Implementacja architektury serwisu społecznościowego

(Implementation of social service architecture)

Marek Kwaśny

Praca inżynierska

Promotor: dr Wiktor Zychla

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

22 września 2021

Streszczenie

Celem pracy było utworzenie platformy społecznościowej i jednoczesne badanie wzorców architektury stojących za największymi serwisami na świecie. Projekt został wykonany za pomocą nowoczesnych rozwiązań. System składa się z dwóch części: aplikacji serwerowej napisanej w Node.js oraz klienckiej utworzonej z użyciem popularnego frameworka React. Komunikacja pomiędzy nimi została zaimplementowana przy pomocy grafowego języka zapytań. Badanie wzorców zostało podzielone w sposób analogiczny do części praktycznej.

The purpose of the study was both to develop a social media website and research the already existing solutions in the biggest platforms in the world. The project was developed with the use of modern technologies. We can define two main parts of the system: server-side application running Node.js and client-side made with the use of the popular framework called React. The communication between two of them was implemented with the use of graph query language. The architecture's research was splitted in the same way as in the practical part of the study.

Spis treści

Rozdział 1.

Wprowadzenie

„Serwis internetowy współtworzony przez społeczność internautów o podobnych zainteresowaniach, który umożliwia kontakt z przyjaciółmi i dzielenie się informacjami, zainteresowaniami itp." [1] - tak słownik języka polskiego definiuje serwis lub portal społecznościowy. Współcześnie wyżej określony typ produktu wydaje się być normalnością, gwarantem na „wyciągnięcie ręki". Firmy odpowiadające za najpopularniejsze witryny są światowymi gigantami, a rozwój technologii webowych w dużej mierze zależy właśnie od nich. Dzisiaj wszyscy jesteśmy „połączeni" niezależnie od kultury, narodowości, wieku czy odległości, kontakt z drugą osobą to kwestia wyłącznie sformułowania swojej wypowiedzi czy też wiadomości. Należy jednak pamiętać, że jesteśmy świadkami rozkwitu nowej ery, ery cyfrowej komunikacji - możliwość interakcji z innymi ludźmi nie jest już tylko domeną portali społecznościowych. Coraz więcej serwisów niezależnie od tematyki czy funkcjonalności implementuje rozwiązania znane nam wszystkim z owych stron. Przesycenie treściami serwuje nowe wyzwania, których rozwiązania powstają tu i teraz. Algorytmy dopasowywania treści, modelowanie użytkowników, schematy predykcji czy inteligentne wczytywanie danych - to tylko nieliczne przykłady niekończącego się biegu po atencję odbiorcy. Ten kto dostarczy żądaną treść przy najmniejszym nakładzie pracy konsumenta wygra jego czas, a wystarczy tylko tyle, by stać się kolejnym światowym gigantem stojącym na czele nie tylko słupków giełdowych, ale także współczesnym pionierem nowych rozwiązań sieciowych.

Tematem pracy jest jednoczesne stworzenie aplikacji internetowej będącej prostą wersją serwisu społecznościowego oraz badanie istniejących wzorców architektury wśród najpopularniejszych portali tego typu. Naturalną kolejnością rzeczy jest wykorzystanie tej wiedzy w praktyce z próbą przeprowadzenia własnej implementacji. Podczas wyboru zestawu technologii starałem się kierować popularnością oraz nowoczesnością danych narzędzi tak, by samo ich wykorzystanie stanowiło dla mnie walor dydaktyczny, który będę mógł wykorzystać w przyszłości. Realizacja pracy doprowadziła do powstania dwóch części aplikacji - serwerowej i klienckiej. Ta pierwsza została stworzona za pomocą technologii Node.js [2], a druga przy użyciu popular-

nej biblioteki React [3]. Stanowią one dwa różne, niezależne od siebie byty, które komunikują się między sobą za pomocą grafowego języka zapytań. Serwis został udostępniony na platformie Heroku [4] i jest dostępny pod adresem: `http://thesis-marekkwasny.herokuapp.com/`.

W swojej pracy prezentuję różne aspekty tematu architektury serwisów społecznościowych. Ze względu na kompleksowość tego obszaru ustrukturyzowałem treści w sposób bliski ich implementacji. Rozdział drugi odnosi się do historii portali społecznościowych. Omawiam w nim wybrane przeze mnie wydarzenia, nakreślam trzy różne serwisy społecznościowe oraz prezentuję programistyczne początki Facebooka. Rozdział trzeci jest pierwszym w pełni technicznym rozdziałem. Stanowi on katalog wzorców architektury stosowanych po stronie serwera. Staram się w nim zarówno prezentować ideę, jak i opisywać realne wykorzystanie. Rozdział czwarty w analogiczny sposób odnosi się do strony klienckiej oprogramowania. W przypadku rozdziału piątego prezentuję własne przykłady implementacji wybranych wzorców zawartych w wcześniej wspomnianych katalogach. Dzięki takiemu podejściu w merytoryczny sposób mogę przedstawić rozwiązania widoczne w portalach społecznościowych - od pomysłu, przez analizę, aż po wykonanie. Rozdział szósty zawiera informacje techniczne potrzebne do lokalnej instalacji projektu.

Rozdział 2.

Historia

Przegląd historii [7]:

- **Lata 90.** - jesteśmy świadkami powstawania pierwszych portali społecznościowych. Serwisy takie jak Classmates, czy LunarStorm dają możliwość odnowienia starych znajomości. Ponadto zaczynają pojawiać się pierwsze, szeroko dostępne komunikatory natychmiastowych wiadomości jak MSN Messenger lub Yahoo! Messenger.
- **2003** - powstaje LinkedIn, Myspace oraz Skype.
- **2004** - założenie Facebooka, początkowo wyłącznie dla studentów Harvardu.
- **2005** - światło dzienne ujrzał popularny dzisiaj agregator wiadomości - Reddit. Facebook udostępnił możliwość dzielenia się zdjęciami.
- **2006** - premiera Twittera, Facebook otwiera się na wszystkich ludzi powyżej 13 roku życia. W tym samym również roku powstaje VK - rosyjska alternatywa dla produktu Marka Zuckeberga oraz polski serwis społecznościowy - nasza-klasa, stworzony przez studentów Instytutu Informatyki Uniwersytetu Wrocławskiego.
- **2007** - Justin.tv (obecnie znane jako Twitch) - platforma służąca do transmitowania na żywo stworzona z myślą o pasjonatach elektronicznej rozrywki.
- **2008** - powstaje serwis społecznościowy dla programistów - Stack Overflow.
- **2010** - rok fotografii, jesteśmy świadkami uruchomienia Pinterest oraz Instagram.
- **2012** - Snapchat - platforma do komunikacji wyłącznie przy użyciu zdjęć. Premierę ma również jeden z najpopularniejszych współcześnie serwisów randkowych - Tinder. Facebook kupuje Instagram.
- **2014** - Amazon pozyskuje Twitch.

- **2015** - powstaje Discord - platforma społecznościowo-komunikacyjna stworzona z myślą o graczach. Współcześnie jest to jeden z najpopularniejszych komunikatorów na świecie.

- **2017** - TikTok - można zaryzykować, że jest to nowa generacja mediów społecznościowych. Pewien generacyjny przeskok, którego system działania i dopasowywanie treści jest dzisiaj powodem wielu publicznych dysput na temat etyki i szkodliwości Internetu.

Powyższe przykłady to tylko wybrane, najważniejsze momenty w historii mediów społecznościowych. Mimo, że serwisy te mają dość krótką historię, to jest ona niezmiernie intensywna. Rynek jakim są social media to ogromny tort, z którego wielu chciało, chce i będzie chcieć połasić się na jak największy jego kawałek.

2.1. Twitter

Twitter to portal społecznościowy o charakterze mikroblogowym. Powstał w 2006 roku. Oryginalnie użytkownicy mieli możliwość tworzenia krótkich postów tzw. Tweetów, lubienia ich lub dzielenia się nimi w gronie obserwujących. Wraz z rosnącą popularnością tego serwisu, wzrastała również ilość funkcjonalności. Dodano możliwość komentowania, dzielenia się zdjęciami czy zwiększono maksymalny limit ilości znaków w wypowiedzi.

2.2. Stack Overflow

Portal społecznościowy dla programistów powstały w 2008 roku. Jego kluczową cechą jest skupienie się na funkcjonalności zadawania pytań. Członkowie społeczności mają możliwość odpowiadać, popierać już istniejące odpowiedzi lub oceniać merytoryczną wartość pytania. Współcześnie strona, która często służy jako zamiennik dla dokumentacji różnych technologii. Nie tylko ułatwia ona radzenie sobie z napotkanymi trudnościami podczas implementacji, ale służy również jako pokaźny zbiór wiedzy.

2.3. Facebook

Obecnie synonim serwisu społecznościowego. Najpopularniejszy portal tego typu na świecie [8]. Oryginalnie produkt do dzielenia się informacjami wśród studentów różnych uczelni powstały w 2004 roku.

