

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

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 odradzane 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
1.1	Przybliżenie domeny pracy	2
2	CEL I ZAKRES PRACY	4
2.1	Cel pracy	4
2.2	Zakres funkcji systemu	4
3	PRZEGLĄD ISTNIEJĄCYCH ROZWIĄZAŃ	6
3.1	Wordpress	6
3.2	Strapi	7
3.3	PostgREST	8
3.4	Podsumowanie	9
4	PROJEKT SYSTEMU	10
4.1	Projekt aplikacji frontend	10
4.2	Projekt aplikacji backend	10
5	IMPLEMENTACJA KLUCZOWYCH FUNKCJI	13
5.1	Główna struktura interfejsu	13
5.2	Tworzenie nowych typów danych	13
5.3	Zarządzanie danymi	14
5.4	Edytor zapytań SQL wykonywanych natychmiast	16
5.5	Edytor punktów końcowych	20
5.6	Wykonywanie SQL przy wywołaniu punktu końcowego	24
6	WERYFIKACJA PRACY PROGRAMU	29
6.1	Testy jednostkowe	29
6.2	Testy integracyjne	31
6.3	Testy manualne	33
7	PODSUMOWANIE	37

1. WSTĘP

1.1. Przybliżenie domeny pracy

W ostatnich latach można zaobserwować zanikanie podziału pomiędzy mediami drukowanymi a internetowymi. Większość organizacji publikujących treści prowadzi strony internetowe. Wiele innych organizacji nie zajmujących się mediami zмага się z problemem zarządzania treścią, którą produkują [9].

Programy, które wspomagają tworzenie i zarządzanie treścią nazywa się systemami zarządzania treścią (ang. content management system — CMS). Do tych aplikacji zaliczają się dość proste aplikacje służące do zarządzania plikami, jak i skomplikowane systemy obsługujące wiele rodzajów danych i urządzeń.

Pośród organizacji, które korzystają z systemów zarządzania treścią są między innymi: CNN, New York Times, Fox News, Wall Street Journal, Armia Stanów Zjednoczonych. Wszystkie te organizacje korzystają z systemu Wordpress. Ponadto, oprócz dużych organizacji, serwis wordpress.com w roku 2009 udostępniał usługi ponad trzem i pół miliona blogom [8]. Mimo to, nie powstał jeszcze uniwersalny system CMS, który spełniłby zapotrzebowania zarówno małych instytucji, jak i dużych organizacji, które przechowują duże ilości danych. Wątpliwe jest nawet, czy taki system mógłby zostać zbudowany, ponieważ wymagania z różnych przypadków użycia są zupełnie inne [9].

Ostatnio, popularne stały się systemy headless CMS [1]. Są to systemy CMS pozbawione warstwy prezentacji. Są one zazwyczaj mniej skomplikowane od tradycyjnych systemów i nie ograniczają wyboru technologii, której można użyć do implementacji niestandardowej warstwy prezentacji. Systemy te dobrze spełniają się w architekturze mikroserwisów, lub gdy produkowane treści są wyświetlane w różnych formach — na przykład na stronie internetowej i w aplikacji mobilnej [1].

Zdaniem autora pracy, mimo że systemy headless CMS nie ograniczają warstwy prezentacji, nie oferują większych, niż tradycyjne systemy możliwości operowania na warstwie danych.

Z tego powodu, w ramach tej pracy, przygotowano system headless CMS, którego założeniem było nie ograniczanie możliwości operowania na bazie danych przez umożliwienie administratorom wykonywania natywnych zapytań SQL.

W rozdziale 2 został omówiony cel i zakres pracy. Opisano system, jakiego przygotowanie było celem pracy i funkcjonalności jego komponentów. W rozdziale 3 przedstawione zostały systemy CMS jak i inne systemy spełniające pewien zakres problemów, który chce spełnić przygotowany w ramach pracy system. W celu uzasadnienia potrzeby przygotowania systemu, zwrócono uwagę na problemy, które nie są rozwiązane przez poszczególne systemy dostępne na rynku. W rozdziale 4 omówiono ogólną strukturę

kodu programu. W rozdziale 5 omówiono szczegółowo, jak zaimplementowano każdą z kluczowych funkcji systemu. W rozdziale 6 omówiono używane przy pisaniu programu sposoby weryfikacji działania kodu. Pracę kończy rozdział z podsumowaniem i planem dalszego rozwoju.

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 systemu, 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 wykorzystująca 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.

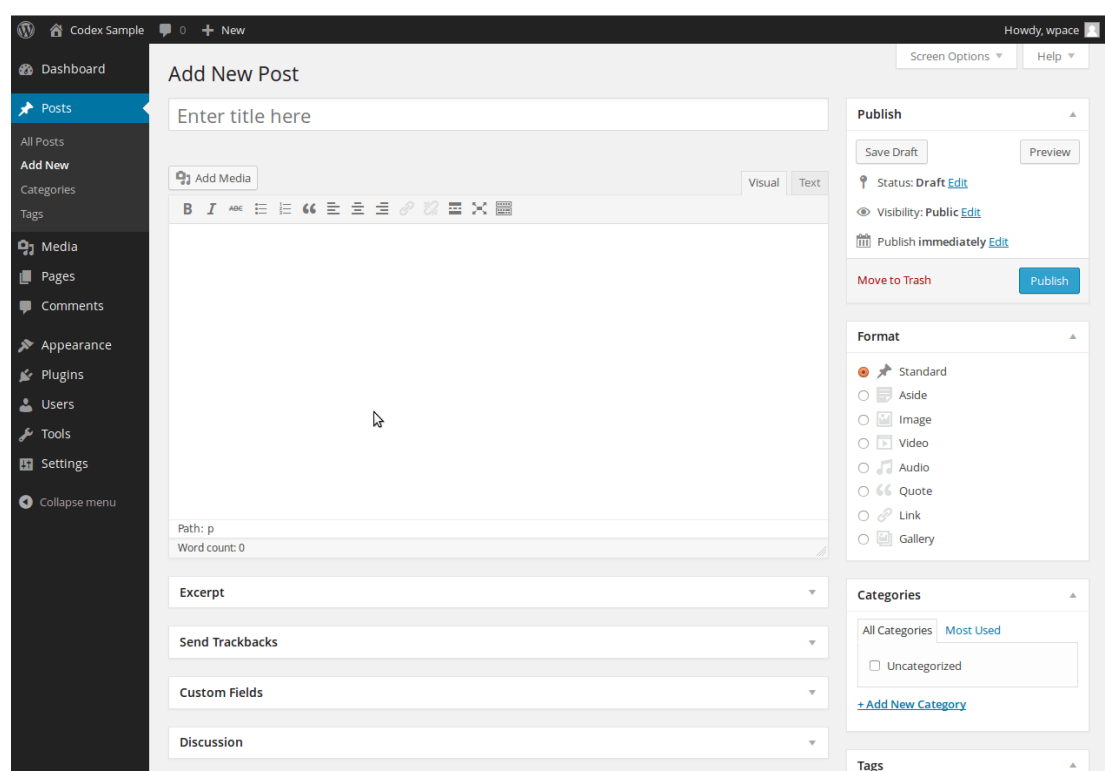
- Backend aplikacji został napisany w języku rust. Jest asynchroniczny, co pozwala na obsługiwanie wielu zapytań na mniejszej ilości wątków. Wykorzystuje asynchroniczny runtime Hyper.
- Współpracuje z bazą danych PostgreSQL. Używa tabel zawierających dane o tabelach w bazie danych, przez co nie musi 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

