POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Mariusz Kudłacz
Nr albumu s32223

Porównanie technologii mapowania obiektowo-relacyjnego dla platformy Node.js

Praca magisterska
napisana pod kierunkiem
Dr inż. Mariusza Trzaski

Warszawa, Luty, 2026

Streszczenie

Niniejsza praca magisterska dotyczy porównania technologii mapowania relacyjno-obiektowego w aplikacjach tworzonych w środowisku Node.js. Autor koncentruje się na analizie wpływu wybranych technologii ORM na wydajność aplikacji korzystających z relacyjnych baz danych. Analizie poddano technologie Prisma, Sequelize, TypeORM, MikroORM oraz Objection.js, uwzględniając również różnice wynikające z zastosowania wzorców projektowych Active Record i Data Mapper.

W ramach pracy magisterskiej powstała aplikacja testowa, umożliwiają niezależne testowanie każdej technologii ORM. Testy przeprowadzono z wykorzystaniem baz danych PostgreSQL oraz MySQL, dla różnych rozmiarów zbiorów danych. Badania obejmowały operacje odczytu, zapisu, zapytania agregujące oraz złożone operacje transakcyjne. Przeanalizowano czas wykonania operacji, zużycie pamięci operacyjnej oraz obciążenie procesora.

Uzyskane wyniki wykazały, że różnice pomiędzy technologiami ORM stają się istotne wraz ze wzrostem liczby rekordów i złożoności operacji. W końcowej części pracy przedstawiono interpretację wyników w kontekście zastosowań produkcyjnych oraz wskazano czynniki, które należy brać pod uwagę przy wyborze technologii ORM.

Słowa kluczowe: mapowanie obiektowo-relacyjne, ORM, Prisma, Sequelize, Objection.js, TypeORM, MikroORM, Node.js

Spis treści

1. Wstęp

Współczesne czasy charakteryzują się tym, że gromadzenie różnorodnych informacji jest uznawane za standardową praktykę, nie tylko w biznesie. Ludzie coraz częściej sami dla siebie zbierają dane dotyczące różnych aspektów swojego życia, takich jak spożywane kalorie, długość snu, codzienną aktywność fizyczną, czy nawet nastrój i samopoczucie. Wzrastająca ilość danych, które są zbierane, przetwarzane i przechowywane w systemach informatycznych wymaga nie tylko odpowiednich narzędzi umożliwiających te czynności, ale również odpowiedniego podejścia do zarządzania danymi.

Znaczna większość współczesnych systemów informatycznych, funkcjonujących w różnorodnych sektorach gospodarki i administracji publicznej, wykorzystuje zaawansowane relacyjne bazy danych do przechowywania, organizacji i kompleksowego zarządzania informacjami biznesowymi, co umożliwia przeprowadzanie efektywnych operacji analitycznych i transakcyjnych na bardzo dużych zbiorach strukturalnych danych.

Relacyjne systemy bazodanowe zapewniają uporządkowany sposób strukturyzacji danych w formie wzajemnie połączonych tabel. Każda tabela odzwierciedla określony element lub jednostkę, a informacje są zorganizowane w wierszach i kolumnach, co w znacznym stopniu usprawnia ich administrowanie i przeszukiwanie.

Tabela 1 Najpopularniejsze systemy zarządzania bazą danych – grudzień 2025. Źródło: [1]

Ranking	**Nazwa SZBD**	**Główny model bazy danych**
1.	Oracle	Relacyjna
2.	MySQL	Relacyjna
3.	Microsoft SQL Server	Relacyjna
4.	PostgreSQL	Relacyjna
5.	MongoDB	Dokumentowy

Tabela 1 przedstawia dane pochodzące z comiesięcznego rankingu popularności systemów zarządzania bazami danych opracowanego przez firmę Redgate Software [1], które wyraźnie pokazują dominację systemów bazujących na relacyjnym modelu. Zajmują one pierwsze miejsca pod względem popularności, a ich udział w rynku nieprzerwanie rośnie. Wynika to z faktu, że podejście oparte na modelu relacyjnym jest uznawane za najbardziej uniwersalne i najbardziej rozwinięte, a także umożliwia łatwe zapewnienie zgodności z różnorodnymi standardami i regulacjami.

Większość współczesnych aplikacji i systemów tworzona jest w sposób obiektowy. W tym paradygmacie programowania dane są reprezentowane w formie obiektów zawierających zarówno atrybuty opisujące ich stan, jak i metody definiujące możliwe operacje. Takie podejście umożliwia programistom intuicyjne modelowanie rzeczywistych encji biznesowych w kodzie aplikacji, gdzie na przykład klient, zamówienie czy produkt są reprezentowane jako konkretne obiekty z właściwymi im właściwościami i metodami. Jednakże relacyjne bazy danych operują na zupełnie innym modelu danych, wykorzystując tabele, wiersze i kolumny, co prowadzi do powstania zjawiska niezgodności impedancji między modelem obiektowym aplikacji a tabelaryczną strukturą bazy danych [2].

Jednym z możliwych rozwiązań tego problemu, jest wykorzystanie technologii mapowania obiektowo-relacyjnego (*Object-Relational Mapping – ORM*), która umożliwia programistom pracę w paradygmacie programowania obiektowego, a jednocześnie zapewnia odpowiednią warstwę abstrakcji odpowiedzialną za odwzorowanie tabel bazodanowych na klasy i obiekty w kodzie źródłowym systemu.

Na współczesnym rynku technologicznym dostępna jest znacząca liczba różnorodnych rozwiązań ORM, które zostały opracowane w celu usprawnienia pracy z bazami danych w różnych środowiskach programistycznych. Niniejsza praca koncentruje się szczególnie na technologiach mapowania obiektowo-relacyjnego dedykowanych środowisku Node.js, które według badania Stack Overflow Developer Survey stanowi od lat jedno z najpopularniejszych i najdynamiczniej rozwijających się platform do tworzenia aplikacji internetowych [3].

Dostępne rozwiązania ORM dla Node.js charakteryzują się znaczącymi różnicami w zakresie architektury, sposobu implementacji, oferowanej funkcjonalności, wydajności oraz podejścia do zarządzania połączeniami z bazami danych. Te różnice wpływają bezpośrednio na sposób projektowania aplikacji, łatwość utrzymania kodu oraz ogólną wydajność systemu, co czyni wybór odpowiedniej technologii ORM kluczowym elementem procesu projektowania aplikacji opartych na Node.js.

1.1 Motywacja podjęcia tematu

Motywacją podjęcia tego tematu jest zrozumienie, które technologie ORM najlepiej sprawdzają się w kontekście aplikacji Node.js oraz jakie są między nimi różnice i z czego wynikają. Współczesny rynek oferuje programistom szeroki wybór rozwiązań ORM, jednak niewiele jest kompleksowych opracowań, które w sposób systematyczny porównywałyby ich wydajność, łatwość implementacji oraz funkcjonalność. Istniejące publikacje często koncentrują się na pojedynczych technologiach lub przedstawiają powierzchowne porównania, które nie dostarczają jasnych odpowiedzi na pytania dotyczące praktycznego zastosowania tych narzędzi w rzeczywistych projektach.

Niniejsza praca ma na celu wypełnienie tej luki poprzez dostarczenie praktycznej wiedzy, która pomoże programistom w podjęciu świadomej decyzji o wyborze odpowiedniej technologii ORM do swoich projektów. Szczególnie istotne jest to w kontekście rosnącej popularności Node.js jako platformy do tworzenia aplikacji internetowych, gdzie właściwy wybór technologii ORM może mieć znaczący wpływ na wydajność, skalowalność i łatwość utrzymania całego systemu.

1.2 Cel i zakres pracy

Celem pracy jest przeprowadzenie analizy porównawczej wybranych technologii mapowania obiektowo-relacyjnego dla platformy Node.js. Zakres obejmuje analizę wydajności poszczególnych rozwiązań, różnice w sposobach implementacji oraz identyfikację i ocenę potencjalnych problemów i ograniczeń związanych z wyborem konkretnego ORM-a w kontekście projektowania i rozwoju skalowalnych aplikacji webowych.

Analizie zostanie poddanych 5 najpopularniejszych i najszerzej adoptowanych przez społeczność Node.js ORM-ów: Prisma, Sequelize, Objection.js, MikroORM oraz TypeORM. Wymienione technologie będą testowane w ramach aplikacji testowej, wykorzystującej ten sam zbiór danych oraz jednolite scenariusze użycia. W celu uzyskania pełnego obrazu możliwości poszczególnych rozwiązań, w porównaniu zostaną dodatkowo uwzględnione różne systemy zarządzania bazami danych, co pozwoli na ocenę elastyczności i uniwersalności badanych technologii.

1.3 Rozwiązania przyjęte w pracy

Aplikacja testowa została oparta o środowisko Node.js, wykorzystany został framework Express.js, wraz z językiem TypeScript. Jako system zarządzania pakietami wykorzystano npm. Użyto PostgreSQL oraz MySQL jako relacyjne systemy zarządzania bazą danych.

1.4 Rezultat pracy

Rezultatem pracy jest określenie mocnych i słabych stron każdej z porównywanych technologii ORM dla Node.js oraz wskazanie potencjalnych problemów i ograniczeń związanych z implementacją i funkcjonowaniem każdej technologii.

Efektem pracy jest także przygotowanie aplikacji testowej, wykorzystującej wszystkie porównywane technologie.

1.5 Organizacja pracy

Praca rozpoczyna się od wprowadzenia w temat mechanizmów ORM, przedstawienia kluczowych pojęć i wzorców. Następnie zaprezentowane zostały istniejące technologie dla Node.js, wraz z przeglądem i omówieniem dostępnych opracowań, które je porównują.

W rozdziale 3 opisano charakterystykę każdej z porównywanych technologii, skupiając się na dostępnych funkcjach, stabilności i mechanizmach działania. W rozdziale 4 zostały przedstawione założenia i scenariusze badań porównujących analizowane technologie.

W rozdziałach 5 i 6 zaprezentowano projekt aplikacji testowej, wraz z przedstawieniem wykorzystanych narzędzi i technologii, a następnie opisano proces jej implementacji.

Rozdział 7 i 8 opisują przeprowadzone testy wydajności w aplikacji testowej badanych wraz z zestawieniem otrzymanych wyników.

W ostatnim rozdziale opisano krótkie podsumowanie całej pracy oraz wskazano możliwe kierunki dalszych badań.

2. ORM w aplikacjach internetowych

Współczesne aplikacje internetowe często operują na dużych zbiorach danych, które zwykle przechowywane są w relacyjnych bazach danych. W celu efektywnej interakcji z danymi, programiści muszą połączyć dwa odmienne modele operowania na danych – model relacyjny bazy danych oraz model obiektowy języku programowania. Różnice pomiędzy modelami prowadzą do wystąpienia zjawiska niezgodności impedancji.

Definicja 1

Niezgodność impedancji [4] - oznacza zespół niezgodności powstający w wyniku połączenia języka wysokiego poziomu (w przypadku baz danych - języka SQL) z językiem programowania (w przypadku środowiska Node.js - języka JavaScript). Niezgodności dotyczą przede wszystkim:

- **składni** – w kodzie danego programu są używane dwa różne style językowe.
- **systemu typów** – język zapytań wykorzystuje typy określone w schemacie bazy danych, natomiast język programowania posiada swój odmienny zestaw typów. Przykładem jest typ relacja, który nie występuje w języku programowania.
- **przestrzeni nazw** – język zapytań i język programowania dysponują własnymi przestrzeniami nazw, które mogą zawierać identyczne nazwy o odmiennych znaczeniach.
- **schematów iteracyjnych** – w językach zapytań iteracje są wbudowane w semantykę operatorów jak selekcja, projekcja i złączenie. W językach programowania iteracje muszą być tworzone bezpośrednio przez instrukcje for, while, repeat. Przetwarzanie rezultatów zapytań w języku programowania wymaga mechanizmów takich jak kursory i iteratory.

Konsekwencją niezgodności impedancji nie są jedynie defekty estetyczne. Niezgodność impedancji powoduje konieczność powstania dodatkowej warstwy oprogramowania pośredniczącego między językiem zapytań a językiem programowania [4].

Mimo że w latach 90. przewidywano zastąpienie baz relacyjnych przez obiektowe bazy danych w celu rozwiązania problemu niezgodności impedancji, bazy relacyjne utrzymały dominację dzięki udziałowi w integracji systemów, standaryzacji SQL oraz podziałowi specjalizacji inżynierów na twórców oprogramowania i administratorów baz danych [5].

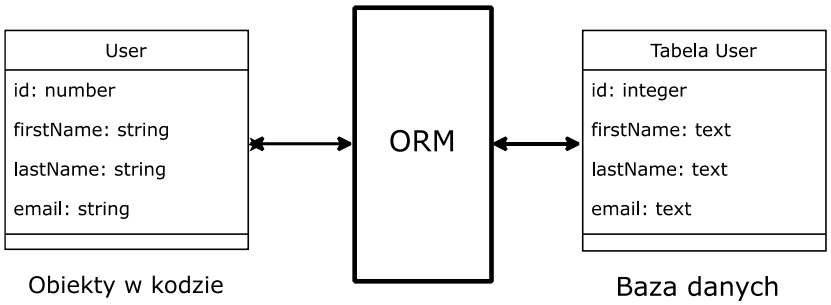
Problem niezgodności został złagodzony przez biblioteki mapujące na model obiektowy. Wykorzystuję one technologię *Object-Relational Mapping* (ORM), która stanowi warstwę abstrakcji pomiędzy językiem programowania a relacyjną bazą danych.

Definicja 2

Mapowanie obiektowo-relacyjne (*Object-Relational Mapping - ORM)* to technika programistyczna umożliwiająca odwzorowanie struktur relacyjnych w bazie danych na obiekty w języku programowania, pozwalająca na manipulowanie danymi bez konieczności pisania bezpośrednich zapytań SQL [6]. ORM pozwala programistom na operowanie na danych w sposób obiektowy, przy użyciu standardowych mechanizmów języków programowania, takich jak klasy, metody czy dziedziczenie. Dzięki temu kod aplikacji staje się bardziej przejrzysty i łatwiejszy w utrzymaniu.

Podstawowym mechanizmem ORM jest mapowanie encji na tabele bazy danych. Każda zdefiniowana klasa w języku obiektowym odpowiada bezpośrednio jednej konkretnej tabeli w bazie danych, a poszczególne atrybuty tej klasy są systematycznie mapowane na odpowiadające im kolumny w strukturze tej tabeli. Operacje CRUD (ang. *Create*, *Read*, *Update*, *Delete*), stanowiące fundament

manipulacji danymi, są realizowane za pomocą metod wysokopoziomowych, które ORM tłumaczy na odpowiednie zapytania SQL.



Rysunek 1 Schemat działania ORM. Źródło: opracowanie własne

Rysunek 1 pokazuje, jak ORM przekształca dane z bazy na obiekty w kodzie i na odwrót. Gdy program potrzebuje informacji z bazy zwraca się do frameworka. On tworzy zapytanie do bazy danych, wykonuje je i zwraca wynik jako obiekty gotowe do użycia. Kiedy program chce zapisać dane przekazuje obiekty do ORM. Ten zamienia je na odpowiedni format do zapisu w bazie i buduje zapytanie SQL do wstawienia lub aktualizacji danych. Powiązanie obiektów z tabelami bazy danych określa się zazwyczaj poprzez metadane. Można je ustalić w plikach konfiguracyjnych lub za pomocą adnotacji. ORM używa tych metadanych, aby wiedzieć jak tłumaczyć obiekty i ich właściwości na tabele i kolumny bazy danych.

Listing 1 Zapytanie z wykorzystaniem narzędzia ORM w JavaScript

```
1. const order = await prisma.order.findUnique({
2.     where: {
3.     id: 1,
4.   },
5. });
```

Listing 1 zawiera przykładowe zapytanie napisane z wykorzystaniem frameworka ORM Prisma. W tradycyjnym podejściu uzyskanie tego samego rezultatu wymagałoby użycia dedykowanego klienta do komunikacji z bazą danych, przygotowania i wykonania bezpośredniego zapytania w języku SQL, a następnie przetworzenia otrzymanych wyników (*record sets*) w celu ich mapowania na obiekty wykorzystywane w warstwie aplikacyjnej.

Istnieje kilka popularnych wzorców ORM [7]:

- ***Active Record*** – obiekt, który reprezentuje pojedynczy wiersz w tabeli lub widoku bazy danych, enkapsuluje dostęp do bazy oraz dodaje logikę domenową do tych danych.
- ***Data Mapper*** – oddziela model danych aplikacji od bazy danych i zapewnia sposób mapowania danych przechowywanych w bazie na obiekty w aplikacji oraz odwrotnie.
- ***Unit of Work*** – śledzi wszystkie zmiany dokonane na obiektach w aplikacji i zapewnia, że baza danych zostanie odpowiednio zaktualizowana po zatwierdzeniu transakcji.
- ***Table Data Gateway*** – mapuje pojedynczą tabelę na odpowiadającą jej klasę w aplikacji.

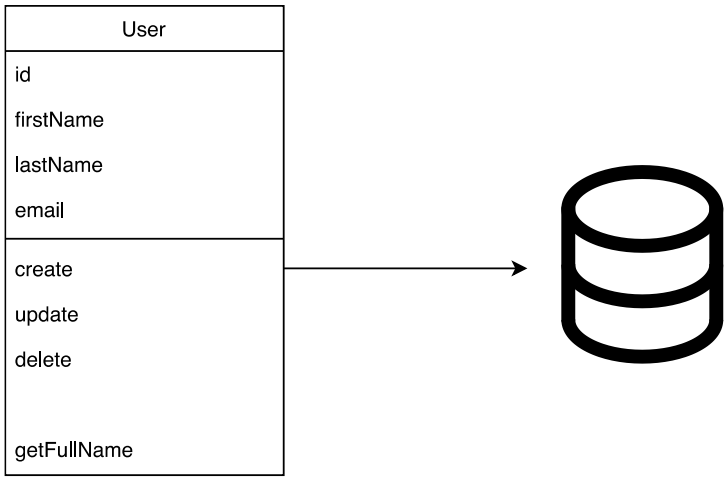
W kontekście ORM dla Node.js najczęściej wykorzystywane są wzorce *Active Record* oraz *Data Mapper* i na nich praca będzie się koncentrować.

2.1 Active Record

ORM-y *Active Record* odwzorowują klasy modeli na tabele w bazie danych, gdzie budowa obu form jest ściśle związana. Na przykład każda właściwość w klasie modelu będzie posiadała odpowiadającą jej kolumnę w tabeli bazy danych. Instancje klas modeli opakowują rekordy bazy danych i przechowują zarówno informacje, jak i mechanizmy dostępu do zarządzania zapisywaniem modyfikacji w bazie danych. Ponadto klasy modeli mogą obejmować logikę domenową charakterystyczną dla informacji zawartych w modelu.

Klasa modelu zazwyczaj posiada metody, które wykonują następujące działania [7]:

- tworzą instancję modelu z zapytania SQL,
- tworzą nową instancję do późniejszego wstawienia do tabeli,
- hermetyzują często wykorzystywane zapytania SQL i dostarczają obiekty *Active Record*,
- modyfikują bazę danych oraz wprowadzają do niej informacje z *Active Record*,
- odczytują i definiują atrybuty,
- realizują część logiki biznesowej.



Rysunek 2 Schemat działania wzorca *Active Record*. Źródło: opracowanie własne na podstawie [7]

W praktyce oznacza to, że obiekt *Active Record* posiada zdolność pobierania własnych danych z bazy, zapisywania zmian oraz usuwania siebie z repozytorium. Rysunek 2 przedstawia schemat klasy `User`. Instancja tej klasy odpowiada jednemu użytkownikowi w bazie, a metody tej klasy pozwalają na bezpośrednią pracę z bazą danych (np. `user.delete()`). Listing 2 prezentuje kod implementujący przedstawiony schemat klasy.

Listing 2 Definicja klasy User w modelu *Active Record* wraz z przykładem użycia

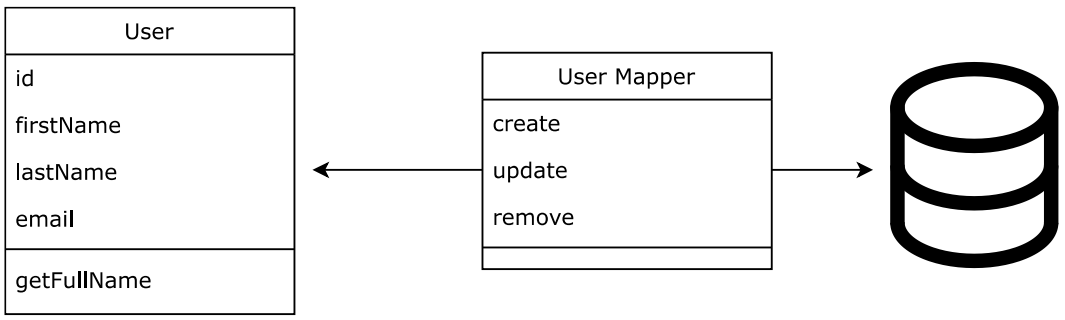
```
1. class User {
2.   id: number;
3.   firstName: string;
4.   lastName: string;
5.   email: string;
6.
7.   constructor(id: number, firstName: string, lastName: string, email: string) {
8.     this.id = id;
```

```
 9.      this.firstName = firstName;
10.      this.lastName = lastName;
11.      this.email = email;
12.    }
13.
14.    static find(id: number): User | null {
15.      // Implementacja wyszukiwania
16.      return null;
17.    }
18.
19.    getFullName() {
20.      return `${this.firstName} ${this.lastName}`;
21.    }
22.
23.    create(): void {
24.      // Implementacja tworzenia nowego użytkownika
25.    }
26.
27.    update(): void {
28.      // Implementacja aktualizacji istniejącego użytkownika
29.    }
30.
31.    remove(): void {
32.      // Implementacja usunięcia
33.    }
34. }
35.
36. // Przykład użycia
37. const user = new User(1, 'Jan', 'Kowalski', 'jan.kowalski@example.com');
38. user.create();
```

Active Record to dobry wybór dla logiki, która nie jest zbyt skomplikowana, takiej jak tworzenie, odczytywanie, aktualizacja i usuwanie danych [7]. Walidacje i operacje oparte na pojedynczych rekordach dobrze sprawdzają się w tej strukturze. Gdy logika biznesowa staje się złożona i potrzebne są bezpośrednie relacje między obiektami, kolekcje, dziedziczenie itp., nie mapują się one łatwo na *Active Record*, a dodawanie ich fragmentarycznie prowadzi do bałaganu.

2.2 Data Mapper

Wzorzec *Data Mapper* polega na całkowitym oddzieleniu logiki biznesowej aplikacji od sposobu przechowywania danych w bazie – prezentuje to Rysunek 3. Separacja następuje poprzez wprowadzenie dodatkowej warstwy - mappera. Jej głównym zadaniem jest przenoszenie danych pomiędzy tymi dwoma światami oraz ich wzajemna izolacja.



Rysunek 3 Schemat działania wzorca *Data Mapper*. Źródło: opracowanie własne na podstawie [7]

W przeciwieństwie do wzorca *Active Record*, *Data Mapper* całkowicie rozdziela reprezentację danych w pamięci aplikacji od ich postaci w bazie. Osiąga się to poprzez podział na dwa typy klas [8]:

- **klasy encji** – służą do reprezentowania danych w aplikacji i nie mają żadnej wiedzy o strukturze czy rodzaju bazy danych.
- **klasy mapperów** – odpowiadają zarówno za przekształcanie danych pomiędzy tymi dwoma reprezentacjami, jak i za generowanie zapytań SQL potrzebnych do pobierania oraz zapisywania danych w bazie.

Takie podejście zwiększa elastyczność i ułatwia rozwój oraz testowanie aplikacji, zwłaszcza w większych i bardziej złożonych projektach. Wynika to z faktu, że wzorzec ten pozwala ukryć szczegóły implementacji bazy danych, dzięki czemu programista nie myśli o modelu domenowym przez pryzmat sposobu przechowywania danych w bazie. Listing 3 prezentuje implementację tego podejścia w kodzie.

Listing 3 Definicja klasy *User* w modelu *Data Mapper* wraz z przykładem użycia

```
// Encja użytkownika - dane i logika biznesowa
class User {
  id: number;
  firstName: string;
  lastName: string;
  email: string;

  constructor(id: number, firstName: string, lastName: string, email: string) {
    this.id = id;
    this.firstName = firstName;
    this.lastName = lastName;
    this.email = email;
  }

  getFullName(): string {
    return `${this.firstName} ${this.lastName}`;
  }
}

// Data Mapper - operacje na danych
class UserMapper {
  static find(id: number): User | null {
    // Implementacja wyszukiwania
  }

  static create(user: User): void {
    // Implementacja tworzenia nowego użytkownika w bazie
  }

  static update(user: User): void {
    // Implementacja aktualizacji użytkownika w bazie
  }

  static remove(user: User): void {
    // Implementacja usunięcia użytkownika z bazy
  }
}

// Przykład użycia
const user = new User(1, 'Jan', 'Kowalski', 'jan.kowalski@example.com');
UserMapper.create(user);
```

Zastosowanie wzorca *Data Mapper* jest uzasadnione przede wszystkim w sytuacjach, gdy wymagane jest niezależne rozwijanie modelu obiektowego aplikacji oraz schematu bazy danych [7]. Rozwiązanie to znajduje szczególne zastosowanie w projektach, w których wykorzystywany jest rozbudowany model domenowy, a projektowanie, implementacja oraz testowanie logiki biznesowej

powinny być możliwe bez konieczności uwzględniania szczegółów struktury bazy danych. W sytuacjach, gdy logika biznesowa jest stosunkowo prosta, a baza danych znajduje się pod pełną kontrolą zespołu programistycznego, bardziej odpowiednim rozwiązaniem może być zastosowanie wzorca *Active Record*, w którym obiekty samodzielnie realizują operacje na bazie danych.

Wadą wzorca *Data Mapper* jest zwiększenie złożoności architektury aplikacji poprzez wprowadzenie dodatkowej warstwy abstrakcji, co może utrudniać zrozumienie całościowego działania systemu. Ponadto jego zastosowanie ogranicza możliwość pełnej kontroli nad szczegółami pobierania i zapisywania danych, ponieważ operacje te są ukryte za warstwą mapującą, co może również generować problemy wydajnościowe.

2.3 Unit of Work

Wzorzec *Unit of Work* polega na śledzeniu wszystkich zmian dokonanych na obiektach w aplikacji i zapewnieniu, że odpowiednie modyfikacje zostaną zapisane w bazie danych po zatwierdzeniu transakcji. Klasa *Unit of Work* pełni rolę centralnego repozytorium zmian i odpowiada za ich zapisanie w bazie, gdy transakcja zostanie zakończona. Dzięki temu wzorcowi zarządzanie trwałością obiektów jest bardziej spójne i wydajne, ponieważ wszystkie zmiany są wysyłane do bazy w jednej operacji, co ogranicza liczbę połączeń z bazą i zwiększa efektywność aplikacji [7].

Unit of Work często współpracuje z innymi wzorcami ORM, np. z *Data Mapper*, gdzie mappery odpowiadają za mapowanie obiektów, a *Unit of Work* za zarządzanie ich trwałością.

Listing 4 Przykład implementacji wzorca *Unit of Work*

```
class UnitOfWork {
  private newEntities: object[] = [];
  private dirtyEntities: object[] = [];
  private removedEntities: object[] = [];

  registerNew(entity: object): void {
    this.newEntities.push(entity);
  }

  registerDirty(entity: object): void {
    if (!this.dirtyEntities.includes(entity)) {
      this.dirtyEntities.push(entity);
    }
  }

  registerRemoved(entity: object): void {
    this.removedEntities.push(entity);
  }

  commit(): void {
    this.insertNew();
    this.updateDirty();
    this.deleteRemoved();
    this.clear();
  }

  private insertNew(): void {
    this.newEntities.forEach((entity) => {
      // Wywołanie metody create na encji
    });
  }

  private updateDirty(): void {
    this.dirtyEntities.forEach((entity) => {
      // Wywołanie metody update na encji
```

```
36.     });
37.   }
38.
39.   private deleteRemoved(): void {
40.     this.removedEntities.forEach((entity) => {
41.       // Wywołanie metody remove na encji
42.     });
43.   }
44.
45.   private clear(): void {
46.     this.newEntities = [];
47.     this.dirtyEntities = [];
48.     this.removedEntities = [];
49.   }
50. }
51.
52. // Przykład użycia
53. // Korzystając z klasy User z Listingu 3
54. const unitOfWork = new UnitOfWork();
55.
56. const user1 = new User(1, 'Jan', 'Kowalski', 'jan.kowalski@example.com');
57. const user2 = new User(2, 'Anna', 'Nowak', 'anna.nowak@example.com');
58.
59. // Rejestruj zmiany
60. unitOfWork.registerNew(user1);
61. unitOfWork.registerNew(user2);
62.
63. // Zatwierdź wszystkie operacje
64. unitOfWork.commit();
```

Listing 4 korzysta z klasy `UnitOfWork` odpowiadającej za śledzenie obiektów, które zostały utworzone, zmodyfikowane lub przeznaczone do usunięcia, przechowując je odpowiednio w osobnych kolekcjach. Metody `registerNew`, `registerDirty` oraz `registerRemoved` rejestrują zmiany zachodzące na encjach, bez natychmiastowego wykonywania operacji na bazie danych. Dopiero wywołanie metody `commit` powoduje zbiorcze wykonanie wszystkich operacji oraz wyczyszczenie stanu.

Zaletą wzorca *Unit of Work* jest centralizacja zarządzania zmianami, co ułatwia utrzymanie spójności i integralności danych oraz pozwala grupować zmiany w jedną transakcję, którą można wycofać w razie błędów. Wadą tego podejścia jest konieczność śledzenia zmian co może wpływać na wydajność oraz trudniejsze zarządzanie błędami, gdyż błąd podczas zatwierdzania transakcji może spowodować wycofanie wszystkich zmian.

2.4 Istotne zagadnienia w kontekście ORM

Wspominając o wzorcach ORM warto wyjaśnić kilka powiązanych z nimi zagadnień, które bezpośrednio wpływają na sposób ich działania oraz efektywność wykorzystania w aplikacjach. Zrozumienie tych mechanizmów pozwala lepiej ocenić konsekwencje projektowe i wydajnościowe wynikające z użycia konkretnych technologii ORM.

Definicja 3

Lazy Loading [7] – w kontekście ORM oznacza, że powiązane dane nie są ładowane od razu przy pobieraniu obiektu z bazy danych. Zamiast tego, atrybuty reprezentujące powiązane dane pozostają puste lub zawierają specjalny wskaźnik. Przy pierwszym odwołaniu się do takiego atrybutu, ORM wykonuje dodatkowe zapytanie do bazy i pobiera potrzebne dane. Pozwala to uniknąć niepotrzebnego ładowania dużej liczby powiązanych obiektów, jeśli nie są one używane. Dzięki temu początkowe

pobieranie obiektów może być szybsze. *Lazy loading* stanowi przeciwieństwo ***eager loadingu***, gdzie wszystkie powiązane dane pobierane są od razu.

Wraz z operacją pobierania danych z bazy pojawia się ryzyko pobrania informacji wykraczających poza rzeczywiste potrzeby aplikacji. Taka sytuacja określana jest mianem ***overfetchingu***. Przykładem jest pobranie wszystkich kolumn tabeli, gdy używane są tylko dwie. Prowadzi to do niepotrzebnego przesyłania i przetwarzania danych, co może obniżać wydajność aplikacji. Przeciwieństwem tej sytuacji jest ***underfetching***, gdzie aplikacja pobiera z bazy zbyt mało danych i musi wykonywać kolejne zapytania, aby uzyskać brakujące informacje, co może spowolnić działanie aplikacji.

Podejście do wczytywania danych ma istotny wpływ na sposób generowania zapytań do bazy danych. Nieodpowiednie dobranie strategii ładowania może prowadzić nie tylko do pobierania nadmiarowych informacji, ale również do zwiększenia liczby zapytań wykonywanych przez aplikację. W konsekwencji pojawiają się problemy wydajnościowe, z których jednym z najczęściej spotykanych w kontekście ORM jest problem N+1.

Definicja 4

Problem N+1 [9] występuje, gdy aplikacja korzysta z mechanizmu *lazy loading* i pobiera powiązane dane dla wielu obiektów. Najpierw ORM wykonuje jedno zapytanie, aby pobrać główne obiekty (np. listę użytkowników). Następnie, gdy aplikacja przechodzi przez każdy z tych obiektów i odwołuje się do powiązanych danych (np. postów użytkownika), ORM wykonuje osobne zapytanie dla każdego obiektu. W efekcie, dla N głównych obiektów powstaje łącznie N+1 zapytań do bazy danych. To prowadzi do niepotrzebnego obciążenia bazy i spowolnienia działania aplikacji.

Problem N+1 może być ograniczany poprzez świadome zarządzanie strategiami ładowania danych w ORM, czyli wykorzystywanie *eager loading* dla wybranych relacji, czy optymalizacja zapytań poprzez jawne określanie potrzebnych danych.

