P O L I T E C H N I K A Ś W I Ę T O K R Z Y S K A
Wydział Elektrotechniki, Automatyki i Informatyki

Grzegorz Bujak
Number albumu: 088943

Opracowanie i implementacja systemu Headless CMS z dodatkową możliwością modelowania warstwy danych przy pomocy natywnego SQLa

Praca dyplomowa
na studiach I-go stopnia
na kierunku Informatyka

Opiekun pracy dyplomowej:
dr inż. Mariusz Jacek Wiśniewski
Katedra Systemów Informatycznych

Konsultant pracy dyplomowej:
dr inż. Karol Wieczorek

Kielce, 2021

tutaj oryginał

tutaj oświadczenie

Opracowanie i implementacja systemu Headless CMS z dodatkową możliwością modelowania warstwy danych przy pomocy natywnego SQLa

Streszczenie

Systemy CMS oraz Headless CMS mają tendencję do ograniczania możliwości operowania na danych do statycznych operacji zdefiniowanych przez system. Może to powodować trudności, gdy wymagane jest wykonanie skomplikowanych operacji na bazie danych. Ponadto, wysyłanie zapytań do bazy danych, z której korzysta system CMS jest często odradzanie i niewspierane przez twórców systemu CMS. Celem pracy dyplomowej jest napisanie systemu Headless CMS, który nie ogranicza możliwości manipulacji bazą danych. Przygotowany system CMS osiąga ten cel, przez umożliwienie administratorom wykonywania natywnych zapytań SQL na bazie danych, z której korzysta system. Kolejną kluczową funkcją jest pozwolenie administratorom na definiowanie zapytań, które zostaną wykonane przy zapytaniu HTTP do systemu CMS, i których dane wynikowe zostaną zwrócone w odpowiedzi HTTP.

Słowa kluczowe: CMS, Headless CMS, bazy danych, SQL, zarządzanie treścią, warstwa danych

The development and the implementation of a Headless CMS system with support of modelling of the data layer by native SQL queries

Summary

CMS and Headless CMS systems have a tendency to limit the ability to operate on data to static operations defined by the system. It can cause difficulties when it's necessary to execute complex operations on the database. Furthermore, sending queries to the database used by the CMS system is often discouraged and not supported by the authors of the CMS system. The aim of the thesis is to prepare a Headless CMS system, which does not limit the ability of manipulating the database. The aim is achieved by allowing the administrators to execute native SQL queries on the database used by the system. Another key feature is to let administrators define queries which will be executed after an HTTP request to the CMS system and the resulting data will be returned in the HTTP response.

Keywords: CMS, Headless CMS, databases, SQL, content management, data layer

SPIS TREŚCI

1. WSTĘP

2. CEL I ZAKRES PRACY

2.1. Cel pracy

Celem pracy było przygotowanie systemu CMS. System powinien zawierać większość funkcji dostępnych w typowym systemie CMS, jak tworzenie nowych typów danych i manipulacja danymi znajdującymi się w systemie, jednak nie są to główne funkcje systemu. Głównym założeniem przy projektowaniu i implementacji było stworzenie sytemu, który nie ogranicza możliwości operowania danymi przez administratora.

Główny cel został osiągnięty przez pozwolenie administratorowi na wykonywanie natywnych zapytań SQL na bazie danych. Zaimplementowano aplikacje frontend i backend, które razem umożliwiają administratorowi wygodne pisanie zapytań SQL i testowanie ich wpływu na bazę danych.

Nie mniej ważnym założeniem było umożliwienie administratorowi definiowania punktów końcowych, których wywołanie zapytaniem HTTP powoduje wykonanie zdefiniowanego przez administratora drzewa zapytań SQL. Przygotowany edytor pozwala na modelowanie struktury danych, które zostaną zwrócone w odpowiedzi HTTP.

2.2. Zakres funkcji systemu

Program przygotowany w ramach pracy można podzielić na dwie części: frontend i backend:

1. Frontend.

    - Aplikacja SPA wykorzystującą bibliotekę react.js. Strona porozumiewa się z serwerem za pomocą API REST.
    - Jest napisany w języku Typescript i wykorzystuje system typów w każdym miejscu, gdzie jest to możliwe.
    - Strona jest responsywna dzięki zastosowanym nowoczesnym opcjom CSS - grid i flexbox. Strona korzysta ze zmiennych CSS. Zmienne CSS umożliwiły łatwą implementację funkcji zmiany motywu aplikacji przez użytkownika.
    - Implementuje ważniejsze funkcje typowego panelu administratora systemu zarządzania treścią: tworzenie nowych tabel, zarządzanie danymi w tabeli.
    - Implementuje zaawansowany edytor SQL z podświetlaniem składni, funkcją zmiany kolejności zapytań, funkcją tymczasowego wyłączania zapytania. Wyświetla wyniki każdego zapytania pod kodem zapytania. Pozwala pisać zapytanie testowe, którego wynik zostanie pobrany przed wykonaniem listy zapytań jak i po wykonaniu. Pozwala na wygenerowanie diagramu ER przed i po wykonaniu listy zapytań. Pozwala na testowanie listy zapytań z pomocą transakcji, która zostanie wycofana przed zwrotem danych.
    - Implementuje edytor punktów końcowych pozwalający na pisanie złożonych drzew zapytań, w których zapytania podrzędne mają dostęp do danych wynikających z wykonania zapytań nadrzędnych. Administrator może testować punkt końcowy przed wdrożeniem z wykorzystaniem edytora danych testowych. Administrator może ograniczać możliwość wywołania punktu końcowego do listy grup użytkowników.
    - Implementuje interfejs tworzenia użytkowników.

2. Backend.

    - Backend aplikacji został napisany w języku rust. Jest asynchroniczny, co pozwala na obsługiwanie wielu zapytań na mniejszej ilości wątków. Wykorzystuje asynchroniczny runtime Hyper.
    - Współpracuje z bazą danych PostgreSQL. Używa tabel zawierających dane o tabelach w bazie danych, przez co nie musi sam przechowywać metadanych o tabelach.
    - Korzysta z transakcji. Są używane do testowania pisanych przez administratora zapytań SQL.
    - Implementuje generowanie diagramu ER w składni mermaid.js z danych pobranych z tabel zawierających metadane o tabelach w bazie danych.
    - Implementuje bezpieczne wykonywanie drzewa zapytań SQL z użyciem niezaufanych danych z przychodzącego zapytania HTTP. Zapytania podrzędne mają dostęp do zapytań nadrzędnych. Każde wykorzystanie danych w zapytaniach jest zabezpieczone przed atakami SQL injection.
    - Implementuje system użytkowników z wykorzystaniem technologii json web token. Bezpiecznie przechowuje hasła użytkowników za pomocą algorytmu Argon2.

3. PRZEGLĄD ISTNIEJĄCYCH ROZWIĄZAŃ

3.1. Wordpress

Pierwszym omawianym systemem zarządzania treścią jest Wordpress. Jest to najbardziej popularny CMS. Ten system został napisany z myślą o prezentowaniu treści w formie strony HTML. Wordpress jest napisany w języku PHP i do działania wymaga zainstalowania serwera HTTP. Pierwsze wydanie systemu miało miejsce w roku 2003 i od tamtej pory system jest ciągle rozwijany.

Wordpress pozwala na definiowanie własnych modeli, ale nie jest to wspierane w domyślnym panelu administratora. Do stworzenia nowego typu danych, wymagane jest wywołanie funkcji PHP udostępnionej przez Wordpress, do której nie ma domyślnie interfejsu użytkownika. Polecanym przez twórców sposobem tworzenia nowych typów danych jest zainstalowanie wtyczki, która implementuje taki interfejs [6].

