****

**SQL i PostgreSQL**

**Wstęp**

Znajomość baz danych to temat nieodłączny i niezbędny dla każdego software developera. Bazy danych wykorzystywane są wszędzie tam, gdzie zachodzi potrzeba **trwałego** przechowywania większej ilości informacji. Ciężko bowiem wyobrazić sobie aplikację, np. serwis internetowy, który nie będzie umożliwiał trwałego zapisywania danych wprowadzonych przez użytkowników (w przeciwnym razie użytkownik nie mógłby choćby rejestrować się do serwisu czy wprowadzać informacji o sobie).

W szkoleniu tym skupimy się na omówieniu właśnie wszystkich zagadnień związanych z **SQL-em** i poznamy dogłębnie jedną z implementacji relacyjnych baz danych - **PostgreSQL**.

| **Zapamiętaj!** *SQL (skrót od Structured Query Language)* – to język zapytań wykorzystywany w praktycznie każdej relacyjnej bazie danych. Za pomocą języka SQL można wykonywać operację na bazie danych takie jak pobieranie informacji, wstawianie danych do tabel, modyfikowanie rekordów, czy po prostu tworzenie struktury bazy danych. |
| --- |

**Czym są bazy danych i jakie możemy wyróżnić rodzaje?**

Wiesz już, że baza danych to po prostu zbiór różnych informacji zapisanych w odpowiednim formacie. W zależności od sposobu przechowywania rekordów, możemy wyróżnić kilka rodzajów baz danych.

Jednym z najbardziej popularnych jest **relacyjna baza danych**. O tym, czym charakteryzuje się taki typ baz, przekonasz się dokładnie w dalszej części tego szkolenia.

Na ten moment jednak bądź świadomy, że wyróżniamy wiele implementacji wspomnianego rodzaju baz. Kilka najczęściej używanych implementacji możesz znaleźć poniżej:

* PostgreSQL,
* MySQL,
* SQL Server,
* SQLite,
* Oracle

Dodatkowo oprócz relacyjnych baz, możemy wyróżnić również **nierelacyjne** (identyfikowane często skrótem noSQL).

Co ciekawe, koncepcja nierelacyjnych baz danych nie jest nowa, a wykorzystanie nosql-owych repozytoriów rozpoczęło się w czasach pierwszych komputerów.

A to dlatego, że mechanizm NoSQL jest często postrzegany jako szybszy i bardziej elastyczny niż SQL. Nie jest jednak tak, że jeden rodzaj jest lepszy od drugiego i powinniśmy ograniczać się tylko do jednego rozwiązania.

Wybór konkretnej bazy danych pod względem formy przechowywania danych zależy od wielu czynników, w tym problemu, który musi ona rozwiązać.

My jednak w szkoleniu tym skupimy się na relacyjnych bazach i PostgreSQL. Choć, jak już się przekonałeś, możemy wyróżnić wiele różnych implementacji tego typu bazy.

Jednak wiele z nich jest do siebie bardzo zbliżona i sprawnemu deweloperowi wystarczy tak naprawdę opanowanie SQL, a przywyknięcie do pracy w różnych środowiskach wykorzystujących ten język, będzie czystą formalnością.

**Zasada działania relacyjnych baz danych**

Bazy danych używane na produkcji składają się z kilku komponentów. W uproszczeniu - zawsze występują dwa elementy: **klient bazy danych** i **serwer bazy danych**.

**Klient** to po prostu aplikacja, która wysyła określone żądania do serwera w celu przeprowadzenia operacji.

**Serwer** bazy danych to natomiast program, który potrafi obsłużyć żądania wysyłane przez klientów. Odpowiedzialny jest on za zapisywanie i udostępnianie danych przechowywanych w bazie.

Dodatkowo, serwer używa pewnego portu, na którym nasłuchuje żądań od klientów. PostgreSQL domyślnie używa portu **5432**. Port oczywiście możemy dowolnie zmieniać i dostosowywać do swoich wymagań, musimy jednak pamiętać, aby był to port z przedziału, tzw. **registered ports** (czyli zakres od 1024 to 49151) i aby nie był zajęty przez jakąś inną aplikację.

| O rodzajach portów możesz przeczytać więcej tutaj:  <https://en.wikipedia.org/wiki/Registered_port> |
| --- |

Serwery mogą używać zarówno protokołu **ODBC** (Open Database Connectivity), różnych adapterów jak i protokołu **JDBC** (Java Database Connectivity) do komunikacji z klientem. Konkretna implementacja połączenia będzie na pewno zależała od tego, jak to jest realizowane w kodzie.

Przykładowy adres URI, który umożliwiać będzie połączenie z bazą wygląda następująco:

jdbc:postgresql://host:port/database

lub przy wykorzystaniu dowolnego adaptera (np. psycopg2):

postgresql://username:password@host/database

Gdybyśmy pochylili się już nad samym sposobem komunikacji - klient wysyła żądania do bazy danych. Żądania te nazywane są inaczej **zapytaniami**, które realizują określone operacje w bazie.

Operacje te mogą zarówno modyfikować, aktualizować i czytać stan danych w bazie. Zanim jednak przejdziemy do zapoznania się z konkretnymi typami zapytań, przeanalizujmy, **jak wygląda struktura bazy**.

**Struktura relacyjnej bazy danych**

**Ogólny obraz**

Zaczynając od najwyższej warstwy grupowania danych - każda baza podzielona jest na **tabele (**nazywane też **encjami)** reprezentujące poszczególne **grupy informacji**.

W obrębie takich struktur mogą istnieć **rekordy (**nazywane też **krotkami)** i **powiązania (relacje)** do innych tabel. Właśnie ze względu na obecność takich relacji, mówimy, że bazy, w obrębie których one się znajdują, są **relacyjne**.

Jak się niedługo sam przekonasz, okazuje się być to bardzo prosty i intuicyjny sposób przedstawiania danych.

Rekordy natomiast (czyli konkretne informacje zawarte w tabeli) składają się z **atrybutów**, które reprezentują pojedyncze cechy rekordu i mają przypisany określony typ danych.

Aby lepiej zobrazować omówione wyżej pojęcia, możesz wyobrazić sobie bazę danych dowolnej firmy szkoleniowej. Firma ta przechowuje informacje o dostępnych ścieżkach nauki, mentorach, klientach i materiałach szkoleniowych.

Każda z tych informacji będzie grupowana w **tabelach**, w której znajdować będą się różne rekordy reprezentowane przez **atrybuty**. Poniżej, np. tabela Mentors z dwoma przykładowymi rekordami.

Tabela Mentors:

| id | name | surname | specialization |
| --- | --- | --- | --- |
| 1 | Jan | Kowalski | Python |
| 2 | Kacper | Nowak | C++ |

Dokładnie zrozumienie, jak wygląda struktury bazy wymaga jednak od nas jeszcze głębszego omówienia, dlatego zatrzymajmy się teraz przez chwilę przy każdym z wyżej wymienionych pojęć (tabela, rekord, atrybut).

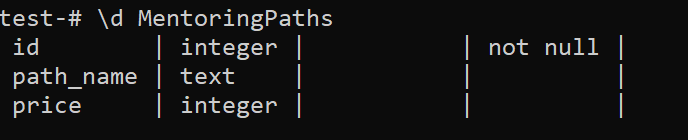
**Tabele (encje), rekordy (krotki) i atrybuty**

Zacznijmy od omówienia encji. Tabele, jak już wiesz służą do przechowywania określonych grup danych. Odpowiednikiem encji w **programowaniu obiektowym jest klasa**. W obrębie takiej struktury znajdują się **rekordy (krotki)**.

Krotki są ściśle związane z tabelami. Krotka to po prostu przechowywany obiekt w tabeli. Często są one reprezentowane pojedynczy wiersz informacji. Każdy taki wiersz posiada określone **atrybuty**.

Wcześniej prezentowałem Ci, jak może wyglądać przykładowa tabela z mentorami, teraz wyobraźmy sobie potrzebę przedstawienia zbioru wszystkich dostępnych ścieżek nauki.

W celu przechowywania informacji o takich gałęziach szkolenia, utworzymy oddzielną tabelę **MenntoringPaths**. Pojedyncza ścieżka mentoringu reprezentowana będzie przez krotkę opisywaną przez atrybuty jak: **id (który jest unikalny)**, **path\_name**, **price**.



Widok na tabelę z poziomu psql (o którym więcej niebawem)

Każdy z atrybutów jasno opisuje przechowywany w tabeli obiekt. Typ atrybutów określamy podczas definiowania tabeli - może to być zarówno *typ tekstowy*, *liczbowy*, *logiczny*, *daty* itd.

Ponadto wybrane atrybuty mogą być **unikalne** lub zachowywać się jak **klucz główny/obcy**. Rozwińmy poniżej, jak rozumieć wymienione rodzaje kluczy.

**Klucz główny**

Okazuje się, że klucz główny (primary key, PK) jest pojęciem ściśle związanym z tworzeniem relacji między tabelami oraz jednoznacznym identyfikowaniem rekordów w encji.

W uprzednio stworzonej tabeli MentoringPaths, kluczem głównym była kolumna - id. To dzięki niej możemy jednoznacznie odwoływać się do określonych rekordów z tabeli. Informacja ta jest niepowtarzalna w obrębie tabeli i jasno identyfikuje ścieżki mentoringu (każda ze ścieżek ma przydzielony odrębny i unikalny id).

Jednak przede wszystkim klucze główne umożliwiają nam tworzenie relacji między poszczególnymi tabelami.

W skrócie - chcąc powiązać między sobą dane zawarte w tabeli, np. MentoringPaths z odpowiednimi informacjami z encji - Mentors (tak, aby przedstawiać, za jaką ścieżkę odpowiada dany nauczyciel), użyjemy właśnie **klucza głównego** i **klucza obcego**. Więcej na ten temat za chwilę.

**Klucz obcy**

**Klucze obce (foreign keys, FKs)** to ważne pojęcie pojawiające się przy tworzeniu zależności między tabelami. Chcąc zobrazować to zagadneinie przez prostą definicję, moglibyśmy przytoczyć coś takiego:

| "Klucze obce to po prostu klucze główne danych krotek z określonej tabeli, umieszczone w **obcej tabeli**. Przykładem klucza obcego może być id ścieżki umieszczone w tabeli mentorów (w celu pokazania, który mentor jest odpowiedzialny za konkretną ścieżkę). |
| --- |

Zastosowanie PK oraz FK w praktyce mogłoby wyglądać następująco:

Tabela *Mentors*

| **id** | **name** | **surname** | **no\_students** | **path\_id (FK)** |
| --- | --- | --- | --- | --- |
| 1 | Kacper | Nowak | 20 | 1 |
| 2 | Jerzy | Sawczyn | 10 | 2 |
| 3 | Kacper | Stysło | 5 | 1 |

Tabela *MentoringPaths*

| **id** | **path\_name** | **price** |
| --- | --- | --- |
| 1 | Junior Python | 750 |
| 2 | Senior Java | 900 |
| 3 | Senior Python | 900 |

Dzięki odpowiedniej manipulacji wartościami oraz jasnego zdefiniowania PK oraz FK, możemy wyciągnąć następujące informacje z obu tabel:

* Obecnie w firmie mamy 3 mentorów o imieniu i nazwisku: Kacper Nowak, Jerzy Sawczyn oraz Kacper Stysło (tabela Mentors)
* Prowadzimy kursy z takich ścieżek rozwoju jak: Junior Python, Senior Java, Senior Pythona (tabela MentoringPaths)
* Kacper Nowak i Kacper Stysło są mentorami odpowiedzialnymi za prowadzenie ścieżki Junior Python (path\_id o wartości 1 z tabeli Mentors odnosi się do rekordu z MentoringPaths o takim samym id)
* Jerzy Sawczyn odpowiada za szkolenie ze ścieżki Senior Java
* Z naszych tabel możemy wyciągnąć również dodatkowe informacje, takie jak: mentor o id 1 ma 20 studentów, mentor z id 2 - 10 studentów, mentor o numerze 3 - 5 studentów
* Ścieżka Junior Pythona kosztuje 750 (zł), Senior Java - 900 (zł) oraz Senior Python - 900 (zł).

**Dlaczego taka struktura?**

Początkujący mogą również zapytać, dlaczego nie możemy wszystkich wymienionych informacji przechowywać w jednej tabeli. Wówczas takie antywzorcowe rozwiązanie mogłoby wyglądać następująco:

Przykładowy antywzorzec tabeli - *CourseInfo*

| **id** | **name** | **surname** | **no\_students** | **path\_name** | **price** |
| --- | --- | --- | --- | --- | --- |
| 1 | Kacper | Nowak | 20 | Junior Python | 750 |
| 2 | Jerzy | Sawczyn | 10 | Senior Java | 900 |
| 3 | Kacper | Stysło | 5 | Junior Python | 900 |

Omawiając taki przykład, chcę Cię przestrzec przed projektowaniem tabeli w taki sposób.

Istnieje kilka uzasadnień, dlaczego powinniśmy decydować się na odpowiednie grupowanie informacji w tabelach:

* Przez brak rozbicia informacji na dwie tabele (np. Mentors oraz MentoringPaths), powielamy dużą część informacji w tej samej tabeli (zamiast po prostu umieszczać FK w postaci id, powielamy długą nazwę ścieżki).
* Ponadto, gdybyśmy potrzebowali umieścić większą ilość atrybutów, tabela CourseInfo byłaby mocno przeładowana.

Widzisz zatem, jak ważne jest przemyślane zaprojektowanie rozkładu danych w tabelach połączonych relacjami.

Okazuje się, że możemy wyróżnić kilka **rodzajów relacji**. To, z jakiej relacji skorzystamy, będzie mocno zależało od problemu projektowego, z którym będziemy się borykali.

Poniżej omówienie dostępnych relacji i wyszczególnienie różnic pomiędzy nimi.

**Rodzaje powiązań**

* **Jeden do jednego (1:1)**

To rodzaj powiązania występujący przy połączeniu dwóch tabel, w których jeden rekord z pierwszego tabeli odnosi się **tylko i wyłącznie** do pojedynczego rekordu z drugiej.

Przykładowo taka relacja może istnieć między tabelą Students (przechowującą dane o wszystkich kursantach uczestniczących w szkoleniu) a tabelą Worksheet (reprezentującą indywidualny plan pracy każdego ze studentów).

Taka relacja sprawdzi się w tym przypadku, ponieważ z jednym studentem będziemy mogli identyfikować tylko i wyłącznie jedną indywidualną rozpiskę pracy oraz odwrotnie - do jednego planu pracy przyporządkujemy pojedynczego studenta.

* **Jeden do wielu (1:n)**

W takiej relacji jeden rekord z danej tabeli może być powiązany z wieloma różnymi rekordami z drugiej tabeli. Ponownie przytoczę tutaj jako przykład tabelę Students oraz tabelę Mentors. Zauważ, że jeden mentor może nauczać wielu różnych studentów. Stąd relacja jeden do wielu sprawdzi się tutaj idealnie.

* **Relacja wiele do wielu (n:m)**

Ostatnim przykładem relacji n:m może być przypadek, w którym łączymy rekordy tabeli TrainingMaterials (reprezentującej materiały szkoleniowe w ramach kursu) z tabelą Students (którzy z takich materiałów korzystają).