Wordpres 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 oraz interpretera PHP. Pierwsze wydanie systemu miało miejsce w roku 2003 i od tamtej pory system jest ciągle rozwijany.

Modele w systemie CMS to typy danych, którymi można zarządzać. Domyślnym modelem, w systemie Wordpress jest wpis w blogu. Zawiera między innymi tytuł, tekst wpisu i datę publikacji. Interfejs tworzenia nowego postu został przedstawiony na rysunku 3.1. 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 [7].



Rysunek 3.1: Tworzenie nowego wpisu w systemie Wordpress

Pisanie własnych zapytań SQL jest możliwe, ale podobnie jak definiowanie nie-standardowych 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 systemu, 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 system 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 tworzenie 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 “resolverów”. Pisanie resolverów 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 [2]. 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 [4]. 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 [6] i korzystanie z funkcji full-text search [5].

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, PostgreSQL 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 [10] 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 4.1. 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.

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 4.2.

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.

Listing 4.1: 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 },
    },
  },
};
```

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.

Listing 4.2: 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);
}

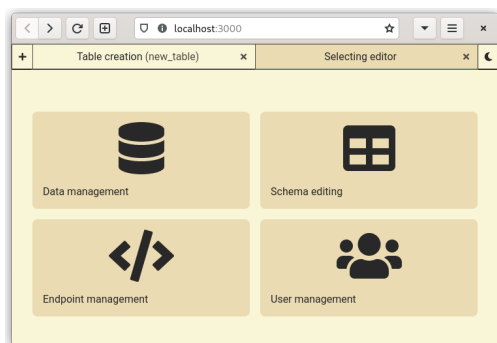
export function setTheme(scheme: string) {
  if (Object.keys(colorschemes).indexOf(scheme) === -1) {
    console.error(`Colorscheme ${scheme} NOT DEFINED!`);
    return;
  }

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

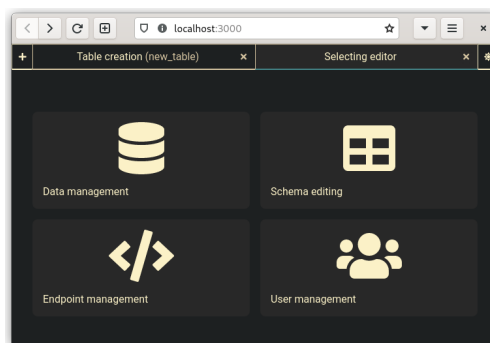
  // Odwrócenie kolorów w diagramie ER mermaid.js, gdy wybrano motyw
  ciemny
  let mermaidFilter = "";
  if (scheme.indexOf("dark") !== -1) mermaidFilter = "invert(1)";
  document.documentElement.style.setProperty("--mermaid-filter",
    mermaidFilter);

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

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



(a) Motyw jasny



(b) Motyw ciemny

Rysunek 5.1: Główny interfejs aplikacji

5. IMPLEMENTACJA KLUCZOWYCH FUNKCJI

5.1. Główna struktura interfejsu

Przygotowano interfejs zgodnie z ustalonymi wymaganiami. W celu ułatwienia pracy w wielu edytorach w tym samym czasie, zaimplementowano system zakładek. Zaimplementowano możliwość przełączania motywu aplikacji z jasnego na ciemny i odwrotnie. Główny interfejs zamieszczono na rysunku 5.1.

5.2. 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.

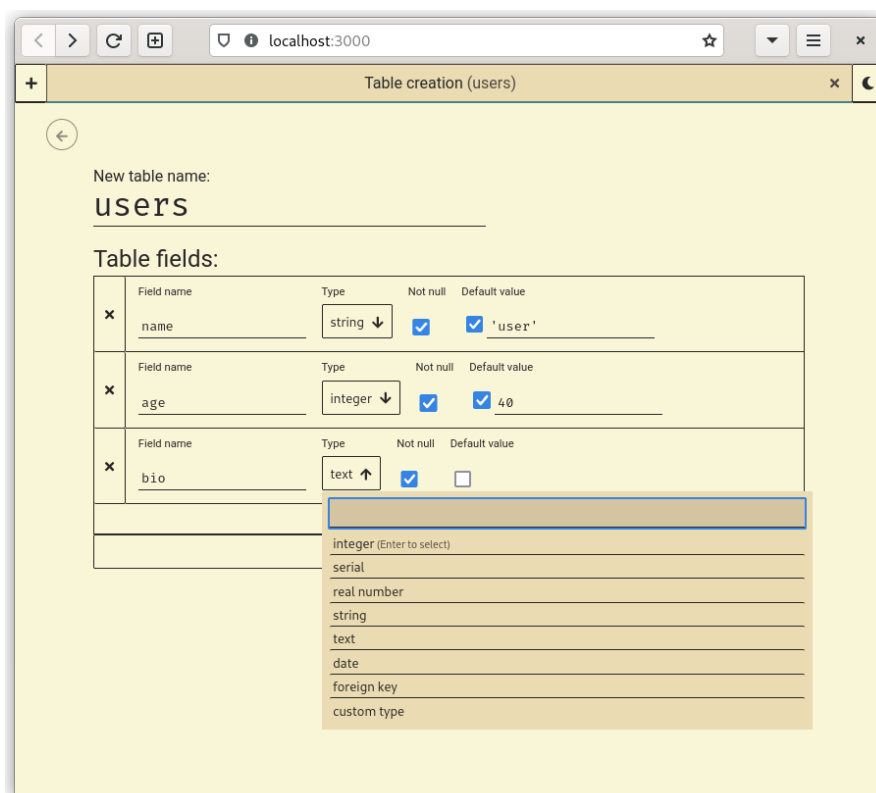
Pojedynczy formularz zawiera następujące elementy:

- 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 5.2.



The screenshot shows a web browser window at localhost:3000 with a tab titled "Table creation (users)". The interface has a yellow background. At the top, there's a "New table name:" label followed by the text "users". Below this is a "Table fields:" section containing a table with three rows. Each row has columns for "Field name", "Type", "Not null", and "Default value". The first row is for "name" with type "string", "Not null" checked, and default value "'user'". The second row is for "age" with type "integer", "Not null" checked, and default value "40". The third row is for "bio" with type "text", "Not null" checked, and an empty default value. Below the "bio" row, a dropdown menu is open, showing a list of data types: "integer (Enter to select)", "serial", "real number", "string", "text", "date", "foreign key", and "custom type".