Pisanie własnych zapytań SQL jest możliwe, ale podobnie jak definiowanie niestandardowych typów, domyślnie nie posiada interfejsu. W celu napisania własnego zapytania, trzeba to zrobić z poziomu PHP, lub zainstalować wtyczkę umożliwiającą pisanie własnych zapytań z panelu administratora.

W wersji 5.0 Wordpress, dodano nowy edytor “Gutenberg” pozwalający na tworzenie postów opartych o bloki. Jest to mniej skomplikowana alternatywa dla stosowanych do tej pory tzw. “online rich-text editor”. Edytory blokowe pozwalają na tworzenie wpisów składających się z bloków. Blok może zawierać tekst lub media. Bloki tekstowe mogą reprezentować nagłówek lub paragraf, a paragrafy mogą mieć fragmenty z ograniczonym stylizowaniem jak na przykład pogrubieniem czcionki. Wpisy w formie bloków są łatwiejsze w przechowywaniu i wyświetlaniu na stronie HTML. Nie jest konieczne na przykład parsowanie treści wpisów, żeby wydobyć informacje o zdjęciach, jakie zostały zamieszczone we wpisie.

Wpisy oparte o bloki mogą być wygodnie tworzone w dużo prostszych edytorach. Zmniejsza to potrzebę stosowania skomplikowanych systemów CMS do zarządzania treścią.

Podsumowując, Wordpress to oprogramowanie, które dobrze spełnia potrzeby autora bloga. Znaczną zaletą systemu Wordpress jest ekosystem wtyczek pozwalających na rozwiązanie większości problemów związanych z zarządzaniem danymi.

System Wordpress domyślnie znacznie ogranicza możliwości administratora. W celu wykonywania niestandardowych operacji na bazie danych, administrator musi polegać na wtyczkach, lub samemu napisać umożliwiające to wtyczki. Zwiększa to znacznie wymagania wiedzy wobec administratora.

Pomimo tego, możliwe jest skonfigurowanie systemu Wordpress w taki sposób, żeby spełniał wszystkie funkcje przygotowanego w ramach tej pracy systemu. Taka konfiguracja byłaby jednak mniej stabilna od sytemu, który powstawał w celu spełnienia tych celów. Niezbędne byłoby poleganie na dużej ilości wtyczek, lub napisanie tych wtyczek samemu, co może być bardziej skomplikowane, niż napisanie systemu backend spełniającego potrzeby jednej strony internetowej.

(TODO: cite Wordpress bible)

3.2. Strapi

Strapi to sytem Headless CMS. Nie posiada on interfejsu użytkownika. Dane wprowadzane do systemu są pobierane za pomocą API REST. Domyślne API ma ograniczoną funkcjonalność. Pobieranie danych z CMS jest ograniczone do pobierania wszystkich wpisów danego typu, lub jednego wpisu, ale tylko po id.

Strapi umożliwia tworznie własnych punktów końcowych. Administrator może ustawić ścieżkę i funkcję, która obsłuży zapytania na daną ścieżkę. Nie da się jednak zrobić tego w panelu administratora. Administrator chcąc stworzyć niestandardowy punkt końcowy musi sam napisać funkcje, które obsłużą zapytania w języku JavaScript. Jest to porównywalnie skomplikowane do napisania kontrolera w zwykłej aplikacji internetowej. Ponadto, na administratora są nałożone pewne ograniczenia. System Strapi wspiera wiele baz danych, w tym MongoDB, która nie wspiera SQL. Z tego powodu, nie jest możliwe pisanie natywnych zapytań do bazy danych. Do operacji na bazie danych Strapi udostępnia "query engine". Jest to biblioteka do operacji na bazie danych z poziomu języka JavaScript. API udostępnione programistom przez "query engine" jest znacznie ograniczone w porównaniu do natywnego SQL. Nie można używać funkcji specyficznych do wybranej bazy danych. Operacje są ograniczone do prostych zapytań CRUD.

Strapi umożliwia operowanie na danych za pomocą GraphQL. Ta opcja pozwala na pobieranie konkretnych danych o wpisie zamiast całości informacji. W celu pobierania niestandardowych informacji, wymagane jest pisanie własnych "resolwerów". Pisanie resolwerów przypomina pisanie funkcji obsługującej zapytania do REST API. Wymaga pisania kodu źródłowego w języku JavaScript.

Podsumowując, Strapi jest dobrym wyborem, jeśli potrzebny jest system CMS pozwalający na bardziej złożone od CRUD operacje na danych. Niezbędne będzie jednak pisanie własnych funkcji obsługujących zapytania z wykorzystaniem ograniczonego API do operacji na bazie danych.

Znacznym minusem Strapi są duże wymagania wobec sprzętu. Minimalna ilość pamięci to 2GB [1]. Nie jest to problem dla przedsiębiorstw, ale może to zwiększyć koszty osób zarządzających mniejszymi stronami internetowymi.

3.3. PostgREST

PostgREST nie jest systemem CMS. Twórcy definiują ten program, jako samodzielny serwer internetowy, który przemienia bazę danych Postgres w API REST [3]. Mimo tego, PostgREST spełnia dużą część założeń tej pracy dyplomowej.

PostgREST jest przeznaczony do pracy tylko z bazą danych PostgreSQL. Dzięki takiej specjalizacji, umożliwia on korzystanie z funkcji specyficznych dla bazy danych PostgreSQL. Umożliwia wyłuskiwanie danych z kolumn o typie `json` [5] i korzystanie z funkcji full-text search [4].

Minusem serwera PostgREST jest to, że zapytania SQL są kodowane w parametrach zapytania HTTP, co zmniejsza czytelność zapytań SQL. Można to zauważyć w przykładowym zapytaniu z dokumentacji PostgREST pobierającym dane z wielu tabel:

```
GET /films?select=title,actors!inner(first_name,last_name)
        &actors.first_name=eq.Jehanne HTTP/1.1
```

Powinno zwrócić dane JSON:

```
[{
    "title": "The Haunted Castle",
    "actors": [{
        "first_name": "Jehanne",
        "last_name": "d'Alcy"
    }]
}]
```

Podsumowując, jeśli aplikacja będzie wymagała dużej ilości niestandardowych operacji na bazie danych, PostgREST z dodatkiem prostej aplikacji serwerowej jest dobrym wyborem, jeśli nie potrzeba panelu administratora. Jest to jedyne omawiane do tej pory narzędzie, które nie ogranicza w większym stopniu możliwości operowania na bazie danych.

3.4. Podsumowanie

Podsumowując, można zauważyć, że na rynku nie ma popularnego narzędzia spełniającego założenia pracy. Istnieje wiele narzędzi, które można dostosować do założeń pracy, ale takie zastosowanie nie jest zamierzone przez twórców, lub wymaga pisania kodu poza zapytaniami SQL.

4. PROJEKT SYSTEMU

4.1. Projekt aplikacji frontend

Aplikacja frontend została zaprojektowana w sposób typowy dla aplikacji SPA [8] z wykorzystaniem biblioteki React.js. Aplikacja nie wymaga ładowania nowej strony przez przeglądarkę podczas pracy.

Składa się z komponentów, z których niektóre są używane w wielu miejscach aplikacji. Komponenty, które implementują skomplikowaną funkcjonalność zawierają komponenty podrzędne.

Główny komponent aplikacji to widok kart (`TabsView`). Komponent zawiera pasek kart (`TabBar`) oraz komponent służący do wybierania edytorów (`EditorSelector`). Komponent `TabBar` implementuje listę kart, która pozwala użytkownikowi na przełączanie się pomiędzy edytorami. Karty, które nie są aktywne są chowane za pomocą ustawienia wartości CSS `display` na `none`. W ten sposób, nie tracą danych przechowywanych w stanie komponentu przy przełączaniu kart. Komponent `EditorSelector` pozwala na wybór edytora. Edytory zawierają interfejs, przy użyciu którego administrator wykonuje pracę na systemie CMS.

