AKADEMIA TECHNICZNO-INFORMATYCZNA W NAUKACH STOSOWANYCH

Wydział Informatyki

Jakub Szpak

Porównanie działania interfejsów programowania aplikacji: REST i GRAPHQL

Praca: magisterska
Kierunek: Informatyka
Specjalność: Programowanie
Nr albumu: 7931

Praca wykonana pod kierunkiem:
dr hab. inż. Katarzyna Pentoś

Wrocław 2025

Spis treści

1 Wprowadzenie **3**

1.1 Cel i zakres pracy 3

2 Wstęp teoretyczny **5**

2.1 Interfejsy programowania aplikacji (API) 5

2.2 Architektura REST 5

2.3 Język GraphQL 6

2.4 Relacyjne bazy danych i model danych 6

2.4.1 Normalizacja danych w bazach relacyjnych 7

3 Koncepcja pracy **8**

3.1 Baza danych 8

3.2 Struktura bazy danych 8

3.3 Generowanie modelu Prisma 10

3.4 Wybór technologii backendowych 12

4 Implementacja **14**

4.1 GraphQL 14

4.1.1 Encje 14

4.1.2 Resolvery 15

4.1.3 Uruchamianie serwera GraphQL 15

4.2 REST 15

4.2.1 Plik serwera Express 15

4.2.2 Routery 16

5 Testy porównawcze **17**

5.1 Metodyka badań 17

5.2 Pobranie pełnych informacji o produkcie 18

5.3 Pobranie danych o kategorii oraz listy produktów (Underfetching) 20

5.4 Pobranie listy użytkowników z ich zamówieniami 23

5.4.1 Problem N + 1 27

5.5 Pobranie częściowych danych o produkcie (Overfetching) 32

6 Pozostałe kluczowe różnice REST i GraphQL **35**
6.1 Uwierzytelnienie i Autoryzacja 35
6.2 Obsługa błędów 37
6.3 Rozwój i dokumentacja API 37
6.4 Podsumowanie 37

7 Wnioski **38**

Rozdział 1

Wprowadzenie

1.1 Cel i zakres pracy

Celem pracy było porównanie dwóch popularnych podejść do tworzenia interfejsów programowania aplikacji (API) - REST (Representational state transfer) oraz GraphQL - pod względem ich architektury, wydajności, elastyczności, złożoności implementacji, zastosowań praktycznych oraz mocnych i słabych stron. Dokonano praktycznego porównania poprzez implementację dwóch aplikacji, korzystających z tej samej bazy danych PostgreSQL.
Porównanie zostało przeprowadzone w kontekście podstawowej wersji aplikacji obsługującej sklep internetowy e-commerce, która umożliwia zarządzanie:

- produktami,
- kategoriami,
- zamówieniami,
- użytkownikami.

Celem jest wskazanie przypadków, w których dane podejście może być bardziej optymalne, oraz przedstawienie kompromisów i mankamentów związanych z wyborem konkretnej technologii.

Zakres pracy obejmuje:

- Wstęp teoretyczny
- Koncepcja pracy
- Implementacja
  - Przygotowanie wspólnej bazy danych PostgreSQL (hostowanej na Supabase)
  - Przygotowanie schematu modeli Prisma, na podstawie bazy danych, do późniejszego wykorzystania przez obie aplikacje
  - Aplikacja REST w opraciu o Express.js

  - Aplikacja GraphQL oparta o Apollo i TypeGraphQL

- Testy porównawcze
  - Przygotowanie realistycznych przypadków testowych umożliwiających porównanie działania REST i GraphQL w kontekście funkcjonującego sklepu internetowego oraz wskazanie ich zalet i ograniczeń
  - Pomiar wydajności zapytań (czas odpowiedzi, rozmiar danych)
  - Analiza przypadków nadmiarowego pobierania danych (over-fetching), braków potrzebnych danych w odpowiedzi na zapytanie (under-fetching) i problemu N + 1

- Pozostałe kluczowe różnice REST i GraphQL

- Omówienie wyników i wnioski

Rozdział 2

Wstęp teoretyczny

2.1 Interfejsy programowania aplikacji (API)

API (Application Programming Interface) to ugruntowane pojęcie informatyczne oznaczające interfejs, pewnego rodzaju kontrakt, który określa zasady komunikacji między różnymi programami lub komponentami programistycznymi. API definiuje, jak jedna aplikacja może korzystać z funkcji lub danych innej poprzez ustalony zestaw metod, formatów danych oraz adresów (endpointów) komunikacyjnych. W odróżnieniu od interfejsu użytkownika (UI), API jest przeznaczone do wykorzystywania przez inne oprogramowanie, a nie bezpośrednio przez człowieka [9, 18, 13].

Termin API ma długą historię w informatyce sięgającą kilkudziesięciu lat – zakres znaczeniowy tego terminu ewoluował wraz z rozwojem technologii. Ogólnie jednak we współczesnym ujęciu API stanowi warstwę abstrakcji udostępniającą pewną funkcjonalność lub dane, niezależnie od wewnętrznej implementacji systemu, tak aby inne programy mogły z niej korzystać w zdefiniowany, ustandaryzowany sposób. Taka abstrakcja ukrywa szczegóły implementacyjne i dostarcza jasno zdefiniowany zbiór usług – dzięki temu komponenty software'owe mogą integrować się ze sobą bez potrzeby ujawniania kodu źródłowego czy wewnętrznych struktur danych [9, 18].

W kontekście aplikacji webowych API najczęściej przyjmuje formę usługi sieciowej udostępniającej dane lub funkcje poprzez protokoły internetowe. Szczególnie popularne stały się tzw. **web API** opierające się na architekturze klient-serwer i protokole HTTP. W ostatnich latach dominującymi paradygmatami projektowania web API są style REST oraz język zapytań GraphQL – uchodzą one za najpowszechniej stosowane podejścia do tworzenia interfejsów usług sieciowych [13].

2.2 Architektura REST

REST (Representational State Transfer) to styl architektury zaproponowany przez Roya Fieldinga w jego pracy doktorskiej [8]. REST definiuje zestaw zasad projektowych dla aplikacji sieciowych: bezstanowość, ujednolicony interfejs, klient-serwer, możliwość buforowa-

nia(caching) oraz warstwową architekturę. Interfejs REST opiera się na protokole HTTP i operacjach na zasobach reprezentowanych przez URI. Kluczowe metody to: GET, POST, PUT, DELETE [19, 12].

REST zapewnia prosty i dobrze skalowalny model, dlatego zyskał dużą popularność. Niemniej jednak wiele implementacji REST odbiega od oryginalnych założeń np. nie wszystkie stosują HATEOAS (Hypermedia As The Engine Of Application State) - założenie, że klient interfejsu REST powinien być w stanie dynamicznie odkrywać dostępne akcje (operacje) na podstawie hipermediów zawartych w odpowiedziach serwera, bez wcześniejszej znajomości struktury API [19, 8]. REST świetnie nadaje się do prostych operacji CRUD i jest wspierany przez wiele narzędzi i bibliotek [13, 9, 12].

2.3 Język GraphQL

GraphQL to język zapytań oraz środowisko wykonawcze API, które umożliwia klientowi jednoznaczne określenie struktury oczekiwanych danych, tak aby serwer odpowiedział dokładnie w tym samym formacie, zwracając wyłącznie żądane pola w formacie JSON. Został opracowany przez Facebook w 2012 roku, a upubliczniony jako projekt otwartoźródłowy w 2015 [3].

GraphQL rozwiązuje problemy REST takie jak underfetching (niedobór danych – konieczność wykonania wielu zapytań w celu otrzymania żądanych danych) i overfetching (nadmiar danych – pobieranie zbędnych informacji), ponieważ klient wysyła jedno zapytanie obejmujące wiele powiązanych danych. Cała komunikacja odbywa się przez jeden endpoint, a dane są zwracane w formacie JSON zgodnym z zapytaniem [16, 2, 21, 15].

Schemat GraphQL jest silnie typowany i pozwala na introspekcję – klient może pobrać informacje o dostępnych typach i zapytaniach. Dodatkowo GraphQL obsługuje mutacje (zmiany danych) oraz subskrypcje (nasłuch na zmiany). Eliminuje potrzebę wersjonowania API – klienci wybierają tylko te pola, które są im potrzebne [3, 16].

2.4 Relacyjne bazy danych i model danych

Większość systemów wykorzystujących API opiera się na relacyjnych bazach danych. Model relacyjny wprowadził E. F. Codd w 1970 roku [4]. Dane są przechowywane w tabelach, a relacje odwzorowuje się przez klucze obce. Każda tabela ma klucz główny, który jednoznacznie identyfikuje wiersze.

Relacyjne bazy danych zapewniają niepodzielność, spójność, izolację, trwałość (ACID - atomicity, consistency, isolation, durability). Operacje wykonywane są za pomocą języka SQL (Structured Query Language). Przykładowe systemy RDBMS (Relacyjny System Zarządzania Bazą Danych) to PostgreSQL, MySQL, Oracle, SQL Server [7, 22, 5].

2.4.1 Normalizacja danych w bazach relacyjnych

Normalizacja to proces optymalizacji struktury bazy danych w celu redukcji redundancji i zapobiegania anomaliom modyfikacyjnym. Główne postacie normalne to:

- 1NF – eliminacja wartości złożonych, każda kolumna zawiera wartości atomowe,
- 2NF – eliminacja częściowych zależności od klucza głównego,
- 3NF – eliminacja zależności przechodnich [4, 6].

Normalizacja prowadzi do lepszej organizacji danych i ułatwia późniejsze zarządzanie bazą. Jednak nadmierna normalizacja może utrudnić dostęp do danych, dlatego czasem stosuje się świadomą denormalizację [7, 5].

Rozdział 3

Koncepcja pracy

3.1 Baza danych

Warstwa bazy danych stanowi fundament każdej aplikacji, która przetwarza i przechowuje dane użytkowników. Zarówno w architekturze opartej na REST, jak i GraphQL, sposób zaprojektowania oraz obsługi bazy danych ma kluczowe znaczenie dla wydajności i stabilności systemu. W niniejszej pracy postanowiono wykorzystać relacyjną bazę danych PostgreSQL, będącą jednym z najczęściej wybieranych rozwiązań w środowisku aplikacji internetowych. Popularność PostgreSQL wśród developerów systematycznie rośnie – zgodnie z wynikami ankiety Stack Overflow, już 49% ankietowanych programistów deklaruje użycie PostgreSQL, co czyni go najpopularniejszą bazą danych drugi rok z rzędu (dla porównania w 2018 roku użycie wynosiło 33%).

> „PostgreSQL debuted in the developer survey in 2018 when 33% of developers reported using it, compared with the most popular option that year: MySQL, in use by 59% of developers. Six years later, PostgreSQL is used by 49% of developers and is the most popular database for the second year in a row.” [11]

Dzięki swojej stabilności, zgodności ze standardem SQL oraz bogatej dokumentacji, PostgreSQL stanowi solidny fundament dla aplikacji e-commerce. W szczególności w przypadku aplikacji takiej jak opisana w niniejszej pracy – sklepu internetowego z obsługą zamówień – kluczowe jest zapewnienie spójności i szybkości operacji bazodanowych. Bazę danych wdrożono przy użyciu usługi hostingu Supabase (oferującej m.in. zarządzaną instancję PostgreSQL w chmurze). Następnie, na potrzeby testów i porównań, zapełniono ją przykładowymi danymi, uruchamiając dedykowany skrypt SQL przygotowujący odpowiednie tabele oraz rekordy.

3.2 Struktura bazy danych

Na potrzeby eksperymentów porównawczych zaprojektowano model danych odwzorowujący podstawowe funkcjonalności sklepu internetowego. Obejmuje on tabele: `products`,

`categories`, `users`, `orders`, `order_items` oraz `product_categories` (tabela pośrednia realizująca relację wiele-do-wielu między produktami a kategoriami). Struktura ta pozwala przechowywać informacje o produktach i ich kategoriach, użytkownikach oraz składanych przez nich zamówieniach wraz ze szczegółami pozycji zamówienia. Kluczową rolę odgrywa tabela `order_items`, która wiąże zamówienia z poszczególnymi produktami, umożliwiając odwzorowanie listy pozycji wchodzących w skład danego zamówienia – jest to typowe rozwiązanie w domenie e-commerce, pozwalające modelować relację zamówienie–produkt jako jeden-do-wielu (jedno zamówienie może zawierać wiele pozycji, a każda pozycja dotyczy jednego produktu).