Field name	Type	Not null	Default value
name	string	<input checked="" type="checkbox"/>	'user'
age	integer	<input checked="" type="checkbox"/>	40
bio	text	<input checked="" type="checkbox"/>	

- integer (Enter to select)
- serial
- real number
- string
- text
- date
- foreign key
- custom type

Rysunek 5.2: Interfejs tworzenia nowego typu danych

5.3. 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 `next val`. 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 5.3.

The screenshot shows a web browser window with the address bar at localhost:3000. The page title is 'Table data management'. Below the title, there is a section 'Managing table' with a dropdown menu showing 'users'. The main content area is divided into two parts: 'Inserting values' and a table view.

Inserting values form:

- id:** Input field with value '8'. A warning message below it says 'Warning, not using default may cause issues!'.
- name:** Input field with value 'Adam Nowak'. The 'Use default' checkbox is checked.
- age:** Input field with value '49'.
- bio:** Textarea with value 'Lubię majsterkować'.
- date_of_birth:** Input field with value '02 / 21 / 1973'. The 'Use null' and 'Use default' checkboxes are unchecked.
- Insert:** Button to submit the form.

Table view:

	id	name	age	bio	date_of_birth
 	2	Adam Nowak	49	Lubię majsterkować	1973-02-21
 	1	Jan Kowalski	49	Lubię łowić ryby	1973-02-21

Rysunek 5.3: Interfejs zarządzania danymi w tabeli

5.4. 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.

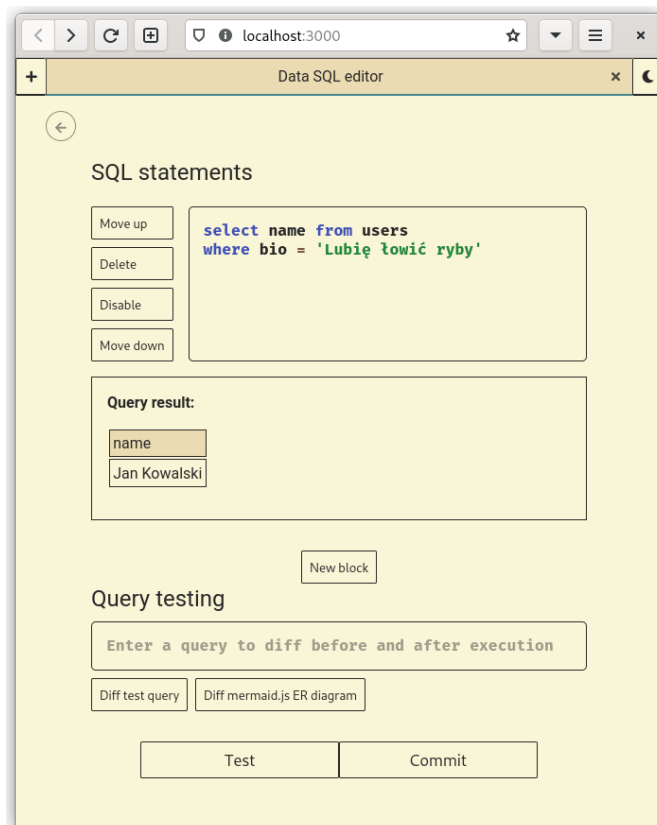
Wybranie opcji “Commit” powoduje wykonanie zapytania w transakcji, która zostanie zatwierdzona, gdy wszystkie zapytania zostaną wykonane bez błędów. Wybranie opcji commit nie pokaże administratorowi informacji zwrotnych poza informacją, czy wykonanie zapytań przebiegło pomyślnie.

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.

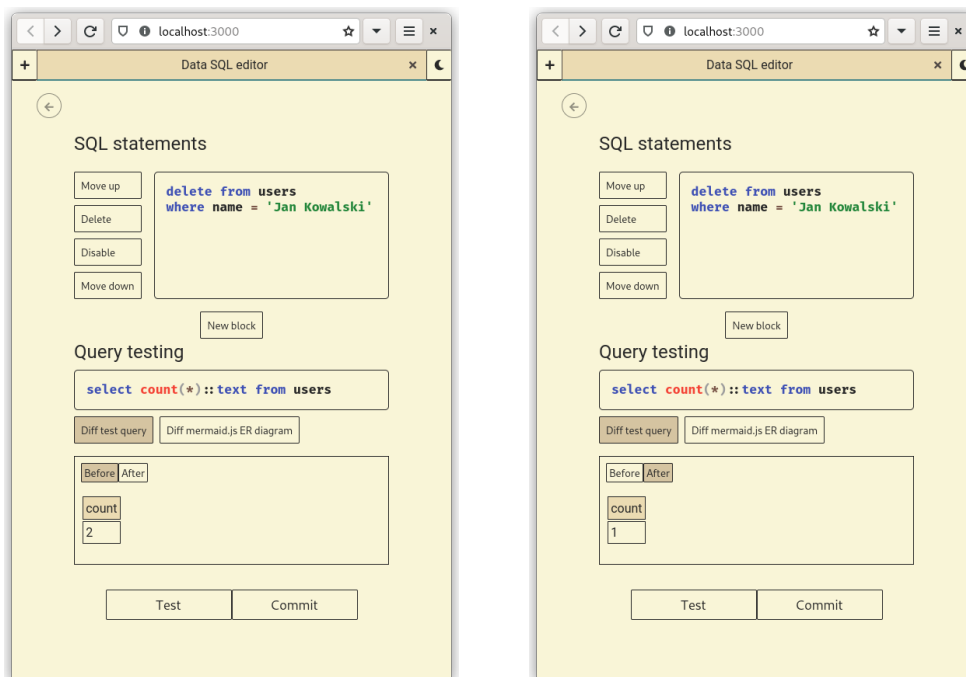
Jeśli administrator zaznaczył opcję “Diff test query”, dane wynikające z zapytania testowego zostaną wyświetlone w specjalnym oknie pod tekstem zapytania testowego z możliwością przełączania pomiędzy wynikiem zapytania przed wykonaniem listy zapytań i po wykonaniu listy zapytań. Interfejs widać na rysunku 5.5.

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 5.6, a widok pełnoekranowy widać na rysunku 5.7.

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.



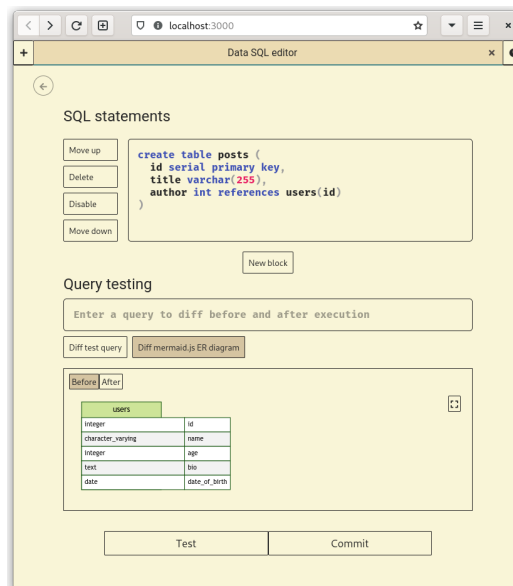
Rysunek 5.4: Edytor SQL wykonywanego natychmiast



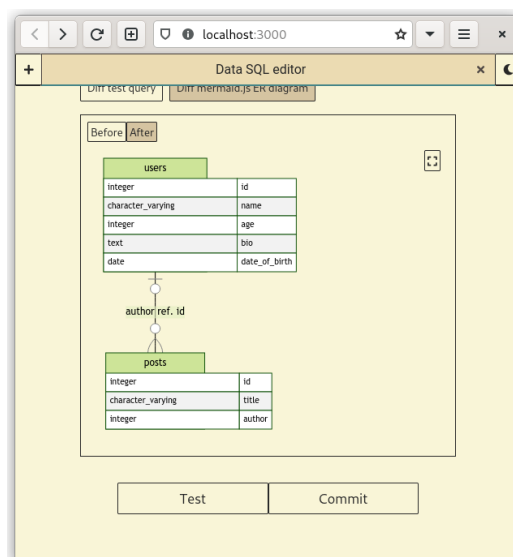
(a) Przed wykonaniem

(b) Po wykonaniu

Rysunek 5.5: Porównanie zapytania testowego

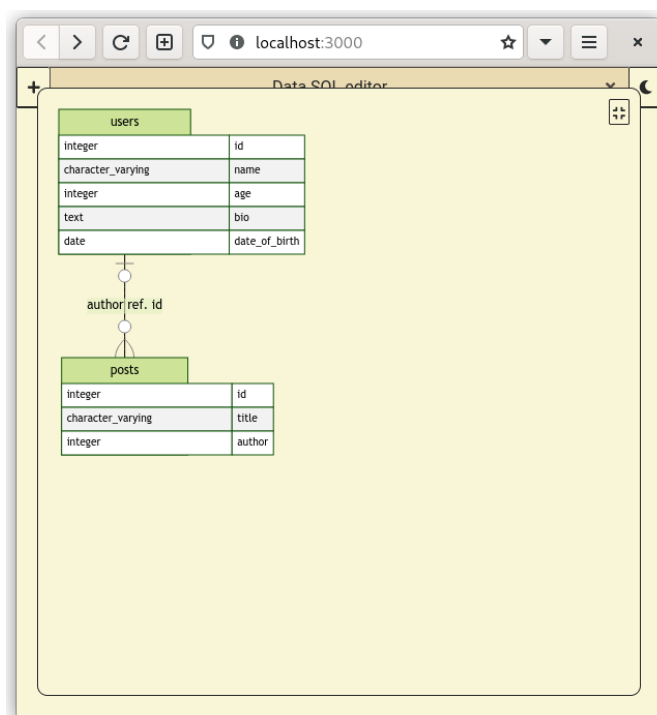


(a) Przed wykonaniem



(b) Po wykonaniu

Rysunek 5.6: Porównanie diagramu ER



Rysunek 5.7: Widok pełnoekranowy diagramu

5.5. 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 5.1.

Sparsowane zapytania SQL składają się z SQL zawierającego odniesienia do parametrów w składni specyficznej dla bazy danych postgresql [3] i listy ciągów znaków, które umożliwią znalezienie zmiennej w czasie wykonywania drzewa zapytań SQL.

Listing 5.1: 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"]);
}
```