2.3.1. Początkowa architektura

Mark Zuckeberg wraz z kolegami, z którymi pracował nad witryną korzystali z zestawu oprogramowania LAMP [9]. Komputery działały pod kontrolą systemu operacyjnego Linux, serwerem HTTP było Apache, bazą danych MySQL, a podstawowym językiem programowania PHP. Każda uczelnia, na której był dostępny Facebook miała własną bazę danych. Ta decyzja pozwoliła na utrzymanie głównej funkcjonalności jaką stanowiło wyświetlanie wspólnych znajomych poprzez ograniczenie liczby użytkowników do przeliczenia. Konsekwencją tego była początkowa niemożliwość interakcji pomiędzy studentami z różnych szkół wyższych. Następnie rozdzielono warstwę danych od warstwy aplikacji. Dzięki temu duża ilość zapytań serwerowych na jednym uniwersytecie nie powodowała komplikacji dla użytkowników spoza tego miejsca. Ponadto pozwoliło to na wygodne skalowanie horyzontalne poprzez dodawanie kolejnych komputerów w celu zwiększenia mocy obliczeniowej. Wraz z rosnącą popularnością Facebooka, rosła ilość zapytań jakie aplikacja musiała wysyłać do warstwy danych. Ma ona jednak swoje limity, a ich przekroczenie może poskutkować całkowitą niemożnością korzystania z serwisu. Wtedy zdecydowano się na skorzystanie z Memcached i utworzenie dodatkowej warstwy dla pamięci cache. Ta decyzja przyśpieszyła działanie serwisu i w sposób znaczący odciążyła bazy danych.

Rozdział 3.

Wzorce architektury po stronie serwera

3.1. Grafowy język zapytań

GraphQL - język zapytań dla interfejsu programowego aplikacji wykorzystujący metodę POST [10]. Pozwala aplikacjom klienckim na dokładne wyspecyfikowanie żądanych danych. Ponadto daje możliwość pozyskania wielu informacji naraz bez potrzeby wysyłania wielu zapytań (rys. 3.1).

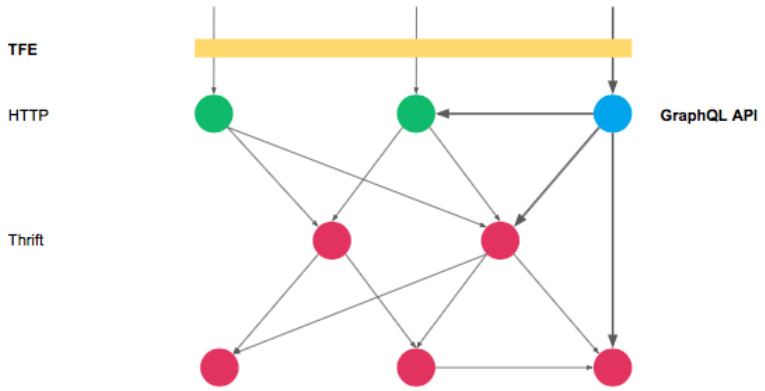
Lee Byron - jeden z oryginalnych twórców grafowego języka zapytań wspomina, że GraphQL był naturalnym następstwem architektonicznych decyzji podjętych przez firmę Marka Zuckeberga [12]. Pierwszy prototyp pojawił się w 2012 roku, a pierwsza przykładowa implementacja w 2015 roku, kiedy Facebook zaprezentował bibliotekę GraphQL.js [13]. Oryginalnie pomysł zrodził się w zespole infrastruktury, który tworzył narzędzia dla zespołów produkcyjnych. W tamtym okresie Facebook pracował nad nową aplikacją dla systemów mobilnych, a komunikacja z serwerem stwarzała coraz większe problemy. Zmiany po stronie klienta wymagały przerabiania odpowiedzi serwerowych, które same w sobie spowalniały działanie aplikacji ze względu na dużą objętość przesyłanych danych. Dodatkowo częste błędy związane z kształtem otrzymywanych danych spowodowane tworzeniem nowych iteracji API

```
{
  human(id: "1000") {
    name
    height(unit: FOOT)
  }
}
```

```
{
  "data": {
    "human": {
      "name": "Luke Skywalker",
      "height": 5.6430448
    }
  }
}
```

Rysunek 3.1: Przykład zapytania wraz z odpowiedzią (źródło rysunku: [11])

komplikowały pracę nad projektem. GraphQL był naturalną odpowiedzią na ówczesne problemy. Typowany system pozwala na samoczynną generację dokumentacji, kształt zapytań definiuje kształt odpowiedzi, a możliwość zagnieżdżania pól wyeliminowała potrzebę wysyłania niebotycznej ilości zapytań. Lee Byron wspomina, że przez cztery lata rozwijania aplikacji mobilnych korzystali wyłącznie z jednej wersji API i było to możliwe dzięki grafowemu językowi zapytań [12].



Rysunek 3.2: Wizualizacja architektury mikroserwisów Twittera (źródło rysunku: [14])

Kolejny ciekawy przykład wykorzystania tego sposobu komunikacji możemy znaleźć w innym, bardzo popularnym serwisie społecznościowym. Twitter jest znany z rozdzielania swoich funkcjonalności na pojedyncze mikroserwisy [14]. Oryginalnie korzystano z dwóch typów serwisów - HTTP oraz Thrift. Każdy z nich udostępniał typowany interfejs co umożliwiło wygodne zaimplementowanie grafowego języka zapytań jako protokołu komunikacyjnego od 2016 roku. Zdecydowano się na skorzystanie z biblioteki Sangria, ponieważ Twitter używa języka programowania Scala. Jednym z pierwszych wyzwań jakie napotkano było monitorowanie. Serwisy HTTP korzystają z szerokiej puli kodów błędów, które dają wystarczające informacje do zaimplementowania ich obsługi. GraphQL jednak pozwala na częściowy sukces, a co za tym idzie, większość zapytań kończy się statusem sukcesu nawet kiedy otrzymamy odpowiedź bez danych. Twitter zdecydował się śledzić ilość wystąpień wyjątków i w ten sposób określać powodzenie. Aby ominąć problem głębokich grafów stworzono system pomiarowy, który odrzuca zbyt zagnieżdżone zapytania. Co jednak z autoryzacją? W architekturze Twittera występuje warstwa The Twitter Frontend (TFE) (rys. 3.2), która rozdziela wewnętrzne serwisy z światem zewnętrznym. Dzięki temu nie występuje potrzeba implementowania specjalnych mechanizmów rozpoznawania uprawnień w samych GraphQL API.

3.2. Wykorzystanie pamięci podręcznej

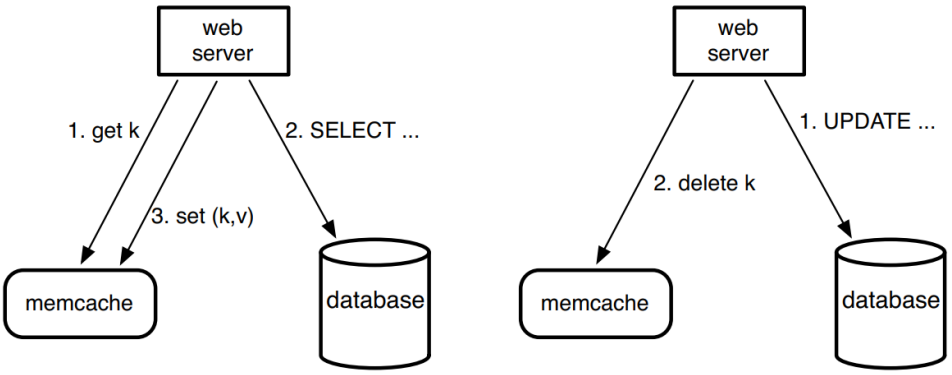
Cache - komponent fizyczny lub programowy, którego celem jest przetrzymywanie informacji na wypadek ponownej potrzeby jej wykorzystania. Znacząco przyśpiesza działanie współczesnych systemów. Znany jako pamięć podręczna. W serwisach webowych spełnia dwa podstawowe zadania: przyśpiesza działanie oraz odciąża bazę danych od obsługi zapytań. W przypadku dużych serwisów wykorzystanie pamięci cache jest praktycznie koniecznością.

3.2.1. Memcached

Przed popularnymi serwisami stoi wiele wyzwań, których trudność rozwiązania wzrasta wraz z ilością użytkowników danej witryny. Podczas konferencji NSDI 13' mogliśmy dowiedzieć się w jaki sposób Facebook radzi sobie z cache'owaniem informacji [15]. Stawiają określone cele, które przyświecają ich infrastrukturze:

- możliwość prawie natychmiastowej komunikacji,
- agregowanie na bieżąco danych z wielu źródeł,
- umiejętność wygodnego dzielenia się i aktualizowania popularnych w danym momencie treści,
- skalowanie, by umożliwić obsługę aktywności milionów użytkowników na sekundę.