Równie istotnym aspektem pracy z ORM jest zarządzanie zmianami w strukturze bazy danych, czyli tzw. **migracje**. W tradycyjnym podejściu każda zmiana (np. dodanie kolumny) wymaga ręcznego pisania skryptów np. `ALTER TABLE`. ORM automatyzuje ten proces, traktując kod aplikacji (modele) jako „źródło prawdy". Dzięki temu można generować skrypty migracyjne bezpośrednio na podstawie różnic między aktualnym stanem modeli a schematem bazy danych, co minimalizuje ryzyko błędów ludzkich i literówek w zapytaniach SQL. Migracje gwarantują, że każdy członek zespołu pracuje na identycznej strukturze danych, eliminując konflikty wynikające z niespójności lokalnych baz.

2.5 Rola ORM w aplikacjach internetowych

ORM odgrywa kluczową rolę w architekturze współczesnych aplikacji internetowych, stanowiąc pomost między obiektowym paradygmatem programowania a relacyjnymi bazami danych. Pozwala na manipulację danymi poprzez obiekty, eliminując konieczność ręcznego pisania zapytań SQL i znacząco upraszczając proces zarządzania danymi.

W kontekście aplikacji webowych, ORM zapewnia abstrakcję warstwy dostępu do danych, co przekłada się na zwiększenie produktywności programistów, redukcję błędów związanych z nieprawidłowym mapowaniem typów danych oraz ułatwienie konserwacji kodu. Dodatkowo, narzędzia ORM oferują zaawansowane funkcjonalności, takie jak automatyczne generowanie schematów baz danych, obsługa migracji czy cache'owanie zapytań.

Zalety ORM w kontekście aplikacjii internetowych:

- **Abstrakcja nad SQL** – ORM pozwala programistom operować na danych w sposób obiektowy, eliminując konieczność bezpośredniego pisania zapytań SQL. Dzięki temu logika aplikacji jest bardziej spójna i łatwiejsza do zrozumienia.

- **Bezpieczeństwo** – chronią przed atakami typu *SQL Injection* poprzez automatyczne parametryzowanie zapytań.
- **Zarządzanie relacjami** – pozwalają na definiowanie relacji między tabelami w sposób naturalny dla programowania obiektowego, co upraszcza implementację złożonych struktur danych.
- **Zarządzanie migracjami** – większość ORM-ów oferuje wbudowane narzędzia do migracji. Dzięki temu można kontrolować wersje schematu i łatwo wprowadzać lub cofać zmiany, co zapewnia spójność środowisk deweloperskich.
- **Niezależność od bazy danych** – pozwalają przełączać się między różnymi systemami baz danych przy minimalnych zmianach w kodzie.
- **Automatyczna walidacja danych** – wiele ORM-ów zawiera wbudowaną walidację, pomagając wychwycić błędy zanim dane dotrą do bazy danych.
- **Mocna kontrola typologiczna** – poprzez mapowanie tabel na klasy każda kolumna z bazy danych zyskuje swój odpowiednik w typach języka programowania. Eliminuje to błędy polegające np. na próbie przypisania tekstu do pola liczbowego, już na etapie pisania kodu, a nie dopiero po wykonaniu zapytania do bazy.
- **Wsparcie dla IDE i autouzupełnianie** – operując na obiektach, środowiska programistyczne mogą podpowiadać nazwy pól i metod. Programista nie musi pamiętać dokładnych nazw kolumn, co redukuje liczbę błędów.
- **Łatwiejszy refaktoring** – jeśli zmienia się nazwa pola w modelu, narzędzia do refaktoryzacji w IDE mogą automatycznie zaktualizować tą zmianę we wszystkich miejscach w kodzie.
- **Mechanizm cache'owania** – pozwala to uniknąć wielokrotnego odpytywania bazy o te same dane, w obrębie jednego cyklu żądania, co pozwala na redukcję opóźnień i odciąża serwer bazy.
- **Zwiększona produktywność** – dzięki ORM programiści mogą szybciej tworzyć i rozwijać aplikacje, koncentrując się na logice biznesowej zamiast na operacjach bazodanowych.

Chociaż ORM-y oferują wiele korzyści, ich użycie wiąże się także z pewnymi wyzwaniami. Korzystanie z ORM-a, wymaga poznania jego składni oraz nauczenia się jego specyfiki i metod. Pochłania to dodatkowy czas i może spowolnić początkowy etap prac nad aplikacją.

Abstrakcja nad SQL może niekiedy prowadzić do generowania mniej optymalnych zapytań, co w dużych aplikacjach potrafi negatywnie wpłynąć na wydajność. Warto mieć na uwadze, że nie każde zapytanie da się napisać z wykorzystaniem metod oferowanych przez ORM.

	id	employee_id	start_time	end_time
1	1	1	2024-01-15 08:00:00.000000	2024-01-15 16:00:00.000000
2	2	1	2024-01-16 08:30:00.000000	2024-01-16 17:00:00.000000
3	3	2	2024-01-15 09:00:00.000000	2024-01-15 17:30:00.000000
4	4	2	2024-01-16 08:00:00.000000	2024-01-16 16:30:00.000000
5	5	3	2024-01-15 07:30:00.000000	2024-01-15 15:30:00.000000
6	6	3	2024-01-16 09:30:00.000000	2024-01-16 18:00:00.000000
7	7	1	2024-01-17 08:15:00.000000	2024-01-17 16:45:00.000000
8	8	2	2024-01-17 08:45:00.000000	2024-01-17 17:15:00.000000

Rysunek 4 Tabela *WorkEntry*. Źródło: opracowanie własne

Rysunek 4 przedstawia tabelę zawierającą informację o czasie rozpoczęcia i zakończenia pracy dla poszczególnych pracowników. Chcąc napisać zapytanie, które zwróci sumę czasu pracy w minutach, dla każdego pracownika, można użyć zapytania PostgreSQL przedstawionego w Listing 5.

Listing 5 Zapytanie PostgreSQL pobierające czas pracy dla każdego pracownika

```
1. SELECT employee_id, SUM(EXTRACT(EPOCH FROM end_time - start_time)/60) AS
total_minutes
2. FROM "WorkEntry"
3. GROUP BY employee_id;
```

Tego zapytania nie da się odwzorować bezpośrednio w ORM takim jak Prisma, ponieważ nie obsługuje ona agregacji na wyrażeniach w pojedynczym zapytaniu. Trzeba wykorzystać przetworzenie danych w języku programowania, aby uzyskać pożądany wynik, jak przedstawia to Listing 6.

Listing 6 Zapytanie pobierające czas pracy dla każdego pracownika przy użyciu Prismy

```
1. const entries = await prisma.workEntry.findMany();
2.
3. const summary = entries.reduce((acc, entry) => {
4.   const minutes = (entry.endTime.getTime() - entry.startTime.getTime()) / 60000;
5.   acc[entry.employeeId] = (acc[entry.employeeId] || 0) + minutes;
6.   return acc;
7. }, {});
```

W takich przypadkach programiści często korzystają z mechanizmów dostosowywania zapytań lub łączą ORM z narzędziami typu *Query Builder*, które oferują większą kontrolę nad generowanym SQL-em.

2.6 Query builder

Query builder to narzędzie umożliwiające dynamiczne budowanie zapytań SQL przy użyciu interfejsu programistycznego zamiast pisania surowych zapytań w SQL [10]. W przeciwieństwie do ORM, który odwzorowuje tabele na obiekty, pozwala na większą kontrolę nad generowanym kodem SQL, jednocześnie eliminując konieczność ręcznego budowania zapytań ze zmiennych tekstowych. Jest on również o wiele lżejszy niż pełen ORM, jednocześnie zapewniając podobną łatwość budowania zapytań. Przykładem *query builder* dla Node.js jest Knex.js oraz Kysely.

Listing 7 Przykład zapytania w SQL oraz w Knex.js

```
 1. // SQL
 2. SELECT *
 3. FROM users
 4. LEFT JOIN accounts ON users.id = accounts.user_id;
 5.
 6. // Knex.js
 7. knex
 8. .select('*')
 9. .from('users')
10. .leftJoin('accounts', 'users.id', 'accounts.user_id');
```

Narzędzia tego typu dodają warstwę abstrakcji nad językiem SQL, ale w przeciwieństwie do ORM-ów, ich działanie jest bliższe natywnym zapytaniom do bazy. Listing 7 pokazuje, że sama składania zapytanie nie odbiega od czystego SQL. Tworzenie zapytań odbywa się poprzez łańcuchowanie metod, co poprawia czytelność kodu. Większość narzędzi tego typu automatycznie dba o oczyszczanie danych wejściowych oraz prawidłowe formatowanie zapytań, minimalizując ryzyko błędów związanych z konkatenacją stringów zapewniając ochronę przed atakami *SQL Injection* [11].

Jednym z głównych atutów narzędzi *query builder* jest ich elastyczność oraz możliwość precyzyjnej kontroli nad optymalizacją zapytań. W przeciwieństwie do ORM-ów, nie narzucają one struktury obiektowej na dane ani nie wprowadzają dodatkowej warstwy abstrakcji, co sprawia, że programista ma pełną kontrolę nad relacjami między tabelami oraz sposobem wykonywania operacji na bazie danych. To sprawia, że są chętnie wykorzystywane w projektach, gdzie wymagana jest wysoka wydajność oraz optymalizacja zapytań. Są również przydatne w aplikacjach, które dynamicznie generują zapytania SQL na podstawie różnych warunków, na przykład w systemach raportowania, wyszukiwania czy filtrowania danych. Pozwalają na łatwe budowanie złożonych zapytań bez konieczności ręcznego łączenia fragmentów tekstu, co zmniejsza ryzyko błędów i ułatwia utrzymanie kodu.

Mimo swoich zalet, *query builder* wymaga większej znajomości SQL niż ORM. Użytkownik musi być świadomy struktury swojej bazy danych oraz zależności między tabelami. Wybór między ORM a *query builderem* zależy od specyfiki aplikacji. ORM jest bardziej odpowiedni w przypadku aplikacji wymagających łatwego odwzorowania encji na tabele, podczas gdy *query builder* sprawdzi się lepiej tam, gdzie konieczna jest kontrola nad wykonywanymi zapytaniami i wydajnością bazy danych.

2.7 Przegląd dostępnych technologii ORM dla Node.js

W ekosystemie Node.js istnieje wiele technologii ORM, które ułatwiają pracę z relacyjnymi bazami danych, eliminując konieczność ręcznego pisania zapytań SQL. W tym podrozdziale przedstawiony zostanie przegląd najpopularniejszych rozwiązań, a ich głębsze omówienie wraz z analizą wydajności i implementacją zostanie opisane w kolejnych rozdziałach.

Na przestrzeni lat powstało wiele różnych technologii ORM dla Node.js. Obecnie najczęściej stosowane rozwiązania to:

- **Prisma** – nowoczesny ORM, który kładzie duży nacisk na auto generowanie typów w TypeScript,
- **Sequelize** – jeden z najstarszych ORM-ów dla Node.js,
- **Objection.js** – ORM oparty na Knex.js, obecnie już nie rozwijany, od 2022 roku podnoszone są w nim tylko wersje pakietów oraz okazjonalnie dołączane poprawki błędów [12],
- **TypeORM** – ORM stworzony specjalnie dla TypeScript, wspierający dekoratory i silne typowanie,
- **MikroORM** – lekki ORM, zoptymalizowany pod kątem wydajności.

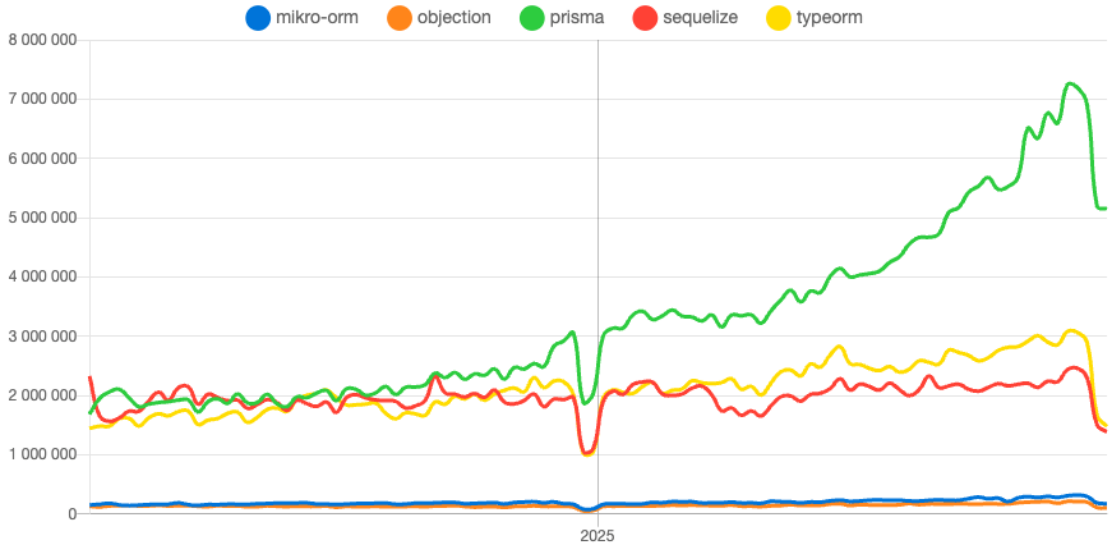
Istnieją również starsze i mniej rozwijane technologie, które wciąż mogą być wykorzystywane w niektórych projektach, ale nie są już aktywnie wspierane lub tracą na popularności. Należą do nich:

- **Bookshelf.js** – oparty na Knex.js, podobnie jak Objection.js, ale nieutrzymywany (ostatnie zmiany w 2020 roku [13])
- **Waterline** (część frameworka Sails [14]) – ORM, który zapewniał uniwersalny interfejs do pracy z różnymi bazami danych, również przestał być utrzymywany. Brakuje także dokumentacji na temat implementacji poza frameworkiem Sails.

Tabela 2 Zestawienie popularności omawianych ORM-ów. Źródło: opracowanie własne

Nazwa	**Pierwsza wersja w Github (rok)**	**Najnowsza aktualizacja**	**Tygodniowa liczba pobrań z npm** [15]
Prisma	2019	grudzień 2025	5 983 000
TypeORM	2016	grudzień 2025	2 026 000
Sequelize	2011	marzec 2025	1 743 000
MikroORM	2018	styczeń 2026	206 000
Objection.js	2015	wrzesień 2024	152 000

Dane przedstawione w Tabela 2 pokazują, jak zróżnicowany jest rynek narzędzi ORM dla Node.js pod względem popularności, wieku oraz tempa rozwoju. Aktualnie najpopularniejszym ORM-em, a zarazem najmłodszym spośród omawianych, jest Prisma, mająca około 6 milionów tygodniowych pobrań. Świadczy to o jej szerokim przyjęciu przez społeczność programistów. Wykres 1 również wyraźnie pokazuje znaczący wzrost dominacji Prismy w ostatnich latach, która wyprzedziła pozostałe technologie pod względem ilości pobrań, a co za tym idzie popularności.



Wykres 1 Popularność technologii ORM dla Node.js w latach 2024-2025. Źródło: [16]

Tabela 2 oraz Wykres 1 obrazują, że mimo pojawiania się nowych technologii, starsze narzędzia wciąż cieszą się dużą popularnością i są aktywnie wykorzystywane w projektach. Warto zwrócić uwagę, że Prisma i TypeORM, które mocno wspierają TypeScript i nowoczesne podejście do typowania, zdobywają coraz większe uznanie wśród deweloperów.

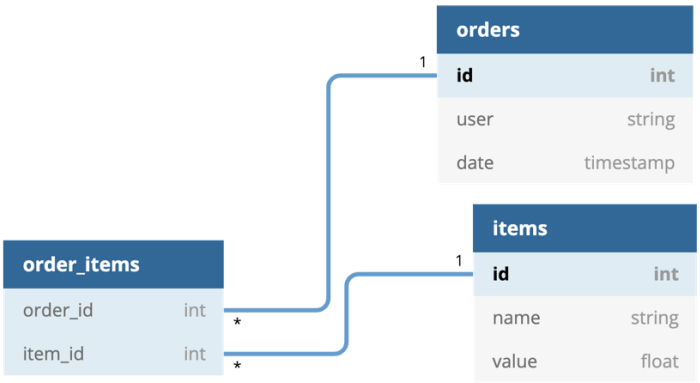
Powyższe dane pokazują również pewien podział narzędzi ORM, na te o ugruntowanej pozycji oraz na te bardziej niszowe. Do drugiej grupy zaliczają się MikroOrm oraz Objection.js. Wynika to z braku rozwijania Objection.js oraz z niszowości MikroOrm.

2.8 Przegląd dostępnych opracowań porównawczych ORM

Przegląd istniejących testów i opracowań porównujących narzędzia ORM pozwala na obiektywną ocenę poszczególnych technologii oraz identyfikację obszarów, które wymagają dalszych badań. W ramach tego podrozdziału zastaną

2.8.1 Emanuel Casco - porównanie Sequelize, Knex, TypeORM i Objection.js z 2019 roku

W ramach przeglądu dostępnych badań przeprowadzonych na ORM-ach w środowisku Node.js warto zwrócić uwagę na opracowanie wykonane w 2019 roku, przez Emanuela Casco [17] gdzie zostały porównane 4 technologie – Sequelize, Knex, TypeORM oraz Objection.js. Autor poddał analizie popularność, dokumentację, obsługę TypeScript oraz wydajność każdej z technologii. Przygotował w tym celu prostą aplikację w Express.js, która pozwalała na wykonywanie zapytań GET i POST.



Rysunek 5 Struktura tabel w porównaniu Emanuela Casco. Źródło: [17]

Rysunek 5 przedstawia strukturę wykorzystanej bazy danych. Zapytania dotyczyły pobrania listy zamówień, opcjonalnie z listą powiązanych elementów zamówienia oraz utworzenia nowych zamówień. Badanie wydajności zostało przeprowadzone poprzez narzędzie Autocannon, służące do testowania wydajności serwerów HTTP. Pozwala ono na symulowanie dużej liczby równoczesnych zapytań, dostarczając szczegółowe statystyki dotyczące czasu odpowiedzi, liczby obsłużonych żądań oraz rozkładu opóźnień [18].

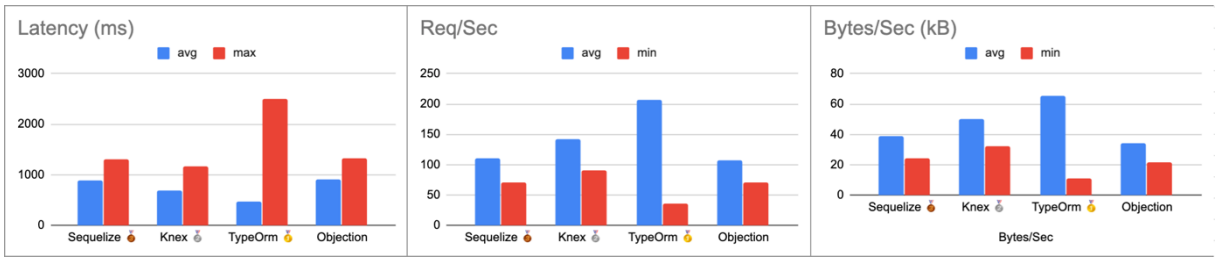
Analizując popularność, autor opierał się na danych z roku 2019, pochodzących z serwisu npmtrends.com. Najpopularniejszą biblioteką w tamtym czasie, spośród analizowanych był Sequelize. Casco zauważył również, że TypeORM potroił liczbę pobrań w okresie roku, co obrazuje jego zyskiwanie na popularności w tamtym czasie.

W porównaniu, autor zwrócił uwagę na różnice w dokumentacjach poszczególnych bibliotek. Najlepiej udokumentowane okazały się TypeORM oraz Objection. Posiadały one przejrzystą i czytelną dokumentację, z wieloma przykładami oraz poradnikami dla TypeScriptu. Dokumentacja Knex oraz Sequelize nie dostarczała informacji na temat implementacji przy użyciu TypeScripta. Dodatkowo w dokumentacji Sequelize zabrakło szczegółowych przykładów i wyjaśnień dla bardziej zaawansowanych przypadków.

Casco porównał również integrację z językiem TypeScript. Wszystkie analizowane biblioteki udostępniały własne typy, ale poziom integracji i wygoda pracy były różne. Sequelize wypadł najsłabiej – wymagał wielu ręcznych deklaracji typów, co utrudniało korzystanie z TypeScript, zwłaszcza początkującym. Definiowanie relacji było skomplikowane, a niektóre opcjonalne właściwości nie były poprawnie typowane. Knex jako *query builder*, wymagał samodzielnego utworzenia interfejsów, ale integracja z TypeScript była akceptowalna. Najlepiej ocenione zostały TypeORM i Objection – obie biblioteki oferowały prostą i intuicyjną integrację, automatycznie generując potrzebne struktury, a definicje modeli były czytelne i łatwe w użyciu. TypeORM wykorzystywał dekoratory, co znacząco upraszczało kod. Objection natomiast pozwalał uniknąć dekoratorów, zachowując prostą składnię i dobrą dokumentację.

W testach wydajności dla zapytań pobierających dane, najlepiej poradził sobie Knex, zaraz po nim był TypeORM i Objection.js – obydwie biblioteki miały bardzo zbliżony wynik przy zapisie

prostego obiektu jak i zagnieżdżonego. Wyniki testów zapisu danych przedstawia Rysunek 6. Najlepiej wypadł TypeORM, kolejny był Knex oraz Sequelize.



Rysunek 6 Wyniki testów zapisu danych dla zagnieżdżonych obiektów. Źródło: [17]

Autor w swoim porównaniu ograniczył się do przedstawienia surowych wyników testów wydajnościowych, bez głębszej interpretacji czy omówienia ich znaczenia w praktycznych zastosowaniach. Brakuje refleksji nad tym, jak różnice w wydajności mogą wpływać na rzeczywiste projekty lub które narzędzie sprawdzi się lepiej w określonych scenariuszach. Porównanie zostało opracowane w 2019, dlatego nie uwzględnia Prismy, która w tamtym czasie dopiero zaczynała być rozwijana. Obecnie Prisma jest jednym z najczęściej wybieranych ORM-ów dla TypeScripta, dlatego jej brak w zestawieniu ogranicza aktualność i kompletność analizy.

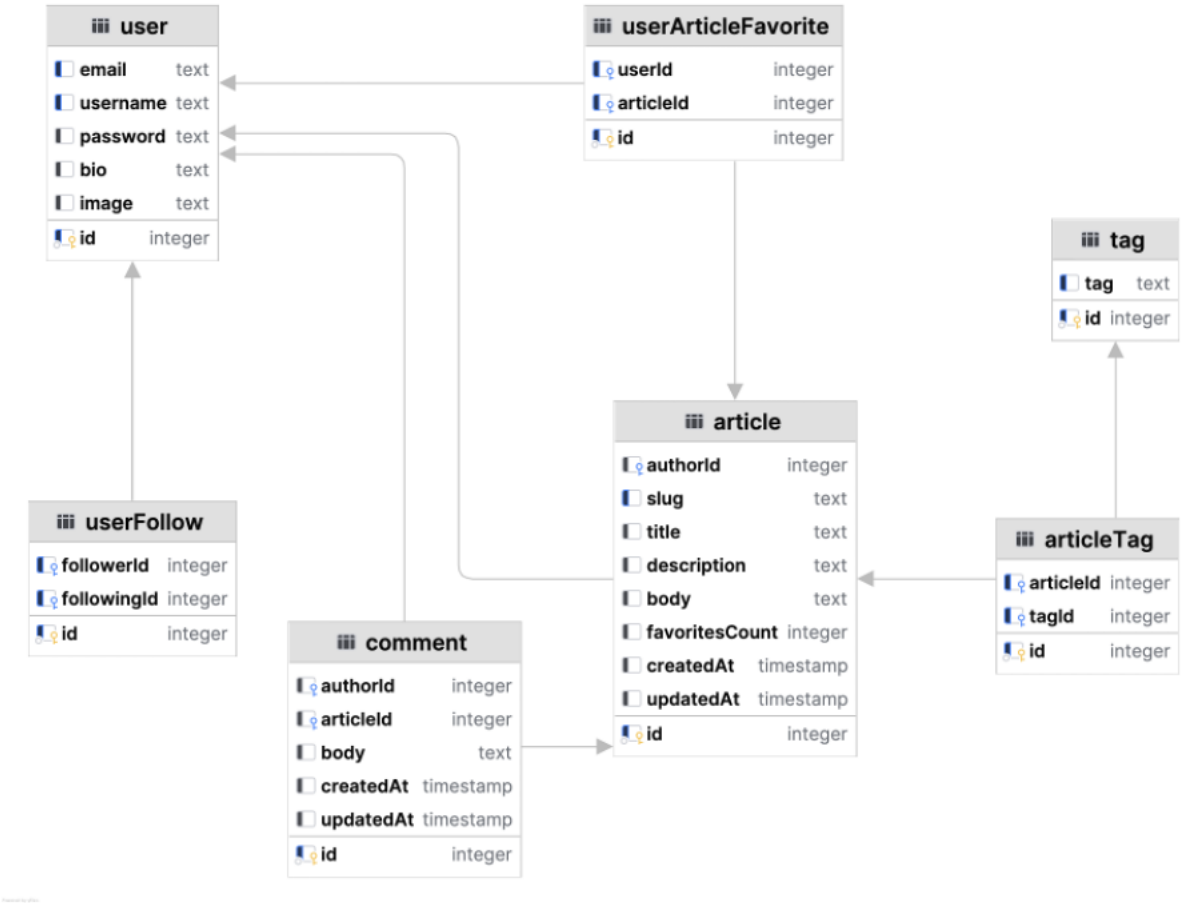
Kolejnym ograniczeniem badania jest zakres testów. Autor skupił się na prostych operacjach oraz niewielkiej liczbie rekordów w bazie danych. Nie przeprowadzono testów z bardziej złożonymi zapytaniami ani na dużych zbiorach danych. Takie testy mogłyby lepiej pokazać, jak poszczególne biblioteki radzą sobie w warunkach zbliżonych do produkcyjnych.

2.8.2 Roman Kushyn – porównanie 7 narzędzi ORM z 2022 roku

Kolejnym dostępnym badaniem jest porównanie Sequelize, TypeORM, MikroORM, Prismy, Objection.js, Knex oraz OrchidORM, autorstwa Romana Kushyn, przeprowadzone w 2022 roku [19]. Autor porównał w nim składnię, implementację oraz wydajność każdej z wymienionych technologii. Do porównania użył prostej aplikacji symulującej działanie bloga, wykonanej w Fastify.

Kluczowym elementem testów było zaimplementowanie złożonego zapytania pobierającego listę artykułów wraz z danymi powiązanymi, takimi jak tagi, profil autora, liczba polubień oraz informacja, czy zalogowany użytkownik polubił dany artykuł. Jako miara wydajności ORM-u służył czas potrzebny do wykonania zapytania. Użyto ponownie bazy danych PostgreSQL, jednak tutaj schemat był bardziej złożony niż w poprzednim badaniu. Składał się z 7 tabel a ich strukturę przedstawia Rysunek 7.

Autor szczegółowo opisuje swoje doświadczenia z każdą z technologii. Sequelize, mimo swojej popularności, został oceniony jako najtrudniejszy w użyciu i najwolniejszy. Jego implementacja wymagała obszernego, powtarzalnego kodu, a słabe wsparcie dla TypeScript i problemy z wydajnością zmusiły autora do sięgania po surowe zapytania SQL. W kontekście tej technologii poruszony został problem N+1, na temat, którego już w 2016 roku powstało zgłoszenie w repozytorium Sequelize w serwisie Github [20]. Problem pojawiał się podczas korzystania z powiązania `hasMany`, gdzie dla każdego pobieranego artykułu konieczne było wykonanie dodatkowego zapytania, aby załadować jego tagi.



Rysunek 7 Struktura tabel w opracowaniu Romana Kushyn. Źródło: opracowanie własne

TypeORM został opisany jako narzędzie problematyczne, z którym praca często polega na *"walce z ORM-em"*. ORM został przedstawiony jako ograniczony i zmuszający do korzystania z wbudowanego *query buildera* dla bardziej skomplikowanych zapytań. Autor napotkał także problemy z powielaniem rekordów przy złączaniu tabel relacyjnych, co wymagało stosowania zaawansowanych, specyficznych dla bazy danych PostgreSQL funkcji `json_agg` oraz `row_to_json`, aby uzyskać pożądany format danych.

MikroORM w ocenie autora jest podobny do TypeORM, ale ma własny zestaw problemów. Konfiguracja była skomplikowana, a dokumentacja niewystarczająca, dodatkowo nie działało w niej wyszukiwanie. Autor zwrócił uwagę na nieoczekiwane zachowania, takie jak automatyczna konwersja nazw tabel i kolumn na format *snake_case*. Dla niestandardowych zapytań, podobnie jak w przypadku TypeORM, konieczne było skorzystanie z *query buildera*. Problemy z wbudowanym mechanizmem *Unit of Work* podczas operacji wstawiania danych sprawiły, że w niektórych przypadkach konieczne było ominięcie go i użycie metod do bezpośredniego zapisu, co komplikowało logikę aplikacji.

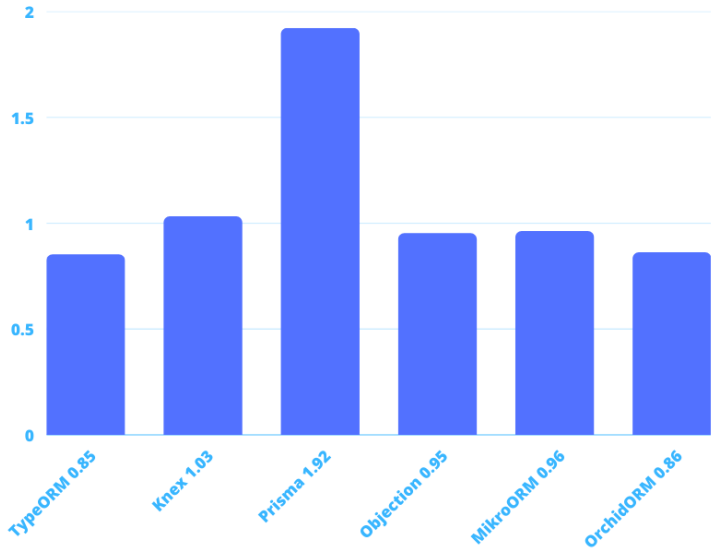
Prisma została uznana za narzędzie oferujące najlepsze doświadczenie deweloperskie. Autor pochwalił ją za łatwość konfiguracji, intuicyjną składnię opartą na własnym języku DSL oraz doskonałe wsparcie dla TypeScript. Główną wadą okazała się jednak wydajność, która była ograniczona przez unikalną architekturę Prismy – komunikację z bazą danych przez dodatkowy serwer pośredniczący napisany w języku Rust.

Objection autor ocenił pozytywnie za łatwość użycia i możliwość budowania dowolnych zapytań, na co pozwalał query builder Knex, na którym jest zbudowana biblioteka. Jej składnia była czysta i nie

wymagała dekoratorów, a specjalna funkcja modifiers pozwalała na reużywalność fragmentów zapytań. Największym minusem okazało się słabe wsparcie dla TypeScript w kontekście typowania wyników.

Knex został przedstawiony jako intuicyjny, dla osób dobrze znających SQL. Brak modeli i predefiniowanych relacji oznacza jednak konieczność ręcznego zarządzania logiką złączeń i większą powtarzalność kodu w dużych projektach. Według autora, żeby używać Knex w większym projekcie, należy zadbać od odpowiednią organizację zapytań, tak aby nie powtarzać wielokrotnie np. tych samych złączeń tabel.

Jako ostatni został przedstawiony OrchidORM, autorskie rozwiązanie autora artykułu, zaprezentowane jako próba połączenia najlepszych cech innych narzędzi. Oferuje ono zarówno interfejs wysokiego poziomu inspirowany Prismą, jak i elastyczność query buildera. Autor zachwala pełne bezpieczeństwo typów bez potrzeby ręcznych adnotacji i możliwość łatwego załączania fragmentów surowego SQL do zapytań budowanych przez ORM.



Wykres 2 Wyniki testu odczytu danych dla wybranych przez autora technologii. Źródło: [19]

Dla przetestowania wydajności odczytu danych, wykonano po 300 zapytań zwracających 100 artykułów wraz z powiązanymi polami. Wykres 2 przedstawia wyniki przeprowadzonego testu. Najwolniejszym narzędziem okazał się Sequelize (8.62 s – nieuwzględniony na wykresie), a zaraz po nim Prisma (1.92 s). Pozostałe narzędzia osiągnęły znacznie lepsze i zbliżone do siebie wyniki, a wśród nich najszybszy okazał się TypeORM (0.85 s), tuż przed OrchidORM (0.86 s). W testach zapisu wykonano masowe wstawianie 1000 artykułów z 5 tagami każdy. Tutaj najszybszy był Knex (4.43 s), a za nim OrchidORM (4.48 s). Czas pozostałych narzędzi był zbliżony (pomiędzy 5.2, a 6.7 s). Najgorzej wypadł MikroORM, z czasem zapisu ponad 7 s.