Przy projektowaniu schematu przyjęto zasady normalizacji opisane w poprzednim rozdziale, w szczególności doprowadzając model do trzeciej postaci normalnej. Zastosowanie normalizacji wyeliminowało redundancję danych (np. informacje o produktach nie powtarzają się w tabeli zamówień, lecz są do nich referencyjnie odwoływane przez klucze obce) oraz zapewniło integralność referencyjną pomiędzy tabelami. Dzięki temu uzyskano spójny i przejrzysty schemat bazy danych, który można łatwo rozszerzać o nowe funkcjonalności. Finalny schemat bazy danych został przedstawiony na rysunku 3.1.

Rysunek 3.1: Schemat bazy danych aplikacji (model relacyjny sklepu internetowego)

3.3 Generowanie modelu Prisma

W celu ułatwienia pracy z bazą danych po stronie backendu oraz zapewnienia pełnego bezpieczeństwa typów w aplikacji napisanej w TypeScript, wykorzystano nowoczesne narzędzie ORM – Prisma. Prisma umożliwia wygenerowanie klienta bazodanowego na podstawie pliku definicji schematu (`schema.prisma`), co gwarantuje typowane zapytania i zmniejsza ryzyko błędów podczas wykonywania operacji na bazie danych. Jak podaje oficjalna dokumentacja tego narzędzia:

> „Prisma is an ORM for Node.js and TypeScript that gives you the benefits of type-safety at zero cost by auto-generating types from your database schema. It's ideal for building reliable data intensive application. Prisma makes you more confident and productive when storing data in a relational database. You can use it with any Node.js server framework to interact with your database." [17]

W ramach projektu, po utworzeniu struktury bazy danych w serwisie Supabase, skorzystano z polecenia `npx prisma db pull`, aby automatycznie wygenerować w pliku `schema.prisma` modele odpowiadające tabelom w bazie danych. Następnie, za pomocą komendy `npx prisma generate`, utworzono klienta *Prisma Client* (pakiet `@prisma/client` dla Node.js), który zapewnia wygodne metody do wykonywania zapytań na bazie danych z gwarancją zgodności typów z zdefiniowanym schematem. Wygenerowany kod zawiera klasy i metody odwzorowujące wszystkie tabele oraz relacje z bazy PostgreSQL, co pozwoliło warstwie backend (zarówno w implementacji REST, jak i GraphQL) korzystać z jednego spójnego modelu danych.

Warto wspomnieć, że Supabase poza dostępem do bazy danych udostępnia również natywny interfejs REST (biblioteka `@supabase/supabase-js`), umożliwiający bezpośrednią komunikację z bazą poprzez zapytania HTTP. Jednak na potrzeby rzetelnego porównania wydajności REST i GraphQL zrezygnowano z tego uproszczenia i wykorzystano Prismę jako jednolitą warstwę pośrednią w obu wariantach backendu. Pozwoliło to na pełną kontrolę nad generowanymi zapytaniami SQL oraz zoptymalizowanie logiki po stronie serwera w analogiczny sposób dla REST i GraphQL. Dzięki temu porównanie wydajności obu technologii odbywało się w porównywalnych warunkach, przy wspólnym podłożu danych i bez wpływu niestandardowych optymalizacji specyficznych dla jednej z technologii.

Finalny kształt pliku `schema.prisma` (zawierającego definicje wszystkich modeli ORM) został przedstawiony w listingu 3.1.

```
1 generator client {
2   provider = "prisma-client-js"
3 }
4
5 datasource db {
6   provider = "postgresql"
7   url      = env("DATABASE_URL")
8 }
```

```
9
10 model categories {
11   id                 Int                 @id @default(autoincrement())
12   name               String              @db.VarChar(100)
13   product_categories product_categories[]
14 }
15
16 model order_items {
17   id         Int       @id @default(autoincrement())
18   order_id   Int?
19   product_id Int?
20   quantity   Int
21   price      Decimal   @db.Decimal(10, 2)
22   orders     orders?   @relation(fields: [order_id], references: [id],
       onDelete: Cascade)
23   products   products? @relation(fields: [product_id], references: [id])
24
25   @@index([order_id], map: "idx_order_items_order_id")
26 }
27
28 model orders {
29   id          Int           @id @default(autoincrement())
30   user_id     Int?
31   total       Decimal       @db.Decimal(10, 2)
32   status      String?       @default("pending") @db.VarChar(50)
33   created_at  DateTime?     @default(now()) @db.Timestamp(6)
34   order_items order_items[]
35   users       users?        @relation(fields: [user_id], references: [id
       ])
36
37   @@index([user_id], map: "idx_orders_user_id")
38 }
39
40 model product_categories {
41   product_id  Int
42   category_id Int
43   categories  categories @relation(fields: [category_id], references: [id
       ], onDelete: Cascade)
44   products    products   @relation(fields: [product_id], references: [id
       ], onDelete: Cascade)
45
46   @@id([product_id, category_id])
47   @@index([category_id], map: "idx_product_categories_category_id")
48   @@index([product_id], map: "idx_product_categories_product_id")
49 }
50
51 model products {
52   id                 Int                 @id @default(autoincrement())
53   name               String              @db.VarChar(255)
```

```
54    description         String?
55    price               Decimal              @db.Decimal(10, 2)
56    stock               Int                  @default(0)
57    created_at          DateTime?            @default(now()) @db.Timestamp
          (6)
58    order_items         order_items[]
59    product_categories  product_categories[]
60  }
61
62  model users {
63    id             Int       @id @default(autoincrement())
64    email          String    @unique @db.VarChar(255)
65    password_hash  String
66    name           String?   @db.VarChar(100)
67    created_at     DateTime? @default(now()) @db.Timestamp(6)
68    orders         orders[]
69  }
```

Listing 3.1: Plik schema.prisma ze schematem bazy danych `schema.prisma`

3.4 Wybór technologii backendowych

W obu zaimplementowanych wariantach backendu — REST oraz GraphQL — kluczowym założeniem był wybór języka TypeScript. Jako nadzbiór JavaScriptu z systemem statycznego typowania, TypeScript pozwala na wcześniejsze wykrywanie błędów typów i niespójności, co znacząco podnosi bezpieczeństwo, skalowalność i jakość kodu źródłowego aplikacji [23]. Jak ujął to autor kursu "Learn Typescript":

> „TypeScript is JavaScript that scales" [20]

Dla implementacji wariantu REST wybrano minimalistyczny framework Node.js – **Express.js**. Jest to sprawdzone rozwiązanie do budowania serwerów HTTP, cechujące się prostotą i bogatym ekosystemem middleware. Express umożliwia definiowanie routerów obsługujących poszczególne ścieżki URL oraz metod HTTP, co idealnie współgra z modelem REST opartym na zasobach. Jego niewielka narzutowość i powszechne zastosowanie w projektach produkcyjnych sprawiają, że stanowi naturalny wybór przy budowie klasycznego API REST.

W przypadku wariantu GraphQL wykorzystano natomiast bibliotekę **Apollo Server** w połączeniu z frameworkiem **TypeGraphQL**. Apollo Server to jedno z najpopularniejszych rozwiązań do obsługi zapytań GraphQL w środowisku Node.js – odpowiada za parsowanie zapytań, wywoływanie odpowiednich resolverów i zwracanie wyników w formacie JSON. Z kolei TypeGraphQL umożliwia definiowanie schematu GraphQL za pomocą dekoratorów i klas w kodzie TypeScript, zamiast ręcznego tworzenia statycznego pliku schematu (SDL). Takie podejście pozwala na ścisłą integrację logiki biznesowej z definicją API. Developer

pisząc kod resolvera jednocześnie poprzez dekoratory określa, jakie typy i pola udostępnia API GraphQL. Eliminowane jest ryzyko rozbieżności między implementacją a specyfikacją API, ponieważ całość jest kontrolowana przez kompilator TypeScript. W efekcie łatwiej jest zapewnić spójność definicji pól, typów i operacji w całym projekcie. Co więcej, podejście to redukuje ilość powtarzalnego kodu i upraszcza zarządzanie zmianami w API.

Podsumowując, warstwa backend została zrealizowana w dwóch wariantach (REST i GraphQL) przy użyciu zbliżonego stosu technologicznego (Node.js + TypeScript) oraz wspólnej bazy danych. Pozwoliło to skupić się na różnicach wynikających bezpośrednio z użycia odmiennych stylów API. Implementacje zostały zaprojektowane tak, by oba interfejsy oferowały zbliżony zakres funkcjonalności aplikacji sklepowej. Dzięki temu w dalszych rozdziałach możliwe jest przeprowadzenie miarodajnego porównania działania i wydajności API REST oraz GraphQL w identycznych scenariuszach użycia.

Rozdział 4

Implementacja

4.1 GraphQL

4.1.1 Encje

Proces implementacji API GraphQL rozpoczął się od zdefiniowania wszystkich encji (entities), które stanowiły podstawę do obsługi zapytań i mutacji w GraphQL. Encje te powstały na bazie modeli generowanych przez Prisma Client (`@prisma/client`) i zostały zmapowane przy użyciu dekoratorów udostępnianych przez TypeGraphQL. Dzięki temu możliwe było odwzorowanie struktur danych zwracanych przez Prisma oraz pełna kontrola nad tym, jakie pola mają być udostępniane w zapytaniach GraphQL.

W praktyce każda klasa reprezentująca encję w GraphQL zawierała dekoratory @ObjectType() oraz @Field(), które pozwalają precyzyjnie zdefiniować typy i ewentualną opcjonalność poszczególnych pól.

Przykładowa definicja encji Order (Zamówienie) została przedstawiona na Listingu 4.1.

```
1  @ObjectType()
2  export class Order {
3    @Field(() => Int)
4    id!: number;
5
6    @Field(() => Float)
7    total!: Decimal;
8
9    @Field(() => [OrderItem])
10   order_items?: OrderItem[];
11 }
```

Listing 4.1: Definicja encji Order

4.1.2 Resolvery

Kolejnym krokiem było przygotowanie resolverów — czyli klas obsługujących logikę wykonywania zapytań i mutacji w GraphQL. Resolvery również wykorzystują dekoratory TypeGraphQL (np. @Query(), @Mutation(), @FieldResolver()), a w połączeniu z klientem Prisma pozwalają w łatwy sposób wykonywać zapytania do bazy danych oraz zwracać wyniki w formacie zgodnym z deklarowanymi typami GraphQL.

Przykładowy resolver zwracający produkty, będące częścią danego zamówienia, pokazano w Listingu 4.2.

```
@Resolver(() => OrderItem)
export class OrderItemResolver {
  @FieldResolver(() => Product, { nullable: true })
  async products(@Root() item: OrderItem) {
    if (!item.product_id) return null;
    return prisma.products.findUnique({
      where: { id: item.product_id },
    });
  }
}
```

Listing 4.2: Definicja resolvera OrderItem

4.1.3 Uruchamianie serwera GraphQL

Ostatnim etapem implementacji API GraphQL było przygotowanie pliku głównego odpowiedzialnego za uruchomienie całego serwera. Wewnątrz użyto funkcji `buildSchema()` z TypeGraphQL, która automatycznie generuje schemat GraphQL na podstawie wcześniej zdefiniowanych resolverów i dekorowanych encji. Finalnym krokiem było utworzenie instancji Apollo Server i przekazanie jej utworzonego schematu.

4.2 REST

4.2.1 Plik serwera Express

W przypadku API REST proces implementacji rozpoczął się od utworzenia pliku głównego `index.ts` i inicjalizacji serwera Express.js. Podobnie jak w API GraphQL, również w tym przypadku, kluczowe znaczenie miały modele danych generowane przez Prisma, co pozwoliło na zachowanie spójności typów w całym projekcie. W odróżnieniu od przypadku GraphQL, konwencja REST, z reguły, w żaden sposób nie narzuca konieczności definiowania typu zwracanych danych.

W przypadku API REST użytkownik nie ma możliwości wyboru konkretnych pól zwracanych w odpowiedzi, chyba że wcześniej zostanie przygotowany specjalny parametr lub

mechanizm umożliwiający taką selekcję. Zdecydowanie bardziej charakterystycznym sposobem rozwiązania problemu nadmiarowego pobierania danych (overfetchingu) w architekturze REST jest definiowanie osobnych endpointów, które dostarczają jedynie wybrane podzbiory informacji. W rezultacie proces projektowania i implementacji endpointów w przypadku REST był zdecydowanie mniej złożony.