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 metodzie HTTP.

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 5.8.

The screenshot shows a web browser window with the address bar set to 'localhost:3000'. The page title is 'Endpoint management'. Below the title, there is a button 'Create new' with a dropdown arrow. The main content area is titled 'Creating new endpoint'. It contains the following fields and controls:

- Path:** A text input field containing '/posts-by-username'.
- Method:** A dropdown menu currently showing 'GET'.
- Allowed groups:** A section with a 'New allowed group:' input and an 'Add' button. Below it, a list shows 'x PUBLIC,'.
- Name:** A text input field containing 'users'.
- SQL Query:** A text area containing the query:

```
select id::text as private_id, name from users
where name like '%' || ${req.name} || '%'
```
- Buttons:** A 'New child' button is located to the right of the first query.
- Second Entry:** Below the first query, there is another section with 'Name:' containing 'posts' and a corresponding SQL query:

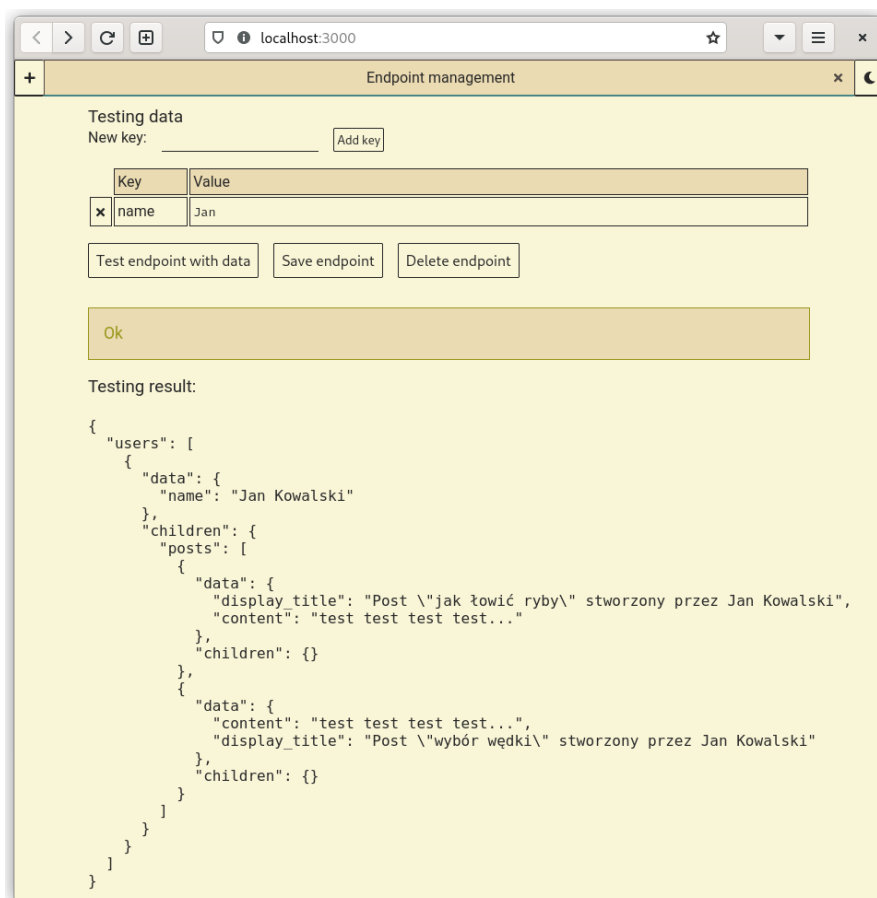
```
select 'Post ' || title || ' stworzony przez ' || ${super.name}
as display_title,
content from posts
where author::text = ${super.private_id}
```
- Buttons:** 'New child' and 'Delete' buttons are located to the right of the second query.
- Footer:** A 'New independent query' button is at the bottom left.