Musimy tutaj wprowadzić relację wiele do wielu, bowiem kilku studentów może korzystać z kilku tych samych materiałów szkoleniowych. Przykładowo Jan Kowalski jest powiązany z dwoma rekordami **Docker** oraz **Python** z tabeli Trainings oraz drugi uczeń - Anna Nowak - uczy się z takich szkoleń jak **JavaScript** oraz **Docker** (tak samo jak Jan Kowalski).

W praktyce taką relację tworzy się przy użyciu kluczy głównych i obcych (to już na pewno wiesz z poprzedniej części szkolenia), ale również przez wprowadzenia **tabeli pośredniczącej** (tzw. proxy).

Przykład relacji wiele do wielu między uproszczonymi encjami Students oraz Trainings mógłby wyglądać następująco:

Tabela *Students*

| **id** | **name** | **surname** |
| --- | --- | --- |
| 1 | Jan | Kowalski |
| 2 | Anna | Nowak |

Tabela *Trainings*

| **id** | **name** | **description** |
| --- | --- | --- |
| 1 | Docker | Docker is an open platform for developing, shipping, and running applications. |
| 2 | Python | Python is an interpreted, object-oriented, high-level programming language with dynamic semantics |
| 3 | JavaScript | JavaScript is a high-level, often just-in-time compiled language that conforms to the ECMAScript standard. |

Tabela *TrainingsStudents* (tabela proxy; umożliwia wprowadzenie relacji wiele do wielu):

| **id** | **training\_id (fk)** | **student\_id (fk)** |
| --- | --- | --- |
| 1 | 1 | 1 |
| 2 | 2 | 1 |
| 3 | 3 | 2 |
| 4 | 1 | 2 |

**Zadanie:**

Odczytaj, z tabeli Trainings Students, jakie materiały przyporządkowane są do danych uczniów. W razie konieczności przeanalizuj tabelę wspólnie z mentorem.

**Praca z diagramami baz danych**

Niejednokrotnie podczas pracy z bazami danych będziesz potrzebował zwizualizować encje wraz z relacjami pomiędzy nimi. Projektowanie takich schematów jest bardzo istotne z punktu widzenia biznesowego. Najczęściej takie diagramy sprawdzają się, gdy musimy przeanalizować i wspólnie przemyśleć strukturę bazy lub zaprezentować biznesowi koncept tworzonego systemu.

Jednym z prostszych diagramów prezentujących encje i powiązania między nimi w obrębie naszej bazy jest, tzw. **ERD** (Entity Relationship Diagram).

Diagramy ERD możemy przedstawiać w poniższy sposób:

* **notacja UML**
* diagramy C4
* notacja Bachmana
* notacja Martina
* notacja Chen
* Crow's Foot

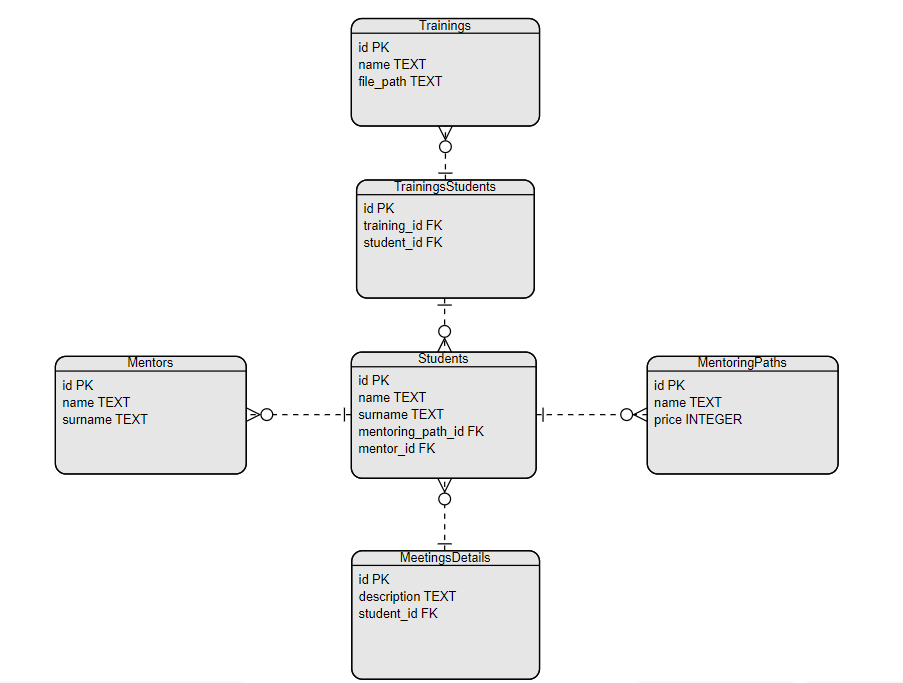
Wyszczególniłem pojedynczą pozycję powyżej, ponieważ jest to jeden z najczęściej wykorzystywanych sposobów na prezentację baz i nie tylko (UML sprawdza się również w modelowaniu klas).

Tego typu diagramy to po prostu graficzne odzwierciedlenie, np. tabel, które będą implementowane w programie. Istnieją jednak pewne reguły, których należy się trzymać, tworząc diagramy. Zasady te gromadzone i określane są właśnie przez **UML** (Unified Modeling Language).

Zalecam zatem zapoznanie się ze wspomnianym tematem i wykorzystanie takich źródeł wiedzy jak:

* [Podstawy UML](https://www.samouczekprogramisty.pl/podstawy-uml/)
* [Welcome To UML Web Site!](https://www.uml.org/)
* [Unified Modeling Language – Wikipedia, wolna encyklopedia](https://pl.wikipedia.org/wiki/Unified_Modeling_Language)

Poniżej zaprezentuję Ci przykładowy schemat UML, który odzwierciedlać będzie bazę danych wewnątrz firmy Devs-Mentoring.pl.



ER diagram

**Zadanie**

Przeanalizuj typy relacji między poszczególnymi encjami.

* Wskaż relacje n:m, 1:1, 1:n
* Zastanów się, jakie ma znaczenie fakt, że klucz obcy reprezentujący relację między tabelami X i Y, umieszczamy w tabeli X, a nie Y? Rozważając tabele Students - Mentors, czy miałoby sens zrekonstruowanie relacji przez umieszczenie FK - student\_id w Mentors zamiast *mentor\_id* w Students?

| **Uwaga!**  Diagramowanie UML jest faktycznie często dobrym rozwiązaniem na zaprojektowanie struktury bazy danych, jednak wiąże się z nim wiele reguł i sztywnych wytycznych, do których musi stosować się projektant. Choćby tak prosta relacja jak jeden do wielu, jeden do jednego czy wiele do wielu - może sprawiać więcej kłopotów, ponieważ reprezentowana jest przez różne rodzaje linii. |
| --- |

Wspominam Ci o tym, ponieważ chciałbym, żebyś wyciągnął z tego ważny wniosek - UML jako język modelowania nie jest najłatwiejszym rozwiązaniem do przedstawiania różnych zależności.

Oczywiście, sprawdzi się on w przypadku tworzenia prostych bazodanowych modeli ER. Problem jednak może pojawić się, gdy potrzebujemy stworzyć złożone schematy bazujące na różnych i skomplikowanych zależnościach.

Wówczas, aby stworzyć diagram zgodny ze wszelkimi założeniami UML, musimy zagłębić się we wszelkie reguły i rodzaje elementów, które ten język udostępnia. Dlatego nierzadko może okazać się, że mniej czasochłonnym rozwiązaniem będzie użycie innego sposobu modelowania encji, jak np. **diagramy C4**.

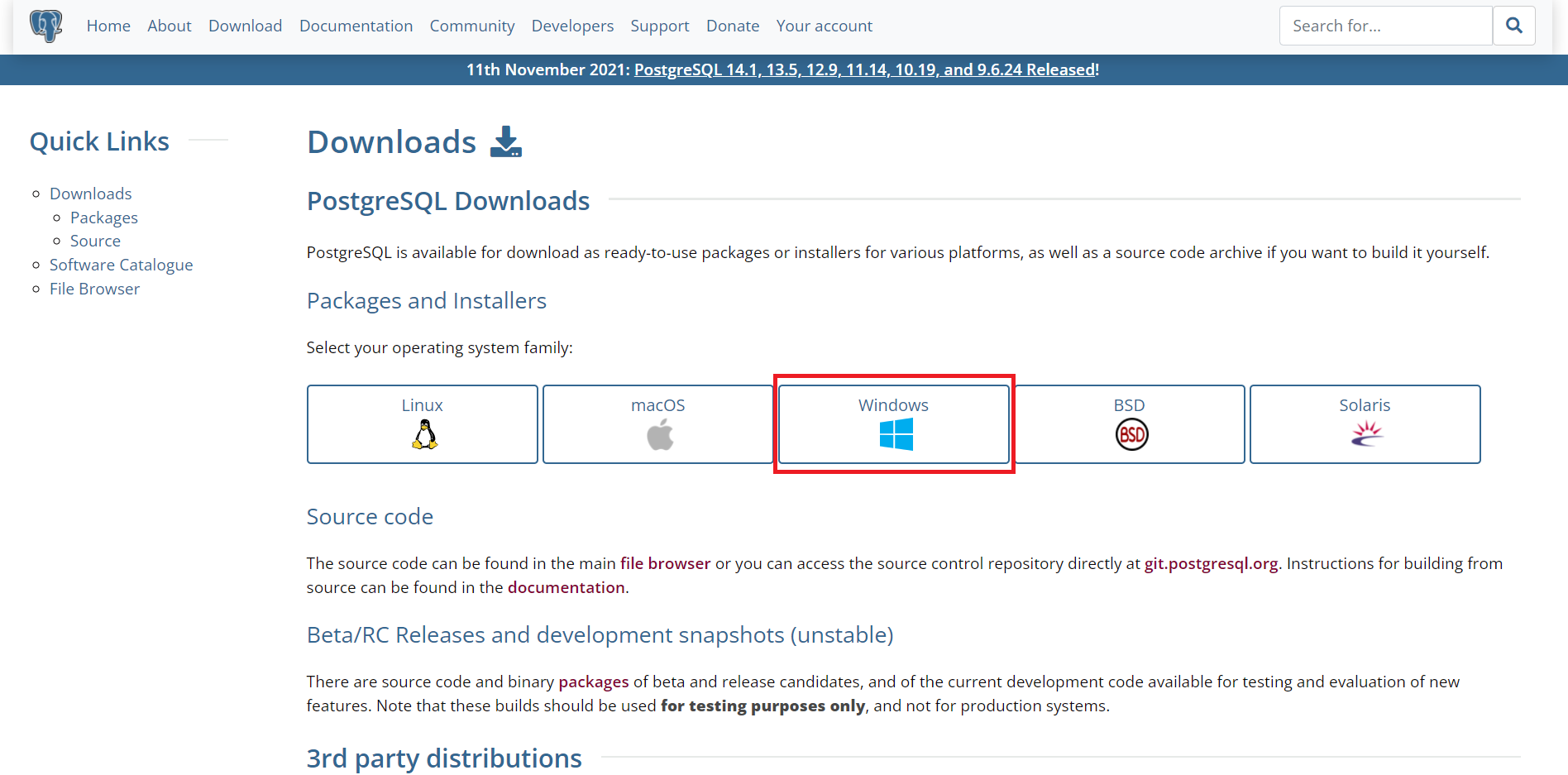
**Instalacja PostgreSQL**

To tyle, jeśli chodzi o podstawową wiedzę na temat baz danych, encji, atrybutów i relacji. Przejdziemy teraz do części związanej z konkretnie już instalacją i konfiguracją klienta i serwera PostgreSQL.

Po utworzeniu bazy i połączeniu się z nią, przedstawię Ci pierwsze polecenia SQL, które umożliwią skuteczne zarządzanie danymi.

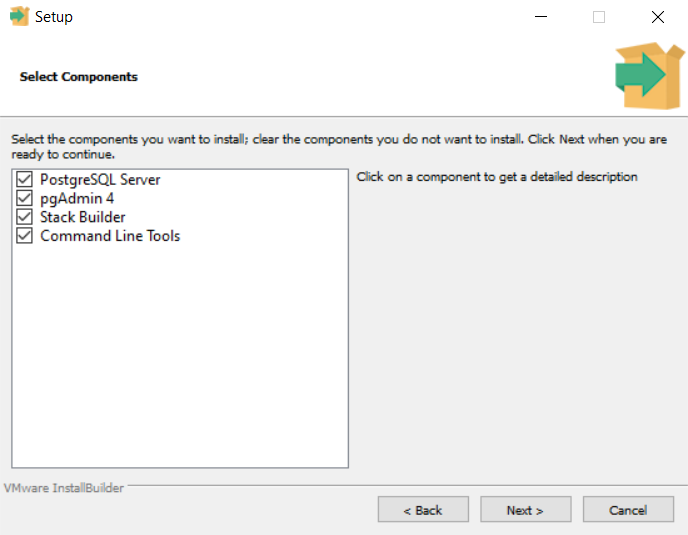
W pierwszej kolejności konieczne będzie jednak pobranie instalatora dla omawianej implementacji relacyjnej bazy danych. Instrukcję instalacji znajdziesz tutaj: [PostgreSQL: Downloads](https://www.postgresql.org/download/).

Ja zaprezentuję Ci proces instalacji dla systemu Windows. Na dzień tworzenia szkolenia, wersja 14 jest najbardziej aktualnym wydaniem PostgreSQL.



Wybór instalatora dla systemu Windows

Podczas instalacji natrafisz na etap, w którym będziesz musiał wybrać komponenty, które chcesz doinstalować w ramach PostgreSQL.

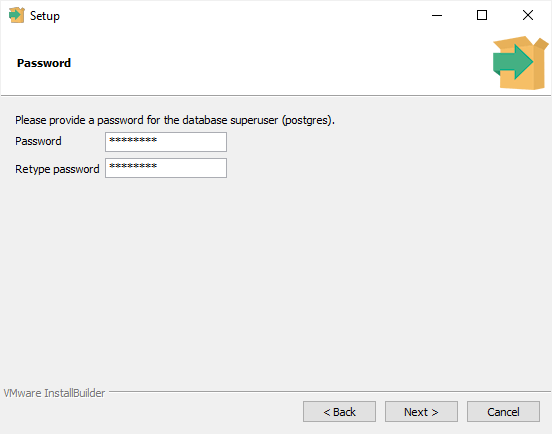


Komponenty

Polecam Ci wybrać każdy z dostępnych komponentów. Dzięki temu nie będziemy ograniczeni podczas zarządzania bazami. Poniżej krótki opis każdego z komponentów:

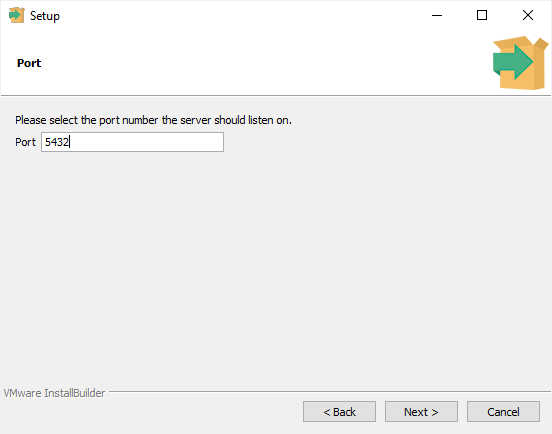
* **PostgreSQL Server** - serwer Postgre, który jest nieodłącznym elementem modelu klient-serwer, na którym się opiera omawiany rodzaj implementacji relacyjnych baz danych.
* **pgAdmin 4** - to graficzny interfejs umożliwiający zarządzanie bazami danych i serwerami PostgreSQL.
* **Stack Builder** - wykorzystywane do pobierania i instalacji dodatkowych narzędzi, sterowników i aplikacji uzupełniających instalację PostgreSQL
* **Command Line Tools** - zaznaczenie tej opcji spowoduje zainstalowanie narzędzi do wiersza poleceń i bibliotek klienckich takich jak: *libpq*, *ecpq*, *pg\_dump*, *pg\_restore*.

