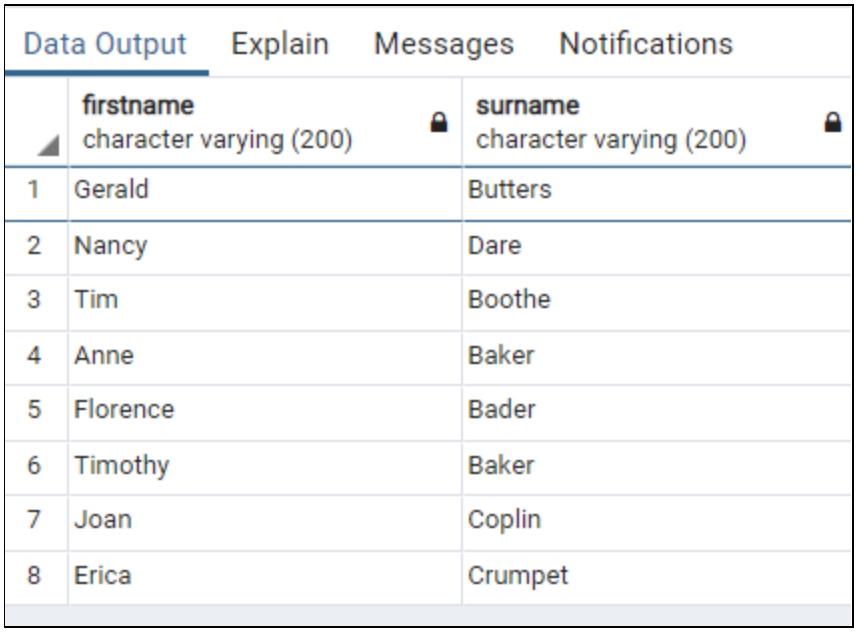
```
1  SELECT firstname, surname
2  FROM members
3  WHERE surname LIKE 'B%'
4  OR surname LIKE 'C%'
```

Data Output | Explain | Messages | Notifications

	firstname character varying (200)	surname character varying (200)
1	Gerald	Butters
2	Tim	Boothe
3	Anne	Baker
4	Florence	Bader
5	Timothy	Baker
6	Joan	Coplin
7	Erica	Crumpet

A co w sytuacji, gdy liczba alternatyw zdecydowanie przekracza dwie? Czy jesteśmy skazani na powielanie słowa kluczowego **OR**? Na szczęście istnieje inne, dużo wygodniejsze rozwiązanie, wykorzystujące sformułowanie **IN**. Tym razem w tabeli wynikowej chcielibyśmy wyświetlić dane osób, których nazwisko zaczyna się na literę B,C lub D. W tym przypadku skorzystamy dodatkowo z funkcji **LEFT()**, która przytnie nam wartość tekstową do określonej liczby znaków licząc od lewej strony:

SELECT firstname, surname
FROM members
WHERE LEFT(surname,1) **IN** ('B','C','D')



Data Output | Explain | Messages | Notifications

	firstname character varying (200)	surname character varying (200)
1	Gerald	Butters
2	Nancy	Dare
3	Tim	Boothe
4	Anne	Baker
5	Florence	Bader
6	Timothy	Baker
7	Joan	Coplin
8	Erica	Crumpet

To, co rzuca się w oczy, to brak uporządkowania danych wynikowych. Tę funkcjonalność przetestujemy jednak w kolejnych ćwiczeniach, gdyż wiąże się ona z zastosowaniem odrębnej klauzuli. Teraz zaś zajmiemy się wyrażeniem warunkowym, przy okazji omawiając zasady nadawania nazwom kolumn tak zwanych aliasów.

WYRAŻENIA WARUNKOWE

Na potrzeby kolejnego ćwiczenia załóżmy następujący scenariusz: zostaliśmy poproszeni o przygotowanie listy nazw obiektów sportowych wraz z przyporządkowaną im kategorią “tanie”/”drogie”, gdzie kategoria “drogie” odnosi się do obiektów, których miesięczny koszt utrzymania jest większy niż 100. W tym miejscu nasuwa się pytanie, w jaki sposób dokonać przeklasyfikowania wartości na wskazane w poleceniu kategorie? Do tego celu znakomicie nadaje się należąca do standardu ANSI SQL instrukcja **CASE**, którą część z Państwa może kojarzyć z *Kalkulatora Pól* programu QGIS. Od strony strukturalnej zapytanie wygląda następująco:

CASE WHEN warunek **THEN** rezultat
WHEN kolejny warunek **THEN** kolejny rezultat
ELSE inny rezultat
END

Jak widać, struktura nie różni się od zapisu, jaki stosuje się w programie QGIS. Zobaczmy zatem, w jaki sposób “wpleść” zaprezentowaną konstrukcję w zapytanie:

```
SELECT name AS nazwa_obiektu,

CASE WHEN monthlymaintenance > 100 THEN 'drogie'
        ELSE 'tanie'
END
AS koszt
FROM facilities
```

Zaczynamy od wskazania kolumny z nazwą obiektu. W tym przypadku zmieniamy jednak jej nazwę domyślną, stosując ***alias***. Jest to zdefiniowana przez użytkownika nazwa kolumny w tabeli wynikowej. W zapytaniu poprzedza ją słowo kluczowe **AS**. Aliasy stosuje się przede wszystkim w złączeniach danych z kilku tabel w celu uproszczenia kodu. W związku z powyższym zwyczajowo przyjmuje się, że długość aliasu nie powinna przekraczać dwóch znaków (DeBarros 2018, 96). Tutaj jednak celem jest zmiana nazwy kolumny, dlatego też pozwoliliśmy sobie na dłuższy zapis.

Po podaniu nazwy kolumny i aliasu wstawiamy przecinek, po którym wprowadzamy do instrukcji wyrażenie warunkowe. Obecność aliasu “koszt” pokazuje, że mamy tu w zasadzie do czynienia z odrębną instrukcją tworzenia kolumny, której wiersze zostaną uzupełnione zreklasyfikowanymi wartościami z kolumny źródłowej “monthlymaintenance”. Widok tabeli wynikowej powininen prezentować się jak na ilustracji poniżej:

Query Editor | Query History

```
1  SELECT name AS nazwa_obiektu,
2  CASE WHEN monthlymaintenance > 100 THEN 'drogie'
3      ELSE 'tanie' END
4  AS koszt
5  FROM facilities
```

Data Output | Explain | Messages | Notifications

	nazwa_obiektu character varying (100)	koszt text
1	Tennis Court 1	drogie
2	Tennis Court 2	drogie
3	Badminton Court	tanie
4	Table Tennis	tanie
5	Massage Room 1	drogie
6	Massage Room 2	drogie
7	Squash Court	tanie
8	Snooker Table	tanie

Daty w systemie PostgreSQL

Obsługa dat i godzin w PosgtreSQL uwzględnia cztery typy danych:

- **timestamp -** zapisuje datę i godzinę. Dodatkowo można posłużyć się słowem kluczowym **with time zone** - daje to pewność, że czas zarejestrowany dla dokumentowanego wydarzenia odnosi się do strefy czasowej, w której ono nastąpiło
- **date** - zapisuje tylko datę
- **time** - zapisuje wyłącznie godzinę
- **interval** - przechowuje wartość odpowiadającą jednostce czasu według zdefiniowanego typu przedziału czasowego, przykładowo 12 dni lub 8 godzin

Daty i godziny zapisywane są zazwyczaj w dedykowanym im standardzie ISO: YYYY-MM-DD HH-MM-SS. Inne formaty, np. MM/DD/YYYY również są obsługiwane.

W naszym kursie ograniczymy się do pracy z **timestamp**. Aby oswoić się z tym typem danych, wykonajmy ćwiczenie polegające na pozyskaniu listy członków *Country Clubu*, którzy należą do organizacji od września 2012. W tabeli wynikowej powinny znaleźć się kolumny z imieniem, nazwiskiem i datą przystąpienia do klubu. Aliasy nie są wymagane - możemy pozostać przy oryginalnych nazwach atrybutów.

SELECT surname, firstname, joindate
FROM members
WHERE joindate >= ‘2012-09-01’

Tabela wynikowa powinna prezentować się następująco:

Query Editor | Query History

```
1  SELECT  surname, firstname, joindate
2  FROM members
3  WHERE joindate >= '2012-09-01'
```

Data Output | Explain | Messages | Notifications

	surname character varying (200)	firstname character varying (200)	joindate timestamp without time zone
1	Sarwin	Ramnaresh	2012-09-01 08:44:42
2	Jones	Douglas	2012-09-02 18:43:05
3	Rumney	Henrietta	2012-09-05 08:42:35
4	Farrell	David	2012-09-15 08:22:05
5	Worthington-Smyth	Henry	2012-09-17 12:27:15
6	Purview	Millicent	2012-09-18 19:04:01
7	Tupperware	Hyacinth	2012-09-18 19:32:05
8	Hunt	John	2012-09-19 11:32:45

Porządkowanie danych z wykorzystaniem klauzul ORBER BY i LIMIT

Pozostańmy przez chwilę przy pozyskanym zbiorze. Dane wynikowe pogrupowane są w kolejności rosnącej według dat, które, jak widać, można filtrować przy użyciu operatorów tak samo jak wartości liczbowe - z tą jednak różnicą, że **timestamp** należy podawać w apostrofach.

Załóżmy jednak, że taki sposób porządkowania danych nie do końca nam odpowiada. Przykładowo chcielibyśmy wyświetlić ten sam zbiór danych według daty, ale tym razem w kolejności malejącej. Umożliwi nam to klauzula **ORDER BY**. Domyślnie zwraca ona dane w kolejności rosnącej, co można szerzej zapisać jako:

ORDER BY wartość **ASC** (od *ASCENDING*, czyli *rosnąco*)

Aby uzyskać uporządkowanie odwrotne, należy słowo **ASC** zastąpić **DESC** (od *DESCENDING*, czyli *malejąco*):

ORDER BY wartość **ASC** (od *ASCENDING*, czyli *rosnąco*)

Pełna instrukcja powinna wyglądać tak:

SELECT surname, firstname, joindate
FROM members
WHERE joindate >= ‘2012-09-01’
ORDER BY joindate **DESC**

I zwracać następujący wynik:

Query Editor | Query History

```
1  SELECT surname, firstname, joindate
2  FROM members
3  WHERE joindate >= '2012-09-01'
4  ORDER BY joindate DESC
```

Data Output | Explain | Messages | Notifications

	surname character varying (200)	firstname character varying (200)	joindate timestamp without time zone
1	Smith	Darren	2012-09-26 18:08:45
2	Crumpet	Erica	2012-09-22 08:36:38
3	Hunt	John	2012-09-19 11:32:45
4	Tupperware	Hyacinth	2012-09-18 19:32:05
5	Purview	Millicent	2012-09-18 19:04:01
6	Worthington-Smyth	Henry	2012-09-17 12:27:15
7	Farrell	David	2012-09-15 08:22:05
8	Rumney	Henrietta	2012-09-05 08:42:35

Ponadto często zdarza się, że z długiej puli wierszy interesuje nas zaledwie kilka wartości. W PostgreSQL możemy sami zdefiniować liczbę zwracanych rekordów stosując klauzulę **LIMIT**. Należy pamiętać, by umieścić ją na końcu wyrażenia, tak jak na poniższym przykładzie:

SELECT surname, firstname, joindate
FROM members
WHERE joindate >= '2012-09-01'
ORDER BY joindate **DESC**
LIMIT 5

Query Editor Query History

```
1  SELECT surname, firstname, joindate
2  FROM members
3  WHERE joindate >= '2012-09-01'
4  ORDER BY joindate DESC
5  LIMIT 5
```

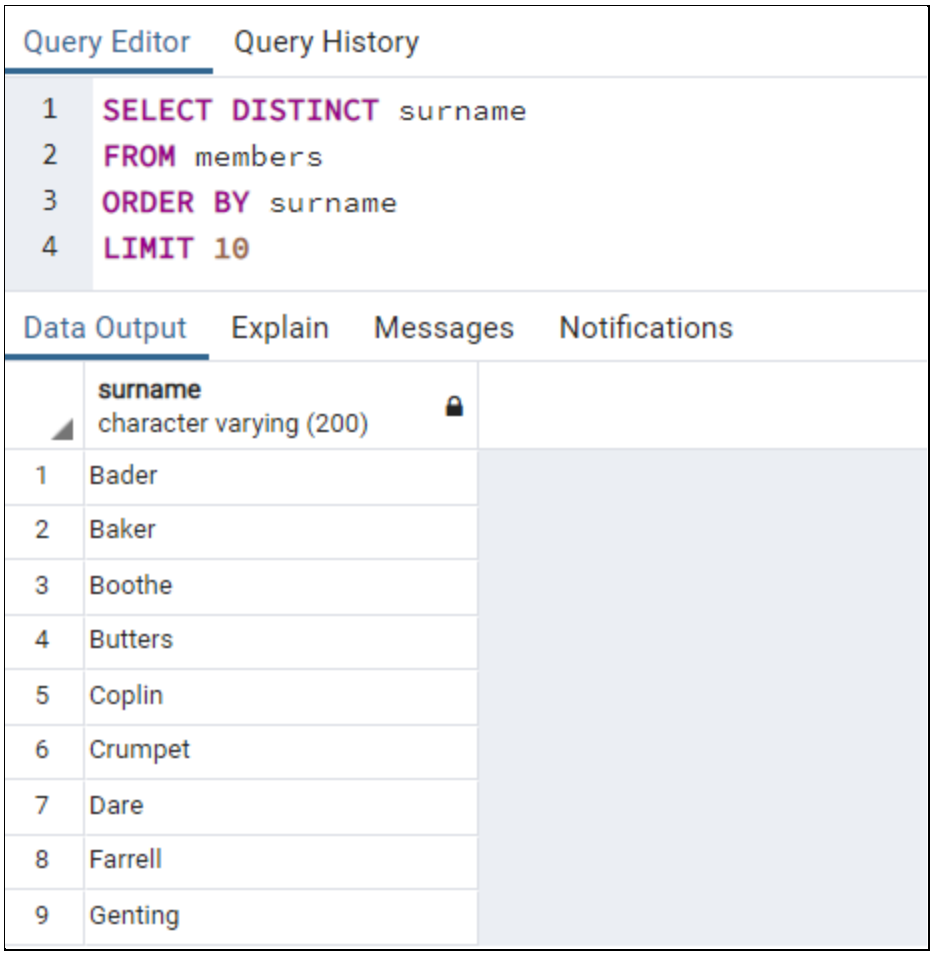
Data Output Explain Messages Notifications

	surname character varying (200)	firstname character varying (200)	joindate timestamp without time zone
1	Smith	Darren	2012-09-26 18:08:45
2	Crumpet	Erica	2012-09-22 08:36:38
3	Hunt	John	2012-09-19 11:32:45
4	Tupperware	Hyacinth	2012-09-18 19:32:05
5	Purview	Millicent	2012-09-18 19:04:01

Pozyskiwanie wartości unikalnych z wykorzystaniem klauzuli DISTINCT

Przeglądane przez nas tabele mogą zawierać kolumny z powtarzającymi się wartościami. Może to stanowić kłopot w sytuacji, gdy zależy nam wyłącznie na wykazie wartości unikalnych z danego atrybutu. Aby tego dokonać, należy zastosować klauzulę **DISTINCT**, którą umieszczamy zaraz po **SELECT**. Przedstawię działanie opisywanego rozwiązania na przykładzie kodu zwracającego pierwsze dziesięć nazwisk członków klubu, uporządkowanej według kolejności alfabetycznej. Dodatkowo zbiór wynikowy nie może zawierać duplikatów.

SELECT DISTINCT surname, firstname
FROM members
ORDER BY surname
LIMIT 10

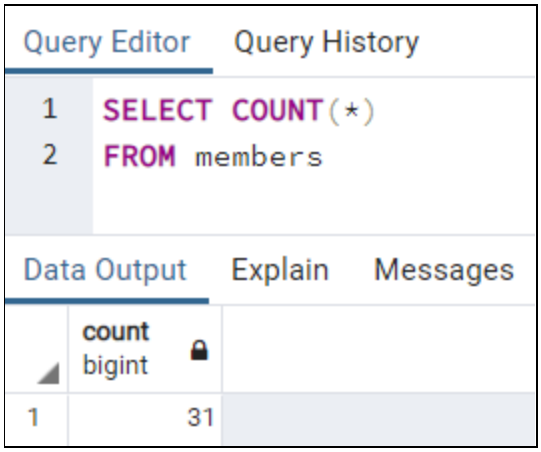


Funkcje agregujące

Oprócz wykonywania operacji matematycznych na poszczególnych komórkach tabeli system PostgreSQL pozwala również obliczyć wynik w ramach jednej kolumny przy użyciu funkcji agregujących. Do najczęściej wykorzystywanych należą **COUNT()**, **MIN()**, **MAX(), AVG(), SUM().**

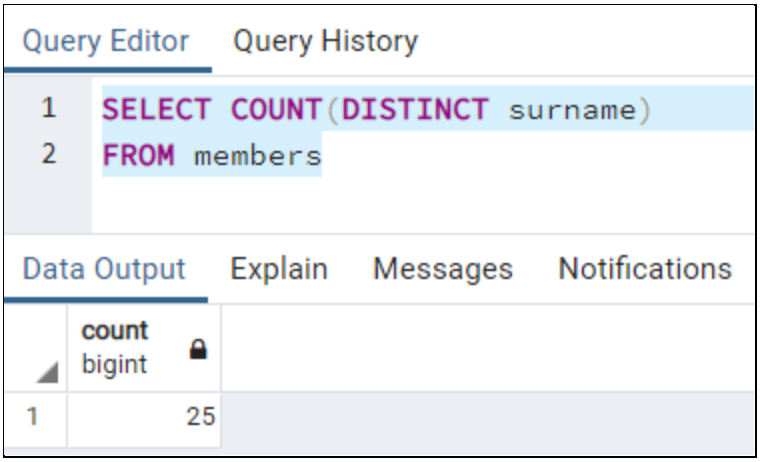
Funkcja **COUNT()** zlicza wartości występujące w kolumnie lub tabeli. Przykładowo, gdybyśmy chcieli poznać łączną liczbę członków klubu, moglibyśmy skorzystać z formuły:

SELECT COUNT(*)
FROM members



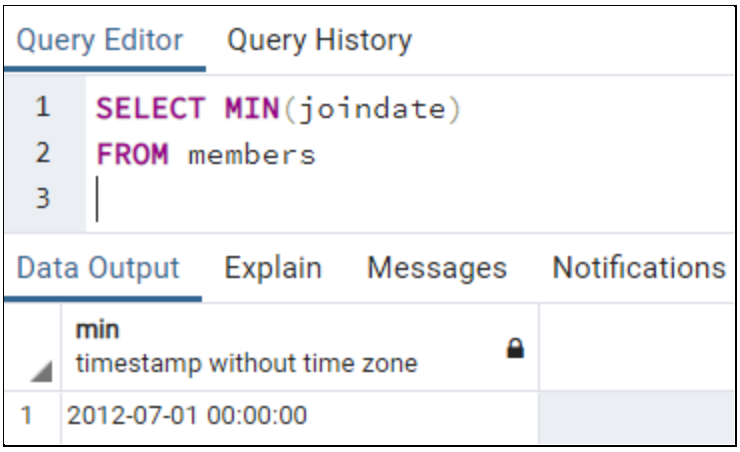
Nieco inaczej wyglądać będzie formuła na obliczenie liczby unikalnych nazwisk:

SELECT COUNT(DISTINCT surname**)**
FROM members



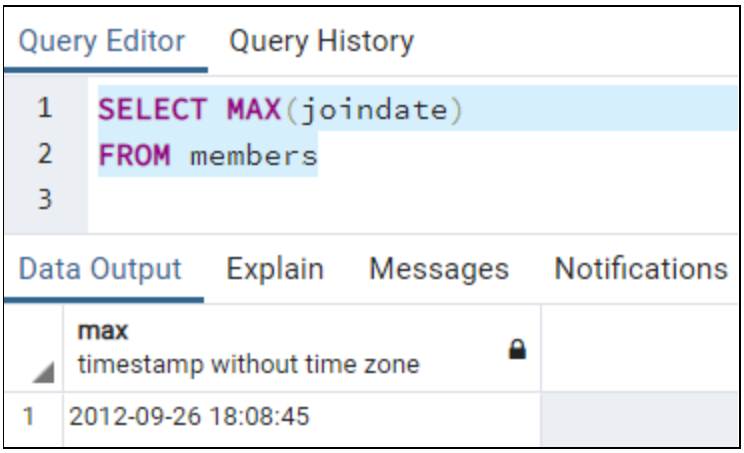
Funkcji **MIN()** i **MAX()** używamy do odczytania najmniejszej i największej wartości w kolumnie:

SELECT MIN(joindate)
FROM members



Analogicznie:

```
SELECT round(AVG(membercost),2) AS cost_member,
       round(AVG(guestcost),2) AS cost_guest
FROM facilities
```



Z kolei funkcja **AVG()** pozwala poznać średnią wartość w kolumnie, tj. sumę wszystkich wartości podzieloną przez liczbę wartości:

Query Editor | Query History

```
1  SELECT round(AVG(membercost),2) AS cost_member,
2         round(AVG(guestcost),2) AS cost_guest
3  FROM facilities
```

Data Output | Explain | Messages | Notifications

	cost_member numeric	cost_guest numeric
1	9.28	28.67

Powyższe zapytanie zwróciło średni koszt korzystania z obiektów sportowych dla członków i gości. Dodatkowo zastosowano funkcję **ROUND()**, zaokrąglającą liczbę wynikową do określonej (w tym przypadku do dwóch) miejsc po przecinku.