Rysunek 5.8: Interfejs tworzenia nowego punktu końcowego

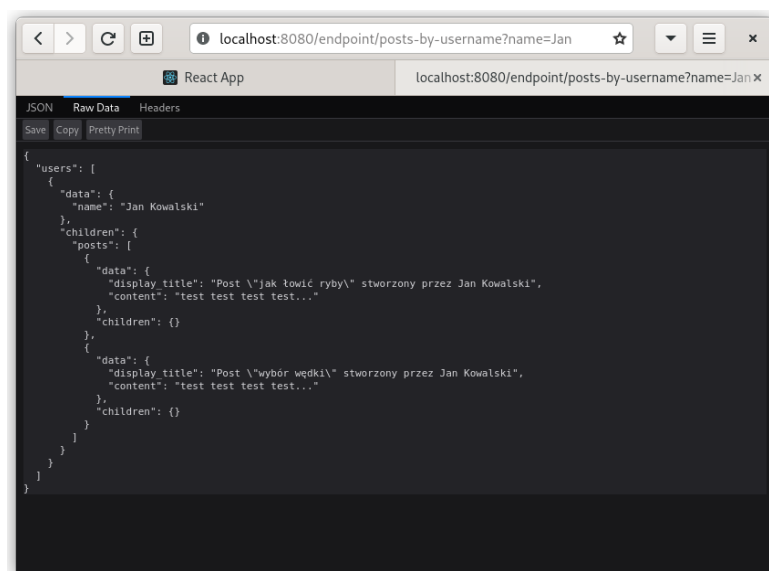
Program implementuje funkcjonalność mającą na celu umożliwienie administratorowi wygodnego testowania tworzonych punktów końcowych. Pod edytorem punktu końcowego znajduje się edytor danych testowych. Umożliwia on tworzenie mapy zmiennych, które będą traktowane, jak wartości formularza przychodzącego 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 5.9.

Zapisane punkty końcowe będą dostępne pod adresem /endpoint/<ścieżka punktu końcowego>. Wywołanie zapisanego punktu końcowego z przeglądarki Firefox - poza panelem administratora widać na rysunku 5.10.



Rysunek 5.9: Interfejs testowania tworzonego punktu końcowego



Rysunek 5.10: Wywołanie punktu końcowego w przeglądarce Firefox

5.6. 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 5.2.

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 5.2: 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 5.3. Diagram obrazujący działanie algorytmu przedstawiono na rysunku 5.11.

Algorytm wykonywania drzewa zapytań SQL można opisać następująco:

1. Dla każdego zapytania SQL przypisanego do punktu końcowego,

Listing 5.3: 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
        ));
    }
}
```

(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.

GET /endpoint/posts-with-limit?limit=50

SELECT id as private_id, name, age FROM Users;

SELECT title, text FROM Posts where
Posts.user_id = \${super.private_id}
FETCH FIRST \${req.limit} ROWS ONLY;

SELECT 'A post titled ' || \${super.title} ||
' by user ' || \${super.super.name}
as result_title;

Rysunek 5.11: Diagram obrazujący działanie algorytmu znajdowania wartości zmiennych

Implementacja algorytmu została zamieszczona na listingu 5.4.

Listing 5.4: Algorytm wykonywania drzewa zapytań SQL

```
#[async_recursion]
pub async fn execute(
    &mut self,
    #[cfg(test)] mock_exec_service: &mut ExecutionMockService,
    #[cfg(not(test))] transaction: &mut Transaction<'_, Postgres>,
    endpoint_infos: &Vec<EndpointInfo>,
) -> Result<HashMap<String, Vec<ExecutionResult>>> {
    let mut final_results = HashMap::<String, Vec<ExecutionResult>>::new();

    for query in endpoint_infos {
        #[cfg(not(test))]
        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)?;

            #[cfg(not(test))]
            {
                exec = exec.bind(val);
            }
            #[cfg(test)]
            {
                mock_exec_service.bind(&val);
            }
        }

        #[cfg(test)]
        let results = mock_exec_service.simulate_call(&query.parsed_sql);

        #[cfg(not(test))]
        let results = exec
            .fetch_all(&mut *transaction)
            .await?
            .into_iter()
            .map(|it| it.into_map())
            .collect::<Vec<_>>();

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

            #[cfg(test)]
            let children_results = self.execute(mock_exec_service, &query.children)
                .await?;
            #[cfg(not(test))]
            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 bardziej skomplikowanej logiki biznesowej programu, zawartej w module `algorithms` zostało zweryfikowane testami jednostkowymi. Wynik wykonania testów jednostkowych można zobaczyć na listingu 6.1.