W następnych krokach instalacji kluczowe będzie podanie hasła dla superusera PostgreSQL. Główny użytkownik omawianej implementacji bazy ma nazwę **postgres** i to właśnie z niego będziemy korzystali w tym szkoleniu. Istotne jest, abyś zapamiętał wpisywane hasło, ponieważ niejednokrotnie będziesz proszony o jego podanie podczas pracy z danymi.



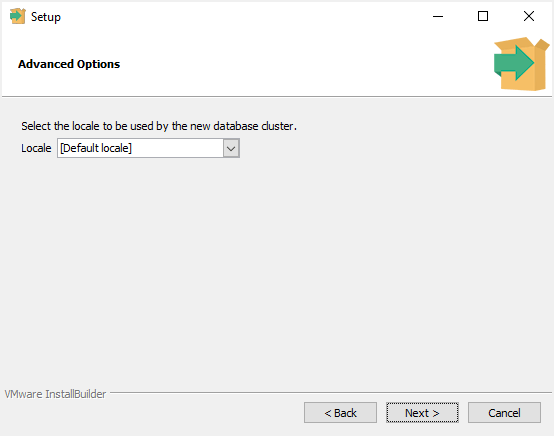
Ustalenie hasła dla postgres

Następnie określ port, na którym ma być postawiony serwer PostgreSQL. Domyślnie jest to wartość równa 5432.



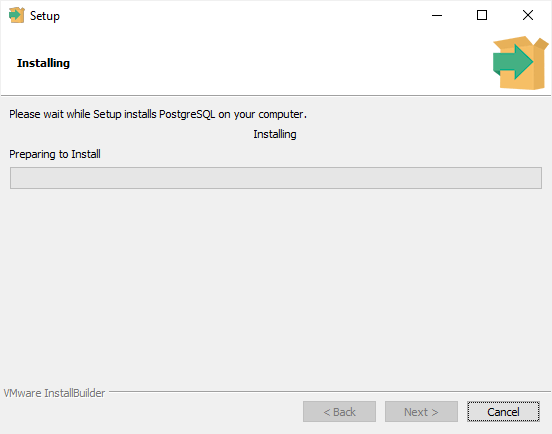
Określenie portu

Finalnie - jednym z ostatnich kroków będzie określenie **regionalizacji (locale)**. Pozostaw tę wartość jako domyślną. Wówczas sposób kodowania znaków w bazie będzie zgodny z tym, jaki wykorzystuje system operacyjny, na którym będzie zainstalowany PostgreSQL.

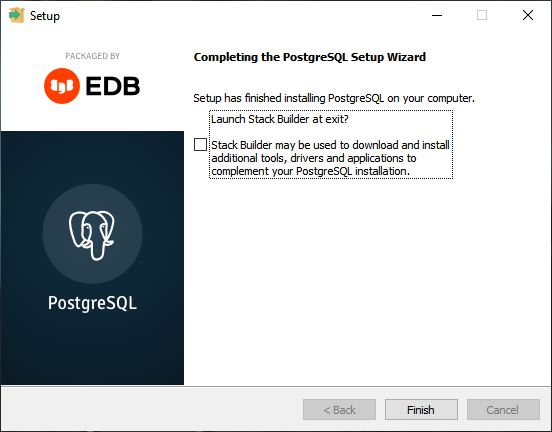


Określenie regionalizacji

Przeklikaj jeszcze pozostałe opcje instalacji, aż PostgreSQL zostanie zainstalowany na Twojej maszynie.



Na końcu instalacji zostaniesz zapytany, czy chcesz uruchomić i ewentualnie pobrać dodatkowe rozszerzenia ze Stack Buildera. Odznacz tę opcję (na ten moment nie będziemy potrzebowali żadnych dodatkowych zależności).

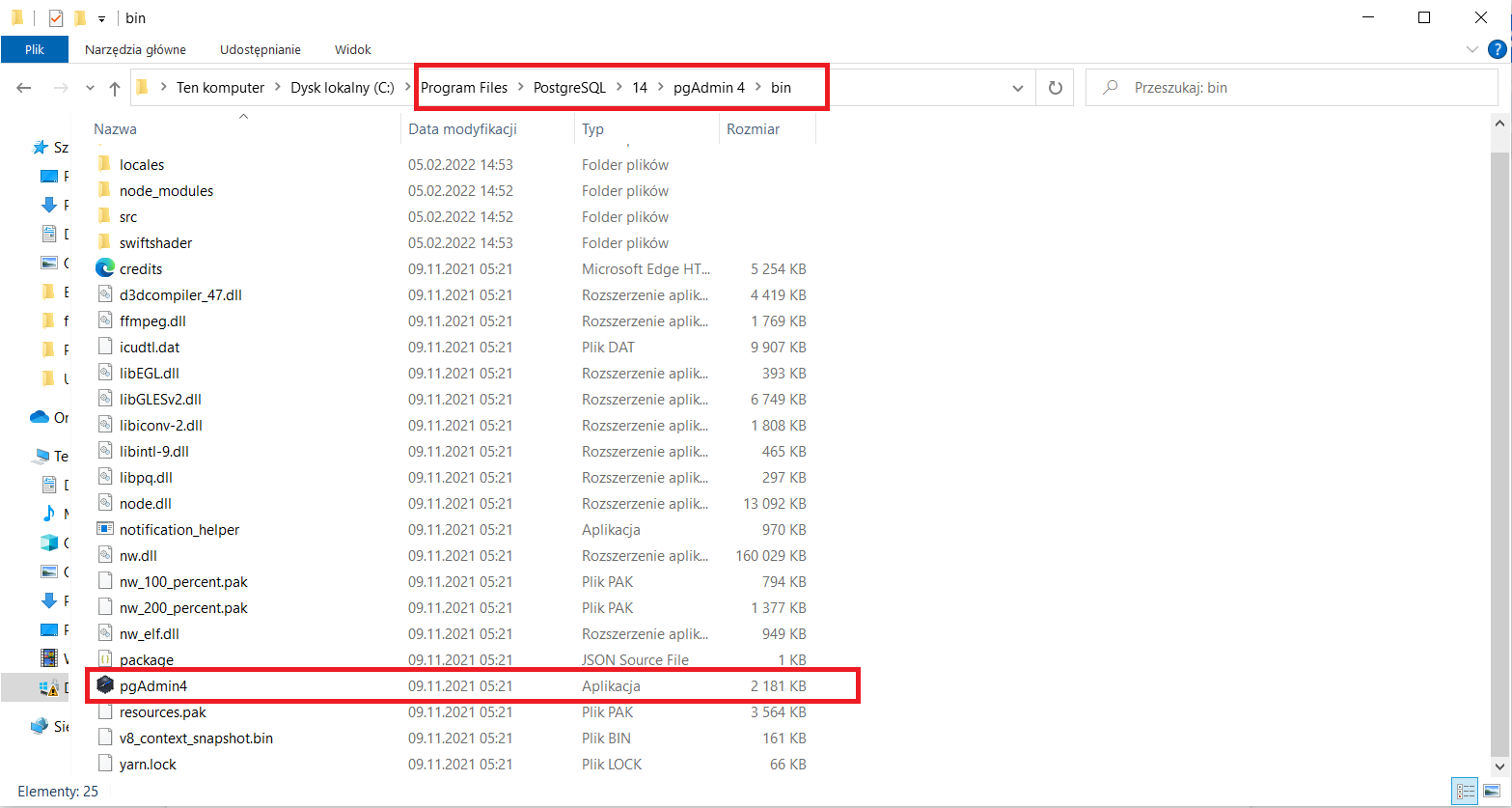


**Obsługa klienta bazy danych - CLI i GUI**

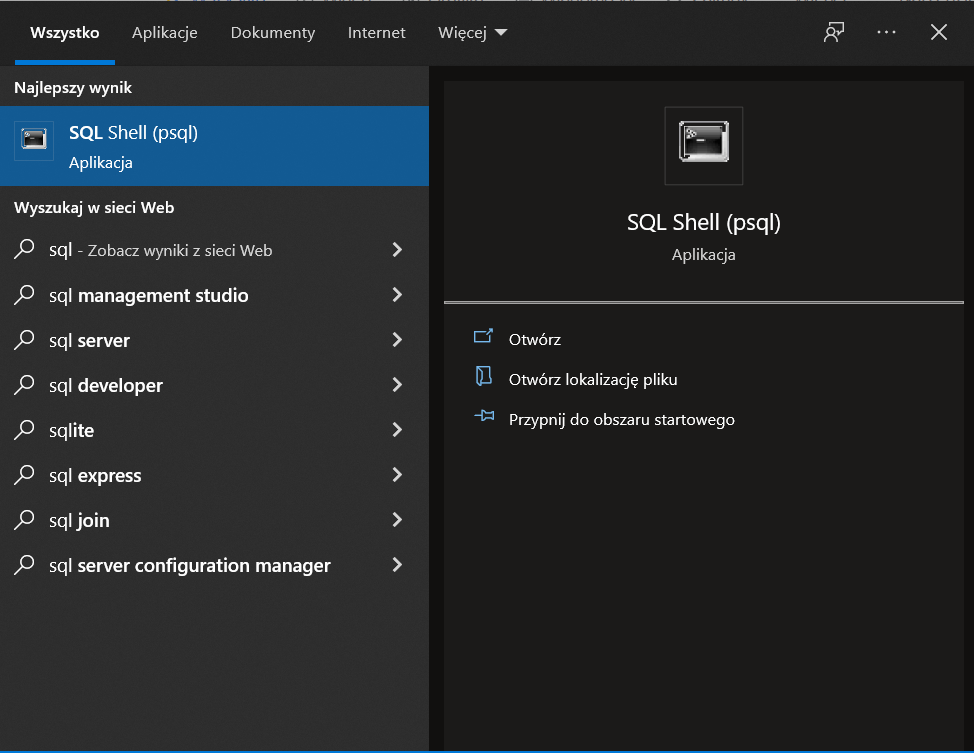
Po udanym procesie instalacji, powinieneś mieć teraz dostęp do dwóch kluczowych aplikacji.

Interfejsu umożliwiającego zarządzanie bazą przez **GUI** (*pgAdmin 4*) oraz do **terminala CLI** (*psql*).

PgAdmina możesz znaleźć w poniższym katalogu:



… a SQL Shell (psql) tutaj (wyszukując program w systemie):



Uświadamiam Cię już na samym początku - oba programy służą do tego samego - umożliwią efektywne zarządzanie zasobami PostgreSQL (np. tworzenie i modyfikowanie baz, tabel). Pełnią one rolę klienta, który komunikuje się z serwerem, zlecając wykonanie odpowiednich operacji.

Tak więc rolę klienta w naszym modelu komunikacji z serwerem może pełnić:

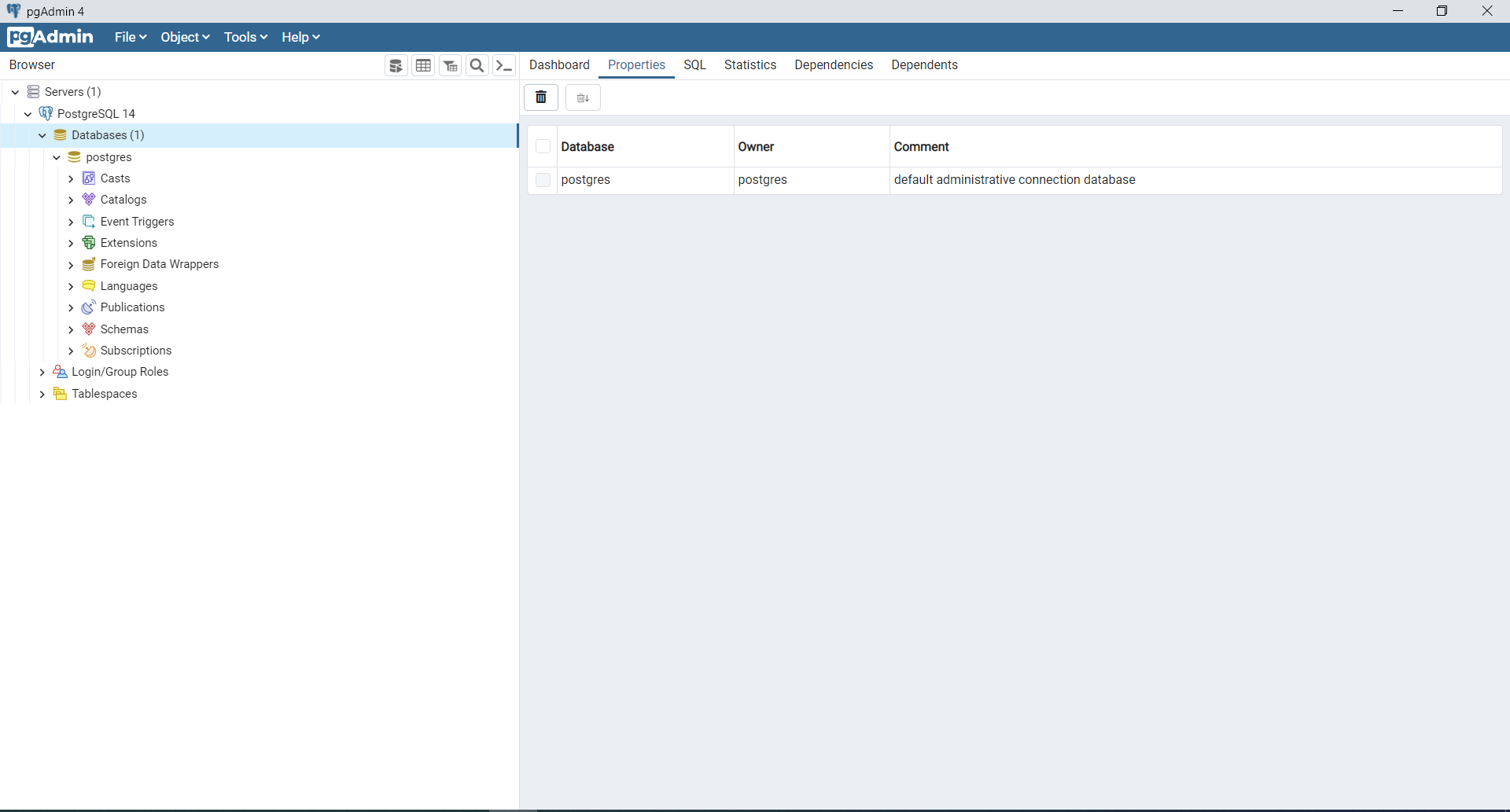
* Klient GUI
* Terminal/CMD/psql
* Aplikacja, np. **DataGrip** lub zaprojektowana przez nas program Springowy łączący się z bazą danych.