Łącząc funkcje agregujące z klauzulą **GROUP BY** możemy uzyskać jeszcze bardziej szczegółowe informacje, jak choćby łączny czas (mierzony w dniach i godzinach) korzystania z obiektów sportowych w danym miesiącu:

```
SELECT
foo.miesiąc,
foo.rok,
MAX(starttime) - MIN(starttime) AS czas
FROM
        (SELECT *,
        EXTRACT(YEAR FROM starttime::date) AS rok,
        EXTRACT(MONTH FROM starttime::date) AS miesiąc
        FROM bookings
        ORDER BY starttime)foo
GROUP BY miesiąc, rok
ORDER BY rok, miesiąc
```

Query Editor | Query History

```
1  SELECT
2  foo.miesiąc,
3  foo.rok,
4  MAX(starttime) - MIN(starttime) AS czas
5  FROM
6  (SELECT *,
7  EXTRACT(YEAR FROM starttime::date) AS rok,
8  EXTRACT(MONTH FROM starttime::date) AS miesiąc
9  FROM bookings
10 ORDER BY starttime)foo
```

Data Output | Explain | Messages | Notifications

	miesiąc double precision	rok double precision	czas interval
1	7	2012	28 days 11:...
2	8	2012	30 days 12:...
3	9	2012	29 days 11:...
4	1	2013	00:00:00

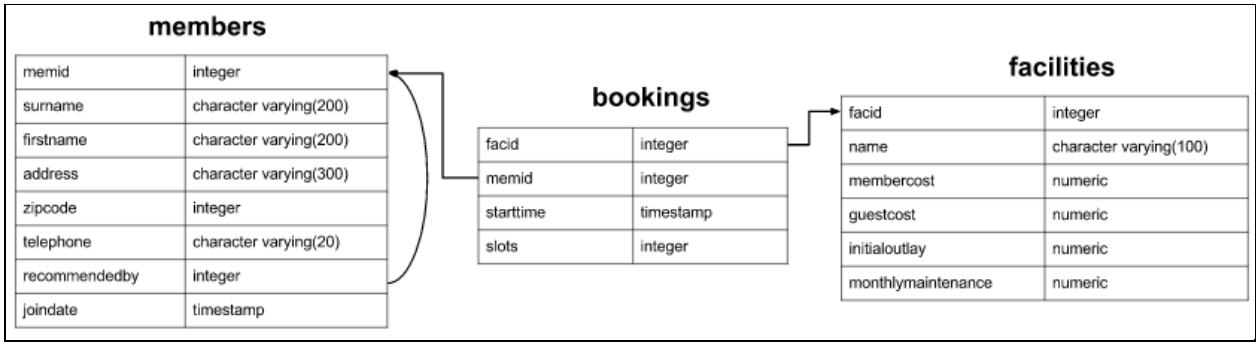
Struktura tego wyrażenia na pierwszy rzut oka wygląda na skomplikowaną. Warto zauważyć, że nazwy kolumn odnoszą się nie do istniejącej tabeli, lecz odrębnej instrukcji przetwarzającej dane zawarte w tabeli bookings; stąd “dziwne” nazwy atrybutów, takie jak czas, rok czy miesiąc. Funkcja **EXTRACT()** stanowi systemowy odpowiednik funkcji **DATE_PART()**, należącej do standardu ANSI SQL. Obie wykorzystuje się do wyodrębniania komponentów z wartości **timestamp**, dlatego też można je stosować wymiennie. Działanie **MAX(**starttime**) - MIN(**starttime**)** zwraca czas korzystania z obiektów sportowych w danym dniu.

Alias “foo” to termin często stosowany w dokumentacji kodu jako zamiennik faktycznych nazw funkcji czy zmiennych w sytuacji, gdy charakter tych nazw nie ma istotnego znaczenia. W praktyce mogą Państwo użyć jakiegokolwiek innego aliasu, którego obecność - przypomnijmy - jest konieczny, gdy po klauzuli **FROM** w miejscu warstwy pojawia się instrukcja SQL.

W tym konkretnym przypadku klauzula **GROUP BY** została wykorzystana do zsumowania czasu dla każdego miesiąca z osobna.

Złączenia i podzapytania

Relacyjność bazy danych polega na możliwości obsługi danych przechowywanych w wielu powiązanych tabelach. Model relacyjny charakteryzuje się tym, że każda tabela zawiera zazwyczaj dane dotyczące **jednej encji** (klasy obiektów). Encją może być np. grupa pracowników, flota samochodowa, rejestr obszarów objętych ustaloną formą ochrony przyrody, etc. Każdy wiersz w tabeli stanowi natomiast **instancję encji**, a więc pojedynczy obiekt (danej klasy). Wiersze z różnych tabel można łączyć, uzyskując tym samym nowe tabele wynikowe. Do wykonania złączenia niezbędna jest obecność kolumn zawierających te same wartości w obu łączonych tabelach. Omówmy tę operację na konkretnym przykładzie: chcemy uzyskać dane o wszystkich rezerwacjach obiektów sportowych, poczynionych na konkretną osobę. Tabela wynikowa powinna zawierać jedną kolumnę, przechowującą dane typu **timestamp**. Zanim przejdziemy do obmyślania struktury zapytania, rzućmy okiem na poniższy schemat:



Grafika przedstawia schemat połączeń między poszczególnymi tabelami. Widzimy, że pole *memid*, pełniące funkcję klucza głównego w tabeli “members”, jest jednocześnie kluczem obcym w tabeli “bookings”. Obie kolumny zawierają te same wartości, przez co mogą zostać wykorzystane do utworzenia złączenia z poziomu zapytania, którego celem jest, jak pamiętamy, wygenerowanie listy wszystkich rezerwacji na konkretną osobę - przyjmijmy, iż będzie to niejaki *David Farrell*.

Żeby nie było jednak zbyt łatwo, imię i nazwisko przechowywane są w dwóch odrębnych kolumnach. W zapytaniu musimy zatem odwołać się do atrybutów *firstname* i *surname* z tabeli “members” oraz *starttime* z tabeli “bookings”. Początek instrukcji wydaje się zatem stosunkowo prosty do przygotowania. Mógłby on wyglądać mniej więcej tak:

SELECT firstname, lastname, starttime

Nie jest to jednak poprawny zapis, gdyż nie uwzględnia faktu, że kolumny należą do dwóch różnych tabel. Informację tę należy więc zawrzeć w kwerendzie:

```
SELECT members.firstname, members.lastname,
bookings.starttime
FROM members, bookings
```

Aby uprościć zapis i nie przywoływać za każdym razem nazwy tabeli, można posłużyć się aliasami:

```
SELECT mb.firstname, mb.lastname, bk.starttime
FROM members mb, bookings bk
```

Przechodzimy do zdefiniowania złączenia. Można to zrobić na co najmniej dwa sposoby. Pierwszy zakłada wykorzystanie klauzuli **WHERE**, z którą spotkaliśmy się w jednym z wcześniejszych ćwiczeń.

```
WHERE mb.memid = bk.memid
```

Drugi natomiast wprowadza słowo kluczowe **JOIN**. W tym przypadku konstrukcja składni prezentuje się następująco:

```
FROM members mb JOIN bookings bk
    ON mb.memid = bk.memid
```

Pamiętajmy również, że rezerwacje dotyczą konkretnej osoby. Niezbędne jest więc dodanie drugiego warunku:

```
WHERE mb.firstname = 'David' AND mb.lastname = 'Farrell'
```

Lub jeśli trzymamy się pierwszego wariantu:

```
WHERE mb.memid = bk.memid
  AND mb.firstname = 'David'
  AND mb.lastname = 'Farrell'
```

Sprawdźmy, jaki wynik zwróci pełna instrukcja (tutaj korzystam z wariantu pierwszego):

```
SELECT mb.firstname, mb.lastname, bk.starttime
FROM members mb, bookings bk
WHERE mb.memid = bk.memid
AND mb.firstname = 'David'
AND mb.lastname = 'Farrell'
```

Query Editor | Query History

```
1  SELECT mb.firstname, mb.surname, bk.starttime
2  FROM members mb, bookings bk
3  WHERE mb.memid = bk.memid
4  AND mb.firstname = 'David'
5  AND mb.surname = 'Farrell'
```

Data Output | Explain | Messages | Notifications

	firstname character varying (200)	surname character varying (200)	starttime timestamp without time zone
1	David	Farrell	2012-09-18 09:00:00
2	David	Farrell	2012-09-18 13:30:00
3	David	Farrell	2012-09-18 17:30:00
4	David	Farrell	2012-09-18 20:00:00
5	David	Farrell	2012-09-19 09:30:00
6	David	Farrell	2012-09-19 12:00:00
7	David	Farrell	2012-09-19 15:00:00

Spróbujmy wykonać kolejne ćwiczenie o nieco większym stopniu złożoności. Tym razem chodzi o przygotowanie listy rezerwacji obu kortów tenisowych z 21 września 2012 roku. Wiersze w tabeli wynikowej powinny być uporządkowane według czasu rezerwacji. Do wykonania zadania będziemy potrzebowali kolumn z tabel bookings i facilities.

```
SELECT fc.name, bk.starttime
FROM facilities fc, bookings bk
        WHERE
        fc.facid = bk.facid
                AND fc.name IN ('Tennis Court 1', 'Tennis
Court 2')
                AND bk.starttime::date = '2012-09-21'
ORDER BY bk.starttime
```

Query Editor Query History

```
1  SELECT fc.name, bk.starttime
2  FROM facilities fc, bookings bk
3      WHERE
4      fc.facid = bk.facid
5          AND fc.name IN ('Tennis Court 1', 'Tennis Court 2')
6          AND bk.starttime::date = '2012-09-21'
7  ORDER BY bk.starttime
```

Data Output Explain Messages Notifications

	name character varying (100)	starttime timestamp without time zone
1	Tennis Court 1	2012-09-21 08:00:00
2	Tennis Court 2	2012-09-21 08:00:00
3	Tennis Court 1	2012-09-21 09:30:00
4	Tennis Court 2	2012-09-21 10:00:00
5	Tennis Court 2	2012-09-21 11:30:00
6	Tennis Court 1	2012-09-21 12:00:00

W wyniku wykonania instrukcji powinniśmy otrzymać 12 rekordów. Po raz kolejny dokonaliśmy złączenia typu **INNER JOIN**, które zwraca wyłącznie pasujące pary rekordów. Co jednak, jeśli zależy nam na zbiorze, w którym niektóre rekordy z prawej bądź lewej kolumny intencjonalnie nie posiadają dopasowania?

W takiej sytuacji możemy skorzystać z dwóch kolejnych typów złączenia, mianowicie **LEFT** lub **OUTER RIGHT JOIN**.

Przetestujmy działanie omawianego typu złączeń na kolejnym przykładzie: mamy do wygenerowania tabelę z imieniem i nazwiskiem wszystkich członków *Country Clubu*, wraz z imieniem i nazwiskiem osoby rekomendującej, umieszczonymi w jednej kolumnie. W tym przypadku kluczowe znaczenie ma informacja, że kolumny *memid* i *recommendedby* z tabeli “members” zawierają te same wartości.

```
SELECT mb.firstname,mb.surname,
rc.firstname||' '||rc.surname AS fullname

FROM members mb
LEFT OUTER JOIN members rc
        ON rc.memid = mb.recommendedby
ORDER BY mb.surname, mb.firstname
```

Query Editor | Query History

```
1  SELECT mb.firstname,mb.surname,
2  rc.firstname||' '||rc.surname AS fullname
3
4  FROM members mb
5  LEFT OUTER JOIN members rc
6  ON rc.memid = mb.recommendedby
7  ORDER BY mb.surname, mb.firstname
8
9
```

Data Output | Explain | Messages | Notifications

	firstname character varying (200)	surname character varying (200)	fullname text
1	Florence	Bader	Ponder Stibbons
2	Anne	Baker	Ponder Stibbons
3	Timothy	Baker	Jemima Farrell
4	Tim	Boothe	Tim Rownam
5	Gerald	Butters	Darren Smith

Prawdopodobnie Państwa uwagę przykuła obecność wartości NULL w kolumnie wynikowej fullname:

Data Output | Explain | Messages | Notifications

	firstname character varying (200)	surname character varying (200)	fullname text
1	David	Farrell	[null]
2	Jemima	Farrell	[null]
3	Max	Iksowski	[null]
4	Tim	Rownam	[null]
5	Darren	Smith	[null]

oraz niestosowany wcześniej zapis:

rc.firstname||' '||rc.surname **AS** fullname

Odnośnie pierwszego wątku - obecność wartości NULL jest pokłosiem zastosowania złączenia typu **LEFT OUTER**. Chodziło nam bowiem o przygotowanie listy wszystkich członków, a nie tylko tych, którzy nabyli prawa członkowskie na drodze polecenia. Z tego właśnie powodu w tym konkretnym przypadku nie moglibyśmy skorzystać ze słowa kluczowego **JOIN** (odpowiadającemu **INNER JOIN**), gdyż zawierające je zapytanie zwróciłoby wyłącznie pasujące do siebie pary rekordów.

Natomiast zastosowany w zapytaniu znak **||** służy, podobnie jak w *Kreatorze wyrażeń* programu QGIS, do łączenia ciągu znaków. Z kolei symbol ‘ ‘ to nic innego jak znak spacji wzięty w apostrofy, wprowadzony w celu rozdzielenia imienia i nazwiska.

Istnieją również inne typy złączeń, jak choćby **FULL OUTER JOIN**, zwracający wszystkie rekordy z obu łączonych tabel oraz **CROSS JOIN**, działający na zasadzie rachunku kartezjańskiego (zwraca każdą możliwą kombinację wierszy z obu tabel). Stosuje się je jednak znacznie rzadziej, w związku z czym ograniczę się jedynie do wzmianki o nich.

Do omówienia pozostaje jeszcze jedna, istotna funkcja, mianowicie wyszukiwanie wierszy zawierających wartości NULL. Nie jest to szczególnie trudne działanie, sprowadza się bowiem do dodania warunku wedle wzoru:

WHERE wartość **IS NULL**

Dodając tę frazę do naszej poprzedniej instrukcji, możemy łatwo wydrukować listę członków, których przyjęto do klubu mimo braku polecenia:

```
SELECT mb.firstname,mb.surname,
rc.firstname||' '||rc.surname AS fullname

FROM members mb
LEFT OUTER JOIN members rc
ON rc.memid = mb.recommendedby
WHERE rc.firstname||' '||rc.surname IS NULL
ORDER BY mb.surname, mb.firstname
```

Query Editor | Query History

```
1  SELECT mb.firstname,mb.surname,
2  rc.firstname||' '||rc.surname AS fullname
3
4  FROM members mb
5  LEFT OUTER JOIN members rc
6  ON rc.memid = mb.recommendedby
7  WHERE rc.firstname||' '||rc.surname IS NULL
8  ORDER BY mb.surname, mb.firstname
9
```

Data Output | Explain | Messages | Notifications

	firstname character varying (200)	surname character varying (200)	fullname text
1	David	Farrell	[null]
2	Jemima	Farrell	[null]
3	Max	Iksowski	[null]
4	Tim	Rownam	[null]
5	Darren	Smith	[null]

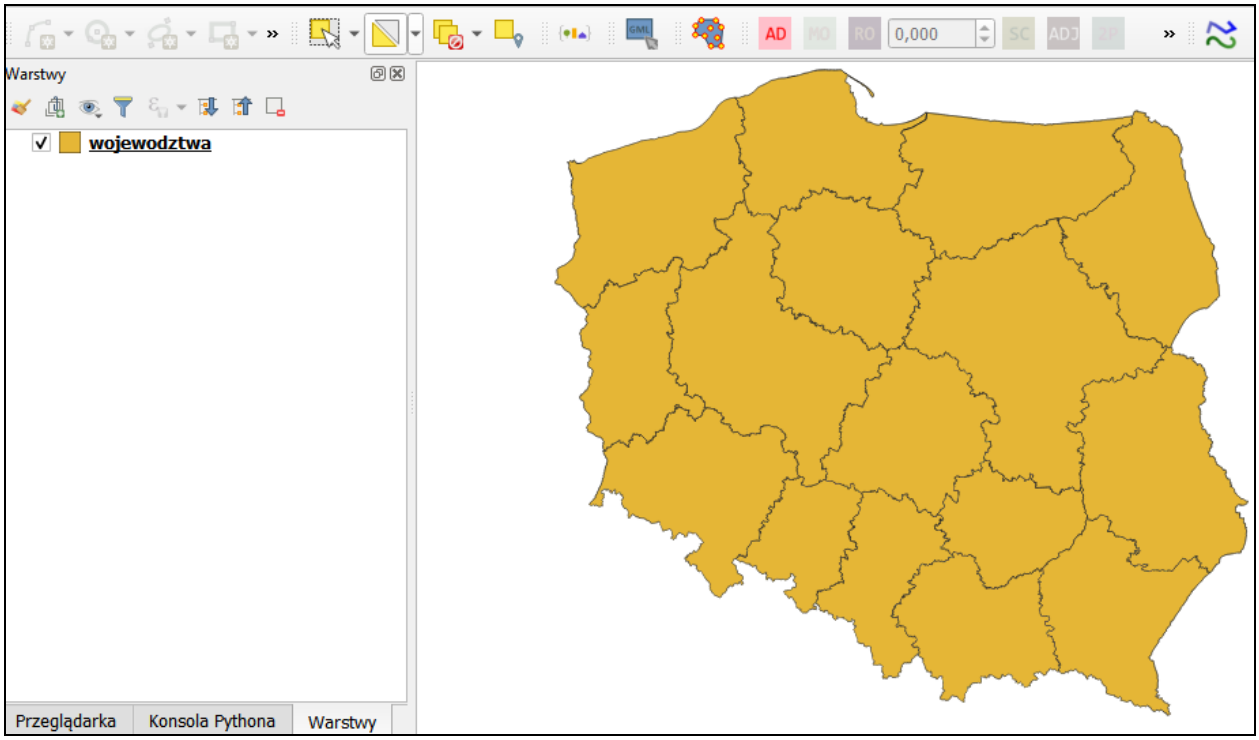
W przypadku małych zbiorów danych, liczących kilka lub kilkanaście wierszy, rekordy z brakującymi wartościami lokalizuje się stosunkowo łatwo. Problem pojawia się w momencie, gdy nasza tabela liczy dziesiątki tysięcy rekordów, a my stoimy przed koniecznością obmyślenia szybkiej i efektywnej strategii kontroli poprawności danych. W analogicznych przypadkach zastosowanie warunku **IS NULL** odgrywa kluczową rolę.

Wprowadzenie do PostGIS - omówienie wybranych metod importu danych przestrzennych do bazy danych

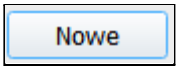
Import danych przestrzennych do PostgreSQL

Najprostsza metoda zasilenia bazy danymi przestrzennymi sprowadza się do wykorzystania programu QGIS.