Listing 6.1: Wykonanie testów jednostkowych

```
running 12 tests
test algorithms::mermaid_diagram_generation::tests::works ... ok
test algorithms::sql_variable_parser::tests::bind_vec ... ok
test algorithms::sql_variable_parser::tests::parsing_endpoint_info ... ok
test algorithms::endpoint_execution::test::error_when_cant_find_req_variable
... ok
test algorithms::endpoint_execution::test::
  error_when_cant_find_super_variable ... ok
test algorithms::endpoint_execution::test::request_variables_work ... ok
test algorithms::endpoint_execution::test::super_variables_work ... ok
test algorithms::endpoint_execution::test::error_when_too_many_supers ... ok
test algorithms::sql_variable_parser::tests::parsing_test ... ok
test algorithms::sql_variable_parser::tests::not_closed_panics - should
  panic ... ok
test algorithms::sql_variable_parser::tests::parsing_multiple ... ok
test algorithms::endpoint_execution::test::it_works ... ok

test result: ok. 12 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
  finished in 0.00s
```

Działanie testów jednostkowych można opisać następująco:

- `mermaid_diagram_generation::tests::works` - test sprawdza poprawność działania algorytmu generującego diagram ER w składni `mermaid.js`. Do implementacji podawana jest przykładowa struktura tabel. Sprawdzane jest, czy wygenerowany diagram jest poprawny.
- `sql_variable_parser::tests::parsing_test` - test sprawdza podstawową funkcję parsującą SQL wykorzystywaną do implementacji bardziej skomplikowanej funkcjonalności parsera.
- `sql_variable_parser::tests::parsing_multiple` - test sprawdza poprawne działanie parsera SQL, gdy w jednym zapytaniu znajduje się większa ilość odwołań do zmiennych.

- `sql_variable_parser::tests::bind_vec` - test sprawdza poprawność działania funkcji, która przekształca tablicę mieszającą zawierającą dostępne zmienne i sparsowany SQL na tablicę wartości, które powinny zostać wysłane do bazy danych.
- `sql_variable_parser::tests::parsing_endpoint_info` - test sprawdza poprawność funkcji parsującej strukturę danych, którą należy wysłać do serwera w celu stworzenia nowego punktu końcowego.
- `sql_variable_parser::tests::not_closed_panics` - test sprawdza, czy program zwraca błąd, gdy do funkcji parsującej SQL podane zostanie zapytanie z niepoprawną składnią (niedomknięta klamra).
- `endpoint_execution::test::it_works` - test sprawdza podstawowe działanie modułu wykonującego drzewo zapytań SQL. Do implementacji podawane jest drzewo składające się z jednego liścia. Sprawdzane jest, czy zapytanie umieszczone w liściu zostało wykonane.
- `endpoint_execution::test::request_variables_work` - test sprawdza poprawne działanie, gdy drzewo zapytań SQL zawiera zapytanie z odniesieniem do zmiennej pochodzącej z zapytania HTTP. Sprawdzane jest, czy do bazy danych wysyłane jest poprawne zapytanie SQL i poprawne zmienne.
- `endpoint_execution::test::super_variables_work` - test sprawdza poprawne działanie, gdy drzewo zapytań SQL zawiera zapytanie z odniesieniem do zmiennej pochodzącej z zapytania nadrzędnego. Do modułu podawane jest drzewo z dwoma węzłami. Węzeł podrzędny odnosi się do zmiennej będącej wynikiem wykonania węzła nadrzędnego. Sprawdzana jest poprawna kolejność wysyłanych zapytań SQL oraz poprawność danych wysyłanych razem z zapytaniem podrzędnym.
- `endpoint_execution::test::error_when_cant_find_req_variable` - test sprawdza, czy program zwróci błąd, gdy zapytanie odnosi się do zmiennej wysyłanej z zapytaniem HTTP, która nie istnieje.
- `endpoint_execution::test::error_when_cant_find_super_variable` - test sprawdza czy program zwróci błąd, kiedy węzeł podrzędny zawiera odniesienie do zmiennej z węzła nadrzędnego, która nie istnieje.
- `endpoint_execution::test::error_when_too_many_supers` - test sprawdza czy program zwróci błąd, gdy zmienna zawarta w drzewie zapytań sql odnosi się do zmiennej, która nie może istnieć, bo nazwa zmiennej zawiera więcej prefiksów “super.”, niż istnieje węzłów nadrzędnych. Do modułu podawane jest drzewo

z dwoma węzłami. Węzeł podrzędny odnosi się do zmiennej z dwoma prefiksami “super.”, ale ma tylko jeden węzeł nadrzędny.

6.2. Testy integracyjne

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 new-man. Pozwala to na automatyczne wykonywanie testów.

Wykonanie testów integracyjnych widać na listingu ??.

Wykonanie testów integracyjnych powinno odbywać się po uruchomieniu aplikacji backend po raz pierwszy z pustą bazą danych. Stan aplikacji zostaje zachowany podczas całego działania testów integracyjnych. Działanie to można opisać następująco:

1. Login admin - test sprawdza, czy możliwe jest poprawne zalogowanie do konta administratora. Token administratora jest zapisywany i korzystają z niego pozostałe testy.
2. Create users - test wysyła do serwera polecenie stworzenia użytkowników, sprawdza czy odpowiedź jest poprawna.
3. Create table form - test wysyła do serwera polecenie stworzenia tabeli. Polecenie nie zawiera zapytania SQL. Jest to wariant polecenia, który wysyła aplikacja frontend przy tworzeniu nowego typu danych za pomocą formularza.
4. Get table info - test wysyła do serwera zapytanie o informacje o tabelach zawartych w bazie danych. Test sprawdza, czy odpowiedź zawiera dane o tabeli stworzonej w teście 3.
5. Insert data - test wysyła do serwera polecenie dodania danych do tabeli. Jest to wariant polecenia, jakie wysyła aplikacja frontend przy dodawaniu danych za pomocą formularza.
6. Get data - test wysyła do serwera zapytanie o informacje zawarte w tabeli stworzonej w teście 3. Jest to wariant zapytania, jakie wysyła aplikacja frontend w edytorze danych opartym o formularze. Test sprawdza, czy dane są takie same, jak te dodane do tabeli w teście 5.
7. Execute query - test wysyła do serwera zapytanie HTTP z zapytaniami SQL. Test wysyła zapytanie tworzące nowe dane w tabeli stworzonej w teście 3 oraz zwracające stworzone dane. Zapytanie zostało umieszczone poniżej.

```
insert into test_table_one (name, age)
values ('Adam Nowak', 42)
returning name, age::text
```

Ponadto, zapytanie zawiera zapytanie SQL wykonywane przed i po wykonaniu drzewa zapytań. Jest to zapytanie pobierające wartości z kolumny name z tabeli test_table_one. Test sprawdza czy zapytanie zostało poprawnie wykonane oraz czy wartości zapytania wykonywanego przed i po drzewie zapytań są poprawne.

