

UNIwersytet Zielonogórski

Wydział Informatyki, Elektrotechniki i Automatyki

Praca dyplomowa

Kierunek: Informatyka

# BAZY DANYCH NoSQL JAKO ALTERNATYWA DLA TRADYCYJNYCH BAZ RELACYJNYCH

Krystian Dziędziola

Promotor:  
dr inż. Artur Gramacki

Zielona Góra, luty 2018



## Streszczenie

Celem pracy jest przedstawienie baz danych NoSQL jako alternatywy dla tradycyjnych baz relacyjnych. Programista stojący przed wyborem bazy danych dla aplikacji nie jest zmuszony używać zawsze tego samego typu, czyli najpopularniejszych baz relacyjnych. Pojawienie się baz NoSQL daje dodatkową możliwość wyboru typu bazy danych, który będzie odpowiedni dla danego systemu.

W pierwszej części pracy zawarte zostały informacje teoretyczne obejmujące takie zagadnienia jak sposób działania nierelacyjnych baz danych, charakterystykę różnych typów baz NoSQL oraz porównanie ich do tradycyjnych baz relacyjnych.

W drugiej części pracy zostało zaprezentowane praktyczne zastosowanie baz danych NoSQL. Tekst pracy zawiera opis zastosowania architektury aplikacji o nazwie CQRS oraz obu typów baz danych. Celem tych działań jest znaczące zwiększenie wydajność typowej aplikacji webowej. Stworzona została przykładowa aplikacja sklepu internetowego, dla której wykonane zostały testy wydajnościowe obrazujące wzrost wydajności po zastosowaniu baz NoSQL. Dodatkowo przedstawiony został sposób skalowania baz NoSQL, który umożliwił osiągnięcie jeszcze większej wydajności aplikacji.

Motywacją podjęcia pracy była chęć zaprezentowania możliwości nierelacyjnych baz danych, które w dobie Internetu XXI wieku są coraz częściej z powodzeniem wykorzystywane w największych światowych firmach.

**Słowa kluczowe:** Bazy danych, NoSQL, CQRS, skalowanie, wydajność.

# Spis treści

<b>1. Wstęp</b>	<b>1</b>
1.1. Wprowadzenie . . . . .	1
1.2. Cel i zakres pracy . . . . .	1
1.3. Przegląd literatury . . . . .	2
1.4. Struktura pracy . . . . .	2
<b>2. Podstawy NoSQL</b>	<b>4</b>
2.1. Definicja oraz pochodzenie NoSQL . . . . .	4
2.2. Charakterystyka . . . . .	5
2.2.1. Brak schematu . . . . .	5
2.2.2. Brak relacji . . . . .	5
2.2.3. Rezygnacja z SQL . . . . .	6
2.2.4. Przystosowanie do pracy w klastrach . . . . .	7
2.2.5. Stworzone na potrzeby XXI wieku . . . . .	8
2.3. Przyczyny powstania NoSQL . . . . .	8
2.3.1. Pierwsze bazy NoSQL . . . . .	8
2.3.2. Skalowanie . . . . .	8
2.3.3. Niezgodność impedancji . . . . .	10
2.4. Porównanie właściwości . . . . .	11
2.4.1. ACID . . . . .	11
2.4.2. Spójność danych . . . . .	12
2.4.3. Teoria CAP . . . . .	12
2.4.4. Warunkowa spójność danych . . . . .	13
2.5. Rodzaje baz NoSQL . . . . .	15
2.5.1. Klucz-wartość . . . . .	15
2.5.2. Kolumnowe . . . . .	16
2.5.3. Dokumentowe . . . . .	17
2.5.4. Grafowe . . . . .	17
2.6. Wady . . . . .	18
2.6.1. Brak transakcji . . . . .	18
2.6.2. Niedojrzałość . . . . .	19
2.6.3. Ograniczone funkcjonalności . . . . .	19
2.6.4. Brak wsparcia technicznego . . . . .	19
2.7. Podsumowanie . . . . .	19
<b>3. Architektura aplikacji wykorzystująca oba typy baz danych</b>	<b>20</b>
3.1. Architektura CQRS . . . . .	20
3.1.1. CQS jako zaczątek CQRS . . . . .	20
3.1.2. Klasyczny CQRS . . . . .	21

3.1.3.	Użycie w CQRS różnych typów baz danych . . . . .	21
3.1.3.1.	Spójność danych . . . . .	21
3.1.3.2.	Sposób przechowywania danych . . . . .	22
3.1.3.3.	Skalowalność . . . . .	22
3.1.3.4.	Wnioski . . . . .	22
3.2.	Wykorzystywane technologie . . . . .	23
3.2.1.	Ubuntu . . . . .	24
3.2.2.	Java 8 . . . . .	24
3.2.3.	Spring . . . . .	24
3.2.4.	Gradle . . . . .	24
3.2.5.	PostgreSQL . . . . .	25
3.2.6.	MongoDB . . . . .	25
3.2.7.	Docker . . . . .	25
3.2.8.	JMeter . . . . .	26
3.2.9.	Git . . . . .	26
3.2.10.	IntelliJ IDEA . . . . .	27
3.3.	Koncepcja aplikacji . . . . .	27
3.4.	Konfiguracja projektu - Gradle . . . . .	28
3.5.	REST . . . . .	30
3.5.1.	Protokół HTTP . . . . .	30
3.5.1.1.	Metody . . . . .	30
3.5.1.2.	Kody statusów . . . . .	31
3.5.2.	Dokumentacja API . . . . .	31
3.6.	Model danych . . . . .	32
3.6.1.	Produkt . . . . .	32
3.6.2.	Użytkownik . . . . .	33
3.6.3.	Zamówienie . . . . .	34
3.6.4.	Element zamówienia . . . . .	35
3.6.5.	Schemat danych . . . . .	35
3.7.	API . . . . .	36
3.7.1.	Kontrolery REST . . . . .	36
3.7.2.	Konfiguracja Swagger . . . . .	39
3.7.3.	Wysyłanie zapytań do aplikacji . . . . .	39
3.8.	Wykonywanie logiki biznesowej . . . . .	40
3.8.1.	Komendy oraz kwerendy . . . . .	41
3.9.	Komunikacja z bazą danych . . . . .	42
3.10.	Zmiana źródła odczytu danych . . . . .	43
3.10.1.	Przełącznik funkcjonalności . . . . .	44
3.10.2.	Wstrzykiwanie zależności oraz polimorfizm . . . . .	44
3.11.	Model danych do odczytu . . . . .	46
3.11.1.	Kolekcja produktów . . . . .	46
3.11.2.	Kolekcja użytkowników . . . . .	47
3.12.	Synchronizacja danych . . . . .	47
3.13.	Tworzenie obrazu Docker z aplikacją . . . . .	49
3.13.1.	Budowanie obrazu z linii komend . . . . .	50
3.13.2.	Budowanie obrazu za pomocą Gradle . . . . .	50
3.14.	Środowisko uruchomieniowe . . . . .	51
3.15.	Skalowanie . . . . .	53