4.2.2 Routery

W implementacji API REST kluczową rolę odgrywa przejrzyste rozdzielenie logiki aplikacji. W projekcie wszystkie endpointy zostały podzielone na osobne `routery`, zorganizowane w dedykowanych plikach. Na przykład, plik `routes/products.ts` odpowiada za obsługę wszystkich tras związanych z operacjami na obiektach Products. Podobnie, dla innych zasobów — takich jak categories, orders czy order_items — przygotowano osobne pliki, w których umieszczono logikę specyficzną dla danej grupy endpointów.

Każdy router wykorzystuje mechanizm udostępniany przez Express, umożliwiający definiowanie tras (`express.Router()`) oraz przypisywanie im odpowiednich funkcji obsługi żądań (kontrolerów). Następnie wszystkie routery są importowane w głównym pliku serwera i rejestrowane w aplikacji Express za pomocą metody `app.use()`. Dzięki temu uzyskano czytelną i łatwą do rozbudowy strukturę, w której każdy zasób aplikacji ma swój wydzielony moduł odpowiedzialny za logikę działania.

Przykładowy router, obsługujący endpoint, który zwraca informacje o produkcie o danym identyfikatorze, został przedstawiony na Listingu 4.3.

```
1  router.get("/:id", async (req: Request, res: Response) => {
2    const id = parseInt(req.params.id);
3    const product = await prisma.products.findUnique({
4      where: { id },
5    });
6
7    if (!product)
8      return void res.status(404).json({ error: "Product not found" });
9    res.json(product);
10 });
11
12 export default router;
```

Listing 4.3: Router products.ts

Rozdział 5

Testy porównawcze

5.1 Metodyka badań

Celem badań było porównanie wydajności i elastyczności dwóch współczesnych podejść do budowy interfejsów API: REST oraz GraphQL. Porównanie zostało przeprowadzone na przykładzie identycznej aplikacji typu e-commerce, zbudowanej w dwóch wersjach, różniących się wyłącznie warstwą API.

Opis środowiska testowego

Wszystkie testy zostały przeprowadzone lokalnie na tym samym komputerze, z wyłączonymi wszystkimi zbędnymi procesami systemowymi i aplikacjami, aby zmniejszyć wpływ czynników zewnętrznych na pomiar.

Sprzęt i środowisko:

- Procesor: AMD Ryzen 7 7800X3D 8-Core 4.20 GHz
- RAM: 32.0 GB
- System operacyjny: Windows 10
- Node.js: 22.14.0
- Prisma: 6.8.2
- Express.js (REST): 4.21.2
- Apollo Server / TypeGraphQL (GraphQL): 3.13.0 / 2.0.0-rc.2

W obu wariantach aplikacja korzystała z tej samej relacyjnej bazy danych PostgreSQL, udostępnionej przez platformę Supabase. Do komunikacji z bazą danych wykorzystano Prisma ORM, co zapewniło jednolity model danych i identyczne zapytania SQL w obu implementacjach.

Narzędzia pomiarowe i metodologia

Do przeprowadzenia testów wykorzystano narzędzie **Artillery**, które umożliwia symulację ruchu HTTP i dokładny pomiar parametrów wydajnościowych serwera. Analizowano m.in. czasy odpowiedzi (średni, minimalny, maksymalny, percentyle 95 i 99), rozmiar zapytań i odpowiedzi w bajtach oraz rozrzut czasów jako miarę stabilności działania. Każdy scenariusz testowy był powtarzany **10 razy**, a wyniki przedstawiono jako **średnie wartości** poszczególnych metryk.

Wszystkie testy były wykonywane na lokalnym hoście, a baza danych znajdowała się w chmurze (Supabase). Choć obecność zdalnej bazy danych wprowadzała dodatkowe opóźnienie sieciowe, to w obu implementacjach REST i GraphQL było ono identyczne, dzięki zastosowaniu tego samego ORM (Prisma) i wspólnego modelu danych.

Charakterystyka testów

Przeprowadzono **cztery testy** odwzorowujące realistyczne przypadki użycia API w aplikacjach e-commerce:

1. Pobranie pełnych informacji o produkcie,
2. Pobranie danych o kategorii wraz z przypisanymi produktami (Underfetching),
3. Pobranie listy użytkowników z ich zamówieniami (Problem N+1),
4. Pobranie częściowych danych o produkcie (Overfetching).

Każdy z przypadków testowych miał na celu uwidocznienie konkretnych różnic pomiędzy REST a GraphQL, zarówno w aspekcie wydajnościowym, jak i praktycznym podejściu do projektowania API. Szczegóły implementacyjne i wyniki każdego z testów zostały przedstawione w kolejnych rozdziałach pracy.

5.2 Pobranie pełnych informacji o produkcie

Pierwszy test skoncentrował się na pobraniu pełnych informacji o jednym produkcie, co stanowi podstawowy przypadek użycia. REST API realizowało to przez endpoint `GET /products/1`, natomiast GraphQL poprzez zapytanie:

```
query {
  product(id: 1) {
    id
    name
    description
    price
    stock
    created_at
  }
}
```

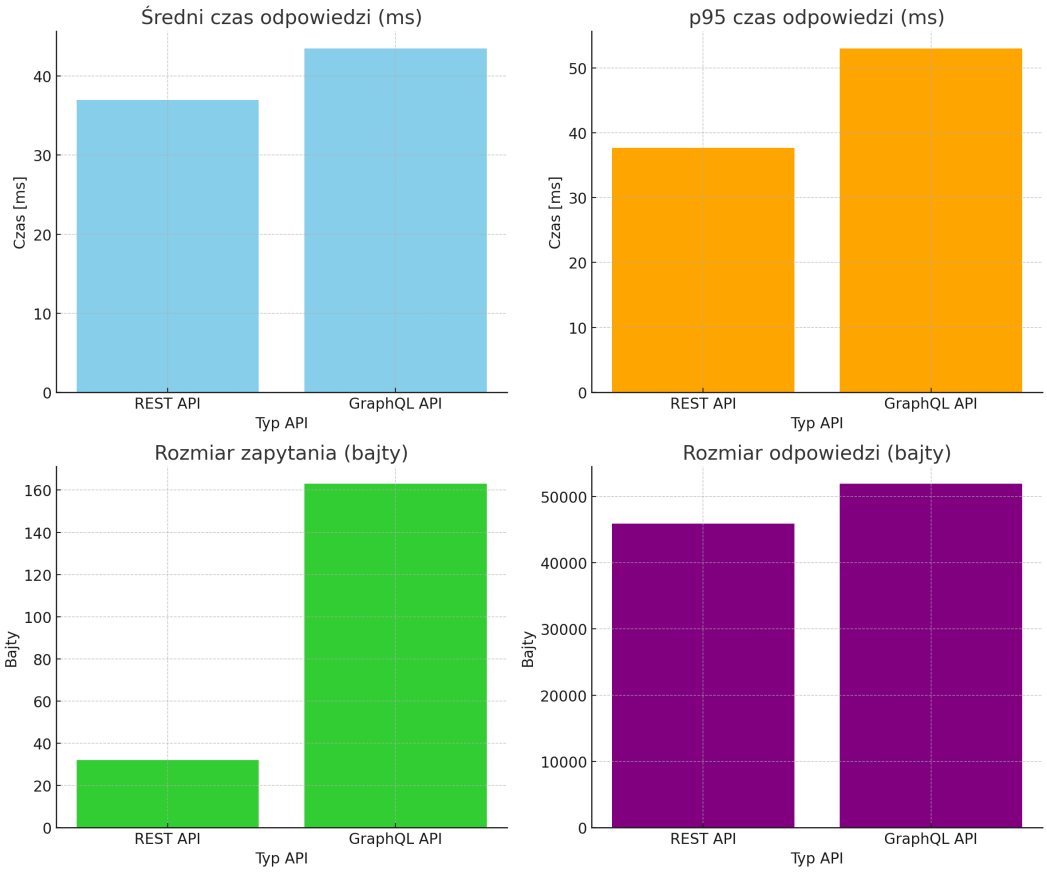
Każdy z testów obejmował 300 żądań w ciągu 30 sekund (10 żądań na sekundę). Wyniki pokazały, że REST API osiągało średni czas odpowiedzi 37 ms, podczas gdy GraphQL – 43.5 ms. Rozmiar odpowiedzi w GraphQL był o około 13% większy (51,9 kB dla GraphQL vs 45,9 kB dla REST). Ponadto REST API wykazało zdecydowanie większą stabilność – percentyl 99 czasów odpowiedzi wynosił 63.4 ms, podczas gdy dla GraphQL aż 327 ms. Te różnice można przypisać dodatkowym operacjom po stronie GraphQL, takim jak parsowanie zapytania, walidacja schematu oraz dynamiczne generowanie odpowiedzi.

Dodatkowo z pomocą narzędzia Artillery utworzono middleware, zbierający rozmiar zapytania (obejmujący zarówno URL, jak i body dla POST). Dla REST był on minimalny i wynosił 32 B, podczas gdy dla GraphQL 163 B – co wynika z konieczności przesyłania pełnego zapytania w formacie JSON. Te różnice ilustrują naturalny narzut związany z elastycznością i uniwersalnością GraphQL, który w prostych przypadkach generuje większy koszt w porównaniu do tradycyjnego REST.

Dokładne wyniki testów zostały przedstawione kolejno w Tabeli 5.1 oraz na wykresach na Rysunku 5.1.

Metryka	**REST API**	**GraphQL API**
Liczba zapytań	300	300
Średni czas odpowiedzi [ms]	37	43.5
Minimalny czas odpowiedzi [ms]	34	33
Maksymalny czas odpowiedzi [ms]	352	389
p95 czas odpowiedzi [ms]	37.7	53
p99 czas odpowiedzi [ms]	63.4	327.1
Średnia długość sesji [ms]	38.8	45.3
Rozmiar odpowiedzi [B]	45900	51900
Rozmiar jednego zapytania [B]	32	163

Tabela 5.1: Porównanie wyników testu 1 - Pobranie pełnego zestawu informacji o produkcie o danym id - tabela

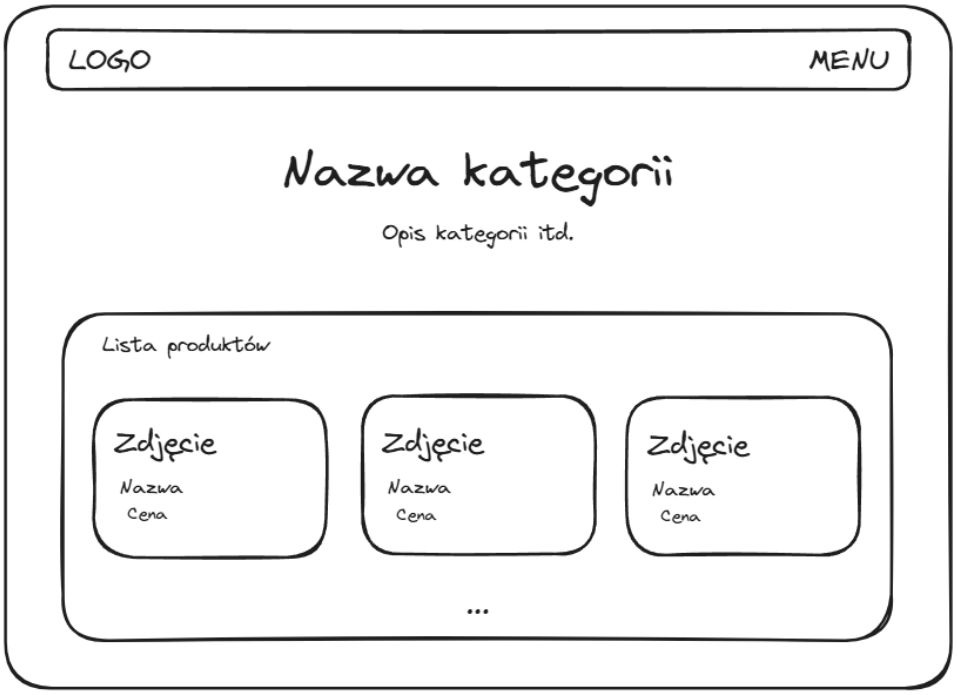


Rysunek 5.1: Porównanie wyników testu 1 - Pobranie pełnego zestawu informacji o produkcie o danym id - wykresy

5.3 Pobranie danych o kategorii oraz listy produktów (Underfetching)

Kolejny test porównawczy został zaplanowany w celu zbadania problemu underfetchingu w REST oraz elastyczności GraphQL przy pobieraniu danych z relacją *kategoria* → *produkty*. Underfetching oznacza sytuację, w której pojedyncze zapytanie API nie dostarcza wszystkich potrzebnych danych, przez co klient musi wykonać więcej niż jedno zapytanie.