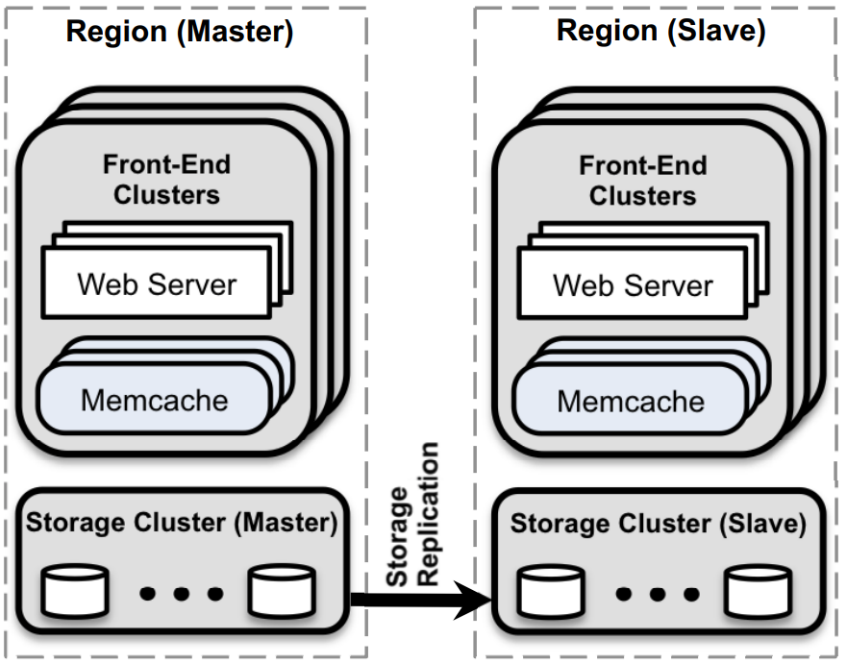
Wybory architektoniczne są bezpośrednio skorelowane z sposobem w jaki użytkownicy korzystają z serwisu. Należy pamiętać, że większość ludzi korzystających z Facebooka przyswaja o wiele więcej treści niż sama tworzy. System zdominowany przez taki typ aktywności o wiele częściej zbiera i poleca kontent niż wprowadza nowy - właśnie dzięki temu rozwiązanie jakim jest klasyczna tablica hashująca oparta na kluczach i wartościach - Memcached [16], ma tak duże znaczenie. Na poniższej ilustracji (rys. 3.3) został przedstawiony podstawowy przypadek wykorzystania Memcached przez Facebooka. Lewa strona demonstruje przykład działania gdy w cache'u nie ma żądanej informacji, natomiast prawa wizualizuje typowe dla zapisu treści zachowanie.



Rysunek 3.3: Zachowanie Memcached dla odczytu i zapisu (źródło rysunku: [15])

Kiedy serwer potrzebuje danych, najpierw odwołuje się do pamięci. Jeśli tam ich nie znajdzie kieruje swoje zapytanie do bazy danych lub konkretnego serwisu serwerowego, a następnie aktualizuje cache o nowe informacje. W przypadku zapisu wszelkie nieprawdziwe informacje w pamięci zostają usunięte. Jest to najprostsze i najwygodniejsze rozwiązanie, które gwarantuje idempotentność w przeciwieństwie do odświeżania.

Każdy konkretny serwer posiada własną instancję Memcached. Kolejnym wyzwaniem jest skorelowanie jego działania w jednej z największych infrastruktur na świecie.



Rysunek 3.4: Replikacja danych pomiędzy clusterami (źródło rysunku: [15])

Facebook osiąga ten cel za pomocą ciągłej replikacji informacji pomiędzy bazodanowymi clusterami (rys. 3.4). Dzięki temu, mimo że pamięć cache jest niezależna w zasięgu swojego serwera to odnosi się zawsze do, jak najbardziej to możliwe, aktualnej wersji trwale zapisanych danych.

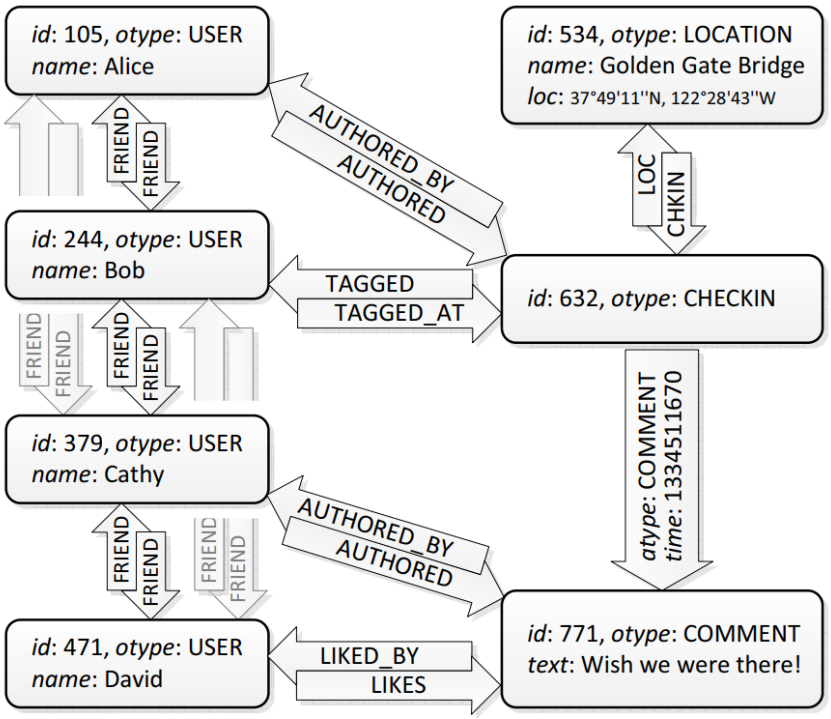
Należy pamiętać, że implementacja działającej architektury z wykorzystaniem współdzielonej pamięci operacyjnej nie rozwiązuje wszystkich problemów. Facebook w ciekawy sposób redukuje opóźnienia i ilość zapytań jakie aplikacja kliencka wysyła. Jednym z przykładów zaprezentowanych podczas konferencji była liczba 521, która reprezentuje ilość oddzielnie identyfikowanych danych jakie należy pobrać, by wyświetlić najpopularniejsze widoki. Portal ten aplikuje równoległe wysyłanie żądań i ich grupowanie, by zoptymalizować ten proces. Minimalizuje to ilość potrzebnych zapytań poprzez tworzenie skierowanych acyklicznych grafów zależności danych, by zmaksymalizować ilość otrzymanych informacji podczas jednego żądania.

3.2.2. TAO

TAO to rozproszony kontener do przechowywania grafów społecznościowych [17]. Jest on jedną z alternatyw dla Memcached wykorzystywanych w Facebooku [18]. Grafy te łączą konkretne obiekty z ich powiązaniami między sobą, a jednocześnie pozwalają na aplikowanie konkretnych ustawień prywatności czy autoryzacji do wszystkich wierzchołków w zależności od użytkownika. TAO samo w sobie faworyzuje wydajność nad aktualność informacji. Przeanalizujmy następujący przykład (rys. 3.5).

Rysunek 3.5: Przykładowa treść społecznościowa (źródło rysunku: [18])

Alice tworzy treść społecznościową podczas zwiedzania mostu Golden Gate. Jednocześnie oznacza, że towarzyszy jej Bob, natomiast Cathy komentuje ten konkrety post, a David go lubi. Powiązania pomiędzy tymi konkretnymi informacjami mogą być zaimplementowane w sposób zaprezentowany na następnej ilustracji (rys. 3.6).



Rysunek 3.6: Przykładowy schemat powiązań interakcji (źródło rysunku: [18])

TAO utrzymuje informacje o tych powiązaniach i za każdym razem, kiedy dana informacja ma zostać wygenerowana w aplikacji klienckiej, udostępnia jej schemat dopasowany do użytkownika.

3.2.3. Redis

Redis to oprogramowanie open-source służące do przechowywania informacji w pamięci, wykorzystywane jako baza danych lub cache [19]. Domyślnie obsługuje wiele struktur danych, jak na przykład: słowniki, listy czy zbiory. Ponadto ma wbudowany mechanizm replikacji i stanowi popularną alternatywę dla Memcached. Narzędzie to jest wykorzystywane w architekturze Twittera [20] oraz Stack Overflow [21].

W przypadku Twittera, ich sposób użycia Redisa jest wysoce skalowalny - ponad 10 tysięcy instancji, które korzystają sumarycznie z 105 terabajtów pamięci RAM. Podkreśla się, że jest to niesamowite narzędzie, ponieważ pozwala dopełnić wykorzystanie wszelkich zasobów do granic ich możliwości [20]. Twitter wykorzystuje Redisa do utrzymywania informacji o zbiorach postów. Zbiory te, nazywane Timelines, to łańcuchy kontentu wyświetlanego w żądanym widoku, na przykład: strona główna zawierająca wszystko co chcemy pokazać użytkownikowi lub strona profilowa, która prezentuje wyłącznie treści utworzone przez daną osobę.