Komponent `EditorSelector` pobiera dane o dostępnych edytorach ze struktury danych. Umożliwia to łatwe dodawanie nowych edytorów do aplikacji. Strukturę danych widać na listingu 1. Struktura danych przechowuje dane o folderach, które mogą zawierać edytory. Foldery i edytory mają ikonę oraz opis, a edytory mają również komponent. Przy wyborze edytora, komponent przypisany do tego edytora zostanie wyświetlony na ekranie.

Listing 1: Struktura danych przechowująca dane o edytorach

```
const editors: EditorTree = {
  "Data management": {
    icon: faDatabase,
    editors: {
      "Table data management": { icon: faTable, component: TableDataManagement, },
      "Data SQL editor": { icon: faCode, component: DataSqlEditor },
    },
  },
  "Schema editing": {
    icon: faTable,
    editors: {
      "Table creation": { icon: faPlus, component: TableCreation },
    },
  },
  "Endpoint management": {
    icon: faCode,
    editors: {
      "Endpoint management": { icon: faCogs, component: EndpointManagement },
    },
  },
  "User management": {
    icon: faUsers,
    editors: {
      "User creation": { icon: faUserPlus, component: UserCreation },
      "User management": { icon: faUsersCog, component: UserManagement },
    },
  },
};
```

Aplikacja frontend pozwala użytkownikowi na zmianę motywu. Użytkownik może wybrać motyw jasny i ciemny. Przełączenie motywu powoduje podmienienie zmiennych CSS z kodu aplikacji. Wybrany przez użytkownika motyw jest zapisywany z wykorzystaniem standardowej funkcji przeglądarki `localStorage` i ładowany przy uruchomieniu aplikacji. Domyślny motyw to motyw jasny (ciemny tekst na jasnym tle). Kod odpowiedzialny za przełączanie motywu został zamieszczony na listingu 2.

4.2. Projekt aplikacji backend

Aplikacja backend nie implementuje żadnego z wzorców architektonicznych. Cała aplikacja jest podzielona na trzy moduły:

- Moduł zawierający algorytmy.
- Moduł zawierający funkcje obsługujące zapytania HTTP.
- Moduł zawierający serwisy.

Moduł zawierający algorytmy zawiera najbardziej skomplikowane logicznie elementy programu. Znajdują się tutaj funkcje:

- generujące diagram ER w składni mermaid.js z tabel zawartych w systemie,
- parsujące zapytania SQL zawierające odniesienia do zmiennych,
- wykonywujące sparsowane drzewo zapytań SQL zawierających odniesienia do zmiennych.

Serwisy zawierają logikę biznesową i wywołują algorytmy. Duża część funkcji zawartych w module serwisów odpowiada za zbieranie danych o tabelach zawartych w bazie danych.

Funkcje obsługujące zapytania HTTP zawierają ograniczoną logikę. Funkcje te powinny jedynie wywoływać serwisy i przekształcać możliwe błędy na odpowiedzi HTTP.

Listing 2: Kod odpowiedzialny za przełączanie motywu

```
export function toggleTheme() {
  let current = getTheme() || defaultTheme;
  let newTheme: string;

  if (current.indexOf("light") !== -1) {
    newTheme = current.replace("light", "dark");
  } else {
    newTheme = current.replace("dark", "light");
  }

  setTheme(newTheme);
}

export function setTheme(scheme: string) {
  if (Object.keys(colorschemes).indexOf(scheme) === -1) {
    console.error(`Colorscheme ${scheme} NOT DEFINED!`);
    return;
  }

  for (let [key, value] of Object.entries(colorschemes[scheme])) {
    document.documentElement.style.setProperty(`--${key}`, value as string);
  }

  // Odwrócenie kolorów w diagramie ER mermaid.js, gdy wybrano motyw ciemny
  let mermaidFilter = "";
  if (scheme.indexOf("dark") !== -1) mermaidFilter = "invert(1)";
  document.documentElement.style.setProperty("--mermaid-filter", mermaidFilter);

  localStorage.setItem("colorscheme", scheme);
}
```

5. IMPLEMENTACJA KLUCZOWYCH FUNKCJI

5.1. Tworzenie nowych typów danych

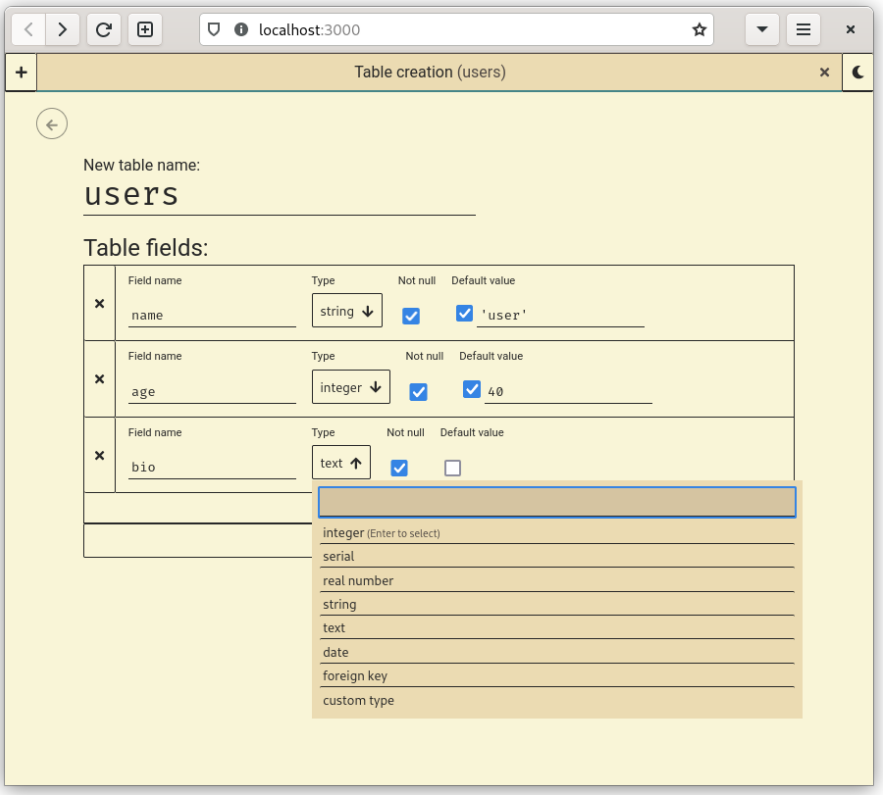
Funkcjonalność tworzenia nowych typów danych nie wyróżnia przygotowanego systemu od systemów CMS dostępnych na rynku. Jest zaimplementowana, jako lista formularzy. Każdy formularz odpowiada jednej kolumnie w nowo utworzonej tabeli. Administrator może dodawać nowy formularz, lub kasować już istniejący.

Pojedyńczy formularz zawiera następujące elementy:

- Pole tekstowe, gdzie administrator podaje nazwę kolumny.
- Menu rozwijane pozwalające na wybór typu kolumny.
- Pole wyboru, które pozwala określić administratorowi, czy kolumna może zawierać wartości `null`.
- Pole wyboru, które pozwala określić czy kolumna posiada wartość domyślną wraz z polem tekstowym, które pojawia się, gdy administrator oznaczy kolumnę jako posiadającą wartość domyślną, gdzie należy podać wartość domyślną.

Pod listą formularzy można znaleźć przycisk umożliwiający wysłanie definicji tabeli do programu backend, który utworzy ją w bazie danych. Pod przyciskiem znajduje się ramka, w której pojawiają się możliwe błędy zwrócone przez bazę danych, lub informacja o pomyślnym stworzeniu tabeli.