Rysunek 5.2 przedstawia przykładowy układ aplikacji, wykonany przy pomocy narzędzia Excalidraw, w którym w górnej części wyświetlana jest nazwa kategorii, a poniżej lista przypisanych produktów. W REST API, aby uzyskać wszystkie te informacje, konieczne jest wykonanie dwóch zapytań: `GET /categories/1` (pobranie informacji o kategorii) oraz `GET /categories/1/products` (pobranie listy produktów) lub edycja jednego z endpointów. W teście założono, że drugie zapytanie jest wysyłane dopiero po zakończeniu pierwszego, co w tym wypadku nie byłoby konieczne, jednak jest częstą praktyką w aplikacjach webowych.



Rysunek 5.2: Przykładowy układ aplikacji sklepu internetowego, przedstawiający problem underfetching'u

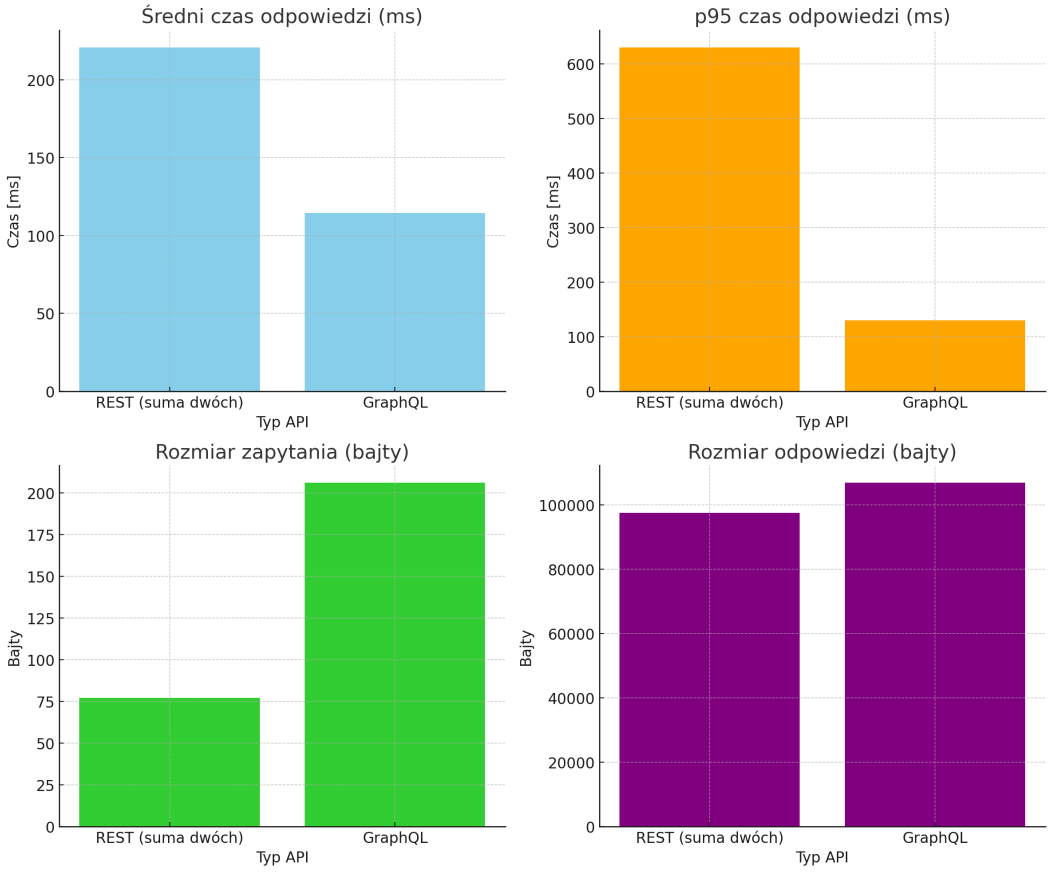
W GraphQL analogiczne dane można pobrać w jednym zapytaniu:

```
query {
  category(id: 1) {
    id
    name
    products {
      id
      name
      price
      description
      created_at
      stock
    }
  }
}
```

Takie rozwiązanie eliminuje problem underfetchingu i zmniejsza łączny czas oczekiwania. Test obejmował 300 cykli (w REST cykl to sekwencja dwóch zapytań, w GraphQL jednego) w ciągu 30 sekund (10 cykli na sekundę). Ponownie mierzono łączny czas odpowiedzi, rozmiar zapytania i odpowiedzi oraz stabilność obu podejść.

Metryka	REST (suma dwóch zapytań)	GraphQL
Liczba zapytań	600	300
Średni czas odpowiedzi [ms]	220.8	114.6
Minimalny czas odpowiedzi [ms]	137.0	104.0
Maksymalny czas odpowiedzi [ms]	785.0	489.0
p50 czas odpowiedzi [ms]	149.5	111.1
p95 czas odpowiedzi [ms]	629.9	130.3
p99 czas odpowiedzi [ms]	726	162.4
Średnia długość sesji [ms]	226.6	116.5
Rozmiar odpowiedzi [B]	97500	106800
Rozmiar jednego zapytania [B]	77	206

Tabela 5.2: Porównanie wyników testu 2 - Pobranie danych o kategorii oraz listy produktów (Underfetching) - tabela



Rysunek 5.3: Porównanie wyników testu 2 - Pobranie danych o kategorii oraz listy produktów (Underfetching) - wykresy

Tabela 5.2 i wykresy z Rysunku 5.3 przedstawiają szczegółowe porównanie wydajności i rozmiaru danych pomiędzy podejściem REST (dwa zapytania), a GraphQL (jedno zapytanie) w przypadku pobierania informacji o kategorii oraz powiązanych z nią produktów. Wyniki pokazują, że w tym przypadku użycie GraphQL wykazuje znaczną przewagę w czasie odpowiedzi – średni czas odpowiedzi w GraphQL wynosi 114.6 ms, natomiast w REST 220.8 ms (przy sumowaniu czasu dwóch zapytań). Podobnie wartości percentylowe (p50, p95, p99) są zauważalnie niższe dla GraphQL.

Różnice w rozmiarze zapytania i odpowiedzi pozostają zgodne z obserwacjami z testu 1 – zapytanie GraphQL jest większe, a odpowiedź nieco większa, co wynika z formatu JSON.

Wyniki jednoznacznie pokazują, że GraphQL eliminuje problem underfetchingu, pozwalając na pobranie pełnych danych w jednym zapytaniu, co skutkuje krótszym czasem oczekiwania i mniejszym opóźnieniem w aplikacji klienckiej. REST wymaga w tym scenariuszu dwóch zapytań, co naturalnie zwiększa sumaryczny czas odpowiedzi i zmienność wyników (wyższy p99). Różnica w rozmiarze zapytania jest jednak warta uwagi, szczególnie w przypadku środowisk mobilnych lub ograniczonych sieciowo.

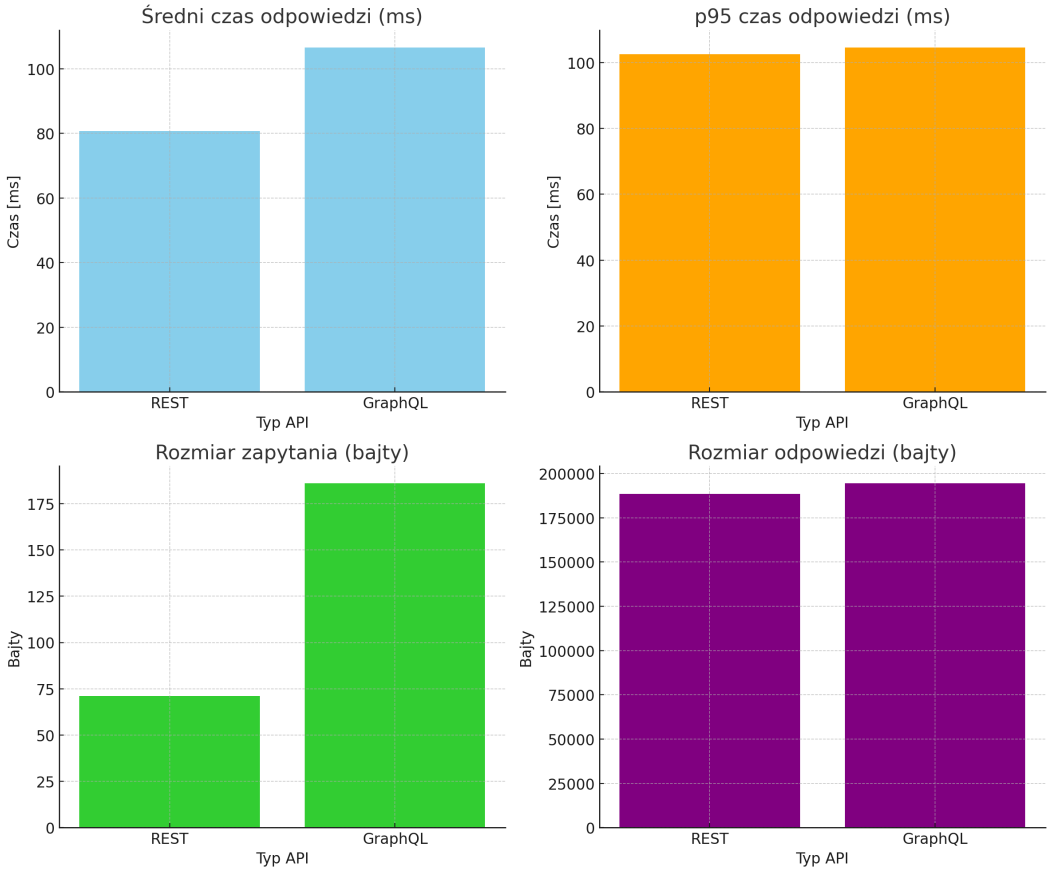
5.4 Pobranie listy użytkowników z ich zamówieniami

Kolejny test porównawczy został zaplanowany w celu zbadania wydajności obu podejść w scenariuszu, w którym klient potrzebuje pełnej listy użytkowników wraz z ich zamówieniami. W tym przypadku REST API realizuje to w jednym zapytaniu do endpointu `/users/orders`, zwracając kompletną strukturę danych dla wszystkich użytkowników. W GraphQL natomiast przygotowano zapytanie o listę użytkowników oraz powiązane z nimi zamówienia:

```
query {
  users {
    id
    email
    name
    created_at
    orders {
      id
      total
      created_at
      status
      user_id
    }
  }
}
```

Metryka	REST	GraphQL
Liczba zapytań	300	300
Średni czas odpowiedzi [ms]	80.7	106.6
Minimalny czas odpowiedzi [ms]	67	69
Maksymalny czas odpowiedzi [ms]	451	5434
p50 czas odpowiedzi [ms]	71.5	73
p95 czas odpowiedzi [ms]	102.5	104.6
p99 czas odpowiedzi [ms]	407.5	671.9
Średnia długość sesji [ms]	82.5	108.5
Rozmiar odpowiedzi [B]	188400	194501
Rozmiar jednego zapytania [B]	71	186

Tabela 5.3: Porównanie wyników testu 3 - Pobranie listy użytkowników z ich zamówieniami - tabela

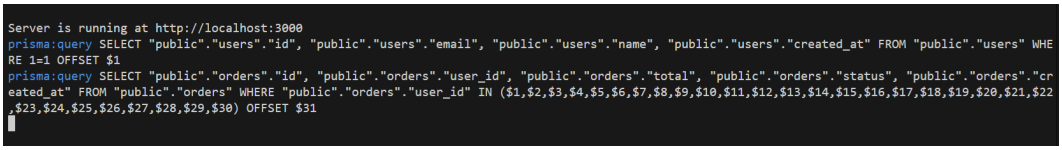


Rysunek 5.4: Porównanie wyników testu 3 - Pobranie listy użytkowników z ich zamówieniami - wykresy