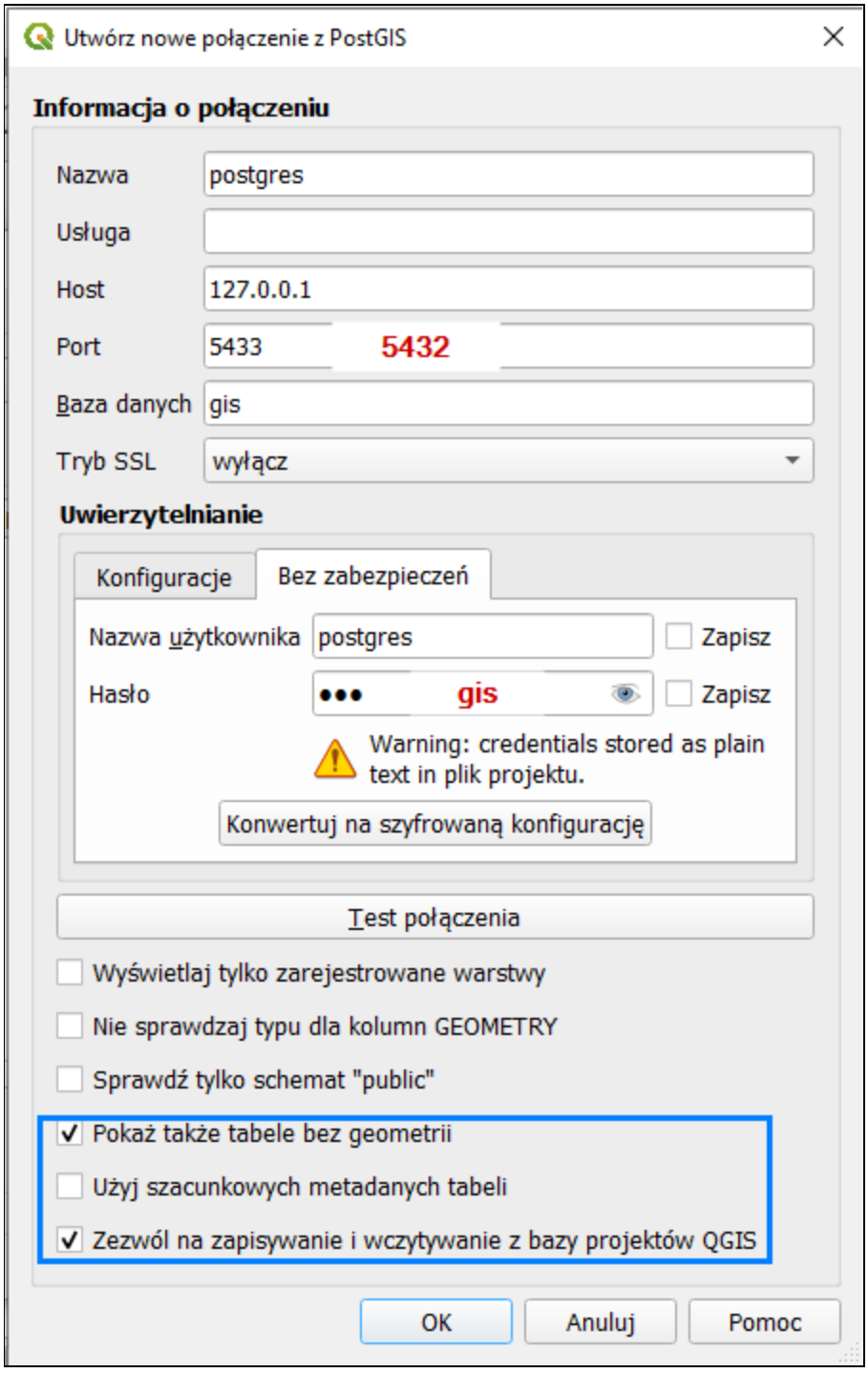
Uruchamiamy program, tworzymy nowy projekt w układzie współrzędnych EPSG:2180 i dodajemy do niego warstwę wektorową *wojewodztwa.shp* (znajdą ją Państwo w katalogu z danymi).



Następnie przechodzimy do *Okna zarządzania źródłami danych*  i z listy zakładek po lewej stronie okna wybieramy 

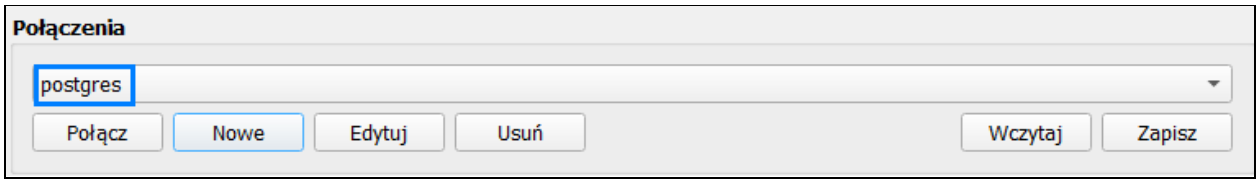
. Stworzymy teraz połączenie z wcześniej utworzoną bazą danych. Klikamy na 

, a następnie wypełniamy okna według wzoru:

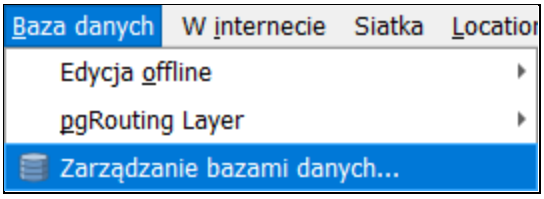


Zaznaczając dwie wyszczególnione opcje uzyskujemy wgląd w tabele pozbawione geometrii oraz możliwość zapisywania i wczytywania naszych projektów QGIS bezpośrednio w bazie danych. Opcja pierwsza przydaje się w sytuacji, gdy oprócz danych przestrzennych planujemy przechowywać również tabele bez geometrii, np. arkusze kalkulacyjne. Druga natomiast znacząco ułatwia zarządzanie projektami.

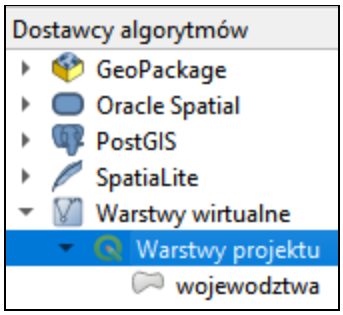
Połączenie z bazą powinno być już widoczne na rozwijanej liście w górnej części okna:



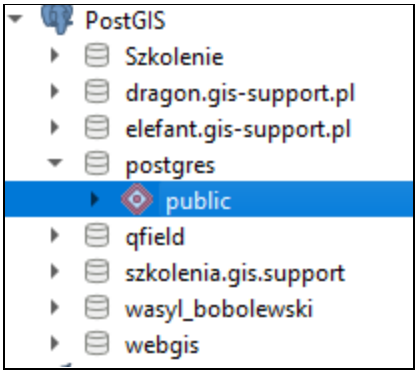
Otwieramy narzędzie *Zarządzanie bazami danych* z zakładki *Bazy danych*, znajdującej się na pasku menu QGIS:



W panelu *Dostawców Algorytmów* rozwijamy kolejno zakładki *Warstwy wirtualne* oraz *Warstwy projektu*. Ta ostatnia zawiera listę warstw z aktualnie edytowanego projektu:

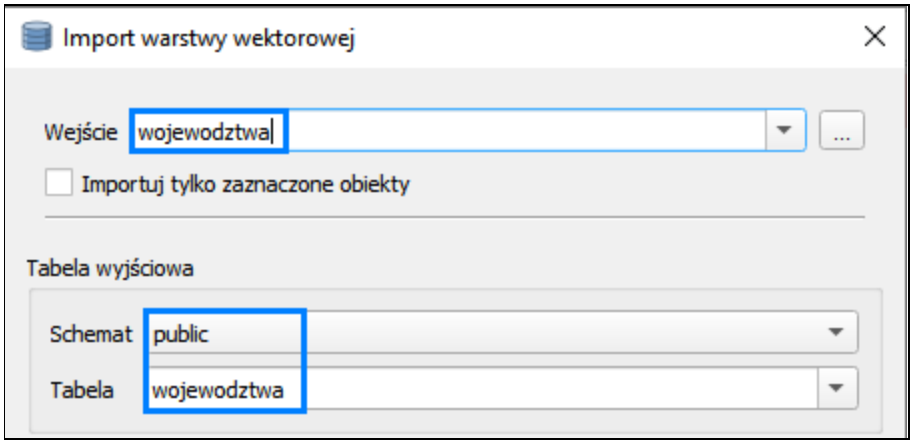


W ten sam sposób rozwijamy zakładkę *PostGIS* oraz drzewo bazy danych, której zasób chcemy zasilić. Pamiętamy również o wskazaniu docelowego schematu:



Klikamy teraz na *Importuj warstwę/plik*. Opcja ta zlokalizowana jest w górnej części okna narzędzia *Zarządzanie bazami danych*:

W polu *wejście* podajemy nazwę warstwy przeznaczonej do importu. Dodatkowo w ustawieniach *Tabeli wyjściowej* należy ostatecznie zdefiniować docelowy schemat bazy i podać nazwę tabeli wynikowej (domyślnie jest ona tożsama z nazwą warstwy wejściowej):



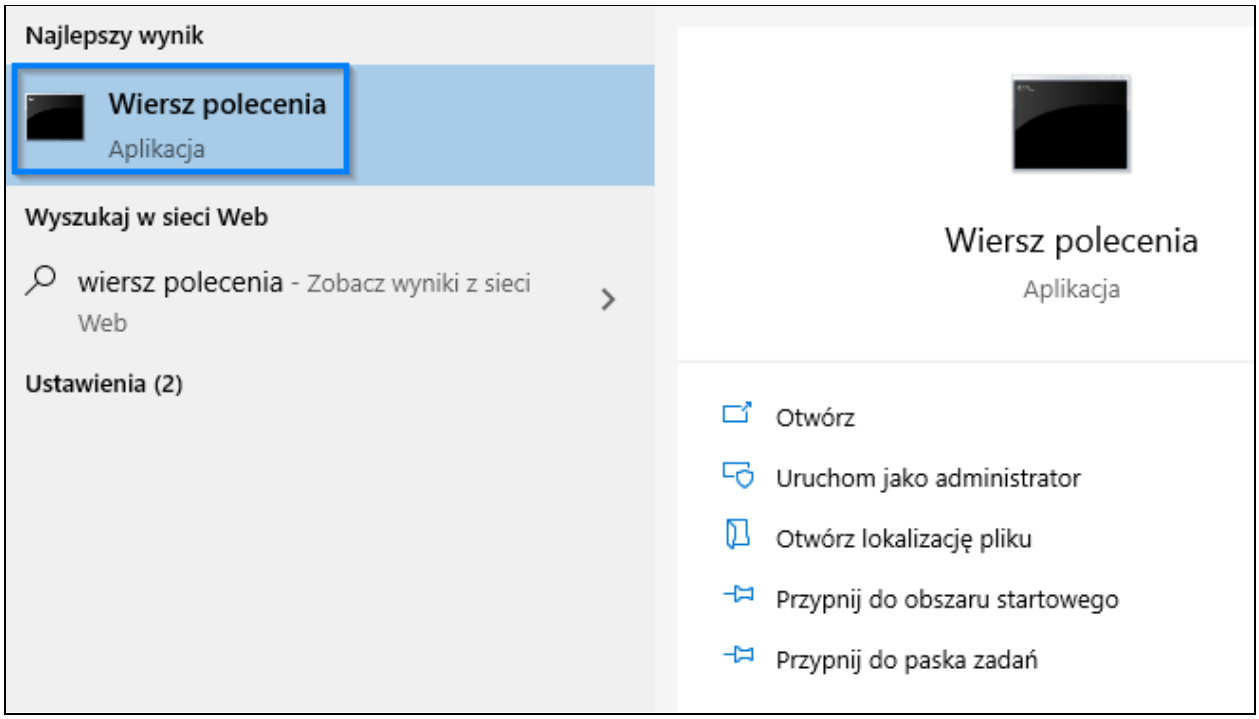
Operację zatwierdzamy, klikając na OK.

Efektywność tego rozwiązania (zwłaszcza w przypadku większych zbiorów) pozostawia jednak wiele do życzenia, zwłaszcza w kwestii czasu oczekiwania na zakończenie procesu importu. Alternatywą jest wykorzystanie narzędzi obsługiwanych z poziomu wiersza poleceń systemu Windows (shp2pgsql) lub powłoki OSGeo4W (ogr2ogr).

*UWAGA! WSZYSTKIE DANE DO POSZCZEGÓLNYCH ĆWICZEŃ MOŻNA POBRAĆ **STĄD** (podobnie jak poprzednio, znajdą Państwo ten zestaw w katalogu z materiałami do ćwiczeń).*

Narzędzie shp2pgsql

Aby uruchomić narzędzie **shp2pqsql**, należy zacząć od otwarcia wiersza poleceń. Najprostszą metodą jest otwarcie menu *Start* i wpisanie frazy “wiersz polecenia”:



Do uruchomienia niezbędna jest znajomość ścieżki do aplikacji. Domyślnie znajduje się ona w katalogu “C:\Program Files\PostgreSQL\13\bin”. W wierszu polecenia należy, wobec tego wpisać “C:\Program Files\PostgreSQL\13\bin\shp2pgsql.exe”:

```
C:\Users\Lenovo>C:\Program Files\PostgreSQL\13\bin\shp2pgsql.exe
```

Co jednak, jeśli po wciśnięciu klawisza *Enter* w oknie pojawia się komunikat następującej treści?:

```
C:\Users\Lenovo>C:\Program Files\PostgreSQL\13\bin\shp2pgsql.exe
'C:\Program' is not recognized as an internal or external command,
operable program or batch file.
```

Najprawdopodobniej oznacza to, że zapomnieli Państwo skopiować wyrażenie wraz z cudzysłowami. Brak cudzysłowu na początku i końcu wyrażenia powoduje, że jest ono traktowane jako dwa odrębne ciągi znaków. Problemem jest bowiem spacja w nazwie katalogu “Program Files”.

Po wpisaniu odpowiedniej frazy ponownie wciskamy *Enter*. Wyświetli się dokumentacja narzędzia wraz z opisem poszczególnych funkcji:

```
C:\Users\Lenovo>"C:\Program Files\PostgreSQL\13\bin\shp2pgsql.exe"
RELEASE: 3.1.2 (3.1.2)
USAGE: shp2pgsql [<options>] <shapefile> [[<schema>.]<table>]
OPTIONS:
  -s [<from>:]<srid> Set the SRID field. Defaults to 0.
      Optionally reprojects from given SRID.
 (-d|a|c|p) These are mutually exclusive options:
     -d  Drops the table, then recreates it and populates
         it with current shape file data.
     -a  Appends shape file into current table, must be
         exactly the same table schema.
     -c  Creates a new table and populates it, this is the
         default if you do not specify any options.
     -p  Prepare mode, only creates the table.
  -g <geocolumn> Specify the name of the geometry/geography column
      (mostly useful in append mode).
  -D  Use postgresql dump format (defaults to SQL insert statements).
  -e  Execute each statement individually, do not use a transaction.
      Not compatible with -D.
  -G  Use geography type (requires lon/lat data or -s to reproject).
  -k  Keep postgresql identifiers case.
  -i  Use int4 type for all integer dbf fields.
  -I  Create a spatial index on the geocolumn.
  -m <filename>  Specify a file containing a set of mappings of (long) column
     names to 10 character DBF column names. The content of the file is one or
     more lines of two names separated by white space and no trailing or
```

W tym ćwiczeniu ograniczymy się do najprostszego wariantu importu - do wcześniej utworzonej bazy *gis* wgramy plik .shp z granicami gmin. Do osiągnięcia celu posłuży nam polecenie:

"C:\Program Files\PostgreSQL\13\bin\shp2pgsql.exe" -I -s 2180 C:\gminy\gminy.shp public.gminy_test C:\gminy\gminy.sql

UWAGA! Zapis ścieżki ma charakter przykładowy. Dla Państwa wygody zaleca się utworzenie katalogu “C:\szkolenie_dane”, w którym przechowywane będą dane szkoleniowe oraz “C:\cwiczenia”, przeznaczonego na warstwy wynikowe.

Zapis komendy jest dość złożony, w związku z czym warto poświęcić chwilę na zapoznanie z się z poszczególnymi argumentami:

-D - użycie masowego importu danych w miejsce instrukcji INSERT
-I - tworzy indeks przestrzenny na kolumnie geometrii w nowej tabeli
-s - pozwala zdefiniować układ współrzędnych danych wynikowych

Po zakończeniu procesu przetwarzania (o ile wszystko zadziałało zgodnie z założeniami) naszym oczom powinien ukazać się następujący komunikat:

```
CREATE INDEX ON "public"."gminy_test" USING GIST ("geom");
COMMIT;
ANALYZE "public"."gminy_test";

C:\Users\Lenovo>
```

W folderze C:\gminy powinien pojawić się plik gminy.sql, będący wynikiem konwersji warstwy wejściowej w formacie .shp. Możemy teraz przejść do kolejnego kroku, polegającego na uruchomieniu wygenerowanej instrukcji sql przy użyciu aplikacji **psql**. Podobnie jak w przypadku **shp2pgsql** zaczynamy od wskazania ścieżki do aplikacji, następnie zaś podajemy argumenty:

C:\Users\Lenovo>"c:\Program Files\PostgreSQL\13\bin\psql.exe" -U postgres -d gis -f C:\gminy\gminy.sql

-U - argument określający nazwę użytkownika
-d - nazwa bazy danych
-f - wskazanie pliku będącego źródłem instrukcji do wykonania (przypomnijmy - jest to wynik konwersji danych wektorowych przy użyciu narzędzia **shp2pgsql**, wygenerowany w poprzednim ćwiczeniu).

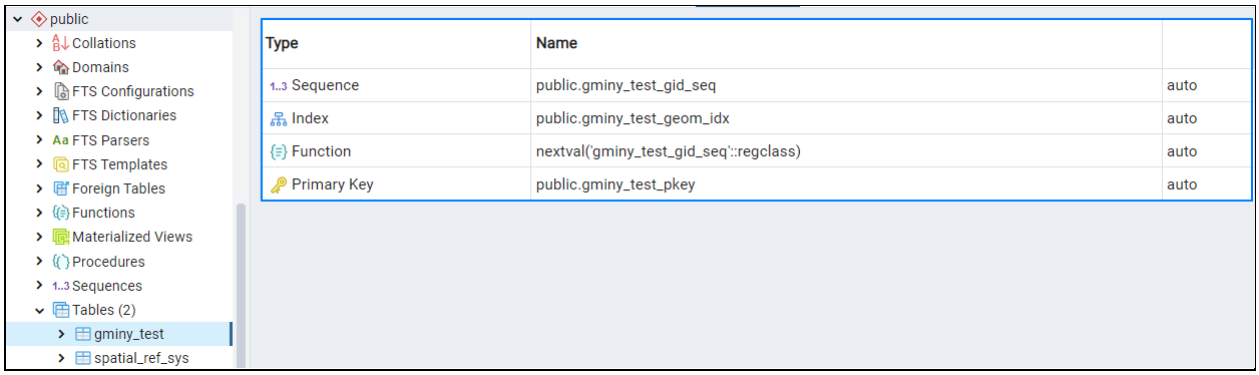
Uruchamiamy komendę, wciskając klawisz *Enter*. W trakcie wykonywania polecenia zostaniemy dodatkowo poproszeni o podanie hasła głównego dla użytkownika postgres (UWAGA! Podczas wpisywania hasła nie są wyświetlane żadne znaki!).

```
C:\Users\Lenovo>"c:\Program Files\PostgreSQL\12\bin\psql.exe" -U postgres -d gis -f C:\gminy\gminy.sql
Password for user postgres:
SET
SET
BEGIN
CREATE TABLE
ALTER TABLE
                    addgeometrycolumn
-------------------------------------------------------------
 public.gminy_test.geom SRID:2180 TYPE:MULTIPOLYGON DIMS:2
(1 row)


COPY 2477
CREATE INDEX
COMMIT
ANALYZE

C:\Users\Lenovo>
```

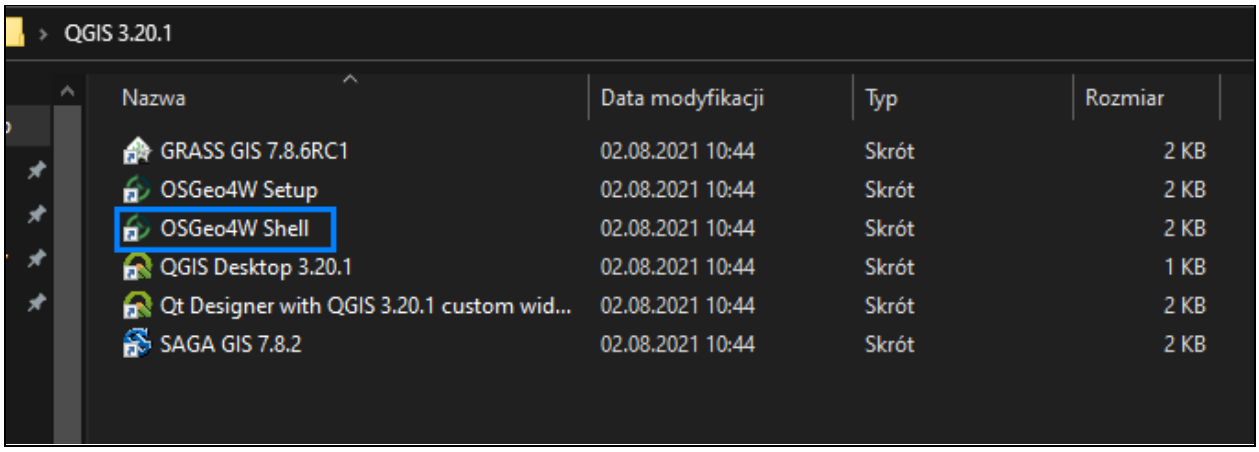
Możemy teraz przejść do aplikacji pgAdmin i sprawdzić, czy instrukcja została wykonana poprawnie. Jeśli tak, na liście *Tables* w bazie danych *gis* pojawi się pozycja o nazwie *gminy_test*:



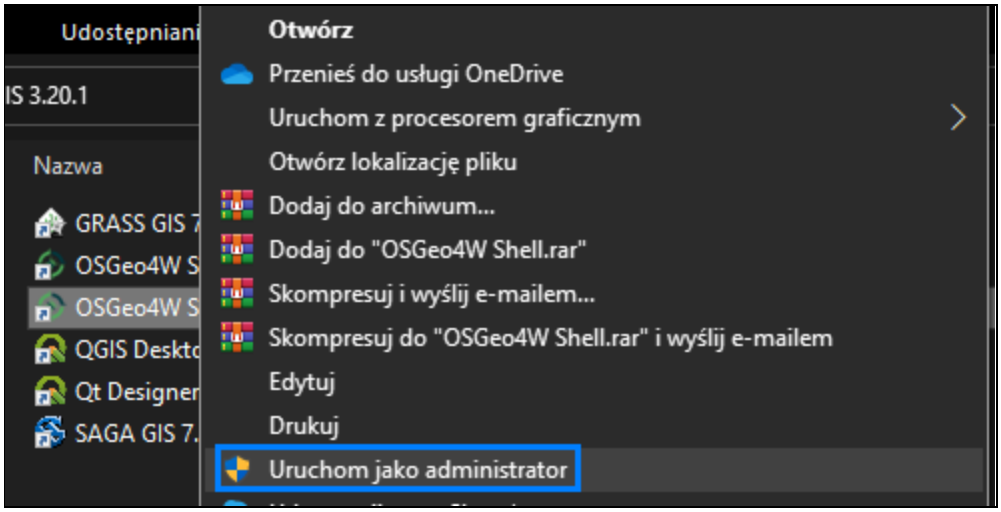
Narzędzie ogr2ogr

Ogr2ogr to potężne narzędzie do będące częścią biblioteki GDAL. Za jego pomocą można konwertować niemal 70 formatów danych wektorowych. W tym ćwiczeniu wykorzystamy je do zaimportowania do bazy danych *gis* pliku w formacie .gpkg, przechowującego geometrie cieków wodnych oraz obszarów zalesionych, znajdujących się w granicach województwa mazowieckiego.

