POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Michał Cywiński

Nr albumu 15027

Generyczne API sieciowe oraz system do zarządzania bazami danych

Praca magisterska napisana

pod kierunkiem:

dr inż. Mariusz Trzaska

Warszawa, czerwiec 2018

Spis treści

Streszczenie

Niniejsze praca poświęcona jest zdefiniowaniu wymagań i opracowania prototypu aplikacji oferującej generyczne API sieciowe do zarządzania bazami danych. Autor dokonał przeglądu istniejących na rynku rozwiązań, na podstawie których wyspecyfikowane zostały wymagania dla prototypu.

W dalszej części prezentowane są funkcje i interfejsy udostępniane przez nowe rozwiązanie. Proponowane podejście wyróżnia zastosowanie mechanizmu wtyczek oraz generycznego REST API, których działanie w ramach zaimplementowanego protypu zostało szczegółowo opisane.

Słowa kluczowe: Bazy danych, API, Interfejs, Generyczność, Wtyczki

1 Wprowadzenie

Niniejszy rozdział przedstawia cele pracy oraz opisuje proponowane rozwiązania dla generycznego narzędzia i API do obsługi baz danych.

1.1 Cel pracy

Celem niniejszej pracy jest opracowanie wymagań i koncepcji narzędzia, które pozwoli na interakcję z dowolnym Systemem Zarządzania Bazami Danych, będzie otwarte na możliwości jego dalszego rozwoju oraz zapewni elastyczne API do komunikacji z zewnętrznymi aplikacjami.

1.2 Przyjęte rozwiązania

W celu zapewnienia otwartości na zmiany oraz różne rozwiązania bazodanowe, koncepcja uwzględnia zastosowanie systemu tzw. *wtyczek* do obsługi baz danych oraz *REST API* zapewniające możliwość współpracy z dowolną aplikacją zewnętrzną.

Do implementacji prototypu bazującego na obranych założeniach wybrano platformę .NET Core oraz szereg narzędzi z ekosystemu języka JavaScript, w tym framework Vue.js z komponentem CodeMirror.

1.3 Rezultaty pracy

Rezultatem pracy jest opracowanie kluczowych wymagań dla narzędzia do zarządzania bazami danych, którego architektura na wielu poziomach będzie otwarta dla programisty. W oparciu o zdefiniowane wymagania stworzony został prototyp złożony z backendu w formie aplikacji webowej, która zapewnia REST API, front-endu wykorzystującego go i pełniącego podstawowy interfejs użytkownika oraz z przykładowej aplikacji mobilnej dla systemu operacyjnego iOS, która prezentuje przykładowe możliwości integracji rozwiązań zewnętrznych.

1.4 Organizacja pracy

Rozdział 2 niniejszej pracy poświęcony jest przeglądowi istniejących na rynku aplikacji oraz przedstawieniu ich zalet i wad.

W rozdziale 3 zdefiniowane są wymagania funkcjonalne i niefunkcjonalne dla nowego rozwiązania oraz zostaje zaproponowana jego modułowa architektura.

Kolejne rozdziały mają na celu przedstawienie szczegółów związanych z implementacją prototypu. W rozdziale 4 przybliżone zostają narzędzia i technologie wykorzystane w części implementacyjnej nowego narzędzia. Na początku rozdziału 5 omówione są kluczowe funkcjonalności aplikacji, a następnie omówione są najważniejsze fragmenty kodu źródłowego odpowiedzialnego za ich realizację.

W końcowym rozdziale podsumowano osiągnięte rezultaty oraz przedstawieno dalsze plany rozwojowe dla opracowanego prototypu.

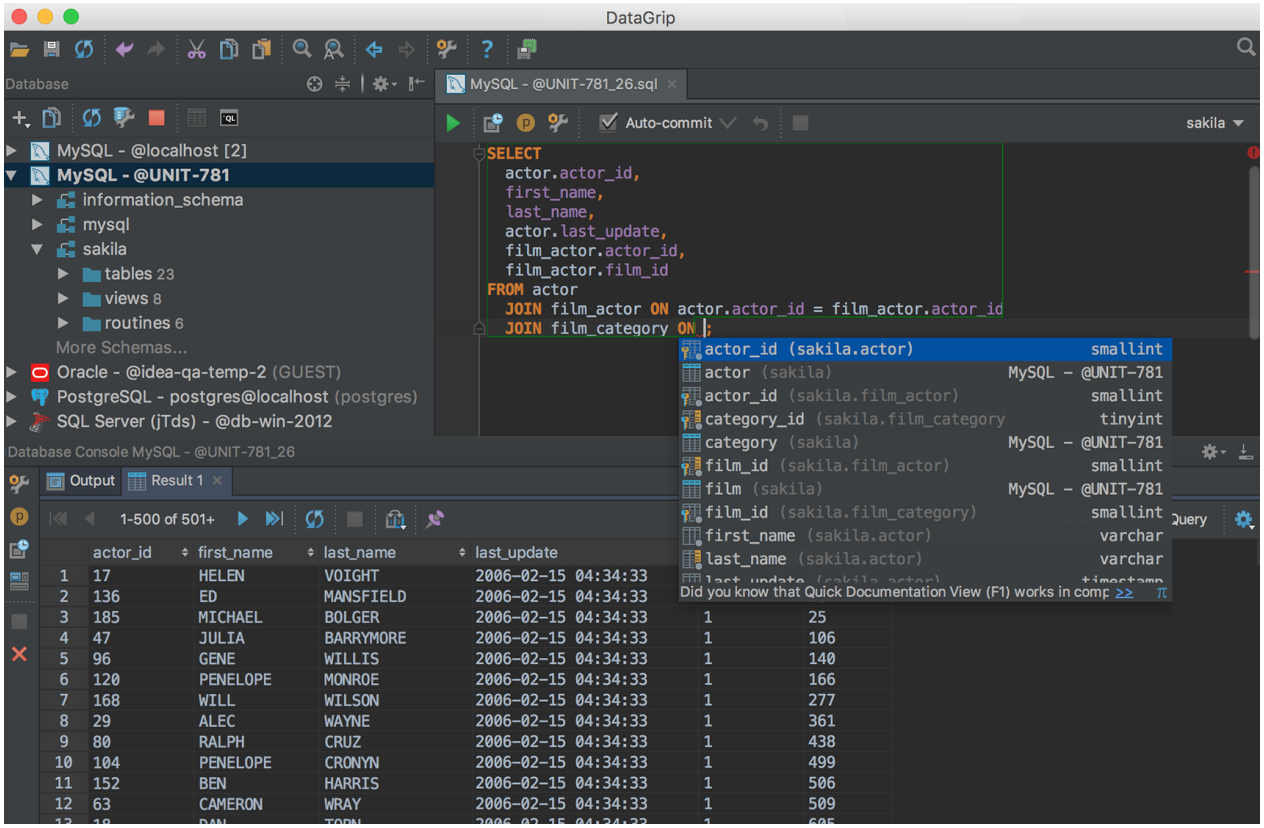
2 Istniejące systemy

Niniejszy rozdział poświęcony jest omówieniu możliwosci obecnego na rynku oprogramowania o zbliżonym zakresie funkcjonalnym. Omówienie istniejących już na rynku rozwiązań umożliwi zdefiniowanie wymagań funkcjonalnych dla rozwiązania autorskiego. Aplikacje służące zarządzaniu bazami danych opartymi o konkretny system zostały pominięte, ponieważ nie są one uniwersalne.

2.1 JetBrains DataGrip

JetBrains DataGrip jest komercyjnym, multiplatformowym narzędziem, które obsługuje najpopularniejsze SZDB (m.in.: Oracle, MySQL, SQL Server)[1]. Aplikacja ta jest zamkniętoźródłowa, jednak pozwala społeczności na rozszerzanie tego narzędzia o dodatkowe funkcjonalności dzięki API do tworzenia wtyczek.

Program pozwala na definiowanie wielu równoczesnych połączeń do różnych baz danych, a następnie wykonywanie na nich zapytań w danym dialekcie SQL. Editor aplikacji oferuje rozbudowany system podpowiedzi działający w oparciu o załadowany do pamięci schemat bazy danych z aktualnego połączenia oraz poprzez analizę zmiennych i obiektów bazodanowych zdefiniowanych w aktualnie edytowanym skrypcie. Ponadto, algorytmy zaszyte w aplikacji okazjonalnie sugerują potencjalne rozwiązania optymalizacyjne. Powyższe funkcjonalności zwizualizowane są na zrzucie ekranowym przedstawionym na rysunku 1.

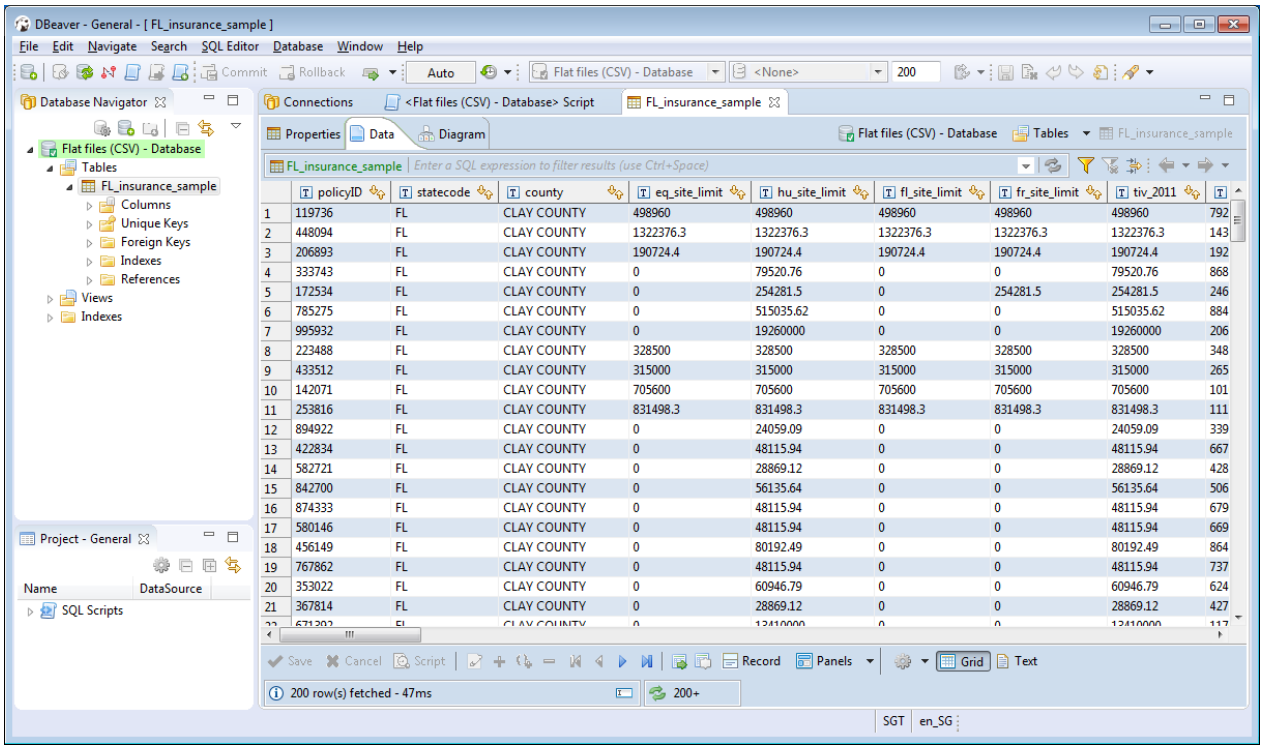


Rysunek 1: Interfejs programu DataGrip

Ułatwieniem dla użytkownika jest zintegrowana z rozwiązaniem obsługa wielu systemów kontroli wersji: Git, SVN, Mercurial oraz Perforce.

2.2 DBeaver

DBeaver to aplikacja Open Source na licencji Apache działająca na systemach operacyjnych z rodziny Windows, Linux, Solaris oraz macOS[2]. Program działa w oparciu o popularne środowisko programistyczne Eclipse i wymaga do uruchomienia środowiska Java w wersji 8.



Rysunek 2: Interfejs aplikacji DBeaver

Architektura DBeaver jest oparta o wtyczki, które dostarczają funkcjonalności interakcji z poszczególnymi SZBD. W tej chwili obsługiwane są najpopularniejsze relacyjne bazy danych oraz wprowadzona została obsługa najpopularniejszych baz NoSQL (MongoDB, Cassandra).

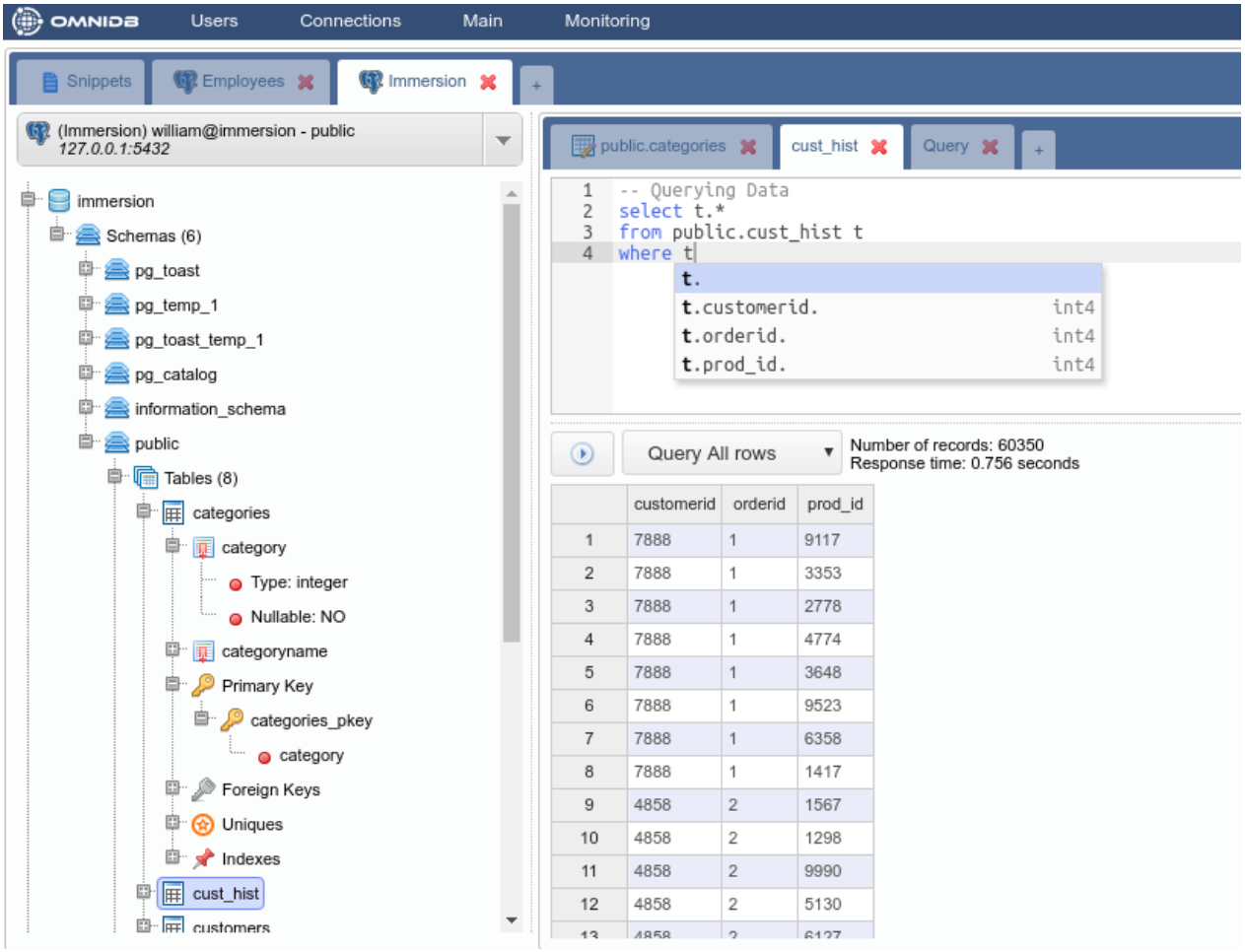
Rozwiązanie to pociąga za sobą konieczność manualnej instalacji sterowników do obsługi połączeń z SZBD przez użytkownika.