Jednym słowem nie jesteśmy ograniczeni do jednego konkretnego sposobu zarządzania bazą - możemy implementować wiele różnych interfejsów.

**PgAdmin i psql**

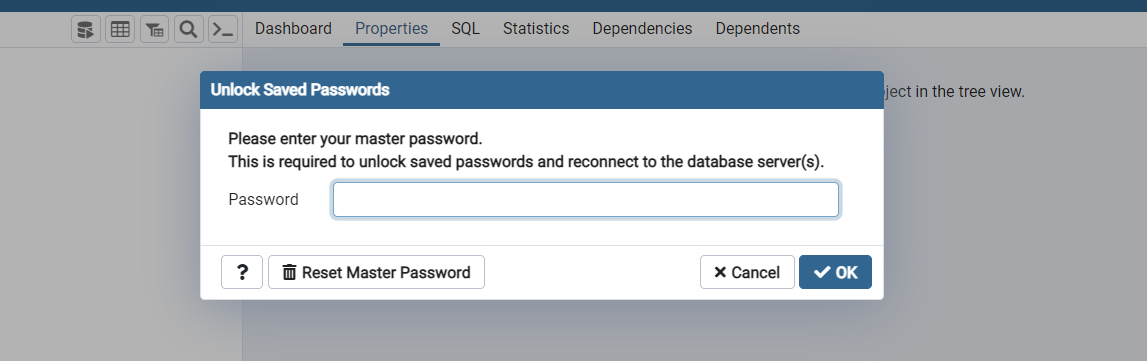
Wróćmy do aplikacji, które zostały domyślnie zainstalowane wraz z PostgreSQL - **pgAdmin 4** oraz **psql**. Przyjrzyjmy się w pierwszej kolejności pgAdminowi - połączenia z bazą realizowanego przez GUI:

Po uruchomieniu aplikacji ujrzymy następujący interfejs:



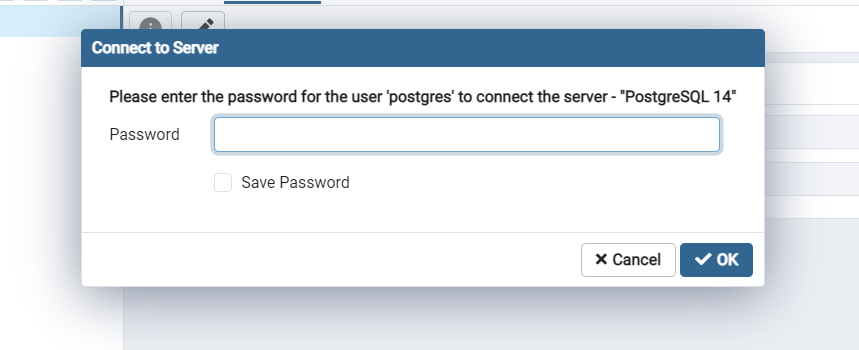
Widok na pgAdmin 4

(Zostaniesz poproszony o zalogowanie się do interfejsu pgAdmina przez podanie **master password**):



Wykorzystaj wówczas hasło, które podałeś podczas wcześniejszego procesu instalacji i jest ono takie samo jak to, które podałeś dla użytkownika postgres.

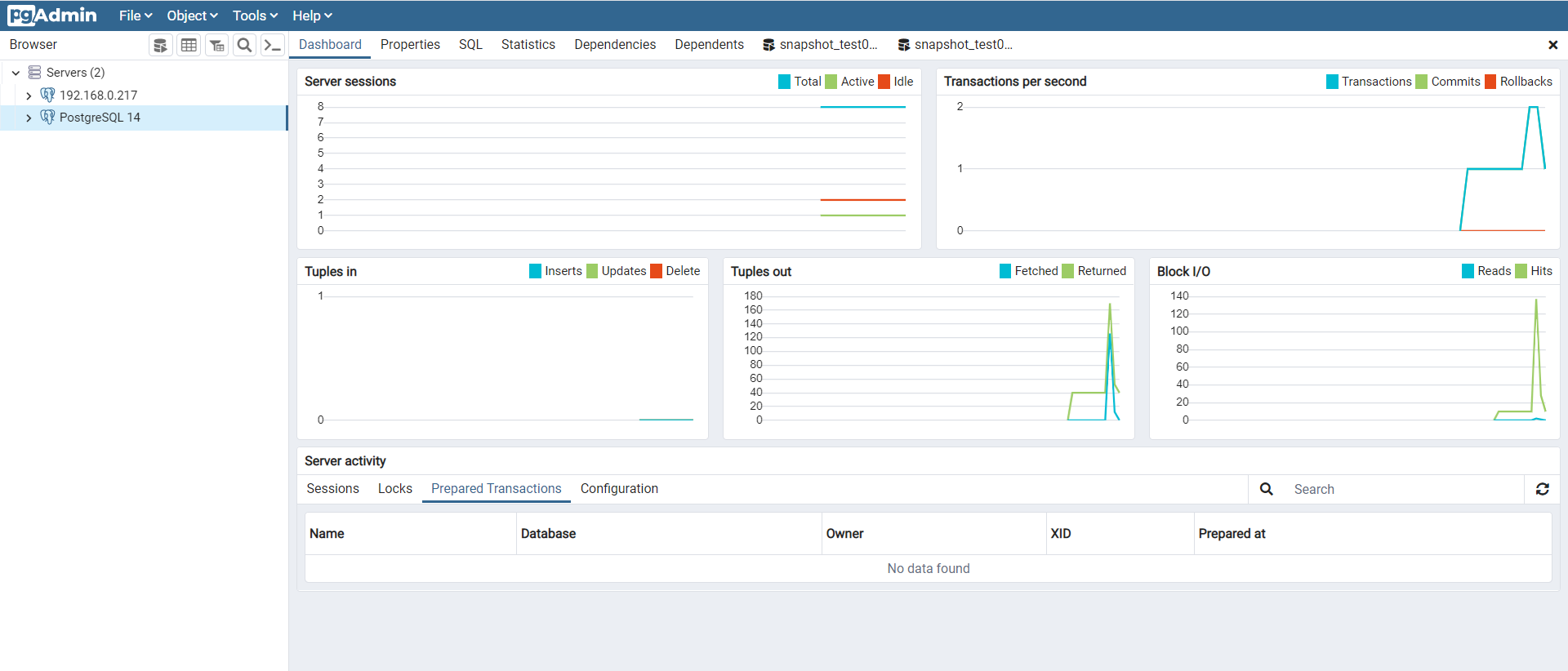
Gdy podasz poprawne hasło, to następnie zostaniesz poproszony o odpowiednią autoryzację przy logowaniu się do serwera już jako właściwy, automatycznie utworzony, użytkownik **postgres**:



Domyślnym hasłem jest **postgres** (lub zgodne z tym, które umieściłeś podczas instalacji).

Wspomniany użytkownik to defaultowo tworzony administrator, którego uprawnieniami będzie zarządzanie bazą. W praktyce będziemy odpowiednio dywersyfikowali uprawnienia między różnych użytkowników klienta, jednak z **postgres** korzystać będziemy przez dalszą część szkolenia, poznając kolejne możliwości bazy.

Po tym, jak zalogujesz się do uruchomionego klienta, zobaczysz, iż GUI składa się z rozwijalnej listy (na której możemy przeglądać dostępne bazy, schematy etc.) oraz sekcji informacyjnej, z której możemy odczytywać informacje o tabelach czy wykresy prezentujące statystyki związane z połączeniem do naszego serwera.



Widok na aplikację pgAdmin

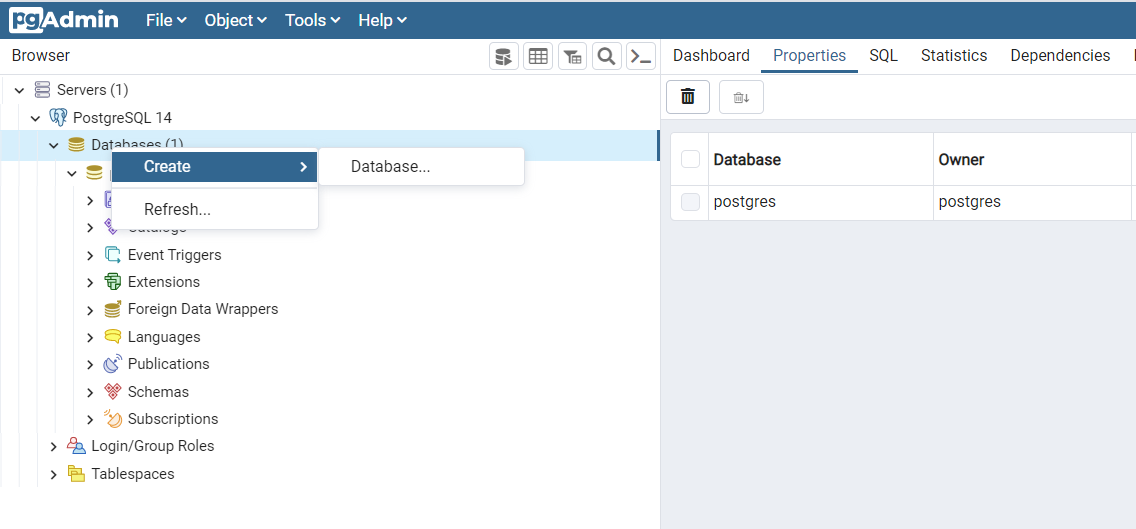
Zauważ również, że na ten moment mamy domyślnie utworzoną bazę **postgres**. Jest to standardowa baza tworzona przy instalacji PostgreSQL.

Jednym słowem - pgAdmin umożliwia intuicyjne zarządzanie bazą “przez przeklikiwanie się”. Nie ma tu żadnej filozofii, wystarczy znać jedynie bazowe koncepty i smaczki na poziomie implementacji Postgre, które będziemy progresywnie poznawali.

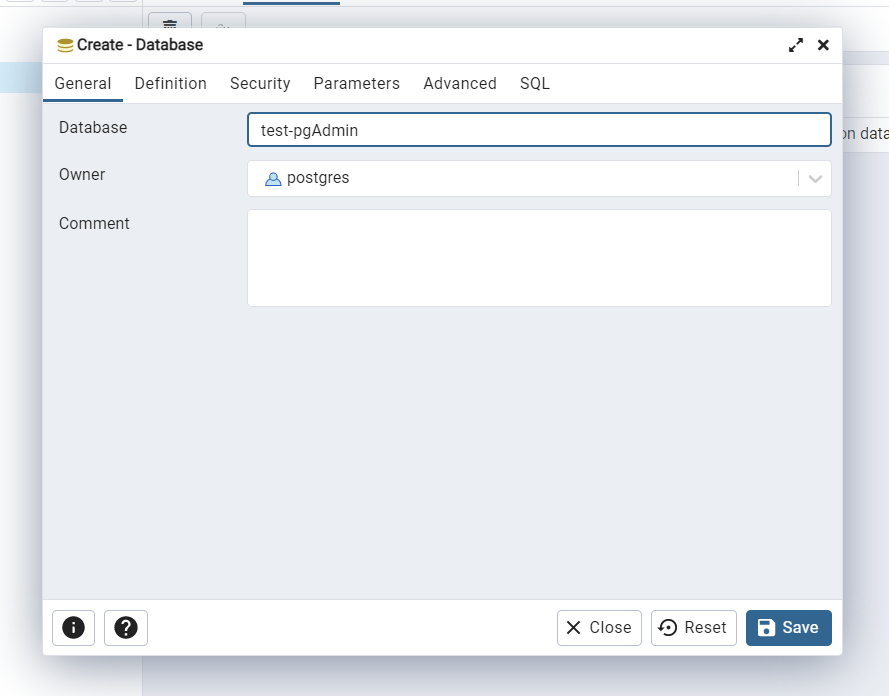
**Utworzenie bazy z pgAdmin**

Abym mógł zaprezentować Ci możliwości, jakie dostarcza PostgreSQL, musimy najpierw stworzyć bazę danych. Nazwiemy ją *“test-pgAdmin”* oraz wykonamy to z poziomu GUI.

1. W tym celu kliknij prawym przyciskiem myszy na zakładkę **Databases** i wybierz opcję **Create 🠆 Database**.

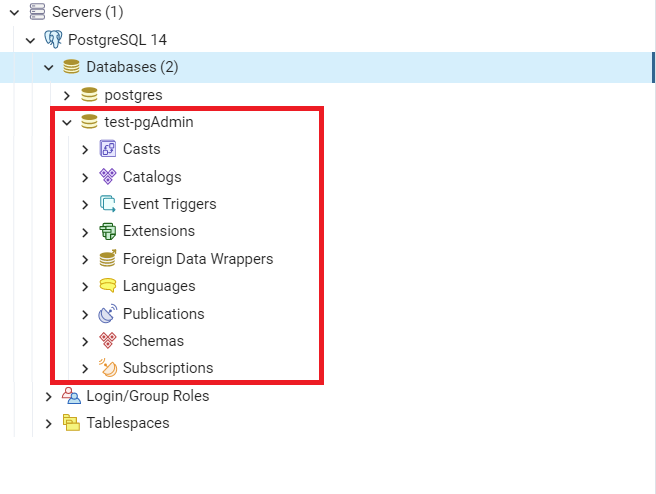


Utworzenie bazy danych z poziomu pgAdmin



Nadanie nazwy bazie danych

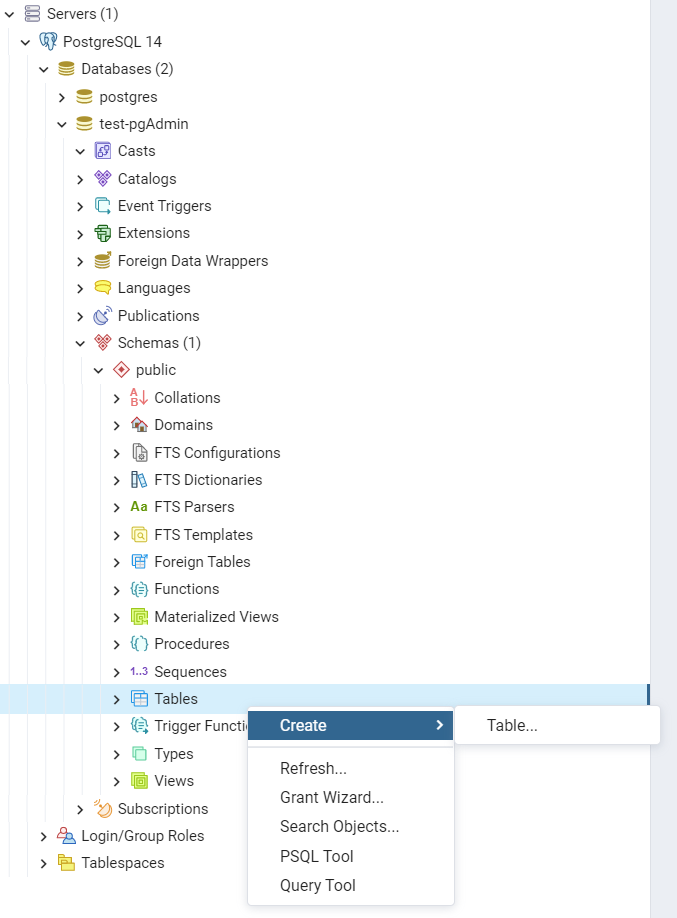
W rezultacie Twoja baza powinna widnieć na liście dostępnych zasobów PostgreSQL.