3.15.1. Idea shardingu . . . . .	53
3.15.2. Strategie shardingu . . . . .	54
3.15.2.1. Hashed sharding . . . . .	54
3.15.2.2. Ranged sharding . . . . .	55
3.15.3. Przygotowanie środowiska do shardingu . . . . .	55
3.15.4. Konfiguracja shardingu . . . . .	58
3.16. Testy wydajnościowe . . . . .	60
3.16.1. Plan testów . . . . .	60
3.16.2. Przygotowanie danych . . . . .	61
3.16.3. Testy odczytu z bazy relacyjnej . . . . .	61
3.16.4. Testy odczytu z bazy NoSQL . . . . .	62
3.16.5. Testy odczytu z klastra baz NoSQL . . . . .	63
3.16.6. Podsumowanie . . . . .	64
<b>4. Zakończenie</b>	<b>67</b>
<b>A. Płyta DVD</b>	<b>68</b>

# Spis rysunków

2.1. Schemat struktury danych dla poszczególnych typów baz danych . . .	7
2.2. Rodzaje skalowania . . . . .	9
2.3. Przykładowe dane zamówienia . . . . .	10
2.4. Impedance mismatch - niezgodność struktury danych . . . . .	10
2.5. Podział klastra w wyniku awarii . . . . .	13
2.6. Zasada warunkowej spójności danych . . . . .	14
2.7. Struktura danych w bazach NoSQL typu klucz-wartość . . . . .	16
2.8. Struktura danych w kolumnowych bazach NoSQL . . . . .	16
2.9. Struktura danych w bazach dokumentowych w formacie JSON . . . .	17
2.10. Struktura danych w grafowych bazach NoSQL . . . . .	18
3.1. Architektura CQRS z pojedynczą bazą danych . . . . .	21
3.2. Architektura CQRS z dwoma typami baz danych . . . . .	23
3.3. Struktura relacyjnej bazy danych . . . . .	36
3.4. Spis dostępnych operacji na API . . . . .	37
3.5. Opis akcji . . . . .	40
3.6. Odpowiedź aplikacji . . . . .	40
3.7. Tryby pracy aplikacji . . . . .	43
3.8. Architektura systemu po zastosowaniu shardingu . . . . .	54
3.9. Podział danych na podstawie skrótu wartości klucza [1] . . . . .	55
3.10. Podział danych na podstawie zakresu wartości klucza [1] . . . . .	55
3.11. Wydajność relacyjnej bazy danych . . . . .	65
3.12. Wydajność pojedynczej bazy NoSQL . . . . .	65
3.13. Wydajność klastra baz NoSQL . . . . .	65

# Spis tabel

3.1.	Czasy odpowiedzi bazy relacyjnej - wyszukiwanie produktu . . . . .	62
3.2.	Czasy odpowiedzi bazy relacyjnej - wyszukiwanie użytkownika . . . . .	62
3.3.	Czasy odpowiedzi bazy NoSQL - wyszukiwanie produktu . . . . .	63
3.4.	Czasy odpowiedzi bazy NoSQL - wyszukiwanie użytkownika . . . . .	63
3.5.	Czasy odpowiedzi klastra baz NoSQL - wyszukiwanie produktu . . . . .	64
3.6.	Czasy odpowiedzi klastra baz NoSQL - wyszukiwanie produktu . . . . .	64



# Rozdział 1

## Wstęp

### 1.1. Wprowadzenie

SQL (ang. *Structured Query Language*) to standard opracowany w latach 60. przez firmę IBM, służący do komunikacji z serwerami relacyjnych baz danych. Stał się on bardzo popularny i jest używany do dnia dzisiejszego. Ze względu na język, który jest używany do komunikacji, relacyjne bazy danych zwane są często bazami SQL.

W informatyce na przestrzeni lat powstało wiele nowych technologii, języków, architektur oraz platform, jednak jedna rzecz pozostawała niezmienna - jeżeli istnieje potrzeba składowania danych, najczęściej używana jest relacyjna baza danych. Do niedawna wybór ograniczał się do tego, której bazy typu SQL użyć. Dopiero w XXI wieku popularność zaczął zyskiwać inny typ baz danych określany mianem NoSQL.

Bazy danych typu NoSQL istniały już o wiele wcześniej, jednak stały się popularne dopiero niedawno. Ze względu na to, że dostęp do internetu stał się znacznie bardziej powszechny, generowany jest coraz to większy ruch w sieci oraz przesyłanych jest o wiele więcej informacji. Wiele aplikacji internetowych potrzebuje przetwarzania olbrzymie ilości danych, co okazuje się wyzwaniem dla tradycyjnych baz SQL. Zaczęto szukać narzędzi rozwiązujących problemy, którymi dotknięte są bazy relacyjne.

To właśnie bazy NoSQL przedstawiane są jako alternatywa dla tradycyjnych, relacyjnych baz danych. Z założenia są one odpowiedzią na niedoskonałości baz SQL.

Czy oznacza to, że bazy NoSQL są pod każdym względem lepsze od SQL? Czy relacyjne bazy danych stracą wkrótce swoją popularność na rzecz nowego rozwiązania? Odpowiedzi na te pytania zostaną udzielone w rozdziale 2 przedstawiającym najważniejsze informacje dotyczące nierelacyjnych baz danych.

### 1.2. Cel i zakres pracy

Celem pracy jest odpowiedź na pytanie czy bazy NoSQL są w stanie zastąpić relacyjne bazy SQL. Dodatkowo praca jest swoistą bazą wiedzy na temat baz danych NoSQL. Wszystkie informacje przedstawione są w sposób umożliwiający osobie nie

znającej tematyki NoSQL pełne zrozumienie treści pracy oraz zdobycie wiedzy na podstawie opisanych zagadnień.

Omówione zostały również praktyczne zastosowania baz NoSQL w aplikacjach napisanych w języku Java z wykorzystaniem biblioteki Spring.

Zakres pracy, to przede wszystkim:

- charakterystyka różnych typów nierelacyjnych baz danych,
- porównanie baz danych NoSQL do SQL,
- przedstawienie sposobu użycia i zarządzania bazami NoSQL,
- zaprezentowanie architektury aplikacji, w której bazy NoSQL przynoszą znaczące korzyści,
- przeprowadzenie testów wydajnościowych potwierdzających słuszność proponowanego rozwiązania.

### 1.3. Przegląd literatury

Część pracy dotycząca teoretycznych pojęć związanych z bazami NoSQL została oparta głównie na pozycjach obszernie opisujących tę tematykę. W szczególności są to [2], [3] oraz [4].

Druga część pracy mówiąca o praktycznym zastosowaniu baz NoSQL razem z bazami relacyjnymi została napisana dzięki wiedzy pozyskanej z literatury na temat architektury CQRS oraz bazy danych MongoDB. Pozycje wykorzystywane w największym stopniu, to [5], [6] oraz [7].