Twitter zaimplementował własne modyfikacje do Redisa, by zoptymalizować ten proces. Wprowadzili dwie nowe struktury danych, które w znaczący sposób

przyśpieszają odwołania do tego rodzaju kontenera: listę hybrydową oraz B-drzewa. Zacznijmy od pierwszej. Redis oferuje dwa typy list w swoim standardzie. Ziplist jest bardzo wydajna, niestety korzysta z umiejscawiania kolejnych wpisów fizycznie obok siebie. Usuwanie konkretnych danych powoduje przesuwanie zaalokowanej pamięci, a sama w sobie ma też limit maksymalnego rozmiaru. Linklist jest o wiele bardziej elastyczna pod tym względem, ale zużywa więcej pamięci, ponieważ musi utrzymywać informację o wskaźnikach na każdy kolejny i poprzedni element. Lista hybrydowa to właściwie połączenie tych dwóch koncepcji - jest to linklista, gdzie kolejnymi elementami są ziplisty. W ten sposób można utrzymywać niewiarygodnie długie łańcuchy treści. B-drzewa natomiast to struktura oferująca możliwość wyszukiwania, dodawania i usuwania w czasie logarytmicznym. Jest ona podobna do drzew binarnych, ale zezwala na węzły z więcej niż dwoma dziećmi. Wykorzystywana w relacyjnych bazach danych. W przypadku Twittera skorzystano z implementacji BSD, a służy im ona do obsługi ustrukturyzowanych zapytań.

3.3. Wzorzec Repository

Podstawowym zadaniem wzorca Repository jest stworzenie dodatkowej warstwy abstrakcji rozdzielającej biznesowy kod od warstwy komunikacji z źródłem informacji [22]. Nie chcemy przejmować się tym skąd i jak pozyskiwane są rekordy, a jedynie je otrzymywać. Ma to ogromny wpływ na skalowalność i wygodę w rozwijaniu produktu. Stworzenie różnych klas lub komponentów dla konkretnych rodzajów danych niesie za sobą następujące korzyści:

- naturalne rozwiązanie problemu redundancji kodu (korzystamy z raz napisanych zapytań),
- łatwa możliwość zmiany źródła informacji ze względu na enkapsulację logiki,
- wygodna edycja działania konkretnych funkcjonalności związanych z warstwą danych,
- prostsza implementacja testów jednostkowych.

W praktyce często łączony z wzorcem Unit of Work, który służy do grupowania zapytań tak, aby wszystkie miały miejsce w trakcie pojedynczej transakcji [23].

3.4. Command and Query Responsibility Segregation

Command and Query Responsibility Segregation (CQRS) - wzorzec rozdzielający logikę zapisu danych od ich odczytu [24]. Zaproponowany przez Bertranda Meyera, który zauważył, że każda metoda powinna być komendą lub zapytaniem. W

tym przypadku komenda powinna wprowadzać zmiany w bazie danych, a zapytanie wyłącznie dostarczać żądany zbiór rekordów bez powodowania efektów ubocznych.

Założenia te niosą za sobą szereg powszechnie cenionych konsekwencji. Rozdzielenie logiki wprowadza możliwość pełnej personalizacji warstwy danych w ramach potrzeb danego produktu. Nic nie stoi na przeszkodzie, by korzystać z różnych podmiotów w zależności od tego, czy ma on służyć do zapisywania danych czy ich odczytywania. To zaś pozwala na osobne skalowanie, co ma ogromne korzyści w systemach takich jak serwisy społecznościowe, które w większości przypadków służą do wyświetlania treści. Dzięki temu możemy tworzyć osobne, zoptymalizowane pod kątem obsługiwanych operacji, modele. Jednym z przykładów takiej optymalizacji są tabele w postaci zdenormalizowanej, których wykorzystywanie pozwala znacząco uprościć zapytania w celu szybszego ich wykonywania.

Podstawowym wyzwaniem podczas implementacji tego wzorca jest ciągła aktualizacja danych pomiędzy źródłem zapisu, a źródłem odczytu. W zależności od poziomu separacji jaki zaimplementujemy, kompleksowość tego zadania może się znacznie różnić.

CQRS jest powszechnie stosowany w połączenie z wzorcem Event Sourcing [25]. Zakłada on zapisywanie wydarzeń i traktowanie ich jako źródła danych. Następnie wydarzenia te są materializowane w odpowiedniej postaci, by przygotować jak najbardziej aktualny stan informacji, z którego będzie korzystać warstwa zapytań. Ciągłe zapisywanie historii wydarzeń pozwala na odwzorowanie konkretnego stanu w każdym momencie co może okazać się pomocne podczas wprowadzania zmian do modelu odczytu danych.

Rozdział 4.

Wzorce architektury po stronie klienta

4.1. Rekomendacja treści

Pierwsze serwisy społecznościowe serwowały treści w porządku chronologicznym. Współcześnie największe platformy ingerują nie tylko w kolejność wyświetlania, ale także personalizują proponowane treści tak, aby użytkownik pozyskał jak najwięcej wartościowej dla niego wiedzy w jak najkrótszym czasie. Serwis Reddit, by określić skalę ważności konkretnej treści bierze pod uwagę trzy kluczowe dla niej aspekty: długość życia, liczbę polubień oraz stosunek pozytywnych głosów do tych negatywnych [26]. Co ciekawe same polubienia tracą na wadze wraz z upływem czasu. W ten sposób omija się problem stagnacji treści, a jednocześnie użytkownicy otrzymują najistotniejsze informacje w skondensowanej formie. Co jednak w przypadku portali, które nie oferują funkcjonalności negatywnego głosowania? Są one przecież podatne na polecanie treści niezgodnych z normami etycznymi. Facebook i Instagram upubliczniają dokumentację, która wyznacza konkretne kwalifikatory określające, czy dany post może zostać przez nich przekazany do innych użytkowników [27]. Wyklucza się następujące przypadki:

- treści wrażliwe,
- treści związane z publikacjami o niskiej jakości,
- treści, które powszechnie są nielubiane,
- treści przeszkadzające w budowaniu bezpiecznej społeczności.

Rekomendacja treści może również bazować na identyfikacji upodobań użytkowników. Instagram w tym celu korzysta z uczenia maszynowego [28]. Na potrzeby wygodnej implementacji nowych rozwiązań stworzono język IGQL, który łączy najlepsze cechy Pythona oraz C++. Dzięki niemu możliwe jest szybkie wytwarzanie i

testowanie nowych rozwiązań bez potrzeby skupiania się na detalach związanych z zużyciem zasobów czy wydajnością. Co ciekawe, Instagram zapisuje kolejno odwiedzane przez użytkowników konta w przedziale konkretnych sesji. Zauważono bowiem, że powtarzalne sekwencje interakcji występują zazwyczaj w treściach podobnych tematycznie [28]. To z kolei poszerza zbiór potencjalnych rekomendacji, które można serwować konsumentom, u których zidentyfikowano dane upodobania.

4.2. Nieskończone przewijanie

Współczesne serwisy społecznościowe jako jedną z podstawowych funkcjonalności oferują spersonalizowany widok dostosowany do potrzeb i wymagań konkretnego użytkownika. Platformy takie jak Facebook, Instagram, Twitter, TikTok lub YouTube serwują te treści w nieskończoność. Wraz z przewijaniem strony aktywnie doładowują się nowe informacje. Pozwala to na zmniejszenie ilości wymaganych danych na start co z kolei działa korzystnie na szybkość załadowania widoku. Jest to swego rodzaju iteracja idei paginacji, która swoje początki miała na serwisach numerujących kolejne strony. Zmniejszenie ilości wymaganych działań użytkownika zwiększa uczucie imersji oraz bardziej zachęca do sprawdzenia kolejnych treści.

4.3. Kontrola sesji

Dołączenie do społeczności danego serwisu wymaga utworzenia konta użytkownika, a wchodzenie w interakcję z treściami tam zawartymi jest dostępne wyłącznie dla zalogowanych konsumentów. Współczesnym standardem jest niewidoczne dla oka użytkownika utrzymywanie jego sesji. Ponadto raz zalogowany użytkownik prawdopodobnie chce takim pozostać, więc serwisy bazujące na takich rozwiązaniach muszą pamiętać status identyfikacji danej osoby, nawet gdy opuści stronę na swojej przeglądarce. Dodatkowo coraz więcej portali oferuje możliwość zdalnego wylogowania się co zwiększa komfort użytkownika oraz poprawia bezpieczeństwo jego wrażliwych informacji, które mógłby w niechciany sposób udostępnić poprzez zapomnienie o ręcznym zakończeniu sesji na innym komputerze. Istnieją różne sposoby jej implementacji, wyróżniamy między innymi: identyfikatory sesji, tokeny, czy protokół OAuth [29].

Rozdział 5.