8. Test endpoint - test wysyła do serwera zapytanie będące wariantem zapytania, jakie wysyła aplikacja frontend w celu przetestowania tworzonego punktu końcowego. Test sprawdza poprawność odesłanej przez serwer odpowiedzi.
9. Create endpoint - test wysyła do serwera zapytanie będące wariantem zapytania, które aplikacja frontend wysyła w celu stworzenia punktu końcowego. Test sprawdza, czy odpowiedź od serwera ma kod HTTP 200 OK.
10. Call endpoint as admin - test wywołuje punkt końcowy stworzony w teście 8. Test sprawdza, czy wysłana przez serwer odpowiedź zawiera poprawne dane. Test sprawdza, czy administrator może wywoływać dowolne punkty końcowe nawet, gdy nie ma go na liście użytkowników, którzy mają dostęp do punktu końcowego.
11. Get endpoints - test wysyła do serwera zapytanie będące wariantem zapytania wysyłanego przez aplikację frontend w celu pobrania od serwera informacji o istniejących punktach końcowych. Test sprawdza, czy w odpowiedzi znajdują się dane o punkcie końcowym stworzonym w teście 8.
12. Create manager user - test tworzy użytkownika należącego do grupy MANAGERS. Ta grupa ma dostęp do punktu końcowego stworzonego w teście 8.
13. Try call worker - test próbuje wywołać punkt końcowy przy pomocy konta użytkownika stworzonego w teście 2. Użytkownik nie ma uprawnień do wywołania tego punktu końcowego. Test sprawdza, czy zwrócono kod HTTP 401 oznaczający brak dostępu.
14. Call manager - test wywołuje punkt końcowy stworzony w teście 8 przy użyciu konta kierownika stworzonego w teście 11. Użytkownik ma możliwość wywołania punktu końcowego. Test sprawdza, czy punkt końcowy został wywołany.

6.3. Testy manualne

Poprawne działanie najważniejszych funkcji aplikacji frontend zostało zweryfikowane za pomocą testów manualnych. Scenariusze testów manualnych zostały przedstawione poniżej.

1. Testowanie poprawnego działania systemu kart.

(a) Testowanie poprawnego tworzenia nowej karty.

Przebieg testu:

- i. Wciśnięcie przycisku “+” po lewej stronie paska kart.

Spodziewany wynik:

Jeśli ostatnia karta znajduje się na widoku startowym, powinna zostać wybrana jako aktywna. Jeśli już była aktywna, powinna pozostać aktywna.

Jeśli ostatnia karta nie znajduje się na widoku startowym, powinna zostać stworzona nowa karta znajdująca się na widoku startowym. Nowa karta powinna zostać wybrana jako aktywna.

(b) Testowanie poprawnego zamykania karty.

Przebieg testu:

- i. Wciśnięcie przycisku “x” po prawej stronie karty, lub wciśnięcie dowolnego miejsca karty środkowym przyciskiem myszki.

Spodziewany wynik:

Jeśli karta, którą spróbowano zamknąć jest jedyną kartą i nie jest na widoku startowym, nie powinno się nic stać. Jeśli karta jest jedyną i nie jest na widoku startowym, w karcie powinien zostać otworzony widok startowy.

Jeśli karta, którą spróbowano zamknąć nie jest jedyną kartą, karta powinna przestać istnieć. Dowolna sąsiedna karta do karty zamkniętej powinna zostać wybrana jako aktywna.

(c) Testowanie poprawnego zachowania stanu w karcie.

Przebieg testu:

- i. Otworzenie w dowolnej karcie komponentu zawierającego stan (na przykład pole tekstowe).
- ii. Zmiana stanu komponentu (na przykład wpisanie dowolnego tekstu do pola tekstowego).
- iii. Przełączenie aktywnej karty na dowolną inną kartę.
- iv. Przełączenie aktywnej karty na kartę otwartą w kroku i.

Spodziewany wynik:

Stan komponentu po ponownym otwarciu karty powinien być identyczny do stanu przed pierwszym przełączeniem aktywnej karty.

2. Testowanie poprawnego działania edytora zapytań SQL wykonywanych natychmiast.

(a) Testowanie poprawnego tworzenia nowego zapytania.

Przebieg testu:

- i. Wciśnięcie przycisku “New block” pod ostatnim polem tekstowym z zapytaniem SQL.

Spodziewany wynik:

Pod ostatnim edytorem zapytania SQL pojawi się nowy edytor zapytania SQL.

(b) Testowanie poprawnego usuwania zapytania SQL.

Przebieg testu:

- i. Wciśnięcie przycisku “Delete” po lewej stronie pola tekstowego z zapytaniem SQL.

Spodziewany wynik:

Jeśli edytor jest jedynym edytorem, treść zapytania SQL zostanie usunięta.

Jeśli edytor nie jest jedynym edytorem, zostanie usunięty z listy edytorów.

(c) Testowanie poprawnego wyłączania zapytania SQL.

Przebieg testu:

- i. Wciśnięcie przycisku “Disable” po lewej stronie pola tekstowego z zapytaniem SQL.
- ii. Wciśnięcie przycisku “Enable” po lewej stronie pola tekstowego z zapytaniem SQL.

Spodziewany wynik:

Po wciśnięciu przycisku “Disable”, tło zostanie przyciemnione w trybie jasnym lub rozjaśnione w trybie ciemnym. Całe pole tekstowe stanie się przezroczyste. Pisanie w polu tekstowym nie będzie możliwe.

Po wciśnięciu przycisku “Enable”, działanie przycisku “Disable” zostanie odwrócone.

3. Testowanie poprawnego działania edytora punktów końcowych.

- (a) Testowanie poprawnego tworzenia nowego zapytania.

Przebieg testu:

- i. Wciśnięcie przycisku “New independent query” pod ostatnim polem tekstowym z zapytaniem SQL.

Spodziewany wynik:

Pod ostatnim zapytaniem zostanie stworzone nowe zapytanie będące korzeniem nowego drzewa niezależnego od pozostałych zapytań.

- (b) Testowanie poprawnego dodawania zapytania podrzędnego.

Przebieg testu:

- i. Wciśnięcie przycisku “New child” pod dowolnym polem tekstowym z zapytaniem SQL.

Spodziewany wynik:

Pod zapytaniem zostanie dodane nowe zapytanie podrzędne do zapytania, którego przycisk wciśnięto. Zapytanie będzie posiadało większy margines z lewej strony. Zapytanie będzie znajdowało się po prawej stronie przedłużenia ramki pola tekstowego z zapytaniem nadrzędnym.

Jeśli zapytanie, którego przycisk wciśnięto posiadało przycisk “Delete”, przycisk ten zniknie i będzie widoczny ponownie dopiero, gdy zostaną usunięte wszystkie zapytania podrzędne do tego zapytania.

- (c) Testowanie poprawnego usuwania zapytania SQL.

Przebieg testu:

- i. Wciśnięcie przycisku “Delete” pod dowolnym polem tekstowym z zapytaniem SQL.