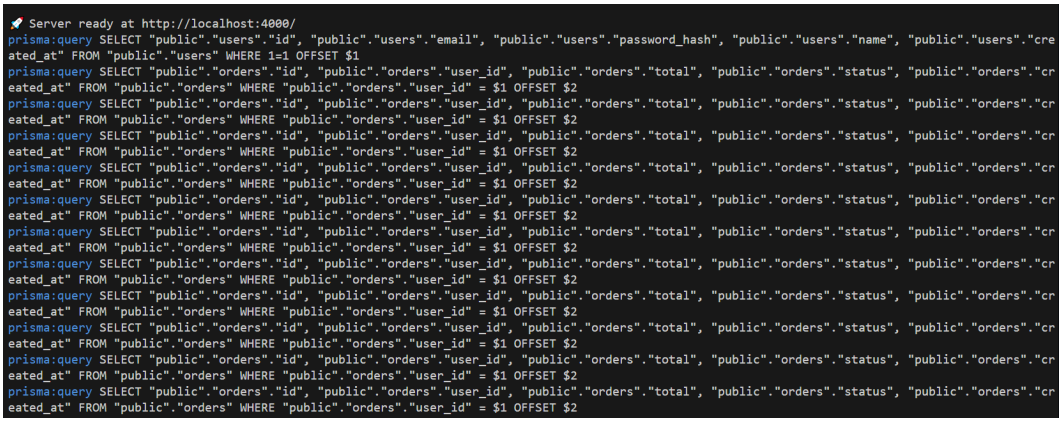
Tabela 5.3 oraz wykresy na Rysunku 5.4 przedstawiają wyniki testu wydajnościowego, w którym porównano pobieranie listy użytkowników wraz z ich zamówieniami. W scenariuszu REST API zastosowano jeden endpoint `/users/orders`, zwracający w jednym zapytaniu pełną strukturę danych. W GraphQL użyto pojedynczego zapytania, które pobiera użytkowników oraz ich zamówienia w ramach jednej operacji.

Średni czas odpowiedzi w tym teście był nieznacznie wyższy w GraphQL (106.6 ms) w porównaniu do REST (80.7 ms). Jednak różnice w sposobie realizacji tych zapytań stają się bardziej widoczne, gdy przyjrzymy się logom bazy danych. Prisma w REST, przy wykorzystaniu metody `findMany()`, generuje jedno zbiorcze zapytanie SQL, w którym korzysta z klauzuli `WHERE user_id IN (<lista_id_użytkowników>)`. Dzięki temu wszystkie zamówienia są pobierane w jednym zapytaniu (Rysunek 5.5).

To podejście wynika z tego, że w REST kontroler w Express wywołuje jedno zapytanie, obejmujące całą listę użytkowników i ich zamówień. Prisma automatycznie optymalizuje takie przypadki, łącząc pobieranie powiązanych rekordów w jednym zapytaniu SQL za pomocą klauzuli `IN`. Z drugiej strony, w GraphQL problem wynika z samej natury tego podejścia: FieldResolver `orders` w klasie `UserResolver` jest wywoływany osobno dla każdego użytkownika. W rezultacie generowane są osobne zapytania do tabeli `orders` dla każdego użytkownika (Rysunek 5.6), co prowadzi do większej liczby zapytań SQL.



Rysunek 5.5: Zrzut ekranu fragmentu logów, przedstawiający zapytania do bazy danych po uruchomieniu testu 3 w implementacji REST

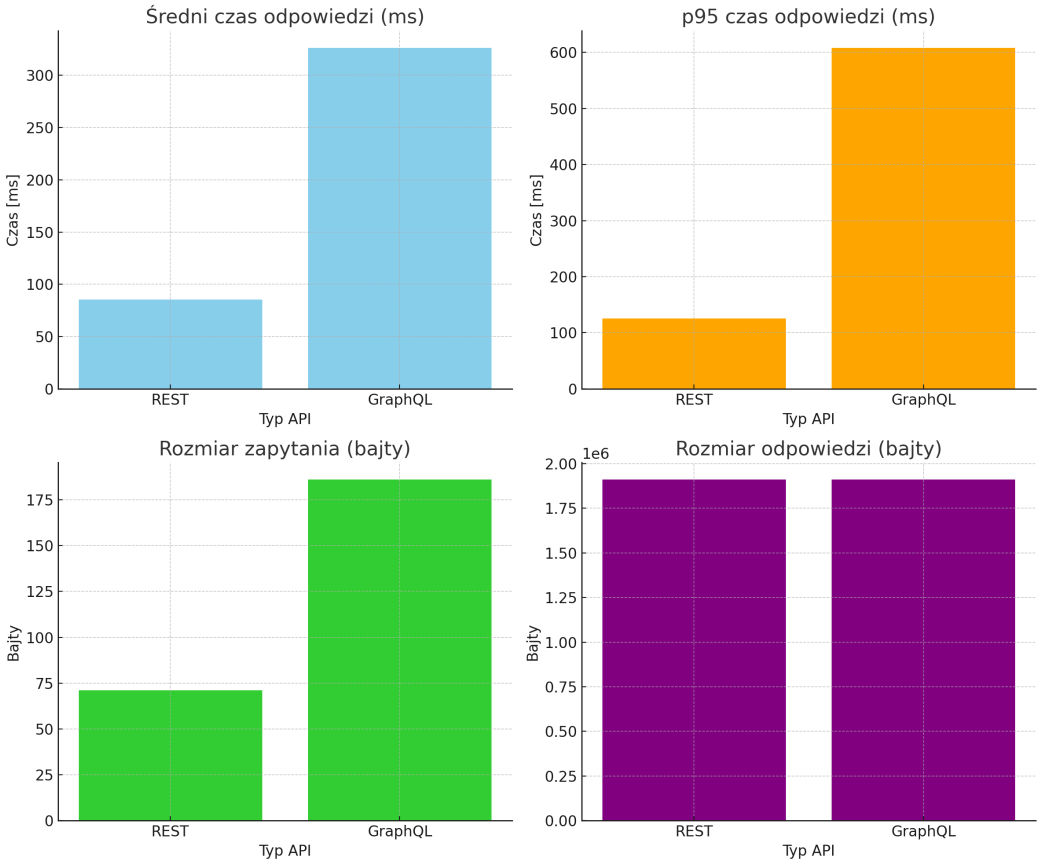


Rysunek 5.6: Zrzut ekranu fragmentu logów, przedstawiający zapytania do bazy danych po uruchomieniu testu 3 w implementacji GraphQL

Test został przeprowadzony w warunkach, gdy w bazie znajdowało się jedynie trzech użytkowników, z których każdy posiadał jedno zamówienie. Rodzi się zatem pytanie: co stanie się, gdy liczba użytkowników wzrośnie dziesięciokrotnie?

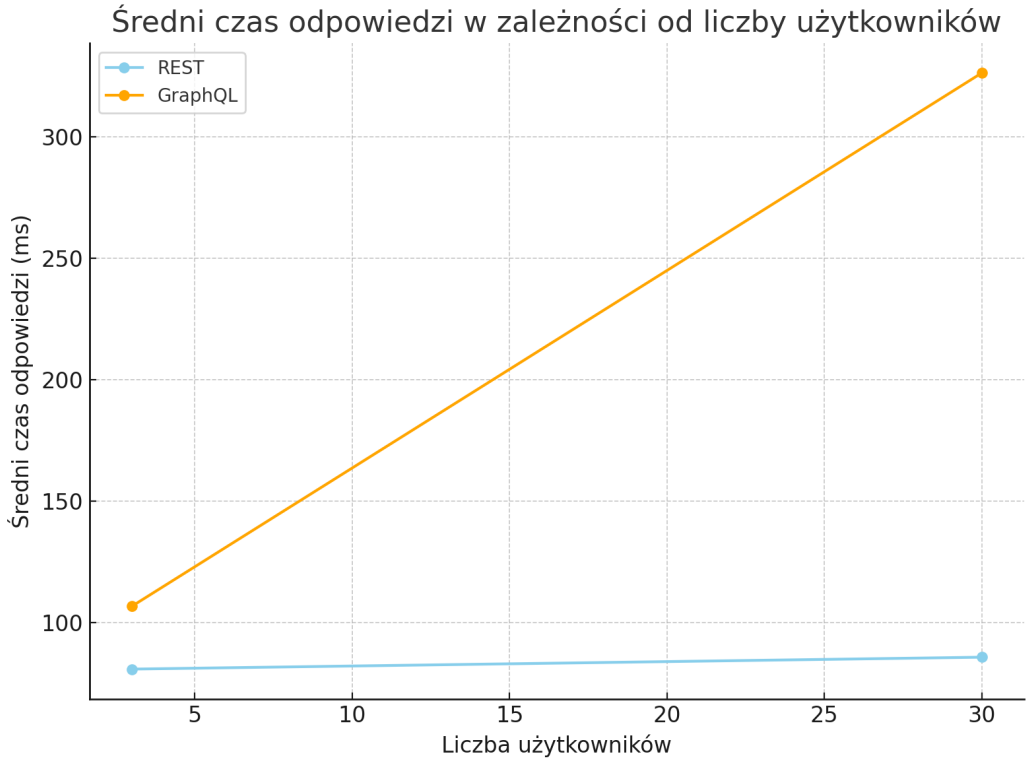
Metryka	REST	GraphQL
Liczba zapytań	300	300
Średni czas odpowiedzi [ms]	85.6	326.2
Minimalny czas odpowiedzi [ms]	66	109
Maksymalny czas odpowiedzi [ms]	467	5412
p50 czas odpowiedzi [ms]	71.5	202.4
p95 czas odpowiedzi [ms]	125.2	608
p99 czas odpowiedzi [ms]	424.2	5168
Średnia długość sesji [ms]	87.4	328.1
Rozmiar odpowiedzi [B]	1911000	1910420
Rozmiar jednego zapytania [B]	71	186

Tabela 5.4: Porównanie wyników testu 3 - Pobranie listy użytkowników z ich zamówieniami (po dziesięciokrotnym zwiększeniu liczby użytkowników i zamówień w bazie danych) - tabela



Rysunek 5.7: Porównanie wyników testu 3 - Pobranie listy użytkowników z ich zamówieniami (po dziesięciokrotnym zwiększeniu liczby użytkowników i zamówień w bazie danych) - wykresy

Po trzykrotnym zwiększeniu liczby użytkowników i zamówień, test wykazał znaczne pogorszenie wyników po stronie GraphQL (Tabela 5.4 i wykresy na Rysunku 5.7). Średni czas odpowiedzi wzrósł ponad trzykrotnie (z 106.6 ms do 326.2 ms), a wartości percentylowe (p95, p99) pokazują wyraźne pogorszenie stabilności i większy rozrzut czasów. Wyniki te jednoznacznie wskazują, że w przypadku większej liczby użytkowników warstwa GraphQL staje się wąskim gardłem wydajnościowym. Z kolei REST, mimo większej ilości danych, utrzymał stabilny poziom czasów odpowiedzi (średni czas wzrósł zaledwie o 5 ms), co pokazuje przewagę prostszego modelu endpointu w tym scenariuszu.