Ostateczne wnioski autora są krytyczne wobec wszystkich ORM-ów w Node.js. Sequelize jest jednoznacznie odradzany, ze względu na problemy z użyciem oraz kiepską wydajność. TypeORM i MikroORM są uznane za ryzykowne, pod względem występowania błędów i problemów. Prisma jest dobrym wyborem dla projektów, którym zależy na jak najlepszym wsparciu TypeScript, a przy tym nie stawiają na pierwszym miejscu wydajności zapytań. Autor konkluduje, że stworzenie wydajnego i elastycznego ORM-a jest możliwe, czego dowodem ma być jego własny projekt – OrchidORM.

Weryfikując te zapewnienia, można zauważyć, że OrchidORM jest bardzo niszowym projektem, obecnie w fazie rozwoju, z tygodniową średnią 1700 pobrań, wg serwisu npmjs.com [21]. Dodatkowo obsługuje wyłącznie bazę danych PostgreSQL. W dokumentacji autor wspomina, że nie planuje dodawania obsługi innych baz danych, koncentrując się na głębszej integracji z bogatym ekosystemem PostgreSQL i jego rozszerzeniami [22]. Kluczowe braki obejmują również brak możliwości zarządzania

relacjami między tabelami w różnych bazach danych, brak wsparcia dla typów zakresowych (*range*) i złożonych typów niestandardowych, a także wymagające rozbudowy możliwości przeszukiwania kolumn JSON. Według dostępnych informacji jest to bardziej ciekawostka niż rozwiązanie, które można zastosować w produkcyjnym projekcie.

2.8.3 Prisma – artykuł opisujący 11 technologii ORM z 2022 roku

Z dostępnych opracowań na omawiany temat występuje także artykuł na stronie Prismy [23] opisujący 11 technologii ORM oraz query builder, dostępnych dla Node.js. Wśród omawianych narzędzi znajdują się:

- Prisma,
- Sequelize,
- TypeORM,
- Mongoose,
- Bookshelf.js,
- Objection.js,
- Waterline,
- Knex.js.

Opracowanie pochodzi on z 2022 roku i jest to tylko porównanie opisowe, bez testów wydajności oraz rzeczywistych implementacji poszczególnych technologii. Na końcu znajduje się tabela zestawiająca opisywane rozwiązania pod względem popularności, aktywności, wsparcia, dojrzałości oraz obsługiwanych baz danych. Oceny poszczególnych kategorii przedstawione są za pomocą ikon, co nie niesie ze sobą wiele informacji. Artykuł nie formułuje jasnych wniosków, sprowadza się do krótkiego przedstawienia każdej z technologii i pokazania przykładowego kodu. Dodatkowo znajduje się na stronie jednego z porównywanych ORM-ów, co poddaje w wątpliwość jego bezstronność.

3. Charakterystyka badanych technologii ORM

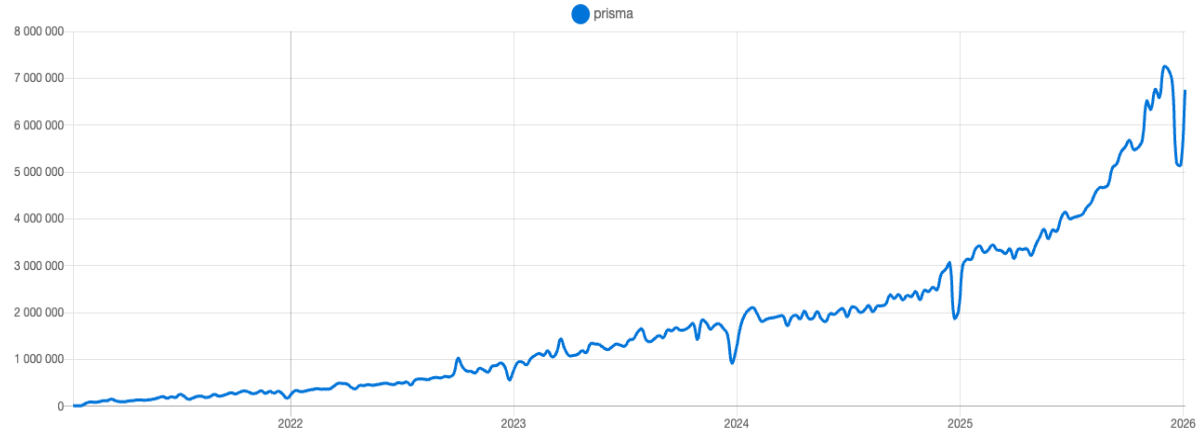
W tym rozdziale zostaną scharakteryzowane wszystkie porównywane technologie ORM, wraz z opisaniem wsparcia baz danych, mechanizmów działania oraz perspektyw rozwoju.

3.1 Prisma

Pierwsza produkcyjna wersja Prismy została opublikowana w 2021 roku. Narzędzie zaczynało jednak powstawać jeszcze w 2016 jako Graphcool - *Graphql based Backend as Service* (*BaaS*) [24]. Projekt został oficjalnie zawieszony w 2020 roku, na rzecz rozwijania Prismy, która w tamtym czasie była narzędziem do obsługi baz danych (*database toolkit*), a nie ORM-em. Z czasem zmieniono kierunek na rozwój pełnoprawnego narzędzia ORM. Ewolucję tych narzędzi przedstawia Rysunek 8.

Rysunek 8 Ewolucja Prismy. Źródło: [24]

Popularność Prismy w ekosystemie JavaScript gwałtownie wzrosła w ostatnich latach, co potwierdzają chociażby liczba pobrań z npm, przedstawiona na Wykres 3. Głównym czynnikiem tego sukcesu jest bardzo dobre doświadczenie deweloperskie, a w szczególności wsparcie dla TypeScript. Programiści cenią ją za to, że eliminuje błędy związanych z niezgodnością typów między kodem aplikacji a schematem bazy danych. Wokół Prismy zbudowała się duża i aktywna społeczność, która tworzy liczne poradniki, biblioteki rozszerzające jej funkcjonalność oraz zapewnia wsparcie na forach dyskusyjnych i platformach takich jak Discord czy Slack.



Wykres 3 Popularność Prismy według liczby tygodniowych pobrań z npm. Źródło: [25]

Za stabilnością Prismy stoi również solidne zaplecze komercyjne. Narzędzie jest rozwijane przez firmę Prisma Data Inc., zatrudniającą według serwisu pitchbook, ponad 130 osób [26]. Firma pozyskała finansowanie od czołowych inwestorów w branży technologicznej, w tym od firmy Vercel (twórców

Next.js), jednego z twórców Githuba, Thomasa Preston-Werner, czy od założyciela Heroku [27]. W 2022 roku firma ogłosiła zebranie 40 milionów dolarów w rundzie finansowania [24].

Korporacyjne wsparcie gwarantuje ciągły rozwój, profesjonalne utrzymanie oraz długoterminową stabilność projektu, co czyni go bezpiecznym wyborem dla zastosowań komercyjnych. Prisma jest uznawana za narzędzie *"enterprise-ready"*, a jej integracja z nowoczesnymi frameworkami, takimi jak Next.js, NestJS czy Express, dodatkowo umacnia jej pozycję jako jednego z wiodących rozwiązań do interakcji z bazą danych w nowoczesnym stosie technologicznym.

3.1.1 Obsługiwane bazy danych

Prisma zapewnia szerokie wsparcie dla różnorodnych systemów zarządzania bazami danych, obejmując zarówno relacyjne, jak i nierelacyjne rozwiązania. Na liście obsługiwanych baz znajdują się [28]:

- PostgreSQL,
- MySQL,
- MariaDB,
- Microsoft SQL Server,
- SQLite,
- MongoDB,
- CockroachDB.

Dodatkowo oferuje również wsparcie dla różnych rozproszonych bazy danych jak: Turso, Cloudflare D1, czy PlanetScale.

3.1.2 Architektura

W skład ORMa wchodzą właściwie 3 narzędzia [29]:

- Prisma Client - automatycznie generowany konstruktor zapytań dla Node.js i TypeScript,
- Prisma Migrate – system zarządzania migracjami,
- Prisma Studio - interfejs graficzny (GUI) do przeglądania i edycji danych w bazie.

Ekosystem Prismy składa się również z dużej ilości pakietów, takich jak generatory np. dokumentacji, schemy dbml, czy json.

Prisma implementuje wzorzec *Data Mapper*, ale inaczej niż w tradycyjnej definicji [8]. Główny obiekt `prisma` (instancja `PrismaClient`) pełni rolę mappera. Przedstawiony został w Listing 8. Jest jedynym punktem kontaktu z bazą danych, wszystkie operacje zapisu i odczytu przechodzą przez niego. Obiekty biznesowe (modele) nie wiedzą nic o tym, jak są zapisywane – nie mają metod `.save()` czy `.delete()`. Są to czyste obiekty z danymi. Cała logika komunikacji z bazą danych jest zamknięta w `PrismaClient`.

Listing 8 Przykładowe zapytanie w Prisma

```
1. const prisma = new PrismaClient()
2.
3. const user = await prisma.user.findUnique({
4.   where: {
5.     email: 'user@example.com',
6.   },
7. })
```

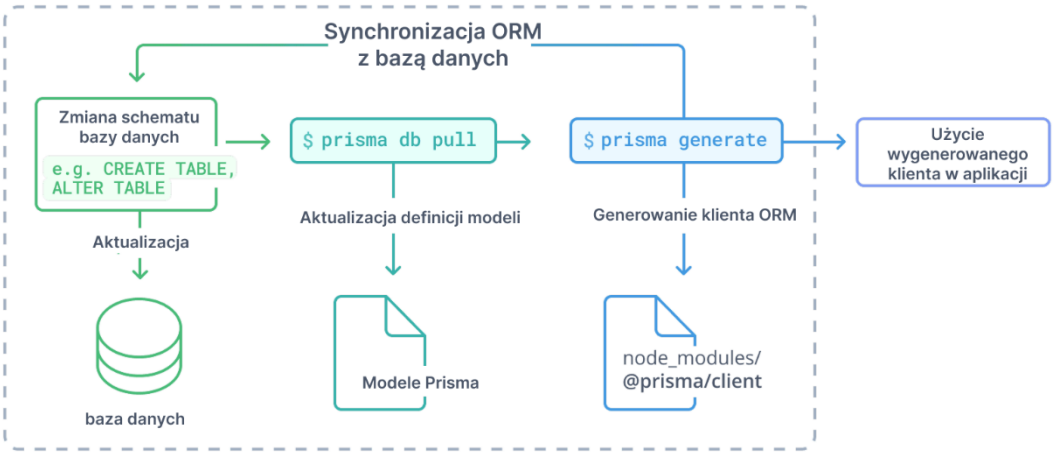
Każdy projekt korzystający z narzędzi Prismy wymaga stworzenia *schema.prisma* [29]. Listing 9 przedstawia przykładową konfigurację *schema.prisma*. Jest w nim zdefiniowany model danych aplikacji, za pomocą języka DSL. Definicja ta polega na utworzeniu bloków `model`, który reprezentuje tabelę w relacyjnych bazach danych. Wewnątrz każdego bloku określa się poszczególne pola, nadając im nazwę, typ oraz atrybuty. Na podstawie tego modelu narzędzie *Prisma Generate* tworzy w pełni otypowany `Prisma Client`, co gwarantuje, że wszystkie zapytania do bazy danych są zgodne ze schematem. Służy on również jako podstawa dla *Prisma Migrate* do automatycznego generowania skryptów migracyjnych.

Listing 9 Przykładowa konfiguracja *schema.prisma*

```
datasource db {
  provider = "postgresql"
}

generator client {
  provider = "prisma-client"
  output   = "./generated"
}

model Post {
  id        Int     @id @default(autoincrement())
  title     String
  content   String?
  published Boolean @default(false)
  author    User?   @relation(fields: [authorId], references: [id])
  authorId  Int?
}

model User {
  id    Int     @id @default(autoincrement())
  email String  @unique
  name  String?
  posts Post[]
}
```

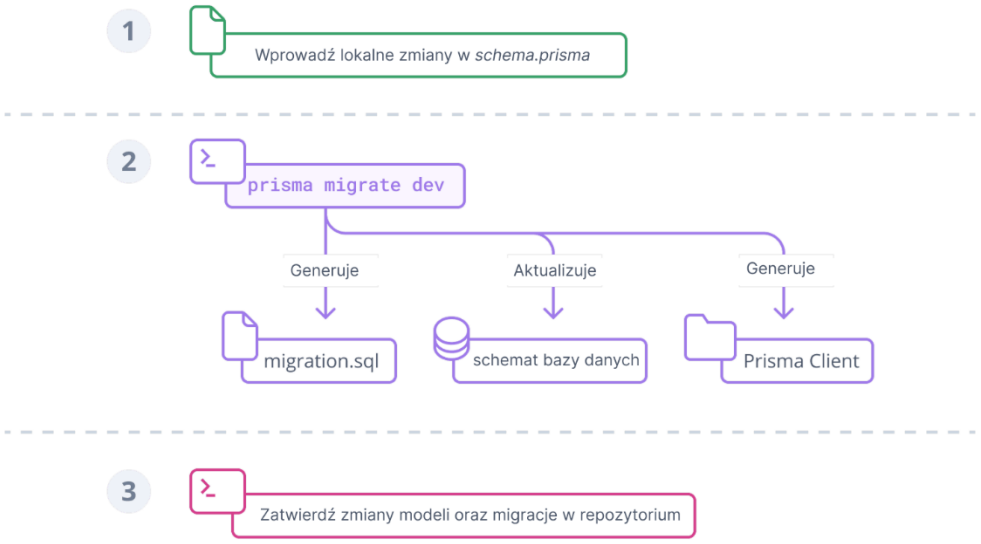
Istnieją dwa sposoby na stworzenie modelu danych w Prismie [29]:

- wygenerowanie modelu danych poprzez introspekcję bazy danych,
- ręczne napisanie modelu danych i zmapowanie go na bazę danych za pomocą Prisma Migrate.



Rysunek 9 Schemat generowania modelu z istniejącej bazy danych. Źródło: [30]

Rysunek 9 przedstawia proces mapowania w oparciu o introspekcję. Polega on na odczytaniu przez narzędzie Prisma istniejącej struktury bazy danych i automatycznym wygenerowaniu na tej podstawie schematu modeli. Jest to rozwiązanie szczególnie przydatne przy pracy z systemami zastanymi (*legacy*), gdzie bazowy model danych jest już zdefiniowany i wymaga jedynie odwzorowania w kodzie aplikacji. Po przeprowadzeniu introspekcji i wygenerowaniu klienta, programista uzyskuje dostęp do w pełni typowanych metod operujących na danych, co pozwala na bezpieczną integrację nowej logiki z istniejącą infrastrukturą.



Rysunek 10 Schemat mapowania modelu na bazę danych. Źródło: [29]

Rysunek 10 przedstawia proces mapowania modelu danych za pomocą narzędzia Prisma Migrate. Programista najpierw ręcznie modyfikuje model danych, a następnie za pomocą komendy CLI `prisma migrate dev` automatycznie generuje pliki migracji SQL oraz aktualizuje strukturę bazy danych. Proces ten kończy się automatyczną aktualizacją klienta (`Prisma Client`), co pozwala na natychmiastowe korzystanie z nowych struktur w kodzie aplikacji przy zachowaniu pełnej spójności między modelem a bazą.

Pod względem technicznym, `Prisma Client` składa się z trzech komponentów:

- biblioteka JavaScript,
- definicje typów TypeScript,
- silnik zapytań (*query engine*).

Kluczowym elementem architektury Prisma ORM, odpowiedzialnym za faktyczną komunikację z bazą danych, jest silnik zapytań (*query engine*). Jest to niskopoziomowy komponent, który pełni rolę pośrednika: przyjmuje zapytania generowane przez `Prisma Client` w języku JavaScript lub TypeScript, a następnie tłumaczy je na język SQL zrozumiały dla konkretnego silnika bazy danych. Odpowiada również za zarządzanie pulą połączeń, optymalizację planów zapytań oraz spójne mapowanie otrzymanych wyników z powrotem na obiekty JavaScript.

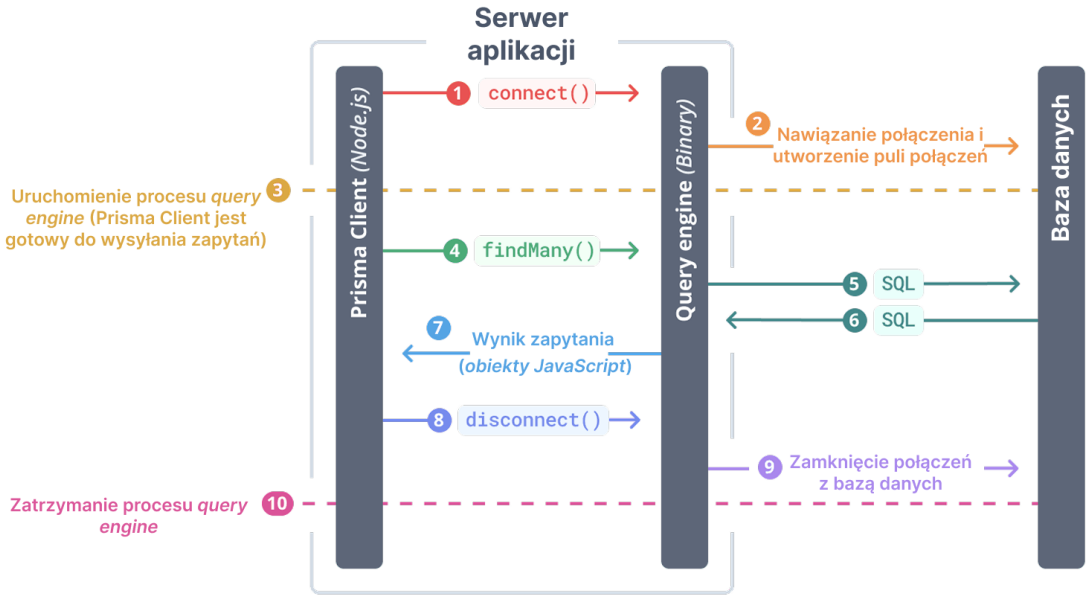
Przez lata komponent ten był dostarczany jako skompilowany plik binarny napisany w języku Rust, co zapewniało Prismie wysoką wydajność, ale wiązało się z koniecznością pobierania wersji silnika dopasowanej do konkretnego systemu operacyjnego. Aby wyeliminować te ograniczenia, twórcy Prismy zdecydowali się na migrację logiki silnika w stronę rozwiązań natywnych dla ekosystemu

Node.js, wykorzystując architekturę opartą na TypeScript i WebAssembly (Wasm). Nowe podejście jest określane w dokumentacji jako *„Rust-free”*.

W rezultacie współczesne wersje Prismy oferują dwa główne sposoby korzystania z silnika zapytań [31]:

- korzystanie z *query engine* napisanego w języku Rust,
- korzystanie z *driver adapters*.

W przypadku pierwszej opcji, silnik zapytań jest zaimplementowany w języku Rust i udostępnia niskopoziomowe API. Do niego przekazywane są zapytania, a on zajmuje się wszystkim, od budowania planu zapytania, poprzez wykonywanie zapytań i zwracanie wyników do klienta JavaScript. Zawiera wbudowane, zoptymalizowane sterowniki (*drivers*), które łączą się z bazą danych (np. PostgreSQL, MySQL) za pomocą połączenia TCP. Jest to standardowe, stanowe połączenie, które jest utrzymywane przez pewien czas.



Rysunek 11 Schemat działania silnika zapytań w języku Rust. Źródło: [31]

Rysunek 11 przedstawia działanie *query engine* w języku Rust. Proces rozpoczyna się od wywołania metody `connect()`, co skutkuje uruchomieniem silnika binarnego, nawiązaniem połączenia i utworzeniem puli połączeń. W trakcie pracy `Prisma Client` przesyła żądania (np. `findMany()`) do silnika, który tłumaczy je na język SQL, a po otrzymaniu odpowiedzi od bazy, mapuje surowe dane na obiekty JavaScript i odsyła je do aplikacji. Cały cykl kończy się zamknięciem połączeń i zatrzymaniem procesu silnika po wywołaniu metody `disconnect()`.

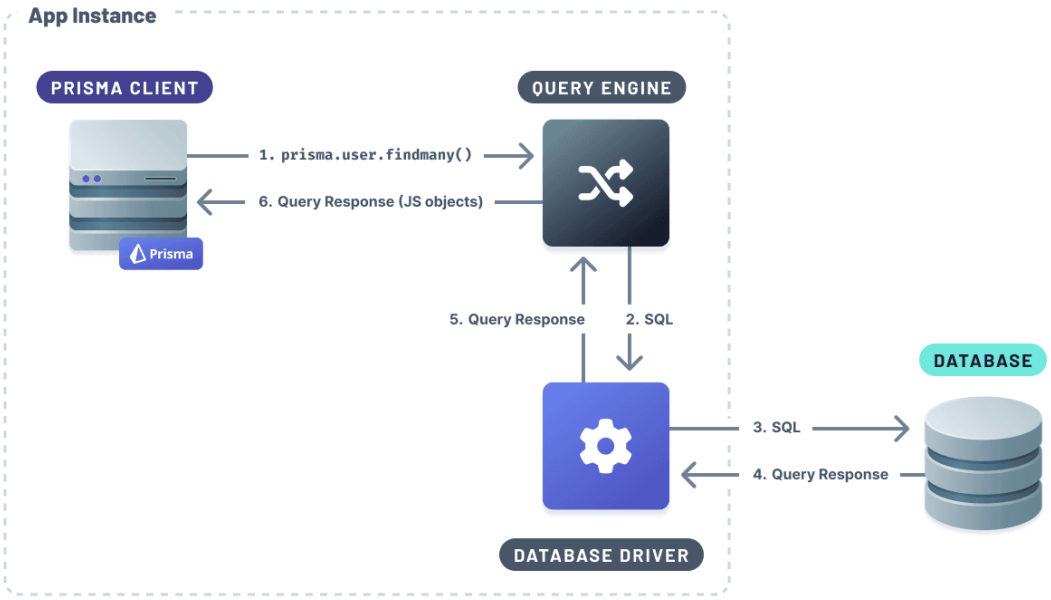
To podejście działa w tradycyjnych aplikacjach serwerowych. Jednak staje się problematyczne w nowoczesnych, bezserwerowych (*serverless*) środowiskach takich jak:

- Vercel Edge Functions,
- Cloudflare Workers,
- AWS Lambda.

W tych środowiskach funkcje są krótkotrwałe i bezstanowe. Ustanawianie nowego połączenia TCP dla każdej funkcji, która uruchamia się na chwilę, jest nieefektywne i powolne. Z tego powodu Prisma wprowadziła *driver adapters*. Jest to niewielka warstwa oprogramowania, która działa jako

pomost między silnikiem zapytań Prismy a specyficznym dla danej bazy danych sterownikiem opartym na JavaScript.

Podczas korzystania z *driver adapters*, silnik zapytań nadal tworzy plan zapytania i generuje instrukcje SQL jednak samo wykonanie tych zapytań i transmisja danych odbywa się za pośrednictwem adaptera. Dzięki temu aplikacje mogą komunikować się z bazą danych nie tylko przez standardowe połączenia TCP, ale również przez HTTP lub WebSokety.



Rysunek 12 Schemat działania silnika zapytań wraz z *driver adapter*. Źródło: [32]

To podejście umożliwia współpracę ze sterownikami napisanymi w JavaScript, ale wiąże się z kompromisem: dane muszą być serializowane z JavaScript do Rust, a następnie z powrotem do JavaScript. Ta komunikacja ma swój koszt i może spowalniać operacje, zwłaszcza przy dużej ilości danych. Powoduje to spadek wydajności i niweluje część korzyści płynących z tej metody.

3.1.3 Perspektywy rozwoju

W grudniu 2024 roku [33], Prisma ogłosiła migrację logiki *query engine* z języka Rust, na TypeScript. W nowej architekturze adaptery nadal są używane. Zamiast jednak polegać na silniku zapytań opartym na Rust, Prisma ORM przekaże zapytanie do kompilatora Wasm (*WebAssembly*), który zwróci plan zapytania. Następnie ten plan zostanie w całości wykonany w TypeScript.

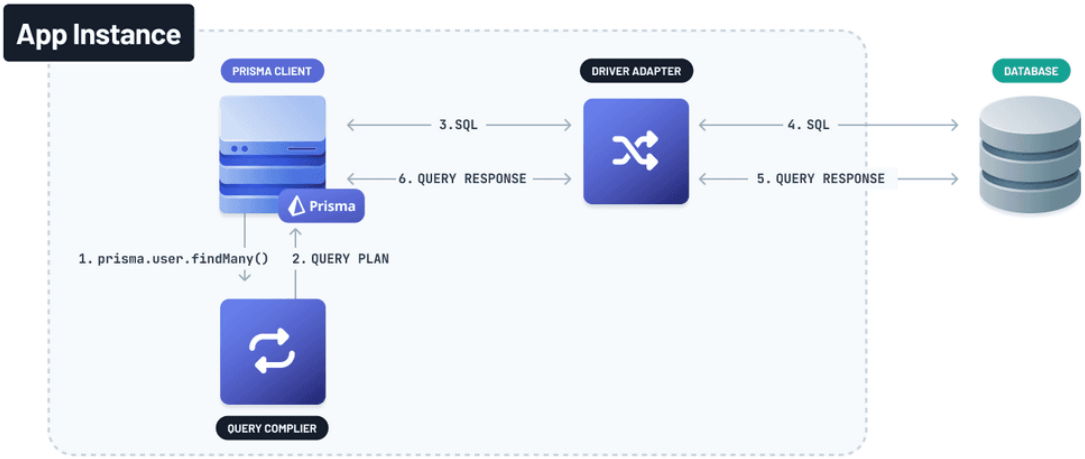
Rysunek 13 przedstawia sposób działania nowej architektury:

- `Prisma Client` współpracuje z *query compiler*, aby przetłumaczyć logikę aplikacji na konkretne zapytanie SQL.
- Wygenerowany kod SQL jest przekazywany do *driver adaptera*, który pełni rolę łącznika z bazą danych.
- Wynik zapytania wraca z bazy danych do adaptera, a następnie jest przekazywany z powrotem do `Prisma Client` jako gotowa odpowiedź dla aplikacji.

Ta uproszczona architektura przynosi kilka korzyści:

- zachowuje wsparcie dla sprawdzonych sterowników baz danych w JavaScript,

- ogranicza potrzebę tłumaczenia i minimalizuje ilość danych przesyłanych między Rust a JavaScript,
- eliminuje konieczność dostarczania zewnętrznego pliku binarnego, ponieważ kompilator zapytań nie zależy już od narzędzi specyficznych dla systemu operacyjnego.



Rysunek 13 Schemat działania nowej architektury Prismy, bez języka Rust. Źródło: [34]

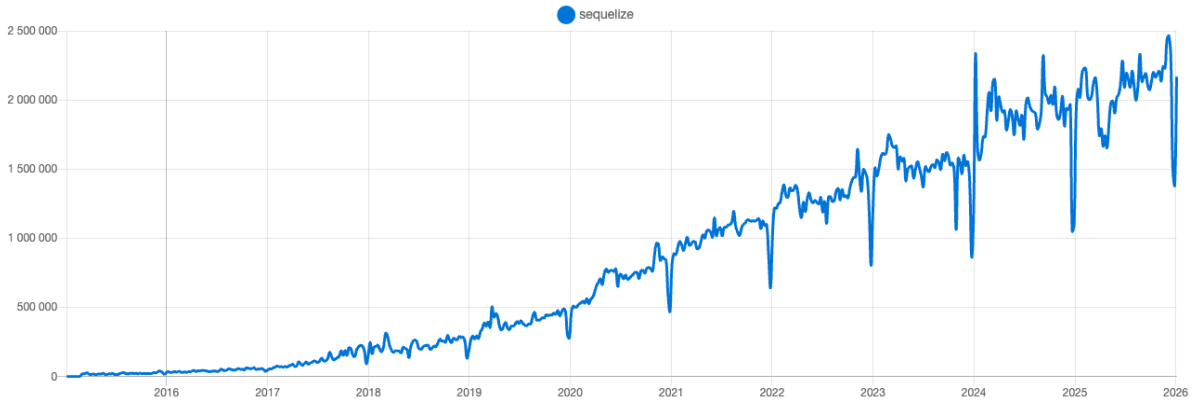
Dodatkowo po zmianie logiki, zmniejszeniu ulegnie rozmiar paczki. Wstępne testy [34] pokazywały, że silnik zapytań Prisma oparty na Rust zajmuje około 14 MB (7 MB po kompresji gzip), podczas gdy nowy *query compiler* to tylko około 1,6 MB (600 KB po kompresji gzip), co oznacza średnio redukcję rozmiaru o 85–90%.

3.2 Sequelize

Sequelize jest jednym z najbardziej dojrzałych ORM-ów w ekosystemie Node.js. Projekt jest aktywnie rozwijany od 2010 roku i posiada dużą społeczność użytkowników. Za rozwój, inaczej niż w przypadku Prismy, odpowiada społeczność programistów, a nie dział dużej firmy. Działalność projektu jest finansowana przez społeczność – firm oraz programistów, którzy na co dzień wykorzystują Sequelize w swoich projektach.

Taki model ma swoje wady i zalety. Z jednej strony, niezależność gwarantuje, że projekt nie zostanie nagle porzucony lub zmieniony z powodu decyzji biznesowej jednej firmy. Z drugiej strony, tempo rozwoju jest znacznie wolniejsze, a wprowadzanie dużych, przełomowych zmian (jak np. pełne, natywne wsparcie dla TypeScript) trudniejsze z powodu ograniczonych zasobów ludzkich i finansowych.

Wykres 4 przedstawia popularność Sequelize, poprzez liczbę tygodniowych pobrań z npm. Ciągle rosnący trend obrazuje dojrzałość technologii. Mimo że w nowo powstających projektach Sequelize coraz częściej ustępuje miejsca nowocześniejszym rozwiązaniom, jest on fundamentem wielu istniejących aplikacji typu *legacy*, co gwarantuje mu stałe miejsce w procesach utrzymania oprogramowania.



Wykres 4 Popularność Sequelize według liczby tygodniowych pobrań z npm. Źródło: [35]

3.2.1 Obsługiwane bazy danych

Na liście wspieranych przez Sequelize baz danych znajdują się [36]:

- MySQL,
- PostgreSQL,
- Snowflake,
- SQLite,
- MariaDB,
- OracleDb,
- Microsoft SQL Server,
- DB2 (dla systemów Linux, Unix, Windows).

Dzięki swojej dojrzałości i rozbudowie, Sequelize oferuje wsparcie dla niszowych lub korporacyjnych rozwiązań, takich jak Snowflake czy IBM DB2. Czyni go to bezpiecznym wyborem dla dużych projektów, gdzie istnieje ryzyko migracji między różnymi systemami bazodanowymi lub konieczność integracji z istniejącą, zróżnicowaną infrastrukturą, która niekoniecznie jest najnowocześniejszym systemem.

3.2.2 Architektura

Architektura narzędzia opiera się na systemie dialektów (*dialects*). Oznacza to, że Sequelize dostarcza uniwersalny, spójny interfejs API do interakcji z danymi, a za tłumaczenie tych operacji na specyficzną składnię SQL danego systemu bazodanowego odpowiada odpowiedni dialekt, np. PostgreSQL. Aby połączyć się z konkretną bazą, należy zainstalować nie tylko samą bibliotekę Sequelize, ale również odpowiedni pakiet sterownika, np. *pg* i *pg-hstore* dla PostgreSQL czy *mysql2* dla MySQL [37] , który odpowiada za fizyczną transmisję danych.

Dzięki temu podejściu programista może zmienić system bazodanowy projektu poprzez prostą modyfikację parametru dialect w konfiguracji, co znacząco zwiększa przenośność kodu i ułatwia migrację między różnymi środowiskami chmurowymi czy lokalnymi.

Fundamentem architektury Sequelize jest implementacja wzorca *Active Record*. Dokumentacja opisuje 2 podejścia do definiowania modeli [38] – przedstawia je Listing 10. Można zrobić to za pomocą metody `sequelize.define` lub za pomocą rozszerzania klasy `Model`. Obydwie metody są równoważne, ponieważ metoda define, wywołuje pod spodem `Model.init`.

Listing 10 Definiowanie modeli w Sequelize

```
// Define
const User = sequelize.define('User', {
    firstName: {
      type: DataTypes.TEXT,
      allowNull: false,
    },
    lastName: type: DataTypes.TEXT,
  },
);

// Class extends
class User extends Model {}

User.init(
  {
    firstName: {
      type: DataTypes.TEXT,
      allowNull: false,
    },
    lastName: {
      type: DataTypes.TEXT,
    },
  },
  {
    sequelize,
    modelName: 'User',
  },
);

const newUser = User.build({ firstName: 'Jan' });
await newUser.save();
```

Ważną część implementacji wzorca *Active Record* w Sequelize, są metody statyczne dostępne bezpośrednio na klasie modelu. Służą one do wyszukiwania i pobierania danych z bazy, czyli do wykonywania zapytań, które zwracają jedną lub więcej aktywnych instancji. Wynikiem takiego zapytania nie są zwykłe obiekty z danymi, ale w pełni funkcjonalne, instancje modelu. Oznacza to, że każdy obiekt zwrócony przez zapytanie można natychmiast modyfikować i zapisywać z powrotem do bazy. Listing 11 przedstawia przykład takiej operacji.