Konwerter obsługujemy z poziomu powłoki *OSGeo4W*. Najprościej uruchomić ją klikając na stosowną ikonę w folderze instalacji programu *QGIS*:



Aby uniknąć uciążliwych komunikatów o błędach, możemy uruchomić aplikację w trybie administratora:



Sprawdźmy, czy wszystko działa jak powinno. W tym celu w oknie powłoki wpiszmy frazę:

ogr2ogr --info

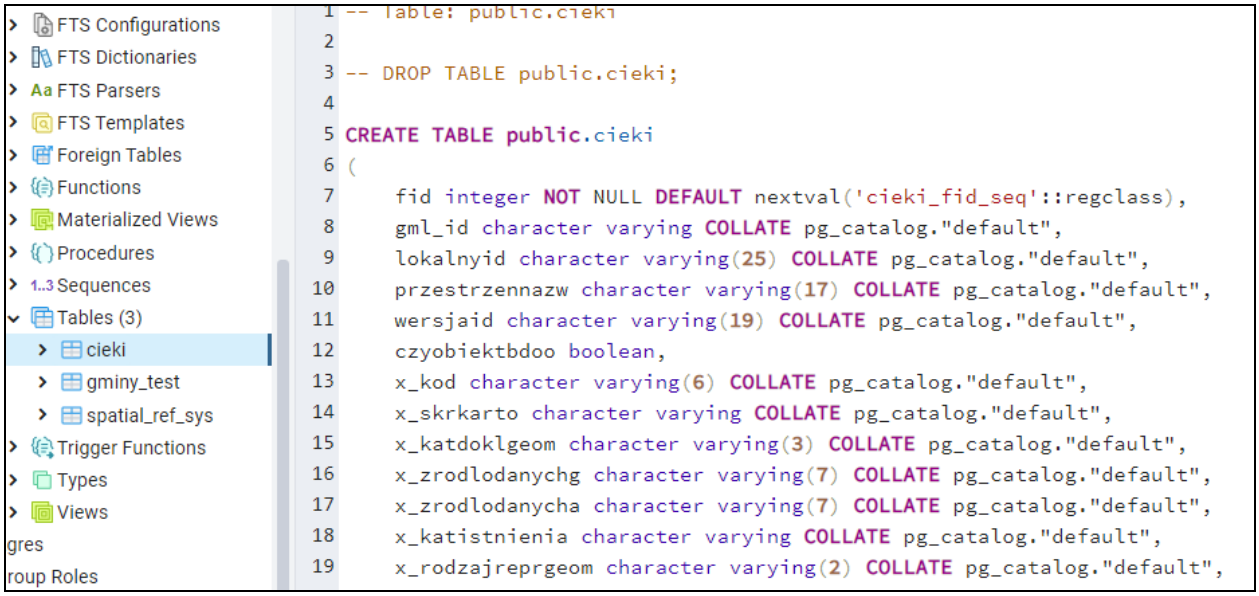
Program powinien wyświetlić listę parametrów dla wywołanego narzędzia:

```
Advanced options :
             [-gt n] [-ds_transaction]
             [[-oo NAME=VALUE] ...] [[-doo NAME=VALUE] ...]
             [-clipsrc [xmin ymin xmax ymax]|WKT|datasource|spat_extent]
             [-clipsrcsql sql_statement] [-clipsrclayer layer]
             [-clipsrcwhere expression]
             [-clipdst [xmin ymin xmax ymax]|WKT|datasource]
             [-clipdstsql sql_statement] [-clipdstlayer layer]
             [-clipdstwhere expression]
             [-wrapdateline][-datelineoffset val]
             [[-simplify tolerance] | [-segmentize max_dist]]
             [-makevalid]
             [-addfields] [-unsetFid] [-emptyStrAsNull]
             [-relaxedFieldNameMatch] [-forceNullable] [-unsetDefault]
             [-fieldTypeToString All|(type1[,type2]*)] [-unsetFieldWidth]
             [-mapFieldType srctype|All=dsttype[,srctype2=dsttype2]*]
             [-fieldmap identity | index1[,index2]*]
             [-splitlistfields] [-maxsubfields val]
             [-resolveDomains]
             [-explodecollections] [-zfield field_name]
             [-gcp ungeoref_x ungeoref_y georef_x georef_y [elevation]]* [-order n | -tps]
             [-nomd] [-mo "META-TAG=VALUE"]* [-noNativeData]
```

Do dyspozycji mamy geopaczkę z dwiema warstwami, zawierającymi geometrie cieków oraz obszarów zalesionych. Zacznijmy od dodania warstwy z ciekami. Pomoże nam w tym poniższa komenda:

ogr2ogr -f PostgreSQL PG:"host=127.0.0.1 port=5432 user=postgres dbname=gis password=gis" C:\gminy\geopaczka.gpkg cieki

Po wykonaniu importu tabela z danymi o ciekach powinna znaleźć się w schemacie *public* bazy danych *gis*:



W analogiczny sposób zaimportujmy również dane o lasach. Możemy w zasadzie użyć poprzedniej komendy; należy zmienić jedynie ostatni parametr odnoszący się do nazwy warstwy:

ogr2ogr -f PostgreSQL PG:"host=127.0.0.1 port=5432 user=postgres dbname=gis password=gis" C:\gminy\geopaczka.gpkg lasy

Dodatkowo możliwe jest zaimportowanie do bazy danych wszystkich plików o tym samym rozszerzeniu, przechowywanych w jednym katalogu. W tym celu należy posłużyć się instrukcją FOR stworzoną według składni wiersza poleceń systemu *Windows*:

>FOR %i IN (F:\GML*.gml) DO ogr2ogr -f PostgreSQL -lco GEOMETRY_NAME=geom -lco FID=id PG:"host=127.0.0.1 port=5433 user=postgres dbname=gis password=gis" "%i"

Słowo kluczowe FOR otwiera pętlę; wszystkie pliki z rozszerzeniem .gml, przechowywane w określonym katalogu, zostaną zaimportowane do zdefiniowanej w instrukcji bazy danych PostgreSQL:

```
C:\Program Files\QGIS 3.20.1>ogr2ogr -f PostgreSQL PG:"host=127.0.0.1 port=5433 user=postgres dbname=gis password=gi
s" "F:\GML\miejscowosci.gml"

C:\Program Files\QGIS 3.20.1>ogr2ogr -f PostgreSQL PG:"host=127.0.0.1 port=5433 user=postgres dbname=gis password=gi
s" "F:\GML\parki_kraj.gml"

C:\Program Files\QGIS 3.20.1>ogr2ogr -f PostgreSQL PG:"host=127.0.0.1 port=5433 user=postgres dbname=gis password=gi
s" "F:\GML\rezerwaty.gml"
```

W niektórych przypadkach może być konieczne skorzystanie z dodatkowego parametru **-nlt** (new layer type). Pozwala on zdefiniować typ geometrii warstwy wynikowej. Funkcja ta jest szczególnie istotna w imporcie warstw poligonowych. Brak parametru **-nlt PROMOTE_TO_MULTI** może bowiem zakończyć się komunikatem o błędzie. Przetestujmy to działanie na przykładzie warstwy z granicami powiatów. Na początek instrukcja bez parametru **-nlt**:

```
C:\Program Files\QGIS 3.20.1> ogr2ogr -f PostgreSQL -lco GEOMETRY_NAME=geom -lco FID=id PG:"host=127.0.0.1 port=5433
 user=postgres dbname=gis password=gis" F:\powiaty_shp\powiaty.shp powiaty
Warning 1: Geometry to be inserted is of type Multi Polygon, whereas the layer geometry type is Polygon.
Insertion is likely to fail
ERROR 1: COPY statement failed.
ERROR:  Geometry type (MultiPolygon) does not match column type (Polygon)
CONTEXT:  COPY powiaty, line 48, column geom: "0106000020320BF0D0003000000010300000001000000B7000000F2916401B9AB1E41A
AEE454BC49E204146B0F0F80EAC1E41..."
```

Komunikat o błędzie znika, gdy do instrukcji dodamy wymieniony wyżej parametr:

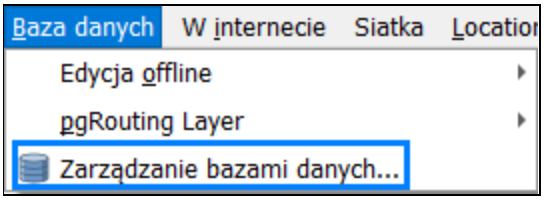
ogr2ogr -f PostgreSQL -lco GEOMETRY_NAME=geom -lco FID=id -nlt PROMOTE_TO_MULTI PG:"host=127.0.0.1 port=5433 user=postgres dbname=gis password=gis" F:\powiaty_shp\powiaty.shp powiaty

DB Manager - zapytania przestrzenne do bazy danych w QGIS

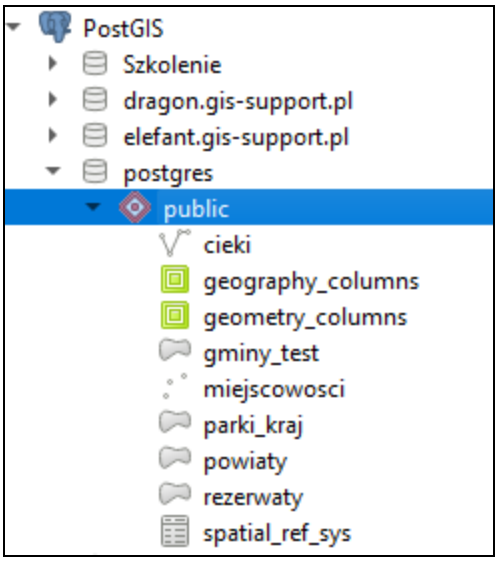
Do tej pory analizowaliśmy zbiory danych nieprzestrzennych. Nadszedł więc czas, by przyjrzeć się możliwościom wykorzystania instrukcji SQL w zarządzaniu danymi mającymi odniesienie do obiektów istniejących (lub wytyczonych) w przestrzeni.

Działania na danych możemy wykonywać na kilka sposobów. W tym przypadku pominiemy wiersz polecenia oraz powłokę *OSGeo4W*, koncentrując się przede wszystkim na natywnej wtyczce QGIS o nazwie DB Manager. Alternatywnie moglibyśmy skorzystać z pgAdmin, jednak DB Manager daje nam dużo większe możliwości w zakresie wizualizacji danych wynikowych.

Możemy teraz przejść do zakładki *Bazy danych* i wybrać *Zarządzanie bazami danych:*



Po lewej stronie okna narzędzia wyświetla się lista dostępnych opcji. Rozwijamy kategorię *Postgis* i przechodzimy do naszej bazy; zawiera ona tylko jeden podstawowy schemat oraz zbiór dodanych wcześniej tabel:



Spróbujmy wykonać proste zapytanie - wyświetlamy wszystkie obiekty z warstwy *powiaty* i uporządkujmy je w kolejności alfabetycznej. Zaznaczamy warstwę na liście i otwieramy *okno SQL*, ukrywające się pod ikonką  .

Zasada pisania instrukcji jest dokładnie taka sama jak w aplikacji pgAdmin:

```
1 SELECT * FROM powiaty
2 ORDER BY jpt_nazwa_
```

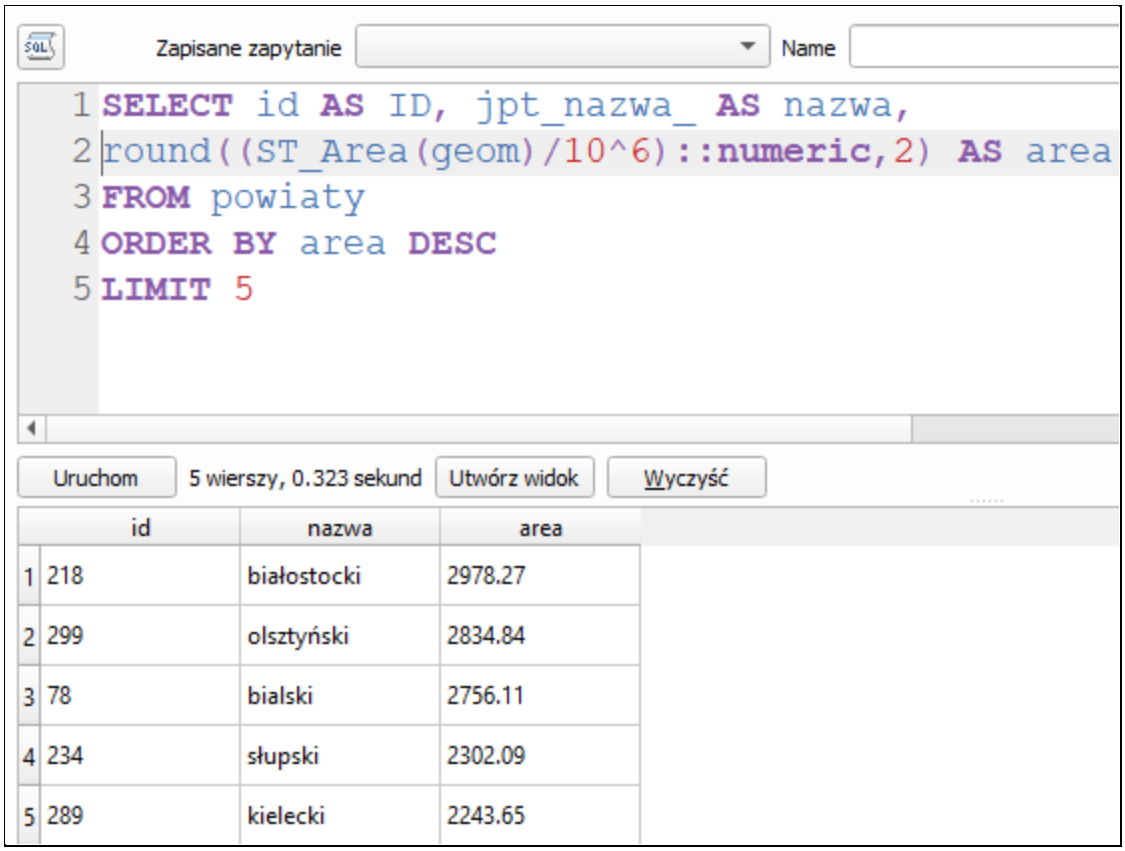
Uruchom | 380 wierszy, 2.039 sekund | Utwórz widok | Wyczyść

	id	fid	iip_przest	iip_identy	jpt_sjr_ko
1	42	42.0	PL.PZGIK.200	20bd1fbc-6150-...	POW
2	217	217.0	PL.PZGIK.200	46c53311-3b2d...	POW
3	308	308.0	PL.PZGIK.200	171c7527-7107-...	POW
4	64	64.0	PL.PZGIK.200	bd275e62-54ee...	POW
5	265	265.0	PL.PZGIK.200	3078fc9c-2cea-...	POW

Aby uruchomić instrukcję klikamy (nomen omen) *Uruchom*. Nie będziemy na razie wyświetlać wizualizacji wyniku, gdyż zawiera on dokładnie te same obiekty, co warstwa źródłowa. Możemy jednak wprowadzić do naszego wyrażenia funkcję obliczającą powierzchnię i pogrupować obiekty według liczby km2. Aby tego dokonać, będziemy musieli przeliczyć jednostki z bazowych metrów kwadratowych. Ponadto tabela wynikowa nie musi zawierać wszystkich kolumn i obiektów - ograniczmy się do atrybutu z numerem porządkowym, nazwą oraz nowoutworzoną kolumną o nazwie AREA, w której zapiszemy wynik działania na obliczenie powierzchni. Ograniczmy się ponadto do pięciu największych powiatów:

```
SELECT id AS id, jpt_nazwa_ AS nazwa,

round((ST_Area(geom)/10^6)::numeric,2) AS area
FROM powiaty
ORDER BY area DESC
LIMIT 5
```



	id	nazwa	area
1	218	białostocki	2978.27
2	299	olsztyński	2834.84
3	78	bialski	2756.11
4	234	słupski	2302.09
5	289	kielecki	2243.65

W przytoczonej instrukcji sporo się dzieje, rozbijmy więc na mniejsze części w celu omówienia ich działania.

SELECT id **AS** id, jpt_nazwa_ **AS** nazwa,

Używamy instrukcji **SELECT** i aliasów do wyboru i przemianowania kolumn wynikowych;

round((ST_Area(geom)/10^6)::numeric,2) **AS** area

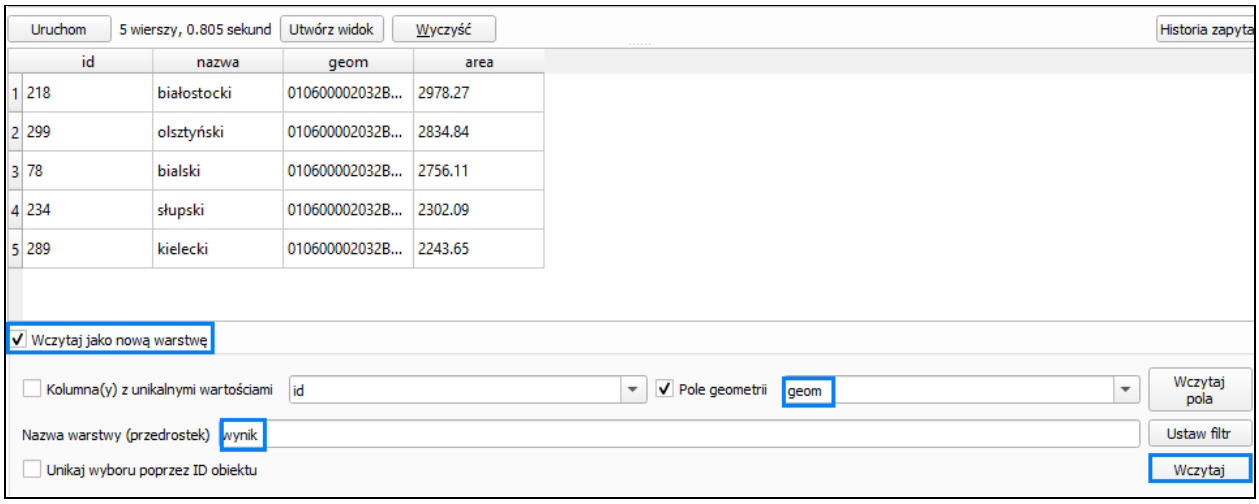
Funkcja ST_Area oblicza pole powierzchni obiektu, zwracając wynik w jednostkach układu warstwy (a więc w metrach). Aby przeliczyć metry kwadratowe na kilometry kwadratowe, musimy podzielić rezultat funkcji przez 10^6, czyli 1 000 000. Wynik chcielibyśmy dodatkowo zaokrąglić do dwóch miejsc po przecinku przy użyciu funkcji **round()**. Tutaj jednak pojawia się problem. Rzecz w tym, że funkcja **round()** jako pierwszy argument przyjmuje typ *numeric***.** Wynik obliczeń powierzchni należy do typu *double precision,* którego funkcja nie akceptuje (więcej szczegółów w tym wątku: https://stackoverflow.com/questions/13113096/how-to-round-an-average-to-2-decimal-places-in-postgresql). Musimy więc zastosować rzutowanie, tj. konwersję *double precision* na *numeric*. Jak pamiętamy, rzutowanie możemy wykonać stosując zapis **::nowy_typ_danych**.