Oparcie DBeaver o środowisko Eclipse może ograniczać grupę potencjalnych użytkowników końcowych. Niektórzy programiści nie preferują domyślnego zachowania Eclipse oraz wyglądu jego interfejsu użytkownika (przedstawiony na rysunku 2) i z tego powodu mogą z góry odrzucić wszelkie aplikacje działające na bazie tego środowiska.

2.3 OmniDB

OmniDB jest aplikacją z interfejsem użytkownika realizowanym przez przeglądarkę internetową oraz rozwijaną na zasadach Open Source przy użyciu liberalnej licencji MIT[3]. Aplikację można używać w dwóch formach:

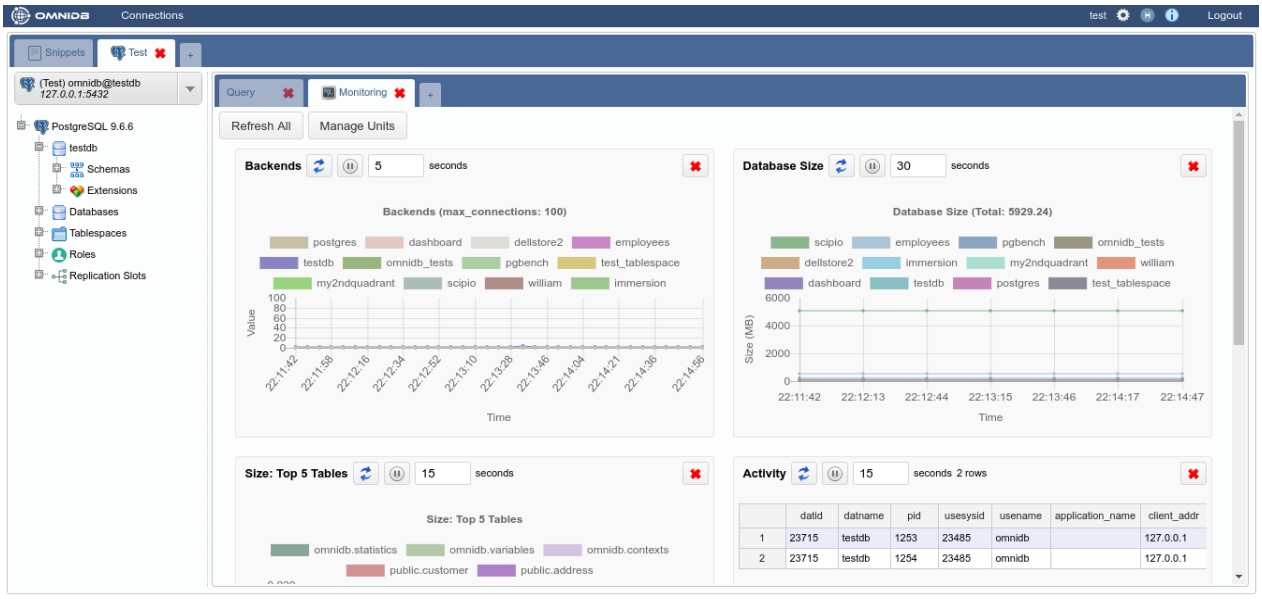
- wersji standalone posiadającej uproszczony interfejs użytkownika, która działa w formie zbliżonej do klasycznej aplikacji okienkowej,
- wersji hostowanej na serwerze, która posiada bardziej skomplikowany interfejs użytkownika oraz zezwala na działanie wielu użytkowników równocześnie.



Rysunek 3: Interfejs aplikacji OmniDB

Obecnie aplikacja posiada pełne wsparcie dla bazy PostgreSQL, jednak aplikacja jest rozwijana w sposób otwary, tak aby możliwe było wprowadzenia wsparcia dla innych SZBD. Autorzy planują wprowadzenie wsparcia dla większości wiodących relacyjnych baz danych, m. in. dla MySQL, Oracle oraz Microsoft SQL Server.

Od strony interfejsu, aplikacja wspiera system podpowiedzi w edytorze (widoczne na rysunku 3) oraz wizualizację planu zapytania.



Rysunek 4: Dashboard aplikacji OmniDB

Ciekawą funkcją aplikacji jest dashboard, z poziomu którego można monitorować stan baz danych, które zostały spięte z aplikacją. Możliwe jest śledzenie zużycia zasobów sprzętowych serwera bazy danych, jak również objętości tabel w postaci różnego rodzaju wykresów (widoczne na rysunku 4).

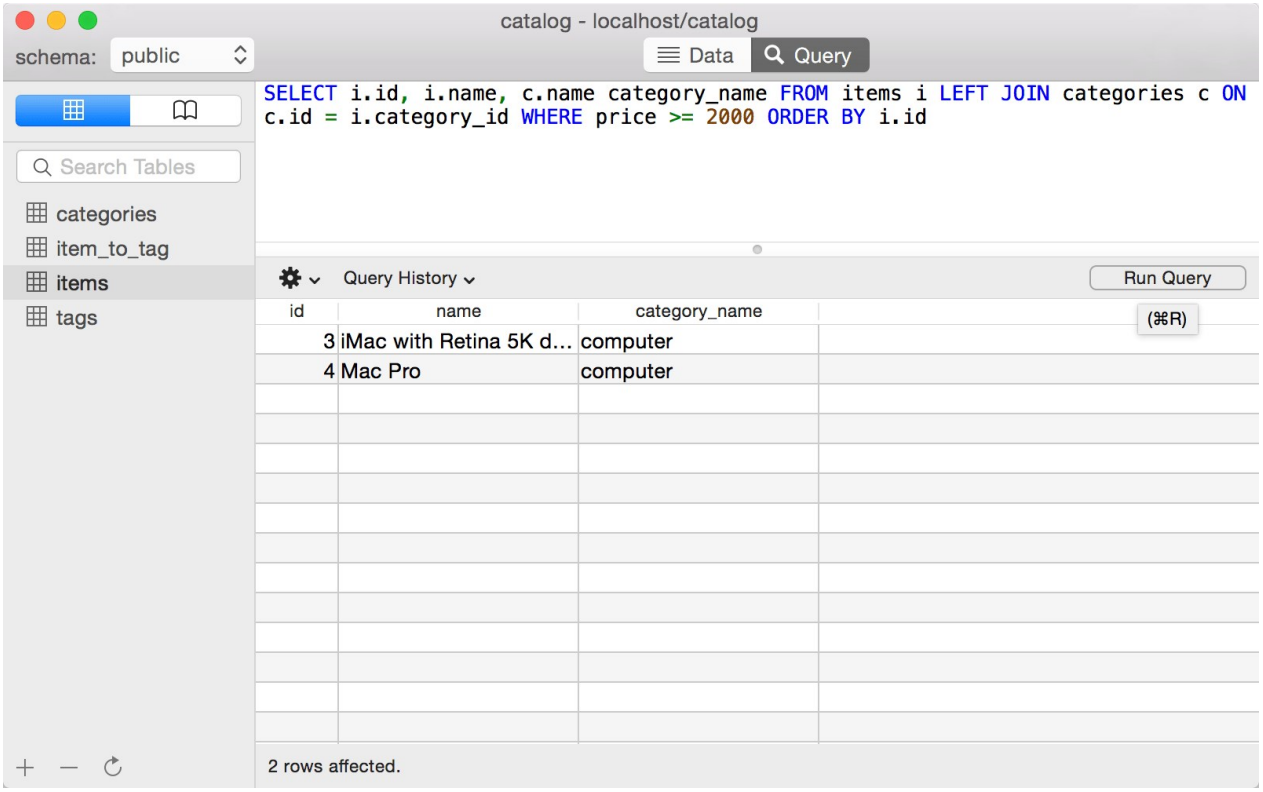
2.4 sqlectron

Aplikacja sqlectron jest najprostszym z przedstawionych rozwiązań rozwijanym w modelu Open Source na licencji MIT[4]. Obecnie wspiera tylko bazy danych PostgreSQL, My-

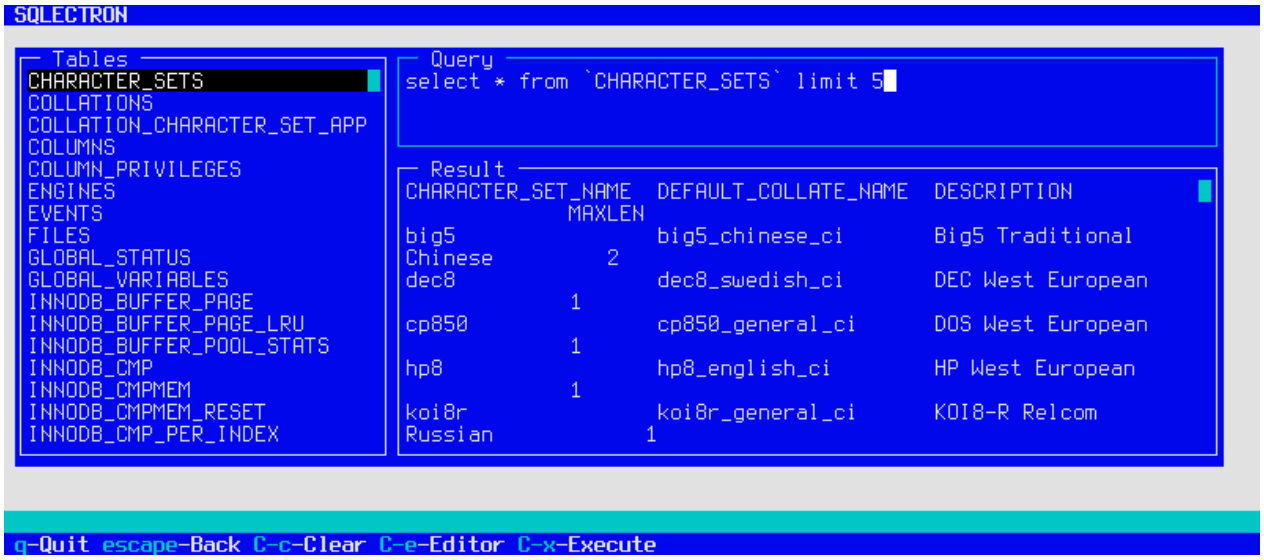
SQL, Microsoft SQL Server oraz SQLite, jednak pozwala o rozszerzenie jej o wsparcie innych SZBD.

Aplikacja rozprowadzana jest w dwóch wariantach, które dostarczają zbliżone możliwości funkcjonalne:

- sqlectron-gui - wersji graficznej (przedstawionq na rysunku 5) z pełnym wsparciem klawiatury i myszki, działającej samodzielnie w oparciu o framework Electron,
- sqlectron-term - wersji działającej z poziomu terminala (przedstawionq na rysunku 6)), wymagającej zainstalowanego Node.js.



Rysunek 5: Interfejs aplikacji sqlectron-gui



Rysunek 6: Interfejs aplikacji sqlectron-term

Osiągnięcie analogicznych możliwości obu wersji było możliwe dzięki zastosowaniu wspólnego backendu aplikacji napisanego w Node.js. Taka decyzja architektoniczna wpłynęła korzystnie na możliwości rozwoju tego programu: Praktycznie każdy programista posiada zainstalowany Node.js, dzięki czemu może w dowolnej chwili rozszerzyć program o wsparcie dodatkowego SZBD, rozbudować rdzeń aplikacji bądź poszerzyć możliwości jednego z dostępnych interfejsów użytkownika.

Od strony funkcjonalnej, aplikacja wspiera prosty system podpowiedzi w edytorze oraz tabelaryczne wizualizacje wykonywanych zapytań.

2.5 Podsumowanie

W powyższym przeglądzie przedstawione zostały rozwiązania najpopularniejszego (JetBrains DataGrip), jak i najbardziej zbliżone architektonicznie do rozwiązania docelowego (OmniDB).

Każde z powyższych rozwiązań oferuje względnie proste możliwości rozszerzenia go o

obsługę dodatkowych SZBD, posiada wygodny interfejs do przeglądania schematów bazy danych oraz zawarty jest w nich wygodny edytor oraz elementy interfejsu do wizualizacji wyników.

Niestety żadne z narzędzi nie posiada możliwości pracy grupowej w formie dzielenia się zapytaniami do baz danych oraz ich wynikami.

3 Propozycja nowego rozwiązania

Poznanie istniejacych rozwiązań, które zostały zaprezentowane w rozdziale 2, pozwoliło wyznaczyć najistotniejsze funkcjonalności, które powinny być zawarte w prototypie nowej aplikacji. Niniejszy rozdział poświęcony jest przybliżeniu kluczowych wymagań funkcjonalnych, których implementacja będzie wyróżnikiem rozwiązania autorskiego.

3.1 Wizja aplikacji

Praca zespołowa stała się standardem w dobie intensywnego wykorzystywania metodyk zwinnych w większości projektów związanych z wytwarzaniem oprogramowania. Jedną z głównych motywacji do stworzenia nowej aplikacji, było umożliwienie swobodnego dzielenia się współpracownikom zapytaniami tworzonymi w jezyku SQL z możliwością ich natychmiastowego wykonania. Rewolucja, która nadal trwa za sprawą pojawienia się smartfonów, sprawiła, że pracujemy nie tylko na komputerach osobistych, a tworzone treści mogą być konsumowane na wielu różnego rodzaju urządzeniach. Stąd kluczowym wymaganiem dla aplikacji jest udostępnienie interfejsu programistycznego, za pomocą którego możliwe stanie się integrowanie zewnętrznych aplikacji oraz urządzeń.

Mając na uwadze powyższe zmiany w branży IT, celem autora było opracowanie wymagań dla potencjalnego rozwiązania pozwalającego na interakcję z systemami zarządzania bazami danych i dzielenie się wynikami pracy, które będzie otwarte na działanie z dowolnymi bazami danych, a równocześnie nie będzie ograniczone na integracje rozwiązań zewnętrznych.

Przedstawione poniżej w sposób szczegółowy wymagania funkcjonalne i niefunkcjonalne w połączeniu z najważniejszymi cechami istniejących rozwiązań będą tworzyły użyteczną i wyróżniającą się na tle innych rozwiązań aplikację, która będzie otwarta na dalszy rozwój.