Listing 11 Użycie metody statycznej na klasie User

```
const user = await User.findOne({ where: { email: 'email@test.com' } });
user.lastName = 'Kowalski';
await user.save();
```

3.2.3 Perspektywy rozwoju

Aktualnie trwają prace nad wersją 7 biblioteki, która ma przynieść zmiany w strukturze modułów – od tej wersji poszczególne dialekty znajdą się w wydzielonych modułach np. *@sequelize/postgres*, co pozwoli na zmniejszenie rozmiaru głównej paczki. Dodatkowo wyeliminuje to konieczność instalowania osobnych pakietów sterowników. Głównym celem wersji 7 jest gruntowna modernizacja biblioteki, aby stała się narzędziem *„TypeScript-first”,* co eliminuje konieczność korzystania z zewnętrznych paczek takich jak *sequelize-typescript*. Będzie to największa aktualizacja od 2020 roku, kiedy opublikowano wersję 6.

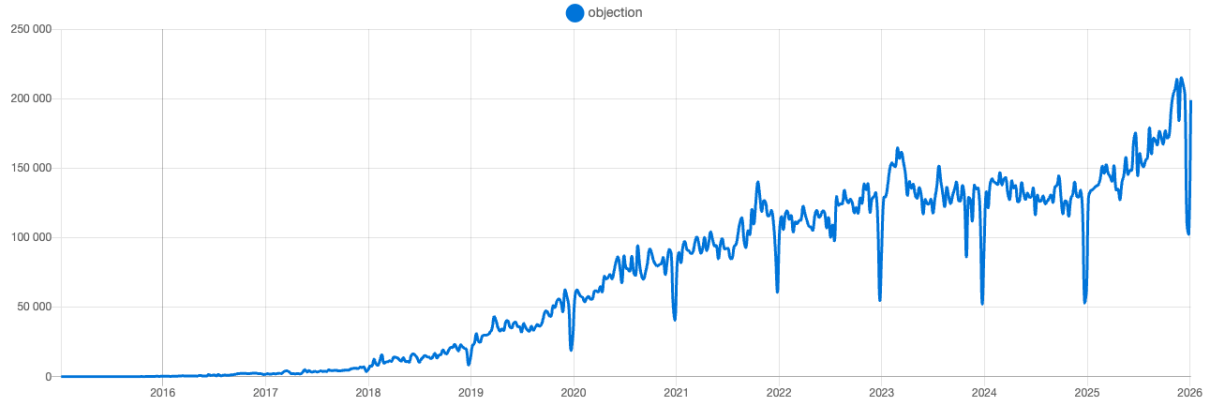
Nie została jednak określona data wydania wersji stabilnej, obecnie wszystkie zmiany są publikowane jako *aplha*. Biorąc pod uwagę, że w ciągu całego roku 2025 opublikowano jedynie cztery

wydania [39] oraz fakt, że projekt wciąż nie opuścił wczesnej fazy testów, można przypuszczać, iż debiut wersji stabilnej nie nastąpi w najbliższym czasie.

3.3 Objection.js

Objection różni się od wcześniejszych ORM-ów tym, że jak wskazuje dokumentacja [40] jest bardziej relacyjnym konstruktorem zapytań (*relational query builder*). Został zbudowany na bazie Knex.js. Łączy w sobie elastyczność Knex.js do budowania dowolnych zapytań SQL oraz strukturę i udogodnienia ORM-a, takie jak modele i zarządzanie relacjami.

Projekt powstał w 2015 roku, a jego autorem jest Sami Koskimäki – fiński programista. Obecnie ORM zajmuje niszową pozycję w ekosystemie Node.js. Notuje regularną liczbę poniżej 200 tysięcy pobrań tygodniowo w npm. Wykres 5 pokazuje, że ten stan utrzymuje się w ostatnich latach.



Wykres 5 Popularność Objection.js według liczby tygodniowych pobrań z npm. Źródło: [41]

3.3.1 Obsługiwane bazy danych

Na liście wspieranych baz danych znajdują się:

- PostgreSQL
- MySQL
- SQLite
- OracleDb

Wsparcie baz danych jest w pełni zależne i dziedziczone po Knex.js. ORM nie posiada własnych sterowników ani logiki do komunikacji z bazami danych. Zamiast tego, w całości deleguje to zadanie do instancji Knex, która musi być skonfigurowana i przekazana do Objection.js podczas inicjalizacji. Oznacza to, że jeśli Knex.js wspiera daną bazę danych, to Objection.js będzie z nią bezproblemowo współpracować.

3.3.2 Architektura

Objection.js implementuje wzorzec *Active Record*. Architektura ta integruje wewnątrz jednej klasy zarówno strukturę danych, jak i logikę dostępu do nich, wykorzystując do komunikacji z bazą zintegrowany interfejs *Query Builder*. W przeciwieństwie do bardziej rygorystycznych implementacji tego wzorca, Objection.js nie wymaga jednak jawnego definiowania atrybutów wewnątrz klasy, dynamicznie mapując właściwości obiektu na kolumny tabeli w czasie wykonania.

Definiowanie modeli w Objection.js charakteryzuje się prostotą i opiera się na standardowych elementach języka JavaScript. Model to czysta klasa JavaScript, która dziedziczy po bazowej klasie

`Model` importowanej z biblioteki. Nie ma tu potrzeby używania dekoratorów ani skomplikowanych funkcji konfiguracyjnych. Listing 12 przedstawia przykład definicji klasy `UserModel`. Co istotne, sama klasa modelu nie zawiera definicji poszczególnych kolumn (jak `name: string`). Jest ona raczej powłoką na dane pobrane z bazy, a prawdziwym źródłem prawdy o schemacie jest sama baza danych, zarządzana przez migracje Knex.js. Opcjonalnie model pozwala na zdefiniowanie statycznej metody `jsonSchema`, która służy do automatycznej walidacji obiektów przed ich zapisem, zapewniając integralność danych bez konieczności powielania struktury tabeli w kodzie.

Listing 12 Definicja modelu w Objection.js

```
import { Model } from 'objection';

class User extends Model {

  static get tableName() {
    return 'users';
  }
  static get idColumn() {
    return 'id';
  }

  static get jsonSchema() {
    return {
      type: 'object',
      required: ['firstName', 'lastName', 'email'],

      properties: {
        id: { type: 'integer' },
        firstName: { type: 'string', minLength: 1, maxLength: 255 },
        lastName: { type: 'string', minLength: 1, maxLength: 255 },
        email: { type: 'string', format: 'email' }
      }
    };
  }
}
```

Pisanie i wykonywanie zapytań w Objection.js opiera się na mechanizmie, którego punktem wyjścia jest zawsze statyczna metoda `query()` dostępna na każdej klasie modelu. Wywołanie tej metody zwraca instancję konstruktora zapytań, na której można łańcuchowo wywoływać kolejne metody w celu zbudowania docelowego zapytania SQL. Listing 13 przedstawia zbudowane w taki sposób zapytanie `UPDATE`. Wszystkie te metody pochodzą z instancji `QueryBuilder` i pozwalają na programistyczne tworzenie zapytań w sposób bardzo zbliżony do czystego SQL, ale z zachowaniem bezpieczeństwa i struktury.

Listing 13 Aktualizacja rekordu użytkownika w Objection.js

```
await User.query()
  .where('email', 'example@test.com')
  .patch({ lastName: 'Kowalski' });
```

Obiekt zwracany przez metodę `Model.query()` jest instancją Knex.js, rozszerzoną o dodatkowe funkcje Objection. Dzięki temu można bez przeszkód wplatać surowy kod SQL do budowanych zapytań za pomocą metody `raw()`. Można jej użyć w dowolnym miejscu łańcucha, co przedstawia Listing 14. Nie ogranicza się to tylko do `raw()`, programista ma dostęp do całego API Knex.js i może bezpośrednio na instancji zapytania Objection wywoływać dowolne metody Knex.

Listing 14 Łączenie metod obiektu wraz z metodą Knex.js

```
1. await User.query()
2.       .select('country')
3.       .count('* as userCount')
4.       .groupBy('country')
5.       .having(User.knex().raw('count(*) > ?', [5]))
6.       .orderBy('userCount', 'desc');
```

3.3.3 Perspektywy rozwoju

W 2022 roku Sami Koskimäki ogłosił, że Objection.js nie będzie dalej rozwijany, ale pozostanie utrzymywany [42]. Twórca przeniósł swoją uwagę na rozwój nowego projektu, którym jest *query builder* Kysley. Oznacza to, że Objection będzie otrzymywać poprawki błędów i aktualizacje bezpieczeństwa, ale nie będą wprowadzanie nowe funkcjonalności. Gwarantuje to pewną stabilność istniejącym systemom *legacy*. Nie ma natychmiastowej konieczności zmiany technologii, z powodu krytycznych luk bezpieczeństwa lub błędów uniemożliwiających działanie w nowych wersjach Node.js.

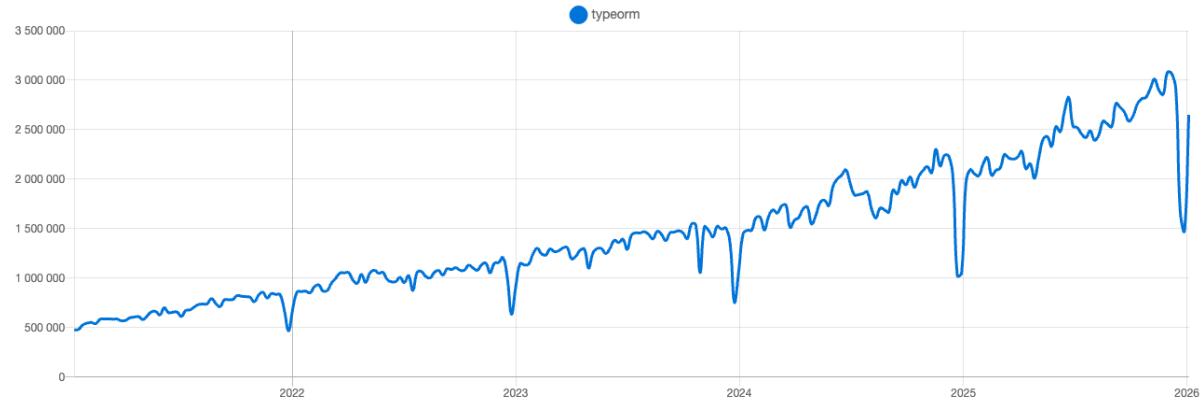
Wybieranie Objection.js do nowych systemów wiąże się z ryzykiem narastania długu technicznego. Brak nowych funkcjonalności oznacza również, że biblioteka nie będzie natywnie wspierać nowoczesnych rozwiązań bazodanowych.

3.4 TypeORM

TypeORM to elastyczny ORM, którego nadrzędną cechą jest to, że został napisany w całości w TypeScript i jest zoptymalizowany pod jego kątem. Jego nazwa nie jest przypadkowa – celem biblioteki jest zapewnienie jak najlepszego doświadczenia i bezpieczeństwa typowania podczas pracy z bazami danych.

Narzędzie jest rozwijane od 2016 roku jako niezależny projekt, stworzony przez Umeda Khudoiberdieva – programistę z Tadżykistanu. Obecnie TypeORM jest jednym z najpopularniejszych ORM-ów w ekosystemie Node.js, a jego popularność jest związana z frameworkiem NestJS. Pod względem liczby tygodniowych pobrań z npm, przedstawionych na Wykres 6, jest drugim co do wielkości ORM-em w środowisku Node.js.

TypeORM jest oficjalnie rekomendowanym i domyślnym narzędziem do interakcji z bazą danych w dokumentacji NestJS [43], co sprawia, że jest naturalnym wyborem dla wielu deweloperów korzystających z tego frameworka.



Wykres 6 Popularność TypeORM według liczby tygodniowych pobrań z npm. Źródło: [44]

3.4.1 Obsługiwane bazy danych

Na liście oficjalnie wspieranych baz danych znajdują się systemy [45]:

- PostgreSQL,
- MySQL,
- MariaDB,
- SQLite,
- MS SQL Server,
- Oracle,
- MongoDB,
- SAP HANA,
- CockroachDB,
- Google Spanner.

TypeORM jest drugim obok Prismy narzędziem, które oferuje wsparcie dla baz NoSQL, w szczególności dla MongoDB. Pozwala to na pracę z dokumentową bazą danych przy użyciu tego samego, API opartego na repozytoriach, co jest dużą zaletą dla zespołów pracujących nad różnorodnymi projektami.

3.4.2 Architektura

TypeORM, podobnie jak Sequelize, opiera się na systemie sterowników, co oznacza, że aby połączyć się z konkretną bazą, należy zainstalować odpowiedni pakiet z npm (np. *pg* dla PostgreSQL). Konfiguracja połączenia opiera się na stworzeniu i zainicjowaniu instancji klasy `DataSource`. Obiekt ten przechowuje wszystkie parametry połączenia i jest centralnym punktem dostępu do bazy danych w całej aplikacji.

Definiowanie modeli w TypeORM jest związane z TypeScriptem i opiera się na wykorzystaniu dekoratorów. Model to standardowa klasa TypeScript, którą "ozdabia się" specjalnymi dekoratorami, aby nadać jej znaczenie w kontekście bazy danych. Dekorator `@Entity()` oznacza klasę jako tabelę, `@Column()` definiuje jej pola wraz z typami (np. `@Column("varchar")`), a dekoratory takie jak `@OneToMany` czy `@ManyToOne` w opisują relacje między encjami. Podejście to różni się od wcześniej opisywanych technologii.

Jedną z najważniejszych i unikalnych cech TypeORM jest jego elastyczność w kwestii wzorców architektonicznych. Oferuje on wbudowane wsparcie dla dwóch głównych wzorców:

- *Active Record* – podobnie jak w Sequelize, instancje modeli są "aktywne" i posiadają metody takie jak `.save()`, `.remove()`, co upraszcza proste operacje CRUD.
- *Data Mapper* – jest to preferowany i bardziej skalowalny wzorzec, w którym logika dostępu do danych jest oddzielona od modeli. Operacje na bazie danych wykonuje się za pomocą tzw. repozytoriów (*repositories*), np. `userRepository.save(user)`. Modele w tym podejściu są "czystymi" obiektami, co ułatwia testowanie i utrzymanie aplikacji.

Listing 15 przedstawia definicję modelu w podejściu *Active Record*. Model dziedziczy pod klasie `BaseEntity`, co daje mu bezpośredni dostęp do metod. Dodatkowo istnieje również możliwość rozszerzania klasy modelu o statyczne metody.

Listing 15 Definicja modelu w TypeORM wraz z przykładem użycia dla wzorca *Active Record*

```
@Entity()
export class User extends BaseEntity {
    @PrimaryGeneratedColumn()
    id: number

    @Column()
    firstName: string

    @Column()
    lastName: string

    static findByName(firstName: string, lastName: string) {
        return this.createQueryBuilder("user")
            .where("user.firstName = :firstName", { firstName })
            .andWhere("user.lastName = :lastName", { lastName })
            .getMany()
    }
}

// Przykład użycia
const user = new User()
user.firstName = "Jan"
user.lastName = "Nowak"
await user.save()

const user1 = await User.findByName("Jan", "Kowalski")
```

Podejście *Data Mapper* zostało przedstawione przez Listing 16 . Model jest "czystą" klasą, a za wszystkie operacje na bazie danych odpowiada repozytorium.

Listing 16 Definicja modelu w TypeORM wraz z przykładem użycia dla wzorca *Data Mapper*

```
@Entity()
export class User {
    @PrimaryGeneratedColumn()
    id: number

    @Column()
    firstName: string

    @Column()
    lastName: string
}

// Przykład użycia
const userRepository = dataSource.getRepository(User)
const user = new User()
user.firstName = "Jan"
user.lastName = "Nowak"
await userRepository.save(user)
```

Dla wzorca *Data Mapper* istnieje również możliwość tworzenia niestandardowych metod na repozytorium. Pozwala to na enkapsulację złożonej logiki zapytań SQL w dedykowanych klasach, co sprzyja zachowaniu czystości kodu. Zamiast rozpraszać logikę bazodanową w kontrolerach czy serwisach, programista może zdefiniować specyficzne metody operujące na encjach bezpośrednio w rozszerzonym repozytorium, wykorzystując do tego celu wbudowany *query builder*. Przykład implementacji niestandardowego repozytorium został przedstawiony w Listing 17.

Listing 17 Implementacja niestandardowego repozytorium dla TypeORM

```
export const UserRepository = dataSource.getRepository(User).extend({
    findByName(firstName: string, lastName: string) {
        return this.createQueryBuilder("user")
            .where("user.firstName = :firstName", { firstName })
            .andWhere("user.lastName = :lastName", { lastName })
            .getMany()
    }})
UserRepository.findByName("Jan", "Kowalski")
```

3.4.3 Perspektywy rozwoju

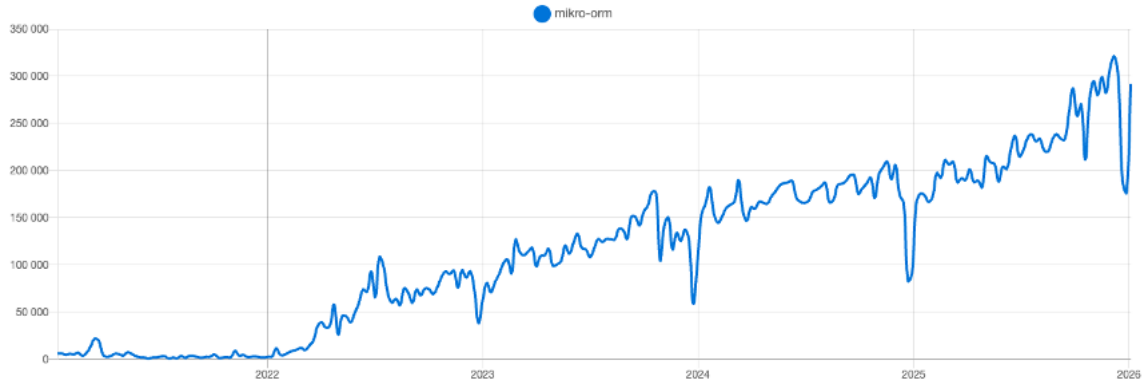
Historia rozwoju TypeORM jest istotna dla zrozumienia jego obecnej sytuacji. Projekt został stworzony i przez wiele lat był prowadzony przez głównego autora, Umeda Khudoiberdieva. W pewnym momencie (w wersji 0.2.x) jego zaangażowanie w projekt znacznie zmalało, co doprowadziło do okresu stagnacji. Mimo sporej popularności biblioteki, kluczowe błędy nie były naprawiane, a pull requesty czekały na zatwierdzenie miesiącami. Wywołało to dużą frustrację w społeczności, która była mocno zależna od narzędzia, zwłaszcza w ekosystemie NestJS [46].

Sytuacja uległa zmianie, gdy projekt został formalnie przejęty przez nowy zespół programistów. Przejęli oni pełną odpowiedzialność za rozwój, co doprowadziło do wydania wersji 0.3.x i wznowienia aktywnego rozwoju. W wydanym przez nowych właścicieli oświadczeniu jest wspomniane, że aby zagwarantować długoterminową stabilność, planowane jest utworzenie fundacji non-profit, zarządzanej przez kluczowe firmy i kontrybutorów [47]. Głównym celem jest zwiększenie sponsoringu od firm korzystających z TypeORM, aby sfinansować zatrudnienie dwóch pełnoetatowych deweloperów, co ma zapewnić ciągłość i profesjonalizację prac nad projektem. Obecnie projekt jest stabilny i aktywnie rozwijany.

3.5 MikroORM

MikroORM jest raczej niszowym rozwiązaniem, w porównaniu do "wielkiej trójki" (Prisma, Sequelize, TypeORM). Jego popularność mierzona w liczbie tygodniowych pobrań z npm, przedstawiona na Wykres 7, w porównaniu z pozostałymi ORM-ami, jest niewielka.

Jego autorem jest Martin Adámek, czeski programista. Projekt powstał z jego osobistej potrzeby i frustracji brakiem w ekosystemie Node.js ORM-a, który w poprawnie implementowałby zaawansowane wzorce znane z innych, bardziej dojrzałych platform. Autor, mając duże doświadczenie z ekosystemów PHP (Doctrine) i Javy (Hibernate), czerpał bezpośrednie inspiracje z tych dwóch narzędzi [48]. Jego celem było przeniesienie na grunt TypeScriptu koncepcji, takich jak wzorzec *Unit of Work* oraz *Identity Map*, których, jego zdaniem, brakowało w istniejących wówczas narzędziach.



Wykres 7 Popularność MikroORM według liczby tygodniowych pobrań z npm. Źródło: [49]

3.5.1 Obsługiwane bazy danych

MikroORM, podobnie jak Prisma i TypeORM oferuje wsparcie nie tylko relacyjnych baz danych. Na liście wspieranych systemów, znajdują się [50]:

- PostgreSQL,
- MySQL,
- MariaDB,
- SQLite,
- MS SQL Server,
- MongoDB.

3.5.2 Architektura

MikroORM, podobnie jak inne ORM-y, posiada elastyczną architekturę opartą na systemie sterowników. Oznacza to, że rdzeń biblioteki (`@mikro-orm/core`) jest niezależny od konkretnej bazy danych, a wsparcie dla danego systemu jest dodawane poprzez instalację odpowiedniego pakietu sterownika (np. `@mikro-orm/postgresql`). Konfiguracja połączenia odbywa się podczas inicjalizacji ORM-a za pomocą metody `MikroORM.init()`, gdzie w opcjach przekazuje się dane dostępowe oraz ścieżkę do encji. Po uzyskaniu instancji `orm`, cała interakcja z bazą danych odbywa się głównie poprzez *EntityManager* (`em`), który zarządza encjami, śledzi zmiany i wykonuje zapytania.

Listing 18 Definicja modelu w MikroORM wraz z przykładem użycia

```
@Entity()
export class User {
    @PrimaryKey()
    id: number;

    @Property({ length: 50 })
    @Unique()
    username: string;

    setUsername(name: string) {
        this.username = name;
    }
}

// Przykład użycia
const user = await em.findOne(User, { id: 1 });
if (user) {
  user.lastName = 'Nowak';
}
await em.flush()
```

Definiowanie modeli w MikroORM opiera się na wykorzystaniu dekoratorów, co na pierwszy rzut oka przypomina TypeORM. Listing 18 przedstawia definicję przykładowego modelu. Model to klasa TypeScript ozdobiona dekoratorami takimi jak *`@Entity()`*, *`@Property()`* czy *`@ManyToOne()`*, które opisują mapowanie na strukturę bazy danych.

Kluczowa różnica polega jednak na tym, że modele w MikroORM są "zarządzane" przez *EntityManager* i działają w oparciu o wzorzec *Unit of Work*. Oznacza to, że instancje encji są pasywnymi kontenerami na dane, które nie wiedzą nic o bazie danych, a wszystkie zmiany na nich są automatycznie śledzone. Zamiast wywoływać metodę *`.save()`* na każdym obiekcie, programista na końcu operacji wywołuje jedną, globalną metodę *`em.flush()`*, która utrwala wszystkie zmiany w jednej transakcji, jak zostało to pokazane w przykładzie użycia w Listing 18.

MikroOrm implementuje również wzorzec *Identity Map* [51]. Mechanizm ten jest zarządzany przez *EntityManager* i działa jak podręczna pamięć cache na poziomie pojedynczego cyklu życia `em` (zazwyczaj jednego zapytania HTTP), umożliwiając porównywanie obiektów encji, co z kolei pozwala ORM-owi na grupowanie operacji (*batching*) do bazy danych.

Listing 19 przedstawia realizację tego wzorca. Przy pierwszym wczytaniu encji z bazy danych, MikroORM tworzy jej instancję i zapisuje ją w swojej wewnętrznej mapie. Każde kolejne zapytanie o ten sam rekord (po tym samym kluczu głównym) w ramach tego samego `em` nie wykonuje już ponownego zapytania do bazy, lecz natychmiast zwraca referencję do tego samego, już istniejącego obiektu w pamięci. To gwarantuje, że w całej aplikacji istnieje tylko jedna, spójna reprezentacja danego wiersza z bazy, co eliminuje ryzyko pracy na nieaktualnych lub sprzecznych danych i poprawia wydajność przez unikanie zbędnych zapytań.

Listing 19 Przykład działania Identity Map w MikroORM

```
1. const userRepository = em.getRepository(User);
2. const user1 = await userRepository.findOne({ id: 1 });
3. // Drugie zapytanie o tego samego użytkownika (nie zostanie wysłane do bazy,
zwróci obiekt z mapy)
4. const user2 = await userRepository.findOne({ id: 1 });
5.
6. // Porównanie referencji - wynik będzie `true`, bo to ten sam obiekt w pamięci
7. console.log(user1 === user2); // --> true
```

3.5.3 Perspektywy rozwoju

Ostatnia znacząca aktualizacja MikroORM, czyli wersja v6, została wydana w styczniu 2024 roku, przynosząc odświeżenie systemu typowania oraz poprawę wydajności [52]. Od tego czasu projekt jest rozwijany w modelu ciągłym – najnowsze wydania stabilne z linii 6.6.x (ze stycznia 2026 r.) wprowadzają głównie poprawki błędów oraz ulepszenia w generatorze encji.

Trwają również prace nad kolejną wersją v7, która ma przynieść szereg fundamentalnych zmian [53]. Należą do nich m.in. wsparcie dla Oracle DB, ulepszenie obsługi TypeScripta, czy usunięcie zewnętrznych zależności z Node.js API. Wstępnie planowane jest wydanie tej wersji w pierwszym kwartale 2026 roku.

Perspektywy rozwoju MikroORM wskazują na dążenie do pełnej niezależności technologicznej co ma uczynić go rozwiązaniem wysoce wydajnym i uniwersalnym. Wyróżnia się na tle pozostałych technologii implementacją wzorca *Unit of Work*. Mimo zaawansowanych funkcji technicznych, MikroORM pozostaje rozwiązaniem niszowym w porównaniu do liderów rynkowych, co jednak wraz z dalszym rozwojem może się zmienić.

4. Założenia badań porównawczych

W tym rozdziale przedstawiono założenia badań porównawczych dotyczących wybranych technologii mapowania obiektowo-relacyjnego, stosowanych w środowisku Node.js. Określono ramy metodologiczne przeprowadzonych testów, w tym uzasadnienie doboru analizowanych rozwiązań, zakresu porównań, przyjętych kryteriów oceny oraz zastosowanych scenariuszy testowych.

Poniższe podrozdziały szczegółowo omawiają cel i charakter badań, zakres porównania oraz przyjęte metody oceny analizowanych technologii.

4.1 Cel i charakter badań

Celem przeprowadzonych badań było porównanie wybranych technologii mapowania obiektowo-relacyjnego (ORM) przeznaczonych dla środowiska Node.js pod kątem ich przydatności w tworzeniu aplikacji. Analiza koncentrowała się w szczególności na aspektach wydajnościowych oraz funkcjonalnych, które mają istotny wpływ na jakość, skalowalność i utrzymanie systemów informatycznych.

Badania miały charakter eksperymentalny i porównawczy. Zastosowana metodyka polegała na przeprowadzeniu serii testów wydajnościowych, na tych samych zbiorach danych, dla każdej z analizowanych technologii. Umożliwiło to uzyskanie obiektywnych i porównywalnych wyników.

Zakres badań obejmował typowe scenariusze wykorzystywane w aplikacjach internetowych, takie jak operacje CRUD, obsługa relacji między encjami, realizacja transakcji oraz przetwarzanie dużych zbiorów danych. Celem nie było jedynie wskazanie najszybszego rozwiązania, lecz także ocena kompromisów pomiędzy wydajnością, elastycznością konfiguracji oraz ergonomią pracy programisty.

4.2 Braki w istniejących opracowaniach

Analiza dostępnych opracowań porównujących technologie ORM dla środowiska Node.js pozwoliła wskazać szereg braków, które uzasadniają potrzebę przeprowadzenia własnych badań.

Porównanie autorstwa Emanuela Casco z 2019 roku [17], obejmujące narzędzia Sequelize, Knex, TypeORM oraz Objection.js, zostało przeprowadzone wyłącznie z wykorzystaniem bazy PostgreSQL, bez uwzględnienia innych popularnych systemów zarządzania bazami danych, takich jak MySQL. Ogranicza to możliwość generalizacji wyników na inne środowiska produkcyjne. Dodatkowo, ze względu na rok publikacji, opracowanie to nie uwzględnia technologii Prisma, która w tamtym czasie dopiero rozpoczynała rozwój, a obecnie należy do najczęściej wybieranych ORM-ów dla języka TypeScript.

Istotnym ograniczeniem pracy Casco jest również sposób prezentacji wyników. Autor ograniczył się głównie do przedstawienia surowych danych z testów wydajnościowych, bez ich pogłębionej interpretacji oraz bez odniesienia do praktycznych konsekwencji dla projektów informatycznych. Brakuje refleksji nad tym, w jakich scenariuszach konkretne narzędzie może sprawdzić się najlepiej oraz jakie kompromisy wiążą się z jego wyborem. Ponadto zakres testów został ograniczony do prostych operacji oraz niewielkich zbiorów danych, co nie odzwierciedla warunków typowych dla aplikacji produkcyjnych, w których często występują złożone zapytania oraz duże wolumeny danych.

Kolejnym istotnym opracowaniem jest porównanie autorstwa Romana Kushyna z 2022 roku [19], obejmujące siedem narzędzi ORM: TypeORM, MikroORM, Prismę, Objection.js, Knex oraz OrchidORM. Choć badanie to wyróżnia się szerokim zakresem analizowanych technologii, jego ograniczeniem jest zakres przeprowadzonych testów, który skupia się wyłącznie na operacjach odczytu

danych. Brak uwzględnienia operacji zapisu czy transakcji, znacząco zawęża obraz rzeczywistej wydajności i użyteczności badanych narzędzi.

Podsumowując, istniejące opracowania charakteryzują się ograniczeniami w zakresie aktualności, kompletności testowanych technologii, różnorodności scenariuszy oraz analizy wyników. W szczególności brakuje badań uwzględniających współczesne ORM-y, takie jak Prisma, testów na dużych zbiorach danych oraz analiz złożonych operacji bazodanowych. Wskazane luki stanowią bezpośrednią motywację do przeprowadzenia własnych badań, których celem jest dostarczenie bardziej aktualnego, kompleksowego i praktycznie użytecznego porównania technologii ORM w środowisku Node.js.

4.3 Kryteria doboru badanych technologii ORM

Dobór technologii poddanych analizie został przeprowadzony w sposób celowy, z uwzględnieniem ich znaczenia w praktyce. Celem było wybranie reprezentatywnej grupy narzędzi, które odzwierciedlają różne podejścia do mapowania obiektowo-relacyjnego oraz są powszechnie stosowane w projektach komercyjnych.

Kryteria doboru badanych technologii:

- **popularność technologii** - rozumiana jako stopień rozpowszechnienia w społeczności programistycznej. Uwzględniono takie czynniki jak aktywność repozytoriów w serwisach kontroli wersji, częstotliwość aktualizacji oraz obecność w dokumentacji branżowej i materiałach edukacyjnych.
- **dojrzałość i stabilność projektów** - wybrane technologie charakteryzują się wieloletnim rozwojem, ustabilizowanymi interfejsami programistycznymi oraz udokumentowanym zastosowaniem w środowiskach produkcyjnych, co zwiększa wiarygodność przeprowadzonych porównań.
- **jakość wsparcia dla TypeScript** - stanowi on obecnie standard w tworzeniu aplikacji w ekosystemie Node.js. Brano pod uwagę stopień integracji z systemem typów oraz możliwości statycznej weryfikacji poprawności kodu.
- **kompatybilność z relacyjnymi systemami zarządzania bazami danych** - w szczególności z bazami PostgreSQL i MySQL, które zostały wykorzystane w badaniach. Wybrane technologie muszą oferować pełne wsparcie dla operacji CRUD, transakcji oraz relacji, co jest niezbędne do realizacji zaplanowanych scenariuszy testowych.
- **dostępność dokumentacji oraz wsparcia społeczności** - narzędzia posiadające rozbudowaną dokumentację oraz liczne przykłady zastosowań ułatwiają zarówno proces wdrożenia, jak i interpretację wyników badań.

Na podstawie powyższych kryteriów do badań wybrano technologie ORM, które reprezentują zróżnicowane podejścia projektowe, a jednocześnie spełniają wymagania funkcjonalne i jakościowe niezbędne do przeprowadzenia rzetelnej analizy porównawczej.

4.4 Zakres porównania

Zakres przeprowadzonych badań został celowo ograniczony do obszarów, które mają kluczowe znaczenie dla projektowania i eksploatacji aplikacji wykorzystujących relacyjne bazy danych. Celem było skoncentrowanie się na aspektach praktycznych, istotnych z punktu widzenia programistów oraz architektów systemów informatycznych, przy jednoczesnym zachowaniu wykonalności i rzetelności badań.

W ramach zakresu funkcjonalnego, analizie poddano podstawowe oraz zaawansowane mechanizmy oferowane przez technologie ORM, niezbędne do realizacji typowych operacji biznesowych. W szczególności uwzględniono:

- obsługę operacji CRUD (tworzenie, odczyt, aktualizacja, usuwanie danych),
- zarządzanie relacjami między encjami, w tym relacjami typu jeden-do-wielu oraz wiele-do-wielu,
- realizację transakcji oraz mechanizmy zapewniania spójności danych,
- implementację oraz wsparcie dla dostępnych wzorców projektowych,
- możliwości walidacji danych na poziomie modeli.