Rysunek 5.8: Średni czas odpowiedzi, w zależności od liczby użytkowników w REST i GraphQL - test 3

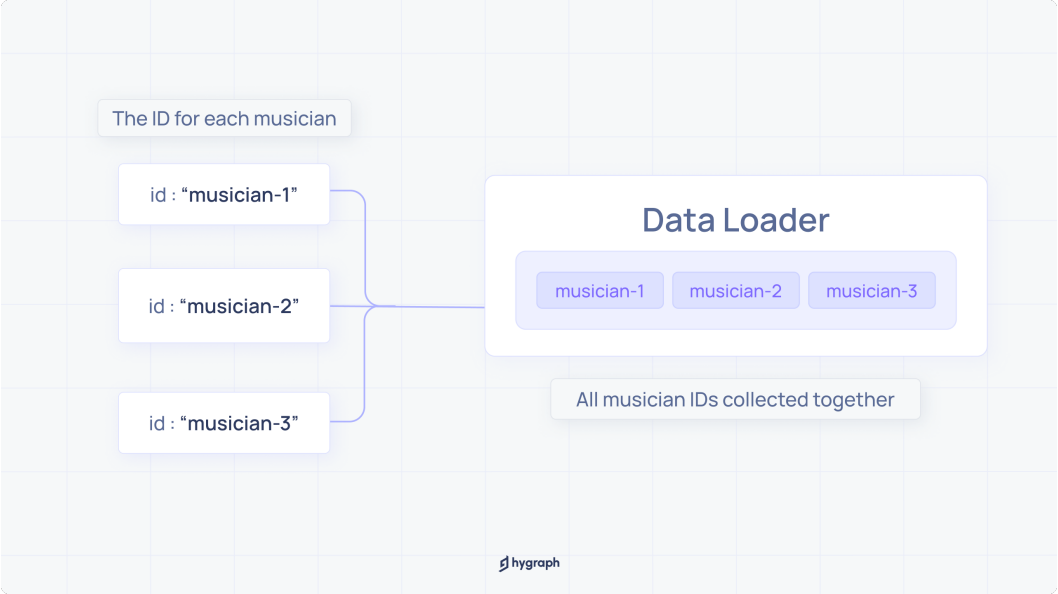
Na Rysunku 5.8 przedstawiono wykres zależności średniego czasu odpowiedzi od liczby użytkowników w bazie danych. Wyniki pokazują, że w przypadku REST wzrost liczby użytkowników nie wpływa znacząco na czas odpowiedzi, natomiast w GraphQL obserwujemy znaczący wzrost średniego czasu odpowiedzi wraz ze wzrostem liczby użytkowników.

5.4.1 Problem N + 1

Opisany przypadek testowy został zaprojektowany, aby zobrazować problem znany jako ”N+1”, który często pojawia się w aplikacjach korzystających z GraphQL. Problem N+1 występuje wówczas, gdy po pobraniu danych nadrzędnych (np. listy użytkowników) aplikacja wysyła osobne zapytanie do bazy dla każdego powiązanego rekordu podrzędnego (np. zamówienia użytkownika). W rezultacie liczba zapytań SQL rośnie liniowo wraz z liczbą rekordów nadrzędnych, co znacząco wpływa na wydajność całego systemu.

> The N+1 problem arises when your API makes one query to retrieve a list of items and then makes additional queries to retrieve related data for each item. This results in a large number of unnecessary database queries, which can slow down your application, especially when dealing with large datasets.[24]

W tym przypadku każdy resolver GraphQL dla zamówień użytkownika generował osobne zapytanie do tabeli `orders`, co prowadziło do znacznego wzrostu średniego czasu odpowiedzi. Rozwiązaniem tego problemu jest zastosowanie mechanizmu DataLoader, który umożliwia grupowanie zapytań do bazy w ramach jednego cyklu renderowania GraphQL i ich optymalizację przy pomocy klauzuli `IN`. Rysunek 5.9 ilustruje zasadę działania DataLoader na przykładzie sytuacji, w której to samo zapytanie o listę albumów, wykonywane jest wielokrotnie, dla zbioru identyfikatorów muzyków. Mechanizm grupuje je w jedną kolekcję i wykonuje jedno zbiorcze zapytanie do bazy danych, eliminując potrzebę wysyłania wielu pojedynczych zapytań. W dalszej części rozdziału opisano sposób implementacji tego rozwiązania w badanym przykładzie.



Rysunek 5.9: DataLoader - Mechanizm grupujący identyfikatory w jedną kolekcję, w celu wykonania jednego zbiorczego zapytania do bazy danych, eliminując potrzebę wysyłania wielu pojedynczych zapytań[1]

Aby skorzystać z mechanizmu DataLoader, należy jawnie zdefiniować funkcję, która jako argument przyjmuje listę identyfikatorów i zwraca strukturę danych mapującą każdy identyfikator na odpowiadający mu obiekt. Wewnątrz tej funkcji wykonywane jest jedno zbiorcze zapytanie do bazy danych, bazujące na przekazanej liście identyfikatorów. Przykład implementacji DataLoadera grupującego zapytania dla encji `orders` przedstawiono na Listingu 5.1.

```
export const orderLoader = new DataLoader<number, orders[]>(async (
    userIds) => {
  const orders = await prisma.orders.findMany({
    where: {
      user_id: {
        in: userIds as number[],
      },
    },
  });

  const ordersMap: Record<number, orders[]> = {};
  orders.forEach((order) => {
    if (!ordersMap[order.user_id!]) {
      ordersMap[order.user_id!] = [];
    }
    ordersMap[order.user_id!].push(order);
  });

  return userIds.map((userId) => ordersMap[userId] || []);
});
```

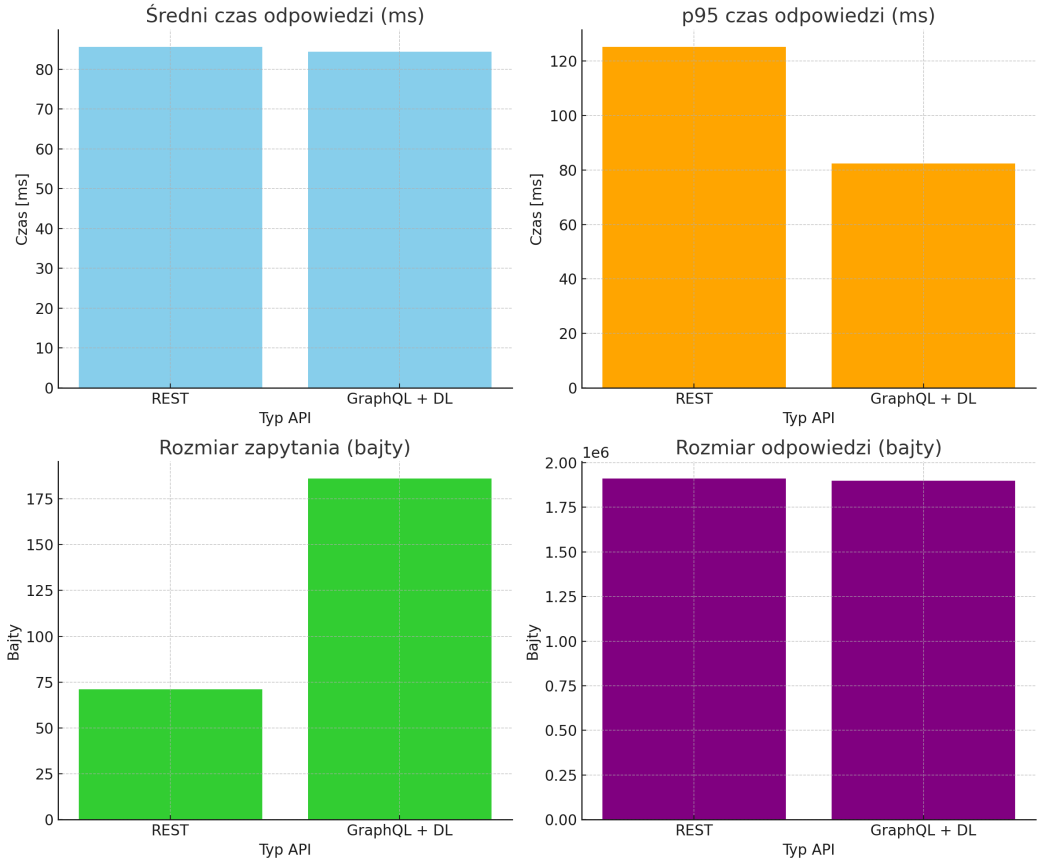
Listing 5.1: DataLoader, grupujący zapytania dla klasy Zamówienie

Po zastosowaniu mechanizmu DataLoader oraz wyłączeniu jego domyślnego mechanizmu buforowania (caching), powtórzono test. Wyniki zostały przedstawione w Tabeli 5.5 oraz na Wykresie 5.10. Zaktualizowano również wykres zależności średniego czasu odpowiedzi od liczby użytkowników w bazie danych (Rysunek 5.11), tak aby uwzględniał wyniki po zastosowaniu DataLoadera.

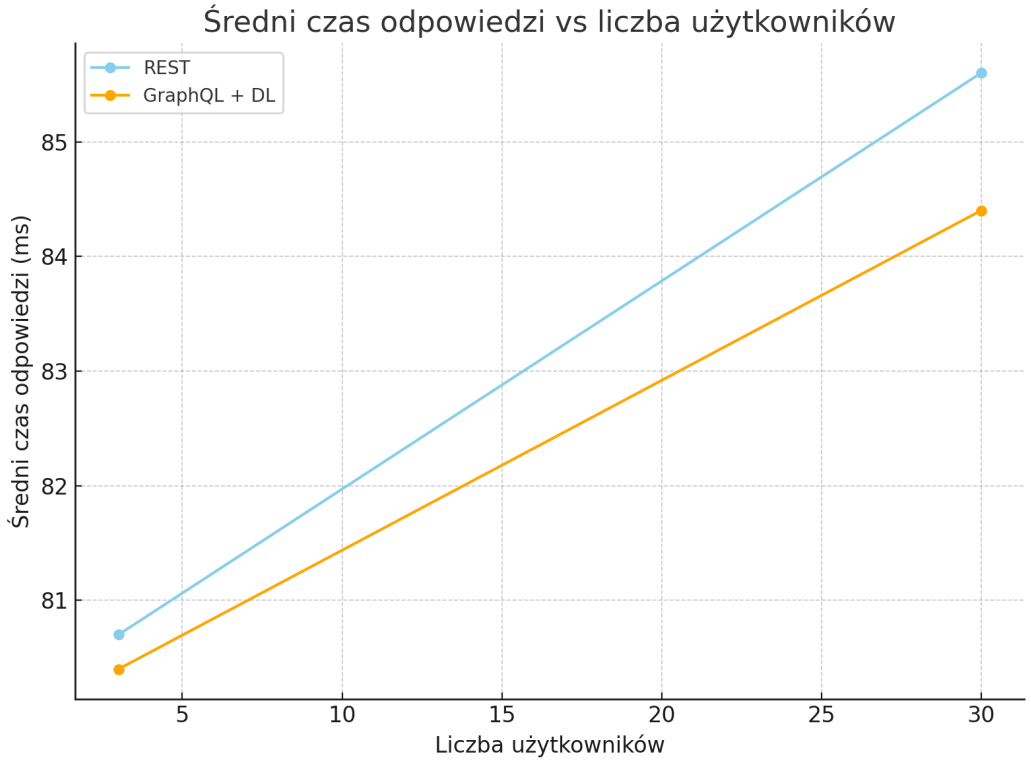
Zastosowanie tej optymalizacji pozwoliło znacząco ograniczyć liczbę zapytań do bazy danych i w efekcie zredukowało różnicę wydajności między GraphQL, a REST. Dla większej liczby użytkowników czasy odpowiedzi w obu podejściach stały się porównywalne.

Metryka	**REST**	**GraphQL + DataLoader**
Liczba zapytań	300	300
Średni czas odpowiedzi [ms]	85.6	84.4
Minimalny czas odpowiedzi [ms]	66	67
Maksymalny czas odpowiedzi [ms]	467	743
p50 czas odpowiedzi [ms]	71.5	71.5
p95 czas odpowiedzi [ms]	125.2	82.3
p99 czas odpowiedzi [ms]	424.2	584.2
Średnia długość sesji [ms]	87.4	86.3
Rozmiar odpowiedzi [B]	1911000	1899000
Rozmiar jednego zapytania [B]	71	186

Tabela 5.5: Porównanie wyników testu 3 - Pobranie listy użytkowników z ich zamówieniami (po dziesięciokrotnym zwiększeniu liczby użytkowników i zamówień w bazie danych) i implementacji mechanizmu DataLoader - tabela



Rysunek 5.10: Porównanie wyników testu 3 - Pobranie listy użytkowników z ich zamówieniami (po dziesięciokrotnym zwiększeniu liczby użytkowników i zamówień w bazie danych) i implementacji mechanizmu DataLoader - wykresy



Rysunek 5.11: Średni czas odpowiedzi, w zależności od liczby użytkowników w REST i GraphQL po zaimplementowaniu mechanizmem DataLoader - test 3

5.5 Pobranie częściowych danych o produkcie (Overfetching)

W ostatnim teście skupiono się na problemie overfetchingu, który polega na pobieraniu z serwera większej ilości danych niż jest faktycznie potrzebna po stronie klienta. Przykładem może być sytuacja, w której aplikacja potrzebuje jedynie nazw produktów przypisanych do danej kategorii, jednak standardowy endpoint REST zwraca całą strukturę obiektów produktów wraz z ich pełnymi danymi. Problem overfetchingu jest bardzo często podkreślany jako kluczowa przewaga GraphQL nad REST. Jak definiuje autor serii „How to GraphQL":

> „Overfetching means that a client downloads more information than is actually required in the app."[10]

Lee Byron, współtwórca GraphQL, podsumowuje tę zaletę słowami:

> „Think in graphs, not endpoints"[14]

— to odwrócenie tradycyjnego podejścia REST, gdzie klient nie ma kontroli nad formatem i objętością zwracanych danych.