3.2 Wymagania funkcjonalne i niefunkcjonalne

Standardowe funkcjonalności, które pojawiają się w większości rozwiązań przedstawionych w rozdziale 2, oraz wizja nowego systemu opisana we wcześniejszej części tego rozdziału, pozwoliły określić najważniejsze wymagania funkcjonalne, które będą cechowały prototyp niniejszej aplikacji.

3.2.1 Możliwość pracy grupowej

Użytkownicy aplikacji powinni mieć możliwość dzielenia się zarówno połączeniami (tzw. *Connection Stringami*) z konkretnymi bazami danych, jak i pojedynczymi zapytaniami. Udostępnione elementy powinny być dla użytkowników odpowiednio wyróżnione, a połączenia niedostępne do edycji, ale umożliwiające ich pełne użycie we własnych zapytaniach. Udostępnione zapytania powinny w pośredni, niejawny sposób umożliwiać ich wykonanie na powiązanych z nimi połączeniami, co sprawi, że udostępniane będą tylko zapytania i możliwość uzyskania ich wyników.

3.2.2 Dostępność przez przeglądarkę internetową

Wymaganie związane z możliwościami pracy grupowej sprawia, że najrozsądniejszym rozwiązaniem będzie stworzenie aplikacji internetowej. Udostępnienie aplikacji w tej formie redukuje konieczność instalacji natywnego oprogramowania oraz konieczność zapewnienia jego aktualizacji.

3.2.3 Obsługa dowolnego systemu zarządzania bazami danych za pomocą mechanizmu wtyczek

Projekt aplikacji zakłada otwartość na interakcję z dowolnym systemem zarządzania baz danych. Funkcjonalność ta powinna być realizowana za pomocą mechanizmu wtyczek (tzw. *Plug-ins*). Wykorzystanie mechanizmu wtyczek pozwoli podmiotom zewnętrznym na

integrację z rozwiązaniem za pomocą udostępnionego interfejsu programistycznego. Wtyczka powinna być łatwa w instalacji, preferowalnie poprzez umieszczenie skompilowanej biblioteki w wyznaczonym katalogu.

Udostępnienie opcji integracji z aplikacją pozostawi społeczności Open Surce możliwość stworzenia dodatkowych wtyczek do rzadziej spotykanych baz danych.

3.2.4 REST API do integracji zewnętrznych aplikacji

Aplikacja powinna udostępniać interfejs programistyczny w celu wykorzystania własnych oraz udostępnionych zapytań i połączeń. Do tej pory programiści chcąc pozwolić na interakcję z bazą danych musieli pisać w tym celu własne API. Stworzenie takiego interfejsu w ramach nowego rozwiązania pozwoli na integrowanie zewnętrznych aplikacji oraz urządzeń. Praca programisty ograniczy się do napisania odpowiedniego zapytania w ramach aplikacji i udostępnieniu go. API powinno być stworzone w stylu architektury REST, który jest obecnie dominującym podejściem w branży, co potencjalnie zwiększy łatwość użycia go.

3.3 Architektura

Wymagania dotyczące nowego rozwiązania zostały przełożone na podstawowe założenia architektoniczne, które przedstawione są w niniejszym podrozdziale.

3.3.1 System wtyczek

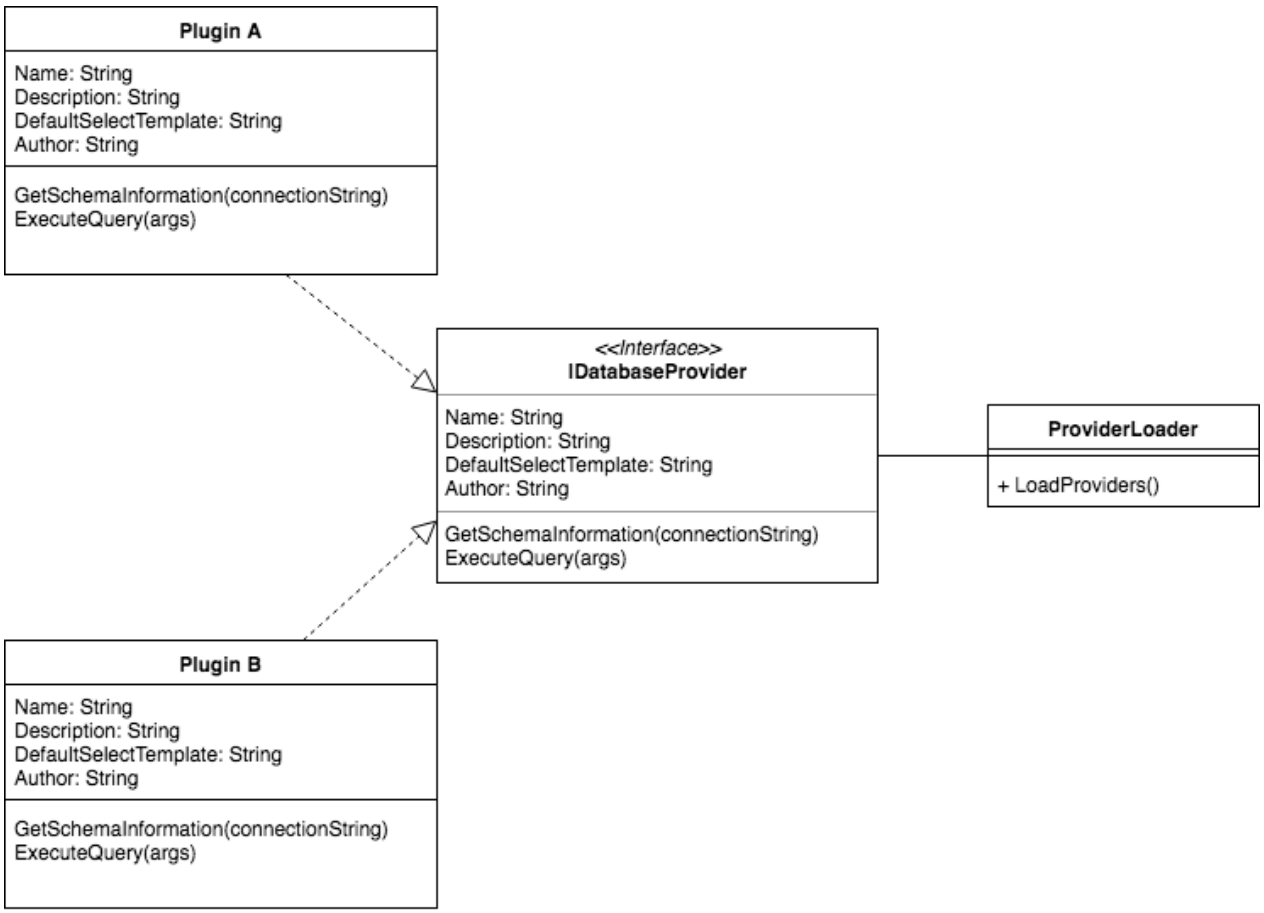
Podstawowym założeniem systemu jest interakcja z różnymi systemami zarządzania bazami danych przy pomocy mechanizmu wtyczek, których instalacja bądź deinstalacja powinna być możliwa bez konieczności rekompilowania aplikacji oraz bez potrzeby restartu serwera aplikacyjnego.

Wtyczka powinna implementować wyznaczony interfejs programistyczny, określający właściwości oraz akcje, które muszą być realizowane przez każdą wtyczkę do aplikacji. Aby

aplikacja wykryła wtyczkę, powinna ona być zlokalizowana w odpowiednim katalogu. Poprawnie zaimplementowana wtyczka udostępnia:

- metadane (nazwa wtyczki, opis, imię i nazwisko autora),
- metoda pobierania informacji o schemacie bazy danych,
- metoda do wykonywania zadanego zapytania,
- szablon zapytania SQL o pierwsze 1000 wierszy zadanej tabeli.

Koncepcja ta przedstawiona jest jest na diagramie klas (Rysunek 7).



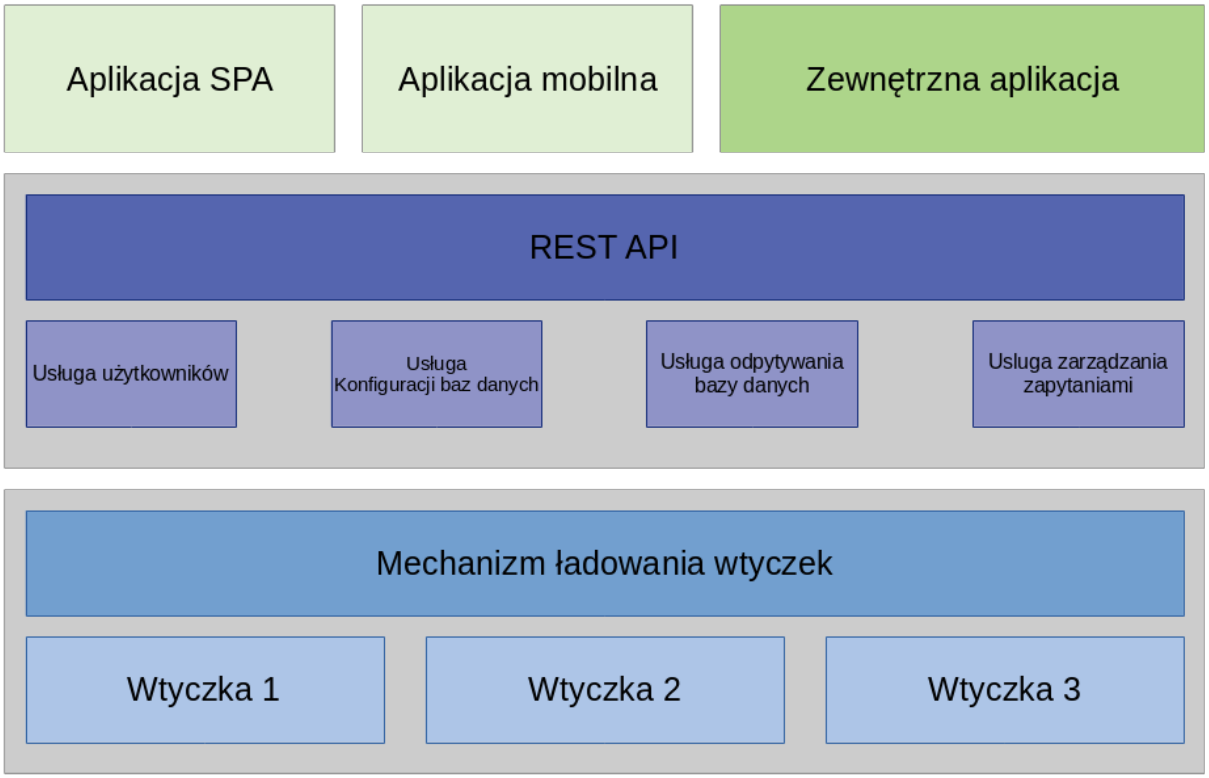
Rysunek 7: Diagram klas przedstawiający interfejs wtyczki i jego implementację, które następnie są agregowane i ładowane do wykorzystania w systemie.

Przy takich założeniach możliwy jest rozwój wtyczek bez znajomości reszty kodu aplikacji, co wpływa na łatwość ich tworzenia.

3.3.2 Model danych oraz backend

Aby zapewnić szerokie możliwości integracji aplikacji z rozwiązaniami zewnętrznymi oraz zapewnić łatwość utrzymania i rozwoju, postanowiono stworzyć backend, który dostarcza swoje funkcjonalności przy pomocy REST API. Dzięki takiemu rozwiązaniu możliwe jest ukrycie usług za wspólną warstwą abstrakcji, z której będą korzystały integrowane aplikacje oraz front-end napisany w formie Single Page Application.

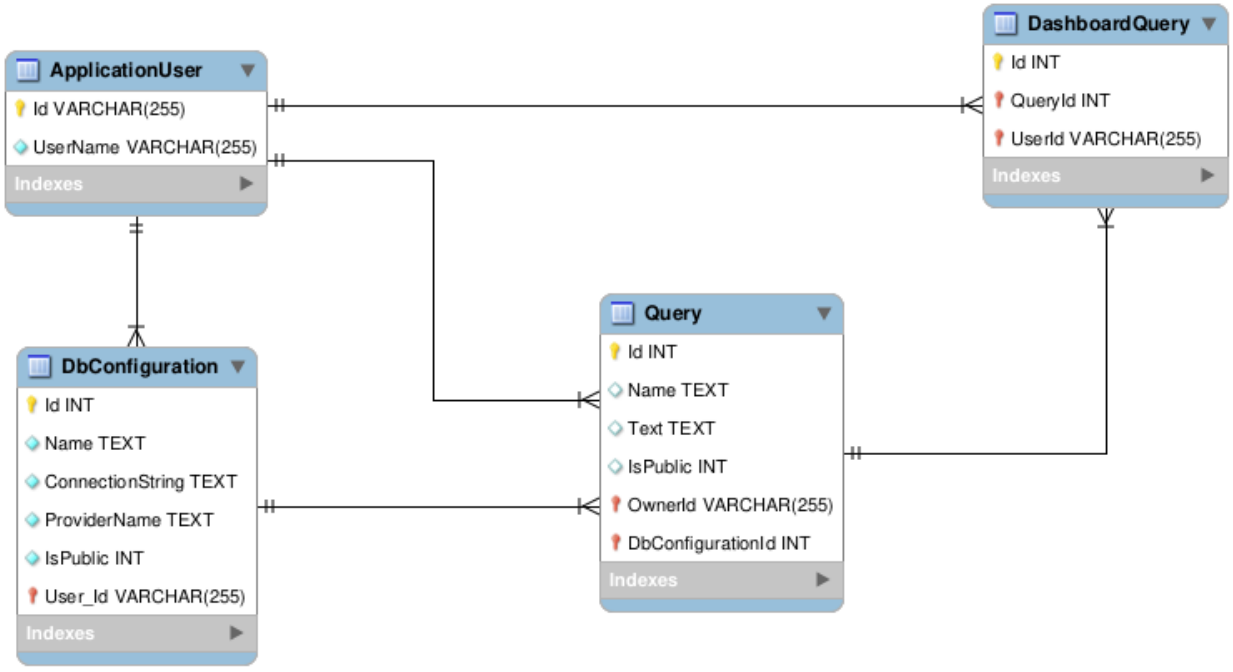
Rysunek 8 przedstawia uproszczoną architekturę backendu oraz współpracy poszczególnych komponentów.



Rysunek 8: Diagram przedstawiający architekturę aplikacji.

Proponowane podejście wpisuje się w Zasadę Pojedynczej Odpowiedzialności (ang. single responsibility principle[5]), według której każdy moduł bądź klasa oprogramowania powinna dostarczać dokładnie jedną funkcjonalność. W przypadku tego rozwiązania, wtyczki działają w sposób niezależny od siebie i są ładowane przeznaczonym do tego modułem. Poszczególne klasy usług korzystają z interfejsu programistycznego tego mechanizmu, aby dostarczać funkcjonalności wystawianych do REST API dla zewnętrznych aplikacji - w tym również aplikacji SPA, która pełni rolę oficjalnego front-endu rozwiązania.

Poniżej przedstawiony jest uproszczony diagram schematu bazy danych, w oparciu o który zbudowana jest aplikacja (Rysunek 9):



Rysunek 9: Diagram przedstawiający strukturę bazy danych aplikacji.