Implementacja wybranych wzorców

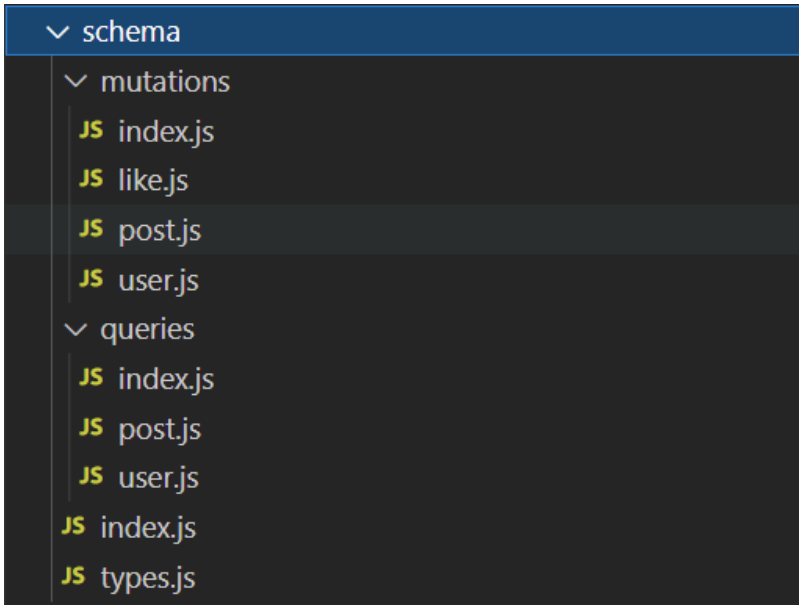
5.1. Grafowy język zapytań

Do stworzenia w pełni funkcjonalnej komunikacji bazującej na grafowym języku zapytań wykorzystałem następujące technologie:

- **Express** - minimalistyczna biblioteka serwerowa stworzona dla Node.js [30].
- **GraphQL.js** - biblioteka referencyjna grafowego języka zapytań wydana przez Facebook [13].
- **express-graphql** - GraphQL middleware dla serwera HTTP. Obsługuje Express, Connect i Restify [31].
- **Apollo Client** - biblioteka obsługująca zarządzanie wysyłaniem i odbieraniem zapytań po stronie klienta [32].
- **Insomnia** - platforma do projektowania i testowania API [33].

GraphQL oferuje dwa typy żądań: zapytania oraz mutacje. Proponuje się różne konwencje ich wykorzystania, ale najważniejszą różnicą pomiędzy nimi jest według mnie sposób ich działania. Zapytania bowiem wykonują się asynchronicznie, natomiast mutacje synchronicznie. Stąd decyzja o twardym podziale - te pierwsze obsługują wszelkiego rodzaju czytanie danych, natomiast te drugie odpowiadają za żądania je modyfikujące. Kolejną decyzją, którą musiałem podjąć był sposób wytwarzania API. Istnieją bowiem dwa podejścia: schema-first oraz code-first. Ten pierwszy zakłada pisanie twardo ustalonych schematów, do których później musimy dopasowywać nasz kod, natomiast drugi w sposób programowy generuje schematy za pomocą napisanego kodu. Wybrałem podejście code-first, aby osiągnąć efekt pojedynczego źródła prawdy. Dzięki takiemu podejściu sama definicja danego żądania

generuje się automatycznie na podstawie kodu, który je obsługuje. Następnym wyzwaniem jakie napotkałem było stworzenie wygodnej dla rozwoju i czytelnej struktury plików po stronie kodu serwerowego. GraphQL zazwyczaj jest prezentowany na bardzo prostych przykładach, których schemat spokojnie może mieścić się w jednym pliku, natomiast w moim przypadku chciałem odpowiednio je podzielić, by w wygodny sposób rozwijać API (rys. 5.1).



Rysunek 5.1: Struktura plików schematu

Powyższy cel został osiągnięty dzięki rozbiciu schematu na konkretne obiekty (rys. 5.2, 5.3, 5.3), których składową jest właśnie ów schemat.

```
export const schema = new GraphQLSchema({
    query: Query,
    mutation: Mutation,
});
```

Rysunek 5.2: Podział na zapytania i mutacje obecny w pliku *index.js* (rys. 5.1)

```
export const Mutation = new GraphQLObjectType({
    name: 'Mutation',
    fields: {
        createUser: createUser,
        loginUser: loginUser,
        logoutUser: logoutUser,
        fullLogoutUser: logoutUserFromAllSessions,
        createPost: createPost,
        likePost: likePost,
        unlikePost: unlikePost,
        createPostOne: createPost1,
        likePostOne: likePost1,
        unlikePostOne: unlikePost1,
        createPostTwo: createPost2,
        likePostTwo: likePost2,
        unlikePostTwo: unlikePost2,
        createPostThree: createPost3,
        likePostThree: likePost3,
        unlikePostThree: unlikePost3,
    },
});
```

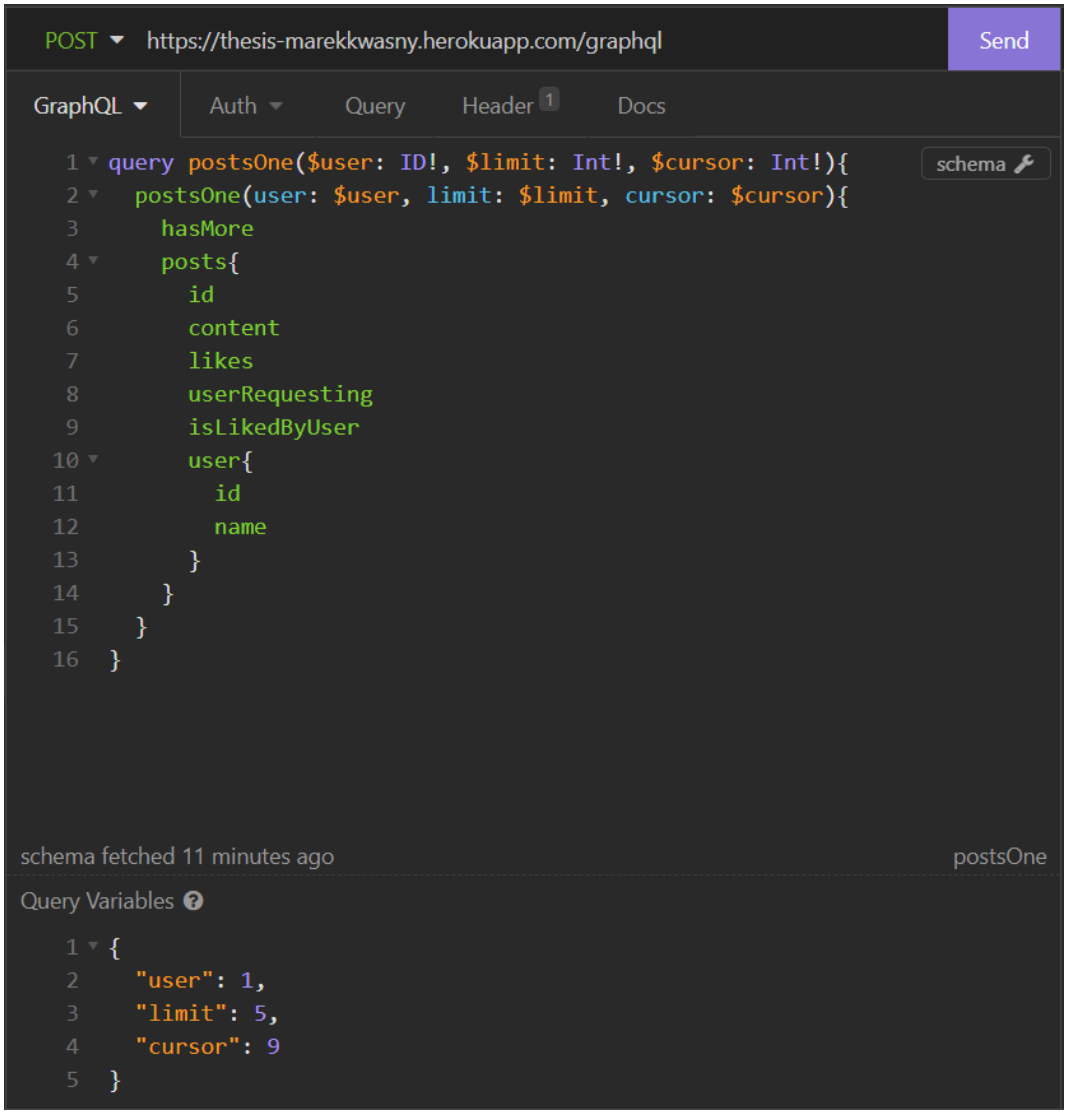
Rysunek 5.3: Definiowanie mutacji z wielu źródeł w *mutations/index.js* (rys. 5.1)

```
export const createPost3 = {
    type: GraphQLBoolean,
    args: {
        user: { type: GraphQLID },
        content: { type: GraphQLString },
    },
    resolve: async (_, args, ctx) => {
        if (!args.user || !args.content) {
            return null;
        }

        const cs = new CommandService3();
        return await cs.createPost(ctx.pool, args.user, args.content);
    },
};
```

Rysunek 5.4: Przykładowa implementacja mutacji odpowiedzialnej za utworzenie treści w *mutations/post.js* (rys. 5.1)

W interaktywnym rozwoju API znacząco pomogła platforma Insomnia. Udostępnia ona podobne możliwości co GraphiQL [34] czy GraphQL Playground [35], ale uzupełnia je również o cały pakiet dodatkowych funkcjonalności, które pozwoliły mi w wygodny sposób zaprojektować między innymi działanie autoryzacji w środowisku grafowego języka zapytań. Poniżej przykład wywołania zapytania zwracającego listę postów wraz ze wszystkimi informacjami, które potrzebowałem do poprawnej implementacji funkcjonalności wyświetlania treści w aplikacji klienckiej (rys. 5.5, 5.6).