W przypadku technologii ORM, które umożliwiają stosowanie więcej niż jednego wzorca (np. Active Record i Data Mapper), przeprowadzono ich równoległą analizę i porównanie w celu oceny wpływu przyjętego podejścia na czytelność kodu, łatwość utrzymania oraz wydajność.

W zakresie niefunkcjonalnym, badania koncentrowały się na aspektach wydajnościowych oraz jakościowych, które mają bezpośredni wpływ na zachowanie aplikacji w środowisku produkcyjnym. W szczególności analizie poddano:

- czas wykonania operacji bazodanowych,
- zużycie zasobów systemowych, w tym pamięci operacyjnej oraz procesora,
- wpływ rozmiaru zbiorów danych na wydajność operacji bazodanowych.

Zakres niefunkcjonalny nie obejmuje natomiast zagadnień związanych z bezpieczeństwem, odpornością na awarie oraz integracją z zewnętrznymi systemami, które stanowią odrębne obszary badawcze.

4.5 Sposoby oceniania

Ocena analizowanych technologii ORM została przeprowadzona na podstawie zestawu obiektywnych i mierzalnych kryteriów, które umożliwiają rzetelne porównanie uzyskanych wyników. Przyjęte sposoby oceniania zostały dostosowane do zakresu funkcjonalnego oraz niefunkcjonalnego badań.

Podstawowym kryterium oceny była wydajność operacji bazodanowych, mierzona czasem wykonania poszczególnych scenariuszy testowych. Dla każdej operacji testowej pomiar czasu był wykonywany trzykrotnie, a następnie wyznaczano średnią arytmetyczną uzyskanych wartości. Takie podejście pozwalało na ograniczenie wpływu czynników losowych oraz zwiększenie wiarygodności wyników.

Drugim kryterium była analiza zużycia zasobów systemowych, w szczególności pamięci operacyjnej oraz obciążenia procesora. Pomiar ten umożliwiał ocenę narzutu, jaki dana technologia ORM wprowadza w stosunku do bezpośredniego dostępu do bazy danych.

Kolejnym elementem oceny był wpływ rozmiaru zbiorów danych na wydajność wykonywanych operacji. W tym celu testy były przeprowadzane dla 3 zbiorów danych o zróżnicowanej liczności rekordów, co pozwalało na obserwację zmian czasu wykonania operacji wraz ze wzrostem wolumenu danych.

Dodatkowym, jakościowym kryterium oceny była ergonomia pracy programisty, obejmująca czytelność interfejsu programistycznego, spójność dokumentacji oraz łatwość implementacji typowych scenariuszy biznesowych. Choć kryterium to ma charakter subiektywny, jego uwzględnienie pozwala na pełniejszą ocenę praktycznej przydatności analizowanych narzędzi.

4.6 Scenariusze testowe

Scenariusze testowe zostały zaprojektowane w oparciu o model danych odzwierciedlający typową strukturę systemu sprzedażowego (*e-commerce*). Model obejmował encje takie jak: adres, klient, produkt, kategoria, zamówienie, pozycja zamówienia oraz płatność, wraz z relacjami między nimi. Taki schemat umożliwiał testowanie zarówno prostych operacji, jak i bardziej złożonych zapytań obejmujących wiele tabel oraz relacji.

Celem doboru scenariuszy było odwzorowanie rzeczywistych przypadków użycia spotykanych w codziennej działalności, w szczególności w systemach obsługujących sprzedaż internetową. Scenariusze zostały podzielone na cztery główne kategorie: operacje odczytu danych, operacje wstawiania danych, zapytania agregujące oraz operacje transakcyjne.

4.6.1 Operacje odczytu danych

Scenariusze odczytu danych zostały zaprojektowane w celu oceny efektywności wykonywania zapytań o różnym stopniu złożoności oraz liczbie zaangażowanych relacji.

Scenariusze:

- pobranie wszystkich zamówień przypisanych do danego klienta - operacja testowała wydajność zapytań filtrujących oraz zdolność ORM do efektywnego mapowania wyników na obiekty.
- pobranie pojedynczego zamówienia wraz z powiązanymi pozycjami zamówienia oraz danymi produktów - umożliwiał ocenę mechanizmów ładowania relacji oraz ich wpływu na liczbę zapytań oraz czas odpowiedzi.
- pobranie wszystkich zamówień z określonego zakresu dat wraz z danymi klientów, adresami oraz informacjami o płatnościach - zapytanie o wysokim stopniu złożoności, obejmujące wiele tabel oraz relacji, odzwierciedlające typowe raportowe zapytania biznesowe.

4.6.2 Operacje wstawiania danych

Kategoria obejmowała testy mające na celu ocenę wydajności związanej z tworzeniem nowych rekordów w bazie danych, w tym również operacji obejmujących relacje pomiędzy encjami.

Scenariusze:

- wstawienie pojedynczego rekordu klienta wraz z powiązanym adresem - operacja obejmowała zapis danych do dwóch tabel oraz utworzenie relacji pomiędzy nimi, co pozwoliło na ocenę wydajności ORM w przypadku zależnych operacji zapisu.
- wstawienie nowego produktu wraz z przypisaniem go do kategorii - operacja testowała obsługę relacji typu wiele-do-wielu, realizowanej poprzez tabelę pośrednią, co stanowi istotny element wielu modeli domenowych.

4.6.3 Zapytania agregujące

W tej kategorii znajdowały się testy dotyczące operacji agregujących, które są powszechnie wykorzystywane w analizie danych biznesowych oraz raportowaniu.

Scenariusze:

- liczba zamówień złożonych w danym zakresie dat - pozwalał na ocenę wydajności prostych agregacji oraz ich obsługi przez poszczególne technologie ORM.
- obliczenie całkowitej wartości sprzedaży w danym zakresie dat - wymagał zastosowania funkcji agregujących oraz odpowiedniego filtrowania danych czasowych.

4.6.4 Transakcje

Ostatnia grupa scenariuszy obejmowała operacje realizowane w ramach transakcji, których celem jest zapewnienie spójności danych w przypadku złożonych operacji zapisu. Scenariusze te odzwierciedlają rzeczywiste procesy biznesowe, w których wiele powiązanych operacji musi zostać wykonanych jako niepodzielna całość.

Scenariusz transakcyjny polegał na utworzeniu nowego zamówienia wraz z wieloma pozycjami, adresem i płatności przypisanym do zamówienia oraz jednoczesnej aktualizacji stanów magazynowych produktów. Operacja ta obejmowała wiele zapisów oraz aktualizacji w różnych tabelach, które muszą zostać wykonane atomowo. Test ten umożliwił ocenę wsparcia dla transakcji, obsługi błędów oraz wpływu transakcji na wydajność systemu.

5. Projekt aplikacji testowej

Niniejszy rozdział poświęcono przedstawieniu projektu aplikacji testowej, która została opracowana w celu realizacji badań porównawczych opisanych w poprzednich rozdziałach. Projekt aplikacji stanowił pomost pomiędzy założeniami badawczymi a ich praktyczną realizacją w postaci implementacji i testów wydajnościowych.

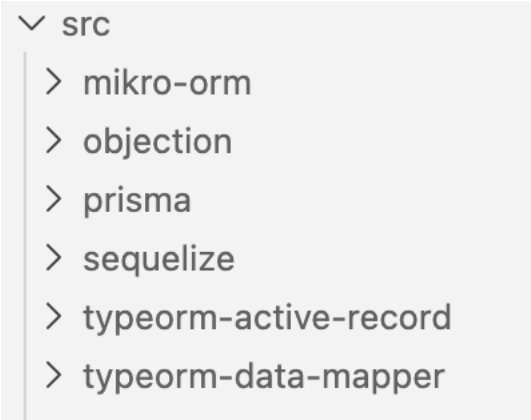
W poniższych podrozdziałach omówiono architekturę aplikacji, strukturę warstwową, model danych oraz sposób organizacji komponentów odpowiedzialnych za realizację scenariuszy testowych. Szczególną uwagę poświęcono zapewnieniu spójności, modularności oraz możliwości powtarzalnego uruchamiania eksperymentów w jednakowych warunkach dla wszystkich analizowanych technologii ORM.

5.1 Architektura aplikacji

Aplikacja testowa została zaimplementowana z wykorzystaniem frameworka Express.js, który stanowi elastyczny fundament do budowy aplikacji backendowych w środowisku Node.js. Wybór tego narzędzia wynikał z jego popularności, dojrzałości oraz minimalnego narzutu architektonicznego, co pozwalało skupić się na badaniu właściwości poszczególnych technologii ORM, a nie na specyfice frameworka aplikacyjnego.

Architektura aplikacji miała charakter modułowy i warstwowy. Została zaprojektowana w sposób umożliwiający oddzielenie logiki biznesowej, warstwy dostępu do danych oraz warstwy odpowiedzialnej za obsługę żądań HTTP. Taki podział sprzyja czytelności kodu, ułatwia jego utrzymanie oraz umożliwia niezależne rozwijanie i testowanie poszczególnych komponentów.

Kluczowym założeniem było zapewnienie izolacji poszczególnych technologii ORM w celu zagwarantowania porównywalności wyników badań. W tym celu każdy ORM został umieszczony w osobnym module, zorganizowanym pod odrębną ścieżką w strukturze projektu. Rysunek 14 przedstawia ogólną strukturę aplikacji testowej.



Rysunek 14 Struktura modułów aplikacji testowej. Źródło: opracowanie własne

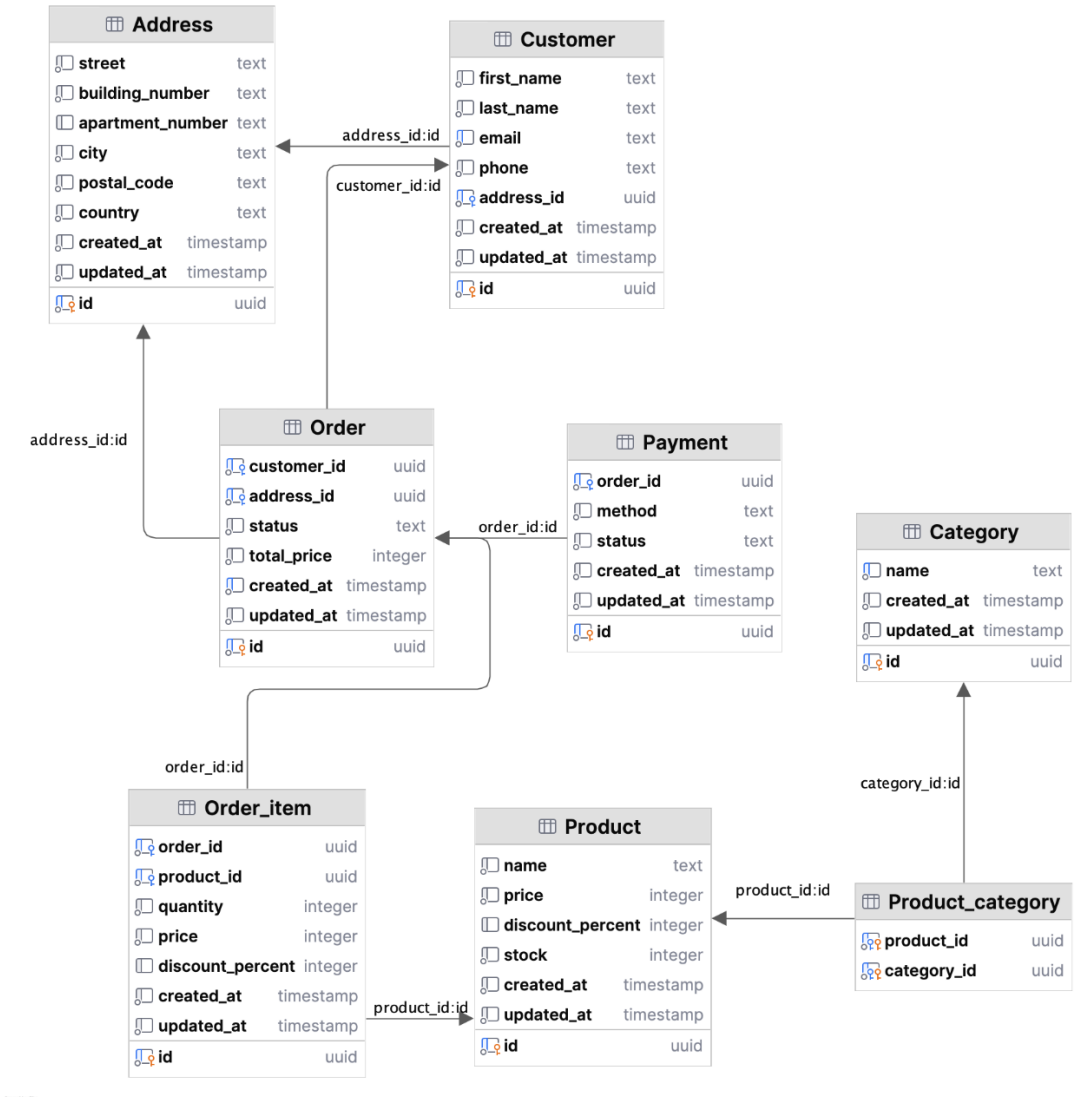
Każdy z modułów implementował identyczny zestaw operacji biznesowych oraz scenariuszy testowych. Dzięki temu możliwe było przeprowadzanie testów w jednakowych warunkach, przy jednoczesnym zachowaniu niezależności implementacyjnej poszczególnych rozwiązań. Warstwa obsługi żądań HTTP została zaprojektowana w sposób umożliwiający wywoływanie tych samych scenariuszy testowych dla każdego ORM. Osiągnięto to poprzez zastosowanie jednolitego interfejsu API oraz wspólnej logiki sterującej przebiegiem testów. Takie podejście pozwoliło na automatyzację eksperymentów oraz ograniczenie wpływu czynników zewnętrznych na uzyskiwane wyniki.

Zaprojektowana architektura wspierała również rozszerzalność aplikacji. Dodanie kolejnej technologii ORM wymagało jedynie utworzenia nowego modułu implementującego ten sam zestaw metod, bez konieczności modyfikowania pozostałych elementów systemu. Rozwiązanie to sprzyja dalszym badaniom oraz umożliwia łatwe porównywanie kolejnych narzędzi w przyszłości.

5.2 Model danych

Model danych zastosowany w aplikacji testowej został zaprojektowany w oparciu o typową domenę systemu sprzedażowego (*e-commerce*), co pozwoliło na realistyczne odwzorowanie relacji oraz operacji spotykanych w rzeczywistych aplikacjach, np. sklepach internetowych. Przyjęta struktura umożliwiła testowanie zarówno prostych operacji CRUD, jak i złożonych zapytań obejmujących wiele tabel, relacji oraz operacji agregujących i transakcyjnych, które mają zastosowanie w codziennym funkcjonowaniu aplikacji.

Projekt modelu danych został podporządkowany dwóm głównym celom: zapewnieniu wystarczającej złożoności do przeprowadzenia rzetelnych testów wydajnościowych, a także zachowaniu czytelności oraz jednoznaczności relacji pomiędzy encjami.



Rysunek 15 Model danych dla aplikacji testowej. Źródło: opracowanie własne

Rysunek 15 przedstawia diagram ERD modelu danych. Składa się on z następujących głównych encji: `Address`, `Customer`, `Product`, `Category`, `Order`, `Order_item` oraz `Payment`. Relacje pomiędzy encjami odzwierciedlają rzeczywiste zależności biznesowe, takie jak powiązanie klienta z adresem, zamówienia z klientem, pozycjami zamówienia i płatnością, czy produktów z kategoriami.

Powiązania zostały zaprojektowane w sposób umożliwiający testowanie różnych typów relacji:

- relacji jeden-do-jeden – np. zamówienie – adres,
- relacji jeden-do-wielu – np. klient – zamówienie,
- relacji wiele-do-wielu – np. produkt – kategoria.

Wszystkie encje wykorzystywały klucze główne typu UUID, co zapewniało jednoznaczną identyfikację rekordów oraz ułatwiło generowanie danych testowych bez ryzyka kolizji identyfikatorów.

Każda z encji zawierała również pola `created_at` oraz `updated_at`, które umożliwiają rejestrowanie momentu utworzenia i modyfikacji rekordu. Obecność tych pól ma istotne znaczenie w kontekście zapytań filtrujących oraz agregujących, w szczególności tych opartych na zakresie dat. Znaczniki czasowe umożliwiły również testowanie mechanizmów automatycznego zarządzania polami czasowymi oferowanych przez poszczególne technologie ORM.

5.2.1 Encja adres

Encja `Address` reprezentuje dane adresowe wykorzystywane zarówno przez klientów, jak i zamówienia. Zawiera informacje takie jak ulica, numer budynku, numer lokalu, miasto, kod pocztowy oraz kraj.

W zaprojektowanym modelu danych przyjęto relację 1:1 pomiędzy encją adresu a encjami `Customer` oraz `Order`, co oznacza, że każdy klient oraz każde zamówienie posiadało dokładnie jeden przypisany adres.

5.2.2 Encja klient

Encja `Customer` przechowuje dane identyfikacyjne klientów, takie jak imię, nazwisko, adres e-mail oraz numer telefonu. Każdy klient został powiązany z jednym adresem, co zrealizowano poprzez klucz obcy `address_id`. Encja klienta stanowiła punkt odniesienia dla dalszych relacji, w szczególności z encją Order.

5.2.3 Encja produkt

Encja `Product` reprezentuje towary dostępne w systemie sprzedażowym. Zawiera informacje o nazwie produktu, cenie, poziomie rabatu oraz stanie magazynowym. Dane te są istotne zarówno dla operacji odczytu, jak i testów transakcyjnych, w których aktualizowany jest stan magazynowy, podczas tworzenia nowego zamówienia.

Obecność pól związanych z ceną i rabatem umożliwiła również testowanie zapytań agregujących oraz obliczeń biznesowych.

5.2.4 Encja kategoria

Encja `Category` umożliwia grupowanie produktów w logiczne kategorie. Relacja pomiędzy produktami, a kategoriami została zrealizowana jako relacja wiele-do-wielu, przy użyciu tabeli pośredniej `Product_Category`, zawierającej klucze obce `product_id` oraz `category_id`. Taki sposób modelowania umożliwia przypisanie jednego produktu do wielu kategorii oraz odwrotnie.

5.2.5 Encja zamówienie

Encja `Order` reprezentuje zamówienia składane przez klientów. Zawiera informacje o kliencie, adresie dostawy, statusie zamówienia oraz łącznej wartości. Każde zamówienie powiązano zarówno z encją `Customer`, `Address`, jak i `Payment`.

Encja ta stanowiła centralny element modelu danych, wokół którego zrealizowano najbardziej złożone scenariusze testowe, w tym operacje odczytu z wieloma złączeniami oraz transakcje obejmujące wiele tabel.

5.2.6 Encja pozycja zamówienia

Encja `Order_Item` reprezentuje pojedyncze pozycje zamówienia i pełni rolę łącznika pomiędzy encjami `Order` a `Product`. Każda pozycja zawiera informację o liczbie zamówionych sztuk, cenie jednostkowej oraz zastosowanym rabacie. Pozwoliło to na przechowywanie danych historycznych, takich jak cena i rabat w momencie zakupu.

W zaprojektowanym modelu danych przyjęto relację jeden-do-wielu pomiędzy encją `Order` a encją `Order_Item`, co oznacza, że jedno zamówienie może zawierać wiele pozycji, natomiast każda pozycja zamówienia była zawsze przypisana do dokładnie jednego zamówienia.

5.2.7 Encja płatność

Encja `Payment` przechowuje informacje dotyczące płatności za zamówienie, w tym metodę płatności oraz jej status realizacji.

5.2.8 Indeksy i optymalizacja struktury bazy danych

W celu zapewnienia porównywalnych oraz wiarygodnych wyników testów wydajnościowych, w projekcie aplikacji testowej zastosowano zestaw indeksów, których zadaniem było odzwierciedlenie typowych praktyk optymalizacyjnych stosowanych w rzeczywistych systemach bazodanowych. Indeksy te zostały zdefiniowane w sposób jednolity dla wszystkich badanych technologii ORM, co pozwoliło wyeliminować wpływ różnic w strukturze bazy danych na uzyskiwane wyniki.

Zastosowane indeksy obejmowały przede wszystkim klucze obce oraz kolumny wykorzystywane w filtrach zapytań, sortowaniu oraz operacjach agregujących. Takie podejście umożliwiło przyspieszenie najczęściej wykonywanych operacji odczytu, a jednocześnie zachowało realistyczny charakter środowiska testowego.

Indeksy zastosowano dla poszczególnych kolumn:

- `address_id` w tabeli `Customer` - wprowadzony w celu usprawnienia zapytań łączących dane klientów z ich adresami. Ponieważ relacja pomiędzy encjami `Customer` i `Address` ma charakter jeden-do-jednego, indeks ten umożliwia szybkie wyszukiwanie powiązanego rekordu adresowego dla danego klienta.
- `created_at` w tabeli `Order` - użyty dla zapytań agregujących, w których dane były filtrowane według zakresu dat np. utworzenia zamówień.
- `customer_id` oraz `address_id` w tabeli `Order` - umożliwia efektywne wykonywanie zapytań pobierających zamówienia konkretnego klienta oraz zapytań łączących zamówienia z danymi adresowymi.
- `order_id` w tabeli `Order_item` - istotny dla zapytań pobierających wszystkie pozycje danego zamówienia.
- `product_id` w tabeli `Order_item` – umożliwia sprawne wykonywanie zapytań związanych z analizą sprzedaży poszczególnych produktów oraz aktualizacją stanów magazynowych.

- `order_id` w tabeli `Payment` - pozwala na szybkie łączenie danych dotyczących płatności z odpowiadającymi im zamówieniami. Jest to szczególnie istotne w scenariuszach obejmujących złożone zapytania odczytu, pobierające jednocześnie informacje o zamówieniach, klientach, adresach oraz płatnościach.

Wszystkie wymienione indeksy zostały dobrane w oparciu o analizę scenariuszy testowych oraz typowych wzorców zapytań występujących w aplikacjach sprzedażowych.

W modelu zastosowano również ograniczenia unikalności, które zwiększają jakość danych oraz eliminują możliwość występowania duplikatów:

- `email` w tabeli `Customer` - zapewnia jednoznaczną identyfikację klienta na podstawie adresu e-mail,
- `name` w tabeli `Category` - zapobiega tworzeniu wielu kategorii o tej samej nazwie.

5.3 Zbiory danych testowych

W celu przeprowadzenia badań wydajnościowych, w aplikacji testowej wykorzystano trzy zestawy danych o zróżnicowanej skali: małe, średnie oraz duże. Zastosowanie wielu poziomów wielkości zbiorów danych umożliwiło ocenę zachowania badanych technologii ORM zarówno w warunkach testowych, jak i w scenariuszach zbliżonych do rzeczywistych środowisk produkcyjnych.

Zbiory danych zostały zaprojektowane w taki sposób, aby zachować spójność ze scenariuszami testowymi oraz strukturą modelu danych opisanymi w poprzednich rozdziałach. We wszystkich wariantach aplikacji testowej wykorzystano identyczne zbiory danych, co zapewniło porównywalność wyników pomiędzy badanymi technologiami.

Każdy zestaw danych został zaprojektowany w sposób odzwierciedlający realistyczne proporcje pomiędzy poszczególnymi encjami. Dzięki temu możliwe było nie tylko badanie wydajności poszczególnych operacji, lecz także analiza skalowalności rozwiązań w miarę wzrostu liczby rekordów w bazie danych. Poszczególne wielkości tabel w każdej z baz zostały przedstawione w Tabela 3.

Tabela 3 Zestawienie ilości rekordów w bazach danych. Źródło: opracowanie własne

	Ilość rekordów w tabeli						
Wielkość bazy	Category	Product	Customer	Adress	Order	Order_Item	Payment
Mała	15	150	100	300	200	~1 100	200
Średnia	500	2 500	50 000	150 000	100 000	~550 000	100 000
Duża	2 500	250 000	500 000	1 500 000	1 000 000	~5 500 000	1 000 000

5.3.1 Małe zbiory danych

Małe bazy danych zostały wykorzystane w celu weryfikacji poprawności działania aplikacji oraz sprawdzenia poprawności implementacji scenariuszy testowych przy niewielkiej liczbie rekordów. Ten poziom wielkości danych umożliwiał szybkie uruchamianie testów, łatwą diagnostykę błędów oraz wstępną ocenę zachowania poszczególnych technologii ORM.

Zastosowane proporcje odzwierciedlają typową strukturę niewielkiej aplikacji sprzedażowej lub środowiska testowego wykorzystywanego na wczesnym etapie rozwoju systemu.

5.3.2 Średnie zbiory danych

Średnie bazy danych zostały zaprojektowane w celu symulacji realistycznego obciążenia aplikacji, odpowiadającego rzeczywistym warunkom eksploatacyjnym spotykanym w niewielkich,

produkcyjnych systemach sprzedażowych. Ten poziom danych umożliwiał ocenę wydajności zapytań oraz operacji transakcyjnych w warunkach zwiększonej liczby rekordów, przy zachowaniu rozsądnego czasu wykonywania testów.

Zbiory te umożliwiały przeprowadzenie wiarygodnych testów wydajnościowych, zachowując przy tym akceptowalny czas trwania eksperymentów.

5.3.3 Duże zbiory danych

Duże bazy danych zostały wykorzystane w celu oceny zachowania aplikacji oraz technologii ORM w warunkach bardzo dużego obciążenia, zbliżonych do środowisk o intensywnym wzroście liczby użytkowników i transakcji. Ten poziom danych umożliwiał analizę skalowalności rozwiązań oraz identyfikację potencjalnych ograniczeń wydajnościowych.

Tak duże wolumeny danych umożliwiły testowanie zapytań oraz operacji w warunkach ekstremalnych, co pozwoliło na ocenę stabilności i wydajności badanych narzędzi przy znacznym obciążeniu warstwy danych.

5.3.4 Ograniczenia zastosowanych zbiorów danych

W przypadku dużych zbiorów danych, w zastosowaniach produkcyjnych standardowym rozwiązaniem są mechanizmy paginacji zapytań oraz partycjonowania tabel, które pozwalają na ograniczenie liczby przetwarzanych rekordów oraz poprawę wydajności operacji odczytu i zapisu.

W niniejszych badaniach celowo nie zastosowano tych technik, ponieważ ich użycie mogłoby znacząco zniekształcić wyniki porównań pomiędzy poszczególnymi ORM-ami. Celem testów było bowiem zbadanie rzeczywistej wydajności warstwy dostępu do danych przy pełnym przetwarzaniu zbiorów, a nie efektywności mechanizmów optymalizacyjnych oferowanych przez bazę danych.

Zastosowanie paginacji ograniczyłoby liczbę rekordów zwracanych przez zapytania, a partycjonowanie wpłynęłoby na sposób planowania zapytań przez silnik bazy danych, co utrudniłoby bezpośrednie porównanie wyników pomiędzy technologiami. W związku z tym świadomie przyjęto uproszczony model pracy na pełnych zbiorach danych.

5.3.5 Metoda generowania danych testowych

Dane testowe zostały wygenerowane automatycznie przy użyciu autorskiego skryptu w języku Python. Skrypt generował polecenia SQL typu `INSERT`, które następnie były wykonywane w celu wypełnienia tabel bazy danych. Do generowania wartości tekstowych, takich jak imiona, nazwiska, adresy, adresy e-mail czy nazwy produktów, wykorzystano bibliotekę Faker. Dzięki rozszerzeniu `faker_commerce` możliwe było również uzyskanie wiarygodnej nomenklatury produktów, co pozwoliło uniknąć stosowania ciągów losowych znaków. Listing 20 przedstawia fragment użycia biblioteki do generowania nazwy produktu.

Listing 20 Przykład użycia biblioteki Faker do wygenerowania nazwy produktu

```
1. from faker import Faker
2. from faker_commerce import Provider as CommerceProvider
3.
4. fake = Faker()
5. fake.add_provider(CommerceProvider)
6.
7. product_name = fake.ecommerce_name()
```

Proces generowania danych został zaprojektowany modularnie, co umożliwiło zachowanie integralności między poszczególnymi encjami. Generowanie odbywało się w kilku etapach:

1. W pierwszej kolejności tworzone były kategorie produktów, stanowiące punkt odniesienia dla dalszych rekordów.

2. Wykorzystując wygenerowane identyfikatory, skrypt tworzył rekordy produktów wraz z przypisaniem ich do kategorii.
3. Następnie generowano profile klientów wraz z przypisanymi do nich adresami zamieszkania.
4. Ostatnim i najbardziej złożonym etapem było generowanie zamówień. Skrypt implementował logikę biznesową, dobierając losową liczbę produktów do każdego zamówienia, obliczając sumaryczną cenę oraz dopasowując statusy płatności do statusów zamówień.

Listing 21 Fragment logiki generatora odpowiedzialny za dane pojedynczego zlecenia

```
def generate_single_order(customer, address, products_list):
  order_id = str(uuid.uuid4())
  order_created_at = random_created_at()
  order_status = random_order_status()
  order_items_count = random.randint(1, 10)

  total_price = 0
  order_items = []
  for j in range(order_items_count):
    product = random.choice(products_list)
    quantity = random.randint(1, 10)

    discount_percent = product["discount_percent"] if
product["discount_percent"] is not None else "NULL"
    discount_value = (product["price"] * product["discount_percent"]) // 100 if
product[

"discount_percent"] is not None else 0
    price_after_discount = product["price"] - discount_value
    total_price += price_after_discount * quantity

    order_items.append({
      "id": str(uuid.uuid4()),
      "order_id": order_id,
      "product_id": product["id"],
      "quantity": quantity,
      "price": product["price"],
      "discount_percent": discount_percent,
      "created_at": order_created_at,
      "updated_at": order_created_at
    })

  order = {
    "id": order_id,
    "customer_id": customer["id"],
    "address_id": address["id"],
    "status": order_status,
    "total_price": total_price,
    "created_at": order_created_at,
    "updated_at": order_created_at,
  }
  payment = {
    "id": str(uuid.uuid4()),
    "order_id": order_id,
    "method": random_payment_method(),
    "status": random_payment_status(order_status),
    "created_at": order_created_at,
    "updated_at": order_created_at,
  }
  return order, order_items, payment
```

W celu optymalizacji procesu zapisu, skrypt korzystał z techniki buforowania (*batching*). Zamiast generowania pojedynczych instrukcji `INSERT` dla każdego rekordu, dane zostały pogrupowane w

paczki po 5000 rekordów na jedną instrukcję SQL. Podejście to redukuje czas procesowania skryptów przez silnik bazy danych.

Zastosowanie automatycznego generowania danych pozwoliło na szybkie tworzenie zbiorów danych oraz elastyczne dostosowywanie ich rozmiarów do potrzeb eksperymentów, bez konieczności ręcznego wprowadzania rekordów.

5.4 Środowisko testowe

W celu zapewnienia porównywalności wyników badań wydajnościowych, wszystkie testy zostały przeprowadzone w jednolitym środowisku testowym. Zarówno aplikacja testowa, jak i instancje baz danych zostały uruchomione w infrastrukturze chmurowej, co umożliwiło odwzorowanie warunków zbliżonych do rzeczywistych środowisk produkcyjnych.

Skorzystano z platformy chmurowej Heroku [54]. Jest ona usługą typu *Platform as a Service* (PaaS), która umożliwia szybkie wdrażanie, uruchamianie oraz skalowanie aplikacji bez konieczności samodzielnego zarządzania infrastrukturą serwerową. Zapewnia zintegrowane mechanizmy monitorowania, automatycznego skalowania oraz zarządzania konfiguracją środowiska, co czyni ją popularnym wyborem dla aplikacji internetowych i środowisk testowych.

Konfiguracja baz danych na platformie Heroku realizowana jest za pomocą tzw. dodatków (*add-ons*), które umożliwiają podłączanie zewnętrznych usług bazodanowych do aplikacji w sposób zautomatyzowany. Po dodaniu odpowiedniego dodatku do aplikacji, dane dostępowe do bazy są automatycznie udostępniane w postaci zmiennych środowiskowych, co upraszcza integrację z warstwą aplikacyjną oraz umożliwia szybkie przełączanie pomiędzy różnymi instancjami baz danych.

5.4.1 Serwer

Środowiskiem testowym był serwer VPS uruchomiony na platformie Heroku. Instancja ta charakteryzowała się następującą specyfikacją sprzętową:

- Procesor: Intel® Xeon® Platinum 8375C CPU @ 2.90GHz,
- Liczba rdzeni vCPU: 8,
- Pamięć RAM: 1 GB.

Aplikacja testowa została uruchomiona w środowisku Node.js w wersji 22 LTS, co zapewnia dostęp do najnowszych funkcjonalności platformy. Wybór wersji LTS (*Long-term support*) gwarantuje również długoterminowe wsparcie oraz stabilność.