Backend działa w oparciu o model danych, w którym, występują 4 główne obiekty:

- tabela ApplicationUser - definicje użytkowników, którzy posiadają prawo zalogowania się do aplikacji,
- tabela DbConfiguration - definicje połączeń z bazą danych, gdzie zapisywane będą

connection stringi w relacji z daną wtyczką,

- tabela Query - zapisane zapytania użytkowników,
- tabela DashboardQuery - zapytania przypięte na dashboard aplikacji.

3.3.3 Warstwa interfejsu użytkownika

W ramach realizacji wspomnianej w poprzednim podrozdziale Zasady Pojedynczej Odpowiedzialności, front-end wydzielony jest do osobnej warstwy w postaci aplikacji SPA. Podejście to pozwala na budowę warstwy interfejsu użytkownika w całkowitym oderwaniu od części serwerowej i charakteryzują je następujące zalety:

- aplikacja jest ładowana tylko raz (podczas pierwszego żądania HTTP),
- dalsza nawigacja w aplikacji sprawia, że podmieniane są tylko jej aktualnie wyświetlane moduły/komponenty i nie jest konieczne kosztowne czasowo ładowanie od początku całego dokumentu HTML oraz dalsza jego intepretacja przez przeglądarkę internetową. Rozwiązanie to jest wyraźnie bardziej responsywne z perspektywy użytkownika,
- zostaje znacząco ograniczony transfer danych - aplikacja SPA wymaga tylko danych zwracanych przez API,
- ułatwiony zostaje rozwój aplikacji, w którym znacznie zostaje uproszczone tzw. mockowanie nieistniejących na daną chwilę funkcjonalności backendowych.

3.4 Podsumowanie

Aplikacja powinna być zaimplementowana z myślą o rozszerzalności dowolnych jej modułów. Architektura aplikacji powinna być zatem otwarta na możliwości jej rozszerzania na wszystkich poziomach. Wymaganie do zostanie zrealizowane przez utworzenie systemu ładowania wtyczek, udostępnienie REST API oraz separację front-endu w postaci aplikacji SPA.

4 Zastosowane technologie i narzędzia

Niniejszy rozdział poświęcony jest przybliżeniu technologii, które zostały użyte do zaimplementowania aplikacji oraz zestawu wykorzystanych narzędzi.

4.1 Technologie

Nowe rozwiązanie powinno być dostępne poprzez przeglądarkę internetową oraz dostępne na wiodących platformach sysemowych, zatem konieczny był wybór specyficznych technologii, o które zostanie oparta aplikacji. Wybrane technologie przedstawione są w tym podrozdziale.

4.1.1 .NET Core oraz ASP.NET Core

.NET Core w jest multiplatformową wersją środowiska uruchomieniowego .NET Framework[6], która jest dostępna dla systemów operacyjnych Windows, Mac oraz Linux. Środowisko .NET Core pozwala na pisanie programów w językach C#, F# oraz Visual Basic. W stosunku do standardowej wersji .NET Framework oferuje lepsze narzędzia konsolowe dla programistów oraz elastyczny model dystrybucji skompilowanych aplikacji, w którym możliwe jest dostarczanie programów razem z odpowiednio zapakowanym środowiskiem uruchomieniowym.

ASP.NET Core jest frameworkiem do tworzenia aplikacji internetowych z wykorzystaniem środowisk .NET Core lub .NET Framework. W stosunku do klasycznych wersji ASP.NET jest rozwiązaniem o wiele wydajniejszym oraz bardziej modularnym, opartym o wiele drobnych bibliotek dostępnych poprzez menadżer pakietów NuGet. ASP.NET Core charakteryzuje możliwość łatwego udostępnienia opartych o niego aplikacji zarówno na systemach z rodziny Windows (serwer IIS), jak również na systemach Linux (serwery Nginx, Apache). Możliwe jest również uruchomienie takiej aplikacji w formie tzw. *self-hostingu*, w której to wystawia ona sama lekki serwer WWW na określonym porcie.

Technologie te zostały wybrane ze względu na wysoką przenośność oraz liczne możli-

wości hostowania aplikacji opartych o nie, co wpisuje się w założenie wysokiej generyczności tego rozwiązania.

Zarówno .NET Core, jak rownież ASP.NET Core są rozwiązaniami otwartoźródłowymi.

4.1.2 SQLite

SQLite jest przenośną, niewymagającą instalacji, transakcyjną bazą danych[7]. Charakteryzuje ją obsługa z poziomu wymagającej jej aplikacji przy pomocy dedykowanej do wykorzystywanej technologii biblioteki.

SQLite upraszcza proces instalacji i pierwszego uruchomienia używającej jej aplikacji oraz jest oprogramowaniem działającym na praktycznie każdym systemie operacyjnym, co stanowi o jego świetnym dopasowaniu do wysoce przenośnej technologii .NET Core, o którą oparta jest aplikacja DataForge.

SQLite jest rozwiązaniem otwartoźródłowym, które równocześnie jest najczęściej używaną bazą danych na świecie oraz należy do pięciu najczęściej używanych modułów oprogramowania.

4.1.3 TypeScript

TypeScript jest językiem programowania będącym typowanym nadzbiorem języka JavaScript[8]. TypeScript jest językiem kompilowanym do języka JavaScript. Zezwala na używanie w jego ramach zarówno języka JavaScript, jak również na pisanie przy użyciu silnego typowania, które jest nieobecne w dynamicznych językach programowania.

Użycie silnego typowania ułatwia prace programistyczne oraz pozwala na wczesne wykrycie wielu błędów na etapie kompilacji.

4.1.4 Vue.js

Vue.js jest frameworkiem dla języka JavaScript, który jest zorientowany na łatwe tworzenie warstwy widoków w aplikacjach internetowych[9].

Biblioteka ta jest stworzona w taki sposób, aby jej główny moduł był łatwo adaptowalny w różnych rozwiązaniach niezależnie od stopnia zaawansowania projektu. Dodatkowe, opcjonalne moduły pozwalają na tworzenie tzw. *Single Page Applications*, czyli aplikacji webowych, które nie wymagają przeładowania przy wysyłaniu formularzy i zmianach routingu.

Obecnie Vue.js jest wiodącą biblioteką, która często jest wymienana obok rozwiązań takich jak Angular czy React.js. Społeczność programistów związana z frameworkiem tworzy dla niego liczne komponenty, co znacząco wpływa na przyjazność tego rozwiązania.

4.1.5 Biblioteka Axios

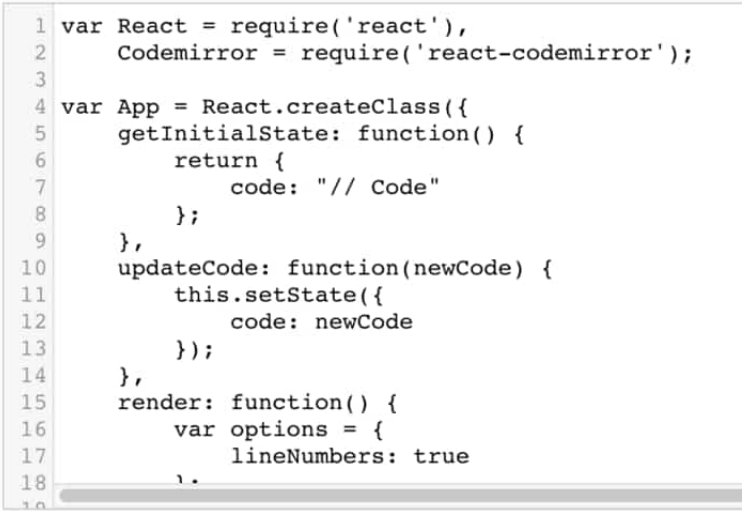
Axios to biblioteka języka JavaScript oparta na mechanie tzw. *Promise'ów*[10], która spełnia rolę klienta HTTP[11]. Axios wspiera standardowe akcje *GET*, *POST*, *PATCH* *PUT*, *DELETE* oraz oferuje programistom bogate API pozwalająca na skonfigurowanie niestandardowych nagłówków wykonywanych żądań i ich niestandardowych transformacji.

Biblioteka jest inspirowana API usługi *$http* z popularnego frameworka *AngularJS*, więc dla wielu programistów jej obsługa jest intuicyjna i stąd nie wymaga pracy z jej dokumentacją.

4.1.6 Komponent Codemirror

CodeMirror jest otwartoźródłowym komponentem zaimplementowanym w języku JavaScript, który służy do edycji fragmentów kodu źródłowego z poziomu aplikacji internetowej[12].

Komponent ten cechuje szerokie wsparcie dla wielu języków programowania, które zapewnia jego otwarte API. W ramach wsparcia języka SQL możliwe jest kolorowanie składni kodu oraz realizacja podpowiedzi kontekstowych dostępnych pod skrótem CTRL + Spacja.



```
var React = require('react'),
    Codemirror = require('react-codemirror');

var App = React.createClass({
    getInitialState: function() {
        return {
            code: "// Code"
        };
    },
    updateCode: function(newCode) {
        this.setState({
            code: newCode
        });
    },
    render: function() {
        var options = {
            lineNumbers: true
        };
```

Rysunek 10: Komponent CodeMirror

W ramach wspieranych dialektów wspierane są m. in. ANSI SQL, T-SQL, MySQL i inne. Wygląd komponentu przedstawiony jest na rysunku 10.

4.1.7 Komponent Handsontable

Handsontable to otwartoźródłowy komponent udostępniający funkcjonalność tabeli i arkusza kalkulacyjnego dla aplikacji internetowych[13]. Komponent posiada szerokie możliwości konfiguracyjne w zakresie dostosowania ustawień wyświetlania nagłówków kolumn, wierszy oraz prezentowanych danych. Jego zaletą jest również pełna współpraca z aplikacją Excel na poziomie kopiowania danych między arkuszami. Dla aplikacji DataForge komponent Handsontable będzie służył do wyświetlania rezultatów pomyślnie wykonanych zapytań.

Komponent Handsontable w podstawowej konfiguracji wygląda w sposób zbliżony do standardowych arkuszy kalkulacyjnych (rysunek 11).

A	B	C	D	E	F
	Maserati	Mazda	Mercedes	Mini	Mitsubishi
2009	0	2941	4303	354	5814
2010	5	2905	2867	412	5284
2011	4	2517	4822	552	6127
2012	2	2422	5399	776	4151

Rysunek 11: Komponent Handsontable

4.1.8 Komponent jsTree

JSTree jest komponentem opartym o bibliotekę *jQuery*, który pozwala na tworzenie interaktywnych drzew w aplikacjach internetowych[14].

Komponent ten można skonfigurować skonfigurować w praktycznie w dowolny sposób, zarówno od strony jego wyglądu (dostosowania ikon węzłów drzewa, możliwości jego stylowania) jak również od strony funkcjonalności (doładowywanie elementów drzewa w locie, bogate API ze zdarzenami).

W prototypie aplikacji DataForge, komponent będzie użyty do wizualizacji struktury wybranej bazy danych.

4.1.9 Swift

Swift jest językiem programowania ogólnego przeznaczenia rozwijanym pod opieką firmy Apple[15]. Został stworzony jako przyjazny programiście następca języków z rodziny C, w szczególności Objective-C. Obecnie język ten jest najlepiej wspierany na platformach macOS oraz iOS, na które można pisać oprogramowanie przy wsparciu środowiska programistycznego Xcode (przedstawione w dalszej części pracy).

Język Swift adresuje często spotykane w innych językach problemy:

- zmienne zawsze są zainicjalizowane,
- oferuje tzw. *Optionals*, które na poziomie kompilacji zapewniają weryfikację wystąpień wartości typu *nil*,
- obsługuje w naturalny sposób asynchroniczność,
- zarządzanie pamięcią odbywa się automatycznie jak w Javie i .NET.

Ze względu na powyższe powody, jak również ze względu na popularność systemu mobilnego iOS, Swift został wybrany jako język do implementacji aplikacji mobilnej integrującej się z rozwiązaniem głównym.

4.1.10 Biblioteka Alamofire

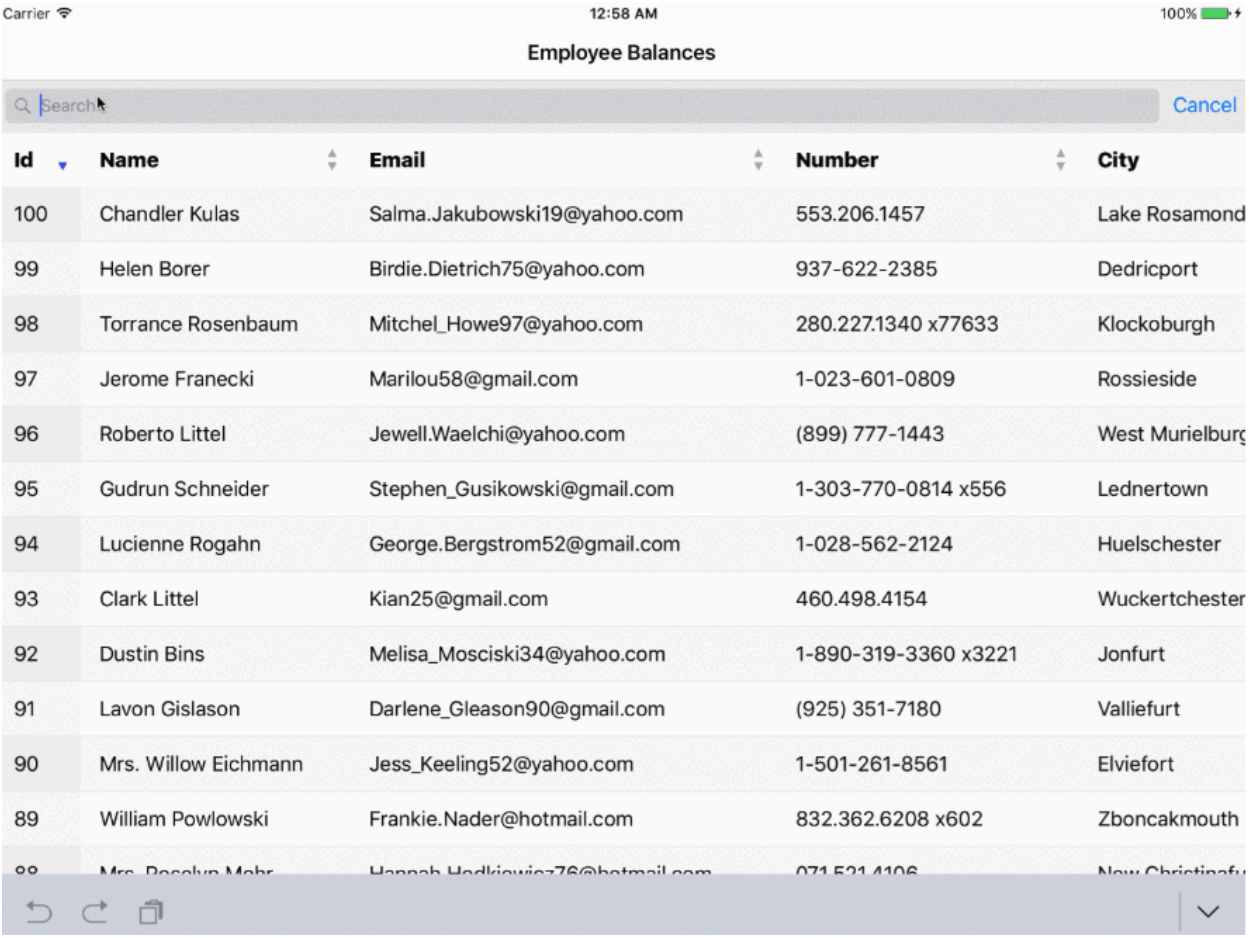
Biblioteka Alamofire jest klientem HTTP dla języka Swift na platformach iOS oraz MacOS[16]. Wyróżnia ją prostota użycia, która wynika z minimalnej potrzeby konfiguracji w stosunku do domyślnego klienta opartego o struktury *URLRequest* oraz *URLSession*.