****

Widok na utworzoną bazę danych

**Utworzenie tabeli**

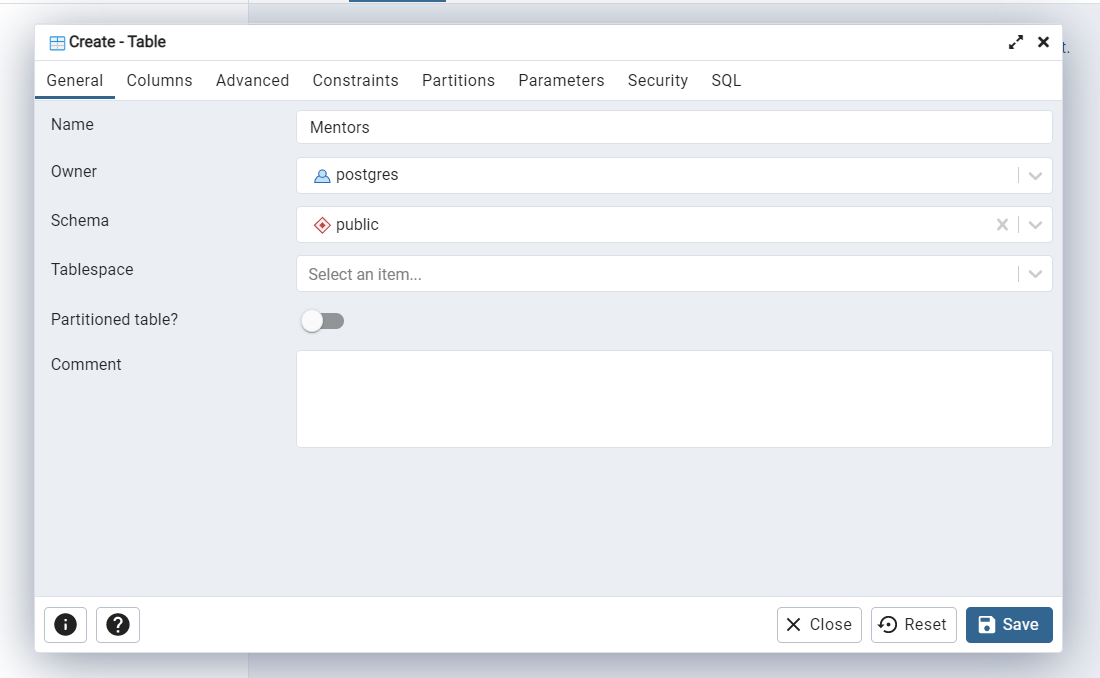
Chcąc teraz utworzyć tabelę w obrębie naszej bazy, przejdziemy do zakładki **test-pgAdmin** 🠆 **Schemas** 🠆 **Tables** 🠆 **Create Table**.



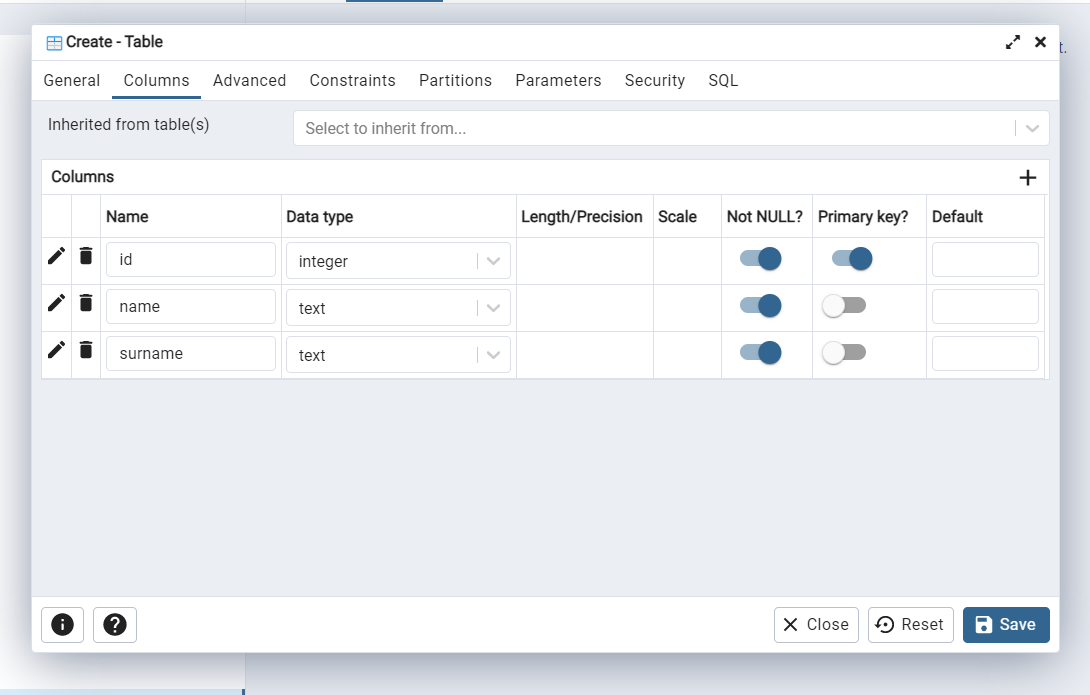
Zakłada Create Table

Następnie wypełniamy pop-up odpowiednimi informacjami dotyczącymi tabeli. My utworzymy uproszczoną tabelę Mentors składającą się z takich kolumn jak: *id*, *name*, *surname* (na ten moment nie będzie powiązywali ją żadną relacją z inną tabelą; ta sekcja ma jedynie zaprezentować Ci podstawowe operacje na interfejsie bazodanowym).

Następnie pozostaje odpowiednio nazwać naszą tabelę i dodać do niej atrybuty.



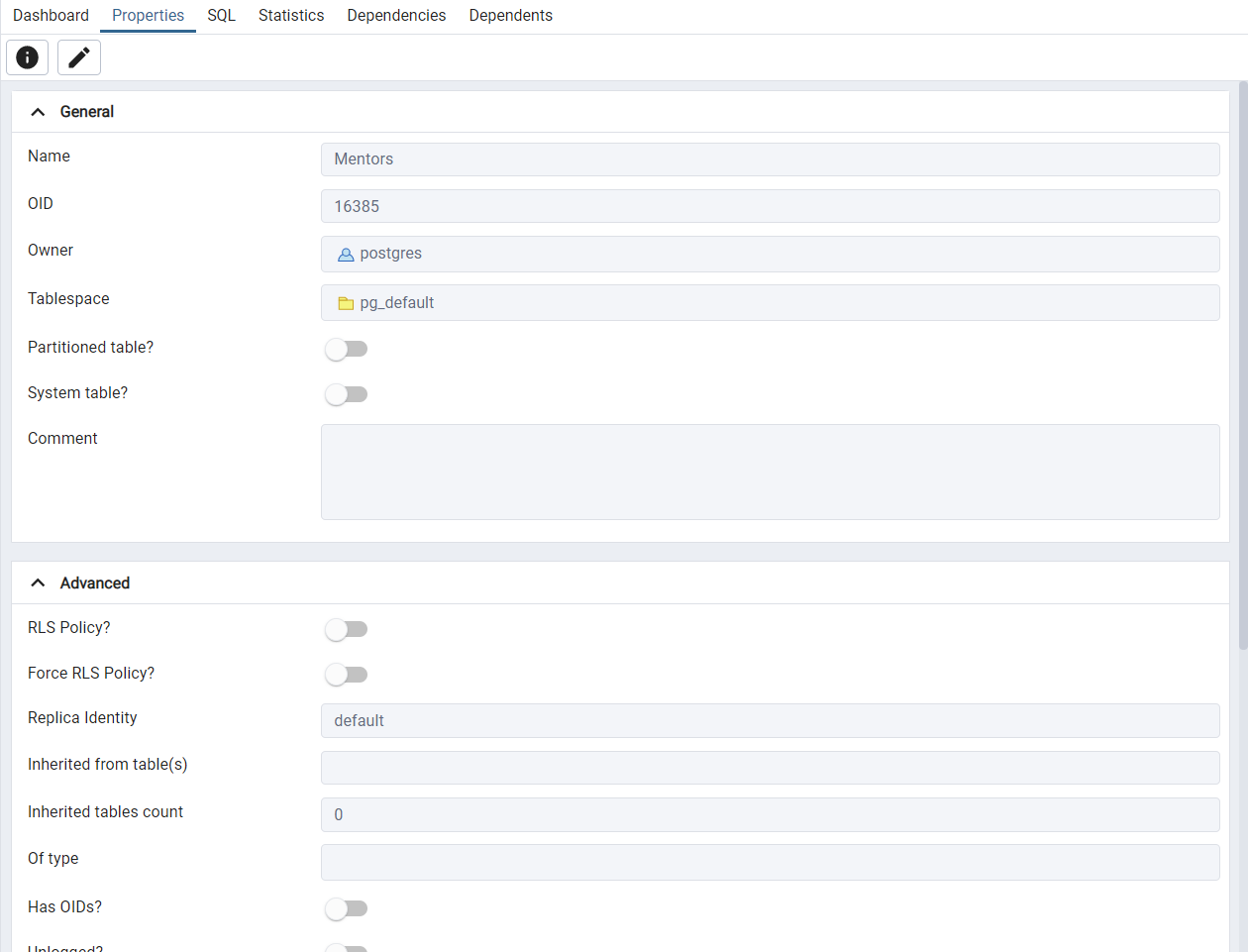
Nazwanie tabeli



Dodanie do tabeli kolumn

Zauważ, że kolumnom nadajemy również odpowiednie właściwości - wszystkie cechy mentora nie mogą być puste (*Not NULL*), a *id* musi być *Primary Key*.

Proces tworzenia tabeli kończymy naciśnięciem przycisku *Save* i w rezultacie otrzymamy utworzoną tabelę:

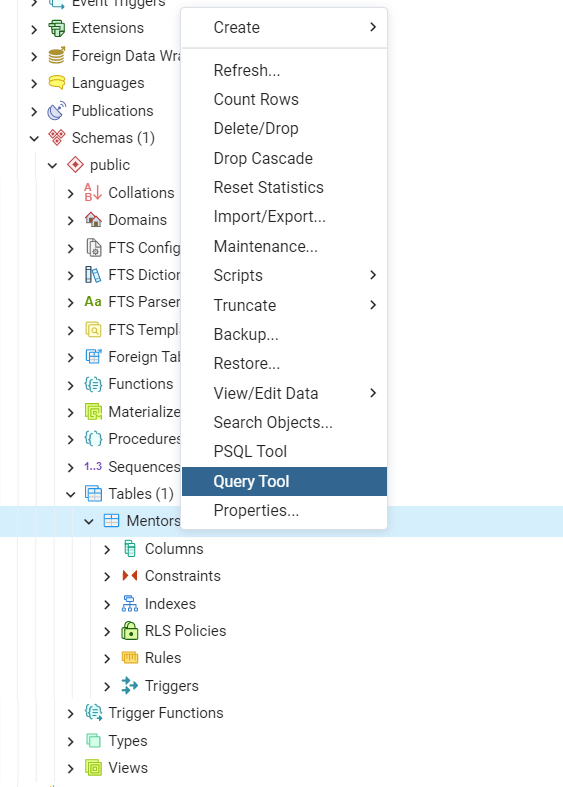


Widok na utworzoną tabelę

**Praca na tabeli**

Następnym krokiem, jakim się zajmiemy będzie dodanie odpowiedniej zawartości do tabeli Mentors przez umieszczenie konkretnych rekordów - mentorów.

Przeprowadzanie operacji bezpośrednio na tabeli umożliwia nam **Query Tool**, do którego możemy się dostać klikając PPM na nazwę tabeli i wybierając odpowiednią zakładkę z listy.



Wyświetli się wówczas edytor, do którego możesz wpisywać odpowiednie zapytania języka SQL.

Zanim jednak będziemy mogli przeprowadzać odpowiednie operacje na tabeli, musimy poznać składnię SQL-a. Nad tym zagadnieniem pochylimy się dokładniej, gdy przejdziemy do zarządzania bazą z poziomu CLI (**psql**).

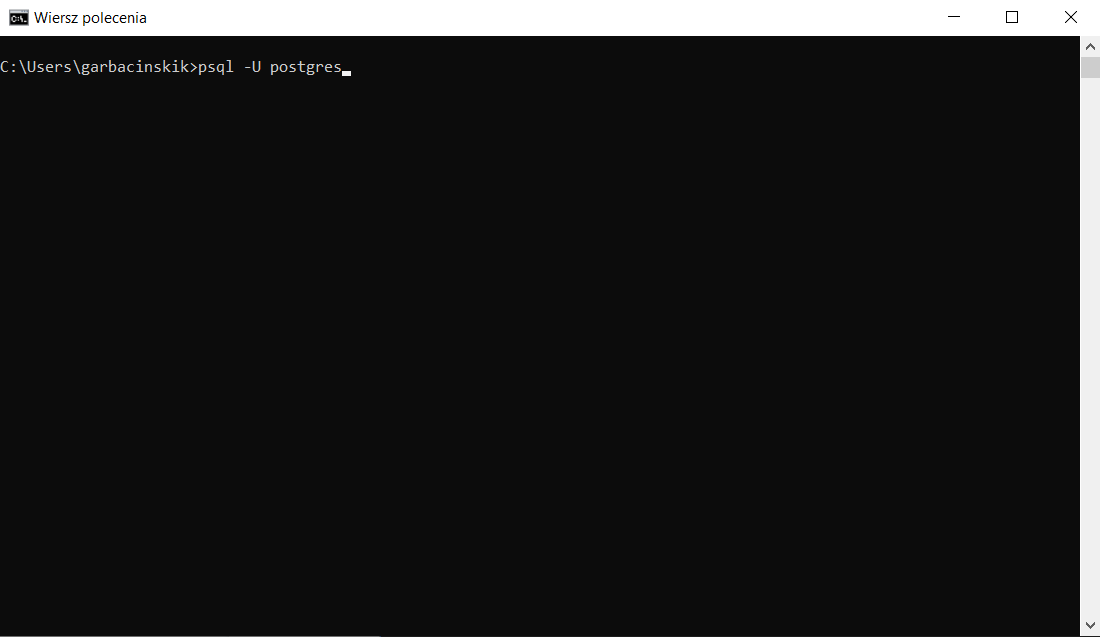
Wtedy pozwolę poświęcić sobie osobną sekcję w szkoleniu na wprowadzenie Cię we wszystkie operacje i rodzaje zapytań, jakie możemy przeprowadzać w PostgreSQL.

Sprawdźmy natomiast teraz, jak przeprowadzić te same procesy, co do tej pory w pgAdmin 4 (czyli *utworzenie bazy danych* i *umieszczenie w niej tabeli*), ale z poziomu interfejsu terminalowego - **psql**.

**Utworzenie bazy z psql**

Nadeszła teraz chwila na to, abym zaprezentował Ci, jak zarządzać Postgresowymi bazami danych z poziomu CLI.