### 1.4. Struktura pracy

Tekst pracy podzielony został na dwie zasadnicze części. Pierwszą z nich stanowi rozdział 2, w którym zawarte są podstawowe informacje na temat nierelacyjnych baz danych. W szczególności jest to:

- przedstawienie definicji oraz pochodzenia pojęcia NoSQL,
- charakterystyka oraz przyczyny powstania baz nierelacyjnych,
- właściwości baz NoSQL,
- porównanie NoSQL do baz relacyjnych,
- przedstawienie różnych rodzajów baz nierelacyjnych,
- omówienie wad baz NoSQL.

Drugą i zarazem najważniejszą częścią tekstu pracy jest rozdział 3, w którym opisane zostało zastosowanie nierelacyjnych baz danych wykorzystując architekturę CQRS. Stworzona została przykładowa aplikacja webowa emitująca sklep internetowy, na podstawie której omówione zostały zagadnienia takie jak:

- 
- tworzenie aplikacji w oparciu o architekturę CQRS,
  - technologie wykorzystywane do tworzenia aplikacji webowych,
  - sposób komunikacji z aplikacją,
  - model danych,
  - konfiguracja aplikacji,
  - automatyzacja tworzenia środowiska uruchomieniowego aplikacji,
  - skalowanie bazy danych MongoDB,
  - testy wydajnościowe.

# Rozdział 2

## Podstawy NoSQL

### 2.1. Definicja oraz pochodzenie NoSQL

Termin NoSQL został użyty po raz pierwszy w 1998 roku [2], jednak wtedy nie miał on nic wspólnego z bazami danych, które współcześnie określa się mianem NoSQL. Początkowo określenie to posłużyło do nazwania relacyjnej bazy danych, która po prostu nie używała standardu SQL.

Dopiero w 2009 roku termin ten został powtórnie użyty, tym razem w zupełnie innym kontekście. Określenie NoSQL posłużyło jako nazwa konferencji zorganizowanej przez Johana Oskarssona w dniu 11.06.2009 w San Francisco. Były na niej prezentowane nowe typy bazy danych takie jak np. Hypertable, HBase czy Cassandra. Bazy te stanowiły alternatywę dla tradycyjnych baz SQL, gdyż były nierelacyjne oraz do przechowywania danych nie używały tabel. Miały one być odpowiedzią na niedoskonałości baz relacyjnych oraz rozwiązywać problemy tychże baz. Od tego czasu nazwa NoSQL została przyjęta jako określenie właśnie takich typów baz danych [2].

W wyniku tego, że termin ten powstał dosyć spontanicznie nie ma jego ścisłej definicji. Jako NoSQL można określić ruch, który miał na celu odnalezienie alternatyw dla tradycyjnych baz relacyjnych. Eric Evans, który również uczestniczył w konferencji w San Francisco powiedział później, że “głównym celem szukania alternatyw jest to, że istnieje potrzeba rozwiązania problemów, z którymi spotykają się relacyjne bazy danych”. [8]

Mimo, iż nie istnieje ścisła definicja terminu NoSQL istnieją kryteria, które pozwalają określić pewne bazy mianem NoSQL. Martin Fowler powiedział, że “NoSQL jest przypadkowym neologizmem - nie posiada ściśle określonej definicji. Nie ma również kogoś, kto mógłby zdefiniować to pojęcie. Jedyne co można zrobić, to omówić ich pewne wspólne cechy.” [2]

Oczywiście przed pojawieniem się baz, które aktualnie określane są mianem NoSQL miały miejsce próby stworzenia innych typów baz danych, takich jak np. bazy obiektowe czy bazy danych oparte na XML. Nie zostały one jednak zaklasyfikowane do NoSQL ze względu na to, że nie były one próbą odnalezienia odpowiedzi na problemy baz relacyjnych, a jedynie czymś niezależnym, stworzonym do innych zastosowań. Tego typu bazy nie będą więc omawiane w tej pracy.

## 2.2. Charakterystyka

Czym więc charakteryzują się bazy danych określane mianem NoSQL? Można wyróżnić kilka ich głównych cech:

- nie posiadają określonego schematu danych,
- brak relacji - tabel oraz więzów integralności,
- przystosowane do pracy w klastrach,
- nie używają języka SQL,
- stworzone na potrzebny Internetu XXI wieku.

### 2.2.1. Brak schematu

Główną cechą baz danych NoSQL jest to, że nie posiadają określonego schematu danych. W przypadku baz typu SQL tabele gwarantują, że dane posiadają prawidłową strukturę. Każdą porcję danych, którą chcemy umieścić w bazie relacyjnej musimy najpierw przygotować tak, aby odpowiadały ustalonemu schematowi. W przypadku potrzeby umieszczenia danych, które mają inną strukturę, niż stworzone tabele, należy zmodyfikować schemat bazy danych. Może to jednak okazać się problematyczne i czasochłonne.

Natomiast, gdy chcemy zapisać dane do bazy NoSQL nie musimy przejmować się strukturą danych. Jeżeli istnieje potrzeba dodania nowego pola do konkretnej porcji danych nie niesie to za sobą zmiany schematu bazy. Jest to szczególnie przydatne, gdy mamy do czynienia z danymi, które nie posiadają spójnej struktury, gdyż brak schematu umożliwia tworzenie tzw. heterogenicznych (ang. *heterogenous*) struktur danych.

Przyspiesza to również proces tworzenia oprogramowania, szczególnie na wczesnych etapach projektu, gdy struktura bazy danych może dynamicznie ulegać zmianom. W takiej sytuacji programista nie musi poświęcać czasu na zajmowanie się takimi technicznymi problemami jak zmiana struktury bazy. Zamiast tego może skupić się na tym co jest ważne, czyli na implementacji logiki biznesowej aplikacji.

### 2.2.2. Brak relacji

Kolejną cechą baz NoSQL jest to, że nie używają one więzów integralności do łączenia danych, które są w pewien sposób ze sobą związane. Często istnieje jednak potrzeba, żeby połączyć ze sobą jakieś zestawy danych. Przykładem może być zamówienie w sklepie internetowym. W przypadku zakupu kilku produktów w ramach jednego zamówienia, każdy z produktów powinien być powiązany z głównym zamówieniem.

W bazach relacyjnych istniałyby tabele przechowujące osobno poszczególne produkty oraz tabela zamówień. Każdy z produktów byłby połączony z zamówieniem za pomocą relacji. Potrzeba wprowadzania relacji wynika z tego, iż dane przechowywane w tabelach nie mogą posiadać zagnieżdżeń, gdyż tabela jest płaską strukturą danych.

Jak jednak dokonać tego w bazach NoSQL? Dzięki temu, że bazy NoSQL obsługują struktury danych posiadające zagnieżdżenia, można przechowywać dane w bardziej naturalny sposób - są one skupiane w formie tzw. agregatów.