Ponadto, biblioteka może dodatkowo wykonywać walidacje zwracanych przez serwer odpowiedzi w postaci weryfikacji kodów HTTP lub sprawdzenia nagłówków, co zwalnia programistę z konieczności walidowania tych danych własnoręcznie.

4.1.11 Komponent SwiftDataTables

SwiftDataTables to komponent dla języka Swift i systemu iOS[17], który rozszerza moliwości standardowego *UICollectionView* o sortowanie, paginację oraz wyszukiwanie wśród danych prezentowanych na komponencie (Rysunek 12).

Biblioteka w stosunku do zwykłego komponentu *UICollectionView* ułatwia szybkie podpięcie prezentowanych danych - nie wymaga od programisty uprzedniego zdefiniowania szablonu wyświetlanych komórek i czasochłonnej konfiguracji. Podstawowy scenariusz użycia tego komponentu zakłada podpięcie dwóch kolekcji elementów: kolumn oraz danych.



Rysunek 12: Przykładowe dane prezentowane z użyciem komponentu SwiftDataTables.

Komponent SwiftDataTables będzie zastosowany w celu prezentowania wyników zapytań na przykładowej aplikacji mobilnej dla systemu iOS.

4.2 Narzędzia

Ten podrozdział poświęcony jest opisaniu wykorzystanych narzędzi przy tworzeniu nowego rozwiązania.

4.2.1 JetBrains Rider

Rider jest zintegrowanym środowiskiem programistycznym firmy JetBrains dostępnym na wiodące systemy operacyjne: Windows, Linux oraz macOS[18].

Środowisko to zapewnia wsparcie dla różnych wersji środowiska uruchomieniowego .NET: .NET Core, .NET Framework, Mono oraz Xamarin. Wspierane są języki .NET'owe (C#, F#, Visual Basic), jak również różne dialekty SQL, co udało się dzięki integracji modułów narzędzia DataGrip. Ponadto, w aplikacji zintegrowano również środowisko Webstorm, dzięki czemu programista zyskuje pełne wsparcie dla narzędzi front-endowych: języków JavaScript oraz TypeScript, a także dla HTML i CSS.

JetBrains Rider ze względu na swoją uniwersalność, wykorzystany był zarówno przy budowie warstwy serwerowej rozwiązania, jak również przy tworzeniu warstwy prezentacji.

4.2.2 Xcode

Xcode jest środowiskiem programistycznym firmy Apple działającym na systemie operacyjnym macOS[19]. Głównym przeznaczeniem aplikacji jest udostepnienie programiście środowiska do programowania aplikacji dedykowanych dla systemów iOS oraz macOS w językach Swift, Objective-C, C oraz C++.

Środowisko składa się z kilku głównych modułów:

- edytora kodu ze wsparciem dla wygodnej refaktoryzacji i nawigacji,
- modułu do budowy interfejsów użytkownika dla systemów iOS oraz macOS,
- emulatora urządzeń iPhone oraz iPad.

Xcode zostało wykorzystane do stworzenia aplikacji dedykowanej dla systemu iOS.

4.2.3 NuGet

NuGet jest menadżerem pakietów dla języków związanych z platformą .NET[20]. Pozwala na zarządzanie zależnościami bez potrzeby ręcznego importu bibliotek **.dll* do projektu, oraz na ich łatwą aktualizację bądź deinstalację przy pomocy narzędzi w wbudowanych w IDE lub poprzez interfejs konsolowy.

Instalacja nowej biblioteki w projekcie polega na wprowadzeniu polecenia *dotnet add package nazwa_pakietu* do terminalu.

4.2.4 CocoaPods

CocoaPods to manedżer pakietów dla języków Swift oraz Objective-C[21]. Apple dla swoich języków nie dostarcza wbudowanych narzędzi do zarządzania należności, stąd powstało CocoaPods, jako projekt społeczności open source. CocoaPods jest oparty o język Ruby, stąd jego instalacja jest przeprowadzana za pomocą RubyGems[22].

Porócz głównego narzędzia obsługiwanego z poziomu linii komend, dostępna jest również graficzna aplikacja dla systemu MacOS, która pozwala w przyjaźniejszej formie zarządzać zależnościami.

4.2.5 npm

Npm jest otwartoźródłowym menadżerem pakietów dla ekosystemu języka JavaScript działającym w oparciu o środowisko uruchomieniowe Node.js[23]. Pozwala na zarządzanie zarówno pakietami front-endowymi jak i backendowymi dla aplikacji opartych o Node.

Instalacja wymaganych bibliotek sprowadza się do wykonania polecenia w terminalu *npm install nazwa_pakietu –save.*

4.2.6 webpack

Webpack jest tzw. *bundlerem* dla kodu języków frontendowych oraz statycznych zasobów aplikacji internetowej. Pozwala na kompilację oraz minimalizację rozmiaru wielu plików aplikacji w jeden plik wynikowy.

Webpack jest wysoce rozszerzalnym narzędziem dzięki użyciu tzw. *loaderów*, które pozwalają programistom na zdefiniowanie własnych, złożonych zadań.

4.2.7 TSLint

TSLint to narzędzie do statycznej analizy kodu[26], które zapewnia zwiększenie jakości kodu pisanego w języku TypeScript oraz zapobiega wystąpieniu najczęstszych usterek w kodzie aplikacji. Jest wspierany przez większość dostępnych edytorów używanych przy tworzeniu warstwy frontendowej aplikacji oraz możliwe jest wpięcie go do pipeline'u systemów realizujących Continuous Integration.

4.2.8 GitHub

GitHub jest serwisem, który zapewnia hosting dla projektów programistycznych używających system kontroli wersji Git[27]. Serwis GitHub oferuje bogate możliwości integracyjne, co ułatwiło stworzenie aplikacji DataForge.

5 Prototyp nowego rozwiązania

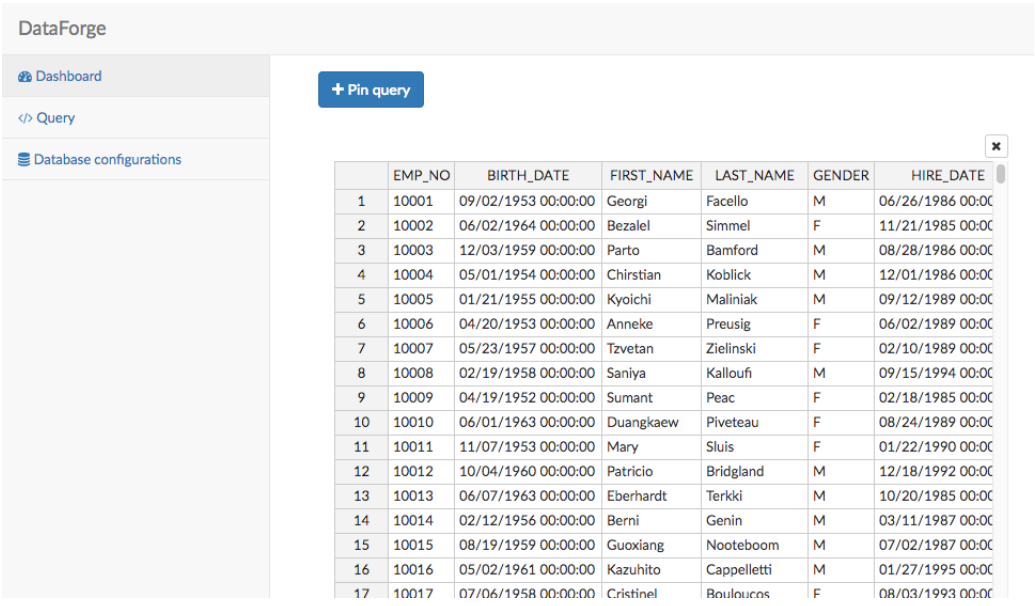
Niniejszy podrozdział poświęcony jest opisaniu opracowanego prototypu nowego rozwiązania. Pierwsze podrozdziały opisują odpowiednio kluczowe ekrany i funkcjonalności głównej aplikacji oraz przykładowej aplikacji mobilnej, natomiast w kolejnych podrozdziałach wyjaśnione są szczegóły implementacyjne prezentowanych możliwości prototypu.

5.1 Przykłady użycia aplikacji webowej

W tej części pracy opisane są najczęściej wykonywane czynności z poziomu aplikacji webowej.

5.1.1 Wyświetlanie dashboardu aplikacji

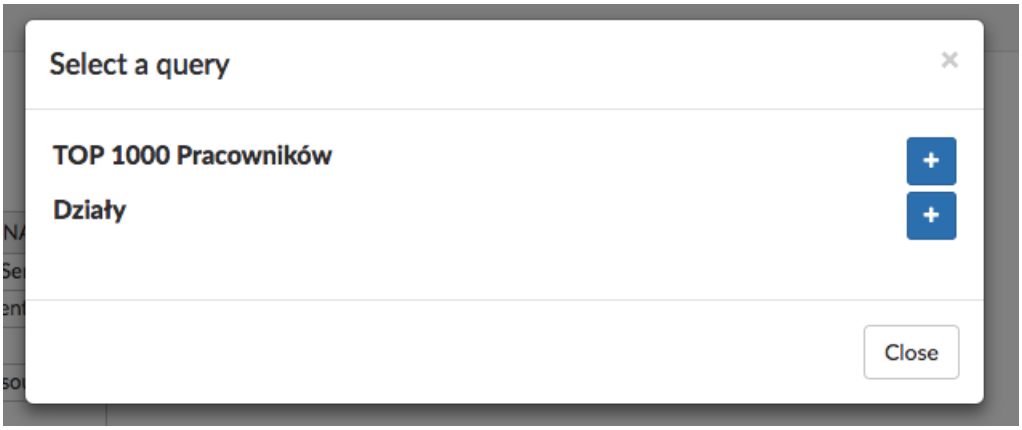
Większość aplikacji internetowych po zalogowaniu się wyświetla ekran początkowy, który krótko nazywany jest jako *dashboard*.



Rysunek 13: Dashboard aplikacji DataForge z przypiętym zapytaniem.

Aplikacja DataForge pozwala zdefiniować użytkownikom własny widok dashboardu poprzez udostępnienie możliwości przypięcia do 2 zapytań, których wyniki będą wyświetlane na dashboardzie (rysunek 13).

Zapytanie do wyświetlania na dashboardzie można wybrać poprzez naciśnięcie przycisku *Pin query* w prawym górnym rogu ekranu. Jego naciśnięcie wywoła okno modalne, w którym możliwy sjest wybór wcześniej zdefiowanego zapytania, które pojawi się na dashboardzie. Funkcjonalność ta jest przedstawiona na rysunku 14.



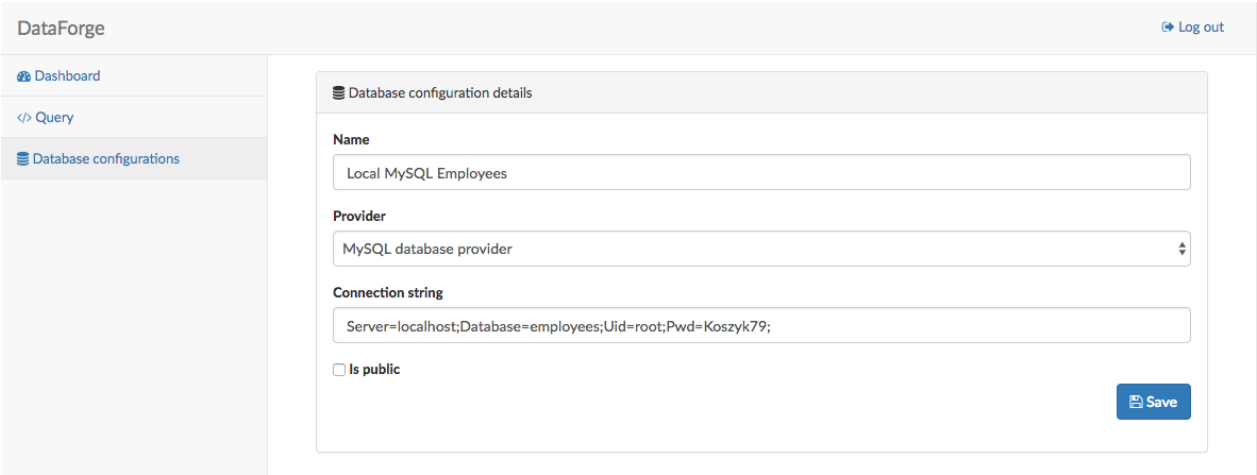
Rysunek 14: Wybór aplikacji do przypięcia na dashboard.

Po wyborze zapytania, zostaje ono przypięte do dashboardu, wykonane, a jego wynik jest zaprezentowany w formie tabelarycznej.

5.1.2 Definiowanie połączeń do baz danych

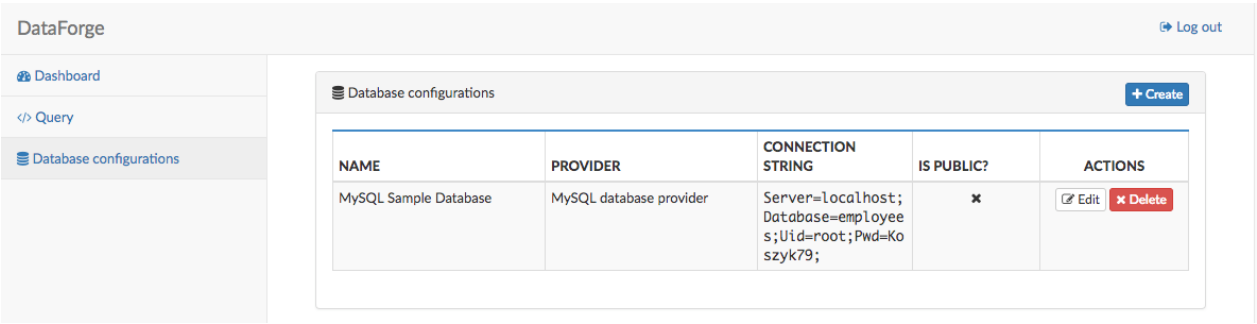
Rysunek 15 przedstawia ekran definiowania połączenia z bazą danych. Aby możliwe było korzystanie z baz danych, konieczne jest uprzednie zdefiniowanie do nich tzw. *connection stringa*, czyli ciągu tekstowego, który pozwala na określenie serwera, nazwy bazy danych oraz danych uwierzytelniających do otwarcia połączenia z tą bazą. Ponadto, użytkownik może na liście rozwijanej wybrać wtyczkę, która będzie używana do otwarcia połączenia z wykorzystaniem zadeklarowanego *connection stringa*.