Rysunek 5.5: Zapytanie *PostsOne* wraz z argumentami

Rysunek 5.6: Odpowiedź serwera z listą treści

5.2. Wzorzec Repository

Do przechowywania informacji wykorzystuję relacyjną bazę danych PostgreSQL [6], a w celu komunikacji z nią stosuję bibliotekę node-postgres [36]. Nie jest to narzędzie służące do mapowania obiektowo-relacyjnego, więc samemu odpowiadam za model oraz napisanie własnych zapytań przy użyciu języka SQL. To pozwala mi przejąć pełną kontrolę nad logiką bazodanową kosztem dłuższego czasu implementacji. Aby przygotować mój projekt do rozwoju i uwspólnić jak największą część

kodu w myśl powszechnie stosowanej zasady DRY (Don't Repeat Yourself) [37] zdecydowałem się skorzystać ze wzorca Repository. Każda z tabel posiada własną klasę służącą do egzekwowania odpowiednich poleceń i zwracania żądanych rezultatów (rys. 5.7). Dzięki takiemu podejściu znacząco zredukowałem powtarzalne fragmenty kodu w kontrolerach, czy też grafowych resolverach.

```
class TokenRepository {
    #pool;

    constructor(pool) {
        this.#pool = pool;
    }

    async find(id) {
        try {
            const query = {
                name: 'fetch-token-version',
                text: 'SELECT token_version FROM refresh_tokens WHERE fk_users = $1 LIMIT 1',
                values: [id],
            };

            logToConsole(query);
            const ret = await this.#pool.query(query);

            return { tokenVersion: ret.rows[0].token_version };
        } catch (err) {
            console.log(err);
            return null;
        }
    }

    async increment(id) {
        try {
            const query = {
                name: `increment-token-version-for-${id}`,
                text:
                    'UPDATE refresh_tokens SET token_version = token_version + 1 ' +
                    'WHERE fk_users = $1',
                values: [id],
            };

            logToConsole(query);

            await this.#pool.query(query);
        } catch (err) {
            console.log(err);
        }
    }
}
```

Rysunek 5.7: Repozytorium dla tabeli utrzymującej informacje o wersjach tokenów

Dodatkowo zdecydowałem się nałożyć delikatną warstwę abstrakcji ponad same repozytoria. Podjąłem decyzję, by nie eksportować klas, a jedynie funkcje, które z nich korzystają (rys. 5.8). Dzięki temu kod po stronie kontrolerów jest czytelniejszy, a jednocześnie eliminuję ryzyko złego wykorzystania repozytoriów.

```
export async function getTokenVersion(pool, userId) {
    const tokens = new TokenRepository(pool);
    return await tokens.find(userId);
}

export async function revokeToken(pool, userId) {
    const tokens = new TokenRepository(pool);
    await tokens.increment(userId);
}
```

Rysunek 5.8: Funkcje wykorzystujące *TokenRepository* (rys. 5.7)

5.3. Command and Query Responsibility Segregation

W przypadku wzorca architektury CQRS postanowiłem zaprezentować kolejne fragmenty rozwoju tej idei w celu prezentacji możliwości rozwoju i samej skalowalności. Efekty tej pracy są widoczne w aplikacji klienckiej pod odpowiednimi widokami, do których można się dostać za pomocą przycisków z przedrostkiem CQRS. Podstawowym krokiem w przerobieniu już istniejących rozwiązań było rozdzielenie ich na komendy i kwerendy. Zaprojektowałem, więc interfejsy *ICommandService* oraz *IQueryService*, po których dziedziczą wszystkie wersje serwisów stworzonych z myślą o tym wzorcu.

5.3.1. Pierwszy krok implementacji

Pierwszy krok polegał wyłącznie na utworzeniu odpowiednich abstrakcji enkapsulujących logikę bazodanową w rozdzieleniu na operacje odczytu i zapisu (rys. 5.9, 5.10). Funkcje wywoływane w metodach tychże serwisów korzystają z już wcześniej zaimplementowanych rozwiązań przy użyciu wzorca Repository.

```
export class CommandService1 extends ICommandService {
    constructor() {
        super();
    }

    async createPost(pool, userId, content) {
        return await db.createNewPost(pool, userId, content);
    }

    async likePost(pool, userId, postId) {
        return await db.likePost(pool, userId, postId);
    }

    async unlikePost(pool, userId, postId) {
        return await db.unlikePost(pool, userId, postId);
    }
}
```

Rysunek 5.9: Implementacja *CommandService* dla pierwszego kroku

```
export class QueryService1 extends IQueryService {
    constructor() {
        super();
    }

    async fetchPosts(pool, userId, limit, cursor) {
        return await db.fetchPosts(pool, userId, limit, cursor);
    }
}
```

Rysunek 5.10: Implementacja *QueryService* dla pierwszego kroku

5.3.2. Drugi krok implementacji

W kroku drugim zdecydowałem się skorzystać z wzorca Event Sourcing. Utworzyłem tabelę *posts_source*, która służy za jedyne źródło zapisywania informacji w postaci wydarzeń (rys. 5.11). Wyróżniamy trzy rodzaje zdarzeń, które mogą wystąpić: utworzenie treści, polubienie lub wycofanie polubienia.

	id	event_log	created_at
1	1	{"create":{"author":"1","author_name":"Marek Kwaśny","content":"Hello CQRS2!"}}	2021-09-16T14:26:41.181569
2	2	{"like":{"user":1,"post":1}}	2021-09-16T17:03:36.482930
3	3	{"create":{"author":"1","author_name":"Marek Kwaśny","content":"CQRS2 is great…	2021-09-16T17:06:14.050481
4	4	{"create":{"author":"6","author_name":"kin","content":"Lorem ipsum dolor sit a…	2021-09-18T01:44:22.259810
5	5	{"create":{"author":"6","author_name":"kin","content":"Formula One (also known…	2021-09-18T02:05:28.425430
6	6	{"like":{"user":6,"post":5}}	2021-09-18T02:05:50.235652
7	7	{"like":{"user":1,"post":5}}	2021-09-18T02:06:03.186771
8	8	{"unlike":{"user":1,"post":5}}	2021-09-20T13:39:15.709933

Rysunek 5.11: Przykładowe wydarzenia zapisywane w *posts_source*

Aby odczytać obecny stan postów *QueryService* pozyskuje informacje o wszystkich zdarzeniach z bazy i symuluje ich przebieg otrzymując aktualne dane, które można zaserwować użytkownikowi jako poprawny zbiór treści w odpowiednim widoku (rys. 5.12). Funkcja *restore()* odpowiada w tym przypadku za interpretację każdego z napotkanym zdarzeń.

```
export class QueryService2 extends IQueryService {
    constructor() {
        super();
    }

    async fetchPosts(pool, userId, limit, cursor) {
        const data = await db.fetchPostLogs(pool);
        const posts = new Map();
        data.map((event) => {
            restore(event, posts, userId);
        });

        const res = [];
        posts.forEach((value) => {
            if (cursor === 0) {
                res.push(value);
            } else if (value.id < cursor) {
                res.push(value);
            }
        });

        return { hasMore: res.length - limit > 0, posts: res.reverse().slice(0, limit) };
    }
}
```

Rysunek 5.12: Implementacja *QueryService* w drugim kroku

5.3.3. Trzeci krok implementacji

Trzeci krok nadal korzysta z własnego źródła wydarzeń, które zawsze w wygodny sposób może służyć jako narzędzie odzyskiwania danych. Natomiast cechą dodaną w tej fazie rozwoju jest tablica *posts_and_likes* w postaci zdenormalizowanej (rys. 5.13). Utrzymuje ona informacje o bieżącym stanie danych.

	id	created_at	is_create	author_id	author_name	content	likes	is_like	like_user_id	like_post_id
1	7	2021-09…	False	*NULL*	*NULL*	*NULL*	*NULL*	True	6	6
2	1	2021-09…	True	1	Marek Kw…	Hello…	0	False	*NULL*	*NULL*
3	4	2021-09…	False	*NULL*	*NULL*	*NULL*	*NULL*	True	1	3
4	6	2021-09…	True	6	kin	Lorem…	1	False	*NULL*	*NULL*
5	5	2021-09…	False	*NULL*	*NULL*	*NULL*	*NULL*	True	4	3
6	3	2021-09…	True	1	Marek Kw…	But C…	2	False	*NULL*	*NULL*
7	8	2021-09…	True	1	Marek Kw…	😳	0	False	*NULL*	*NULL*

Rysunek 5.13: Przykładowy stan tabeli *posts_and_likes*

Dzięki zastosowaniu projekcji wydarzeń zapamiętywanej przez tablicę *posts_and_likes* byłem w stanie znacząco uprościć, a tym samym przyśpieszyć zapytanie służące do pobierania wszystkich potrzebnych informacji związanych z wyświetlanymi treściami w aplikacji klienckiej. Poniżej porównanie obu zapytań (rys. 5.14, 5.15).

```
const fetchQuery = {
    name: `fetch-posts-for-${userId}`,
    text:
        'SELECT posts.id, posts.content, posts.likes, posts.fk_users, users.name, ' +
        'users.email, fk_users_likes = $1 AS is_liked_by_user ' +
        'FROM posts ' +
        'LEFT JOIN likes ON (fk_posts_likes = posts.id AND fk_users_likes = $1) ' +
        'JOIN users ON posts.fk_users = users.id ' +
        'WHERE posts.id < $2 ' +
        'ORDER BY posts.id DESC LIMIT $3',
    values: [userId, cursor, limit],
};
```