Pojęcie agregatu (ang. *aggregate*) wywodzi się z książki Erica Evansa pt. *Domain-Driven Design* [9]. Według Evansa agregat, to zbiór powiązanych ze sobą danych, które są traktowane jako jednostka. Bazy NoSQL korzystają z idei agregatów, gdyż właśnie w taki sposób przechowują dane. Nawiązując do przykładu z zamówieniem sklepowym, to właśnie zamówienie jest agregatem, który scala (agreguje) wszystkie powiązane z nim produkty. W konsekwencji dane te będą przechowywane w bazie danych jako nierozłączna całość, a nie tak jak w przypadku baz relacyjnych - rozbite na osobne tabele. Eliminuje to potrzebę wykonywania na bazie operacji łączenia danych (ang. *join*), co znacznie upraszcza tworzenie zapytań służących do pobierania danych oraz wprowadza znaczący wzrost wydajności.

Dane w bazach NoSQL są często przechowywane w formacie JSON (ang. *JavaScript Object Notation*). Omawiane zamówienie sklepowe w tym formacie mogłoby wyglądać następująco:

---

```
1 {
2   "order": {
3     "id": 1,
4     "name": "zamówienie_1",
5     "items": [
6       {
7         "id": 400400,
8         "name": "Komputer PC",
9         "price": {
10          "value": 3000,
11          "currency": "PLN"
12        }
13      },
14      {
15        "id": 300300,
16        "name": "Monitor",
17        "price": {
18          "value": 800,
19          "currency": "PLN"
20        }
21      }
22    ]
23  }
24 }
```

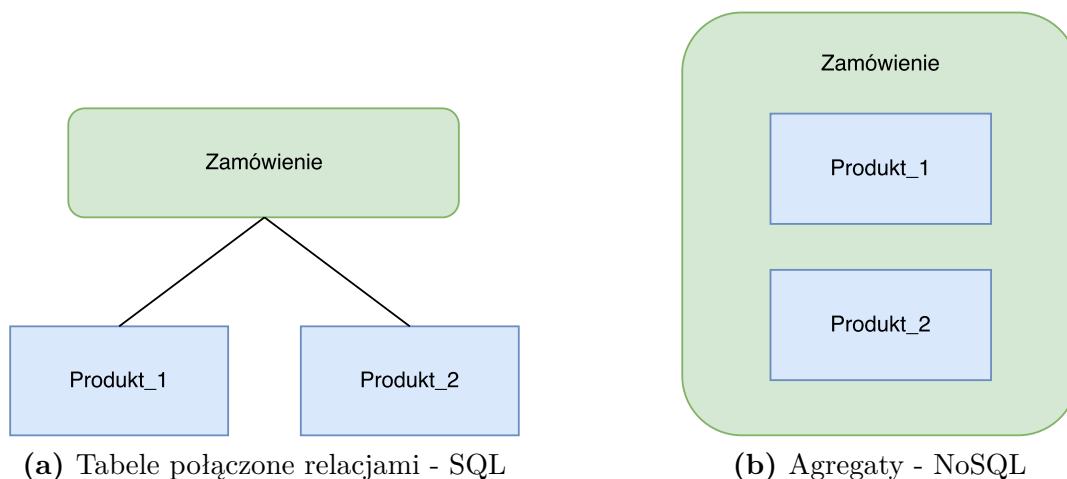
---

Jak można zauważyć, poszczególne produkty (*items*) są przechowywane bezpośrednio wewnątrz zamówienia (*order*). Rysunek 2.1 pokazuje różnice w sposobie przechowywania danych w bazach typu SQL oraz NoSQL.

### 2.2.3. Rezygnacja z SQL

Język SQL został stworzony do pracy z danymi, które posiadają strukturę wykorzystywaną w relacyjnych bazach danych, czyli przechowywane są w formie tabel połączonych ze sobą relacjami. W bazach NoSQL istnieje całkowicie inna struktura danych, która różni się w zależności od rodzaju bazy NoSQL. Zostało to opisane w rozdziale 2.5.

Następstwem przechowywania danych w zupełnie inny sposób jest to, że język SQL nie znajduje zastosowania w bazach NoSQL, gdyż jest on przystosowany do pracy z danymi w zupełnie innej postaci. Używanie języka SQL byłoby jedynie pro-



**Rys. 2.1.** Schemat struktury danych dla poszczególnych typów baz danych

blematyczne i nie przynosiłoby żadnych korzyści. Bazy NoSQL posiadają stosunkowo proste API (ang. *Application Programming Interface*), które służy do przeprowadzania operacji na danych.

Mimo, że standard SQL nie jest używany, to niektóre bazy danych NoSQL takie jak np. Cassandra używają języków, które wywodzą się z języka SQL. W przypadku Cassandra jest to język CQL (ang. *Cassandra Query Language*).

#### 2.2.4. Przystosowanie do pracy w klastrach

Kolejną bardzo istotną cechą jest przystosowanie baz danych NoSQL do pracy w klastrach.

Tradycyjne bazy relacyjne zostały zaprojektowane z myślą o pracy na tylko jednej maszynie. Wszystkie dane muszą znajdować się na jednym serwerze z uwagi na m.in. mechanizmy zapewniania integralności danych, które posiadają bazy SQL.

Istnieją specjalne bazy danych SQL, które posiadają wsparcie dla pracy w klastrach np. Microsoft SQL Server lub Oracle RAC, jednak te rozwiązania działają poprzez współdzielenie przestrzeni dyskowej pomiędzy poszczególnymi maszynami w klastrze. Niesie to jednak za sobą podatność na awarię całego klastra, gdy system plików ulegnie uszkodzeniu.

Oczywiście istnieje możliwość przechowywania danych w bazach SQL na kilku niezależnych serwerach (ang. *sharding*). Wadą tego podejścia jest to, że gdy aplikacja chce pobrać z bazy dane musi wiedzieć dokładnie, na którym serwerze się one znajdują. Dodatkowo uniemożliwia to korzystanie z zapytań, które łączą dane znajdujące się na oddzielnych maszynach. Niemożliwe jest również wykorzystanie transakcji czy zapewnienie integralności danych rozmieszczonych na osobnych serwerach. Martin Fowler pisze, że ludzie, którzy próbowali uruchamiać bazy relacyjne w klastrach często mówią, że jest to bardzo nienaturalne podejście. [2]

Bazy danych typu NoSQL zostały zaprojektowane tak, aby praca w klastrach była dla nich czymś naturalnym. Umożliwia to struktura danych oparta na agregatach oraz to, że rezygnują z niektórych mechanizmów zapewniania integralności danych na rzecz tzw. ewentualnej spójności danych (ang. *eventual consistency*), co

zostało opisane w rozdziale 2.4.4.

Korzyściami wynikającymi z tego faktu, że bazy NoSQL są przystosowane do pracy w klastrach jest znacznie lepsza skalowalność, niż w przypadku baz relacyjnych oraz odporność na awarie. Skalowalność została opisana w rozdziale 2.3.2.