Dodatkowo, w momencie deklarowania nowego połączenia z bazą danych możliwe jest określenie czy będzie ono dostępne publicznie. Ten wybór sprawia, że połączenie jest możliwe do użycia przez innych użytkowników, jednak nie będą znali wartości pola *connection string*.



Rysunek 15: Definiowanie nowego połączenia z bazą danych.

Po utworzeniu połączenia tą metodą, jest ono wyświetlane na liście zadeklarowanych połączeń (rysunek 16). Widok listy umożliwia szybki przegląd wtyczek użytych z danym połączeniem (kolumna *Provider*) oraz odpowiadających im *connection stringów*.

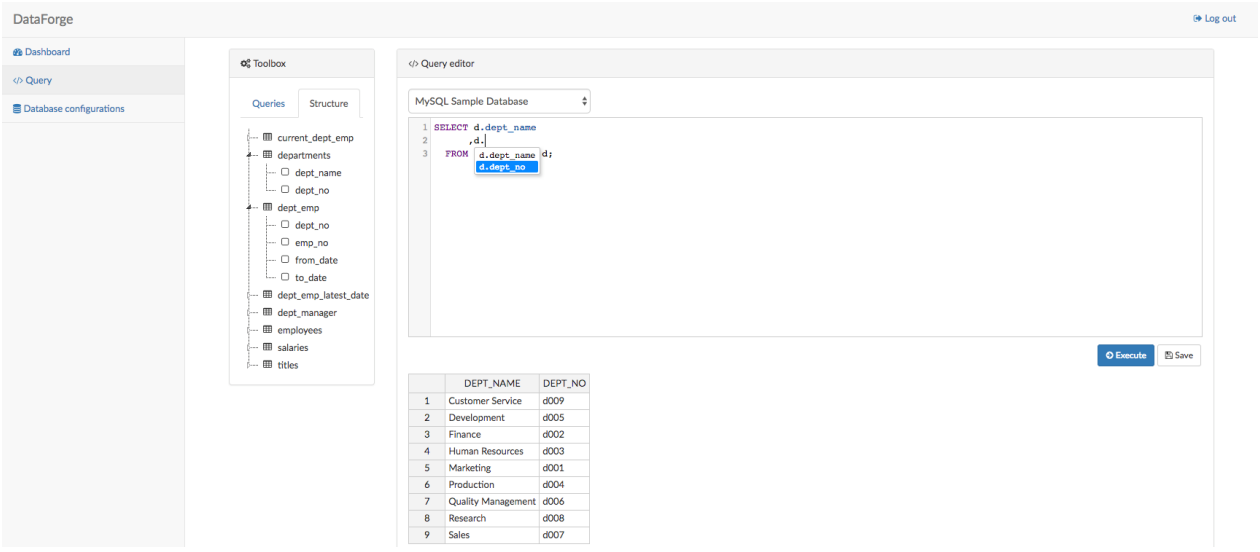


Rysunek 16: Lista zadeklarowanych połączeń z bazami danych.

Przyciski akcji po prawej stronie pozwalają na modyfikację parametrów połączenia lub jego usunięcie.

5.1.3 Wykonywanie zapytań i przeglądanie wyników

Wybór pozycji *Query* w menu aplikacji otwiera okno przedstawione na rysunku 17. W ramach tego widoku możliwy jest wybór zapisanego zapytania do wykonania lub edycji (zakładka *Queries*), bądź też przeglądanie struktury bazy danych, wynikającej z wybranego na liście rozwijanej połączenia (zakładka *Structure*).



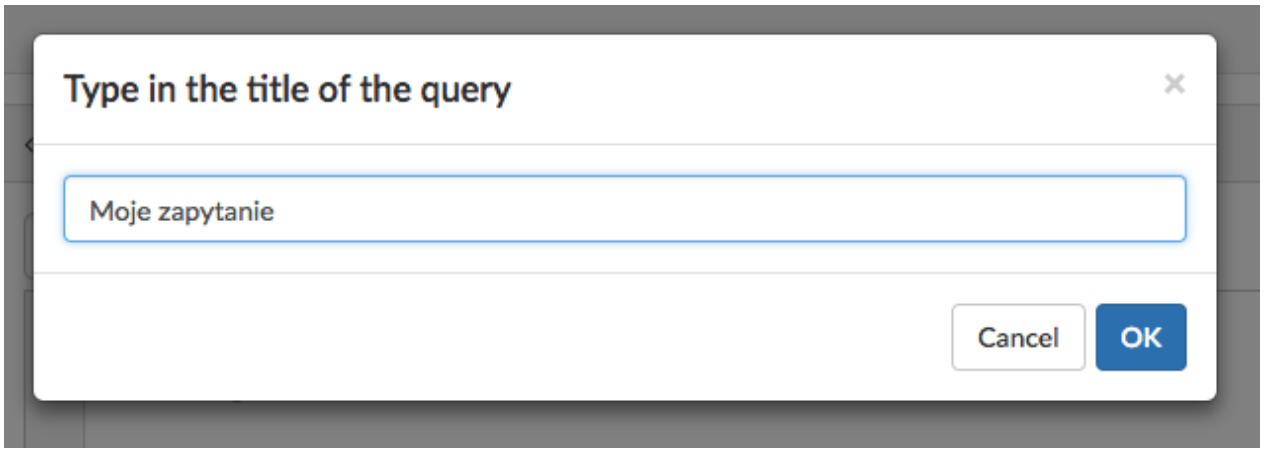
Rysunek 17: Widok edytora zapytań z podpowiedziami dla programisty, listy wyników oraz wizualizacji schematu bazy danych (po lewej).

W centralnej części ekranu zlokalizowany jest edytor kodu SQL z kolorowaniem składni języka. Podczas tworzenia kodu zapytania, możliwe jest wywołanie menu kontekstowego z podpowiedziami słów kluczowych, tabel i kolumn.

Gotowe zapytanie można wykonać poprzez kliknięcie w przycisk *Execute*. Podczas wykonania zapytania wyświetlana jest animacja sygnalizująca pracę aplikacji. Po wykonaniu zadanego zapytania zostaje wyświetlona tabela z wynikami pod obszarem edytora lub komunikat błędu zwrócony przez bazę danych w przypadku niepowodzenia.

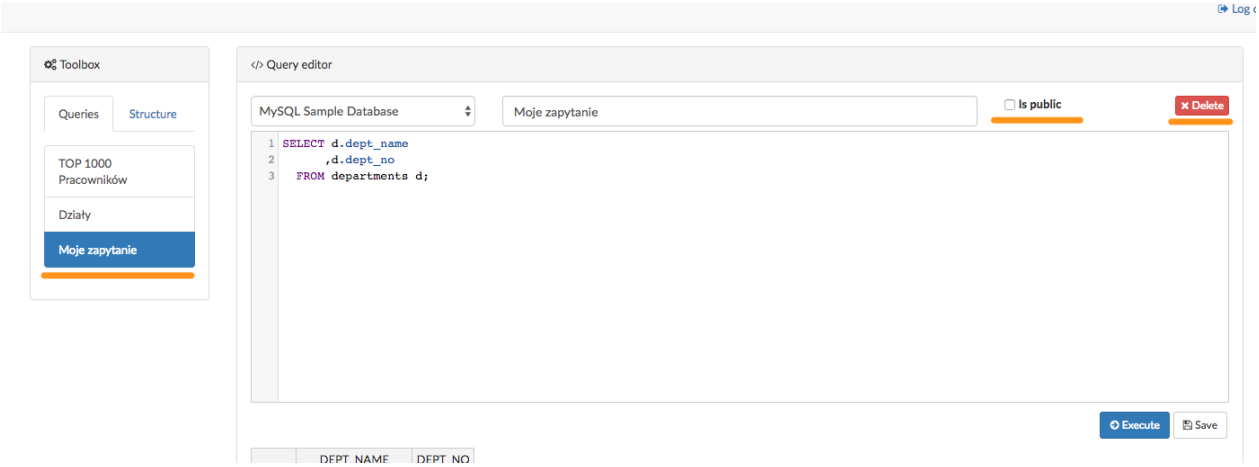
5.1.4 Zapisywanie i udostępnianie zapytań

Zapisanie kodu SQL stworzonego na ekranie edytora zapytań (rysunek 17) możliwe jest poprzez kliknięcie w przycisk *Save*. Akcja ta wywołuje wyświetlenie okna modalnego, w którym możliwe jest podanie nazwy dla zapisywanego zapytania (rysunek 18).



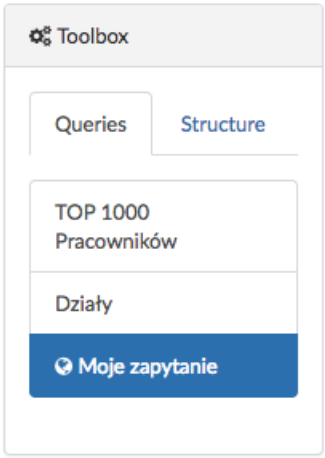
Rysunek 18: Widok okna modalnego do definiowania nazwy zapisaywanego zapytania.

Po pomyślnym zapisaniu nowego zapytania, zostaje ono wyświetlone i wybrane z listy dostępnych zaptytań. Dla zapisanego zapytania pojawiają się dwa nowe przyciski: przycisk do usuwania zapytania z listych zapisanych oraz pole typu *checkbox* do określania, czy zapisane zapytanie jest udostępnione dla innych użytkowników (podkreślone kolorem pomarańczowym na rysunku 19).



Rysunek 19: Widok fragmentu okna z zapisanym zapytaniem.

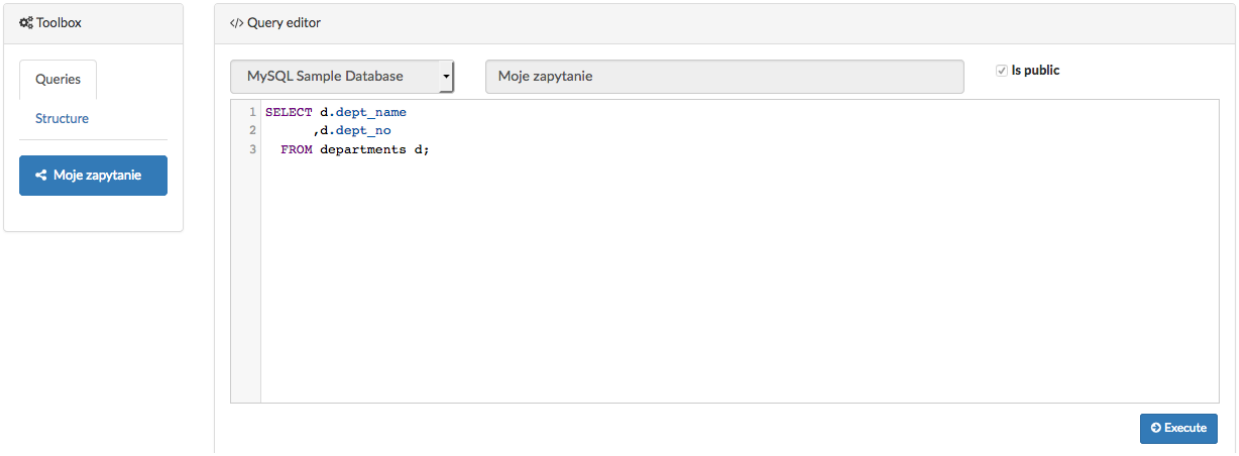
Na rysunku 20 przedstawione jest wyświetlanie zapytania, dla którego zaznaczono *checkbox* z opcją *Is public* z perspektywy właściciela zapytania - wyświetlona zostaje kula ziemska na lewo od nazwy zapytania.



Rysunek 20: Widok listy zapytań z zapytaniem udostępnionym.

Rysunek 21 przedstawia ekran do wykonywania zapytań z wybranym zapytaniem udostępnionym przez inną osobę. W tej perspektywie kontrolki na tym oknie są zablokowane, a

zapytanie na liście oznaczone jest alternatywną ikonką.



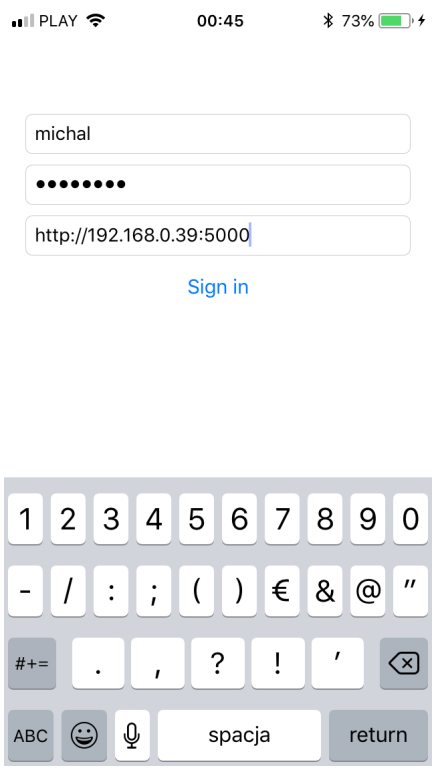
Rysunek 21: Widok fragmentu okna z zapytaniem udostępnionym przez inną osobę.

5.2 Przykładowa integracja z poziomu aplikacji na system iOS

W tej części pracy opisane są najczęściej wykonywane czynności z poziomu przykładowej aplikacji na system iOS. Demonstrowane funkcjonalności pokazują możliwe opcje integracji z API udostępnianym przez prototyp aplikacji DataForge.

5.2.1 Logowanie się do aplikacji mobilnej

Na rysunku 23 przedstawiony jest ekran logowania do aplikacji. Aby przejść dalej, użytkownik musi podać swój login, hasło oraz adres serwera aplikacji DataForge.

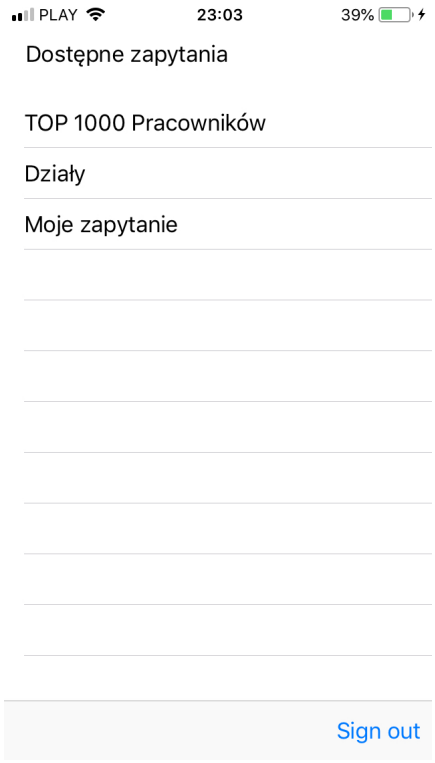


Rysunek 22: Widok ekranu logowania w aplikacji dla iOS.

Po zalogowaniu się, aplikacja zapisuje tzw. *token* do uwierzytelniania w API. Przy kolejnym uruchomieniu aplikacji następuje detekcja czy *token* jest już zapisany, a jeżeli jest, to aplikacja pomija krok wyświetlenia ekranu logowania.