Interfejs tworzenia nowych typów danych jest pokazany na rysunku 1.



Rysunek 1: Interfejs tworzenia nowego typu danych

5.2. Zarządzanie danymi

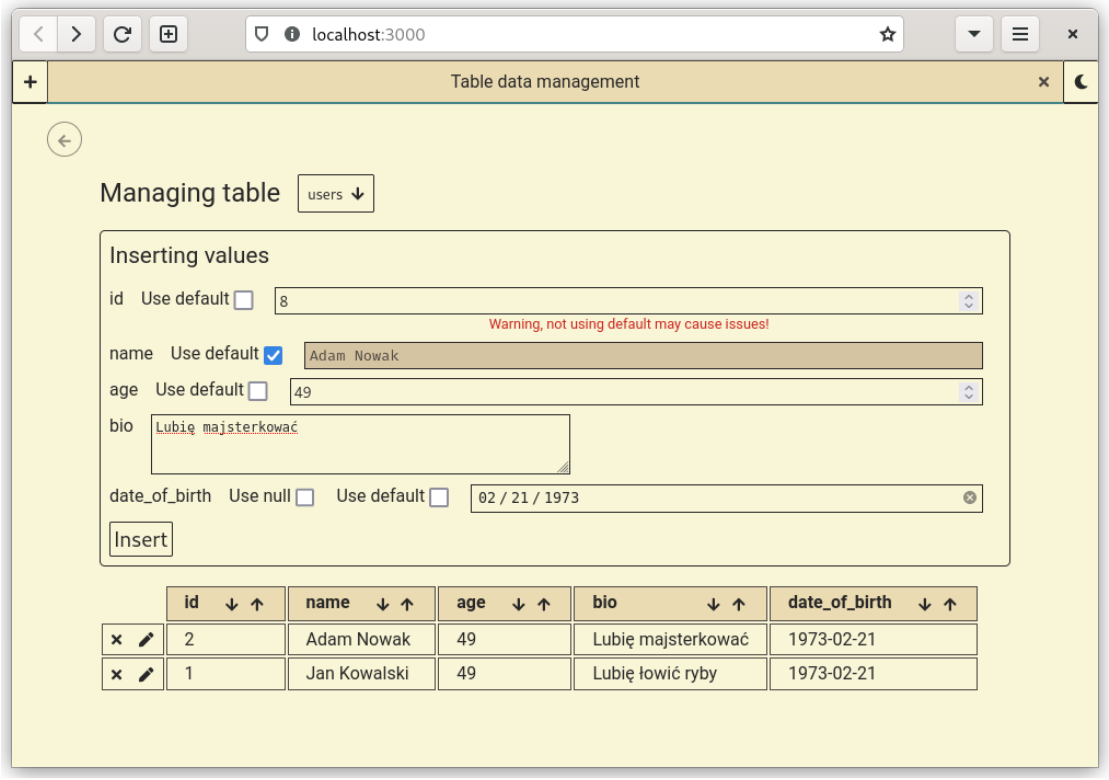
Interfejs zarządzania danymi zawiera menu rozwijane, które administrator używa do wyboru tabeli, której danymi chce zarządzać. Komponent używany do tego menu to ten sam komponent, który jest użyty do wyboru typu danych w interfejsie tworzenia nowych typów danych.

Pierwszym elementem, który administrator zauważy po wybraniu tabeli jest formularz wprowadzania nowych danych. Składa się on z listy formularzy, podobnie, jak interfejs tworzenia nowych typów danych. Aplikacja dobiera odpowiedni typ formularza HTML do typu danych kolumny, której dane daje możliwość wprowadzania. Przykładowo, kolumny typu `varchar` spowodują wyświetlenie elementu `<input type="text">`, elementy typu `text` spowodują wyświetlenie elementu `<textarea>`, a kolumny typu `date` spowodują wyświetlenie elementu `<input type="date">`.

Funkcją, na którą warto zwrócić uwagę jest wyświetlanie ostrzeżenia, przy próbie wprowadzenia danych niedomyślnych do kolumny, która jest kluczem głównym, i której wartość domyślna zawiera `nextval`. Może to spowodować błąd w późniejszej próbie dodania danych do tej tabeli przez powtórzenie klucza głównego.

Pod interfejsem wprowadzania nowych danych znajduje się widok danych w tabeli. Widok ten zawiera funkcje zmiany sortowania, edycji oraz usuwania danych. Interfejs wspiera paginację, domyślnie wyświetla sto rzędów.

Interfejs jest pokazany na rysunku 2.



Rysunek 2: Interfejs zarządzania danymi w tabeli

5.3. Edytor zapytań SQL wykonywanych natychmiast

Edytor zapytań SQL wykonywanych natychmiast to jeden z najważniejszych elementów programu. Pozwala administratorowi na wygodne edytowanie, testowanie i wykonywanie natywnych zapytań SQL.

Interfejs komponentu składa się z edytora zapytań, edytora zapytania testowego i przycisków służących do wykonywania lub testowania zapytań.

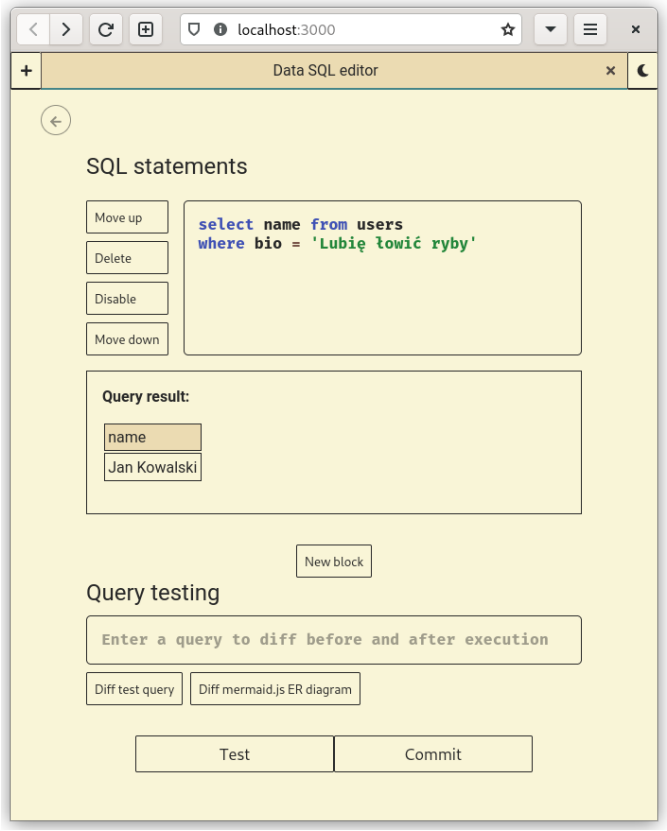
Wybranie opcji “Commit” powoduje wykonanie zapytania w transakcji, która zostanie zatwierdzona, gdy wszystkie zapytania zostaną wykonane bez błędów. Wybranie opcji commit nie pokaże administratorowi informacji zwrotnych poza informacją, czy wykonanie zapytań przebiegło pomyślnie.

Wybranie opcji “Test” spowoduje wykonanie transakcji, która zostanie wycofana przed zwróceniem danych wynikających z zapytań SQL. Dane wynikające z zapytań z listy zapytań zostaną wyświetlone pod odpowiednim zapytaniem.