Rysunek 5.14: Zapytanie pozyskujące treści dla widoku *Home*

```
const query = {
    name: `fetch-denormalized-posts-for-${userId}`,
    text:
        'SELECT * FROM posts_and_likes ' +
        'WHERE is_create = TRUE OR (is_like = TRUE AND like_user_id = $1) ' +
        `ORDER BY id`,
    values: [userId],
};
```

Rysunek 5.15: Zapytanie pozyskujące treści dla widoku *CQRS 3*

5.4. Nieskończone przewijanie

Implementacja nieskończonego przewijania była wyśmienitą okazją do zaprojektowania mechanizmu paginacji w celu pobierania wyłącznie treści, które w danym momencie chcemy doładować. Zdecydowałem się na oparcie tej funkcjonalności o kursor. Takie podejście gwarantuje brak doładowywania duplikatów w przypadku powstania nowych treści w trakcie procesu przewijania widoku przez użytkownika. Ponadto wśród zwróconych przez serwer danych widnieje odpowiednie pole *hasMore* utrzymujące wartość logiczną informującą klienta o istnieniu kolejnych treści, które można doładować. Mechanizm inicjujący doładowywanie wykorzystuje bibliotekę React Waypoint [38]. Służy ona do rejestracji wydarzenia przewinięcia kontenera do odpowiedniego komponentu - wlicza się w to również okno przeglądarki. Dzięki temu byłem w stanie uzależnić wywołanie odpowiedniej logiki doładowującej i łączącej już załadowane treści w oparciu o pozycję paska przewijania po stronie klienta. Poniżej przykład komponentu implementującego funkcjonalność nieskończonego przewijania (rys. 5.16).

```
export const PostsData = ({ user, loading, data, fetchMore }) => {
    if (loading) return <div>...</div>;

    return (
        <div>
            <center>
                {data.posts.posts.map((x, i) => (
                    <React.Fragment key={`${i}-${user}-${x.id}`}>
                        <List>
                            <Post item={x} user={user} />
                        </List>
                        {data.posts.hasMore && i === data.posts.posts.length - 5 && (
                            <Waypoint
                                onEnter={() => {
                                    fetchMore({
                                        variables: {
                                            user: user,
                                            limit: 5,
                                            cursor: parseInt(
                                                data.posts.posts[
                                                    data.posts.posts.length - 1
                                                ].id
                                            ),
                                        },
                                        updateQuery: (pv, { fetchMoreResult }) => {
                                            if (!fetchMoreResult) {
                                                return pv;
                                            }

                                            return {
                                                posts: {
                                                    __typename: 'PostsType',
                                                    hasMore:
                                                        fetchMoreResult.posts.hasMore,
                                                    posts: [
                                                        ...pv.posts.posts,
                                                        ...fetchMoreResult.posts.posts,
                                                    ],
                                                },
                                            };
                                        },
                                    });
                                }}
                            />
                        )}
                    </React.Fragment>
                ))}
            </center>
        </div>
    );
};
```

Rysunek 5.16: Komponent *PostsData* serwujący kolejne treści

5.5. Kontrola sesji

Autoryzacja to proces identyfikacji połączony z sprawdzeniem praw dostępu do danej treści. W przypadku mojego projektu zdecydowałem się na oparcie tego procesu o tokeny JWT [39]. W momencie logowania serwer wystawia klientowi dwa tokeny: *AccessToken* oraz *RefreshToken*. Ten pierwszy ma krótki czas życia (pięć minut) i jest utrzymywany w pamięci przeglądarki, ten drugi przeciwnie - jego ważność to aż rok, a utrzymuję informacje o nim w odpowiednim ciasteczku. Każde wysyłane z aplikacji klienckiej zapytanie posiada nagłówek z tokenem dostępu. Jeśli jego walidacja się powiedzie to znaczy, że użytkownik jest zalogowany. Ponadto posiada on informację o identyfikatorze użytkownika, by ograniczyć zbędne operacje po stronie kodu serwerowego (rys. 5.17). Mechanizm dodawania nagłówka został zaimplementowany przy pomocy biblioteki Apollo Link [40], która służy do personalizacji przepływu danych pomiędzy klientem, a serwerem.

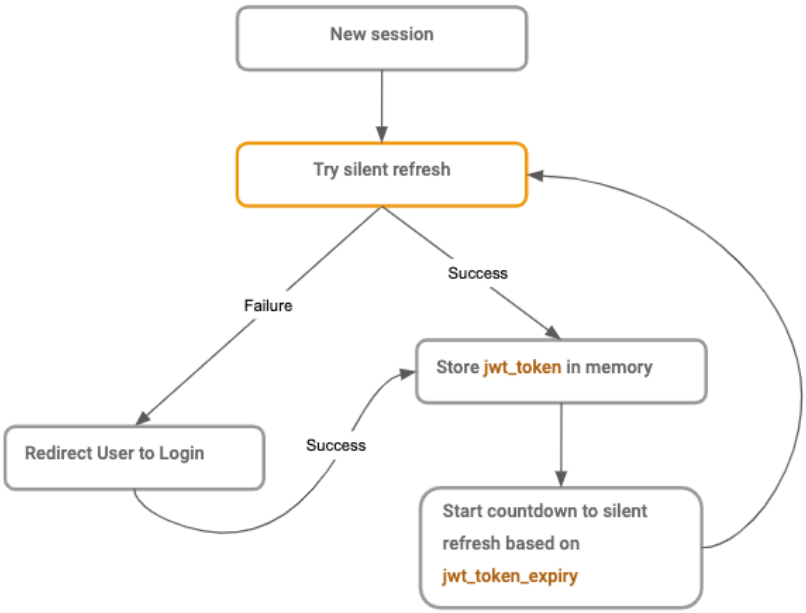
```
{
  "id": 1,
  "iat": 1632155775,
  "exp": 1663713375
}
```

Rysunek 5.17: Rozszyfrowana postać tokenu dostępu

```
export const tokenRefreshLink = new TokenRefreshLink({
    accessTokenField: "accessToken",
    isTokenValidOrUndefined: () => isAccessTokenValid(),
    fetchAccessToken: () => {
        return fetch("/auth", {
            method: "POST",
            credentials: "include",
        });
    },
    handleFetch: (accessToken) => {
        setAccessToken(accessToken);
    },
});
```

Rysunek 5.18: Implementacja metody odpowiedzialnej za odświeżanie *AccessTokenu* po stronie klienta

Aplikacja kliencka przed wykonaniem każdego zapytania sprawdza ważność tokenu (rys. 5.18). W przypadku przedawnienia kieruje zapytanie typu POST pod punkt końcowy */auth*. W tym momencie serwer bazując na statusie walidacji tokenu odświeżania wystawia nową instancję *AccessToken* oraz *RefreshToken* lub informuje klienta o braku dostępu. Funkcjonalność ta została zaimplementowana za pomocą biblioteki Token Refresh Link [41], która korzystając z Apollo Link [40] pozwala na czytelniejsze zapisanie kodu. Mechanizm cichego odświeżania tokenu można zaprezentować za pomocą poniższego diagramu (rys. 5.19).



Rysunek 5.19: Wizualizacja mechanizmu cichego odświeżania tokenów (źródło rysunku: [42])

```
export async function grantToken(pool, req, res) {
    const token = req.cookies.REFRESH_TOKEN;

    if (!token) {
        return res.send(getAccessToken(''));
    }

    let payload = null;
    try {
        payload = verify(token, process.env.REFRESH_TOKEN_SECRET);
    } catch (err) {
        console.log(err);
        return res.send(getAccessToken(''));
    }

    const { tokenVersion } = await db.getTokenVersion(pool, payload.id);

    if (tokenVersion !== payload.version) {
        return res.send(getAccessToken(''));
    }

    sendRefreshToken(res, createRefreshToken(payload.id, tokenVersion));

    return res.send(getAccessToken(createAccessToken(payload.id)));
}
```

Rysunek 5.20: Funkcja odpowiedzialna za wydanie nowego tokenu

Dodatkowym atutem takiego podejścia jest prosta implementacja funkcjonalności wylogowania użytkownika ze wszystkich jego sesji . *RefreshToken* posiada spe-

cjalne pole informujące o swojej wersji. W momencie wysłania przez użytkownika żądania zakończenia wszystkich instancji zalogowania serwer inkrementuje jej numer co powoduje, że następna próba cichego odświeżenia tokenu dostępu zakończy się porażką ze względu na negatywny status walidacji *RefreshTokenu* (rys. 5.20).

Taka implementacja kontroli sesji jest wysoce skalowalna ze względu na brak potrzeby każdorazowego odwoływania się do bazy danych, by przeprowadzić proces autoryzacji.