### 2.2.5. Stworzone na potrzeby XXI wieku

Jak już zostało powiedziane, na przestrzeni lat powstało wiele prób stworzenia nowych typów baz danych, które będą różnić się od tradycyjnych baz relacyjnych. Pomimo, że bazy te posiadają niektóre cechy, dzięki którym można by je było zaklasyfikować do NoSQL, to istnieje pewien czynnik, który nie pozwala określić ich tym terminem.

Bardzo ważną charakterystyką baz danych NoSQL jest to, że powstały one na potrzeby Internetu XXI wieku. Oznacza to, że główną motywacją do stworzenia tego typu baz było rozwiązanie problemów, z którymi zaczęły zmagać się relacyjne bazy danych wkraczając w erę tzw. Internetu rzeczy (ang. *Internet of Things*).

Przyczyny powstania ideologii NoSQL zostaną bardziej szczegółowo omówione w rozdziale 2.3.

## 2.3. Przyczyny powstania NoSQL

### 2.3.1. Pierwsze bazy NoSQL

Została już przedstawiona geneza terminu NoSQL, jednak skąd wzięły się pierwsze bazy danych typu NoSQL?

Wraz z początkiem XXI wieku nastąpiła nowa era Internetu. W dzisiejszych czasach prawie każdy ma dostęp do sieci. Oznacza to, że serwisy internetowe zyskują z dnia na dzień coraz więcej użytkowników, dlatego istnieje potrzeba magazynowania oraz przetwarzania coraz większej ilości danych.

Największe serwisy internetowe w pierwszej kolejności zauważyły, że relacyjne bazy danych nie są przystosowane do tak dużej ilości danych. Firmy Google oraz Amazon, które posiadają ogromną liczbę użytkowników postanowiły stworzyć swoje własne bazy danych, które byłyby przystosowane do tzw. *Big Data*, czyli przechowywania oraz przetwarzania wielkiej ilości danych. Google stworzyło bazę o nazwie *Bigtable*, natomiast w Amazon powstała baza *Dynamo*. Były to pierwsze bazy danych, które mogły sprostać wymaganiom stawianym przez Internet XXI wieku.

### 2.3.2. Skalowanie

Pierwszym i zarazem najważniejszym z wymagań stawianym bazom danych używanych w aplikacjach internetowych jest skalowalność. Skalowalność oznacza zdolność systemu do rozbudowy i przystosowania się do narastającego obciążenia.

Serwis posiadający coraz więcej użytkowników w końcu napotka problemy wydajnościowe. Należy wtedy rozważyć zwiększenie wydajności serwisu, a w szczególności bazy danych. Można tego dokonać poprzez skalowanie.



Istnieją dwa typy skalowania:

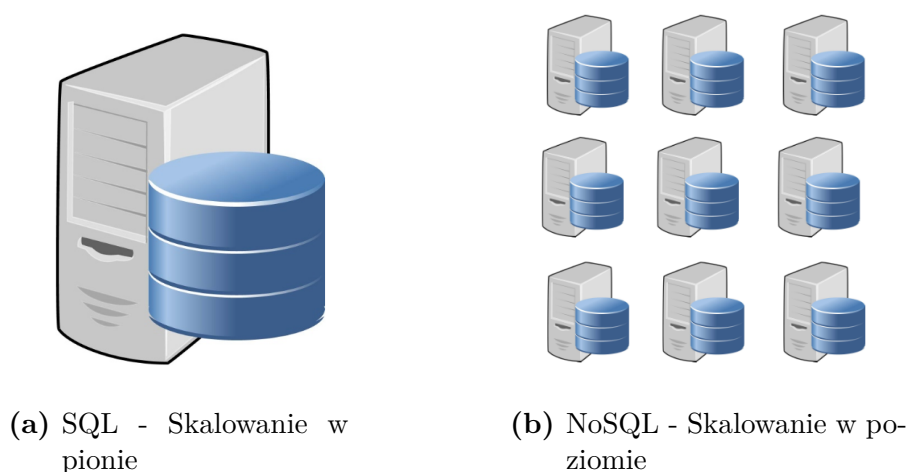
- skalowanie w pionie (ang. *vertical scaling*)
- skalowanie w poziomie (ang. *horizontal scaling*).

Pierwszy typ skalowania, czyli skalowanie w pionie polega na tym, że posiadając jeden serwer wyposażamy go w większą moc obliczeniową, np. poprzez dodanie większej liczby procesorów, pamięci operacyjnej czy też rozbudowę przestrzeni dyskowej. Zaletą tego podejścia jest jego prostota, jednak niesie za sobą kilka poważnych wad. Przede wszystkim jest to stosunkowo drogie podejście, gdyż ceny sprzętu rosną nieproporcjonalnie względem ich wydajności. Dodatkowo istnieje pewna granica jak bardzo rozbudować można jedną maszynę, gdyż do jednego serwera można dołożyć tylko skończoną ilość procesorów oraz innych komponentów.

Drugim, znacznie wydajniejszym podejściem jest skalowanie w poziomie. Polega ono na tym, że zamiast zwiększać moc obliczeniową jednego serwera, należy stworzyć tzw. klaster (ang. *cluster*), czyli sieć połączonych ze sobą komputerów, które pracują razem jako jedna, potężna maszyna. Zaletą klastrów jest to, że są bardzo dobrze skalowalne. Kiedy występuje potrzeba zwiększenia wydajności klastra wystarczy dołożyć do niego kolejny komputer. Dodatkowo klastry są bardziej odporne na awarię, gdyż szansa na to, że wszystkie maszyny w tej samej chwili ulegną awarii są znikome. Uszkodzenie jednej maszyny nie wpływa znacząco na pracę całego klastra.

Można zauważyć, że skalowanie w poziomie daje wiele korzyści i umożliwia prostszą rozbudowę. Niestety, relacyjne bazy danych są przystosowane jedynie do pracy na jednej maszynie i wykorzystanie ich do pracy w klastrze jest bardzo trudne. Z kolei bazy typu NoSQL zostały stworzone właśnie w tym celu, dlatego umożliwiają one znacznie lepszą, prostszą oraz tańszą możliwość skalowania. Jest to jeden z głównych argumentów na to, aby w aplikacji internetowej mającej problemy z wydajnością użyć właśnie bazy NoSQL.

Na rysunku 2.2 przedstawione są oba rodzaje skalowania.