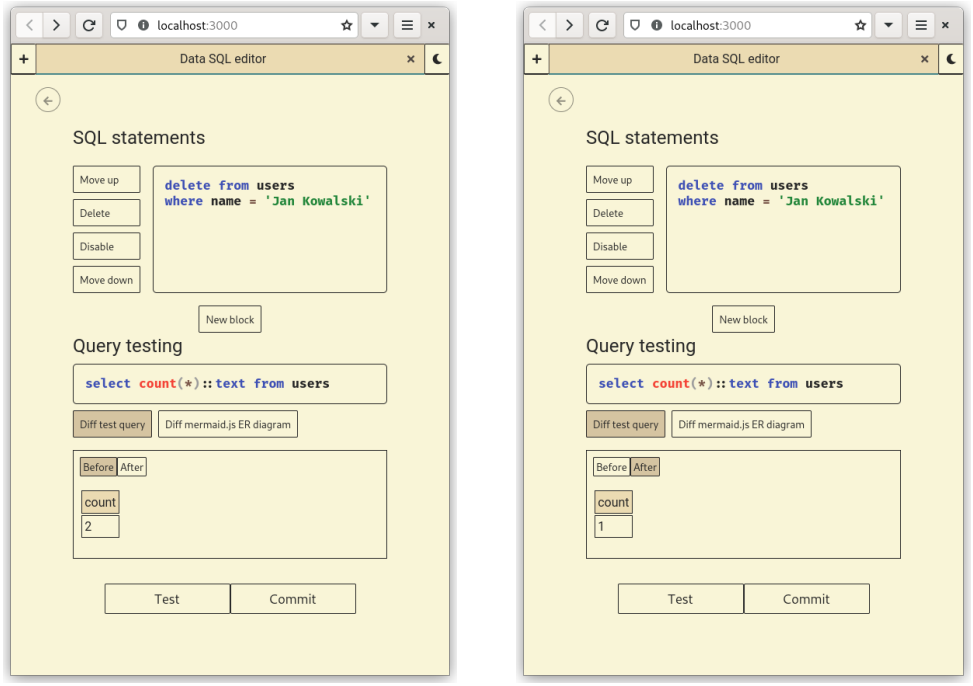
Jeśli administrator zaznaczył opcję “Diff test query”, dane wynikające z zapytania testowego zostaną wyświetlone w specjalnym oknie pod tekstem zapytania testowego z możliwością przełączania pomiędzy wynikiem zapytania przed wykonaniem listy zapytań i po wykonaniu listy zapytań. Interfejs widać na rysunku 4.

Jeśli administrator zaznaczył opcję “Diff mermaid.js ER diagram”, program wygeneruje diagram ER tabel w bazie danych przed wykonaniem listy zapytań i po wykonaniu listy zapytań. Diagramy zostaną pokazane w specjalnym oknie z możliwością przełączania pomiędzy diagramem przed wykonaniem listy zapytań i po wykonaniu listy zapytań oraz z możliwością pokazania diagramu w trybie pełnoekranowym. Porównanie widać na rysunku 5, a widok pełnoekranowy widać na rysunku 6.

Edytor składa się z listy pól tekstowych, gdzie administrator może wpisywać pojedyncze zapytania SQL. Administrator może dodawać nowe pola tekstowe, zmieniać kolejność pól, kasować pola i wyłączać pola. Pola wyłączone nie będą wykonywane. Jest to przydatne do testowania wpływu wykonania zapytania na wynik kolejnych zapytań.



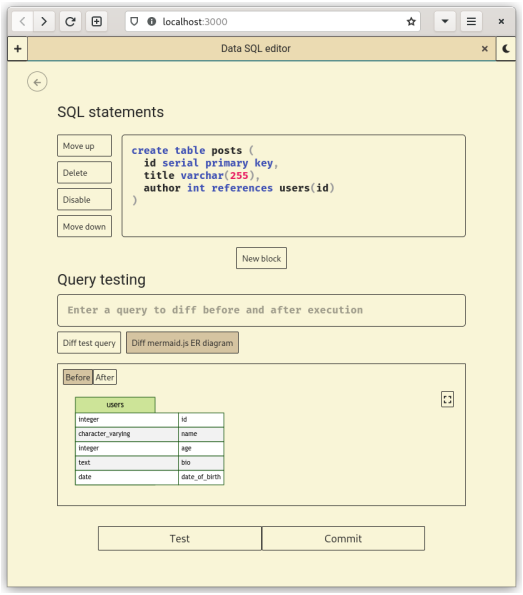
Rysunek 3: Edytor SQL wykonywanego natychmiast



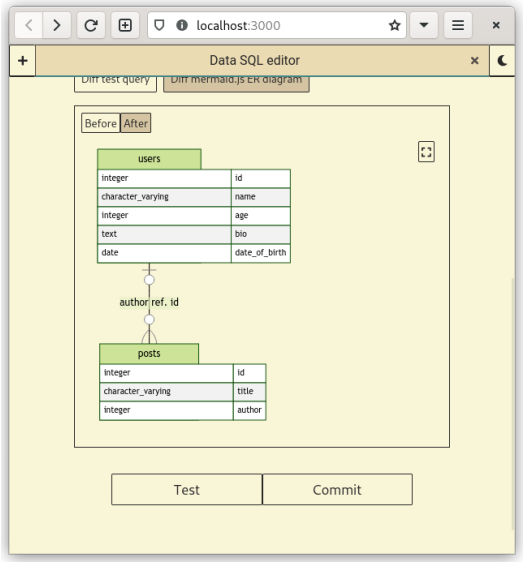
(a) Przed wykonaniem

(b) Po wykonaniu

Rysunek 4: Porównanie zapytania testowego

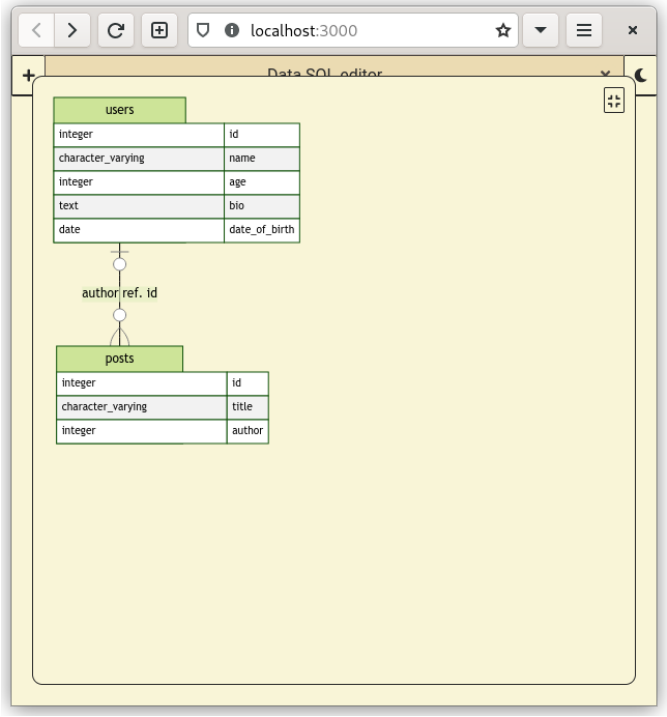


(a) Przed wykonaniem



(b) Po wykonaniu

Rysunek 5: Porównanie diagramu ER



Rysunek 6: Widok pełnoekranowy diagramu

5.4. Edytor punktów końcowych

Edytor punktów końcowych to jedna z ważniejszych funkcji programu. Umożliwia administratorowi definiowanie punktów końcowych HTTP, których wywołanie spowoduje wykonanie napisanego przez administratora drzewa zapytań SQL.

Dane wynikające z wykonania drzewa zapytań zostaną zwrócone w formacie json. Struktura zwróconego dokumentu json będzie odpowiadała strukturze drzewa zapytań SQL zaprojektowanego przez administratora.

Zapytania podrzędne mogą korzystać z danych wynikających z zapytania nadrzędnego. Dane można pobrać za pomocą specjalnej składni, która zostanie sparsowana przez program. Dane zostaną umieszczone w zapytaniach nadrzędnych w sposób zabezpieczony przed atakami SQL injection.