FROM powiaty
ORDER BY area **DESC**
LIMIT 5

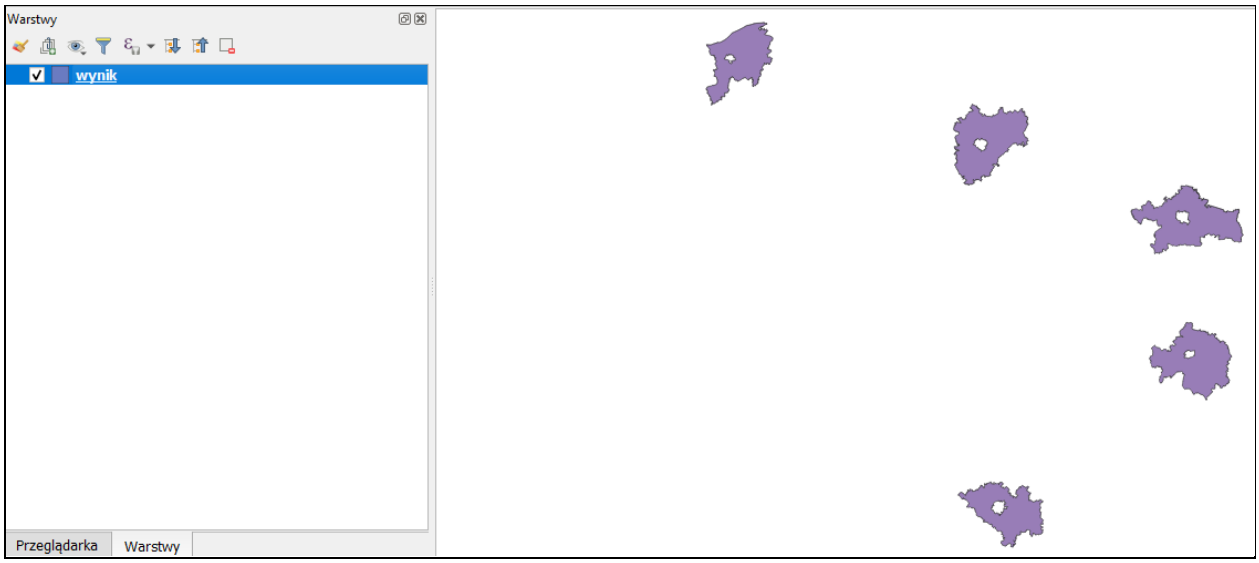
Pozostała część instrukcji jest mniej skomplikowana. Wskazujemy tabelę, z której pozyskujemy kolumny, wprowadzamy porządkowanie według pola powierzchni w kolejności malejącej (słowo kluczowe **DESC**) i ograniczamy liczbę rekordów w tabeli wynikowej do 5. Zwróćmy również uwagę na to, że w instrukcji nie uwzględniliśmy geometrii obiektów. Z perspektywy obliczeń nie jest to problematyczne, jednakże w tym momencie nie moglibyśmy wyświetlić wizualizacji wyniku. Zmodyfikujmy więc bazową instrukcję o kolumnę z geometrią:

SELECT id **AS** id, jpt_nazwa_ **AS** nazwa,
geom,
round((ST_Area(geom)/10^6)::numeric,2) **AS** area
FROM powiaty
ORDER BY area **DESC**
LIMIT 5

Aby wyświetlić wynik działania klikamy na *Wczytaj jako nową warstwę*. Możemy również wpisać unikalny przedrostek (czyli *de facto* nazwę warstwy wynikowe). Poszczególne kroki puentujemy, klikając na *Wczytaj*.



Możemy teraz zminimalizować widok okna *DB Managera* i przejść do oględzin wyniku w oknie mapy:



W tym momencie dysponujecie Państwo w pełni funkcjonalną instrukcją, którą można wykorzystać z powodzeniem w kolejnych działaniach. W ramach utrwalenia przyswojonej wiedzy poproszę teraz o samodzielne wykonanie następującego zadania:

Ćwiczenie

Wykonaj zbliżone do powyższego działanie dla gmin. Zastosuj dobór i nazewnictwo kolumn analogiczne do tych z przykładu. Tabela wynikowa powinna jednak zawierać nie 5, a 10 geometrii. Dane porządkujemy **ROSNĄCO** według pola powierzchni *gminy*, wyrażonego w **HEKTARACH**.

Proponowany sposób rozwiązania:

```
SELECT id AS id, jpt_nazwa_ AS nazwa,
geom,
round((ST_Area(geom)/10^4)::numeric,2) AS area
FROM gminy
ORDER BY area
LIMIT 10
```

Łączenie danych na podstawie relacji geometrycznych

Prawdziwa moc PostGISa drzemie w możliwości analizowania relacji przestrzennych między obiektami z różnych tabel. Przyjęte kryteria lokalizacji możemy zakodować w instrukcji,

uzyskując tym samym wymierny wynik. Przykład - interesuje nas wykaz rezerwatów przyrody znajdujących się w powiecie otwockim. Dane do naszej instrukcji pozyskamy z tabel *powiaty* i *rezerwaty*. Dokonamy również złączenia, tym razem jednak na podstawie relacji przestrzennej. Zanim jednak przystąpimy do obliczeń, sprawdźmy czy nasze dane o powiatach i rezerwatach mają poprawnie określone układy współrzędnych. W tym celu zaznaczamy warstwę na liście po lewej stronie, po prawej zaś klikamy na zakładkę *Informacje*:

Z informacji wynika, że tabela *rezerwaty* nie ma zadeklarowanego układu współrzędnych. Można byłoby również zmienić nazwę kolumny przechowującej geometrię (geometryproperty). Obie modyfikacje możemy wykonać bezpośrednio na tabeli źródłowej. Zacznijmy od przyporządkowania układu współrzędnych. Wykorzystamy do tego klauzulę UpdateGeometrySRID. Przyjmuje ona trzy argumenty: nazwę tabeli, nazwę kolumny przechowującej geometrię oraz kod EPSG docelowego układu współrzędnych. Zapis działania przedstawia się następująco:

SELECT UpdateGeometrySRID(‘rezerwaty’,’geometryproperty’,2180)

Po wykonaniu instrukcji zostaniemy poinformowani o przeprowadzonej zmianie:

```
1 SELECT UpdateGeometrySRID('rezerwaty','geometryproperty',2180)
```

Uruchom | 1 wierszy, 0.434 sekund | Utwórz widok | Wyczyść

	updategeometrysrid
1	public.rezerwaty.geometryproperty SRID changed to 2180

Pozostaje jeszcze zmiana nazwy kolumny. Tym razem posłużymy się zapisem ALTER TABLE / RENAME COLUMN:

ALTER TABLE rezerwaty
RENAME COLUMN geometryproperty **TO** geom

Sprawdźmy jeszcze raz zakładkę z informacjami nt. tabeli *rezerwaty*:

rezerwaty

Informacje

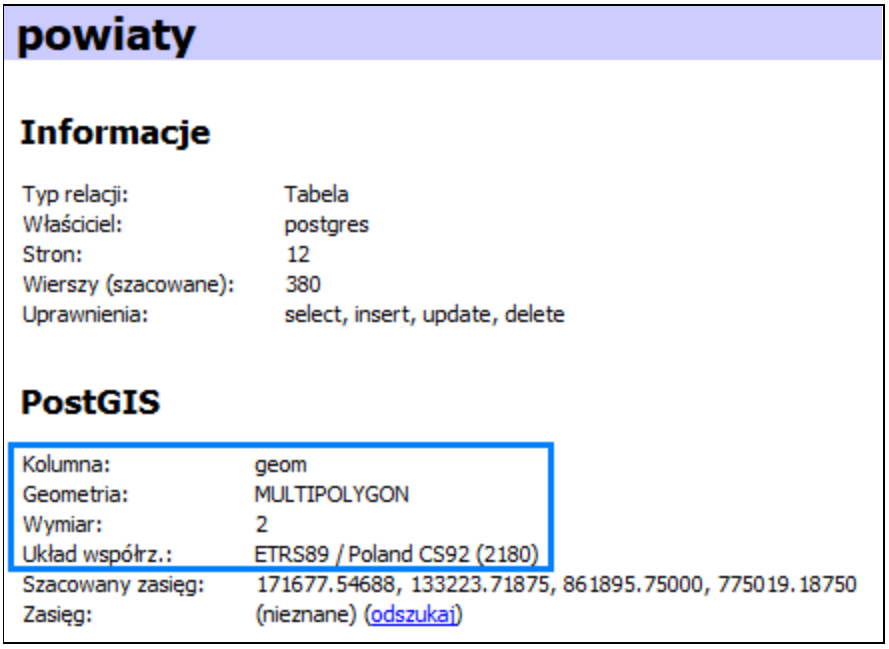
Typ relacji:	Tabela
Właściciel:	postgres
Stron:	23
Wierszy (szacowane):	296
Uprawnienia:	select, insert, update, delete

PostGIS

Kolumna:	geom
Geometria:	POLYGON
Wymiar:	2
Układ współrz.:	ETRS89 / Poland CS92 (2180)
Zasięg:	(nieznane) (odszukaj)

Jak widać na załączonej grafice, zmiany zostały utrwalone.

W przypadku tabeli *powiaty* sprawa przedstawia się nieco inaczej. Posiada ona bowiem przyporządkowany układ współrzędnych, których różni się od tego dla tabeli *rezerwaty*. Metoda rozwiązania problemu pozostaje jednak ta sama - możemy skorzystać z funkcji **UpdategeometrySRID()**. Po kliknięciu na refresh  zmiany powinny wyświetlić się w informacjach o tabeli:



Możemy zatem wrócić do przygotowania wykazu rezerwatów leżących w granicach powiatu otwockiego:

```
SELECT rz.gml_id, rz.nazwa, rz.geom
FROM rezerwaty rz, powiaty pw
WHERE ST_Within(rz.geom,pw.geom)
AND pw.jpt_nazwa_ = 'otwocki'
```

W zapytaniu pojawiają się odniesienia do dwóch warstw - rezerwatów i powiatów. Złączenie wykonaliśmy na podstawie relacji przestrzennej “zwróć tylko te obiekty, które w całości znajdują się w granicach innego obiektu”. Wykorzystaliśmy do tego celu funkcję **ST_Within()**, przyjmującą dwa argumenty: pierwszy to geometria zawierająca się w-, druga zaś to geometria, w ramach której zawierają się geometrie z argumentu pierwszego. W tym zapytaniu moglibyśmy również skorzystać z funkcji **ST_Contains()**. Zwraca ona ten sam wynik różniący się jednak kolejnością argumentów. W pierwszej kolejności podajemy geometrię zawierającą, w drugim zaś tę (lub zbiór tych), które zawierają się w pierwszej:

```
SELECT rz.gml_id, rz.nazwa, rz.geom
FROM rezerwaty rz, powiaty pw
WHERE ST_Contains(pw.geom,rz.geom)
AND pw.jpt_nazwa_ = 'otwocki'
```

Zauważmy, że wyrażenie zwróciło 8 obiektów, niemniej liczba unikalnych nazw jest mniejsza:

TCRZA-14-28CE...	Bagno Bocianowskie	010300002084080000010...
TCRZA-14-DEC...	Wymięklizna	010300002084080000010...
TCRZA-14-AF52...	Szerokie Bagno	010300002084080000010...
TCRZA-14-A780...	Mszar Pogorzelski	010300002084080000010...
TCRZA-14-28CE...	Bagno Bocianowskie	010300002084080000010...

Możemy pozbyć się zduplikowanych nazw stosując słowo kluczowe **DISTINCT**, lub pogrupować wyniki według nazwy:

```
1 SELECT DISTINCT rz.gml_id, rz.nazwa, rz.geom
2 FROM rezerwaty rz, powiaty pw
3 WHERE ST_Contains(pw.geom,rz.geom)
4 AND pw.jpt_nazwa_ = 'otwocki'
5
```

Uruchom | 4 wierszy, 0.663 sekund | Utwórz widok | Wyczyść

	gml_id	nazwa	geom
1	TCRZA-14-28CE...	Bagno Bocianowskie	010300002084080000010000000140...
2	TCRZA-14-A780...	Mszar Pogorzelski	010300002084080000010000000060...
3	TCRZA-14-AF52...	Szerokie Bagno	010300002084080000010000000090...
4	TCRZA-14-DEC...	Wymięklizna	0103000020840800000100000000B0...

Lub

```
1 SELECT rz.gml_id, rz.nazwa, rz.geom
2 FROM rezerwaty rz, powiaty pw
3 WHERE ST_Contains(pw.geom,rz.geom)
4 AND pw.jpt_nazwa_ = 'otwocki'
5 GROUP BY 1,2,3
```

Uruchom | 4 wierszy, 0.391 sekund | Utwórz widok | Wyczyść

	gml_id	nazwa	geom
1	TCRZA-14-28CE...	Bagno Bociano...	0103000020840...
2	TCRZA-14-A780...	Mszar Pogorzel...	0103000020840...
3	TCRZA-14-AF52...	Szerokie Bagno	0103000020840...
4	TCRZA-14-DEC...	Wymięklizna	0103000020840...

Proszę zauważyć, że po klauzuli **GROUP BY** zamiast nazw kolumn podaliśmy po prostu ich numery określające kolejność występowania po klauzuli **SELECT**. W obu przypadkach efekt końcowy jest jednak dokładnie taki sam.

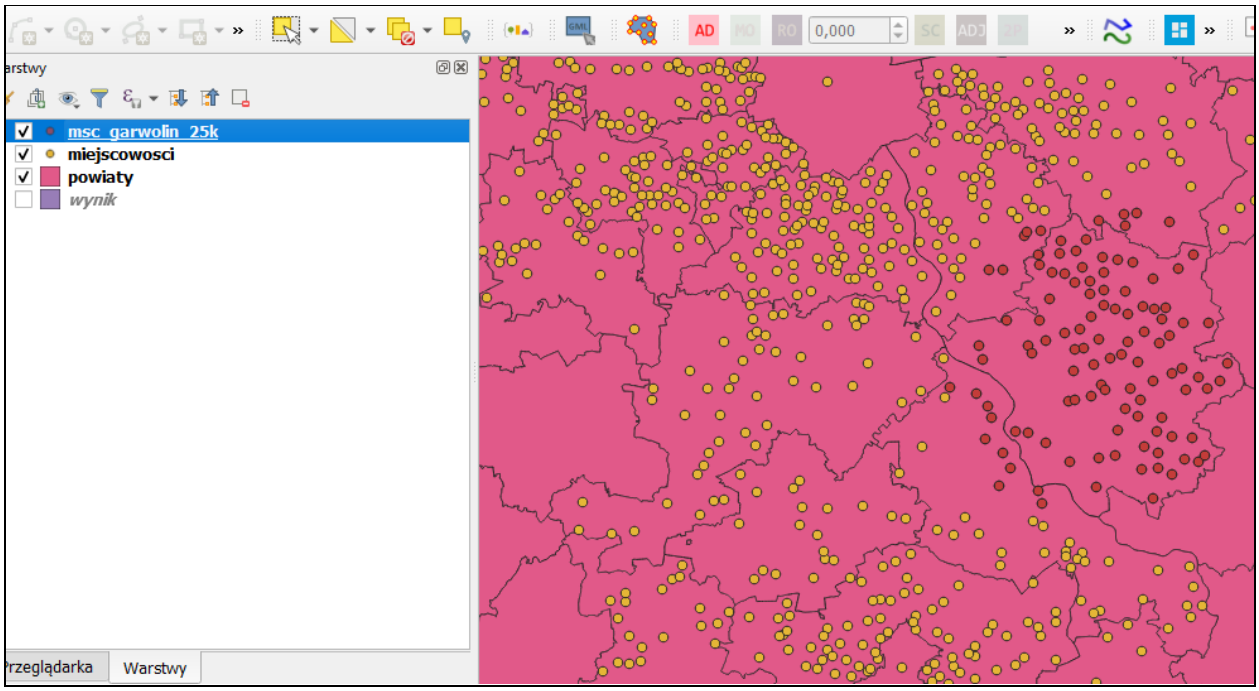
Postgis daje nam również możliwość wyszukiwania obiektów znajdujących się w określonej odległości od celu. Załóżmy, że interesuje nas, ile miejscowości znajduje się w odległości 25 km od centralnego punktu powiatu garwolińskiego. Zaczynamy od ujednolicenia układów współrzędnych (na początku wykonujemy tylko poniższą instrukcję):

SELECT UpdategeometrySRID('miejscowosci','geom','2180')

Następnie przechodzimy do stworzenia właściwego wyrażenia:

SELECT ms.nazwa as msc, ms.geom as geom
FROM miejscowosci ms, powiaty pw
WHERE ST_Dwithin(ms.geom,
ST_Centroid(pw.geom),25000)
AND pw.jpt_nazwa_ = 'garwoliński'

W tym przypadku zastosowaliśmy funkcję **ST_DWithin()**, która zwraca informację prawda/fałsz na pytanie, czy obiekty z tabeli a znajdują się w określonej odległości od obiektów z tabeli b. Funkcję ST_Centroid zastosowaliśmy, by uzyskać geometrię punktu leżącego w centralnej części poligonu reprezentującego granice powiatu garwolińskiego.

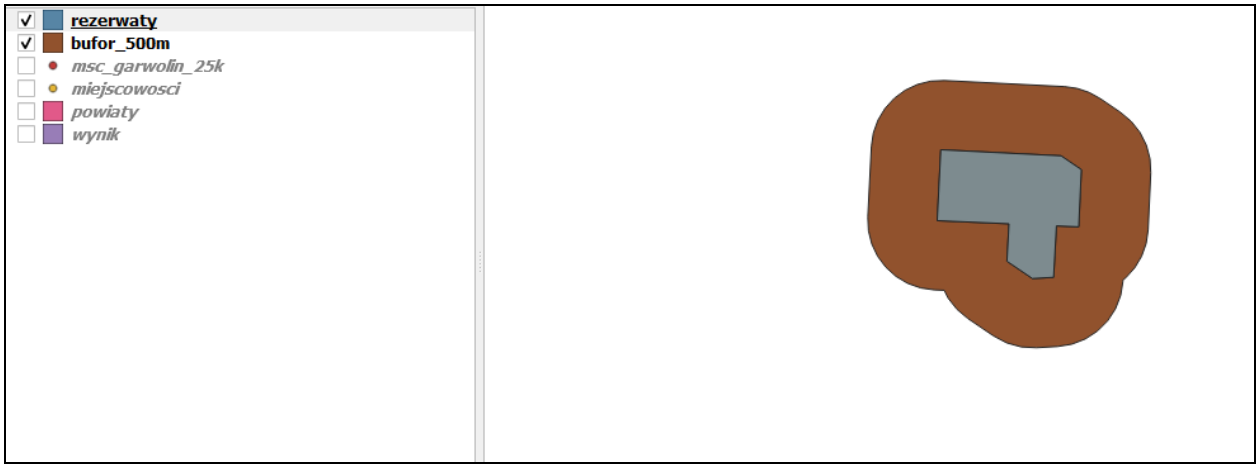


Analizy przestrzenne

Oprócz zapytań przestrzennych, PostGIS posiada funkcje służące do analiz geometrii, które na podstawie warstw źródłowych tworzą tabelę wynikową. Przykładowe funkcje to bufor, suma przestrzenna (Agreguj, Dissolve), iloczyn przestrzenny (Iloczyn, Intersect), różnica przestrzenna (Różnica, Difference). W kolejnych ćwiczeniach zapoznamy się z ich działaniem.