5.4.2 Baza danych PostgreSQL

Do testów z wykorzystaniem bazy danych PostgreSQL zastosowano plan *Stackhero for PostgreSQL* o następującej specyfikacji [55]:

- Wersja silnika: PostgreSQL 18.0.0,
- Liczba vCPU: 1,
- Pamięć RAM: 2 GB,
- Przestrzeń dyskowa: 20 GB NVMe/SSD.

Konfiguracja ta odpowiada typowym ustawieniom spotykanym w produkcyjnych środowiskach średniej skali. Zastosowanie szybkich dysków NVMe/SSD umożliwia ograniczenie wpływu operacji wejścia-wyjścia na uzyskiwane wyniki.

5.4.3 Baza danych MySQL

Do testów z wykorzystaniem bazy danych MySQL zastosowano plan *Stackhero for MySQL* o następującej specyfikacji [56]:

- Wersja silnika: MySQL 8.4.6,
- Liczba vCPU: 1,
- Pamięć RAM: 4 GB,
- Przestrzeń dyskowa: 20 GB NVMe/SSD.

Jest to analogiczny pod względem wielkości i wydajności plan, jak w przypadku bazy PostgreSQL.

6. Implementacja

Rozdział ten poświęcony został opisowi procesu implementacji aplikacji testowej oraz sposobu realizacji założeń badawczych przedstawionych w poprzednich rozdziałach. W poniższych podrozdziałach przedstawiono praktyczny wymiar wykorzystania wybranych technologii ORM w kontekście rzeczywistego projektu, a także omówiono zastosowane rozwiązania oraz napotkane trudności.

W szczególności zaprezentowano strukturę kodu aplikacji, sposób integracji poszczególnych ORM-ów z warstwą serwerową opartą na Express.js, a także implementację operacji bazodanowych wykorzystywanych w scenariuszach testowych.

6.1 Konfiguracja połączenia z bazą danych

Połączenie z bazą danych zrealizowano za pomocą zmiennych środowiskowych, w których przechowywane są m.in. adres hosta, numer portu, nazwa bazy danych, nazwa użytkownika oraz hasło. Takie podejście umożliwiło łatwe przełączanie pomiędzy różnymi środowiskami (lokalnym, testowym oraz produkcyjnym) bez konieczności modyfikacji kodu źródłowego aplikacji. Dodatkowo, mechanizm ten pozwolił na zachowanie spójnej konfiguracji dla wszystkich testowanych technologii ORM.

Każdy ORM wykorzystywał własny mechanizm inicjalizacji połączenia oraz zarządzania pulą połączeń (*connection pool*). Pomimo różnic implementacyjnych, zadbano o to, aby parametry konfiguracyjne były możliwie zbliżone, co pozwoliło na ograniczenie wpływu warstwy konfiguracyjnej na wyniki pomiarów wydajności.

W kolejnych podrozdziałach przedstawiono sposób konfiguracji połączenia z bazą danych dla każdej z badanych technologii ORM, wraz z omówieniem kluczowych ustawień oraz fragmentami kodu.

6.1.1 Prisma

Konfiguracja połączenia z bazą danych dla Prismy wymagała utworzenia pliku konfiguracyjnego *schema.prisma*. Listing 22 przedstawia fragment tego pliku. Przekazywany jest URL do połączenia z bazą danych (zawierający w sobie nazwę bazy, użytkownika oraz hasło) oraz określany rodzaj bazy. Dla bazy MySQL wyglądało to analogicznie, z ustawieniem `provider = "mysql"`.

Listing 22 Konfiguracja połączenia z bazą danych w Prisma

```
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}
```

6.1.2 Sequelize

W przypadku Sequelize połączenie z bazą danych zostało zainicjalizowane poprzez utworzenie instancji klasy `Sequelize`, do której przekazywane są dane konfiguracyjne oraz opcje połączenia. Listing 23 przedstawia inicjalizację tej klasy.

Istotnym elementem konfiguracji w Sequelize jest parametr `dialect`, definiujący typ używanej bazy danych (`postgres` lub `mysql`), oraz `models` określający ścieżkę do plików zawierających definicje modeli.

Listing 23 Konfiguracja połączenia z bazą danych w Sequelize

```
import { Sequelize } from 'sequelize-typescript';
import { Dialect } from 'sequelize';

export const sequelize = new Sequelize({
  host: process.env.POSTGRES_HOST,
  port: Number(process.env.POSTGRES_PORT),
  database: process.env.POSTGRES_DB,
  username: process.env.POSTGRES_USER,
  password: process.env.POSTGRES_PASSWORD,
  dialect: 'postgres' as Dialect,
  schema: process.env.POSTGRES_SCHEMA,
  dialectOptions: {
    ssl: {
      rejectUnauthorized: false,
    },
  },
  models: [__dirname + '/models'],
});
```

6.1.3 Objection.js

Objection.js jest biblioteką opartą na *query builderze* Knex.js, dlatego konfiguracja połączenia z bazą danych realizowana jest pośrednio poprzez konfigurację Knex. W pierwszym kroku utworzono instancję klienta Knex z odpowiednimi parametrami połączenia, a następnie przekazano ją do Objection.js, który korzysta z niej wykonując operacje bazodanowe. Listing 24 przedstawia konfigurację tej instancji. Definiowanie rodzaju bazy danych odbywa się poprzez parametr `client`, który oznacza nazwę adaptera używanego do obsługi połączenia (`pg` dla PostgreSQL, `mysql2` dla MySQL).

Dokumentacja w przykładzie konfiguracji [57] podaje jeszcze parametr `useNullAsDefault`, który decyduje o sposobie traktowania pól o wartości `undefined` podczas operacji zapisu. Po jego włączeniu wartości niezdefiniowane są zastępowane przez `NULL` zamiast domyślnej wartości `DEFAULT`.

Listing 24 Konfiguracja połączenia z bazą danych w Objection.js

```
import Knex from 'knex';
import { Model } from 'objection';

const config: Knex.Config = {
  client: 'pg',
  connection: {
    host: process.env.POSTGRES_HOST,
    port: Number(process.env.POSTGRES_PORT),
    database: process.env.POSTGRES_DB,
    user: process.env.POSTGRES_USER,
    password: process.env.POSTGRES_PASSWORD,
    ssl: { rejectUnauthorized: false },
  },
};

const knex = Knex(config);
Model.knex(knex);
```

6.1.4 TypeORM

TypeORM oferuje dwa podejścia do pracy z bazą danych: wzorzec *Active Record* oraz wzorzec *Data Mapper*. W ramach testów zaimplementowano oba podejścia w celu porównania ich ergonomii, czytelności kodu oraz potencjalnego wpływu na wydajność.

Konfiguracja połączenia z bazą danych była taka sama w każdym z wybranych wzorców. Różnice pojawiły się dopiero na etapie implementacji modeli encji. Listing 25 przedstawia konfigurację połączenia z bazą. Rodzaj bazy danych określany jest parametrem `type`, z wartościami `postgres` lub `mysql`. W parametrze `entities` przekazywana jest tablica zawierająca wszystkie modele encji. Można również przekazać w niej ścieżki do plików zawierających modele lub do konkretnego katalogu z modelami.

Listing 25 Konfiguracja połączenia z bazą danych w TypeORM

```
import { DataSource } from 'typeorm';

export const dataSource = new DataSource({
  type: 'postgres',
  host: process.env.POSTGRES_HOST,
  port: Number(process.env.POSTGRES_PORT),
  database: process.env.POSTGRES_DB,
  username: process.env.POSTGRES_USER,
  password: process.env.POSTGRES_PASSWORD,
  schema: process.env.POSTGRES_SCHEMA,
  entities: [Address, Category, Customer, Order, OrderItem, Payment, Product],
  synchronize: false,
  ssl: true,
  extra: {
    ssl: {
      rejectUnauthorized: false,
    },
  },
});

dataSource
  .initialize()
  .then(() => {
    console.log('Data Source has been initialized!');
  })
  .catch((error) => console.log(error));
```

6.1.5 MikroORM

W przypadku MikroORM konfiguracja połączenia z bazą danych została zrealizowana poprzez bezpośrednią inicjalizację instancji ORM z użyciem metody `initSync`. Parametr `driver` określa sterownik bazy danych (`PostgreSqlDriver` lub `MySqlDriver`). Sterowniki trzeba zainstalować ręcznie poprzez pakiety `@mikro-orm/postgresql` lub `@mikro-orm/mysql`.

Listing 26 Konfiguracja połączenia z bazą danych w MikroORM

```
import { MikroORM } from '@mikro-orm/core';
import { PostgreSqlDriver } from '@mikro-orm/postgresql';

export const orm = MikroORM.initSync({
  driver: PostgreSqlDriver,
  host: process.env.POSTGRES_HOST,
  port: Number(process.env.POSTGRES_PORT),
  dbName: process.env.POSTGRES_DB,
  user: process.env.POSTGRES_USER,
  password: process.env.POSTGRES_PASSWORD,
  schema: process.env.POSTGRES_SCHEMA,
```

```
12.   entities: [Address, Category, Customer, Order, OrderItem, Payment, Product],
13.   driverOptions: {
14.     connection: { ssl: { rejectUnauthorized: false } },
15.   },
16. });
```

6.1.6 Różnice w konfiguracjach

Prisma wyróżnia się na tle pozostałych rozwiązań poprzez deklaratywnego języka DSL (*Domain Specific Language*) w konfiguracji. Pozostałe biblioteki wykorzystują konfigurację programową wewnątrz kodu TypeScript co pozwala na większą dynamikę, np. poprzez warunkowe modyfikowanie parametrów w czasie wykonania.

MikroORM wyróżnia się sposobem definiowania silnika bazy danych. Wymaga zainstalowania pakietu odpowiedniego sterownika a następnie przekazania jego klasy w konfiguracji.

6.2 Definicja modeli encji

W ramach aplikacji testowej każda z badanych technologii ORM została skonfigurowana w taki sposób, aby jak najwierniej odzwierciedlać wcześniej zaprojektowany model danych, obejmujący encje, ich atrybuty oraz relacje między nimi. Celem było zachowanie spójności semantycznej modeli we wszystkich ORM-ach, tak aby różnice w wynikach testów wynikały z odmiennych mechanizmów implementacyjnych, a nie z rozbieżności w strukturze danych.

6.2.1 Prisma

Definicja modeli encji w Prisma odbyła się tam, gdzie konfiguracja połączenia, czyli w pliku *schema.prisma*.

Listing 27 Definicja modelu encji Order w Prisma

```
 1. model Order {
 2.   id          String       @id @default(uuid())
 3.   status      String       @default("PENDING")
 4.   total_price Int
 5.   created_at  DateTime     @default(now()) @db.Timestamp(6)
 6.   updated_at  DateTime     @default(now()) @db.Timestamp(6)
 7.
 8.   customer_id String       @db.Uuid
 9.   customer    Customer     @relation(fields: [customer_id], references: [id])
10.
11.   address_id  String       @unique @db.Uuid
12.   address     Address      @relation(fields: [address_id], references: [id])
13.
14.   order_items Order_item[]
15.   payments    Payment[]
16.
17.   @@index([created_at])
18. }
```

Listing 27 przedstawia definicję encji. Taką definicję można używać zarówno w PostgreSQL jak i MySQL. W powyższym przykładzie, oprócz typów danych widać:

- oznaczenie klucza głównego - `@id`,
- generowanie wartości domyślnej dla kolumny - `@default`,
- oznaczenie wartości unikalnych - `@unique`,
- relację 1:1 z encją adres - `@relation`.

Listing 27 przedstawia generowanie domyślnego id poprzez funkcję ORMa. Istnieje również możliwość użycia funkcji dostępnych w konkretnym systemie baz danych. PostgreSQL udostępnia natywną funkcję `gen_random_uuid()`, z której można skorzystać. Przedstawia to Listing 28. Wykorzystuje się do tego instrukcję `dbgenerated()`.

Listing 28 Użycie natywnej funkcji PostgreSQL do generowania id w Prisma

```
1. model Order {
2.   id    String    @id @default(dbgenerated("gen_random_uuid()")) @db.Uuid
3. }
```

Po wprowadzeniu zmian w definicjach modeli w pliku *schema.prisma* konieczne było wygenerowanie klienta Prisma, który będzie odzwierciedlał aktualny stan modeli. W tym celu została uruchomiona odpowiednia komendy CLI Prisma: `npx prisma generate`.

6.2.2 Sequelize

W Sequelize definicja modeli encji realizowana jest poprzez klasy dziedziczące po klasie `Model`. Korzystając z TypeScript, konieczne było zastosowanie paczki `sequelize-typescript`, która pozwala na korzystanie z dekoratorów oraz typowania w definicjach modeli. Oficjalna dokumentacja nie jest przejrzysta pod tym względem, ponieważ konfiguracja biblioteki dla TypeScript została opisana na dole sekcji *Other topics* [58], co pokazuje, że obsługa typowania nie jest dopracowana. Ma się to poprawić w zapowiadanej wersji 7.

Listing 29 Definicja modelu encji Order w Sequelize

```
 1. @Table({
 2.   modelName: 'Order',
 3.   tableName: 'Order',
 4.   schema: 'public',
 5.   timestamps: true,
 6.   deletedAt: false,
 7. })
 8. class Order extends Model<Order> {
 9.   @PrimaryKey
10.   @Column({
11.     type: DataType.UUID,
12.     defaultValue: DataType.UUIDV4,
13.   })
14.   declare id: string;
15.
16.   @Column({ type: DataType.TEXT, defaultValue: OrderStatus.Pending })
17.   status!: string;
18.
19.   @Column(DataType.INTEGER)
20.   total_price!: number;
21.
22.   @ForeignKey(() => Customer)
23.   @Column(DataType.UUID)
24.   customer_id!: string;
25.
26.   @BelongsTo(() => Customer, { foreignKey: 'customer_id' })
27.   customer!: Customer;
28.
29.   @BelongsTo(() => Address, { foreignKey: 'address_id' })
30.   address!: Address;
31.
32.   @HasMany(() => OrderItem, { foreignKey: 'order_id' })
33.   orderItems!: OrderItem[];
34.
35.   @HasMany(() => Payment, { foreignKey: 'order_id' })
36.   payments!: Payment[];
```

```
37.
38.     @Index('order_created_at_key')
39.     @CreatedAt
40.     @Column(DataType.DATE)
41.     created_at!: Date;
42.
43.     @UpdatedAt
44.     @Column(DataType.DATE)
45.     updated_at!: Date;
46. }
```

Listing 29 przedstawia definicję modelu Order w Sequelize. Poszczególne kolumny definiowane są za pomocą dekoratora `@Column`. Podobnie realizowane są relacje pomiędzy encjami - `@BelongsTo`, `@HasMany`.

Aby korzystać z auto generowania UUID, oprócz zdefiniowanie dekoratora `@PrimaryKey` oraz określenia typu kolumny jako `DataType.UUID`, należało dodać do dekoratora `@Column` opcję `defaultValue: DataType.UUIDV4`. Bez niej ORM nie ustawiał domyślnej wartości.

Podczas implementacji modelów w Sequelize można natknąć się na różne problemy. Jednym z nich jest domyślne dodawanie do modeli pól *timestamp* o nazwach `createdAt`, `updatedAt`, i `deletedAt`. Można tą opcję wyłączyć korzystając z opcji `timestamps: false` dla dekoratora `@Table`. Jeśli w tabeli znajdują się niektóre z tych pól, pozostałe można wyłączyć korzystając z np. z opcji `deletedAt: false` (wyłącza pole `deletedAt`, zostawiając pozostałe).

6.2.3 Objection.js

W Objection.js definicja modeli encji również została zrealizowana poprzez klasy dziedziczące po klasie `Model`. W przeciwieństwie do Sequelize, ORM ten pozwala na zdefiniowania klasy bazowej (abstrakcyjnej), która zawiera wspólne pola oraz konfigurację wykorzystywaną przez wiele encji. Takie podejście pozwala ograniczyć duplikację kodu oraz ułatwia dalszą rozbudowę aplikacji. Listing 30 przedstawia definicję klasy `DefaultEntity`, która realizuje to założenie. Dzięki niej encja automatycznie otrzymuje powtarzające się pola identyfikatora oraz znaczników czasowych, co upraszcza definicje modeli.

Listing 30 Definicja klasy bazowej w Objection.js

```
 1. import { Model } from 'objection';
 2.
 3. export abstract class DefaultEntity extends Model {
 4.   id!: string;
 5.   created_at!: string;
 6.   updated_at!: string;
 7.
 8.   static get idColumn() {
 9.     return 'id';
10.   }
11. }
```

Każdy model wymagał zdefiniowania kolumny klucza głównego, poprzez metodę lub właściwość `idColumn`. Wymagane również było zdefiniowanie metody lub właściwości `tableName`, zwracającej nazwę tabeli.

Listing 31 przedstawia definicję modelu Order. Relacje pomiędzy encjami zostały zdefiniowane za pomocą metody `relationMappings`, gdzie określa się rodzaj relacji, klasę modelu docelowego oraz sposób łączenia tabel za pomocą kluczy obcych.

Objection.js pozwala wewnątrz modelu zdefiniować opcjonalną właściwość `jsonSchema`, która wykorzystuje standard JSON Schema do opisu typów pól, ich wymagalności oraz wartości domyślnych. Wykorzystywane jest to do walidacji danych wejściowych dla encji.

Listing 31 Definicja modelu encji Order w Objection.js

```
export class Order extends DefaultEntity {
  static tableName = 'Order';

  status!: string;
  total_price!: number;
  customer?: Customer;
  address?: Address;
  orderItems?: OrderItem[];
  payments?: Payment[];

  static get relationMappings() {
    return {
      customer: {
        relation: Model.BelongsToOneRelation,
        modelClass: Customer,
        join: {
          from: 'Order.customer_id',
          to: 'Customer.id',
        },
      },
      address: {
        relation: Model.BelongsToOneRelation,
        modelClass: Address,
        join: {
          from: 'Order.address_id',
          to: 'Address.id',
        },
      },
      orderItems: {
        relation: Model.HasManyRelation,
        modelClass: OrderItem,
        join: {
          from: 'Order.id',
          to: 'Order_item.order_id',
        },
      },
      payments: {
        relation: Model.HasManyRelation,
        modelClass: Payment,
        join: {
          from: 'Order.id',
          to: 'Payment.order_id',
        },
      },
    };
  }

  static get jsonSchema() {
    return {
      type: 'object',
      required: ['status', 'total_price'],
      properties: {
        id: { type: 'string', format: 'uuid' },
        status: { type: 'string', default: OrderStatus.Pending },
        total_price: { type: 'integer' },
        created_at: { type: 'string', format: 'date-time' },
        updated_at: { type: 'string', format: 'date-time' },
      },
```

```
59.     };
60.   }
61. }
```

6.2.4 TypeORM

W TypeORM modele danych definiowane są jako klasy TypeScript, które są mapowane na tabele w bazie danych za pomocą dekoratorów. Podobnie jak w Objection.js, również w TypeORM możliwe jest wydzielenie wspólnych pól do klasy bazowej, z której dziedziczą wszystkie encje.

Listing 32 przedstawia kod klasy bazowej `DefaultEntity`. Dziedziczy ona po klasie `BaseEntity`, co umożliwia korzystanie z metod *Active Record*. Dla podejścia *Data Mapper*, definicja klasy `DefaultEntity` pomija fragment `extends BaseEntity`, poza tym kod modelów jest identyczny.

Listing 32 Definicja klasy bazowej w TypeORM dla wzorca *Active Record*

```
1. import { BaseEntity, Column, PrimaryGeneratedColumn } from 'typeorm';
2.
3. export abstract class DefaultEntity extends BaseEntity {
4.   @PrimaryGeneratedColumn('uuid')
5.   id: string;
6.
7.   @Column('timestamp', {
8.     name: 'created_at',
9.     default: () => 'CURRENT_TIMESTAMP',
10.  })
11.  created_at: Date;
12.
13.  @Column('timestamp', {
14.    name: 'updated_at',
15.    default: () => 'CURRENT_TIMESTAMP',
16.  })
17.  updated_at: Date;
18. }
```

Listing 33 przedstawia definicję modelu Order. Podobnie jak w Sequelize, tak i w tej technologii wykorzystuje się dekoratory `@Column`. Relacje są modelowane za pomocą `@ManyToOne`, `@OneToOne`, `@OneToMany`.

Listing 33 Definicja modelu encji Order w TypeORM

```
1. @Entity('Order', { schema: 'public' })
2. @Index('order_created_at_key', ['created_at'])
3. export class Order extends DefaultEntity {
4.   @Column('text', { default: () => OrderStatus.Pending })
5.   status: string;
6.
7.   @Column('integer')
8.   total_price: number;
9.
10.  @ManyToOne(() => Customer, (customer) => customer.orders, { cascade: true })
11.  @JoinColumn([{ name: 'customer_id', referencedColumnName: 'id' }])
12.  customer: Customer;
13.
14.  @OneToOne(() => Address, { cascade: true })
15.  @JoinColumn([{ name: 'address_id', referencedColumnName: 'id' }])
16.  address: Address;
17.
18.  @OneToMany(() => OrderItem, (orderItem) => orderItem.order, {cascade: true })
19.  orderItems: OrderItem[];
20.
21.  @OneToMany(() => Payment, (payment) => payment.order, { cascade: true })
```

```
22.   payments: Payment[];
23. }
```

Warto również podkreślić, że TypeORM umożliwiał definiowanie relacji wiele-do-wielu bez konieczności tworzenia osobnej encji dla tabeli pośredniej. Zamiast tego relacja była deklarowana bezpośrednio w modelach za pomocą dekoratorów `@ManyToMany` oraz `@JoinTable`. Takie rozwiązanie jest szczególnie wygodne w sytuacjach, gdy tabela pośrednia nie zawiera dodatkowych atrybutów biznesowych, a służy jedynie do odwzorowania relacji pomiędzy encjami. Przykładem takiej konfiguracji jest relacja pomiędzy encjami `Category` oraz `Product`, z wykorzystaniem tabeli pośredniej `Product_category`. Przedstawia ją Listing 34.

Listing 34 Definicja modelu encji Category w TypeORM, wraz z relacją po tabeli pośredniej

```
 1. @Entity('Category', { schema: 'public' })
 2. export class Category extends DefaultEntity {
 3.   @Column('text', { unique: true })
 4.   name: string;
 5.
 6.   @ManyToMany(() => Product, (product) => product.categories)
 7.   @JoinTable({
 8.     name: 'Product_category',
 9.     joinColumns: [{ name: 'category_id', referencedColumnName: 'id' }],
10.     inverseJoinColumns: [{ name: 'product_id', referencedColumnName: 'id' }],
11.     schema: 'public',
12.   })
13.   products: Product[];
14. }
```

6.2.5 MikroORM

MikroORM wykorzystuje dekoratory do definiowania modeli encji w sposób zbliżony do podejścia znanego z TypeORM.

Podobnie jak w Objection.js i TypeORM, w MikroORM wydzielono klasę bazową, zawierającą wspólne pola, takie jak identyfikator oraz znaczniki czasowe. Listing 35 przedstawia definicję tej klasy. Typ `Opt` służy do oznaczania właściwości jako opcjonalnych na poziomie TypeScript, bez zmiany ich obowiązkowości w bazie danych. Jest on szczególnie przydatny w sytuacjach, gdy pola są automatycznie generowane i nie muszą być podawane podczas tworzenia nowej instancji encji.

Listing 35 Definicja klasy bazowej w MikroORM

```
 1. export abstract class DefaultEntity {
 2.   @PrimaryKey({ type: 'uuid', defaultRaw: 'gen_random_uuid()' })
 3.   id!: string;
 4.
 5.   @Property({
 6.     type: 'timestamp',
 7.     defaultRaw: 'CURRENT_TIMESTAMP',
 8.   })
 9.   created_at: Date & Opt = new Date();
10.
11.   @Property({
12.     type: 'timestamp',
13.     defaultRaw: 'CURRENT_TIMESTAMP',
14.     onUpdate: () => new Date(),
15.   })
16.   updated_at: Date & Opt = new Date();
17. }
```

MikroORM wymagał specjalnego typu `Collection` zamiast zwykłych tablic dla relacji typu jeden-do-wielu oraz wiele-do-wielu. Właściwości oznaczone dekoratorami `@OneToMany` oraz

`@ManyToMany` są przechowywane w obiektach typu `Collection` które oferują dodatkowe mechanizmy, głównie integrację z mechanizmem *Unit of Work*. Można również korzystać na nich z pętli for … of, ponieważ implementują interfejs iteratora [59]. Listing 36 przedstawia definicję modelu Order. Wykorzystuje także również klasę `Collection`.

Listing 36 Definicja modelu encji Order w MikroORM

```
@Entity({ tableName: 'Order', schema: 'public' })
@Index({ name: 'order_created_at_key', properties: ['created_at'] })
export class Order extends DefaultEntity {
  @Property({ type: 'text', default: OrderStatus.Pending })
  status!: string;

  @Property({ type: 'integer' })
  total_price!: number;

  @ManyToOne(() => Customer)
  customer!: Customer;

  @OneToOne(() => Address, { unique: true })
  address!: Address;

  @OneToMany(() => OrderItem, (orderItem) => orderItem.order)
  orderItems = new Collection<OrderItem>(this);

  @OneToMany(() => Payment, (payment) => payment.order)
  payments = new Collection<Payment>(this);
}
```

MikroORM, podobnie jak TypeORM, umożliwiał definiowanie relacji wiele-do-wielu bez konieczności tworzenia osobnej encji dla tabeli pośredniej. W tym celu zastosowano dekorator `@ManyToMany` wraz z konfiguracją tabeli pośredniej. Listing 37 przedstawia definicję modelu Category z relacją do Product po tabeli pośredniej. Opcja `pivotTable` wskazuje nazwę tabeli pośredniej, a `joinColumn` oraz `inverseJoinColumn` określają kolumny kluczy obcych.

Listing 37 Definicja modelu encji Category w MikroORM, wraz z relacją po tabeli pośredniej

```
@Entity({ tableName: 'Category' })
export class Category extends DefaultEntity {
  @Property({ type: 'text', unique: true })
  name!: string;

  @ManyToMany({
    entity: () => Product,
    owner: true,
    pivotTable: 'Product_category',
    joinColumn: 'category_id',
    inverseJoinColumn: 'product_id',
  })
  products = new Collection<Product>(this);
}
```

6.2.6 Różnice w modelach encji

W analizowanych technologiach ORM zauważalne są istotne różnice w sposobie definiowania modeli encji oraz ergonomii pracy z nimi:

- Sequelize, pomimo korzystania z klasy bazowej `Model`, charakteryzuje się problemami z pełnym typowaniem w TypeScript, przez co nie można zdefiniować własnej klasy bazowej, która uwspólnia część logiki pomiędzy modelami.

- Prisma stosuje zupełnie inne podejście — modele definiuje się w pliku schematu, a następnie generowany jest klient ORM. Oznacza to brak bezpośrednich klas encji w kodzie aplikacji, co ogranicza elastyczność modelowania na poziomie obiektowym.
- Objection.js nie wykorzystuje dekoratorów i wymaga ręcznego definiowania schematów JSON oraz mapowań relacji, co prowadzi do większej ilości kodu (*boilerplate*) i większego nakładu pracy przy utrzymaniu modeli.
- TypeORM oferuje zwięzłą i czytelną składnię opartą na dekoratorach.
- MikroORM posiada składnię zbliżoną do TypeORM, lecz dodatkowo wprowadza specjalizowane typy, takie jak `Opt` dla pól opcjonalnych oraz `Collection` dla relacji.

6.3 Implementacja operacji bazodanowych

Podrozdział ten poświęcony został opisowi implementacji operacji bazodanowych w poszczególnych technologiach ORM, z uwzględnieniem wybranych podstawowych, jak i bardziej złożonych scenariuszy użycia. Wykorzystano mechanizmy do realizacji operacji odczytu, zapisu oraz agregacji danych, zgodnie z założonymi scenariuszami testowymi. Celem było zapewnienie możliwie jednolitej logiki biznesowej we wszystkich implementacjach.

6.3.1 Prisma

Po utworzeniu modeli i wygenerowaniu klienta, należało utworzyć jego instancję. Przedstawia to Listing 38. Mając instancję klienta można było przejść do realizacji poszczególnych operacji.

Listing 38 Utworzenie instancji klienta Prismy

```
1. import { PrismaClient } from '@prisma/client'
2.
3. const prisma = new PrismaClient()
```

W celu pobrania zamówień z określonego zakresu dat wraz z dołączeniem powiązanych danych zastosowano metodę `findMany` z użyciem filtrów oraz opcji `include`, umożliwiającej *eager loading* relacji. Operacja została przedstawiona w Listing 39.

Listing 39 Realizacja operacji pobierania zleceń w Prisma

```
 1. const orders = await prisma.order.findMany({
 2.   where: {
 3.     created_at: {
 4.       gte: from,
 5.       lte: to,
 6.     },
 7.   },
 8.   include: {
 9.     address: true,
10.     payments: true,
11.     customer: true,
12.   },
13. });
```

Tworzenie nowego produktu wraz z przypisaniem go do kategorii zostało zrealizowane przy użyciu zagnieżdżonych operacji zapisu oraz mechanizmu `connect`, który pozwala na powiązanie nowego produktu z istniejącą kategorią. Operację przedstawia Listing 40.

Listing 40 Realizacja operacji tworzenia produktu w Prisma

```
const createdProduct = await prisma.product.create({
  data: {
    ...product,
    categories: {
      create: {
        category: {
          connect: {
            id: categoryId,
          },
        },
      },
    },
  },
});
```

Do obliczenia sumy wartości zamówień oraz liczby zamówień w zadanym przedziale czasu wykorzystano wbudowane metody agregujące. Listing 41 przedstawia kod wykorzystujący te metody.

Listing 41 Realizacja operacji agregujących zlecenia w Prisma

```
const sum = await prisma.order.aggregate({
  where: {
    created_at: {
      gte: from,
      lte: to,
    },
  },
  _sum: {
    total_price: true,
  },
});

const orderCount = await prisma.order.count({
  where: {
    created_at: {
      gte: from,
      lte: to,
    },
  },
});
```

6.3.2 Sequelize

Pobieranie zamówień z określonego zakresu dat wraz z danymi powiązanych encji zrealizowano przy użyciu metody `findAll` oraz opcji `include`. Operację przedstawia Listing 42. W odróżnieniu od Prismy, aby użyć operatorów porównania <= lub >= trzeba zaimportować obiekt `Op` zawierający wszystkie dostępne operatory porównań.

Listing 42 Realizacja operacji pobierania zleceń w Sequelize

```
const orders = await Order.findAll({
  where: {
    created_at: {
      [Op.gte]: from,
      [Op.lte]: to,
    },
  },
  include: [Address, Payment, Customer],
});
```

Z racji tego, że Sequelize korzysta z wzorca *Active Record*, tworzenie nowego produktu wraz z przypisaniem do kategorii zostało zrealizowane poprzez utworzenie instancji modelu oraz zapisanie jej do bazy danych metodą `save`. Operację przedstawia Listing 43.

Listing 43 Realizacja operacji tworzenia produktu w Sequelize

```
const category = await Category.findByPk(categoryId);
const product = new Product({
  name: productInput.name,
  price: productInput.price,
  discount_percent: productInput.discount_percent || null,
  stock: productInput.stock,
  categories: [category?.dataValues!],
});
const createdProduct = await product.save();
```

Do obliczenia sumy wartości zamówień oraz liczby zamówień w zadanym zakresie dat wykorzystano metody agregujące `sum` oraz `count`. Listing 44 przedstawia kod wykorzystujący te metody.

Listing 44 Realizacja operacji agregujących w Sequelize

```
const sum = await Order.sum('total_price', {
  where: {
    created_at: {
      [Op.gte]: from,
      [Op.lte]: to,
    },
  },
});

const orderCount = await Order.count({
  where: {
    created_at: {
      [Op.gte]: from,
      [Op.lte]: to,
    },
  },
});
```

6.3.3 Objection.js

W Objection.js operacje bazodanowe realizowane są za pomocą interfejsu zapytań opartego na wzorcach znanych z bibliotek typu *query builder*. Zapytania konstruowane są metodami łańcuchowymi. Takie podejście pozwala na realizację zapytań w sposób zbliżony do SQL.

Pobieranie zamówień z określonego zakresu dat wraz z powiązanymi encjami zostało zrealizowane z wykorzystaniem metody `query()` oraz `withGraphJoined`, która realizuje dołączenia po relacjach. Operację przedstawia Listing 45.

Listing 45 Realizacja operacji pobierania zleceń w Objection.js

```
const orders = await Order.query()
  .whereBetween('created_at', [from, to])
  .withGraphJoined('[address, payments, customer]');
```

Tworzenie produktu wraz z przypisaniem go do kategorii wykonano przy użyciu metody `insertGraph`, która umożliwia zapis obiektów z relacjami w jednej operacji. Opcja `relate` umożliwia powiązanie istniejących rekordów z nowo wstawianymi obiektami. Operację przedstawia Listing 46.

Listing 46 Realizacja operacji tworzenia produktu w Objection.js

```
const createdProduct = await Product.query().insertGraph(
  {
    ...product,
    categories: [{ id: categoryId }],
  } as PartialModelGraph<Product>,
  { relate: true },
);
```

Do obliczenia sumy wartości oraz liczby zamówień w zadanym zakresie dat wykorzystano metody agregujące `sum` oraz `resultSize`. Listing 47 przedstawia funkcje wykorzystujące te metody.