W tym celu przejdź do terminala (Windowsego lub Unixowego) i wpisz następujące polecenie, aby połączyć się z klientem PostgreSQL jako superuser **postgres**:

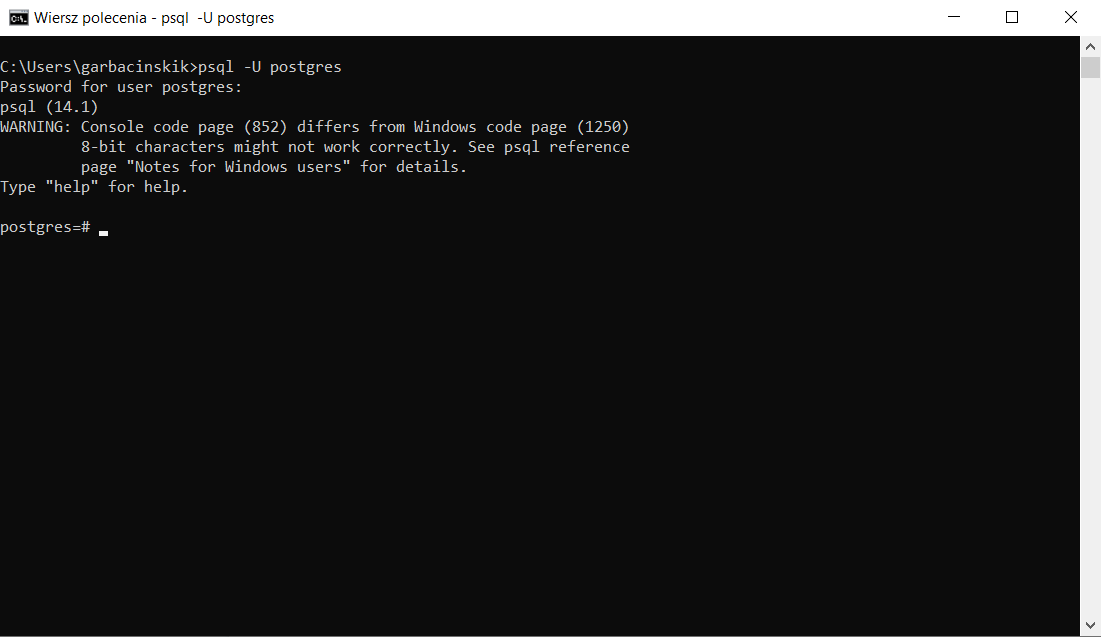


Polecenie umożliwiające połączyć się z klientem jako użytkownik postgres:

| psql -U postgres |
| --- |

Po wpisaniu tego polecenia zostaniesz poproszony o podanie credentiali określonego użytkownika (postgres).

W rezultacie powinieneś otrzymać widok jak poniżej:



Znak zachęty **postgres=#** informuje nas, że terminal jest w stanie przyjmować już konkretne Postgresowe polecenia. Poniżej przedstawię Ci zestaw komend, które będziemy najczęściej wykorzystywali. Warto zapamiętać poniższy zestaw, ponieważ łatwiej i efektywniej będzie Ci się pracowało z klientem psql.

**Zestawienie poleceń psql**

| **Polecenie** | **Opis** |  |
| --- | --- | --- |
| **\c** *baza\_danych* | Zmienia połączenie z konkretną bazą danych |  |
| **\l** | Listuje wszystkie dostępne bazy |  |
| **\dt** | Listuje wszystkie dostępne tabele |  |
| **\d** *table\_name* | Opisuje wybraną tabelę (jej kolumny, atrybuty) |  |
| **\dn** | Listuje wszystkie schematy w bieżącej bazie danych |  |
| **\df** | Listuje dostępne funkcje w bieżącej bazie danych |  |
| **\dv** | Listuje dostępne widoki w bieżącej bazie danych |  |
| **\du** | Listuje wszystkich dostępnych użytkowników wraz z opisem uprawnień |  |
| **SELECT version();** | Wyświetla aktualną wersję PostgreSQL |  |
| **\g** | Wykonuje ponownie ostatnio wpisane polecenie |  |
| **\s** | Wyświetl historię wykonanych poleceń |  |
| **\q** | Wychodzi z psql shell |  |

Część poleceń z powyższej listy będziemy wykorzystywali częściej, inną część radziej.

**Utworzenie bazy danych**

Będąc już na tym etapie, możemy utworzyć bazę danych, ale tym razem z poziomu psql. Stworzymy bliźniaczą wersję do tej, która powstała podczas pracy z GUI - pgAdminem. To dlatego, że po prostu chcę przedstawić Ci, jak zrobić to samo, ale z użyciem terminala. Nasza baza danych nazywać się będzie: *test-psql*.

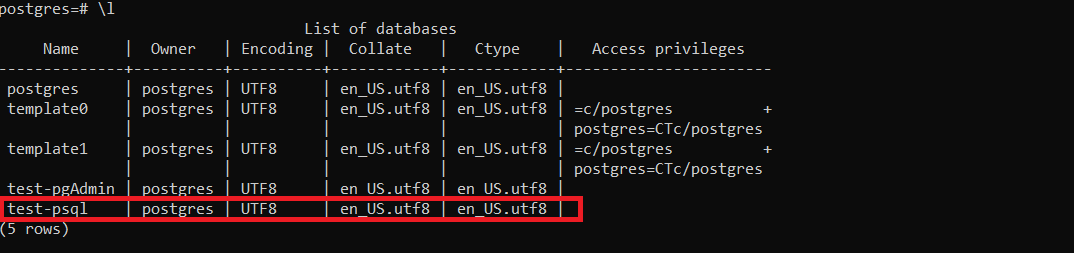
Aby ją utworzyć, wykonamy następujące polecenie:

| CREATE DATABASE test-psql; |
| --- |

**Efekt**



To, czy ona faktycznie została utworzona, możemy sprawdzić, wpisując polecenie **\l**.

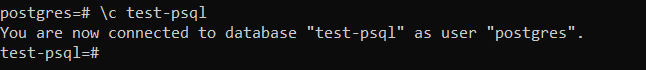


**Połączenie z bazą danych**

Utworzywszy bazę test-psql, możemy teraz nawiązać z nią połączenie, wpisując:

| \c test-psql; |
| --- |

**Efekt**



Jesteśmy teraz gotowi na bezpośrednie modyfikowanie bazy, poprzez tworzenie nowych tabel wraz z odpowiednimi atrybutami i rekordami! Jednak…

Aby móc to realizować za pośrednictwem terminala, będziemy musieli poznać SQL-a w implementacji PostgreSQL i wszystkie polecenia z nim związane.

Dlatego oderwijmy się teraz nieco od psql i przejdźmy do omówienia, jakie wyróżniamy polecenia SQL-owe.

**Operacje w języku SQL**

W języku SQL i implementacji PostgreSQL możemy wyróżnić kilka rodzajów operacji przetwarzania danych.

Są to:

* **DDL** **- Data Definition Language** - służą do manipulowania **strukturą** danych, np. dodawanie/usuwanie atrybutów, zmiana typu danych
* **DML - Data Manipulation Language** - instrukcje manipulujące danymi, np. usuwanie i dodawanie rekordów, modyfikacja zawartości**.**
* **DCL - Data Control Language** - służą do sterowania uprawnieniami grup/użytkowników na serwerze.
* **DQL - Data Query Language** - polecenia tego pokroju umożliwiają odczytywanie danych z bazy

Omówmy każdą z powyższych grup. Zaczniemy od DML:

**Operacje DDL**

W sekcji tej omówimy następujące polecenia:

* **CREATE** - tworzenie struktury (np. bazy, tabeli)
* **DROP** - usunięcie struktury
* **ALTER** - zmiana struktury (np. dodanie kolumny do tabeli)
* **TRUNCATE** - usunięcie danych w obrębie tabeli

Jak już się przekonałeś, utworzona baza danych nie zawiera w sobie żadnych tabel. Dlatego musisz poznać sposób na tworzenie encji w obrębie danej bazy. W tym celu użyjemy polecenia **CREATE TABLE**.

| **UWAGA!**  Pamiętaj, aby przed wykonaniem poniższych komend, być połączonym ze wcześniej utworzoną bazą danych (w naszym przypadku test-psql; zrealizowaliśmy to już przy użyciu wcześniejszego polecenia  **\c test-psql**). |
| --- |

**Tworzenie tabeli**

W celu stworzenia tabeli, wykorzystamy syntax:

| CREATE TABLE <table\_name> (column\_name\_1 datatype, column\_name\_2 datatype, . . column\_name\_n datatype ); |
| --- |

Zauważ, że pomiędzy nawiasami takiego polecenia, umieszczamy dowolną ilość kolumn, określając ich datatype. PostgreSQL dostarcza nam możliwość wykorzystania naprawdę wielu różnych typów danych, jak m.in: **integer** (wartość liczbowa), **boolean** (wartość true/false), **date** (reprezentacja daty), **double precision** (liczba z częścią dziesiętną).

Listę wszystkich dostępnych typów danych w PostgreSQL możesz znaleźć tutaj: [Getting Started with PostgreSQL Data Types](https://www.postgresqltutorial.com/postgresql-data-types/).

Zwróć szczególną uwagę na możliwość wykorzystywania typu JSON do wprowadzania rekordów tabeli. Jest to jedna z cech PostgreSQL, która znaczącą rozróżnia implementacje bazodanowe na tyle innych. Poza tym o to zagadnienie często zahaczają rozmowy rekrutacyjne, dlatego jest to dodatkowa motywacja do dokładnego zapoznania się z typami w Postgre.

**Utworzenie tabeli Mentors**

Znając już podstawową składnię umożliwiającą utworzenie tabeli, możemy przejść do zbudowania tabeli Mentors (o dokładnie takich atrybutach jak przy korzystaniu z klienta pgAdmin).

Tabela Mentors składać się będzie zatem z następujących kolumn:

* **id** - *PK, typ integer*
* **name** - *not null, text*
* **surname** - *not null, text*

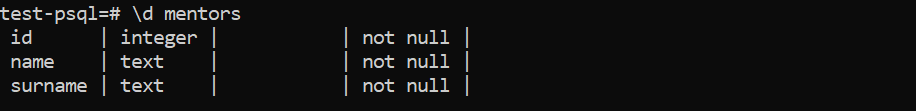
Wykonajmy zatem następujące zapytanie w psql:



Operacja przebiegła bez żadnego errora, możemy teraz sprawdzić, czy po wpisaniu *\dt* wyświetla się utworzona tabela.



Możemy sprawdzić również, jakie kolumny znajdują się w obrębie tabeli mentors (polecenie *\d mentors*).



Wygląda na to, że wszystko udało się stworzyć poprawnie i skutecznie przeprowadziliśmy tworzenie tabeli przez psql!

Jak już jesteśmy przy poleceniach DDL-owych, to zaprezentuję Ci również możliwości pozostałych poleceń. Spróbujmy nieco zmodyfikować strukturę naszej tabeli.

**Modyfikacja struktury tabeli Mentors**

W celu zmienienia struktury encji w obrębie danej bazy danych, wykorzystamy polecenie ALTER TABLE, które ma następujący syntax:

| ALTER TABLE <tableName> <OPERATION>  Gdzie <OPERATION> to: ADD/SET/DROP |
| --- |

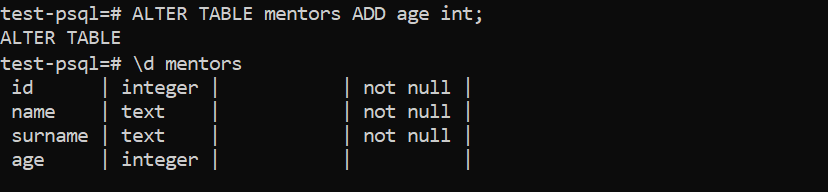
* **ADD** - umożliwia dodanie nowej kolumny do istniejącej tabeli
* **SET** - zmienia właściwość wskazanej kolumny w istniejącej tabeli, np. utworzenie danej kolumny jako PK
* **RENAME** - umożliwia zmianę nazwy wybranego atrybutu w istniejącej tabeli
* **DROP** - usuwa wybraną kolumnę (lub określoną cechę) z istniejącej tabeli

1. **Polecenie ALTER TABLE … ADD**

Załóżmy, że chcielibyśmy dodać do tabeli Mentors dodatkowy atrybuty określający, w jakim wieku jest mentor. Wykonajmy zatem następujące zapytanie:

| ALTER TABLE mentors ADD age int; |
| --- |

**Wynik**



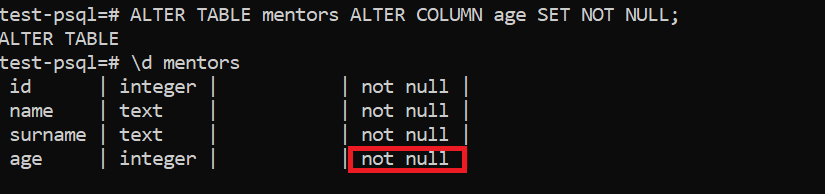
Jak widzisz z powyższego screena, tabela mentors zawiera również od tego momentu kolumnę age, która jest typu integer i może być nullem (czyli może być pusta).

1. **Polecenie ALTER TABLE … SET**

Załóżmy, że zdecydowaliśmy teraz, aby w rekordzie reprezentującym mentora - atrybut age nie mógł być pusty. Zapiszemy wówczas:

| ALTER TABLE mentors ALTER COLUMN age SET NOT NULL; |
| --- |

W rezultacie możemy zauważyć, że do kolumny *age* została dodana właściwość *not null*.

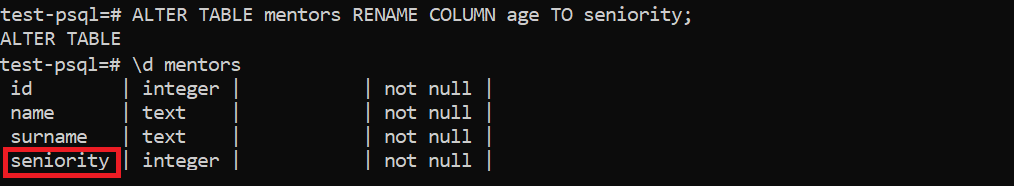


1. **Polecenie ALTER TABLE … RENAME**

Zmieńmy teraz nazwę kolumny age na seniority. Wykorzystamy polecenie:

| ALTER TABLE mentors RENAME COLUMN age TO seniority; |
| --- |

Wynik

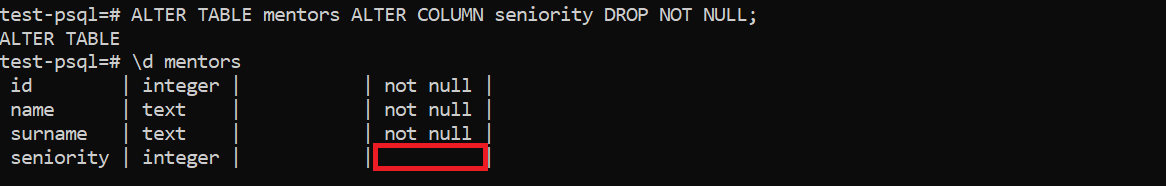


1. **Polecenie ALTER TABLE … DROP**

W trakcie pracy z tabelami może się okazać, że chcemy usunąć daną cechę przypisaną do kolumny. Załóżmy, że z kolumny seniority chcemy usunąć cechę *NOT NULL*:

| ALTER TABLE mentors ALTER COLUMN seniority DROP NOT NULL; |
| --- |

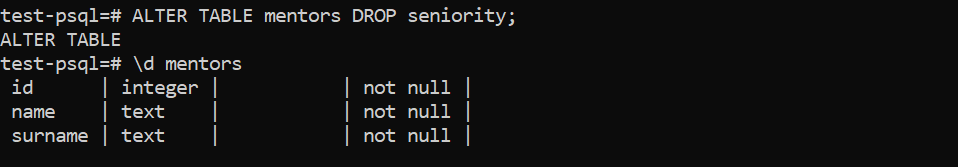
Wynik



DROP może nam też posłużyć do usunięcia całej kolumny. Załóżmy, że kolumna seniority jest już nam niepotrzebna. Wówczas bez problemu usuniemy ją poleceniem:

| ALTER TABLE mentors DROP seniority; |
| --- |

Wynik



To tyle, jeśli chodzi o najczęściej wykorzystywane wariancje poleceń DDL. Czas na omówienie DML-a.