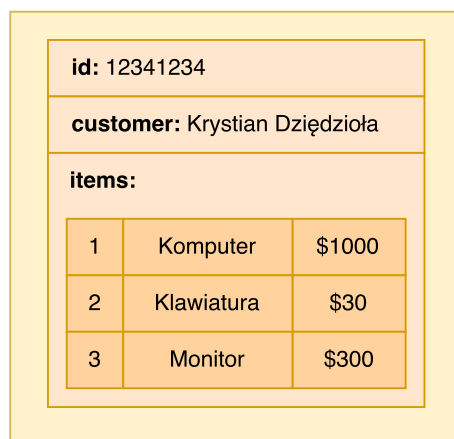


**Rys. 2.2.** Rodzaje skalowania

### 2.3.3. Niezgodność impedancji

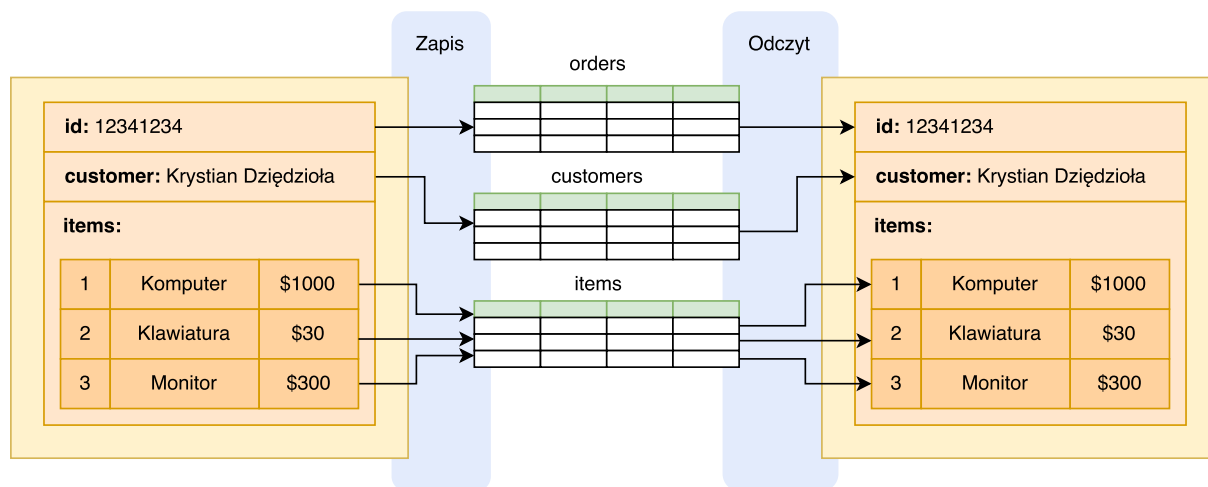
Kolejnym powodem powstania baz danych typu NoSQL jest problem, który w języku angielskim nosi nazwę *impedance mismatch*. Termin ten został przetłumaczony na język polski jako *niezgodność impedancji* [10], jednak nazwa sama w sobie brzmi dosyć enigmatycznie. Jest to problem dotyczący złego odzwierciedlenia struktury danych w bazach SQL.

Za przykład można wziąć dane reprezentujące szczegóły zamówienia w sklepie internetowym, które zostały pokazane na rysunku 2.3.



Rys. 2.3. Przykładowe dane zamówienia

Przedstawione dane są jedną logiczną strukturą, gdy wyświetlane są na interfejsie użytkownika, składają się one również na jedną wspólną całość, gdy przechowywane są jako obiekty w pamięci oraz gdy implementowane są w kodzie przez programistę. Nie jest jednak możliwe zapisanie do bazy SQL danych w zaprezentowanej strukturze, gdyż tabele przechowują jedynie proste typy danych, które nie mogą posiadać zagnieżdżeń. W procesie zapisywania oraz odczytywania danych z bazy relacyjnej dane muszą ulegać więc zmianie struktury, czyli tzw. mapowaniu (ang. *mapping*).



Rys. 2.4. Impedance mismatch - niezgodność struktury danych

Właśnie na tym polega problem określany jako *impedance mismatch*. Jest to niezgodność pomiędzy modelem relacyjnym a rzeczywistą strukturą danych. Takie

podejście reprezentacji danych w bazie niesie za sobą kilka poważnych wad.

Mapowanie danych do modelu relacyjnego jest konieczne, jednak powoduje pewne problemy. Z uwagi na dużą popularność obiektowych języków programowania, preferowany jest obiektowy sposób przedstawiania oraz przechowywania danych. Dzielenie danych i umieszczanie ich w tabelach, a następnie scalanie ich w całość przy próbach pobrania niesie za sobą dodatkowe komplikacje. Jest to szczególnie uciążliwe dla rozbudowanych struktur danych.

Problem ten nie jest w dzisiejszych czasach aż tak uciążliwy z uwagi na biblioteki służące do mapowania relacyjno-obiektowego, czyli ORM (ang. object-relational mapping). Przykładem biblioteki ORM jest *Hibernate*.

Biblioteki te co prawda znacznie upraszczają tworzenie zapytań do bazy danych, jednak niosą za sobą niewielkie problemy wydajnościowe. Nowe rozwiązania są w stanie konstruować stosunkowo wydajne zapytania, jednak przeważnie nie są one optymalne. Lepsze wyniki można uzyskać idąc trudniejszą drogą, czyli pisząc zapytania SQL ręcznie.

Kolejnym problemem relacyjnego modelu danych jest wydajność. Operacje łączenia danych (ang. *join*) w dzisiejszych bazach danych są odpowiednio zoptymalizowane i działają wydajnie, jednak sama potrzeba dzielenia oraz łączenia danych powoduje spadki wydajności w porównaniu do sytuacji, w której można uniknąć wykonywania tych operacji.

Bazy typu NoSQL dzięki strukturze danych w formie agregatów pozwalają uniknąć problemu mapowania danych. Umożliwia to osiągnięcie znacznie większej wydajności przy operacjach zapisu i odczytu danych oraz ułatwia korzystanie z bazy danych, co przekłada się na wydajność programistów, gdyż nie muszą oni tracić czasu na pisanie skomplikowanych zapytań SQL.

## 2.4. Porównanie właściwości

Relacyjne bazy danych różnią się od NoSQL pod wieloma względami. Najistotniejszą różnicą są właściwości poszczególnych typów baz danych.

### 2.4.1. ACID

Bazy danych SQL dzięki transakcyjności oferują dużą niezawodność. Definiuje się ich cztery właściwości określone akronimem ACID (ang. *Atomicity*, *Consistency*, *Isolation*, *Durability*) [11].

- *Atomicity* - atomowość transakcji. Oznacza to, że operacje wykonywane w obrębie jednej transakcji powinny być atomowe, czyli niepodzielne. W przypadku wystąpienia błędu w którejkolwiek z operacji, cała transakcja powinna zostać anulowana i nie powinny być wprowadzone żadne zmiany. System posiadający tę własność powinien zapewniać atomowość transakcji bez względu na typ i przyczynę błędu. Wlicza się w to również awarie zasilania i sprzętowe oraz awarie systemu,
- *Consistency* - spójność danych. Jest to właściwość zapewniająca, że dane po wykonaniu transakcji zostaną zmienione tylko i wyłącznie w dozwolony sposób

i pozostaną w prawidłowym stanie. Oznacza to, że każde zapisywane dane muszą spełniać zdefiniowane reguły takie jak ograniczenia (ang. *constraints*),

- *Isolation* - izolacja transakcji. Właściwość ta zapewnia, że wyniki transakcji wykonywanych wielowątkowo (równolegle) będą takie same jak w przypadku wykonania ich sekwencyjnie (jedna po drugiej). Skutkuje to tym, że poszczególne transakcje nie mają dostępu do niekompletnych zmian wprowadzanych przez inne transakcje,
- *Durability* - trwałość danych. Oznacza to, że w momencie, gdy transakcja zostanie wykonana, zmiany które zostały wprowadzone nie powinny być utracone np. w przypadku wystąpienia awarii. W celu obrony przed utratą zmian w przypadku odcięcia zasilania rezultaty wykonywanej transakcji muszą być przechowywane w pamięci nieulotnej (ang. *non-volatile memory*).