5.2.2 Przeglądanie dostępnych zapytań

Po przejściu kroku logowania się do aplikacji, na ekran przedstawiony na rysunku 23 ściągana jest lista własnych oraz udostępnionych zapytań.

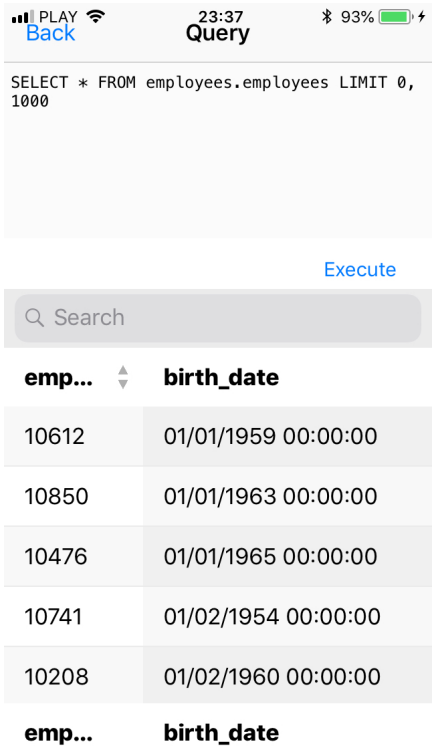


Rysunek 23: Widok ekranu z listą dostępnych zapytań w aplikacji dla iOS.

Z poziomu tego widoku możliwy jest wybór zapytania, którego kod chcielibyśmy obejrzeć lub wykonać, bądź też poprzez przycisk w prawym dolnym rogu ekranu istnieje możliwość wylogowania się.

5.2.3 Wykonywanie zapytań i przeglądanie wyników

Na rysunku 24 przedstawiony jest ekran podglądu kodu zapytania, który pojawia się po wyborze dokonanym na ekranie z rysunku 23.



Rysunek 24: Widok ekranu z kodem zapytania i jego wynikami w aplikacji dla iOS.

W pierwszej sekcji od góry, możliwy jest podgląd kodu zapytania, poniżej którego jest zlokalizowany przycisk *Execute*, który pozwala na wykonanie zapytania. Po wykonaniu zapytania, jego wyniki są wizualizowane w formie listy reprezentowanej przez komponent typu DataGrid, odpowiadającej kolejnym wierszom zwróconym przez API DataForge.

5.3 Istotne rozwiązania implementacyjne

W tej części pracy omówione są najważniejsze rozwiązania, które zastosowano w prototypie DataForge. Opisywane fragmenty kodu źródłowego zostały logicznie podzielone na kolejne podrozdziały, tak aby najpierw omówić backend aplikacji webowej, warstwę frontendu, a na końcu przykładową aplikację mobilną dla systemu iOS.

5.3.1 Backend - Biblioteka DataForge.Programmability

DataForge.Programmability jest biblioteką, z której korzysta zarówno główna aplikacja DataForge, jak również wszystkie implementacje wtyczek. W tej bibliotece zawarto główne klasy i interfejsy, które są wymagane do zaimplementowania własnej wtyczki do obsługi bazy danych:

- interfejs *IDatabaseProvider*,
- klasę *ExecuteQueryArgs*,
- klasę *ExecuteQueryException*,
- klasę *ExecuteResult*,
- klasę *SchemaInformation*.

Wymienione wyżej elementy biblioteki DataForge.Programmability są bezpośrednio związane z mechanizmem obsługi i ładowania wtyczek, którego uproszczona wizualizacja przedstawiona jest na rysunku 8. Działanie i praktyczne wykorzystanie tych komponentów opisane jest w dalszej części tego rozdziału.

5.3.2 Backend - Interfejs IDatabaseProvider

Interfejs *IDatabaseProvider* jest udostępniany przez bibliotekę DataForge.Programmability. Definiuje on metody oraz właściwości, które musi dostarczać konkretna implementacja wtyczki do obsługi bazy danych. Jego kod przedstawiony jest na listingu 1.

Listing 1: Interfejs *IDatabaseProvider*

```
public interface IDatabaseProvider
{
    string Name { get; }

    string Description { get; }
```

```
    string Author { get; }

    SchemaInformation GetSchemaInformation(string connectionString);

    ExecuteResult ExecuteQuery(ExecuteQueryArgs args);

    string DefaultSelectTemplate { get; }
}
```

Interfejs odpowiada w pełni założeniom przedstawionym na rysunku 7. Atrybuty *Name*, *Description*, *Author* są metadanymi opisującymi wtyczkę i odpowiadają odpowiednio nazwie wtyczki, krótkiemu opisowi oraz imieniu i nazwisko autora.

Interfejs wymaga także od programisty zdefiowania schematu domyślnej konstrukcji zapytania SQL dla pierwszych 1000 wierszy za pomocą pola *DefaultSelectTemplate*. Schemat tego zapytania, w którym nazwę tabeli należy zastąpić wyrażeniem *#table_name#*, używany jest później w aplikacji internetowej w akcji kontekstowej tabeli, która służy do szybkiego podglądu jej zawartości.

Metoda *GetSchemaInformation* powinna zwracać na podstawie podanego w argumencie parametru *connectionString* informacje o schemacie bazy danych zdefiniowanej w połączeniu bazodanowym.

Metoda *ExecuteQuery* wykonuje zadane zapytanie języka SQL dla wskazanego połączenia i zwraca jego wyniki. Argumenty wejściowe są przekazane w argumencie będącym instanacją klasy ExecuteQueryArgs z listingu 2.

Listing 2: Klasa *ExecuteQueryArgs*

```
public class ExecuteQueryArgs
{
    public string QueryText { get; set; }

    public string ConnectionString { get; set; }
}
```

Wyniki działania metody *ExecuteQuery* są zwracane w postaci instancji klasy *ExecuteResult*, której kod przedstawia listing 3.

Listing 3: Klasa *ExecuteResult*

```
public class ExecuteResult
    {
        public ExecuteResult(IEnumerable<string> resultColumns,
            IEnumerable<IEnumerable<string>> resultRows)
        {
            Columns = resultColumns;
            Rows = resultRows;
        }

        public IEnumerable<string> Columns { get; }

        public IEnumerable<IEnumerable<string>> Rows { get; }
    }
```

ExecuteResult posiada dwie kolekcje: *Columns* oraz *Rows*. *Columns* jest kolekcją zmiennych typu *string*, która reprezentuje nazwy kolumn, które zostały wybrane w wykonanym zapytaniu do bazy danych. Właściwość *Rows* zawiera kolekcje, które odpowiadają kolejnym wierszom i wartościom kolumn zwróconym przez zapytanie.

5.3.3 Backend - Interfejs IProviderLoader i jego implementacja

Aby wtyczka do obsługi bazy danych mogła zostać wykorzystana, należy ją najpierw załadować. W tym celu stworzono interfejs *IProviderLoader* oraz implementującą go klasę *ProviderLoader*, który udostępnia metodę *LoadProviders* służącą do ładowania wtyczek z katalogu *databaseProviders*. Kod metody *LoadProviders* przedstawia listing 4.

Listing 4: Metoda *LoadProviders*

```
public IEnumerable<IDatabaseProvider> LoadProviders()
{
    var databaseProviders = new List<IDatabaseProvider>();
    var pluginFiles = GetPluginFilenames();

    foreach (var filePath in pluginFiles)
    {
        Assembly assembly;
        try
        {
```

```
            assembly =
                AssemblyLoadContext.Default.LoadFromAssemblyPath(filePath);
        }
        catch (Exception ex)
        {
            _logger.LogError($"An error occured while loading assembly
                {filePath}.", ex);
            throw new ProviderLoadException($"Cannot load assembly from path
                {filePath}. See inner exception for details.", ex);
        }

        var assemblyProviders = assembly.GetTypes()
            .Where(t => !t.IsInterface)
            .Where(t => typeof(IDatabaseProvider).IsAssignableFrom(t))
            .Select(t => Activator.CreateInstance(t) as IDatabaseProvider);
        databaseProviders =
            databaseProviders.Union(assemblyProviders).ToList();
    }

    return databaseProviders;
}
```

Działanie metody *LoadProviders* zostało oparte o mechanizm tzw. *refleksji*[25][28]. Na początku działania, wywoływana jest prywatna metoda GetPluginFileNames, która w katalogu wtyczek wyszukuje wszystkie pliki o rozszerzeniu **.dll* i zwraca ich listę (Listing 5).

Następnie, w sekwencyjny sposób dla każdego pliku podejmowana jest próba załadowania go do pamięci operacyjnej i odnalezienia we wczytanej bibliotece klas, które implementują typ *IDatabaseProvider*. Klasy spełniające te warunki są następnie powoływane metodą *Activator.CreateInstance(t) as IDatabaseProvider*.

W ostatnim kroku, metoda zwraca wszystkie powołane w ten sposób instancje wtyczek.

Listing 5: Metoda *GetPluginFilenames*

```
IEnumerable<string> GetPluginFilenames()
{
    const string providersDirectory = "databaseProviders";
    string providersDirectoryPath = Path.Combine(AppContext.BaseDirectory,
        providersDirectory);
```

```
    try
    {
        if (!Directory.Exists(providersDirectoryPath))
            throw new ProviderLoadException("Providers directory does not
               exist.");
        return Directory.GetFiles(providersDirectoryPath)
            .Where(file => file.EndsWith(".dll"))
            .ToList();
    }
    catch (Exception ex)
    {
        _logger.LogError("An error occured while trying to load database
           providers.", ex);
        throw new ProviderLoadException("An error occured while trying
           access files located in databaseProviders directory. See inner
           exception for details.", ex);
    }
}
```

5.3.4 Wtyczka dla SQL Server - mechanizm wykonania zapytań

Wykonywanie zapytań we wtyczce dla SQL Server realizowane jest przez klasę *QueryExecutionEngine*, której metoda *ExecuteQuery* zwraca wyniki zapytania SQL dla zadanych argumentów wejściowych (Listing 6).

Listing 6: Metoda *ExecuteQuery*

```
public static ExecuteResult ExecuteQuery(ExecuteQueryArgs args)
{
    using (var sqlConnection = new SqlConnection(args.ConnectionString))
    {
        using (var sqlCommand = sqlConnection.CreateCommand())
        {
            sqlCommand.CommandText = args.QueryText;
            sqlCommand.CommandType = CommandType.Text;

            var resultColumns = new List<string>();
            var resultRows = new List<IEnumerable<string>>();
            try
            {
                sqlConnection.Open();
```

```
                var reader = sqlCommand.ExecuteReader();

                for (var i = 0; i < reader.FieldCount; i++)
                    resultColumns.Add(reader.GetName(i));

                while (reader.Read())
                {
                    var row = new List<string>();
                    for (var i = 0; i < reader.FieldCount; i++)
                        row.Add(reader[i].ToString());

                    resultRows.Add(row);
                }

                return new ExecuteResult(resultColumns, resultRows);
            }
            catch (Exception ex)
            {
                throw new ExecuteQueryException("An error occured while
                    executing a query. See inner exception for details.", ex);
            }
            finally
            {
                sqlConnection.Close();
            }
        }
    }
}
```

Wykonywanie zapytań oraz odczyt ich wyników, oparte są o klasę *DataReader* z pakietu *System.Data.SqlClient* dostępnego jako paczka NuGet. Metoda dla przekazanego connection string tworzy nowe połączenie bazodanowe oraz obiekt typu *SqlCommand* dla zadanego zapytania SQL. W następnym kroku następuje próba wykonania zapytania. Dla zwróconych wyników następuje odczyt nazw kolumn i ich zapis na listę, oraz utworzenie wierszy z wartościami kolumn przy użyciu obiektu typu *DataReader*[29].

Po udanym odczycie powyższych danych tworzony jest obiekt typu *ExecuteResult* (wspomniany we wcześniejszej części rozdziału, listing 3). W przypadku wystąpienia błędu, metoda rzuca wyjątek typu *ExecuteQueryException*, będącym częścią standardu narzucanego przez

bibliotekę *DataForge.Programmability*.

5.3.5 Wtyczka dla SQL Server - mechanizm pobierania informacji o schemacie bazy danych

Wtyczka do bazy danych SQL Server realizuje pobieranie informacji o schemacie bazy danych poprzez klasę *SchemaInformationProvider*. Klasa ta odwołuje się do systemowych widoków SQL Servera, z których możliwe jest wybranie informacji dotyczących nazw tabel, ich kolumn oraz typów zawartych w nich danych[30]. Dane te są zwracane w postaci instanacji klasy *SchemaInformation* (reprezentuje tabele oraz ich kolumny), która jest generowana w metodzie *GetSchemaInformation* (Listing 7).

Listing 7: Kluczowe fragmenty metody *GetSchemaInformation* w klasie *SchemaInformationProvider*

```
public static SchemaInformation GetSchemaInformation(string
    connectionString)
{
    // (...)

    sqlCommand.CommandText = "SELECT '[' + s.name + '].' + '[' + o.Name +
        ']' AS TableName, c.Name AS ColumnName, t.Name AS DataType" +
                             " FROM sys.columns c" +
                             " JOIN sys.objects o ON o.object_id =
                                c.object_id" +
                             " JOIN sys.types t ON t.user_type_id =
                                c.user_type_id" +
                             " JOIN sys.schemas s ON s.schema_id =
                                o.schema_id" +
                             " WHERE o.type = 'U'" +
                             " ORDER BY o.Name, c.Name";

    // (...)

    sqlConnection.Open();
    var reader = sqlCommand.ExecuteReader();
    var columns = new List<SchemaInformationColumn>();

    while (reader.Read())
```

```
    {
        columns.Add(new SchemaInformationColumn
        {
            ColumnName = reader["ColumnName"].ToString(),
            DataType = reader["DataType"].ToString(),
            TableName = reader["TableName"].ToString()
        });
    }

    reader.Close();

    var schemInformation = new SchemaInformation();
    var tableColumns = columns.GroupBy(sic => sic.TableName);
    var tables = new List<SchemaInformationTable>();

    foreach (var table in tableColumns)
    {
        tables.Add(new SchemaInformationTable
        {
            Name = table.Key,
            Columns = table.ToList()
        });
    }

    schemInformation.Tables = tables;
    return schemInformation;

    // (...)
}
```

Zapytanie SQL wybiera z widoków systemowych złączone dane, które zostaną użyte do zbudowania obiektu typu *SchemaInformation*. Kluczowym momentem przygotowania tych danych, jest zgrupowanie ich po nazwie tabeli za pomocą odpowiedniego wyrażenia LINQ, dzięki czemu możliwe jest stworzenie listy obiektów klasy *SchemaInformationTable* z odpowiadającymi im kolumnami.

5.3.6 Aplikacja webowa - Wizualizacja schematu bazy danych oraz konfiguracja podpowiedzi w edytorze kodu SQL

Schemat bazy danych jest wizualizowany w ramach funkcjonalności zaprezentowanej we wcześniejszej części pracy na rysunku 17. Funkcjonalność ta jest oparta o dane z backendu reprezentowane przez klasę *SchemaInformation*, która jest generowana za pomocą metody z listingu 7.