W podejściu REST zastosowano zapytanie do endpointu `/categories/:id/products`, które zwraca wszystkie informacje o każdym produkcie w danej kategorii. Natomiast w GraphQL wykonano zapytanie ograniczające odpowiedź wyłącznie do pola `name`:

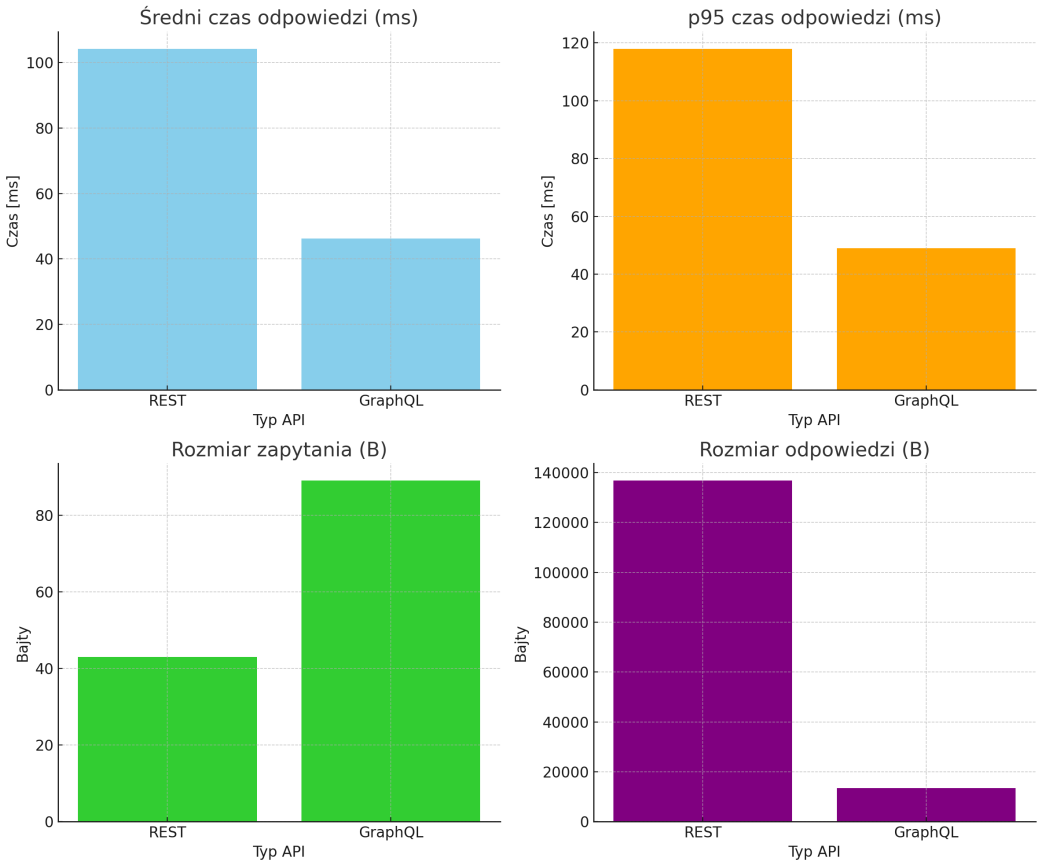
```
query($productId: Int!) {
  product(id: $productId) {
    name
  }
}
```

Pozwoliło to precyzyjnie dopasować odpowiedź do potrzeb klienta i ograniczyć jej rozmiar. Warto zaznaczyć, że w architekturze REST możliwe jest przygotowanie osobnego endpointu zwracającego jedynie potrzebne dane, na przykład same nazwy produktów. W praktyce jednak często rezygnuje się z tego rozwiązania na rzecz wykorzystania istniejących, bardziej ogólnych endpointów, szczególnie gdy różnice w czasie odpowiedzi są niewielkie w kontekście nowoczesnych systemów i gdy struktura danych po stronie klienta ulega częstym zmianom.

Celem testu było porównanie rozmiarów odpowiedzi oraz czasów odpowiedzi dla obu podejść w sytuacji, gdy wymagane dane stanowią jedynie niewielką część całej struktury zasobu.

Metryka	REST	GraphQL
Liczba zapytań	300	300
Średni czas odpowiedzi [ms]	104.2	46.2
Minimalny czas odpowiedzi [ms]	97	35
Maksymalny czas odpowiedzi [ms]	194	625
p50 czas odpowiedzi [ms]	102.5	36.2
p95 czas odpowiedzi [ms]	117.9	48.9
p99 czas odpowiedzi [ms]	179.5	407.5
Średnia długość sesji [ms]	105.9	48.1
Rozmiar odpowiedzi [B]	136800	13500
Rozmiar jednego zapytania [B]	43	89

Tabela 5.6: Porównanie wyników testu 4 - Pobranie części danych o produkcie (Overfetching) - tabela



Rysunek 5.12: Porównanie wyników testu 4 - Pobranie części danych o produkcie (Overfetching) - wykresy

Wyniki przedstawione w Tabeli 5.6 oraz na wykresie z Rysunku 5.12 ilustrują wpływ problemu overfetchingu na działanie obu podejść. Test został przeprowadzony na przykładzie jednej kategorii zawierającej trzy produkty. Po stronie REST zastosowano standardowy endpoint `/categories/:id/products`, który zwraca całą strukturę każdego produktu, niezależnie od faktycznego zapotrzebowania po stronie klienta. W zapytaniu GraphQL natomiast ograniczono odpowiedź wyłącznie do pola `name`, odpowiadającego rzeczywistym potrzebom klienta.

Efekty overfetchingu są widoczne przede wszystkim w znacząco większym rozmiarze odpowiedzi REST (136800 B wobec 13500 B w GraphQL) oraz w wyższym średnim czasie odpowiedzi (104.2 ms w REST kontra 46.2 ms w GraphQL). Co istotne, różnice te będą rosły proporcjonalnie do liczby produktów w danej kategorii, ponieważ REST przy każdym zapytaniu pobiera pełne dane obiektów, natomiast GraphQL pozwala dynamicznie ograniczyć zakres żądanych pól, co znacząco poprawia efektywność komunikacji między klientem, a serwerem.

Rozdział 6

Pozostałe kluczowe różnice REST i GraphQL

Niniejszy rozdział stanowi podsumowanie kluczowych różnic między podejściami REST (Express) a GraphQL (Apollo). Omówione poniżej aspekty uzupełniają wcześniejsze testy wydajnościowe, dostarczając kompleksowego obrazu porównania obu technologii.

6.1 Uwierzytelnienie i Autoryzacja

W podejściu REST uwierzytelnienie jest realizowane za pomocą middleware, który sprawdza np. token JWT dołączany do nagłówka każdego żądania HTTP. Middleware może również kontrolować dostęp do specyficznych zasobów, jak w przypadku endpointu `/orders/:userId` aplikacji e-commerce, który zapewnia, że użytkownik może pobrać jedynie swoje zamówienia. Listing 6.1 przestawia przykład zastosowania middleware, który realizuje takie działanie.

```
1  const authenticate = (req, res, next) => {
2    const token = req.headers.authorization.split(" ")[1];
3    const decoded = jwt.verify(token, "secret");
4    req.userId = decoded.userId;
5    req.isAdmin = decoded.role === "admin";
6    next();
7  };
8
9  app.get("/orders/:userId", authenticate, (req, res) => {
10       if (req.userId !== req.params.userId && !req.isAdmin) {
11         return res.status(403).json({ error: "Forbidden" });
12       }
13       // return user orders ...
14 });
```

Listing 6.1: Middleware `authenticate` służący do uwierzytelniania użytkownika i przekazywania `userId` z tokena JWT w architekturze REST

W tym momencie endpoint wie już, że użytkownik wykonujący zapytanie to użytkownik o identyfikatorze `userId` i może zwrócić jego zamówienia. Middleware może również sprawdzić, czy użytkownik posiada rolę zwiększającą jego uprawnienia, jak na przykład rola administratora (`admin`), umożliwiając wówczas pobranie zamówień wszystkich użytkowników.

GraphQL wykorzystuje jeden punkt końcowy i obsługę autoryzacji wewnątrz resolverów, które analizują kontekst zapytania zawierający dane o zalogowanym użytkowniku. Przykładowo, użytkownik trafia do kontekstu za pomocą specjalnej funkcji `context`, która wykonuje parsowanie tokenu JWT i zwraca obiekt użytkownika. Listing 6.2 przedstawia przykład takiej implementacji oraz resolvera wykorzystującego ten kontekst.

```
const context = async ({ req }) => {
  const token = req.headers.authorization?.split(" ")[1];
  if (token) {
    const decoded = jwt.verify(token, "secret");
    return { user: decoded };
  }
  return { user: null };
};

const resolvers = {
  Query: {
    orders: (parent, args, context) => {
      if (!context.user) throw new Error("Not authenticated");
      if (
        args.userId &&
        context.user.role !== "admin" &&
        context.user.userId !== args.userId
      ) {
        throw new Error("Forbidden");
      }
      // return user orders ...
    },
  },
};
```

Listing 6.2: Ładowanie danych użytkownika z tokena i umieszczanie ich w kontekście zapytania GraphQL (resolver)

Funkcja `context` przetwarza token JWT z nagłówka żądania HTTP i umieszcza informacje o użytkowniku w obiekcie kontekstu. Resolver następnie wykorzystuje te dane do sprawdzenia autoryzacji, umożliwiając administratorom dostęp do wszystkich zamówień, a zwykłym użytkownikom tylko do ich własnych.

6.2 Obsługa błędów

REST opiera obsługę błędów na standardowych kodach HTTP, np. `401 Unauthorized`, `404 Not Found` lub `500 Internal Server Error`. Kody te przekazują podstawowe informacje o przyczynie błędu, natomiast szczegóły mogą być dodatkowo dostarczane w formacie JSON. Takie podejście jest proste i powszechne, lecz nie pozwala na przekazywanie częściowych danych, jeśli wystąpi błąd.

W przypadku GraphQL błędy są zwracane w specjalnym polu `errors` odpowiedzi, natomiast status odpowiedzi HTTP zwykle pozostaje `200 OK`. Pozwala to na częściowe sukcesy zapytań, gdzie klient otrzymuje zarówno dane, jak i informacje o błędach. To podejście jest szczególnie przydatne w sytuacjach, gdy jedno zapytanie dotyczy wielu pól, a tylko niektóre generują błędy, co nie jest możliwe w standardowym REST.

6.3 Rozwój i dokumentacja API

W REST utrzymanie dokumentacji API (np. Swagger/OpenAPI) wymaga regularnej aktualizacji przy każdej zmianie w API, co często prowadzi do rozbieżności między dokumentacją, a stanem faktycznym API. Dodatkowo, rozwój REST wymaga tworzenia nowych endpointów lub wersjonowania API przy istotnych zmianach struktury danych, co zwiększa złożoność systemu.

W GraphQL dokumentacja API jest generowana automatycznie za pomocą introspekcji schematu. Każda zmiana wprowadzona do schematu jest od razu widoczna w dokumentacji, co znacząco ułatwia utrzymanie spójności między implementacją a opisem API. Ponadto, GraphQL promuje ewolucję schematu bez potrzeby wersjonowania, co ułatwia równoległy rozwój backendu i frontendu oraz obniża koszty utrzymania.

6.4 Podsumowanie

Oba podejścia, choć odmienne, zapewniają skuteczne mechanizmy rozwiązywania wspomnianych wyżej zagadnień. REST zapewnia prostotę, przejrzystość i kompatybilność z szeroko stosowanymi standardami HTTP, co czyni go szczególnie odpowiednim do stabilnych, jasno zdefiniowanych aplikacji. GraphQL natomiast oferuje wyższą elastyczność, precyzyjniejsze zarządzanie danymi, efektywną obsługę błędów oraz automatycznie generowaną dokumentację, co jest niezwykle korzystne w dynamicznych projektach o szybko zmieniających się wymaganiach.