### 2.4.2. Spójność danych

Spójność danych w bazach jest czymś bardzo ważnym, a wręcz wydawałoby się, że niezbędnym. Czasami istnieją jednak sytuacje, w których trzeba zrezygnować z zapewnienia stuprocentowej spójności danych na rzecz innych właściwości.

Przeważnie możliwe jest zaprojektowanie systemu w taki sposób, aby uniknąć niespójności danych. Często jednak wiąże się to z dużymi poświęceniami odnośnie innych parametrów systemu takich jak wydajność bądź skalowalność. Przy projektowaniu systemu należy zwrócić uwagę jak ważna jest spójność danych w kontekście danej aplikacji.

W relacyjnych bazach danych mechanizmem zapewniającym spójność danych są transakcje. Są one niezawodne, jednak niosą za sobą dosyć duże problemy wydajnościowe. Już nawet w kontekście tego typu baz można zaobserwować rezygnację z próby zapewnienia stuprocentowej spójności danych. Często systemy transakcyjne mają możliwość zmniejszenia stopnia izolacji w celu zwiększenia wydajności.

Niektóre wielkie firmy takie jak np. *eBay* czy *Facebook* musiały zastąpić część baz relacyjnych bazami NoSQL, aby ich serwisy działały z chociażby akceptowalną prędkością. [3]

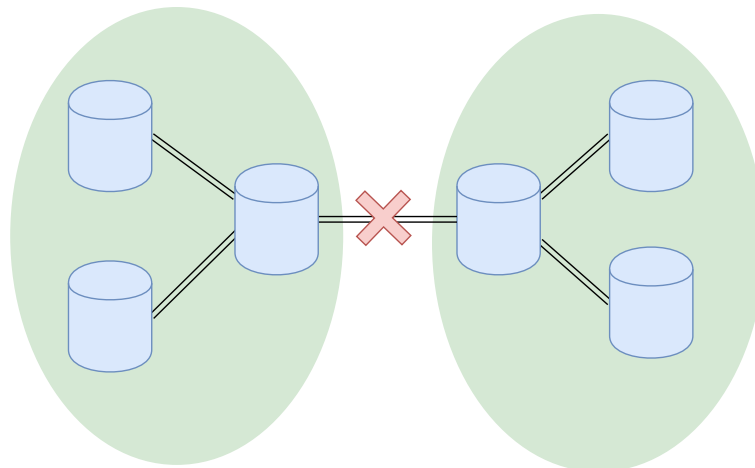
### 2.4.3. Teoria CAP

Teoria CAP (ang. *Consistency, Availability, Partition tolerance*) została zaproponowana przez Erica Brewera w 2000 roku [12] a następnie udowodniona kilka lat później przez Setha Gilberta oraz Nancy Lynch.

Podstawą teorii Brewera są trzy właściwości systemów rozproszonych:

- *Consistency* - spójność danych. Oznacza, że dane mogą być zmienione jedynie w zdefiniowany sposób i zawsze muszą być poprawne względem wyznaczonych reguł. Jest to jedna z cech ACID 2.4.1,
- *Availability* - dostępność. Jest to cecha systemu, która mówi o tym, że gwarantuje on odpowiedź (nie będącą błędem) na każde odebrane żądanie,

- *Partition tolerance* - odporność na partycjonowanie. Oznacza to, że system jest odporny na sytuację, w której np. w wyniku awarii klastr straci łączność pomiędzy niektórymi węzłami i zostanie podzielony na dwie odrębne części, które nie mogą się ze sobą komunikować. Ilustruje to poniższy rysunek.



Rys. 2.5. Podział klastra w wyniku awarii

Teoria CAP mówi o tym, że nie jest możliwe zapewnienie wszystkich trzech właściwości systemu. Systemy jedno-serwerowe, takie jak większość baz relacyjnych określa się jako CA (*Consistency, Availability*), czyli charakteryzujące się spójnością danych oraz dostępnością, jednak nie gwarantują one odporności na partycjonowanie, gdyż ciężko jest mówić o podziale w przypadku jednej maszyny.

Bazy danych NoSQL przeważnie pracują w klastrach, więc wymagana jest odporność na partycjonowanie. Zgodnie z teorią CAP należy wtedy dokonać wyboru pomiędzy dostępnością a spójnością, gdyż w systemie rozproszonym można zapewnić jedynie dwie z trzech cech.

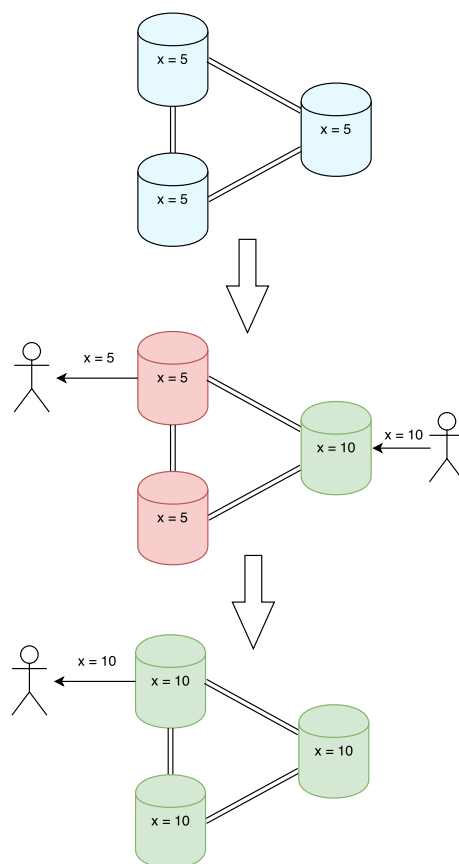
#### 2.4.4. Warunkowa spójność danych

Bazy danych NoSQL nie oferują właściwości ACID. Czasami ich właściwości są określane akronimem BASE (ang. *Basically Available, Soft-state, Eventual consistency*).

BASE oznacza, że baza zapewnia wysoką dostępność oraz odporność na partycjonowanie kosztem spójności danych. Nie oznacza to, że przechowywane dane są całkowicie niespójne, gdyż byłoby to niedopuszczalne. Zamiast spójności pod postacią tej zdefiniowanej w teorii CAP bazy NoSQL oferują tzw. warunkową spójność (ang. *Eventual consistency*).