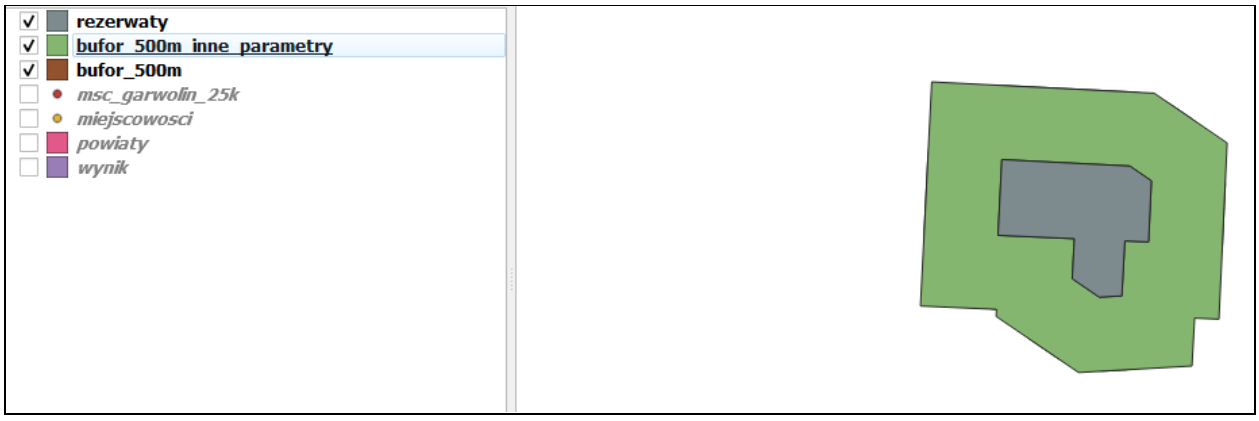
Zacznijmy od bufora. W ramach ćwiczenia spróbujemy wygenerować granice strefy rozciągającej się na odległość 500 m od rezerwatu “Wymięklizna”. Struktura takiego zapytania jest stosunkowo prosta:

```
SELECT DISTINCT nazwa, ST_Buffer(geom,500) AS
geom
FROM rezerwaty
WHERE nazwa = 'Wymięklizna'
```

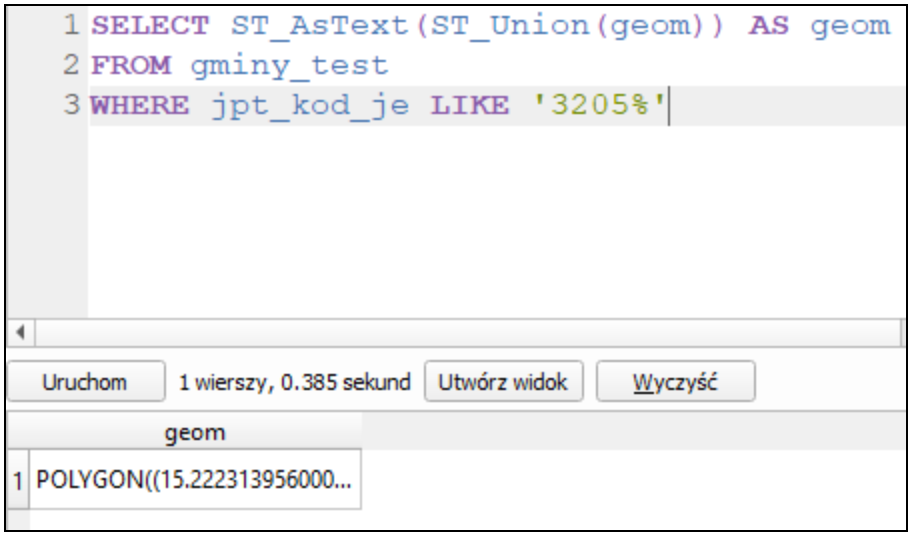


Domyślny bufor posiada zaokrąglone granice. Szybki rzut oka na oficjalną dokumentację (https://postgis.net/docs/ST_Buffer.html) pokazuje, że parametrami wpływającymi na kształt złączenia i zakończenia możemy sterować za pomocą odpowiednich argumentów:

```
SELECT DISTINCT nazwa,
ST_Buffer(geom,500, 'endcap=flat join=mitre') AS geom
FROM rezerwaty
WHERE nazwa = 'Wymięklizna'
```

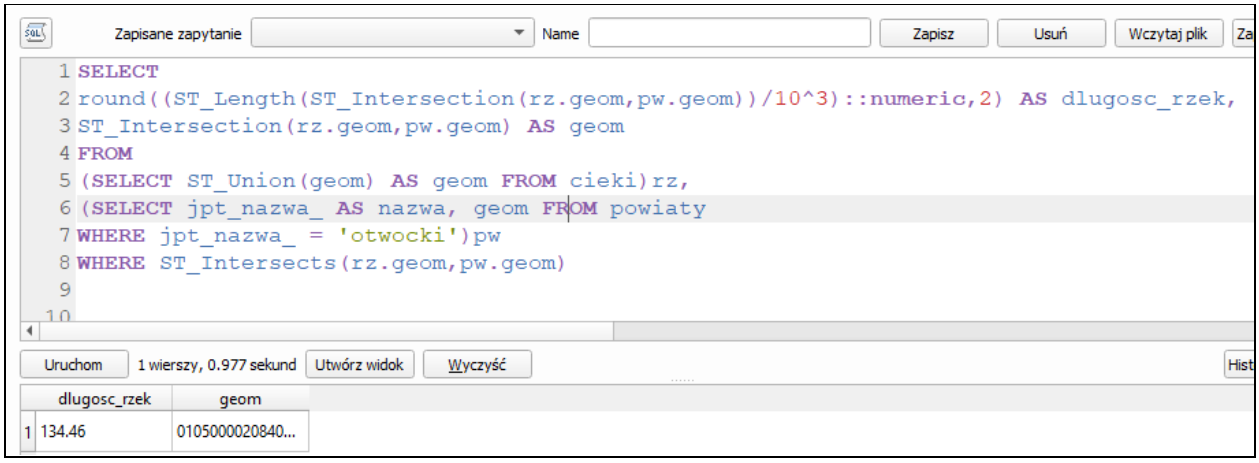


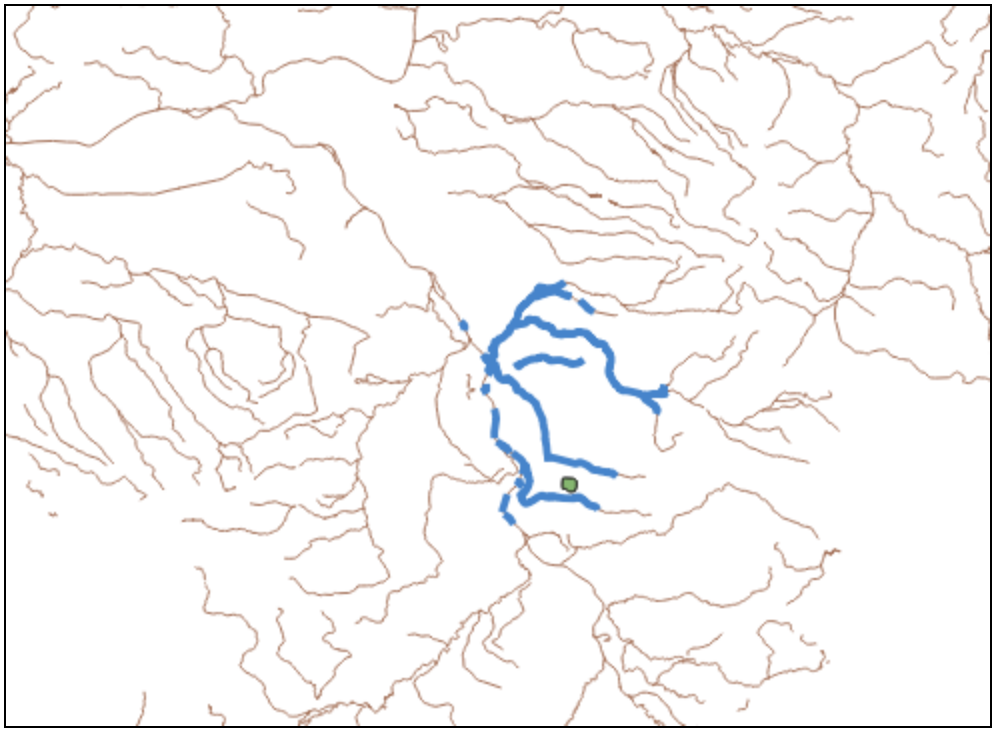
Funkcja ST_Union umożliwia agregację pomniejszych obiektów przestrzennych w jedną zwartą geometrię. W trakcie tej operacji granice wewnętrzne obiektów ulegają zatarciu. Korzystając z ST_Union możemy przykładowo uzyskać granicę powiatu na podstawie gmin posiadających te same cztery pierwsze cyfry kodu TERYT:



Iloczyn przestrzenny (Intersect) zwraca część wspólną dwóch geometrii. Wykorzystujemy go chociażby do obliczenia własności części obiektów znajdujących się w granicach innych obiektów, np. długości obiektów liniowych w granicach poligonów. Zademonstrujemy działanie iloczynu wykonując ćwiczenie polegające na obliczeniu długości sieci rzecznej w powiecie otwockim:

SELECT
round((ST_Length(ST_Intersection(rz.geom,pw.geom))/10^3)::numeric,2) AS dlugosc_rzek,
ST_Intersection(rz.geom,pw.geom) AS geom
FROM
(**SELECT** ST_Union(geom) AS geom **FROM** cieki)rz,
(**SELECT** jpt_nazwa_ AS nazwa, geom **FROM** powiaty
WHERE jpt_nazwa_ = 'otwocki')pw
WHERE ST_Intersects(rz.geom,pw.geom)





Różnica (Difference) zwraca część geometrii A, która nie przecina się z geometrią B. Rezultat działania różnicy można przedstawić równaniem A - ST_Intersection(A,B). Funkcję tę można wykorzystać między innymi do wygenerowania pierściennych stref zasięgu danego zjawiska, rozchodzących się koncentrycznie od miejsca centralnego (np. centrum skażenia, miejsce zagrożone wybuchem, emiter hałasu, etc.).

W kolejnym ćwiczeniu napiszemy instrukcję generującą tego typu pierścienie. Za punkt centralny przyjmiemy centroid z granic powiatu otwockiego. Pierwszy bufor będzie miał promień 5 km, natomiast dwa kolejne pierścienie obejmować będą obszary między 5-10 km i 10-25 km. W tabeli wynikowej powinny pojawić się trzy rekordy (po jednym dla każdej ze stref) oraz dwa atrybuty + pole z geometrią (pow. strefy w kilometrach kwadratowych oraz liczba miejscowości znajdujących się w granicach poszczególnych stref). Zadanie jest stosunkowo złożone. Zaczynamy bowiem od stworzenia geometrii dla poszczególnych stref; następnie złączymy je w jedną warstwę z trzema rekordami przy użyciu funkcji **UNION**. Dla tak przygotowanej warstwy obliczymy liczbę miejscowości, których geometrie zawierają się w granicach poszczególnych stref.

Definicja pierwszej strefy przedstawia się następująco:

```
SELECT ST_Buffer(cn.geom,5*10^3) AS geom,
ST_Area(ST_Buffer(cn.geom,5*10^3)) AS pow_strefy
FROM
(SELECT ST_Centroid(geom) AS geom
FROM powiaty
WHERE jpt_nazwa_ = 'otwocki')cn
```

Do tak utworzonego bufora o promieniu 5km (5*10^3) musimy dołączyć kolejne strefy. Do łączenia warstw zawierających te same kolumny możemy użyć słowa kluczowego **UNION**.

```
SELECT ST_Buffer(cn.geom,5*10^3) AS geom,
ST_Area(ST_Buffer(cn.geom,5*10^3)) AS pow_strefy
FROM
(SELECT ST_Centroid(geom) AS geom
FROM powiaty
WHERE jpt_nazwa_ = 'otwocki')cn

UNION

SELECT
ST_Difference(
ST_Buffer(cn.geom,10*10^3),ST_Buffer(cn.geom,5*10^3)) AS
geom,
ST_Area(
ST_Difference(
ST_Buffer(cn.geom,10*10^3),ST_Buffer(cn.geom,5*10^3))) AS
pow_strefy
FROM
(SELECT ST_Centroid(geom) AS geom
FROM powiaty
WHERE jpt_nazwa_ = 'otwocki')cn

UNION
```

Kolejna instrukcja generuje pierścień będący geometryczną wizualizacją różnicy dwóch buforów (10 i 5 km). Reszta wyrażenia nie różni się od poprzedniego. Pamiętamy o dodaniu słowa **UNION** na końcu tekstu - nasz “agregat” warstw musimy rozszerzyć o trzeci, ostatni segment:

```
SELECT
ST_Difference(
ST_Buffer(cn.geom,10*10^3),ST_Buffer(cn.geom,5*10^3)) AS
geom,
ST_Area(
ST_Difference(
ST_Buffer(cn.geom,25*10^3),ST_Buffer(cn.geom,10*10^3))) AS
pow_strefy
FROM
(SELECT ST_Centroid(geom) AS geom
FROM powiaty
WHERE jpt_nazwa_ = 'otwocki')cn
```

Całe to wyrażenie weźmiemy niebawem w nawias i potraktujemy jako tabelę. Teraz jednak musimy stworzyć instrukcję zwracającą liczbę miejscowości w poszczególnych strefach:

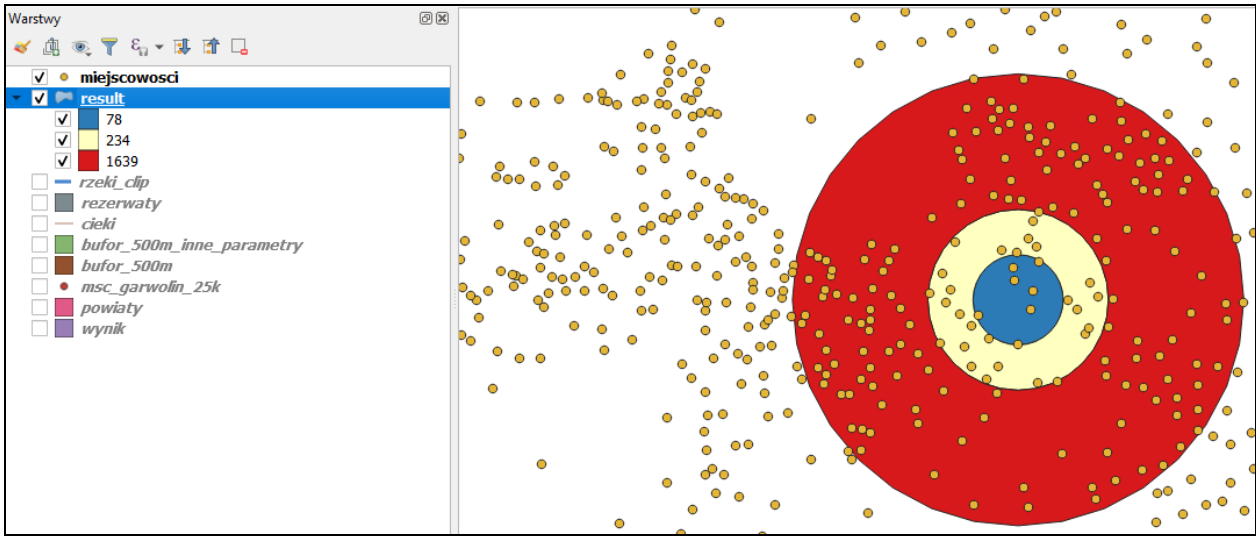
```
SELECT str.geom,
round((str.pow_strefy/10^6::numeric)) AS pow_strefy
,
COUNT(msc.geom) AS liczba_msc
FROM miejscowosci msc,

        tutaj wklejamy wcześniejszą instrukcję
              ujętą w nawiasy

WHERE ST_Intersects(msc.geom, str.geom)
GROUP BY 1,2
```

Wykonujemy analizę. W rezultacie otrzymujemy trzy obiekty poligonowe wraz z atrybutem zawierającym liczbę miejscowości w każdej ze stref:

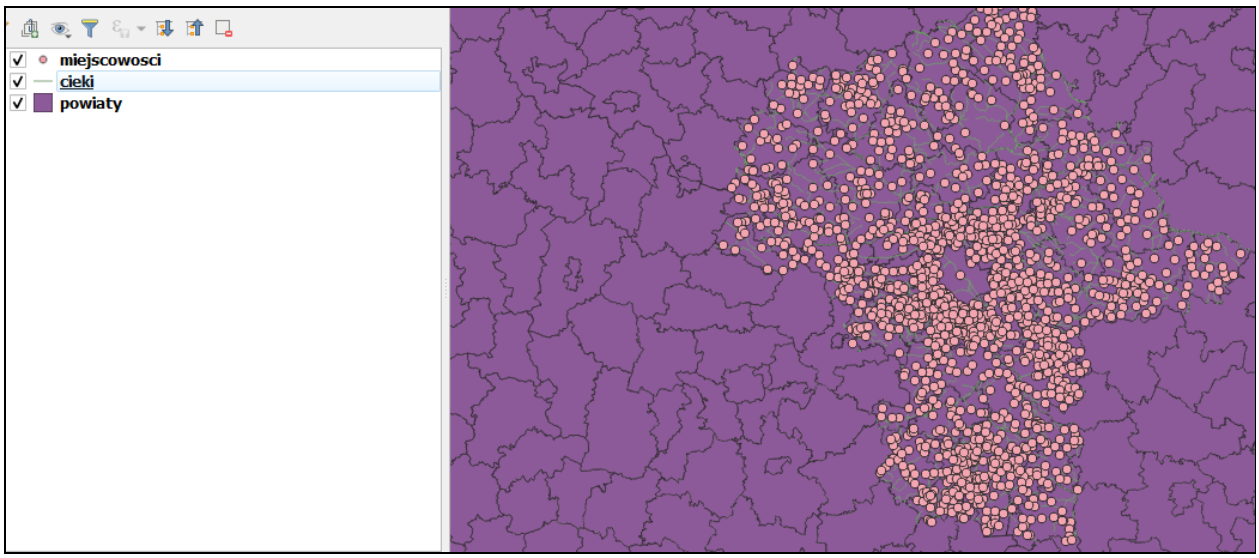
geom	pow_strefy	liczba_msc
0103000020840...	78.0	7
0103000020840...	234.0	25
0103000020840...	1639.0	152



Zaawansowana integracja QGIS i PostGIS

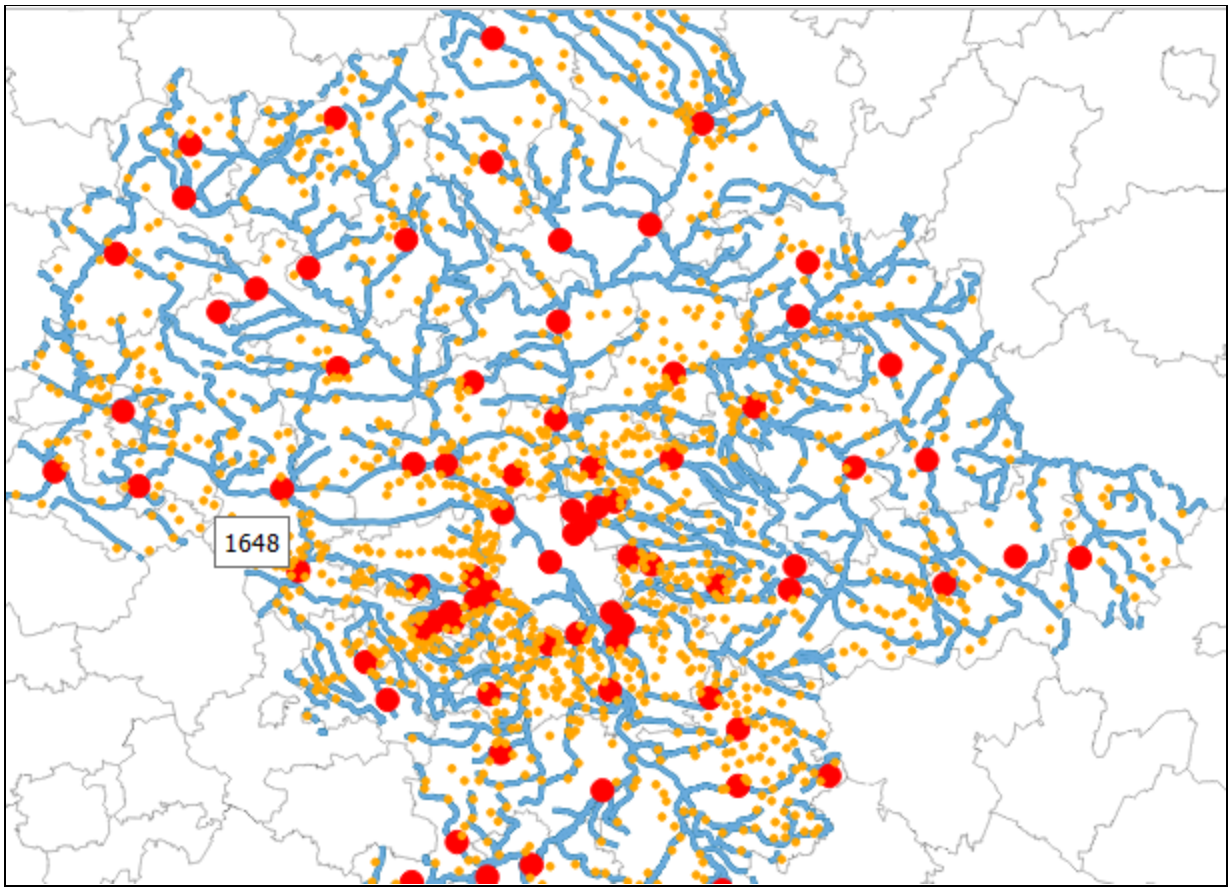
Integracja aplikacji PostgreSQL z nakładką PostGIS z programem QGIS stwarza dodatkowe możliwości w zakresie przechowywania i wizualizacji danych, jak również zapisywania projektów. Symbolizacje przygotowane dla poszczególnych warstw można zapisywać bezpośrednio w bazie danych.