Składnia dostępu do danych była zaprojektowana z myślą o prostocie, powinna być trywialna w nauce dla kogoś, kto ma nawet podstawową wiedzę o językach programowania. Składa się z dwóch wariantów:

1. Wariant dostępu do danych przychodzących w zapytaniu HTTP wygląda następująco:

   (a) rozpoczęcie `${`,

   (b) prefiks `req.`,

   (c) nazwa zmiennej,

   (d) zakończenie `}`.

   Przykładowo, dostęp do pola “nazwa” wygląda tak: `${req.nazwa}`.

2. Wariant dostępu do danych wynikających z zapytań nadrzędnych wygląda następująco:

   (a) rozpoczęcie `${`,

   (b) jeden, lub więcej prefiksów `super.`,

   (c) nazwa zmiennej,

   (d) zakończenie `}`.

   Przykładowo, dostęp do pola “user_id” będącego wynikiem zapytania nadrzędnego do zapytania nadrzędnego wygląda tak: `${super.super.user_id}`.

Jeśli administrator potrzebuje danych do zapytań podrzędnych, ale nie chce, żeby były umieszczone w odpowiedzi HTTP, powinien nazwać je tak, by ich nazwa zaczynała się od prefiksu “`private_`”. Pola o takiej nazwie można wykorzystywać w zapytaniach podrzędnych (na przykład `${super.private_user_id}`), ale nie będą zwrócone w odpowiedzi HTTP.

Działanie i odporność na ataki SQL injection parsera SQL wchodzącego w skład programu można zrozumieć na przykładzie zawartym w teście jednostkowym zamieszczonym w listingu 3.

Sparsowane zapytania SQL składają się z SQL zawierającego odniesienia do parametrów w składni specyficznej dla bazy danych postgresql [2] i listy ciągów znaków, które umożliwią znalezienie zmiennej w czasie wykonywania drzewa zapytań SQL.

Interfejs edytora punktów końcowych zawiera menu rozwijane, które pozwala na wybór edytowania już istniejącego punktu końcowego, lub tworzenia nowego punktu końcowego.

Poniżej znajduje się formularz, za pomocą którego tworzony jest punkt końcowy. Pierwszym elementem formularza jest pole tekstowe, do którego należy wpisać ścieżkę punktu końcowego. Poniżej znajduje się menu, którego należy użyć do ustawienia akceptowanych metod HTTP. Opcje to GET, POST i ANY. Wybranie opcji ANY powoduje akceptowanie zapytań o dowolnej medodzie HTTP.

Listing 3: Test sprawdzający poprawność parsowania SQL zawierającego odniesienia do zmiennych

```
#[test]
fn parsing_multiple() {
    let sql = "select * from users where name=${req.name} and age=${req.age} or name =
         upper(${req.name})";
    let parsed = SqlWithVariables::from_sql(sql).unwrap();

    assert_eq!(
        &parsed.sql,
        "select * from users where name=$1 and age=$2 or name = upper($3)"
    );
    assert_eq!(&parsed.variables, &vec!["req.name", "req.age", "req.name"]);
}
```

Kolejnym elementem jest menu, gdzie administrator może ustawić listę grup użytkowników, które będą miały dostęp do edytowanego punktu końcowego.

Kolejnym, najbardziej skomplikowanym, elementem jest edytor drzewa zapytań SQL. Administrator może tworzyć kilka niezależnych drzew. Umożliwia to opcja “New independent query”. Każdy węzeł drzewa powinien mieć nazwę, która zostanie użyta, jako klucz w mapie węzłów w wynikowym dokumencie json.

Węzły będące liśćmi można usunąć za pomocą przycisku “delete”. W celu zapobiegnięcia niezamierzonego skasowania pracy, węzły posiadające podzapytania nie posiadają opcji usunięcia. W celu usunięcia tych węzłów, należy najpierw usunąć ich wszystkie węzły podrzędne.

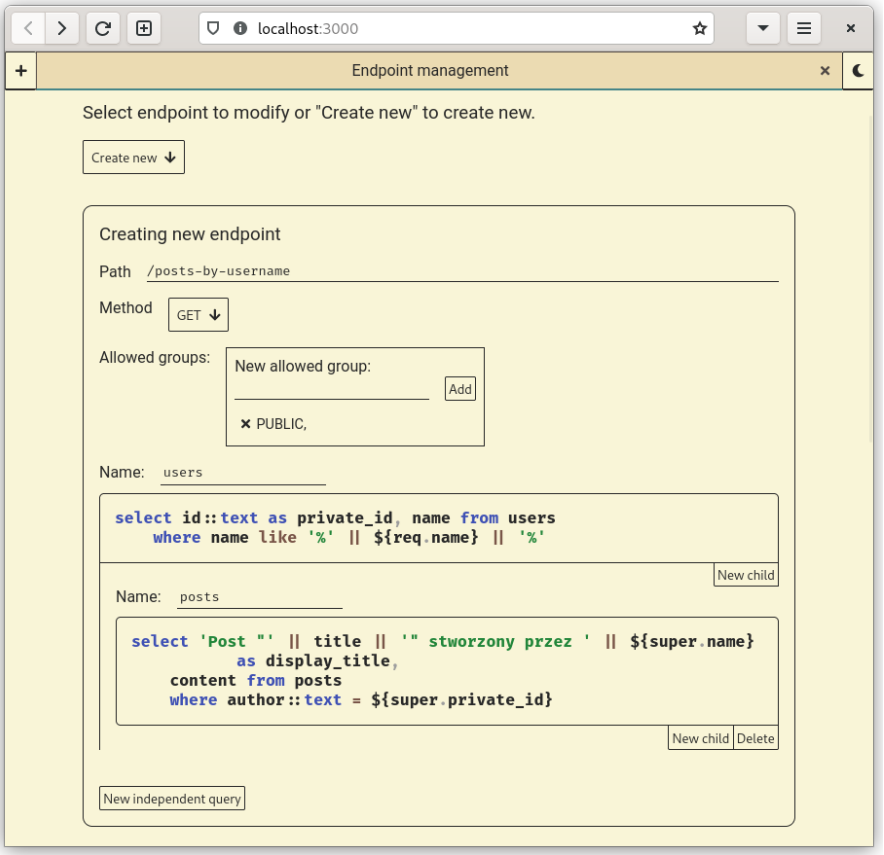
Każdy węzeł posiada opcję dodania nowego węzła podrzędnego. Można to zrobić za pomocą przycisku “New child”.

Opisany do tej pory interfejs widać na rysunku 7.