Spodziewany wynik:

Jeśli zapytanie było jedynym zapytaniem przypisanym do punktu końcowego i posiadało treść w polu tekstowym, treść pola tekstowego zostanie wykasowana.

Jeśli zapytanie było jedynym zapytaniem przypisanym do punktu końcowego i nie posiadało treści w polu tekstowym, nic się nie stanie.

Jeśli zapytanie nie było jedynym zapytaniem, całe pole tekstowe zostanie wykasowane.

- (d) Testowanie poprawnego dodawania testowych danych pochodzących z zapytania.

Przebieg testu:

- i. Wpisanie dowolnej nazwy klucza w polu tekstowym “New key”.

ii. Wciśnięcie przycisku “Add key”.

Spodziewany wynik:

Do tabeli znajdującej się pod polem tekstowym zostanie dodany nowy rząd.

W kolumnie “Key” znajdzie się klucz wpisany do pola tekstowego.

7. PODSUMOWANIE

Według aplikacji GNOME System Monitor,
`python -m http.server` - 8.3MB
1.9MB

Przygotowane w rozdziale 6 testy potwierdzają, że przygotowany system spełnia założenia projektowe.

Udało się przygotować system, który pozwala na wykonywanie dowolnych działań na bazie danych za pomocą natywnych zapytań SQL. Program udostępnia administratorom funkcje niedostępne w innych systemach CMS i headless CMS dostępnych na rynku.

Aplikacja posiada większość funkcji typowego systemu CMS, jak tworzenie nowego typu danych i zarządzanie danymi za pomocą formularzy, a nie przez pisanie SQL. Brakuje jednak możliwości zmiany struktury tabel za pomocą formularza. Niewygodne jest również dodawanie kluczy obcych, które wymaga pamiętania nazwy tabeli, do której chce się odnieść oraz nazwy kolumny przechowującej klucze główne.

Funkcją dostępną w typowych systemach CMS, której zupełnie brakuje w programie jest zarządzanie mediami. Należałoby się zastanowić, jak połączyć system, którego głównym sposobem zarządzania jest pisanie natywnych zapytań SQL z systemem zarządzania mediami.

Można by użyć do tego celu jednego z typów danych bazy Postgres przeznaczonego do przechowywania danych binarnych. Popularnym rozwiązaniem jest przechowywanie danych binarnych w systemie plików i przechowywanie nazwy pliku w bazie danych. Takie rozwiązanie utrudniłoby jednak administratorom zarządzanie danymi. W celu ustalenia optymalnego rozwiązania problemu, wymagane są dalsze badania.

Kolejną funkcją, jaką warto w przyszłości zaimplementować jest udostępnienie użytkownikom nie będącym administratorami okrojonego panelu administratora. Można by użyć do tego komponentów z panelu administratora, lub stworzyć niezależny panel, który zawiera interfejs przeznaczony do wywoływania punktów końcowych przygotowanych przez administratorów. Zaimplementowanie obydwu pomysłów stworzyłoby interfejs nie gorszy, niż w innych popularnych systemach CMS.

Kolejną funkcją, jaką warto w przyszłości zaimplementować jest zwracanie danych przy wywołaniu punktu końcowego nie jako dokument json, lecz jako szablon HTML wypełniony danymi wynikowymi. Taka funkcja byłaby prosta w implementacji. Najtrudniejszą częścią implementacji byłoby przygotowanie interfejsu zarządzania szablonami w panelu administratora.

Literatura dotycząca systemów CMS jest dość uboga. Większość dostępnych książek skupia się na profesjonalnych systemach CMS, z których korzystają firmy publikujące treści multimedialne. Książki opisujące systemy CMS przystosowane do mniejszych stron internetowych zazwyczaj skupiają się na konkretnym systemie i nie poruszają ogólnych zagadnień dotyczących systemów zarządzania treścią.

Jeśli chodzi o systemy headless CMS, literatura jest jeszcze bardziej uboga. Z tego powodu, jeśli omawiane były systemy headless CMS, praca cytowała głównie strony internetowe.

Literatura

- [1] Cloud-first headless cms: What it is and why you should use it. <https://www.cmscritic.com/cloud-first-headless-cms-what-it-is-and-why-you-should-use-it/>. Dostęp: 2020-12-31.
- [2] Deployment - strapi developer docs. <https://docs.strapi.io/developer-docs/latest/setup-deployment-guides/deployment.html>. Dostęp: 2020-12-10.
- [3] Postgres documentation - sql prepare. <https://www.postgresql.org/docs/current/sql-prepare.html>. Dostęp: 2020-12-30.
- [4] Postgrest documentation. <https://postgrest.org/en/stable/index.html>. Dostęp: 2020-12-10.
- [5] Postgrest documentation - full-text search. <https://postgrest.org/en/stable/api.html#fts>. Dostęp: 2020-12-10.
- [6] Postgrest documentation - json columns. <https://postgrest.org/en/stable/api.html#json-columns>. Dostęp: 2020-12-10.
- [7] Wordpress theme handbook - post types. <https://developer.wordpress.org/themes/basics/post-types/#custom-post-types>. Dostęp: 2020-12-10.
- [8] Aaron Brazell. Wordpress bible. John Wiley & Sons, 2010.
- [9] Andreas Mauthe and Peter Thomas. Professional Content Management Systems. John Wiley & Sons, Ltd, jan 2004.
- [10] Michael Mikowski and Josh Powell. Single page web applications: JavaScript end-to-end. Simon and Schuster, 2013.

Spis rysunków

3.1	Tworzenie nowego wpisu w systemie Wordpress	6
5.1	Główny interfejs aplikacji	13
5.2	Interfejs tworzenia nowego typu danych	14
5.3	Interfejs zarządzania danymi w tabeli	15
5.4	Edytor SQL wykonywanego natychmiast	17
5.5	Porównanie zapytania testowego	17
5.6	Porównanie diagramu ER	18
5.7	Widok pełnoekranowy diagramu	19
5.8	Interfejs tworzenia nowego punktu końcowego	22
5.9	Interfejs testowania tworzonego punktu końcowego	23
5.10	Wywołanie punktu końcowego w przeglądarce Firefox	23
5.11	Diagram obrazujący działanie algorytmu znajdowania wartości zmien- nych	26

Spis listingów

4.1	Struktura danych przechowująca dane o edytorach	11
4.2	Kod odpowiedzialny za przełączanie motywu	12
5.1	Test sprawdzający poprawność parsowania SQL zawierającego odnie- sienia do zmiennych	21
5.2	Struktura danych, której metody wykonują SQL przy wywołaniu punktu końcowego	24
5.3	Algorytm znajdowania wartości zmiennej	25
5.4	Algorytm wykonywania drzewa zapytań SQL	27
6.1	Wykonanie testów jednostkowych	29