Rozdział 6.

Informacje dla programisty

Dostęp do kodu źródłowego jest możliwy przez publiczne repozytorium pod adresem: `https://github.com/marekkwasny/thesis`. Wymagane narzędzia:

- system kontroli wersji Git [5],
- środowisko uruchomieniowe Node.js [2],
- serwer bazodanowy PostgreSQL [6].

Aby zainstalować projekt lokalnie i móc testować go na własnym podmiocie należy wykonać poniższe polecenia:

1. Sklonować repozytorium komendą:

   ```
   git clone https://github.com/marekkwasny/thesis.git
   ```

2. Przejść do katalogu i uruchomić instalację poleceniem:

   ```
   cd thesis && npm install
   ```

3. Odwzorować bazę danych za pomocą kwerend z pliku *queries.sql*
4. W razie potrzeby zmodyfikować plik *.env*
5. Uruchomić projekt przy użyciu komendy:

   ```
   npm run start
   ```

Metryka kodu:

Język programowania	Aplikacja	Ilość linii
JavaScript	Serwer	1304
JavaScript	Klient	1744

Rozdział 7.

Podsumowanie

Badanie architektury serwisów społecznościowych oraz jej implementacja stanowiły dla mnie wartościowy walor dydaktyczny.

Stworzyłem własny serwis inspirując się wiedzą pozyskaną w trakcie badania nowo mi poznanych wzorców. Z pozytywnym skutkiem zaprogramowałem komunikację bazującą na grafowym języku zapytań, zaimplementowałem autoryzację opartą o tokeny, wykorzystałem w praktyce wiedzę o wzorcu CQRS i Repository oraz nauczyłem się biblioteki React tworząc za jej pomocą aplikację kliencką z przejrzystym interfejsem zbudowanym przy użyciu Material-UI [43]. Dowiedziałem się również jak ważną rolę stanowi skalowalność w dużych serwisach.

Odwołując się do mojego osobistego planu mogę stwierdzić, że nie wszystko udało się wykonać. Brakuje mi implementacji wzorca powiadomień. Ponadto oryginalnie chciałem skorzystać z narzędzia DataLoader [44], które przygotowałoby mój projekt na wykorzystanie różnych źródeł danych oraz zapobiegło problemowi N+1, który może wystąpić w przyszłości wraz z rozwojem serwisu [45].

W dalszym rozwoju projektu chciałbym w szczególności dopracować stronę serwerową. Wprowadziłbym kolejny krok implementacji dla wzorca CQRS. Wiązałby się on z rozdzieleniem danych zapisu i odczytu pomiędzy dwa różne podmioty. Takie rozwiązanie wymagałoby zaimplementowania mechanizmu replikacji informacji pomiędzy tymi źródłami. Dodatkowo w myśl wzorca wykorzystania pamięci podręcznej użyłbym Redisa, by stworzyć warstwę pamięci cache w celu przyśpieszenia działania *QueryService*.

Bibliografia

[1] Słownik języka polskiego PWN
(ostatni dostęp: 2021-09-10)
https://sjp.pwn.pl/sjp/serwis-spolecznosciowy;5579205.html

[2] Node.js
(ostatni dostęp: 2021-09-10)
https://nodejs.org/en/

[3] React.js
(ostatni dostęp: 2021-09-10)
https://reactjs.org/

[4] Heroku: Cloud Application Platform
(ostatni dostęp: 2021-09-10)
https://www.heroku.com

[5] Git
(ostatni dostęp: 2021-09-10)
https://git-scm.com/

[6] PostgreSQL: The World's Most Advanced Open Source Relational Database
(ostatni dostęp: 2021-09-10)
https://www.postgresql.org/

[7] Timeline of social media
(ostatni dostęp: 2021-09-10)
https://en.wikipedia.org/wiki/Timeline_of_social_media

[8] Most popular social networks worldwide as of July 2021
(ostatni dostęp: 2021-09-10)
https://www.statista.com/statistics/272014/global-social-networks-ranked-by-number-of-users/

[9] CS50 Lecture by Mark Zuckerberg
(ostatni dostęp: 2021-09-10)
https://www.youtube.com/watch?v=xFFs9UgOAlE

[10] GraphQL
(ostatni dostęp: 2021-09-10)
https://graphql.org/

[11] Queries and Mutations
(ostatni dostęp: 2021-09-10)
https://graphql.org/learn/queries/

[12] "GraphQL: Designing a Data Language" by Lee Byron
(ostatni dostęp: 2021-09-10)
https://youtu.be/Oh5oC98ztvI

[13] The JavaScript reference implementation for GraphQL
(ostatni dostęp: 2021-09-10)
https://github.com/graphql/graphql-js

[14] GraphQL at Twitter
(ostatni dostęp: 2021-09-10)
https://about.sourcegraph.com/graphql/graphql-at-twitter/

[15] Scaling Memcache at Facebook
(ostatni dostęp: 2021-09-10)
https://www.usenix.org/system/files/conference/nsdi13/nsdi13-final170_update.pdf

[16] Memcached
(ostatni dostęp: 2021-09-10)
https://memcached.org/

[17] TAO: The power of the graph
(ostatni dostęp: 2021-09-10)
https://engineering.fb.com/2013/06/25/core-data/tao-the-power-of-the-graph/

[18] TAO: Facebook's Distributed Data Store for the Social Graph
(ostatni dostęp: 2021-09-10)
https://www.usenix.org/system/files/conference/atc13/atc13-bronson.pdf

[19] Redis
(ostatni dostęp: 2021-09-10)
https://redis.io/

[20] Scaling Redis at Twitter
(ostatni dostęp: 2021-09-10)
https://youtu.be/rP9EKvWt0zo

[21] Stack Overflow: How We Do App Caching - 2019 Edition
(ostatni dostęp: 2021-09-10)
`https://nickcraver.com/blog/2019/08/06/stack-overflow-how-we-do-app-caching/`

[22] Repository
(ostatni dostęp: 2021-09-10)
`https://martinfowler.com/eaaCatalog/repository.html`

[23] Unit of Work
(ostatni dostęp: 2021-09-10)
`https://martinfowler.com/eaaCatalog/unitOfWork.html`

[24] CQRS
(ostatni dostęp: 2021-09-10)
`https://martinfowler.com/bliki/CQRS.html`

[25] Event Sourcing
(ostatni dostęp: 2021-09-10)
`https://martinfowler.com/eaaDev/EventSourcing.html`

[26] How the Reddit Algorithm Works
(ostatni dostęp: 2021-09-10)
`https://www.datadial.net/blog/how-the-reddit-algorithm-works/`

[27] What are recommendations on Facebook?
(ostatni dostęp: 2021-09-10)
`https://www.facebook.com/help/1257205004624246`

[28] Powered by AI: Instagram's Explore recommender system
(ostatni dostęp: 2021-09-10)
`https://ai.facebook.com/blog/powered-by-ai-instagrams-explore-recommender-system/`

[29] OAuth 2.0 Authorization Framework
(ostatni dostęp: 2021-09-10)
`https://auth0.com/docs/authorization/protocols/protocol-oauth2`

[30] Express
(ostatni dostęp: 2021-09-10)
`https://expressjs.com/`

[31] GraphQL HTTP Server Middleware
(ostatni dostęp: 2021-09-10)
`https://github.com/graphql/express-graphql`

[32] Introduction to Apollo Client
(ostatni dostęp: 2021-09-10)
`https://www.apollographql.com/docs/react/`

[33] Insomnia
(ostatni dostęp: 2021-09-10)
https://insomnia.rest/

[34] GraphQL IDE
(ostatni dostęp: 2021-09-10)
https://github.com/graphql/graphiql

[35] GraphQL Playground
(ostatni dostęp: 2021-09-10)
https://github.com/graphql/graphql-playground

[36] node-postgres
(ostatni dostęp: 2021-09-10)
https://node-postgres.com/

[37] Don't repeat yourself
(ostatni dostęp: 2021-09-10)
https://en.wikipedia.org/wiki/Don%27t_repeat_yourself

[38] React Waypoint
(ostatni dostęp: 2021-09-10)
https://www.npmjs.com/package/react-waypoint

[39] JSON Web Tokens
(ostatni dostęp: 2021-09-10)
https://jwt.io/

[40] Apollo Link overview
(ostatni dostęp: 2021-09-10)
https://www.apollographql.com/docs/react/api/link/introduction/

[41] Token Refresh Link
(ostatni dostęp: 2021-09-10)
https://www.npmjs.com/package/apollo-link-token-refresh

[42] The Ultimate Guide to handling JWTs on frontend clients (GraphQL)
(ostatni dostęp: 2021-09-10)
https://hasura.io/blog/best-practices-of-using-jwt-with-graphql/

[43] Material-UI
(ostatni dostęp: 2021-09-10)
https://next.material-ui.com/

[44] DataLoader
(ostatni dostęp: 2021-09-10)
https://github.com/graphql/dataloader

[45] What is the N+1 Problem in GraphQL?
(ostatni dostęp: 2021-09-10)
`https://medium.com/the-marcy-lab-school/what-is-the-n-1-problem-in-graphql-dd4921cb3c1a`