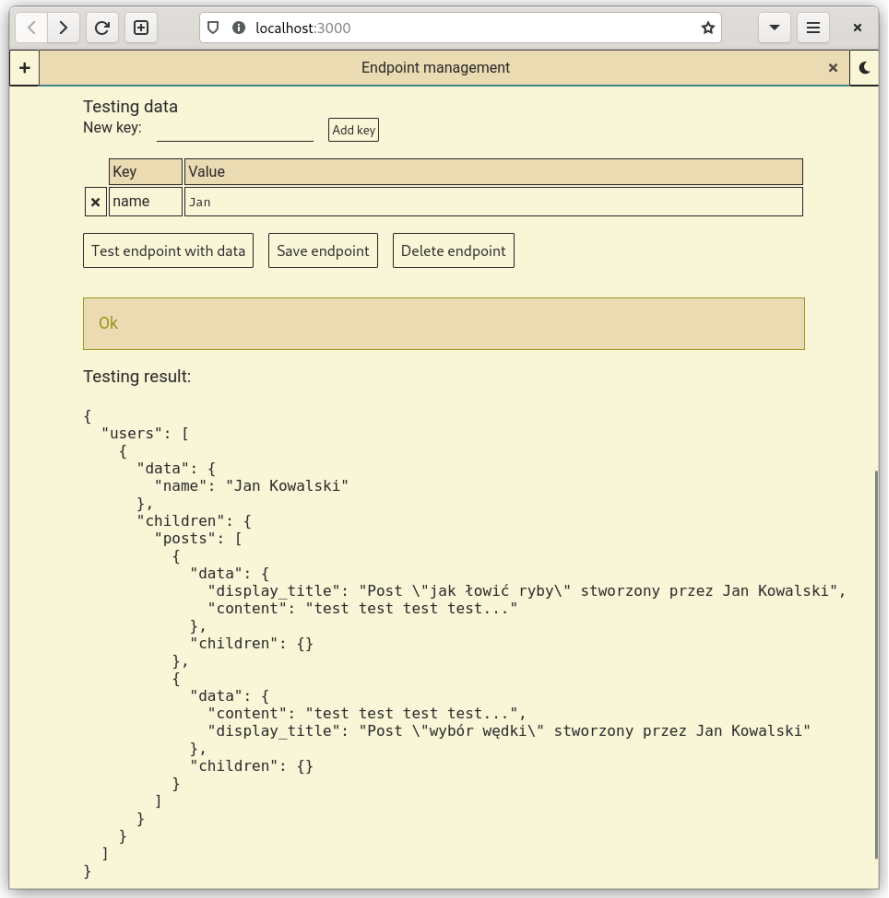
Program implementuje funkcjonalność mającą na celu umożliwienie administratorowi wygodnego testowania tworzonych punktów końcowych. Pod edytorem punktu końcowego znajduje się edytor danych testowych. Umożliwia on tworzenie mapy zmiennych, które będą traktowane, jak wartości formularza przychodzącego w zapytaniu HTTP.

Testowe wykonanie drzewa zapytań SQL używa transakcji, która jest cofana przed zwrotem wynikowego dokumentu json. Wynikowy dokument json jest wyświetlany na dole interfejsu. Wynik testowania punktu końcowego widać na rysunku 8.

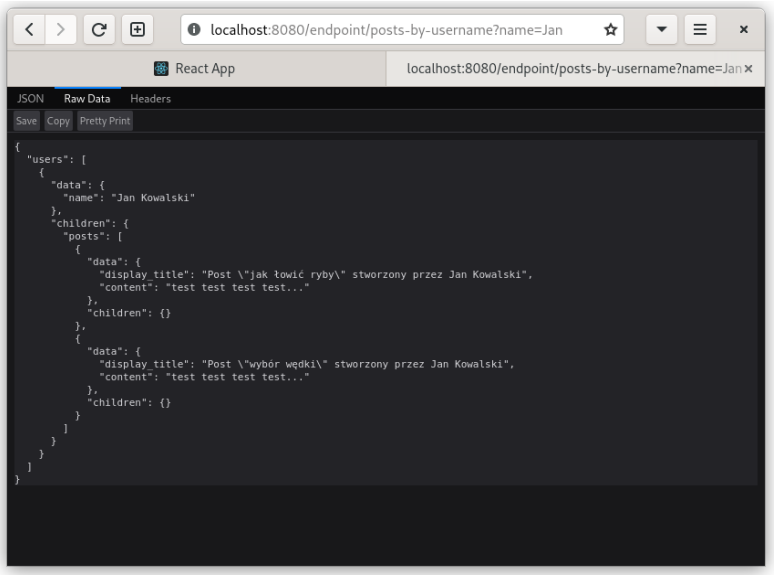
Zapisane punkty końcowe będą dostępne pod adresem `/endpoint/<ścieżka punktu końcowego>`. Wywołanie zapisanego punktu końcowego z przeglądarki Firefox - poza panelem administratora widać na rysunku 9.



Rysunek 7: Interfejs tworzenia nowego punktu końcowego



Rysunek 8: Interfejs testowania tworzonego punktu końcowego



Rysunek 9: Wywołanie punktu końcowego w przeglądarce Firefox

5.5. Wykonywanie SQL przy wywołaniu punktu końcowego

Wykonywanie SQL przy wywołaniu punktu końcowego to najbardziej skomplikowana logicznie funkcja programu. Algorytmy są napisane w metodach struktury danych `EndpointExecutionRuntime`. Definicja struktury danych jest widoczna na listingu 4.

Zmienna `request_map` przechowuje dane pobrane z zapytania HTTP. Są pobierane za pomocą odniesienia `${req.<nazwa>}` w zapytaniu SQL.

Zmienna `execution_maps` przechowuje stos wyników zapytań nadrzędnych. Przed wykonaniem zapytań podrzędnych, wynik bieżącego zapytania jest odkładany na stos przechowywany w tej zmiennej.

Zapytania podrzędne są wykonywane dla każdego rzędu będącego wynikiem bieżącego zapytania SQL, więc na stosie `execution_maps` przechowywane są dane wynikowe jednego rzędu.

Listing 4: Struktura danych, której metody wykonują SQL przy wywołaniu punktu końcowego

```
#[derive(Debug)]
pub struct EndpointExecutionRuntime {
    request_map: HashMap<String, String>,
    execution_maps: Vec<HashMap<String, String>>,
}
```

Algorytm znajdowania wartości zmiennej można opisać następująco:

1. Jeśli nazwa zaczyna się od "`req.`", pobierz z danych z zapytania HTTP,
2. Jeśli nazwa zaczyna się od "`super.`",
   (a) Ustaw index na rozmiar stosu tablic wynikowych operacji nadrzędnych,
   (b) Dopóki nazwa zaczyna się od "`super.`",
      i. Zmniejsz indeks o 1,
      ii. Usuń "`super.`" z początku nazwy zmiennej,
   (c) Pobierz wartość z tablicy mieszającej ze stosu pod wynikowym indeksem, gdzie klucz jest równy nazwie zmiennej.

Implementacja algorytmu została zamieszczona na listingu 5.

Algorytm wykonywania drzewa zapytań SQL można opisać następująco:

1. Dla każdego zapytania SQL przypisanego do punktu końcowego,
   (a) Dla każdego argumentu pobieranego przez zapytanie,
      i. Pobierz wartość argumentu z zapytania HTTP lub wyników zapytań nadrzędnych,
      ii. Wykonaj zapytanie SQL na bazie danych,
      iii. Dla każdego wiersza, który zwróci baza danych, zapisz kolumny w tablicy mieszającej,
         A. Odłóż wynikową tablicę na stos,
         B. Rekurencyjnie wywołaj algorytm dla zapytań podrzędnych,

Całość algorytmu jest wykonywana z użyciem asynchronicznego dostępu do bazy danych.

Implementacja algorytmu została zamieszczona na listingu 6.