Stwórzmy nowy projekt QGIS, przejdźmy do bazy danych *postgres* i dodajmy do widoku mapy warstwy *powiaty*, *miejscowości* i *cieki*:

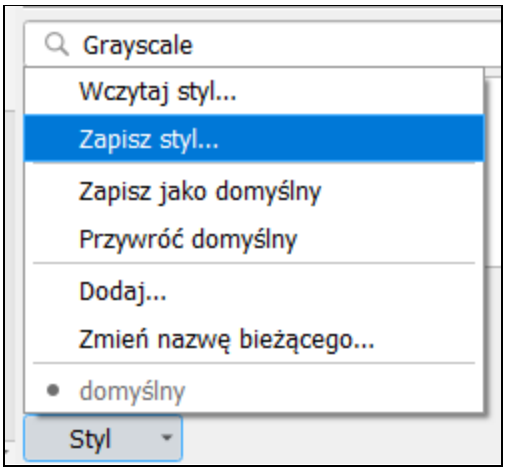


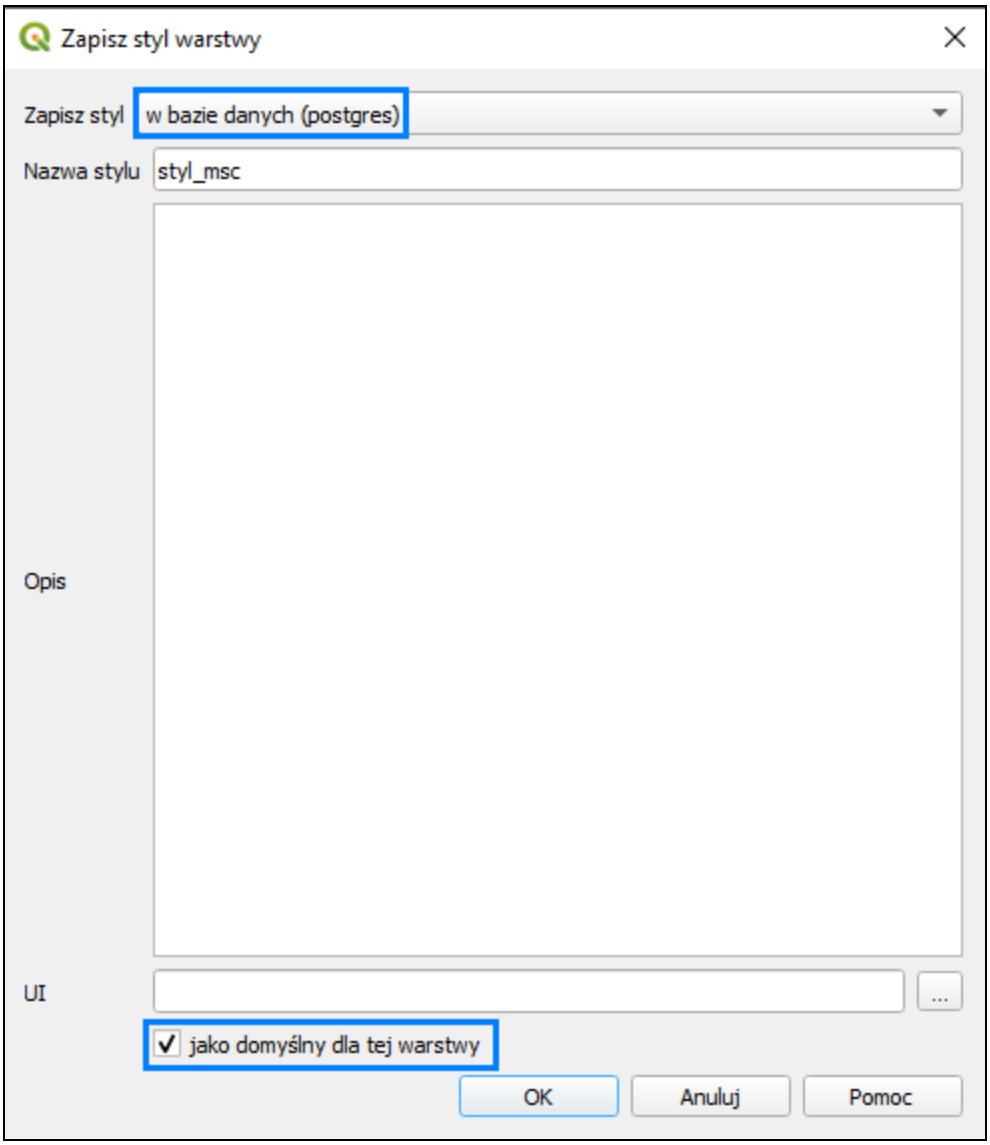
Żadna z tabel nie ma przypisanej symbolizacji, w związku z czym u każdego wyświetlą się one inaczej. Na szczęście nic nie stoi na przeszkodzie, by zmodyfikować bazową symbolizację i zapisać zmiany bezpośrednio w bazie.

Zacznijmy od stylizacji warstw (w tym zakresie pozostawiam Państwu wolną rękę):

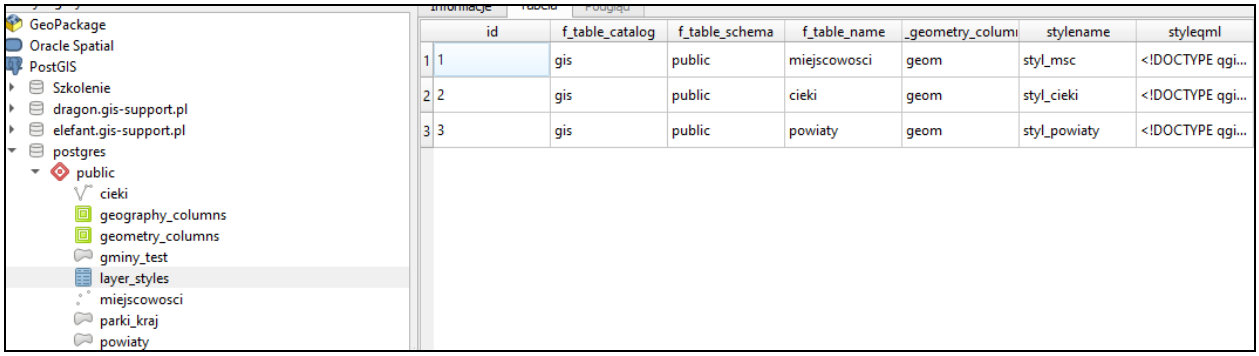


Przechodzimy do zapisu symbolizacji:

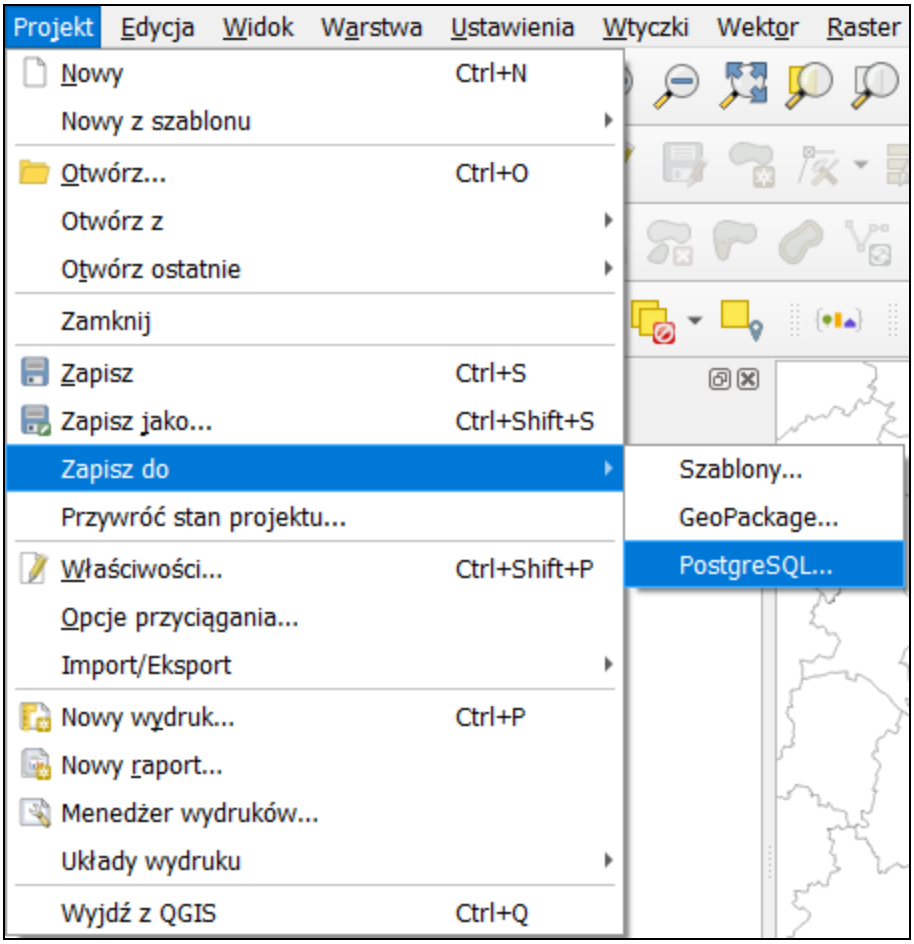

Grayscale
Wczytaj styl...
Zapisz styl...
Zapisz jako domyślny
Przywróć domyślny
Dodaj...
Zmień nazwę bieżącego...
domyślny
Styl

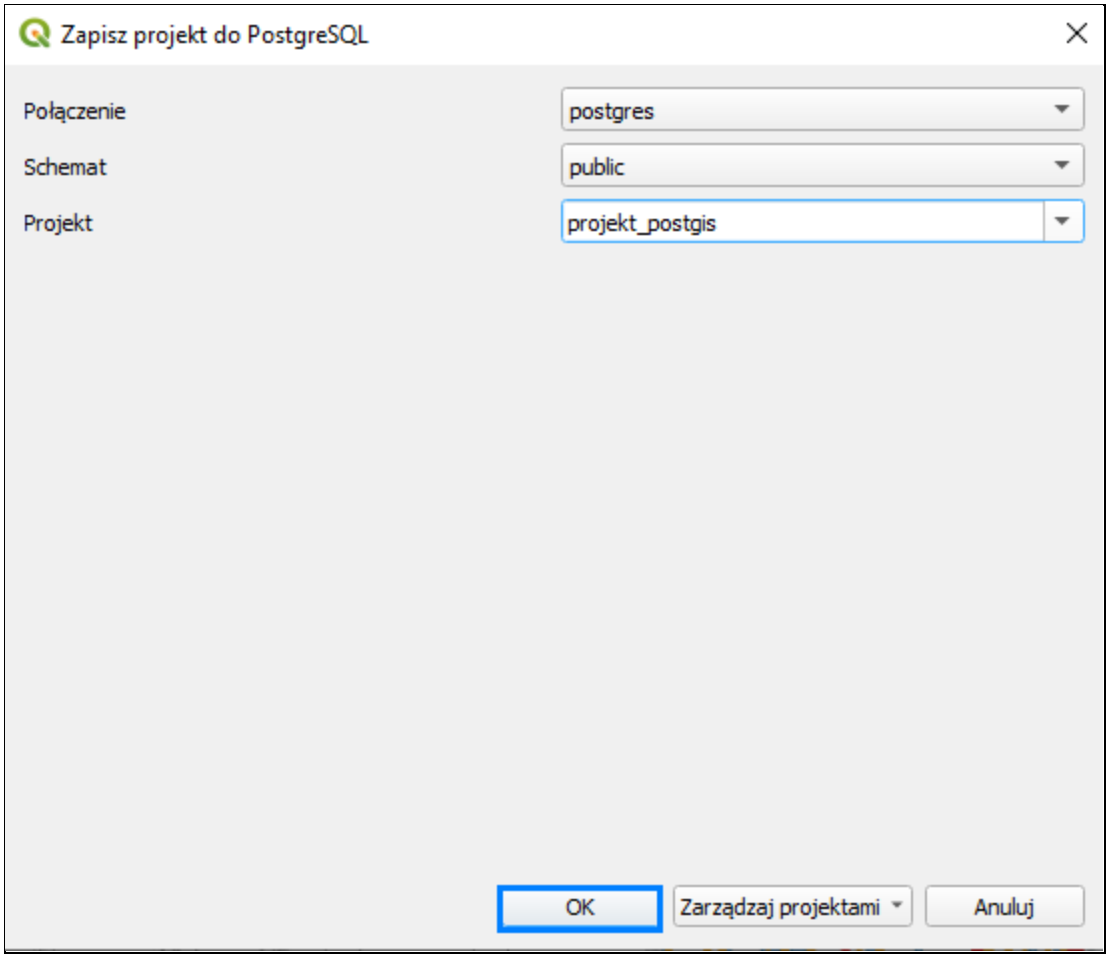

Zapisz styl warstwy
Zapisz styl
w bazie danych (postgres)
Nazwa stylu
styl_msc
Opis
UI
...
jako domyślny dla tej warstwy
OK
Anuluj
Pomoc

W analogiczny sposób utrwalamy symbolizacje dla pozostałych warstw. Od teraz zapisane style przechowywane będą w nowej tabeli *layerstyles* w schemacie *public*:

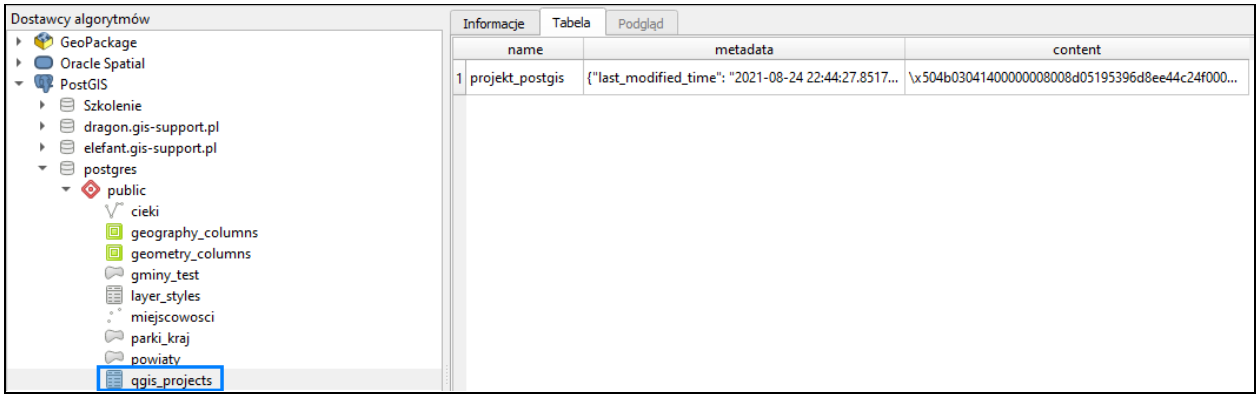


Jak już wspomniano, w bazie danych możemy również zapisywać całe projekty:

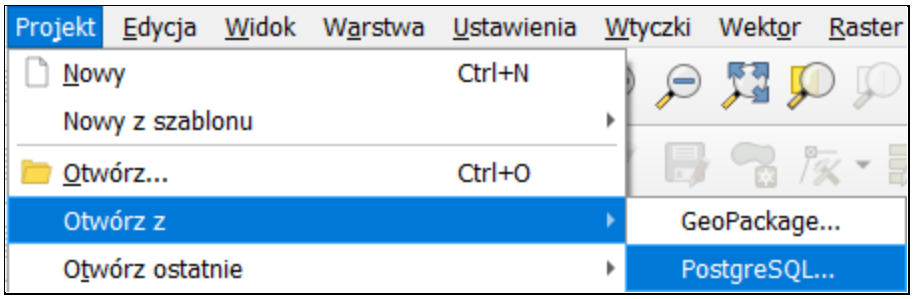




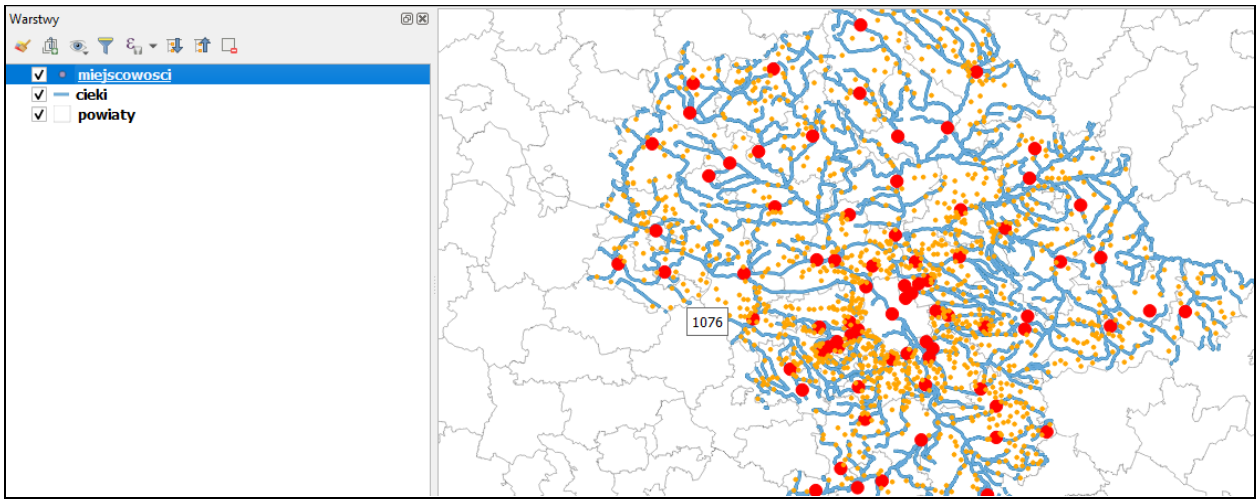
Po kliknięciu na *OK* projekt zostanie zapisany w bazie w odrębnej tabeli o nazwie *qgis_projects*:



Zamknijmy teraz bieżący projekt bez zapisywania a następnie spróbujmy wczytać ten, który zapisaliśmy przed chwilą w bazie *postgres*:



Efekt końcowy jak najbardziej pozostaje w zgodzie z naszymi oczekiwaniami:



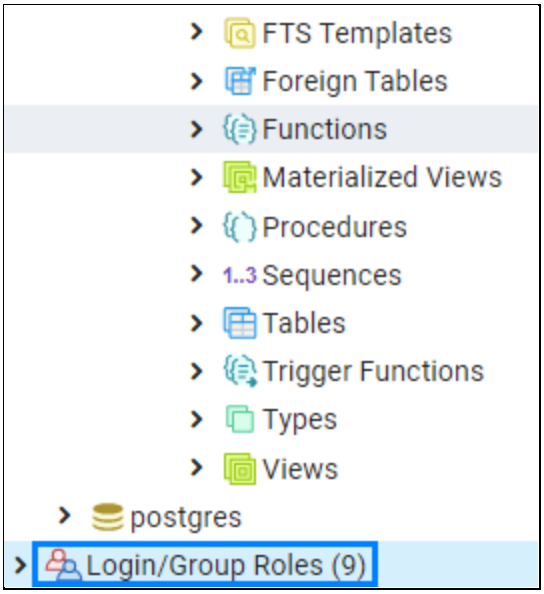
Administrowanie bazą danych - tworzenie użytkowników, nadawanie uprawnień

Tworzenie nowego użytkownika

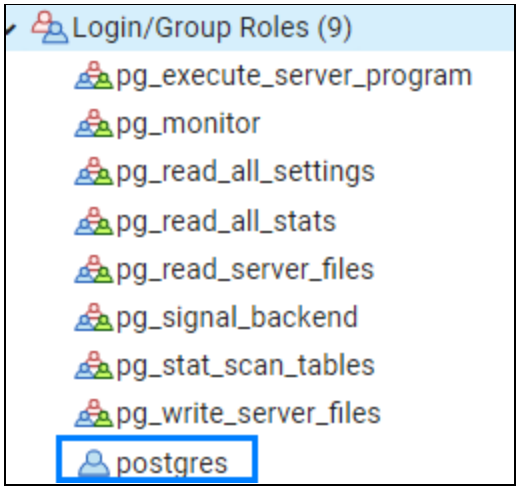
Jedną z fundamentalnych idei stojących za systemami baz danych jest możliwość udostępniania przechowywanych w niej informacji innym użytkownikom. PostgreSQL udostępnia narzędzia do tworzenia nowych kont użytkowników oraz zarządzanie ich

uprawnieniami, takimi jak prawo do wyświetlania, edycji, czy tworzenia nowych baz. Operacje te możemy przeprowadzać z poziomu interfejsu graficznego lub poprzez uruchomienie stosownej instrukcji. W tym przypadku skupimy się na opcji “graficznej”, gdyż jest prostsza i wygodniejsza w obsłudze.