Listing 47 Realizacja operacji agregujących w Objection.js

```
const result = await Order.query()
  .sum('total_price')
  .whereBetween('created_at', [from, to]);
const sum = parseInt((result[0] as any).sum ?? 0);

const orderCount = await Order.query()
.whereBetween('created_at', [from, to])
.resultSize();
```

6.3.4 TypeORM

W TypeORM operacje bazodanowe można realizować dwiema głównymi metodami: przy użyciu wzorca *Active Record* lub *Data Mapper*. W obu podejściach możliwe jest wykonywanie analogicznych operacji, jednak różnią się one sposobem organizacji kodu. Wszystkie implementowane operacje zostały zrealizowane zarówno w podejściu *Active Record*, jak i *Data Mapper*.

Pobieranie zamówień z określonego zakresu dat wraz z powiązanymi encjami zrealizowano przy użyciu opcji `relations` do załadowania relacji. Operację przedstawia Listing 48. Podobnie jak w Sequelize, operatory porównań trzeba importować z biblioteki. Podłączanie relacji jest podobne do występującego w Prismie.

Listing 48 Realizacja operacji pobierania zleceń w TypeORM

```
// Active record
const orders = await Order.find({
  where: {
    created_at: Between(from, to),
  },
  relations: {
    address: true,
    payments: true,
    customer: true,
  },
});

// Data mapper
const orders = await orderRepository.find({
  where: {
    created_at: Between(from, to),
  },
  relations: {
    address: true,
    payments: true,
    customer: true,
  },
});
```

Tworzenie nowego produktu wraz z przypisaniem do kategorii wygląda podobnie w obu podejściach, przy czym w *Data Mapper* operacje wykonywane są przez odpowiedni *repository*, a w *Active Record* bezpośrednio na modelu. Przedstawia je Listing 49.

Listing 49 Realizacja operacji tworzenia produktu w TypeORM

```
// Active record
const category = await Category.findOneByOrFail({
  id: categoryId,
});
const product = new Product();
product.name = productInput.name;
product.price = productInput.price;
product.discount_percent = productInput.discount_percent || null;
product.stock = productInput.stock;
product.categories = [category!];
const createdProduct = await product.save();

// Data mapper
const category = await categoryRepository.findOneByOrFail({
  id: categoryId,
});
const product = new Product();
product.name = productInput.name;
product.price = productInput.price;
product.discount_percent = productInput.discount_percent || null;
product.stock = productInput.stock;
product.categories = [category!];
const createdProduct = await productRepository.save(product);
```

Do obliczenia sumy wartości oraz liczby zamówień w zadanym przedziale dat wykorzystano wbudowane metody agregujące `sum` oraz `countBy`, dostępne zarówno w *Active Record*, jak i *Data Mapper*. Operacje przedstawia Listing 50.

Listing 50 Realizacja operacji agregujących w TypeORM

```
// Active record
const sum = await Order.sum('total_price', {
  created_at: Between(from, to),
});

// Data mapper
const sum = await orderRepository.sum('total_price', {
  created_at: Between(from, to),
});

// Active record
const orderCount = await Order.countBy({
  created_at: Between(from, to),
});

// Data mapper
const orderCount = await orderRepository.countBy({
  created_at: Between(from, to),
});
```

6.3.5 MikroORM

W MikroORM operacje bazodanowe realizowane są za pomocą jednostki zarządzającej encjami (*Entity Manager*) oraz wbudowanego *query buildera* opartego na Knex. Dzięki temu możliwe jest zarówno korzystanie z wysokopoziomowego API ORM, jak i wykonywanie bardziej złożonych zapytań SQL w razie potrzeby.

Pobieranie zamówień z określonego zakresu dat wraz z powiązanymi encjami zrealizowano przy użyciu metody `find` oraz opcji `populate` dołączającej encje po relacjach. Operację przedstawia Listing 51. Składnia pod względem warunków i dołączania relacji jest zbliżona do Prismy.

Listing 51 Realizacja operacji pobierania zleceń w MikroORM

```
const orders = await orm.em.find(
  Order,
  {
    created_at: {
      $gte: from,
      $lte: to,
    },
  },
  {
    populate: ['address', 'payments', 'customer'],
  },
);
```

Z racji tego, że MikroORM korzysta z wzorca *Unit of Work*, tworzenie nowego produktu oraz przypisanie go do kategorii zrealizowano poprzez utworzenie encji metodą `create`, a następnie zapis do bazy przy użyciu `persistAndFlush`. Operację przedstawia Listing 52.

Listing 52 Realizacja operacji tworzenia produktu w MikroORM

```
const createdProduct = orm.em.create(Product, {
  ...product,
  categories: [categoryId],
});
await orm.em.persistAndFlush(createdProduct);
```

Do obliczenia liczby zleceń wykorzystano metodę `count` bezpośrednio z *Entity Manager*. Obliczenie sumy wartości zamówień było bardziej skomplikowane, ponieważ należało do tego użyć *query buildera* Knex. Metoda `getKnex()` pozwoliła na utworzenie jego instancji, którą następnie wykorzystano do przeprowadzenia zapytania `sum`. Sposób użycia metod przedstawia Listing 53.

Listing 53 Realizacja operacji agregujących w MikroORM

```
const knex = orm.em.getKnex();
const result = await knex
  .queryBuilder()
  .sum('total_price')
  .from('Order')
  .whereBetween('created_at', [from, to]);
const sum = parseInt(result[0].sum ?? 0);

const orderCount = await orm.em.count(Order, {
  created_at: { $gte: from, $lte: to },
});
```

6.3.6 Różnice pomiędzy technologiami

Pomimo pewnych podobieństw, każda z bibliotek narzucała specyficzne podejście do interakcji z danymi.

Pierwszą istotną różnicę można było zaobserwować w sposobie definiowania warunków filtrowania (operacje *WHERE*):

- Prisma i MikroORM wykorzystywały obiekty JavaScript do opisu filtrów,

- Sequelize i TypeORM wymagały jawnego importowania dedykowanych operatorów lub funkcji (np. `Op.gte` w Sequelize lub `Between` w TypeORM),
- Objection.js stosował metody łańcuchowe (np. `.whereBetween()`), co było najbardziej zbliżone do naturalnej składni języka SQL.

Różnice występowały również w sposobie konfiguracji dołączania danych powiązanych:

- Prisma oraz TypeORM wykorzystywały obiekty z flagami logicznymi (`include` lub `relations`),
- Sequelize oraz MikroORM bazowały na tablicach nazw modeli lub właściwości (`include` lub `populate`),
- Objection.js wyróżniał się osobną metodą `withGraphJoined()`.

Największe różnice ujawniały się podczas operacji zapisu:

- W Sequelize i TypeORM (we wzorcu *Active Record*) tworzenie obiektu odbywało się poprzez instancjonowanie klasy modelu, a jego utrwalenie wymagało wywołania metody `.save()`.
- TypeORM w podejściu *Data Mapper* oddzielał definicję danych od logiki zapisu, wykorzystując repozytoria (`repository.save()`), co sprzyjało lepszej separacji warstw aplikacji.
- Prisma i Objection.js oferowały zaawansowane mechanizmy tworzenia relacji w jednym zapytaniu (`connect` w Prismie, `insertGraph` w Objection.js).
- MikroORM wprowadzał najwyższy poziom abstrakcji poprzez *Entity Manager*. Operacja `persistAndFlush` pozwalała na kolejkowanie zmian i wykonanie ich w jednej transakcji bazodanowej.

Dla zapytań agregujących dane (obliczanie sum, liczebności) można było zauważyć następujący podział:

- Prisma, Sequelize i TypeORM dostarczały gotowe metody (np. `.count(), .sum()`), które ukrywały złożoność SQL pod swoim interfejsem,
- Objection.js oraz MikroORM w przypadku bardziej złożonych agregacji (jak suma wartości) wymagały odwołania się do bazowego budowniczego zapytań (Knex.js). Wskazuje to na mniejszy stopień automatyzacji tych bibliotek w obszarze funkcji analitycznych.

6.4 Implementacja mechanizmów pomiarowych

W celu zapewnienia obiektywnego pomiaru wydajności operacji bazodanowych zastosowano mechanizm pomiarowy w postaci *wrappera*, który otaczał wywołanie testowanej operacji. Pozwoliło to na automatyczne zbieranie danych dotyczących czasu wykonania, zużycia procesora oraz pamięci operacyjnej, niezależnie od używanego ORM.

Do pomiaru zużycia zasobów systemowych wykorzystano wbudowane mechanizmy środowiska Node.js, bez użycia zewnętrznych bibliotek. Informacje o czasie procesora pobierane były za pomocą funkcji `process.cpuUsage()`, która zwracała czas zużyty przez proces w trybie użytkownika oraz systemowym.

Zużycie pamięci monitorowano przy użyciu funkcji `process.memoryUsage()`, która dostarczała szczegółowych danych na temat aktualnego wykorzystania pamięci przez proces, w tym. wielkość pamięci faktycznie używanej przez obiekty (*heap used RAM)*.

Podstawową funkcją pomiarową była `measurePerformance`, która przyjmowała jako argument funkcję asynchroniczną reprezentującą testowaną operację, a następnie zwracała obiekt typu `MeasurementResult`.

Proces pomiaru składał się z trzech etapów:

- Rozpoczęcie pomiaru – zapisywano aktualny czas, zużycie CPU oraz pamięci. Realizowała go funkcja `startMeasurement`, przedstawiona na Listing 54.
- Wykonanie operacji - wywoływana była właściwa operacja bazodanowa poprzez wywołanie funkcji przekazanej jako argument (`fn`).
- Zakończenie pomiaru - ponownie odczytywano wartości CPU, pamięci oraz czas zakończenia. Realizowała go funkcja `endMeasurement`, przedstawiona na Listing 54.

Listing 54 Funkcje zbierające pomiary wydajności

```
const startMeasurement = () => {
  const startCpu = process.cpuUsage();
  const startMem = process.memoryUsage();
  const startTimestamp = new Date();
  return { startCpu, startMem, startTimestamp };
};

const endMeasurement = (startCpu: NodeJS.CpuUsage) => {
  const endTimestamp = new Date();
  const endCpu = process.cpuUsage(startCpu);
  const endMem = process.memoryUsage();
  return { endTimestamp, endCpu, endMem };
};
```

Na podstawie zebranych danych obliczono:

- czas wykonania operacji - w milisekundach,
- zużycie zasobów CPU i pamięci - wartości CPU przeliczono na milisekundy, natomiast zużycie pamięci wyrażono jest w megabajtach.

Listing 55 przedstawia logikę obliczania poszczególnych metryk. Na samym końcu zwracano obiekt typu `MeasurementResult`, zawierający pola:

- `duration` – całkowity czas wykonania operacji (w ms), obejmujący komunikację z bazą danych, wykonanie zapytania oraz przetwarzanie wyników po stronie aplikacji,
- `cpu.user` – czas procesora zużyty na wykonywanie kodu w trybie użytkownika (logika aplikacji, przetwarzanie danych),
- `cpu.system` – czas procesora zużyty na operacje systemowe (np. obsługa wejścia/wyjścia, operacje jądra systemu),
- `cpu.total` – suma czasu użytkownika i systemowego, reprezentująca całkowite zużycie CPU przez daną operację,
- `heapUsed` – przyrost ilości pamięci zaalokowanej podczas wykonania operacji. *Heap* to obszar pamięci, w którym JavaScript przechowuje obiekty i struktury danych tworzone dynamicznie w trakcie działania programu.

Listing 55 Funkcje realizujące logikę pomiarów wydajności

```
export const calculateStats = (
  endCpu: NodeJS.CpuUsage,
  startMem: NodeJS.MemoryUsage,
  endMem: NodeJS.MemoryUsage,
) => {
  const cpu = {
    user: (endCpu.user / 1000).toFixed(2),
    system: (endCpu.system / 1000).toFixed(2),
    total: ((endCpu.user + endCpu.system) / 1000).toFixed(2),
  };
  const heapUsed = ((endMem.heapUsed - startMem.heapUsed) / 1024 /
1024).toFixed(2)
  return { cpu, heapUsed };
};

export interface MeasurementResult {
  duration: number;
  cpu: {
    user: string;
    system: string;
    total: string;
  };
  heapUsed: string;
}

export async function measurePerformance<T>(
  fn: () => Promise<T>,
): Promise<MeasurementResult> {
  const { startCpu, startMem, startTimestamp } = startMeasurement();

  await fn();

  const { endTimestamp, endCpu, endMem } = endMeasurement(startCpu);
  const duration = endTimestamp.getTime() - startTimestamp.getTime();
  const stats = calculateStats(endCpu, startMem, endMem);

  return {
    duration,
    ...stats,
  };
}
```

7. Analiza wyników testów wydajności

W niniejszym rozdziale przedstawiono wyniki testów wydajnościowych przeprowadzonych dla wybranych bibliotek ORM w środowisku Node.js. Celem testów było porównanie ich zachowania w typowych scenariuszach aplikacyjnych, takich jak operacje odczytu, zapisu, zapytania agregujące oraz obsługa transakcji. Analizie poddano czas wykonania operacji, zużycie zasobów procesora oraz pamięci, co pozwoliło na kompleksową ocenę efektywności poszczególnych rozwiązań.

7.1 Analiza czasu wykonania operacji

Czas wykonania operacji bazodanowych stanowi jeden z kluczowych czynników wpływających na wydajność aplikacji. W niniejszym podrozdziale przeanalizowano wyniki pomiarów czasu realizacji operacji dla poszczególnych technologii ORM. Analizie poddano zarówno operacje odczytu i zapisu danych, jak i zapytania agregujące, a także wpływ rozmiaru zbiorów danych na uzyskiwane rezultaty. Uzyskane wyniki pozwoliły na ocenę efektywności poszczególnych rozwiązań w różnych scenariuszach obciążenia.

7.1.1 Operacje odczytu

W ramach analizy operacji odczytu zbadano wydajność trzech zapytań, odzwierciedlających rzeczywiste scenariusze biznesowe:

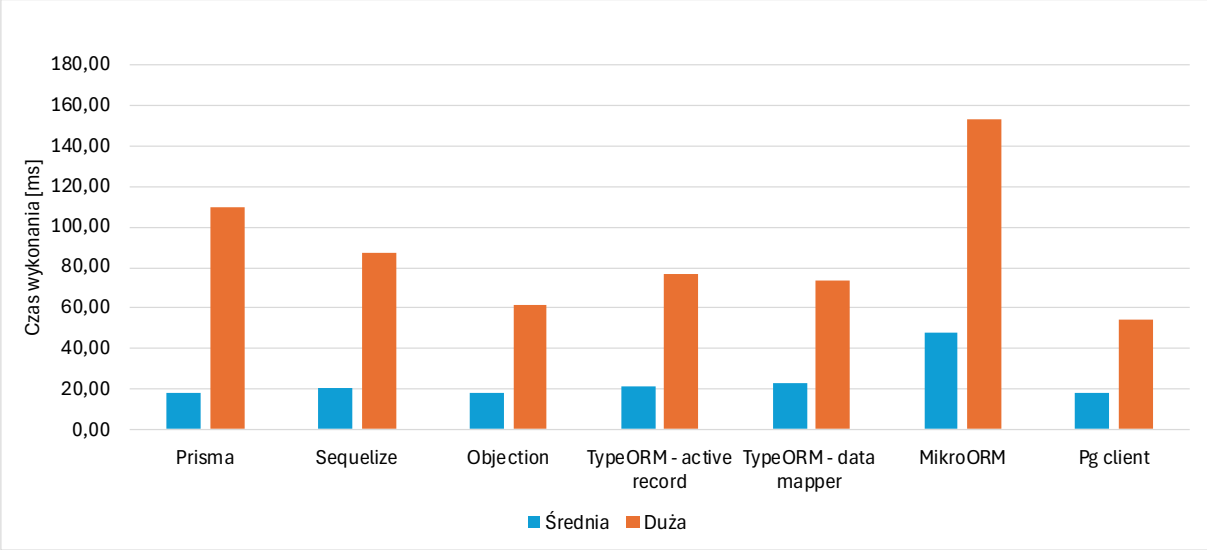
- Zapytanie 1 – pobranie wszystkich zamówień przypisanych do danego klienta,
- Zapytanie 2 – pobranie pojedynczego zamówienia wraz z powiązanymi pozycjami zamówienia oraz danymi produktów,
- Zapytanie 3 – pobranie wszystkich zamówień z określonego zakresu dat wraz z danymi klientów, adresami oraz informacjami o płatnościach.

Liczba przetwarzanych rekordów znacząco rosła wraz ze wzrostem rozmiaru bazy danych, co zostało przedstawione w Tabela 4. Dla małych zbiorów zapytania obejmowały od kilkunastu do około stu wierszy, natomiast w przypadku dużych baz danych liczba ta sięgała nawet kilkudziesięciu tysięcy rekordów, szczególnie dla zapytania trzeciego.

Tabela 4 Liczba wierszy przetwarzanych w zapytaniach odczytu. Źródło: opracowanie własne

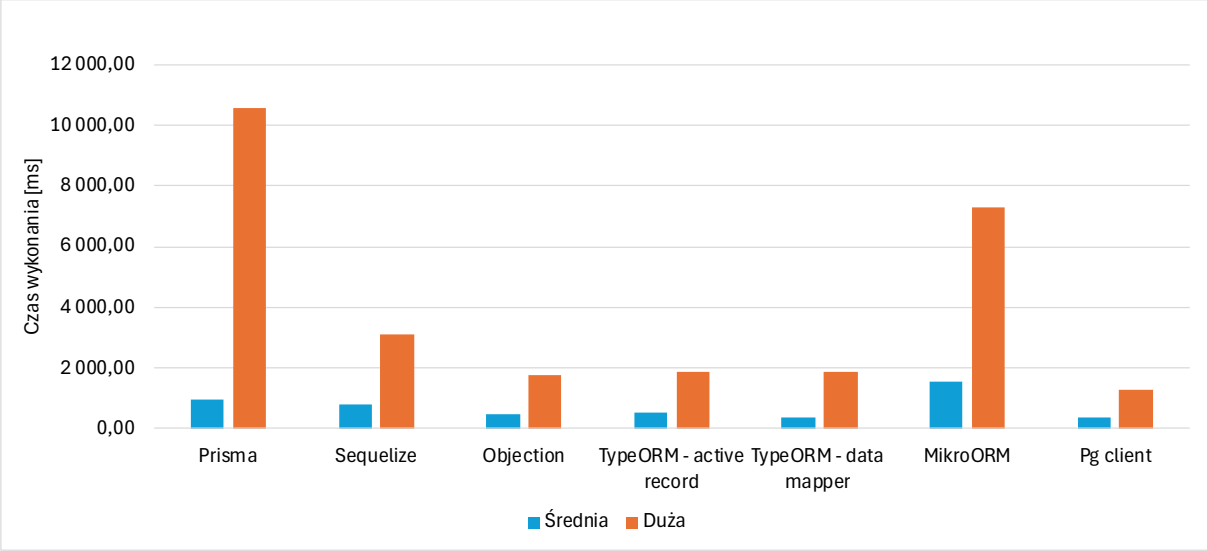
Wielkość bazy danych	**Liczba wierszy przetwarzanych w zapytaniu**		
	Zapytanie 1	Zapytanie 2	Zapytanie 3
Mała	98	102	21
Średnia	988	1 019	9 987
Duża	9 810	10 018	49 329

Analizę czasów wykonania zapytania 1 (prostego odczytu listy zamówień klienta) przedstawia Wykres 8. W przypadku średnich zbiorów danych wszystkie technologie ORM osiągały relatywnie zbliżone wyniki, mieszczące się w przedziale do 20 milisekund. Najlepsze rezultaty uzyskały Objection.js oraz Prisma, natomiast nieco wyższe czasy odnotowano dla MikroORM.



Wykres 8 Czas wykonania (w milisekundach) operacji prostego odczytu (zapytanie 1), dla PostgreSQL, mniej = lepiej. Źródło: opracowanie własne

Wraz ze wzrostem rozmiaru bazy danych różnice pomiędzy poszczególnymi rozwiązaniami stawały się bardziej widoczne. W szczególności MikroORM wykazywał wyraźnie wyższe czasy wykonania w porównaniu do pozostałych ORM-ów, co może wynikać z dodatkowych mechanizmów zarządzania kolekcjami. Najbardziej stabilne wyniki, zarówno dla średnich, jak i dużych zbiorów danych, uzyskały Objection.js oraz TypeORM (w obu wzorcach).



Wykres 9 Czas wykonania (w milisekundach) operacji złożonego odczytu (zapytanie 3), dla PostgreSQL, mniej = lepiej. Źródło: opracowanie własne

Wykres 9 przedstawia czasy wykonania operacji złożonego odczytu (zapytanie 3), obejmującego pobranie zamówień z określonego zakresu dat wraz z danymi klientów, adresami oraz informacjami o płatnościach. Zapytanie to charakteryzowało się wysoką złożonością oraz dużą liczbą przetwarzanych rekordów, szczególnie w przypadku średnich i dużych zbiorów danych.

Dla średnich baz danych najlepsze wyniki uzyskały Objection.js oraz TypeORM (w obu wzorcach), których czasy wykonania utrzymywały się na poziomie kilkuset milisekund. Nieco wyższe czasy odnotowano dla Sequelize, natomiast wyraźnie gorsze wyniki uzyskały Prisma oraz MikroORM.

W przypadku dużych zbiorów danych różnice wydajnościowe stały się jeszcze bardziej widoczne. Prisma oraz MikroORM osiągnęły czasy wykonania rzędu kilku do ponad dziesięciu sekund, podczas gdy Objection.js, TypeORM zachowały znacznie lepszą skalowalność, realizując zapytanie w czasie poniżej dwóch sekund.

7.1.2 Operacje agregujące

Operacje agregujące obejmowały scenariusze zapytań przedstawione w Tabela 5:

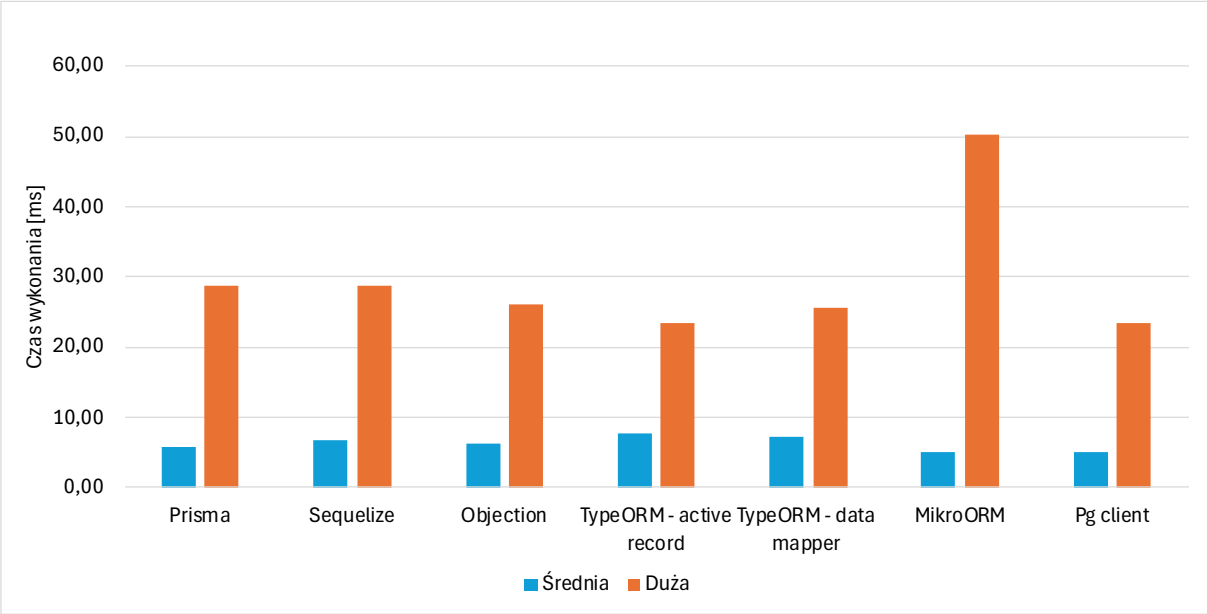
- Zapytanie 4 – liczba zamówień złożonych w danym zakresie dat,
- Zapytanie 5 – obliczenie całkowitej wartości sprzedaży w danym zakresie dat.

Oba zapytania operowały na porównywalnej liczbie rekordów, której wielkość rosła wraz ze skalą zbioru danych, co przedstawiono w Tabela 5.

Tabela 5 Liczba wierszy przetwarzanych z zapytaniach agregujących. Źródło: opracowanie własne

Wielkość bazy danych	Liczba wierszy przetwarzanych w zapytaniu	
	Zapytanie 4	Zapytanie 5
Mała	98	102
Średnia	988	1 019
Duża	9 810	10 018

Wykres 10 prezentuje czasy wykonania operacji agregującej polegającej na obliczeniu sumy wartości sprzedaży (zapytanie 5). Dla małych zbiorów danych różnice pomiędzy poszczególnymi technologiami ORM były niewielkie, a czasy wykonania utrzymywały się na poziomie kilku milisekund. Wraz ze wzrostem liczby przetwarzanych rekordów różnice wydajnościowe stawały się jednak coraz bardziej wyraźne.



Wykres 10 Czas wykonania (w milisekundach) operacji agregującej (zapytanie 5), dla PostgreSQL, mniej = lepiej. Źródło: opracowanie własne

Najlepsze wyniki dla średnich i dużych zbiorów danych uzyskały Objection.js i TypeORM (w obu wzorcach), co sugeruje niski narzut związany z generowaniem zapytań SQL dla operacji agregujących. Nieco słabsze rezultaty osiągnął Sequelize, natomiast wyraźnie najdłuższe czasy wykonania odnotowano dla MikroORM, szczególnie w przypadku dużych baz danych.

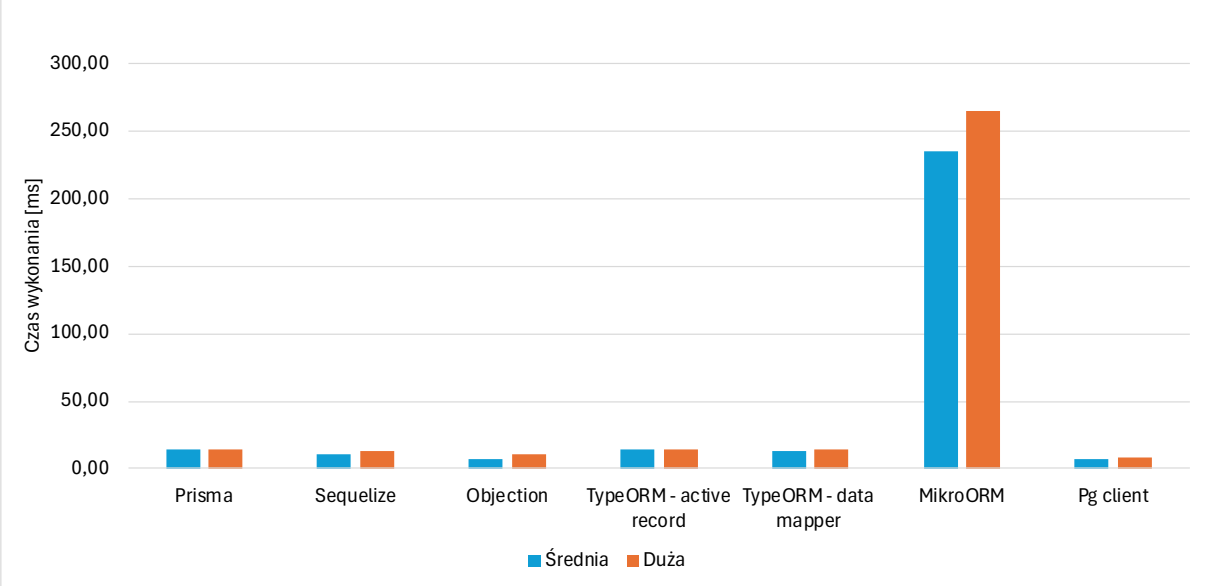
7.1.3 Operacje zapisu

Operacje zapisu obejmowały dwa scenariusze:

- Zapytanie 6 – wstawienie pojedynczego rekordu klienta wraz z powiązanym adresem,
- Zapytanie 7 – wstawienie nowego produktu wraz z przypisaniem go do kategorii.

Oba przypadki odzwierciedlały typowe operacje zapisu danych w aplikacjach biznesowych, w których zachodzi konieczność jednoczesnej obsługi powiązanych encji.

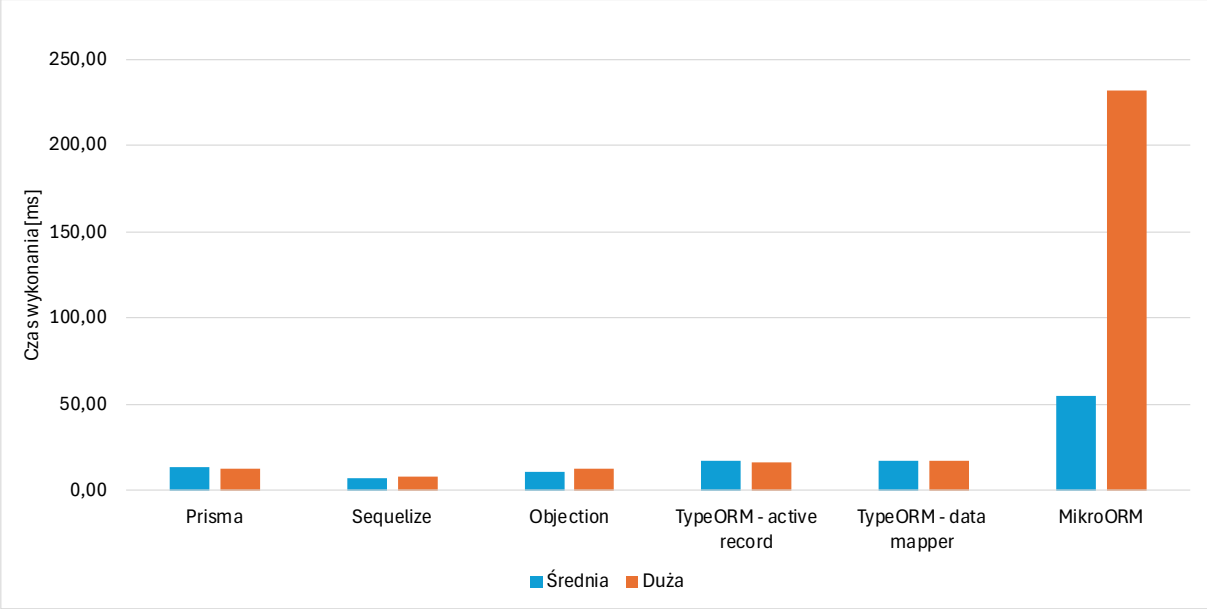
Wykres 11 przedstawia czasy wykonania operacji zapisu (zapytanie 6) dla bazy PostgreSQL. Zdecydowanie najsłabsze wyniki uzyskał MikroORM, którego czasy wykonania były wielokrotnie wyższe niż w przypadku pozostałych technologii ORM. Różnice te były widoczne niezależnie od rozmiaru zbioru danych, co sugeruje istotny narzut związany z mechanizmami zapisu oraz zarządzania encjami w MikroORM.



Wykres 11 Czas wykonania (w milisekundach) operacji zapisu (zapytanie 6), dla PostgreSQL, mniej = lepiej. Źródło: opracowanie własne

Najlepsze rezultaty dla PostgreSQL osiągnął Objection.js, a bardzo zbliżone czasy uzyskały również Sequelize oraz TypeORM w obu wariantach wzorców. Prisma uplasowała się w środkowej części zestawienia, wykazując nieco wyższe czasy zapisu.

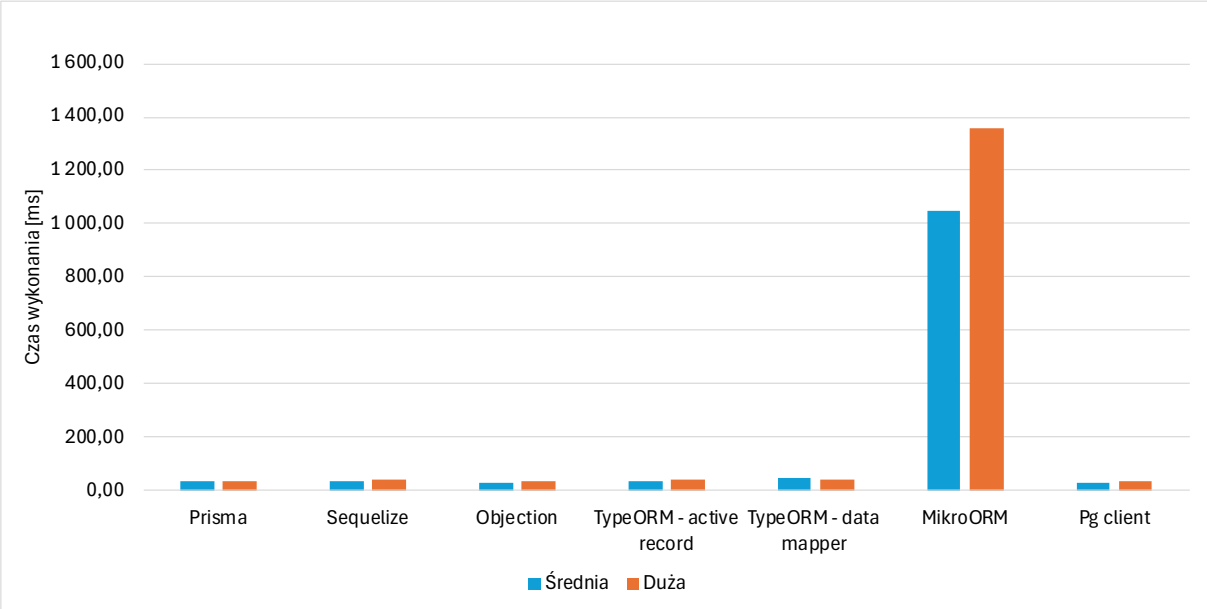
Analogiczne zależności zaobserwowano dla bazy MySQL, co przedstawia Wykres 12. Również w tym przypadku MikroORM charakteryzował się najwyższymi czasami wykonania, szczególnie dla średnich i dużych zbiorów danych. Pozostałe technologie ORM uzyskały zbliżone i relatywnie niskie czasy zapisu, przy czym najkorzystniejsze wyniki osiągnął Sequelize, a następnie Objection.js oraz Prisma.