Listing 5: Algorytm znajdowania wartości zmiennej

```
fn get_variable_clone(&self, key: &str) -> Result<String> {
    if key.len() >= 4 && &key[0..4] == "req." {
        let key = &key[4..];
        return self
            .request_map
            .get(key)
            .map(|it| it.clone())
            .ok_or(anyhow!("Request key {} not found", key));
    } else if key.len() >= 6 && &key[0..6] == "super." {
        let mut counter = 0_usize;
        let mut inner_key = key;

        while inner_key.len() >= 6 && &inner_key[0..6] == "super." {
            inner_key = &inner_key[6..];
            counter += 1;
        }

        if counter > self.execution_maps.len() {
            return Err(anyhow!(
                "Too many 'super.'s, reached negative index ({})",
                key
            ));
        }

        let vec_index = self.execution_maps.len() - counter;
        let map = &self.execution_maps[vec_index];
        return map
            .get(inner_key)
            .map(|it| it.clone())
            .ok_or(anyhow!("Execution key {} not found", key));
    } else {
        return Err(anyhow!(
            "Bad variable name ({}). Should begin with super. or req.",
            key
        ));
    }
}
```

Listing 6: Algorytm wykonywania drzewa zapytań SQL

```
#[async_recursion]
pub async fn execute<'a>(
    &mut self,
    transaction: &mut Transaction<'a, Postgres>,
    endpoint_infos: &Vec<EndpointInfo>,
) -> Result<HashMap<String, Vec<ExecutionResult>>> {
    let mut final_results = HashMap::<String, Vec<ExecutionResult>>::new();

    for query in endpoint_infos {
        let mut exec = sqlx::query_as::<Postgres, ArbitrarySqlRow>(&query.parsed_sql);
        for var_name in &query.variables {
            let val = self.get_variable_clone(var_name)?;
            exec = exec.bind(val);
        }
        let results = exec
            .fetch_all(&mut *transaction)
            .await?
            .into_iter()
            .map(|it| it.into_map())
            .collect::<Vec<_>>();

        dbg!(&results);

        for result in results.into_iter() {
            self.push_execution_map(result);

            let children_results = self.execute(transaction, &query.children).await?;

            let mut result_map = self
                .pop_execution_map()
                .ok_or(anyhow!("Could not pop execution map"))?;

            // delete private fields
            result_map.retain(|key, _value| key.len() < 8 || &key[0..8] != "private_")
                ;

            if final_results.contains_key(&query.name) {
                final_results
                    .get_mut(&query.name)
                    .unwrap()
                    .push(ExecutionResult {
                        data: result_map,
                        children: children_results,
                    })
            } else {
                final_results.insert(
                    query.name.clone(),
                    vec![ExecutionResult {
                        data: result_map,
                        children: children_results,
                    }],
                );
            }
        }
    }

    Ok(final_results)
}
```

6. WERYFIKACJA PRACY PROGRAMU

6.1. Testy jednostkowe

Działanie wszystkich funkcji wchodzących w skład logiki biznesowej programu zostało zweryfikowane testami jednostkowymi. Wynik wykonania testów jednostkowych można zobaczyć na listingu 7.

Listing 7: Wykonanie testów jednostkowych

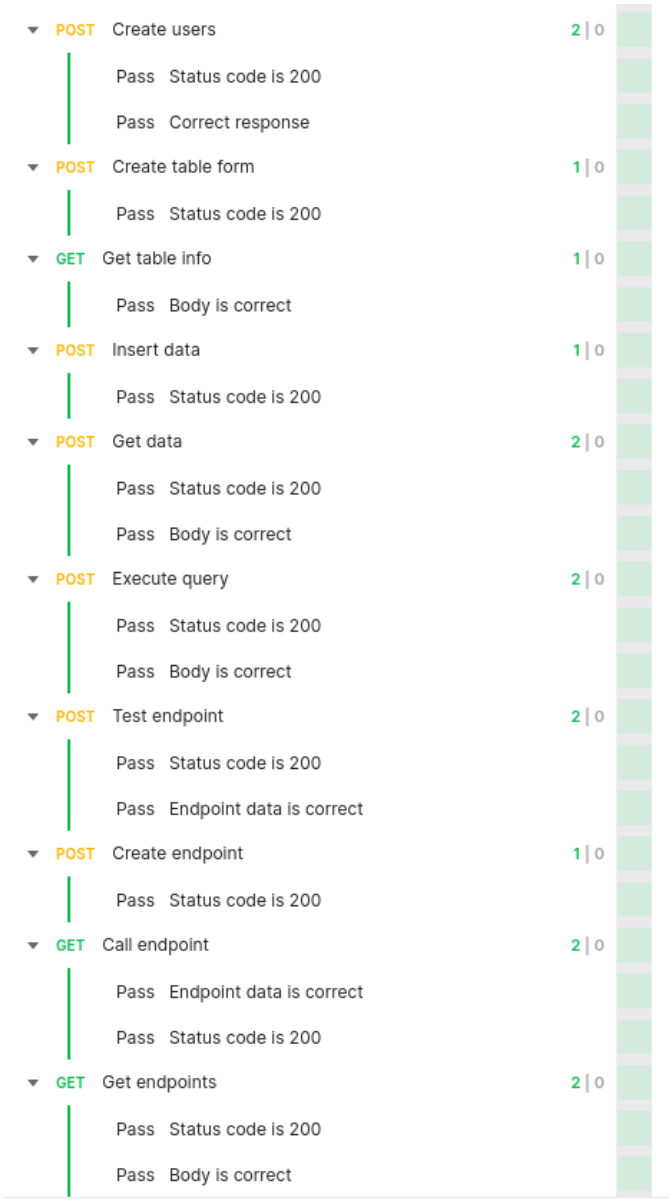
```
running 6 tests
test algorithms::mermaid_diagram_generation::tests::works ... ok
test algorithms::sql_variable_parser::tests::bind_vec ... ok
test algorithms::sql_variable_parser::tests::not_closed_panics - should panic ... ok
test algorithms::sql_variable_parser::tests::parsing_multiple ... ok
test algorithms::sql_variable_parser::tests::parsing_endpoint_info ... ok
test algorithms::sql_variable_parser::tests::parsing_test ... ok

test result: ok. 6 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished
    in 0.00s
```

6.2. Testy integracyjne

Działanie niektórych funkcji programu jest zależne od bazy danych. Weryfikacja działania tych funkcji za pomoca testów jednostkowych wymagałaby napisania dużej ilości serwisów mock. Z tego powodu, przetestowanie tych funkcji programu, jak i tych, które posiadają testy jednostkowe przeprowadzono za pomocą testów integracyjnych.

Testy integracyjne zostały napisane w programie Postman. Jest to program służący do testowania API. Testy napisane w tym programie można wyeksportować do pliku json, który można wykonać z poziomu powłoki tekstowej za pomocą programu newman. Pozwala to na automatyczne wykonywanie testów.

Wykonanie testów integracyjnych widać na rysunku 10.



Rysunek 10: Wykonanie testów integracyjnych w programie Postman

7. test

7.1. test sub

hello, world! lorem ipsum hello hello raz dwa trzy hello, world! lorem ipsum hello hello raz dwa trzy hello, world! lorem ipsum hello hello raz dwa trzy hello, world! lorem ipsum hello hello raz dwa trzy hello, world! lorem ipsum hello hello raz dwa trzy hello, world! lorem ipsum hello hello raz dwa trzy hello, world! lorem ipsum hello hello raz dwa trzy hello, world! lorem ipsum hello hello raz dwa trzy [7]

Literatura

[1] Deployment - strapi developer docs. `https://docs.strapi.io/developer-docs/latest/setup-deployment-guides/deployment.html`. Dostęp: 2020-12-10.

[2] Postgres documentation - sql prepare. `https://www.postgresql.org/docs/current/sql-prepare.html`. Dostęp: 2020-12-30.

[3] Postgrest documentation. `https://postgrest.org/en/stable/index.html`. Dostęp: 2020-12-10.

[4] Postgrest documentation - full-text search. `https://postgrest.org/en/stable/api.html#fts`. Dostęp: 2020-12-10.

[5] Postgrest documentation - json columns. `https://postgrest.org/en/stable/api.html#json-columns`. Dostęp: 2020-12-10.

[6] Wordpress theme handbook - post types. `https://developer.wordpress.org/themes/basics/post-types/#custom-post-types`. Dostęp: 2020-12-10.

[7] Andreas Mauthe and Peter Thomas. *Professional Content Management Systems*. John Wiley & Sons, Ltd, jan 2004.

[8] Michael Mikowski and Josh Powell. *Single page web applications: JavaScript end-to-end*. Simon and Schuster, 2013.