W celu skonstruowania wizualizacji schematu bazy danych, dane są pobierane z serwera aplikacji do frontendu, gdzie używane są do inicjalizacji wtyczki JSTree (komponent do tworzenia interaktywnych struktur drzewiastych). Pobieranie danych do drzewa następuje w momencie zmiany wybranego połączenia z bazą danych, co skutkuje zmianą na polu *dbConfigurationId*. Na listingu 8, który przedstawia najważniejszą część logiki komponentu *DbStructureComponent*, ta funckjonalność jest realizowana przez adnotację *@Watch* na funkcji *dbConfigurationIdChanged*.

Listing 8: Pobieranie informacji o schemacie bazy danych do frontendu aplikacji.

```
export default class DbStructureComponent extends Vue {
    private defaultSelectTemplate: string = '';

    @Watch('dbConfigurationId')
    dbConfigurationIdChanged(value: number) {
        this.createTree(value);
    }

    private createTree(dbConfigurationId: number) {
        dbConfigurationService.getSchemaInfo(dbConfigurationId)
            .then(schemaInfo => {
                AppState.Bus.$emit('dbStructure.schemaLoaded',
                    schemaInfo.tables);
                this.defaultSelectTemplate = schemaInfo.defaultSelectTemplate;

                let jsTreeData = schemaInfo.toJsTreeData();
                $('#db-structure-tree').jstree('destroy');
                $('#db-structure-tree').jstree({
                    core: { data: jsTreeData },
                    plugins: ['contextmenu'],
```

```
                contextmenu: {
                    // (...)
                }
            });
        })
        .catch(() => notyf.genericError());
    }
}
```

Aby dane działały dobrze z komponentem JSTree oraz aby miały odpowiednie ikony w strukturze drzewa, muszą zostać zmapowane funkcją *toJsTreeData*, której kod przedstawiony jest na listingu 9.

Listing 9: Transformacja danych do formatu komponentu JSTree przy pomocy funkcji *toJsTreeData*

```
toJsTreeData() {
    return this.tables.map(table => {
        return {
            text: table.name,
            icon: 'fa fa-table',
            type: 'table',
            children : table.columns.map(column => {
                return {
                    text: column.columnName,
                    icon: 'fa fa-square-o',
                    type: 'column'
                }
            })
        }
    });
}
```

W pierwszym kroku funkcja dla każdego elementu mapuje tabele do węzłów 1-wszego stopnia, oraz nadaje im stosowne ikony oraz ustawia cechę *type* na wartość *table*. W drugim kroku, kolumny tabel również są sprowadzane do formatu kompatybilnego z JsTree na poziomie węzłów 2-giego stopnia.

Zdarzenie zmiany wcześniej wspomianej właściwości *dbConfigurationId*, wywołuje również komunikat dla całej aplikacji, w którym przekazane zostają do ewentualnego przetworze-

nia dane dotyczące aktualnego schematu bazy danych. Dzięki tej implementacji wzorca projektowego Publish/Subscribe[31], którą wywołuje kod *AppState.Bus.$emit(dbStructure.schemaLoaded, schemaInfo.tables)* z listingu 8, komponent *QueryExecutionComponent* może załadować do biblioteki CodeMirror informacje dotyczące podpowiedzi tabel i kolumn w edytorze SQL (listing 10).

Listing 10: Załadowanie podpowiedzi tabel i kolumn do biblioteki Handsontable w ramach działania komponentu *QueryExecutionComponent*.

```
AppState.Bus.$on('dbStructure.schemaLoaded', (tables:
   SchemaInformationTable[]) => {
    let tablesObj = { };
    tables.forEach(table => {
        tablesObj[table.name] = table.columns.map(c => c.columnName);
    });

    this.queryEditorOptions.hintOptions.tables = tablesObj;
});
```

W komponencie *QueryExecutionComponent* następuje rejestracja obsługi zdarzenia *dbStructure.schemaLoaded*. W tym momencie następuje załadowanie nowych tabel i kolumn do modułu podpowiedzi języka SQL. Moduł podpowiedzi SQL biblioteki CodeMirror wymaga do działania odpowiednio skonstruowanego obiektu. W tym celu, kolekcja tabel jest rekonstruowana w obiekt, w którym pod właściwościami odpowiadającymi nazwom tabel są zawarte nazwy kolumn.

5.3.7 Aplikacja iOS - widok wykonywania zapisanych zapytań

Przykładowa aplikacja dla systemu iOS pozwala na wykonywanie zdefiniowanych wcześniej zapytań o zadanym ID (rysunek 24). Wykonanie zapytania inicjowane jest zdarzeniem naciśnięcia przycisku, które obsługuje funkcja *executeAction* (listing 11).

Listing 11: Obsługa wykonywania zapytań w aplikacji mobilnej po stronie *UIViewController*.

```
@IBAction func executeAction(_ sender: UIButton) {
    let spinner = UIViewController.displaySpinner(onView: self.view)
    queryService.Execute(queryId: query.Id, onSuccess: { result in
```

```
            self.columns = result.columns
            self.rows = result.rows

            let queryResultsViewController = (self.childViewControllers.filter {
                $0 is QueryResultsViewController }).first as!
                QueryResultsViewController
            queryResultsViewController.display(columns: self.columns, rows:
                self.rows)

            UIViewController.removeSpinner(spinner: spinner)
        }, onFailure: {
            AlertHelper.Error(self, title: "An error occurred!", message: "Could
                not execute the query...")
            UIViewController.removeSpinner(spinner: spinner)
        })
    }
```

Aplikacja odwołuje się do obiektu *queryService* i jego funkcji *Execute*, która prócz ID zadanego do wykonania zapytania po stronie backendu przyjmuje również tzw. *callbacki* - funkcje w języku Swift, które wykonywane są asynchronicznie w momencie ukończenia głównego zadania[32].

Zdefiniowane *callbacki* obsługują odpowiednio zdarzenie poprawnego wykonania zapytania, jak również wyświetlania komunikatu błędu. W przypadku sukcesu zapisywane są kolumny oraz wiersze zwrócone przez serwer aplikacji DataForge, a następnie odświeżany jest widok z tabelą wyników.

Do obsługi komunikacji z backendem użyto biblioteki Alamofire, której działanie również jest oparte o *callbacki*. Wywołanie wykonania zapytania w API przedstawia listing 12.

Listing 12: Komunikacja z metodą API do wykonywania zapytań.

```
func Execute(queryId: Int, onSuccess: @escaping (ExecuteResult) -> Void,
    onFailure: @escaping () -> Void) {
    let headers = apiService.GetAuthorizedRequestHeaders()
    let executeQueryUrl = GetApiUrl(segment: "query/" + String(queryId))

    Alamofire.request(executeQueryUrl, method: .post, headers: headers)
        .validate(contentType: ["application/json"])
        .validate(statusCode: [200])
```

```
        .responseJSON { response in

            guard let rawResult = response.result.value as? [String: Any]
                else {
                onFailure()
                return
            }

            let result = ExecuteResult()
            result.columns = rawResult["columns"] as! [String]
            result.rows = rawResult["rows"] as! [[String]]

            onSuccess(result)
        }
}
```

Działania podejmowane po zwróceniu kodu HTTP 200 i otrzymaniu odpowiedzi typu JSON obsługiwane są *callbackiem* przyjmującym parametr o nazwie *response*.

W dalszej części funkcji następuje próba odczytania odpowiedzi serwera przy pomocy mechanizmu tzw. *guard*-ów[33]. Jest to mechanizm, który pozwala bez używania wielokrotnie zagłębionych instrukcji *if-else* obsłużyć wartości typu *nil*, nierzutowalne do zadanego typu lub nieprzechodzące zadanych reguł walidacyjnych. Dzięki użyciu słowa kluczowego *guard*, zadeklarowana została zmienna, do której przypisano nieprzetworzone wyniki zapytania, natomiast w klauzuli *else* obsłużono sytuację wyjątku.

W końcowej części funkcji tworzony jest obiekt z wynikowymi kolumnami oraz wartościami wierszy, który jest przekazywany do *callbacku* zdefiniowanego na listingu 11.

6 Podsumowanie

Niniejszy rozdział poświęcony jest prezentacji zalet i wad opracowanego prototypu oraz przedstawieniu dalszych możliwości rozwoju tego rozwiązania.

6.1 Zalety i wady rozwiązania

Prezentowany prototyp rozwiązania realizuje stawiane przed nim wymagania. Jego interfejs użytkownika jest dostępny w formie aplikacji internetowej i wykorzystuje przygotowane REST API, co ograniczyło redundantny kod, który mógłby powstać w przypadku stworzenia klasycznej aplikacji napisanej np. z użyciem ASP.NET MVC, co stanowi o zalecie związanej z łatwością utrzymania kodu źródłowego. Ponadto, zastosowanie podejścia SPA pozwala na dalszy rozwój warstwy związanej z GUI w sposób praktycznie oderwany od zmian w backendzie.

Kolejną zaletą opracowanej aplikacji jest absolutna dowolność w wykorzystaniu jej przy integracji zewnętrznych systemów ze wskazanymi bazami danych. Bez pisania dedykowanego API oraz obiektów DTO możliwe jest udostępnianie zapytań do wykonania z wyłączeniem bezpośredniego dostępu do jakichkolwiek obiektów bazodanowych oraz *connection stringów*. Taka możliwość znacząco ogranicza czas, który programista musiałby poświęcić na wystawienie odpowiedniej metody w pisanym własnoręcznie API.

Dodatkowo, w przypadku braku wsparcia dla nietypowej bazy danych, możliwe jest doimplementowanie wtyczki, która zrealizuje taki dostęp. Włączenie obsługi takiego SZBD nie wymaga rekompilacji aplikacji, czy też jej restartu, co świadczy o otwartości proponowanej platformy.

Wadą stworzonego prototypu jest brak wsparcia dla systemów zarządzania bazami danych innych niż Microsoft SQL Server oraz MySQL. Stworzenie wtyczki, która realizuje wsparcie dla innych SZBD nie jest jednak czasochłonna i wiąże się z implementacją prostego interfejsu w języku C#.

6.2 Plany rozwojowe

W niedalekiej przyszłości udokumentowany prototyp aplikacji zostanie udostępniony społeczności *Open Source*. Krok ten jest naturalnym kierunkiem, który powinien znacząco przyspieszyć rozwój nowych funkcjonalności.

Kolejnym krokiem implementacyjnym, powinno być stworzenie nowych wtyczek do integracji z najpopularniejszymi systemami zarządzania bazami danych, takimi jak Oracle, PostgreSQL, czy też Firebird.

Potencjalną funkcjonalnością, która może zostać doceniona przez użytkowników, jest zaawansowana kontrola dostępu do udostępnionych połączeń oraz zapytań. W kolejnych wydaniach aplikacji, powinny zostać wydzielone uprawnienia odpowiednio do wykonywania zapytania, podglądu jego kodu oraz edycji.

Dalsze wersje programu powinny również oferować prosty mechanizm wersjonowania zapisanych zapytań.

6.3 Zakończenie

Narzędzia do obsługi baz danych powoli ewoluują i pozostają względnie zamknięte na możliwości pracy grupowej. Proponowane rozwiązanie oferuje rozwinięcie koncepcji programu do zarządzania bazami danych o możliwość pracy w zespole. Stworzono system, który również dobrze sprawdza się w roli narzędzia integracyjnego, co sprawia, że w przyszłości może się on stać ciekawą alternatywą w stosunku do istniejących na rynku aplikacji.

Bibliografia

[1] *JetBrains DataGrip* https://www.jetbrains.com/datagrip/, dostęp 2018-06-11

[2] *DBeaver* https://dbeaver.io/, dostęp 2018-06-11

[3] *OmniDB* https://omnidb.org/en/, odstęp 2018-06-11

[4] *Sqlectron* https://sqlectron.github.io/, dostęp 2018-06-11

[5] R. C. Martin. *Czysty kod*, Helion, ISBN 9788328302341

[6] S. Chiaretta, U. Lattanzi. *ASP.NET Core Succinctly*, strona 13, https://www.syncfusion.com/ebooks, dostęp 2018-06-01

[7] *About SQLite* https://sqlite.org/about.html, dostęp 2018-06-11

[8] *TypeScript - JavaScript that scales* https://www.typescriptlang.org/, dostęp 2018-06-11

[9] *Introduction - Vue.js* https://vuejs.org/v2/guide/, dostęp 2018-06-11

[10] *JavaScript Promises: an Introduction* https://developers.google.com/web/fundamentals/primers/pron dostęp 2018-05-17

[11] *axios: Promise based HTTP client for the browser and node.js* https://github.com/axios/axios, dostęp 2018-06-11

[12] *CodeMirror* https://codemirror.net/, dostęp 2018-06-11

[13] *Handsontable - JavaScript Spreadsheet Component For Web Apps* https://handsontable.com/, dostęp 2018-06-11

[14] *jsTree* https://www.jstree.com/, dostęp 2018-06-11

[15] *About Swift* https://swift.org/about/, dostęp 2018-06-11sw

[16] *Alamofire: Elegant HTTP Networking in Swift* https://github.com/Alamofire/Alamofire, dostęp 2018-06-11

[17] *SwiftDataTables: A Swift Data Table package, subclassing UICollectionView that allows ordering, searching, and paging with extensible options.* https://github.com/pavankataria/SwiftDataTables, dostęp 2018-06-11

[18] *Rider: Cross-platform .NET IDE by JetBrainsi* https://www.jetbrains.com/rider/, dostęp 2018-06-11

[19] *Xcode 10* https://developer.apple.com/xcode/, dostęp 2018-06-11

[20] *About NuGet* https://www.nuget.org/policies/About, dostęp 2018-06-11

[21] *CocoaPods.org* https://cocoapods.org/about, dostęp 2018-06-11

[22] *What is a gem?* https://guides.rubygems.org/what-is-a-gem/, dostęp: 2018-05-17

[23] M. Cantelon, M. Harter, T. J. Holowaychuk, N. Rajilich. *Node.js w akcji*, strony 416-419, Helion, ISBN 9788324696789

[24] *webpack* https://webpack.js.org/, dostęp 2018-06-11

[25] *How C# Reflection Works With Code Examples* https://stackify.com/what-is-c-reflection, dostęp: 2018-05-10

[26] *TSLint* https://palantir.github.io/tslint/, dostęp 2018-06-11

[27] *GitHub* https://github.com/about, dostęp 2018-06-11

[28] Praca zbiorowa. *Pro .NET 1.1 Remoting, Reflection, and Threading*, Apress, ISBN 9781430200253

[29] *Lesson 04: Reading Data with the SqlDataReader* https://csharp-station.com/Tutorial/AdoDotNet/Lesson04, dostęp: 2018-05-10

[30] *SQL Server System Views: The Basics* https://www.red-gate.com/simple-talk/sql/learn-sql-server/sql-server-system-views-the-basics/, dostęp: 2018-05-11

[31] *Publish/Subscribe* https://msdn.microsoft.com/en-us/library/ff649664.aspx, dostęp 2018-05-13

[32] *Fundamentals of Callbacks for Swift Developers* https://www.andrewcbancroft.com/2016/02/15/funda of-callbacks-for-swift-developers/, dostęp 2018-05-14

[33] *Guard Statements* https://thatthinginswift.com/guard-statement-swift/, dostęp 2018-05-14