Rozdział 7

Wnioski

W pracy dokonano porównania dwóch współczesnych podejść do budowy interfejsów API: REST oraz GraphQL. Głównym celem było zbadanie różnic w sposobie przetwarzania danych, wydajności, elastyczności i ergonomii pracy z tymi technologiami. Analizie poddano cztery scenariusze testowe, które odwzorowywały typowe przypadki użycia w aplikacjach webowych typu e-commerce:

1. Pobranie wszystkich danych o produkcie,
2. Pobranie danych o kategorii wraz z listą przypisanych produktów (underfetching),
3. Pobranie listy użytkowników wraz z ich zamówieniami (problem N+1),
4. Pobranie częściowych danych o produkcie (overfetching).

Wnioski wynikające z testów wydajnościowych są jednoznaczne w odniesieniu do pierwszego przypadku: klasyczny REST, dzięki swojej prostocie i bezpośredniemu odwzorowaniu zasobów HTTP, okazał się szybszy przy pobieraniu w całości pojedynczego zasobu (np. produktu). Dla takich zastosowań, gdzie klient wymaga wszystkich danych i struktura odpowiedzi nie ulega częstej zmianie, REST może być efektywniejszym wyborem zarówno pod względem wydajnościowym, jak i deweloperskim.

Jednak pozostałe trzy scenariusze zostały skonstruowane tak, aby pokazać ograniczenia podejścia REST, w szczególności gdy endpointy muszą obsługiwać bardziej złożone zależności danych, a potrzeby klienta ulegają częstym zmianom. Przykład problemu underfetchingu pokazał, że REST wymaga w takiej sytuacji osobnego endpointu lub sekwencji zapytań, natomiast w GraphQL wystarczy zdefiniować strukturę zapytania. Podobnie w przypadku overfetchingu – REST zmusza do korzystania z ogólnych endpointów, które zwracają nadmiarowe dane, podczas gdy GraphQL pozwala na precyzyjne wskazanie potrzebnych pól.

W scenariuszu N+1, w którym należało pobrać użytkowników z ich zamówieniami, początkowa implementacja GraphQL była mniej wydajna z powodu wielokrotnych zapytań do bazy. Jednak zastosowanie optymalizacji z wykorzystaniem mechanizmu DataLoader (agregacja zapytań) znacząco poprawiło czas odpowiedzi, co pozwala stwierdzić, że po optymalizacji wyniki dla obu podejść były porównywalne.

Podobne porównanie REST i GraphQL zostało przeprowadzone w pracy Mateusza Mikuły i Mariusza Dzieńkowskiego - ”Porównanie wydajności technologii webowych REST i GraphQL”, gdzie również wskazano na przewagę REST w podstawowych operacjach oraz na elastyczność GraphQL w bardziej złożonych scenariuszach [15].

Wybór technologii zależny od potrzeb Podsumowując, **REST oferuje lepszą wydajność i prostotę implementacyjną dla prostych, jednozasobowych zapytań**, podczas gdy **GraphQL daje większą elastyczność i możliwość ograniczenia transferu danych w scenariuszach złożonych relacji lub szybko zmieniających się potrzeb klienta**. Optymalizacja po stronie serwera (jak DataLoader) pozwala znacznie zredukować różnice wydajnościowe w bardziej wymagających przypadkach, co czyni GraphQL realną alternatywą – zwłaszcza gdy projekt wymaga elastycznego definiowania struktury danych po stronie klienta.

Doświadczenia w pracy nad projektem Z perspektywy autora niniejszej pracy, implementacja backendu w architekturze REST przebiegła szybciej i była bardziej intuicyjna. Wynika to przede wszystkim z wcześniejszego doświadczenia zawodowego i projektowego z REST, podczas gdy praca z GraphQL była pierwszym kontaktem z tą technologią. Warto jednak zauważyć, że według badania przeprowadzonego przez Gleison'a Brito i Marco Tulio Valente [2], uczestnicy danego eksperymentu implementowali zadane zapytania średnio szybciej w GraphQL niż w REST. Pokazuje to, że subiektywna łatwość implementacji zależy przede wszystkim od doświadczenia i preferencji osoby programującej.
Zatem decyzja o wyborze REST lub GraphQL powinna wynikać **nie z wyników testów wydajnościowych, a głównie z analizy charakterystyki aplikacji, jej wymagań projektowych oraz kompetencji zespołu deweloperskiego.**

Bibliografia

[1] Joanna Wallace Aagam Vadecha. *How to solve the GraphQL n+1 problem*. URL: https://hygraph.com/blog/graphql-n-1-problem (term. wiz. 20. 06. 2025).

[2] Gleison Brito i Marco Tulio Valente. *REST vs GraphQL: A Controlled Experiment*. arXiv preprint. arXiv:2003.04761. 2020. URL: https://arxiv.org/abs/2003.04761 (term. wiz. 20. 06. 2025).

[3] Lee Byron. *GraphQL Specification*. 2015. URL: https://spec.graphql.org/October2021/ (term. wiz. 20. 06. 2025).

[4] E. F. Codd. „A Relational Model of Data for Large Shared Data Banks". W: *Communications of the ACM* 13.6 (1970), s. 377–387.

[5] Thomas Connolly i Carolyn Begg. *Database Systems: A Practical Approach to Design, Implementation, and Management*. 6th. Pearson, 2014.

[6] *Database Normalization*. URL: https://en.wikipedia.org/wiki/Database_normalization (term. wiz. 20. 06. 2025).

[7] Ramez Elmasri i Shamkant B. Navathe. *Fundamentals of Database Systems*. 7th. Pearson, 2016.

[8] Roy T. Fielding. „Architectural Styles and the Design of Network-based Software Architectures". PhD thesis. University of California, Irvine, 2000.

[9] JJ Geewax. *API Design Patterns*. Manning Publications, 2021.

[10] How to GraphQL. *GraphQL is the better REST*. URL: https://www.howtographql.com/basics/1-graphql-is-the-better-rest (term. wiz. 20. 06. 2025).

[11] Stack Exchange Inc. *Stack overflow survey - Most popular technologies - Databases*. URL: https://survey.stackoverflow.co/2024/technology#most-popular-technologies (term. wiz. 20. 06. 2025).

[12] S. Ruby L. Richardson M. Amundsen. *RESTful Web APIs*. O'Reilly Media, 2013.

[13] Arnaud Lauret. *Designing Web APIs: Building APIs That Developers Love*. Manning Publications, 2019.

[14] GraphQL/Facebook Lee Byron. *Lessons from 4 Years of GraphQL*. 2016. URL: https://www.youtube.com/watch?v=zVNrqo9XGOs&ab_channel=ApolloGraphQL (term. wiz. 20. 06. 2025).

[15] Mariusz Dzieńkowski Mateusz Mikuła*. „Comparison of REST and GraphQL web technology performance”. W: *Journal of Computer Sciences Institute* (2020). URL: https://ph.pollub.pl/index.php/jcsi/article/view/2077/1986 (term. wiz. 20. 06. 2025).

[16] Falco Nogatz i Dietmar Seipel. *Implementing GraphQL as a Query Language for Deductive Databases in SWI-Prolog Using DCGs, Quasi Quotations, and Dicts*. arXiv preprint. arXiv:1701.00626. 2017. URL: https://arxiv.org/abs/1701.00626 (term. wiz. 20. 06. 2025).

[17] Inc. Prisma Data. *TypeScript ORM with zero-cost type-safety for your database*. URL: https://www.prisma.io/typescript (term. wiz. 20. 06. 2025).

[18] Ramesh Reddy. *API Design for C++*. Elsevier Science & Technology, 2011.

[19] Leonard Richardson i Sam Ruby. *RESTful Web Services*. O'Reilly Media, 2007.

[20] Carl Rippon. *Learn TypeScript: A Free Interactive Course for JavaScript Developers - What is TypeScript?* URL: https://learntypescript.dev/01/l1-what-is-ts (term. wiz. 20. 06. 2025).

[21] Marcin Rzepecki. *REST vs GraphQL - porównanie*. 2021. URL: https://bulldogjob.pl/readme/rest-vs-graphql-porownanie (term. wiz. 20. 06. 2025).

[22] Abraham Silberschatz, Henry F. Korth i S. Sudarshan. *Database System Concepts*. 7th. McGraw-Hill, 2019.

[23] B. Ali Syed. *TypeScript Deep Dive*. Samurai Media Limited, 2017.

[24] Muhammad Usman. *Understanding the N+1 Problem in GraphQL and How to Solve It*. URL: https://engrmuhammadusman108.medium.com/understanding-the-n-1-problem-in-graphql-and-how-to-solve-it-1799e928066a (term. wiz. 20. 06. 2025).

Spis rysunków

3.1 Schemat bazy danych aplikacji (model relacyjny sklepu internetowego) . . 9

5.1 Porównanie wyników testu 1 - Pobranie pełnego zestawu informacji o produkcie o danym id - wykresy 20
5.2 Przykładowy układ aplikacji sklepu internetowego, przedstawiający problem underfetching'u 21
5.3 Porównanie wyników testu 2 - Pobranie danych o kategorii oraz listy produktów (Underfetching) - wykresy 22
5.4 Porównanie wyników testu 3 - Pobranie listy użytkowników z ich zamówieniami - wykresy 24
5.5 Zrzut ekranu fragmentu logów, przedstawiający zapytania do bazy danych po uruchomieniu testu 3 w implementacji REST 25
5.6 Zrzut ekranu fragmentu logów, przedstawiający zapytania do bazy danych po uruchomieniu testu 3 w implementacji GraphQL 25
5.7 Porównanie wyników testu 3 - Pobranie listy użytkowników z ich zamówieniami (po dziesięciokrotnym zwiększeniu liczby użytkowników i zamówień w bazie danych) - wykresy 26
5.8 Średni czas odpowiedzi, w zależności od liczby użytkowników w REST i GraphQL - test 3 27
5.9 DataLoader - Mechanizm grupujący identyfikatory w jedną kolekcję, w celu wykonania jednego zbiorczego zapytania do bazy danych, eliminując potrzebę wysyłania wielu pojedynczych zapytań[1] 28
5.10 Porównanie wyników testu 3 - Pobranie listy użytkowników z ich zamówieniami (po dziesięciokrotnym zwiększeniu liczby użytkowników i zamówień w bazie danych) i implementacji mechanizmu DataLoader - wykresy 30
5.11 Średni czas odpowiedzi, w zależności od liczby użytkowników w REST i GraphQL po zaimplementowaniu mechanizmem DataLoader - test 3 31
5.12 Porównanie wyników testu 4 - Pobranie części danych o produkcie (Overfetching) - wykresy 33

Spis tabel

5.1 Porównanie wyników testu 1 - Pobranie pełnego zestawu informacji o produkcie o danym id - tabela 19
5.2 Porównanie wyników testu 2 - Pobranie danych o kategorii oraz listy produktów (Underfetching) - tabela 22
5.3 Porównanie wyników testu 3 - Pobranie listy użytkowników z ich zamówieniami - tabela 24
5.4 Porównanie wyników testu 3 - Pobranie listy użytkowników z ich zamówieniami (po dziesięciokrotnym zwiększeniu liczby użytkowników i zamówień w bazie danych) - tabela 26
5.5 Porównanie wyników testu 3 - Pobranie listy użytkowników z ich zamówieniami (po dziesięciokrotnym zwiększeniu liczby użytkowników i zamówień w bazie danych) i implementacji mechanizmu DataLoader - tabela 29
5.6 Porównanie wyników testu 4 - Pobranie części danych o produkcie (Overfetching) - tabela 33

Spis listingów

3.1 Plik schema.prisma ze schematem bazy danych `schema.prisma` 10
4.1 Definicja encji Order 14
4.2 Definicja resolvera OrderItem 15
4.3 Router products.ts 16
5.1 DataLoader, grupujący zapytania dla klasy Zamówienie 29
6.1 Middleware `authenticate` służący do uwierzytelniania użytkownika i przekazywania `userId` z tokena JWT w architekturze REST 35
6.2 Ładowanie danych użytkownika z tokena i umieszczanie ich w kontekście zapytania GraphQL (resolver) 36