**DML - Data Manipulation Language**

Najczęściej stosowanymi poleceniami wykonywanymi w ramach tej grupy operacjami są:

* **INSERT** - służy do umieszczania danych w tabeli,
* **UPDATE** - odpowiada za zmianę danych
* **DELETE** - usuwa dane

1. **Dodawanie rekordów do tabeli - INSERT**

Nadeszła teraz idealna chwila na to, aby do wcześniej utworzonej tabeli Mentors dodać kilka rekordów - 3 nowych mentorów. Aby to zrealizować, wykorzystamy polecenie *INSERT INTO*, które przyjmuje następujący syntax:

| INSERT INTO table\_name (column1, column2, column3,..) VALUES ( value1, value2, value3,..); |
| --- |

Chcąc zatem dodać do naszej tabeli następujących mentorów…

Kacper Nowak

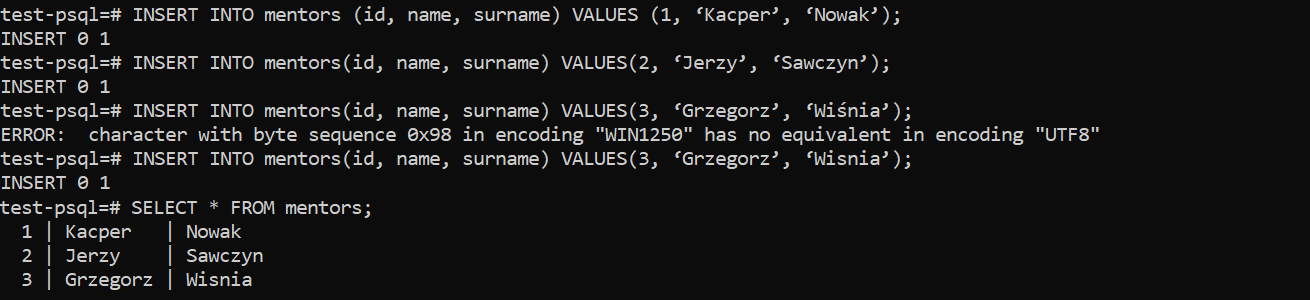
Jerzy Sawczyn

Grzegorz Wiśnia

… przygotujemy następujące zapytanie (zakładamy, że nasza tabela Mentors składa się z takich kolumn jak: *id*, *name*, *surname*):

| INSERT INTO mentors (id, name, surname) VALUES (1, 'Kacper', 'Nowak');  INSERT INTO mentors(id, name, surname) VALUES(2, 'Jerzy', 'Sawczyn');  INSERT INTO mentors(id, name, surname) VALUES(3, 'Grzegorz', 'Wisnia'); |
| --- |

Realizacja w terminalu:



Wykonane na końcu polecenie *SELECT \* FROM mentors;* wyświetla wszystkich mentorów dostępnych w tabeli. Ukazuje to nam, że zostali oni poprawnie dodani do wskazanego miejsca. O poleceniu *SELECT* powiemy szerzej w kontekście grupy zapytań DQL.

Niektórzy z programistów wolą zapisywać polecenie INSERT w nieco krótszy sposób. Wówczas taki zapis:

| INSERT INTO mentors(id, name, surname) VALUES(3, 'Grzegorz', 'Wisnia'); |
| --- |

Możemy przekształcić w:

| INSERT INTO mentors VALUES(3, 'Grzegorz', 'Wisnia'); |
| --- |

Spotkałem się jednak z wieloma opiniami, że pierwszy sposób jest lepszy i bardziej czytelny, ponieważ jawnie listuje kolumny, do których mają być umieszczane dane.

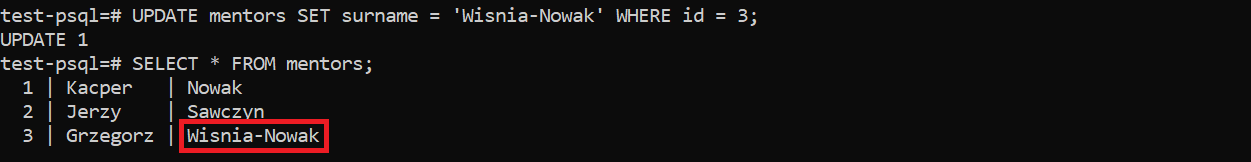
1. **Aktualizacja danych - UPDATE**

Przy użyciu następnego polecenia należącego do grupy DML - *UPDATE*, możemy aktualizować znajdujące się w danej tabeli rekordy. Omawiane polecenie przyjmuje następujący syntax:

| UPDATE table\_name SET column1 = value1, column2 = value2,... columnn = valuen WHERE condition; |
| --- |

Załóżmy, że chcemy w tym momencie zmienić nazwisko mentora Wisnia na Wisnia-Nowak (mentor ten ma id równe 3).

Zapiszemy:



Zwróć szczególną uwagę na to, w jaki sposób określiliśmy, który rekord ma zostać usunięty z bazy. Doprecyzowaliśmy to przez użycie warunku umieszczonego po słowie kluczowym **WHERE**. Jest to nieodłączny operator przy określaniu warunków, które muszą spełniać dane.

Warunki te mogą występować pojedynczo lub być dowolnie łączone przy użyciu takich słów jak *AND* czy *OR*. O *WHERE* opowiemy szerzej, gdy przejdziemy do wykorzystywania polecenia *SELECT* w naszych queries.

1. **Usuwanie danych - DELETE**

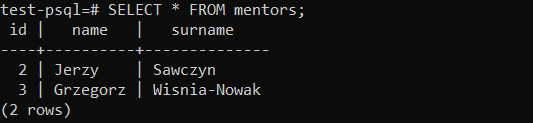
Przypuśćmy, że chcielibyśmy usunąć teraz dane rekordy (np. wszystkich mentorów, których imię to ‘Kacper’). Zapiszemy wówczas zapytanie zgodnie z szablonem…

| DELETE FROM table\_name WHERE condition; |
| --- |

… czyli:

| DELETE FROM mentors WHERE name = 'Kacper'; |
| --- |

Mentor o imieniu Kacper zostanie usunięty:



Wraz z poleceniem DELETE mogą być również związane dodatkowa ciekawa operacja jak: **IN…** Przykładowo - umożliwia ona określenie grupy rekordów, które mają zostać usunięte z bazy.   
  
Tak jak poniżej:

| DELETE FROM mentors WHERE name IN ('Kacper', 'Jan'); |
| --- |

W wyniku takiego polecenia, usunięci zostaną Ci mentorzy, których imię to: Kacper lub Jan.

| UWAGA!  Jeżeli chcesz usunąć **wszystkie** rekordy z tabeli, wykorzystaj operację TRUNCATE, które przybiera postać:  TRUNCATE nazwa\_tabeli; |
| --- |

**Tworzenie relacji w tabelach**

Aby móc efektywnie korzystać z zapytań z grupy DML, musimy wiedzieć również, jak tworzyć relacje między tabelami. Nierzadko się bowiem będzie zdarzało, że w tabeli znajdować się będzie klucz obcy do innej tabeli. Weźmy choćby pod uwagę niejednokrotnie występującą już w tym szkoleniu strukturę bazy, gdzie wyróżnialiśmy powiązane ze sobą relacją **1:n** - tabele Students oraz Mentors.

**Students**

| **id** | **name** | **surname** | **mentor\_id (FK)** |
| --- | --- | --- | --- |
| 1 | Kacper | Nowak | 1 |
| 2 | Jerzy | Sawczyn | 2 |
| 3 | Kacper | Stysło | 3 |
| 4 | Kinga | Szewczyk | 2 |

**Mentors**

| **id** | **name** | **surname** | **specialization** |
| --- | --- | --- | --- |
| 1 | Kacper | Nowak | Python |
| 2 | Jerzy | Sawczyn | C++ |
| 3 | Kacper | Stysło | Java |

Sprawdźmy, jak odzwierciedlić powyższe tabele w PostgreSQL (wykorzystamy tak naprawdę znajome Ci już polecenia CREATE TABLE, INSERT, jednak z zaznaczeniem, że niektóre z kolumn mają przechowywać klucze obce i odnosić się do konkretnych tabel).

Istnieje kilka sposobów na dodanie klucza obcego do tabeli.

| **Uwaga**  W celu dodaniu klucza obcego odnoszącego się do tabeli Mentors, konieczne jest, abyś przed utworzeniem Students, stworzył najpierw tabelę (Mentors), do której relacja będzie się odnosiła.  Nie możemy bowiem tworzyć kolumny reprezentującej FK do tabeli, która jeszcze nie została utworzona. |
| --- |

Pierwszym, najbardziej powszechnym sposobem na utworzenie tabeli powiązanej relacją z inną jest:

| CREATE TABLE Students (   id INT PRIMARY KEY ,  name VARCHAR(100) NOT NULL,  surname VARCHAR(100) NOT NULL,  mentor\_id INT REFERENCES mentors(id) );  Lub drugi, też popularny sposób - przy użyciu słowa kluczowego **CONSTRAINT:**  CREATE TABLE Students (   id INT PRIMARY KEY,  name VARCHAR(100) NOT NULL,  surname VARCHAR(100) NOT NULL,  mentor\_id INT,  CONSTRAINT fk\_mentor\_id   FOREIGN KEY(mentor\_id)   REFERENCES mentors(id) ); |
| --- |

**Zadanie**

Do tak utworzonej tabeli Students, spróbuj dodać kolejne rekordy odnoszące się do odpowiednich mentorów. Spróbuj jednak dodać studenta, który będzie odnosił się do mentora o nieistniejącym id. Czy taka operacja przebiegnie bez błędu?

**DQL - Data Query Language**

Czas na omówienie ostatniej grupy poleceń SQL. W jej zakres wchodzi polecenie - **SELECT**. Wykorzystywaliśmy je, gdy przykładowo potrzebowaliśmy wyświetlić wszystkie rekordy z tabeli. Polecenie przyjmowało wówczas postać:

| SELECT \* FROM table\_name; |
| --- |

Zatrzymajmy się teraz na chwilę przy strukturze takiego polecenia. Okazuje się, że ogólny schemat najbardziej podstawowego polecenia *SELECT* może wyglądać następująco:

| SELECT column1, column2, ..., columnn FROM table\_name; |
| --- |

W ten sposób wskazujemy, z jakich kolumn z podanej tabeli chcemy pobierać rekordy. Jeżeli natomiast chcemy, aby wyświetlał się **każdy** atrybut, wtedy zamiast ręcznie listować każdą z kolumn, po prostu zapisujemy **\***.

**WHERE, GROUP BY, JOIN, HAVING**

SELECT jest również idealnym poleceniem na zrozumienie innych, równie często wykorzystywanych poleceń SQL, jak: **WHERE**, **GROUP BY**, **JOIN, HAVING**.

Wiele z nich nie muszą być nawet wykorzystywane przy SELECT, ale sprawdzą się również przy innego typu poleceniach (zauważ, że WHERE wykorzystywaliśmy przy, np. DELETE).

**O WHERE kilka słów więcej…**

Poznałeś wcześniej następujący zapis:

DELETE FROM table\_name WHERE condition;

Teraz chciałbym pokazać Ci, z jakimi poleceniami może dodatkowo łączyć się klauzula WHERE. Zacznijmy od **AND** oraz **OR**.

**Operatory AND, OR**

Wspomniane AND oraz OR są operatorami logicznymi. Mają one takie samo zastosowanie jak odpowiedniki w Pythonie - umożliwiają łączenie kilku predykatyw w jedną całość. Dzięki temu możemy tworzyć jeszcze bardziej szczegółowe warunki, np. filtrowania informacji.

Załóżmy, że mamy następującą tabelę:

*Students*

| **id** | **name** | **surname** | **age** |
| --- | --- | --- | --- |
| 1 | Julian | Kazek | 33 |
| 2 | Adam | Nowak | 29 |
| 3 | Jan | Kowalski | 18 |
| 4 | Anna | Kowalczyk | 25 |
| 5 | Tadeusz | Janowski | 40 |
| 6 | Karol | Filipiuk | 15 |

**Zadanie**

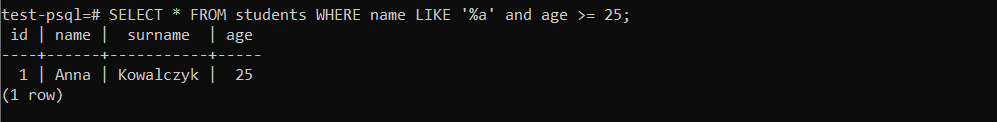
Spróbuj utworzyć bliźniaczą tabelę u siebie w programie, wykorzystując polecenia CREATE i INSERT.

Załóżmy, że chcielibyśmy teraz przefiltrować studentów tak, aby wyświetlić tylko tych, którzy **mają co najmniej 25 lat i są płci żeńskiej** (czyli ich imię kończy się na literę *a*).

Zapiszemy zatem:

| SELECT \* FROM Students WHERE name LIKE '%a' AND age >= 25; |
| --- |

Rezultat



W powyższym przykładzie połączyliśmy za pomocą operatora AND dwa warunki: LIKE '%a'oraz age >= 25. Dzięki temu filtrowane są tylko te rekordy, które **równocześnie** spełniają oba z warunków.

Nie wątpię, że **LIKE** może być dla Ciebie nieco mniej oczywistym operatorem, omówimy go za chwilę. Przejdźmy jeszcze na chwilę do przedstawienia Ci operatora OR.

**Operator OR**

Wyobraźmy sobie, że teraz chcielibyśmy pobrać tylko tych uczniów, których imię rozpoczyna się na J **lub** K. Aby to zrealizować, zapiszemy polecenie takie jak poniżej:

| SELECT \* FROM Students WHERE name LIKE 'J%' OR name LIKE 'K%'; |
| --- |

Tutaj sytuacja ma się nieco inaczej. Wyświetlone zostaną tylko te krotki, które spełniają **przynajmniej jeden** z przedstawionych warunków.

**Operator LIKE, ILIKE**

Operator LIKE, jak już zapewne zauważyłeś, służy do, tzw. **pattern matching**, czyli określania, po jakich warunkach ma być filtrowany tekst.

LIKE umożliwia wyszukiwanie takich tekstów, które które **zaczynają się na określoną literę**, **mają w środku dany ciąg znaków** itd.

Co istotne, przy określaniu wzoru danego tekstu, w jego treści umieszczamy **%** lub **\_**.