Wykres 12 Czas wykonania (w milisekundach) operacji zapisu (zapytanie 6), dla MySQL, mniej = lepiej. Źródło: opracowanie własne

7.1.4 Transakcje

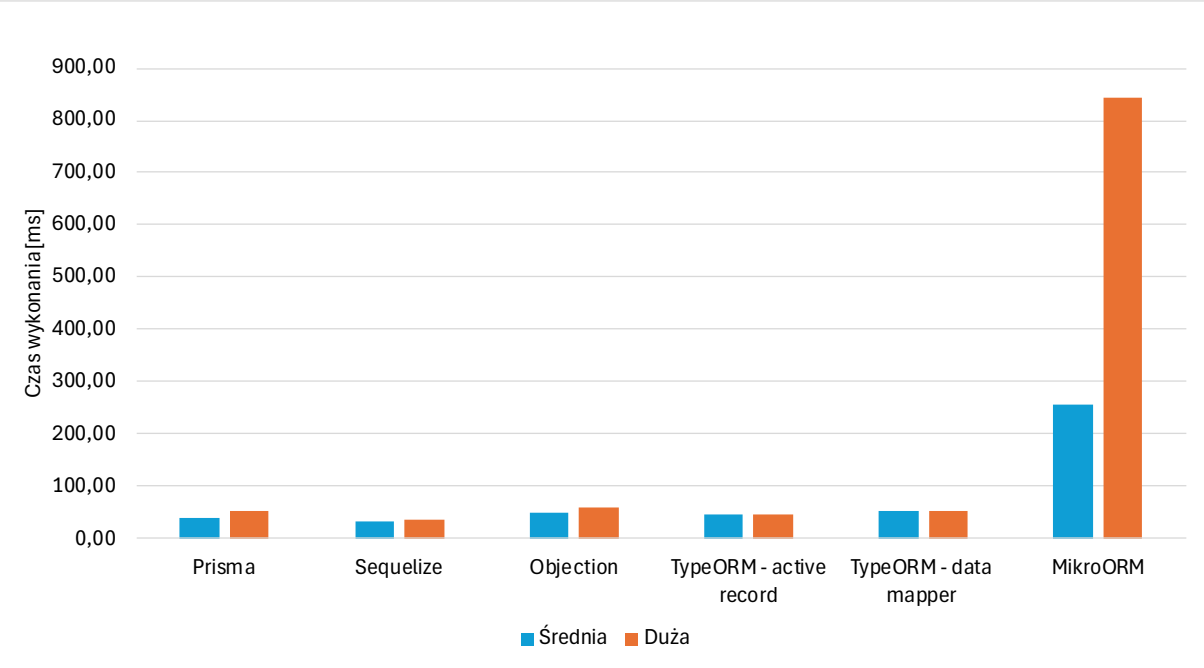
W przeprowadzonych testach scenariusz transakcyjny polegał na utworzeniu nowego zamówienia wraz z wieloma pozycjami zamówienia, adresem oraz płatnością przypisaną do zamówienia, a także na jednoczesnej aktualizacji stanów magazynowych produktów. Cała operacja musiała zostać wykonana atomowo, tak aby w przypadku błędu żadne zmiany nie zostały trwale zapisane w bazie danych.



Wykres 13 Czas wykonania (w milisekundach) operacji transakcyjnej, dla PostgreSQL, mniej = lepiej. Źródło: opracowanie własne

Wykres 13 przedstawia wyniki uzyskane dla bazy PostgreSQL. Najlepsze czasy osiągnął Objection.js. Zbliżone, choć nieco wyższe czasy uzyskały Sequelize oraz TypeORM w obu wariantach

wzorców projektowych. decydowanie najsłabsze rezultaty odnotowano dla MikroORM, którego czasy wykonania były kilkadziesiąt razy wyższe od pozostałych rozwiązań, co wskazuje na znaczący narzut związany z obsługą transakcji i zarządzaniem *Unit of work*.



Wykres 14 Czas wykonania (w milisekundach) operacji transakcyjnej, dla MySQL, mniej = lepiej. Źródło: opracowanie własne

Dla bazy MySQL (Wykres 14) obserwowany trend był podobny, choć różnice pomiędzy technologiami były nieco mniejsze dla małych zbiorów danych. MikroORM w przypadku średnich i dużych zbiorów danych również wykazywał wyraźnie gorszą wydajność, szczególnie dla dużej bazy danych, gdzie czas wykonania transakcji znacząco przewyższał pozostałe rozwiązania.

7.2 Analiza zużycia zasobów systemowych

Oprócz czasu wykonania operacji, istotnym aspektem oceny technologii ORM był ich wpływ na zużycie zasobów systemowych, w szczególności pamięci operacyjnej oraz mocy obliczeniowej procesora. W środowiskach produkcyjnych nadmierne zużycie zasobów może prowadzić do obniżenia stabilności aplikacji, zwiększenia kosztów infrastruktury oraz pogorszenia skalowalności systemu.

W niniejszym podrozdziale przeanalizowano wyniki pomiarów zużycia pamięci oraz obciążenia procesora dla poszczególnych technologii ORM podczas wykonywania testowanych operacji. Analiza ta pozwoliła na ocenę efektywności zarządzania zasobami oraz identyfikację potencjalnych wąskich gardeł wydajnościowych.

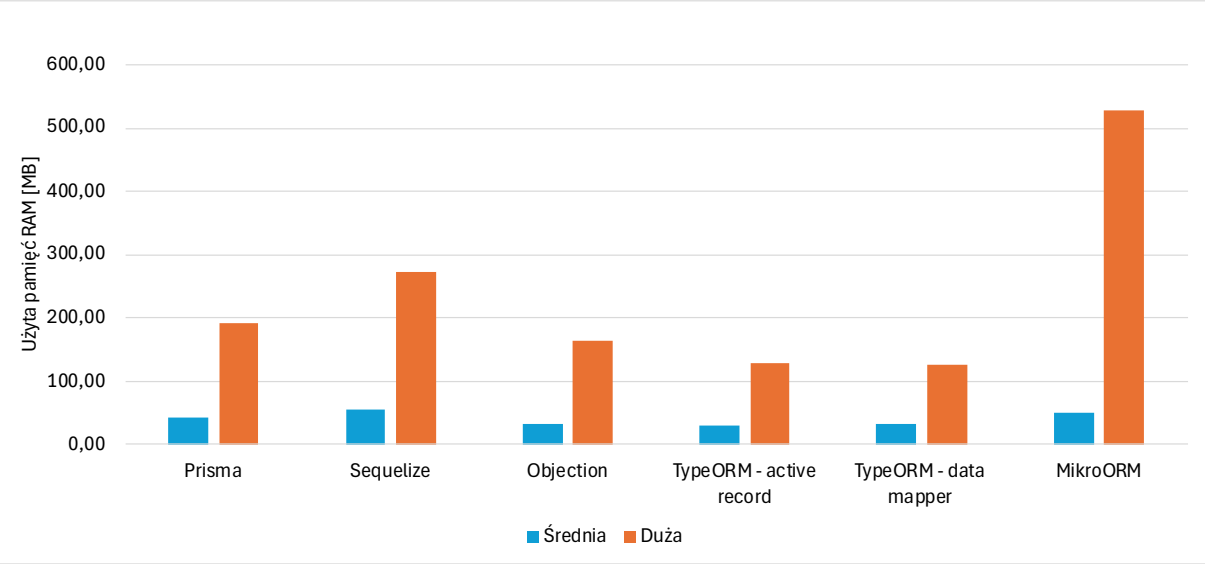
7.2.1 Zarządzanie pamięcią

W ramach analizy zarządzania pamięcią badano wartość *heap used*, czyli ilość pamięci faktycznie używanej przez obiekty w procesie podczas wykonywania operacji bazodanowych. Parametr ten pozwala ocenić, jak duży narzut pamięciowy generuje dana technologia ORM oraz jak efektywnie radzi sobie z alokacją i zwalnianiem zasobów.

W przypadku prostych zapytań, takich jak pojedyncze inserty czy zapytania agregujące, wszystkie badane technologie charakteryzowały się niskim i stabilnym zużyciem pamięci, mieszczącym się zazwyczaj poniżej 1 MB, niezależnie od rozmiaru bazy danych. Wyjątkiem był MikroORM, który

już przy operacjach zapisu wykazywał wyraźnie wyższe zużycie pamięci (do kilkunastu MB dla dużej bazy danych).

Znacznie większe różnice pomiędzy technologiami zaobserwowano dla złożonych zapytań odczytu, szczególnie dla pobrania wszystkich zamówień z określonego zakresu dat wraz z danymi klientów, adresami oraz informacjami o płatnościach (zapytanie 3), gdzie przy dużym rozmiarze bazy zużycie pamięci sięgało ponad 500 MB w przypadku MikroORM, podczas gdy TypeORM i Prisma mieściły się w przedziale od około 125 do 190 MB. Różnice przedstawia Wykres 15.



Wykres 15 Zużycie pamięci RAM dla operacji złożonego odczytu (zapytanie 3), dla PostgreSQL, mniej = lepiej. Źródło: opracowanie własne

Wyniki te wskazują, że w aplikacjach wymagających wysokiej efektywności pamięciowej, zwłaszcza przy intensywnych operacjach odczytu, wybór technologii ORM może mieć istotny wpływ na stabilność i skalowalność systemu.

7.2.2 Obciążenie procesora

Drugim analizowanym aspektem było obciążenie procesora, mierzone za pomocą parametru *total CPU time*, który obejmuje zarówno czas użytkownika, jak i czas systemowy zużyty przez proces Node.js podczas wykonywania operacji. Metryka ta pozwala ocenić, jak intensywnie poszczególne technologie ORM wykorzystują zasoby obliczeniowe.

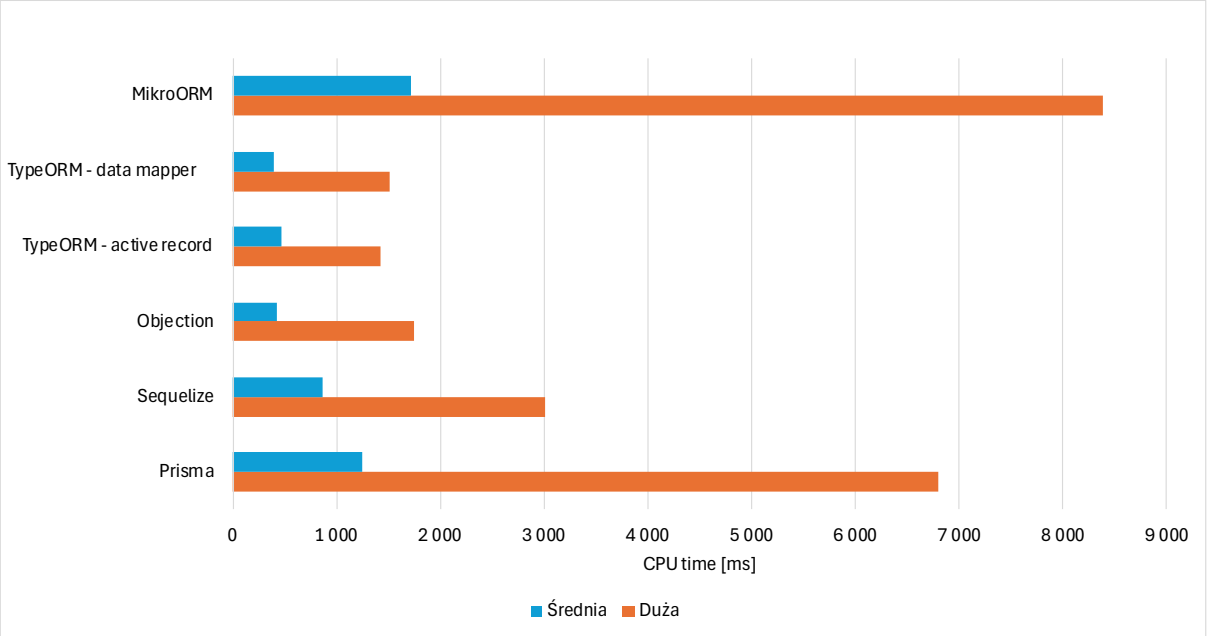
Dla operacji odczytu o niewielkiej złożoności, takich jak pobranie wszystkich zamówień danego klienta, różnice pomiędzy ORM-ami były stosunkowo niewielkie, szczególnie dla małych i średnich zbiorów danych. Wraz ze wzrostem rozmiaru bazy zauważalny był jednak wyraźny wzrost zużycia CPU, przy czym najniższe wartości osiągały Objection.js oraz TypeORM, a najwyższe – MikroORM, co wskazuje na większy narzut związany z przetwarzaniem obiektów, prawdopodobnie wynikający z użycia *IdentityMap*.

Znacznie większe różnice odnotowano dla operacji złożonego odczytu (zapytanie 3). W przypadku średnich i dużych zbiorów danych obciążenie procesora rosło dla wszystkich technologii, jednak największy wzrost wystąpił dla MikroORM oraz Prisma. Objection.js oraz TypeORM wykazywały relatywnie niższe wartości, co sugeruje bardziej efektywne generowanie zapytań oraz mniejszy narzut przetwarzania danych po stronie aplikacji.

Dla operacji zapisu oraz zapytań agregujących zużycie CPU pozostawało na niskim i stabilnym poziomie, niezależnie od rozmiaru zbioru danych. Różnice pomiędzy ORM-ami były w tych

przypadkach niewielkie, co wskazuje, że tego typu operacje nie stanowią istotnego obciążenia obliczeniowego dla aplikacji.

Największe różnice pomiędzy technologiami zaobserwowano w scenariuszu transakcyjnym (**Błąd! Nie można odnaleźć źródła odwołania.**). W tym przypadku MikroORM generował zdecydowanie najwyższe obciążenie procesora, kilkukrotnie przewyższające pozostałe ORM-y, niezależnie od rozmiaru bazy danych. Pozostałe technologie, takie jak Prisma, Sequelize, Objection.js oraz TypeORM, charakteryzowały się znacznie niższymi i bardziej stabilnymi wartościami *total CPU time*.



Wykres 16 Całkowity czas CPU (w milisekundach) dla operacji transakcyjnej, dla PostgreSQL, mniej = lepiej. Źródło: opracowanie własne

7.3 Skalowalność i wpływ rozmiarów danych

Skalowalność rozwiązań ORM stanowi kluczowy aspekt w kontekście aplikacji obsługujących rosnące wolumeny danych. W niniejszym podrozdziale przeanalizowano, w jaki sposób wzrost rozmiaru zbiorów danych wpływał na czas wykonania operacji, zużycie pamięci operacyjnej oraz obciążenie procesora dla poszczególnych technologii ORM.

Dla małych zbiorów danych różnice pomiędzy analizowanymi rozwiązaniami były zazwyczaj niewielkie. Większość ORM-ów osiągała zbliżone czasy wykonania operacji, a zużycie zasobów systemowych pozostawało na niskim poziomie. W takich warunkach wybór technologii ORM może być determinowany innymi czynnikami, takimi jak ergonomia pracy, dokumentacja czy wsparcie dla TypeScript.

W przypadku średnich zbiorów danych zaczęły pojawiać się pierwsze różnice w skalowalności poszczególnych rozwiązań. Szczególnie widoczne było to dla operacji złożonego odczytu oraz transakcji, gdzie niektóre ORM-y wykazywały szybszy wzrost czasu wykonania oraz zużycia zasobów. Objection.js oraz TypeORM charakteryzowały się stabilnym wzrostem kosztów obliczeniowych, natomiast Prisma i MikroORM zaczynały wykazywać wyraźnie większy narzut, zwłaszcza w zakresie zużycia pamięci.

Najbardziej wyraźne różnice zaobserwowano dla dużych zbiorów danych. Dla części ORM-ów wzrost czasu wykonania operacji oraz zużycia zasobów następował w sposób stopniowy, natomiast inne rozwiązania wykazywały gwałtowne pogorszenie wydajności wraz ze wzrostem liczby przetwarzanych rekordów. Szczególnie MikroORM cechował się znacznym wzrostem zarówno zużycia pamięci RAM, jak i obciążenia procesora, co było widoczne dla operacji złożonego odczytu oraz transakcji. Z kolei Objection.js oraz TypeORM zachowywały względnie stabilną charakterystykę skalowania, nawet przy bardzo dużej liczbie przetwarzanych rekordów.

Podsumowując, wpływ rozmiaru danych na wydajność aplikacji wykorzystującej ORM staje się istotny dopiero przy średnich i dużych zbiorach danych. W takich scenariuszach wybór technologii ORM może mieć kluczowe znaczenie dla dalszej skalowalności systemu, stabilności działania oraz kosztów infrastruktury.

Należy jednak podkreślić, że na wydajność aplikacji przy dużych zbiorach danych wpływa nie tylko wybór technologii ORM. W praktycznych zastosowaniach produkcyjnych kluczowe znaczenie mają również odpowiednie mechanizmy po stronie bazy danych oraz warstwy dostępu do danych, takie jak partycjonowanie tabel czy paginacja zapytań. Ich zastosowanie pozwala ograniczyć liczbę rekordów przetwarzanych jednorazowo oraz zmniejszyć ilość danych ładowanych do pamięci operacyjnej, co może istotnie poprawić skalowalność systemu niezależnie od użytego ORM.

8. Podsumowanie

Celem pracy było porównanie wybranych technologii ORM dostępnych dla środowiska Node.js pod kątem wydajności, zużycia zasobów systemowych oraz skalowalności w zależności od rozmiaru przetwarzanych danych. Przeprowadzone testy objęły typowe scenariusze aplikacyjne, takie jak operacje odczytu, zapisu, zapytania agregujące oraz złożone operacje transakcyjne, realizowane na bazach danych PostgreSQL oraz MySQL.

Uzyskane wyniki wykazały, że dla niewielkich zbiorów danych różnice pomiędzy analizowanymi technologiami ORM są stosunkowo niewielkie i mają ograniczone znaczenie praktyczne. Wraz ze wzrostem liczby rekordów oraz złożoności operacji różnice te stawały się coraz bardziej widoczne, szczególnie w zakresie czasu wykonania operacji oraz zużycia pamięci i procesora. Największy wpływ na wydajność miały operacje złożonego odczytu oraz transakcje, w których narzut warstwy ORM był najbardziej zauważalny.

Na podstawie przeprowadzonych testów można stwierdzić, że TypeORM stanowi najbardziej uniwersalny wybór do zastosowań produkcyjnych. Oferuje on dobrą wydajność, czytelną i zwięzłą składnię, rozwijaną dokumentację oraz dobre wsparcie dla TypeScript. Istotną zaletą TypeORM jest możliwość wyboru wzorca pracy – *Active Record* lub *Data Mapper* – co pozwala dopasować sposób implementacji do potrzeb projektu. Dodatkowym atutem jest jego silne powiązanie z frameworkiem Nest.js, co czyni go naturalnym wyborem dla nowoczesnych aplikacji.

Prisma może być rozważana jako rozwiązanie produkcyjne, szczególnie w projektach o prostszej strukturze zapytań. Jej zaletami są dynamiczny rozwój, duża popularność oraz wysoki poziom ergonomii pracy. Jednocześnie zauważalne problemy z wydajnością złożonych zapytań oraz ograniczona kontrola nad generowanym SQL mogą stanowić istotne ograniczenie w bardziej zaawansowanych scenariuszach.

Sequelize w świetle uzyskanych wyników oraz aktualnego stanu rozwoju nie wydaje się dobrym wyborem do nowych projektów. Problemy z integracją z TypeScript oraz ograniczona dynamika rozwoju biblioteki wpływają negatywnie na jej przydatność w nowoczesnych aplikacjach.

Objection.js wyróżniał się dobrą wydajnością jednak jego przestarzała architektura oraz duża ilość kodu konfiguracyjnego sprawiają, że trudno rekomendować go do nowych projektów, mimo dobrych wyników wydajnościowych.

MikroORM osiągał najsłabsze wyniki pod względem wydajności oraz zużycia zasobów, szczególnie dla dużych zbiorów danych i operacji transakcyjnych. Stosunkowo niewielka popularność oraz ograniczona dojrzałość technologii powodują, że rozwiązanie to należy traktować raczej jako ciekawostkę technologiczną niż realną alternatywę produkcyjną.

Na zakończenie warto zaznaczyć, że przedstawione wyniki nie wyczerpują tematu badań nad wydajnością ORM. Niniejszą pracę można w przyszłości rozszerzyć o uwzględnienie nowych narzędzi pojawiających się na rynku lub ponowną analizę wybranych technologii w przypadku istotnych aktualizacji wpływających na ich architekturę i wydajność. Dodatkowo wartościowym kierunkiem dalszych badań byłoby przeprowadzenie testów obciążeniowych całych aplikacji wykorzystujących ORM, co pozwoliłoby ocenić ich zachowanie w warunkach równoległego dostępu oraz rzeczywistego obciążenia produkcyjnego.

Bibliografia

[1] „Ranking popularności SZBD opracowany przez Redgate Software”, DB-Engines. Dostęp: 18 grudnia 2025. [Online]. Dostępne na: https://db-engines.com/en/ranking

[2] M. Trzaska, *Modelowanie i implementacja systemów informatycznych 2.0, ISBN: 978-83-976442-0-5*. Warszawa: Mariusz Trzaska, 2025.

[3] „Badanie Stack Overflow Developer Survey”. Dostęp: 18 grudnia 2025. [Online]. Dostępne na: https://survey.stackoverflow.co/2024/technology#1-web-frameworks-and-technologies

[4] K. Subieta, *Słownik terminów z zakresu obiektowości*. Warszawa: Akademicka Oficyna Wydawnicza PLJ, 1999.

[5] P. J. Sadalage i M. Fowler, *NoSQL. Kompendium wiedzy*. Gliwice: Helion, 2015.

[6] „Definicja ORM”, Amazon Web Services, Inc. Dostęp: 18 grudnia 2025. [Online]. Dostępne na: https://aws.amazon.com/what-is/object-relational-mapping/

[7] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2012.

[8] „Dokumentacja Prisma - Is Prisma ORM an ORM?” Dostęp: 22 grudnia 2025. [Online]. Dostępne na: https://www.prisma.io/docs/orm/overview/prisma-in-your-stack/is-prisma-an-orm

[9] „Dokumentacja SQLAlchemy - N plus one problem”. Dostęp: 22 grudnia 2025. [Online]. Dostępne na: https://docs.sqlalchemy.org/en/20/glossary.html#term-N-plus-one-problem

[10] „Blog AppMaster - Query Builder”. Dostęp: 22 grudnia 2025. [Online]. Dostępne na: https://appmaster.io/blog/sql-query-builder

[11] „Prisma blog, Comparing SQL, query builders, and ORMs”, Prisma’s Data Guide. Dostęp: 22 grudnia 2025. [Online]. Dostępne na: https://www.prisma.io/dataguide/types/relational/comparing-sql-query-builders-and-orms

[12] „Repozytorium projektu Objection.js - The future of Objection.js”, GitHub. Dostęp: 22 grudnia 2025. [Online]. Dostępne na: https://github.com/Vincit/objection.js/issues/2335

[13] „Repozytorium projektu Bookshelf.js”, GitHub. Dostęp: 22 grudnia 2025. [Online]. Dostępne na: https://github.com/bookshelf/bookshelf

[14] „Repozytorium projektu waterline”, GitHub. Dostęp: 22 grudnia 2025. [Online]. Dostępne na: https://github.com/balderdashy/waterline

[15] „Serwis npmjs”. Dostęp: 22 grudnia 2025. [Online]. Dostępne na: https://www.npmjs.com/

[16] „Serwis npmtrends.com”. Dostęp: 22 grudnia 2025. [Online]. Dostępne na: https://npmtrends.com/mikro-orm-vs-objection-vs-prisma-vs-sequelize-vs-typeorm

[17] E. Casco, „Repozytorium w serwisie Github”, GitHub. Dostęp: 29 grudnia 2025. [Online]. Dostępne na: https://github.com/emanuelcasco/typescript-orm-benchmark

[18] „Repozytorium projektu autocannon”, GitHub. Dostęp: 29 grudnia 2025. [Online]. Dostępne na: https://github.com/mcollina/autocannon

[19] Kushyn, „Node.js ORMs overview and comparison”, Roman’s blog. Dostęp: 29 grudnia 2025. [Online]. Dostępne na: https://romeerez.hashnode.dev/nodejs-orms-overview-and-comparison

[20] „Sequelize - github issue”, GitHub. Dostęp: 29 grudnia 2025. [Online]. Dostępne na: https://github.com/sequelize/sequelize/issues/5193

[21] „Orchid orm w serwisie npmjs”, npm. Dostęp: 29 grudnia 2025. [Online]. Dostępne na: https://www.npmjs.com/package/orchid-orm

[22] „Dokumentacja OrchidORM - Current status and limitations”. Dostęp: 29 grudnia 2025. [Online]. Dostępne na: https://orchid-orm.netlify.app/guide/current-status-and-limitations.html

[23] „Prisma’s Data Guide - Top 11 Node.js ORMs, query builders & database libraries in 2022”, Prisma’s Data Guide. Dostęp: 29 grudnia 2025. [Online]. Dostępne na: https://www.prisma.io/dataguide/database-tools/top-nodejs-orms-query-builders-and-database-libraries

[24] N. Burk, „How Prisma ORM Became the Most Downloaded ORM for Node.js”, Prisma. Dostęp: 10 stycznia 2026. [Online]. Dostępne na: https://www.prisma.io/blog/how-prisma-orm-became-the-most-downloaded-orm-for-node-js

[25] „Serwis npmtrends.com - prisma”. Dostęp: 10 stycznia 2026. [Online]. Dostępne na: https://npmtrends.com/prisma
[26] „Prisma - profil firmy w servisie pitchbook”. Dostęp: 10 stycznia 2026. [Online]. Dostępne na: https://pitchbook.com/profiles/company/171490-60
[27] „Prisma - about us”, Prisma. Dostęp: 10 stycznia 2026. [Online]. Dostępne na: https://www.prisma.io/about
[28] „Dokumentacja Prisma - Databases supported by Prisma ORM”. Dostęp: 10 stycznia 2026. [Online]. Dostępne na: https://www.prisma.io/docs/orm/reference/supported-databases
[29] „Dokumentacja Prisma - What is Prisma ORM?” Dostęp: 10 stycznia 2026. [Online]. Dostępne na: https://www.prisma.io/docs/orm/overview/introduction/what-is-prisma
[30] „Dokumentacja Prisma - What is introspection?” Dostęp: 10 stycznia 2026. [Online]. Dostępne na: https://www.prisma.io/docs/orm/prisma-schema/introspection
[31] „Dokumentacja Prisma - Engines”. Dostęp: 10 stycznia 2026. [Online]. Dostępne na: https://www.prisma.io/docs/orm/more/under-the-hood/engines
[32] „Dokumentacja Prisma - Database drivers”. Dostęp: 10 stycznia 2026. [Online]. Dostępne na: https://www.prisma.io/docs/orm/overview/databases/database-drivers
[33] „Prisma ORM Manifesto”, Prisma. Dostęp: 11 stycznia 2026. [Online]. Dostępne na: https://www.prisma.io/blog/prisma-orm-manifesto
[34] „Prisma - From Rust to TypeScript”, Prisma. Dostęp: 11 stycznia 2026. [Online]. Dostępne na: https://www.prisma.io/blog/from-rust-to-typescript-a-new-chapter-for-prisma-orm
[35] „Serwis npmtrends.com - sequelize”. Dostęp: 11 stycznia 2026. [Online]. Dostępne na: https://npmtrends.com/sequelize
[36] „Dokumentacja Sequelize”. Dostęp: 11 stycznia 2026. [Online]. Dostępne na: https://sequelize.org/releases/
[37] „Dokumentacja Sequelize - Getting Started”. Dostęp: 11 stycznia 2026. [Online]. Dostępne na: https://sequelize.org/docs/v6/getting-started/
[38] „Dokumentacja Sequelize - Model Basics”. Dostęp: 11 stycznia 2026. [Online]. Dostępne na: https://sequelize.org/docs/v6/core-concepts/model-basics/
[39] „Repozytorium sequelize w serwisie github”, GitHub. Dostęp: 11 stycznia 2026. [Online]. Dostępne na: https://github.com/sequelize/sequelize/releases
[40] „Dokumentacja Objection.js”. Dostęp: 12 stycznia 2026. [Online]. Dostępne na: https://vincit.github.io/objection.js/
[41] „Serwis npmtrends.com - objection”. Dostęp: 12 stycznia 2026. [Online]. Dostępne na: https://npmtrends.com/objection
[42] „The future of Objection.js”, GitHub. Dostęp: 12 stycznia 2026. [Online]. Dostępne na: https://github.com/Vincit/objection.js/discussions/2463
[43] „Dokumentacja Nest.js”, Documentation | NestJS - A progressive Node.js framework. Dostęp: 12 stycznia 2026. [Online]. Dostępne na: https://docs.nestjs.com
[44] „Serwis npmtrends.com - TypeORM”. Dostęp: 12 stycznia 2026. [Online]. Dostępne na: https://npmtrends.com/typeorm
[45] „Dokumentacja TypeORM - Getting Started”. Dostęp: 12 stycznia 2026. [Online]. Dostępne na: https://typeorm.io/docs/getting-started/
[46] „Future of TypeORM”, GitHub. Dostęp: 13 stycznia 2026. [Online]. Dostępne na: https://github.com/typeorm/typeorm/issues/3267
[47] „The Future of TypeORM”. Dostęp: 13 stycznia 2026. [Online]. Dostępne na: https://typeorm.io/docs/future-of-typeorm/
[48] M. Adámek, „Introducing MikroORM, TypeScript data-mapper ORM with Identity Map”, DailyJS. Dostęp: 14 stycznia 2026. [Online]. Dostępne na: https://medium.com/dailyjs/introducing-mikro-orm-typescript-data-mapper-orm-with-identity-map-9ba58d049e02
[49] „Serwis npmtrends.com - MikroORM”. Dostęp: 14 stycznia 2026. [Online]. Dostępne na: https://npmtrends.com/mikro-orm

[50] „Dokumentacja MikroORM - Usage with MySQL, MariaDB, PostgreSQL or SQLite”. Dostęp: 14 stycznia 2026. [Online]. Dostępne na: https://mikro-orm.io/docs/usage-with-sql
[51] „Dokumentacja MikroORM - identity map”. Dostęp: 14 stycznia 2026. [Online]. Dostępne na: https://mikro-orm.io/docs/identity-map
[52] M. Adámek, „MikroORM 6: Polished”, Medium. Dostęp: 14 stycznia 2026. [Online]. Dostępne na: https://itnext.io/mikroorm-6-polished-c03fbf10b917
[53] „MikroORM v7”, GitHub. Dostęp: 14 stycznia 2026. [Online]. Dostępne na: https://github.com/mikro-orm/mikro-orm/discussions/6116
[54] „Heroku: The Custom Cloud Application AI Platform”, Salesforce. Dostęp: 19 stycznia 2026. [Online]. Dostępne na: https://www.salesforce.com/heroku/
[55] „Stackhero for PostgreSQL”. Dostęp: 19 stycznia 2026. [Online]. Dostępne na: https://devcenter.heroku.com/articles/ah-postgresql-stackhero
[56] „Stackhero for MySQL”. Dostęp: 19 stycznia 2026. [Online]. Dostępne na: https://elements.heroku.com/addons/ah-mysql-stackhero
[57] „Dokumentacja Objection.js - Getting started”. Dostęp: 24 stycznia 2026. [Online]. Dostępne na: https://vincit.github.io/objection.js/guide/getting-started.html
[58] „Dokumentacja Sequelize - TypeScript”. Dostęp: 24 stycznia 2026. [Online]. Dostępne na: https://sequelize.org/docs/v6/other-topics/typescript/
[59] „Dokumentacja MikroORM - Collections”. Dostęp: 24 stycznia 2026. [Online]. Dostępne na: https://mikro-orm.io/docs/collections

Dodatki

Wykaz rysunków

Wykaz tabel

Wykaz wykresów

Wykaz listingów