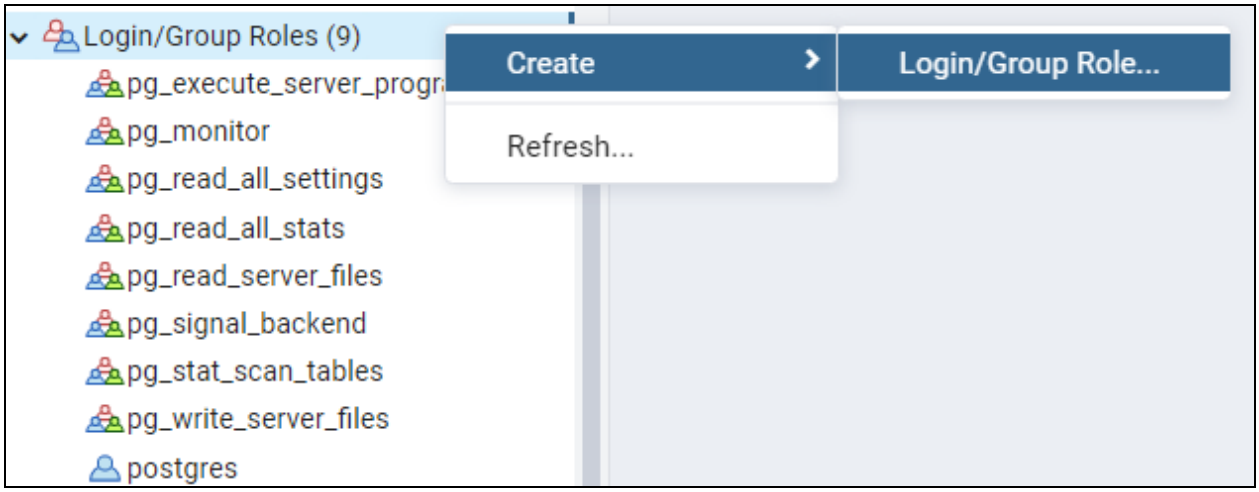
Uruchamiamy aplikację pgAdmin; wśród kategorii widocznych po lewej stronie odszukujemy *Login/Group roles*.



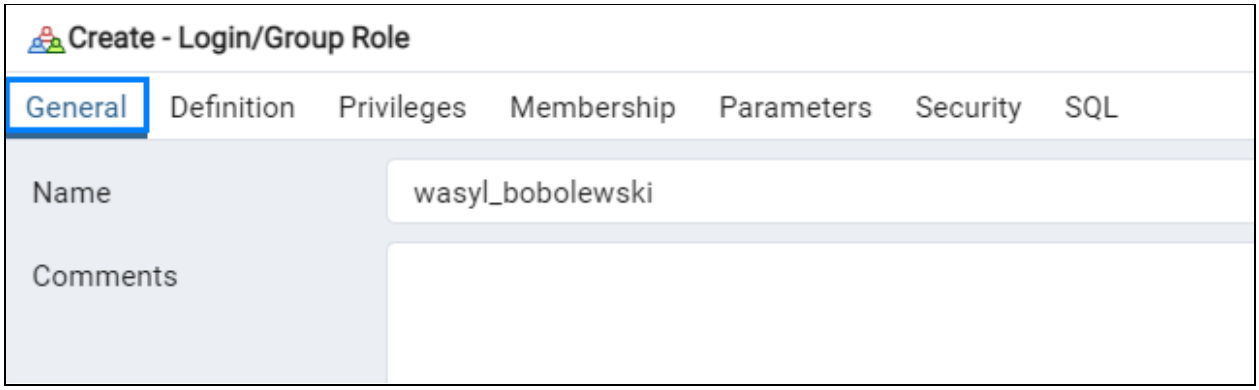
W wersji bazowej zakładka zawiera szereg uprawnień systemowych oraz jednego *superusera*, (postgres) tworzonych w procesie instalacji:



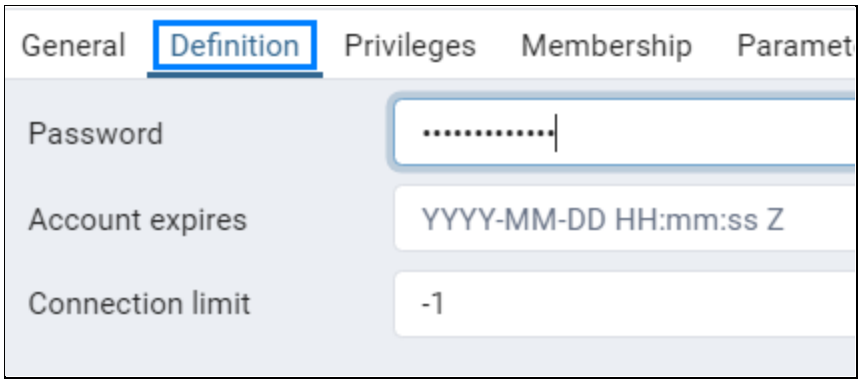
Nowego użytkownika dodajemy, klikając prawym przyciskiem myszy na *Login/Group roles* i wybierając



Następnie w zakładce *General* wprowadzamy nazwę użytkownika;

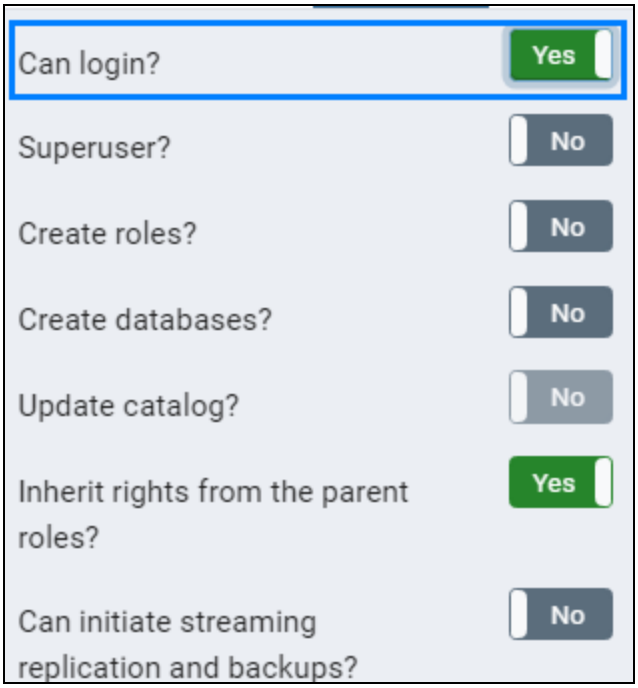


Hasło użytkownika w zakładce *Definition*;



Oprócz ustanowienia hasła mamy również możliwość określenia terminu upływu ważności konta (w zapisie YYYY-MM-DD (...), czyli rok-miesiąc-dzień godzina-minuta-sekunda) oraz limitu liczby połączeń, wyrażonego za pomocą dodatniej liczby całkowitej. Wartość -1 oznacza brak jakichkolwiek limitów.

Następnie przechodzimy do zakładki *Privileges*:



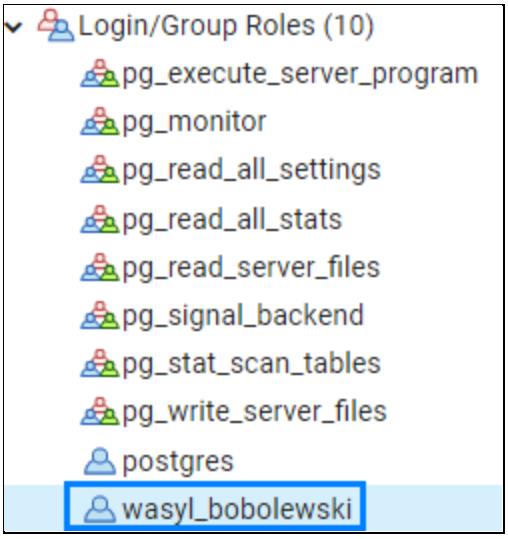
Na tym etapie nadamy użytkownikowi jedynie prawo do logowania. Kwestią dostępu do edycji zajmiemy się później.

Dodatkowo w zakładce *SQL* możemy podejrzeć, jak proces tworzenia użytkownika oraz nadawania uprawnień wygląda od strony kodu:

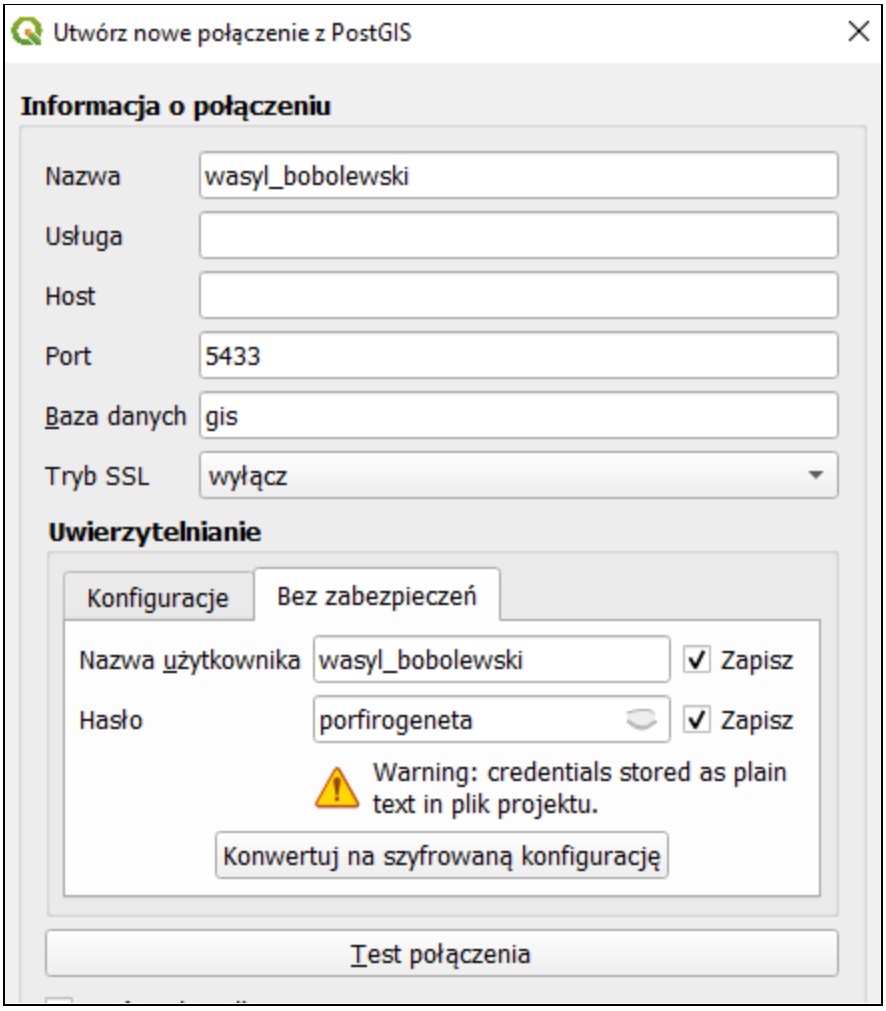
General Definition Privileges Membership Parameters Security SQL

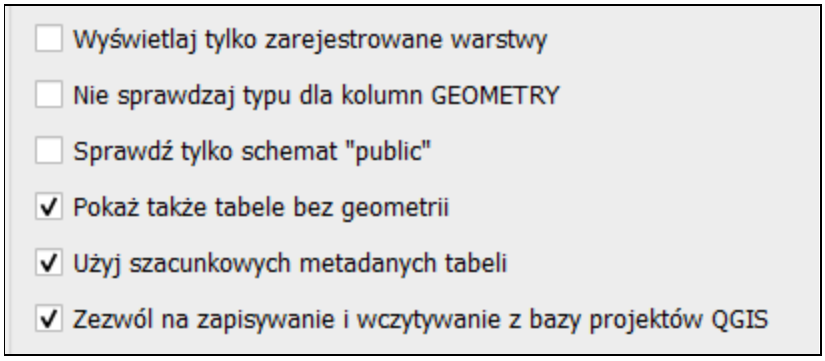
```
1 CREATE ROLE wasyl_bobolewski WITH
2     LOGIN
3     NOSUPERUSER
4     NOCREATEDB
5     NOCREATEROLE
6     INHERIT
7     NOREPLICATION
8     CONNECTION LIMIT -1
9     PASSWORD 'xxxxxx';
```

Zapisujemy zmiany, klikając na *Save*. Nowy użytkownik powinien być już widoczny na liście:



Sprawdźmy teraz, czy możemy utworzyć nowe połączenie z bazą danych, wykorzystując do tego dane z nowoutworzonego konta użytkownika:

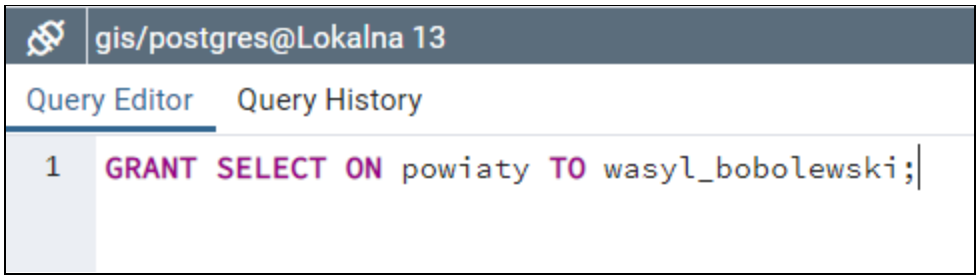




Próba połączenia zakończyła się sukcesem, niepokoi jednak brak danych przestrzennych w schemacie *public*:

Schemat	Tabela	Komentarz	Kolumna	Typ danych	Dane przestrzeni	SRID	ID obiektu	Wybierz po
information_schema								
pg_catalog								
public								
public	geography_co...			brak	NoGeometry		f_table_catalog	☑
public	geometry_col...			brak	NoGeometry		f_table_catalog	☑
public	spatial_ref_sys			brak	NoGeometry			☑

Stało się tak, gdyż domyślnie utworzony użytkownik nie praw do odczytu informacji zawartych w jakiejkolwiek tabeli. Spróbujemy zatem nadać mu uprawnienia do odczytu przy użyciu stosownej instrukcji w pgAdmin:



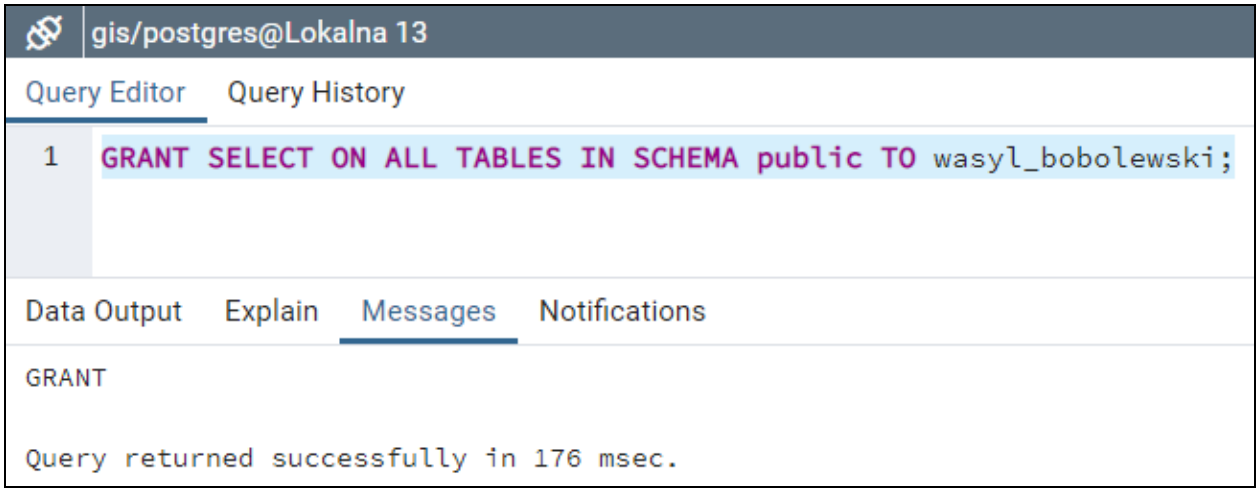
Po wykonaniu instrukcji przejdźmy do aplikacji QGIS i sprawdźmy, czy nasz użytkownik uzyskał dostęp do przeglądania zawartości wymienionej w wyrażeniu tabeli *powiaty*:

public							
public	geography_co...		brak	NoGeometry		f_table_catalog	☑
public	geometry_col...		brak	NoGeometry		f_table_catalog	☑
public	powiaty	geom	Geometry	MultiPolygon	2180		☑
public	spatial_ref_sys		brak	NoGeometry			☑

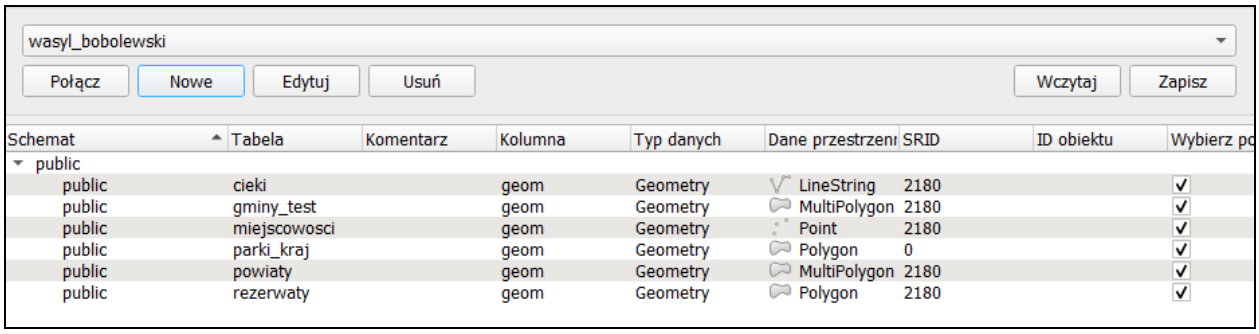
Jak widać na załączonym obrazku, na liście udostępnionych tabel pojawiła się jedna pozycja o nazwie *powiaty*.

Co więcej, możemy w prosty sposób wydatnie rozszerzyć prerogatywy naszego użytkownika, nadając mu prawo do przeglądania wszystkich tabel ze schematu *public*:

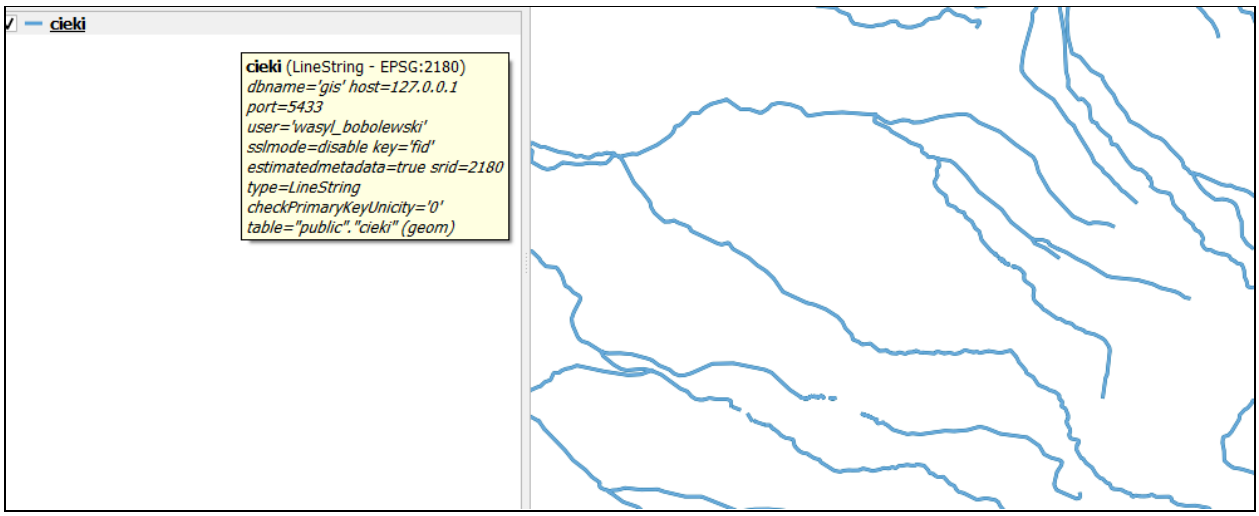
GRANT SELECT ON ALL TABLES IN SCHEMA public TO wasyl_bobolewski;



co oczywiście przełoży się na widok listy udostępnionych warstw:



Nadając prawo do przeglądania zawartości wszystkich tabel, umożliwiliśmy naszemu użytkownikowi dodawanie warstw z przypisaną symbolizacją oraz wczytywanie projektów bezpośrednio z bazy danych.



W sytuacji, gdy któremuś z użytkowników trzeba nadać dodatkowe uprawnienia, np. w zakresie edycji, możemy skorzystać z instrukcji

GRANT SELECT, INSERT UPDATE ON nazwa_tabeli **TO** nazwa_użytkownika

lub

GRANT ALL ON nazwa_tabeli **TO** nazwa_użytkownika.