Są to tzw. "**SQL wildcards**":

Znak **%** dopasowuje dowolną ilość liter w danym ciągu (0 lub więcej)

Znak **\_** odpowiada dowolnej pojedynczej literze.

Różnicę między powyższymi wildcard'ami lepiej zrozumiesz na pewno na przykładach:

* ***s%*** - odpowiada dowolnemu stringowi, który zaczyna się na literę **s** i ma **dalej dowolne litery** (lub ich brak)
* ***s\_***- odpowiada dowolnemu dwuelementowemu napisowi, który zaczyna się na s
* ***%per%***- odpowiada (wybacz za angielskie przykłady) słowom pep**per, per**cent
* natomiast *h\_nt* słowom **h**u**nt**, **h**i**nt**

SQL Wildcards połączone z operatorem LIKE umożliwiają nam tworzenie efektywnych predykatyw sprawdzających, czy tekst jest dopasowany do danego wzorca.

ILIKE, który może być również wykorzystywany do sprawdzania wzorców, koncepcyjnie służy do tego samego co LIKE, ale jest “*case insensitive*”.

Oznacza to, że, wykorzystując ILIKE, wielkość liter jest pomijana i przykładowo wzorzec: “Devs-Ment%” jest traktowany tak samo jak: “devs-MeNt%’.

**Podzapytania SELECT**

Nierzadko może się zdarzać, że będziesz potrzebował zagnieździć w sobie dwa SELECT-y. Takie rozwiązanie sprawdza się, gdy potrzebujemy przefiltrować elementy według bardziej złożonego warunku. Najlepiej będzie to zrozumieć na przykładzie:

Wyobraź sobie następującą tabelę (Employees):

| id | name | surname | email | salary |
| --- | --- | --- | --- | --- |
| 1 | Joanna | Nowak | jn@gmail.com | 10 000 |
| 2 | Katarzyna | Kowalska | kk@gmail.com | 45 000 |
| 3 | Kinga | Sawczyn | ks@gmail.com | 5 000 |
| 4 | Kacper | Wiśniewski | kw@hotmail.com | 20 000 |
| 5 | Jerzy | Nowaczuk | jnn@hotmail.com | 25 000 |
| 6 | Grzegorz | Lubimił | gl@o2.pl | 30 000 |

Załóżmy teraz, że chcielibyśmy wyświetlić tylko takie kobiety, których zarobki są większe od maksymalnej pensji dowolnego mężczyzny.

Takie zapytanie będziemy musieli podzielić na dwie części:

* Zapytanie wewnętrzne wyłuska maksymalnę kwotę zarobków u mężczyzn
* Zapytanie zewnętrzne znajdzie tylko takie kobiety, których pensja jest wyższa od wartości wyżej

Przykładowy kod:

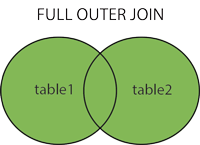
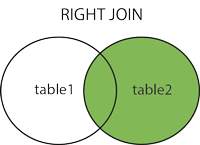
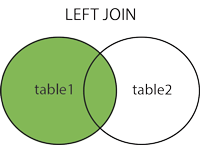
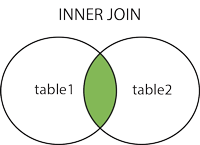
| SELECT \* FROM Employees WHERE name LIKE '%a' AND salary > (SELECT max(salary) FROM Employees WHERE name NOT LIKE '%a'); |
| --- |

**JOINy**

Polecenie JOIN przydaje się w momencie, gdy potrzebujemy wyłuskać dane z powiązanych ze sobą **relacją tabel**. W zależności od tego, jaki efekt chcemy otrzymać - wyróżniamy następujące rodzaje JOINÓW:

* **(INNER) JOIN**: Zwraca wszystkie dopasowane rekordy z obydwu tabel
* **LEFT (OUTER) JOIN**: Zwraca **wszystkie** rekordy z lewej tabeli i niektóre (tylko te dopasowane) z prawej
* **RIGHT (OUTER) JOIN**: Zwraca wszystkie rekordy z prawej tabeli i niektóre (tylko te dopasowane) z lewej
* **FULL (OUTER) JOIN**: Zwraca wszystkie rekordy z obu tabel, łącznie z dopasowanymi

Wartości podane między nawiasami są opcjonalne, nie podaje się ich w zapytaniu, jednak możesz czasami spotkać się z dodawaniem słowa “OUTER” do nazw operacji.

Źródło: w3schools.com

JOINy tylko na początku wyglądają na zawiłe i skomplikowane. Aby maksymalnie przystępnie przedstawić Ci sens ich wykorzystywania, przeanalizujmy poniższe tabele:

**Students**

| **id** | **name** | **surname** | **mentor\_id (FK)** |
| --- | --- | --- | --- |
| 1 | Kacper | Nowak | 1 |
| 2 | Jerzy | Sawczyn | 2 |
| 3 | Kacper | Stysło | 3 |
| 4 | Kinga | Szewczyk | 2 |
| 5 | Jan | Siawcz |  |

**Mentors**

| **id** | **name** | **surname** | **specialization** |
| --- | --- | --- | --- |
| 1 | Grzegorz | Gregorian | Python |
| 2 | Agnieszka | Konieczna | C++ |
| 3 | Stanisław | Rychło | Java |
| 4 | Marcin | Tyszka | JavaScript |

Zauważ, że tabela **Students** jest powiązana relacją jeden do wielu z **Mentors**. Taka sytuacja umożliwia nam trafne zaprezentowanie możliwość JOIN-ów. Tego typu tabelę tworzyliśmy już we wcześniejszej sekcji tego szkolenia - *“Tworzenie relacji w tabelach”*.

Przetestujmy **INNER JOIN**:



| SELECT \* FROM Students INNER JOIN Mentors ON Students.mentor\_id = Mentors.id;  Wynikiem powyższego zapytania będzie output poniższej postaci:  1 Kacper Nowak 1 1 Grzegorz Gregorian Python  2 Jerzy Sawczyn 2 2 Agnieszka Konieczna C++  3 Kacper Stysło 3 3 Stanisław Rychło Java  4 Kinga Szewczyk 2 2 Agnieszka Konieczna C++  Tak jak już wspominaliśmy - pobrane zostały informacje o wszystkich rekordach połączonych relacją na poziomie tabel Students - Mentors.  **LEFT JOIN:**  Kod:  SELECT \* FROM Students LEFT JOIN Mentors ON Students.mentor\_id = Mentors.id;  Efekt:  1 Kacper Nowak 1 1 Grzegorz Gregorian Python  2 Jerzy Sawczyn 2 2 Agnieszka Konieczna C++  3 Kacper Stysło 3 3 Stanisław Rychło Java  4 Kinga Szewczyk 2 2 Agnieszka Konieczna C++ |
| --- |

5 Jan Siawcz

**RIGHT JOIN:**

Po wykonaniu następującego polecenia:

SELECT \* FROM Students RIGHT JOIN Mentors ON Students.mentor\_id = Mentors.id;

Otrzymamy poniższy wynik:

1 Kacper Nowak 1 1 Grzegorz Gregorian Python

2 Jerzy Sawczyn 2 2 Agnieszka Konieczna C++

3 Kacper Stysło 3 3 Stanisław Rychło Java

4 Kinga Szewczyk 2 2 Agnieszka Konieczna C++

4 Marcin Tyszka JavaScript

**FULL JOIN:**

Zapytanie:

SELECT \* FROM Students FULL JOIN Mentors ON Students.mentor\_id = Mentors.id;

Wynik:

1 Kacper Nowak 1 1 Grzegorz Gregorian Python

2 Jerzy Sawczyn 2 2 Agnieszka Konieczna C++

3 Kacper Stysło 3 3 Stanisław Rychło Java

4 Kinga Szewczyk 2 2 Agnieszka Konieczna C++

Null 4 Marcin Tyszka JavaScript

5 Jan Siawcz Null

Jak zatem widzisz JOINy są skutecznym i wygodnym rozwiązaniem na prezentowanie powiązań między określonymi tabelami. Będziemy z nich szeroko i często korzystali w momencie, gdy potrzebować będziesz obsłużyć relacje (zarówno 1:1, 1:n, m:n) między tabelami.

**Przydatne polecenia wspomagające**

Powyżej zapoznałeś się z najważniejszymi (zarówno łatwymi jak i nieco trudniejszymi) poleceniami z grup DML, DQL, DDL. Wraz z takimi poleceniami mogą również pojawiać się polecenia wspomagające, które ułatwiają, np. **porządkowanie danych**. Na zwieńczenie całego szkolenia, pochylimy się nad następującymi operacjami: **ORDER BY**, **LIMIT**, **COUNT***.*

1. **ORDER BY**

Polecenie to umożliwia sortowanie wyników według danego atrybutu, np. sortowanie danych wynikowych po nazwisku.

Jak się zapewne domyślasz, sortować możemy w sposób **niemalejący** jak i **nierosnący**. Dlatego w celu określenia, w jaki sposób dane mają być ułożone, oprócz samego **ORDER BY** użyjemy **DESC** (descending)oraz **ASC (**ascending - wartość domyślna).

Załóżmy, że chcemy posortować **rosnąco** według daty rozpoczęcia nauki wszystkich pobranych użytkowników, **których nazwisko ma więcej niż 10 liter**:

| SELECT \* FROM Uczniowie WHERE length(nazwisko) > 10 ORDER BY data\_rozp ASC; |
| --- |

Chcąc przeprowadzić podobną operację, ale tym razem sortując tych samych uczniów **malejąco**,zapiszemy:

| SELECT \* FROM Uczniowie WHERE length(nazwisko) > 10 ORDER BY data\_rozp DESC; |
| --- |

1. **LIMIT**

Używany, gdy chcemy ograniczyć ilość wyświetlanych rzędów danych, np. gdy chcemy pobrać **10 ostatnio** zapisanych użytkowników na kurs programowania.

| SELECT \* FROM Uczniowie ORDER BY start\_date DESC LIMIT 10; |
| --- |

1. **GROUP BY (COUNT, SUM)**

Polecenie to dzieli w grupy zwrócone rekordy z SELECT. Dla każdej z grup możesz stosować, tzw. operacje agregujące - **COUNT** i **SUM**. Syntax wspomnianej operacji wygląda tak jak poniżej:

| SELECT   column\_1,   column\_2,  ...,  aggregate\_function(column\_3) FROM   table\_name GROUP BY   column\_1,  column\_2,  ...; |
| --- |

W powyższym poleceniu, kolejno:

* Wybieramy te kolumny, które chcemy grupować, np. column1 i column2, a następnie kolumnę, na której chcemy zaaplikować funkcję agregującą (column\_3).
* Natomiast po słowach kluczowych GROUP BY, wybieramy te kolumny, po których ma następować grupowanie.

Załóżmy, że opierając się na poniższej tabeli, chcielibyśmy zsumować i wyświetlić, ile wynosi suma wszystkich płatności wykonanych przez każdego z uczniów z osobna (tabela ta jest powiązana relacją ze Students):

**Payments**

| id | student\_id (FK) | amount |
| --- | --- | --- |
| 1 | 1 | 750 |
| 2 | 3 | 750 |
| 3 | 2 | 900 |
| 4 | 1 | 750 |
| 5 | 3 | 900 |

Wówczas zapiszemy poniższe zapytanie:

| SELECT student\_id, SUM(amount) FROM Payments GROUP BY student\_id; |
| --- |

W rezultacie otrzymamy poniższy wynik:

| id | student\_id | sum |
| --- | --- | --- |
| 2 | 1 | 1500 |
| 3 | 2 | 900 |
| 3 | 3 | 1650 |

Aby zmienić wyświetlaną nazwę, obecnie kolumny sum, na *total\_amount* możemy użyć słowa kluczowego AS i zapisać:

| SELECT student\_id, SUM(amount) AS total\_amount FROM Payments GROUP BY student\_id; |
| --- |

Chcąc natomiast zliczyć, ile razy każda z osób sumarycznie dokonała wpłat, użyjemy COUNT i zapiszemy:

| SELECT student\_id, COUNT(student\_id) FROM Payments GROUP BY student\_id; |
| --- |

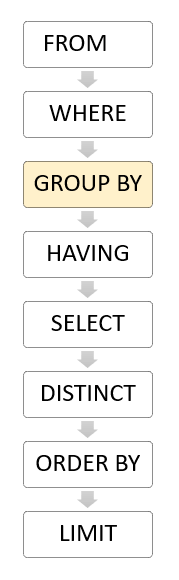
Wynikiem będzie poniższy output:

| id | student\_id | count |
| --- | --- | --- |
| 2 | 1 | 2 |
| 3 | 2 | 1 |
| 3 | 3 | 2 |

1. **HAVING**

Będąc już przy temacie związanym z GROUP BY, musimy koniecznie zatrzymać się przy operacji **HAVING**. Umożliwia ona określenie warunku, po którym mają być agregowane dane (przy użyciu poznanych już choćby operacji SUM, COUNT).

W poleceniu - HAVING występuje zawsze po GROUP BY (możemy to zobaczyć na ciekawym diagramie prezentującym kolejność umieszczanych operacji w zapytaniu SQL):



Źródło: [www.postgresqltutorial.com/](http://www.postgresqltutorial.com/)

Wróćmy na chwilę do zapytania, gdzie sumowaliśmy przychody od każdego klienta. Załóżmy, że chcielibyśmy teraz nieco zmienić polecenie i dodać warunek określający, iż wyświetlane mają być tylko te osoby, które sumarycznie zapłaciły więcej niż 1000 (zł).

Wówczas polecenie przyjmie postać:

| SELECT student\_id, SUM(amount) AS total\_amount FROM Payments GROUP BY student\_id HAVING SUM(amount) > 1000; |
| --- |

**HAVING vs WHERE**

Jeżeli zastanawiasz się, dlaczego w powyższym przykładzie, tworząc warunek, nie wykorzystaliśmy WHERE, tylko HAVING, to istnieje na to proste uzasadnienie. Oba słowa kluczowe służą do sprawdzenia warunku, jednak WHERE wykorzystywany jest, gdy odnosimy się do rekordów, a HAVING do całych grup (powstałych po wywołaniu niektórych z funkcji agregujących).

Zapamiętanie powyższej zasady pomoże Ci skutecznie manipulować poleceniami SQL.

**Zakończenie**

To tyle, jeśli chodzi o zapoznanie się z podstawami, jak i nieco bardziej zaawansowanymi konceptami SQL i PostgreSQL. Szkolenie wyszło dość obszerne, dlatego konieczne jest, abyś na spokojnie poświęcił odpowiednio dużą ilość czasu na naukę i zapoznanie się ze wszystkim wyżej przedstawionymi konceptami.

Najważniejsze informacje, jakie chciałbym, abyś wyciągnął z tego szkolenia to:

* Jakie wyróżniamy rodzaje baz danych?
* Jakie są najpopularniejsze implementacje relacyjnych baz danych?
* Omów przykładowe polecenia DML, DDL, DQL
* Omów różnice między JOINA-mi
* Czym jest HAVING, WHERE, COUNT, SUM?

Oraz abyś potrafił…

* Efektownie obsługiwać pgAdmina oraz psql
* Znał polecenia charakterystyczne dla PostgreSQL (np. listowanie dostępnych baz, tabel, nawiązywanie połączenia ze wskazaną tabelą)