Zasada warunkowej spójności mówi o tym, że pobierane dane nie zawsze muszą być aktualne. Gwarantuje jednak, że dane te będą spójne po upływie skończonej ilości czasu (przeważnie od kilku milisekund do kilku sekund). Jest to spowodowane tym, że możliwy jest odczyt danych podczas wykonywania ich aktualizacji np. na jednym węźle klastra dane zostały zaktualizowane i zmiana musi być rozpropagowana na inne węzły, z których w tym czasie dane mogą zostać pobrane. Wydaje się to być czymś niedopuszczalnym, jednak okazuje się, że w większości przypadków jest to całkowicie akceptowalne.

Warunkowa spójność danych przedstawiona jest na rysunku 2.6.



**Rys. 2.6.** Zasada warunkowej spójności danych

Przykładem takiej sytuacji może być korzystanie przez użytkownika z wyszukiwarki artykułów na blogu lub wiadomości w serwisie z wiadomościami. Jak często zdarza się, że użytkownik wyszuka artykuł akurat w momencie jego dodania? Jest to raczej rzadka sytuacja, a nawet jeżeli ma już miejsce, to nie jest to nic złego, gdyż zanim zdąży on odświeżyć stronę, dane będą już aktualne.

Kolejnym przykładem może być zakup produktu w sklepie internetowym. Użytkownik widzi, że produkt jest dostępny (choć ostatnia sztuka została przed momentem sprzedana) i naciska przycisk *kup*. System oczywiście nie pozwoli zakupić produktu, gdyż w tym momencie powinna być zapewniona spójność danych, aby nie doprowadzić do zakupu przedmiotów, które są niedostępne. Po odświeżeniu strony użytkownik zobaczy, że niestety produkt nie jest już dostępny.

Z kolei aplikacją, która nie może zrezygnować z absolutnej spójności danych jest na przykład serwis bankowy, w którym użytkownik nie może mieć możliwości wypłacenia z konta pieniędzy, których aktualnie nie posiada.

Przykład ze sklepem internetowym pokazuje, że nawet w obrębie jednej aplikacji istnieją miejsca, w których ewentualna spójność jest wystarczająca, natomiast w innych jest nieakceptowalna. Widać tutaj, że do zbudowania takiej aplikacji najlepiej nadawałyby się oba typy baz danych. Okazuje się, że jest to możliwe do zrealizowania. Takie podejście do budowania aplikacji nosi nazwę *Polyglot Persistence* i oznacza używanie różnych typów baz danych w jednym systemie, bądź w jednej aplikacji.

Przykładem architektury aplikacji wykorzystującej oba typy baz danych jest architektura o nazwie CQRS (ang. *Command Query Responsibility Segregation*), która zostanie omówiona w rozdziale 3.1.2.

## 2.5. Rodzaje baz NoSQL

Termin NoSQL jest określeniem, które obejmuje kilka rodzajów baz danych. Poszczególne typy różnią się od siebie sposobem składowania informacji. Przeważnie dzieli się je na cztery grupy:

- klucz-wartość,
- dokumentowe,
- grafowe,
- kolumnowe.

W tej części zostaną pokrótce omówione wszystkie wymienione typy baz danych NoSQL.

### 2.5.1. Klucz-wartość

Bazy danych NoSQL typu klucz-wartość (ang. *key-value*) przechowują dane w postaci mapy, czyli prostej struktury danych zawierającej klucz oraz powiązaną z nią wartość. Są one użyteczne w sytuacji, gdy dane pobierane są przeważnie po kluczu głównym np. identyfikatorze. Bazy klucz-wartość są najprostsze w użyciu spośród wszystkich typów baz NoSQL.

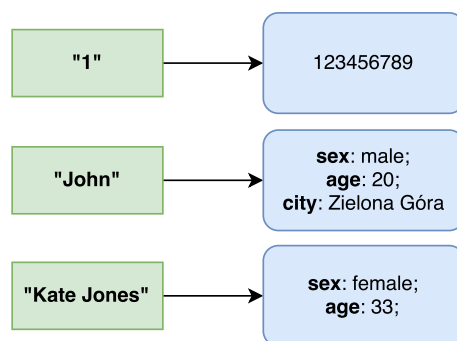
Typowe operacje dostępne dla użytkownika podobne są do tych, które wykonywać można na hash mapie:

- *insert* - dodanie pary klucz-wartość do bazy. W przypadku, gdy klucz istnieje wartość jest nadpisywana,
- *lookup* - wyszukanie wartości na podstawie klucza,
- *delete* - usunięcie pary klucz-wartość o danym kluczu.

Klucz musi być unikalny, gdyż jest identyfikatorem, natomiast jako wartość można przechowywać zróżnicowane struktury danych np. liczby, ciągi znaków, obiekty lub listy obiektów. Ilustruje to rysunek 2.7

Baza danych nie pilnuje typu przechowywanej wartości - jest to zadaniem aplikacji. Z uwagi na to, że zawsze używają one klucza głównego do pobierania wartości cechują się dużą wydajnością oraz umożliwiają łatwą skalowalność kosztem elastyczności i ograniczonych możliwości.

Przykładami tego typu baz danych są m.in. *Riak*, *Redis*, *Memcached DB*, *Berkley DB*, *HamsterDB*, *Project Voldemort* i wiele innych.

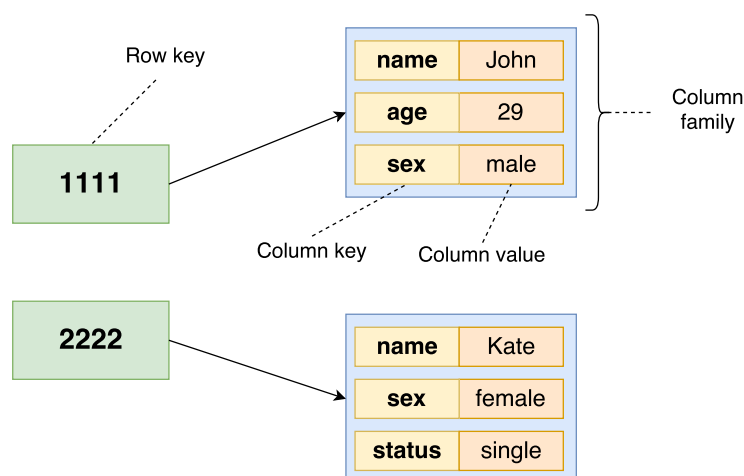


Rys. 2.7. Struktura danych w bazach NoSQL typu klucz-wartość

### 2.5.2. Kolumnowe

Kolumnowe bazy danych NoSQL (ang. *column-family*) umożliwiają przechowywanie danych w postaci pogrupowanych rodzin kolumn (ang. *column family*), do których można odwołać się za pomocą klucza zwanego kluczem wiersza (ang. *row key*). Rodziny kolumn są grupami powiązanych ze sobą danych, które są często przetwarzane razem. Każda z kolumn jest z kolei parą klucz-wartość, w której kluczem jest nazwa kolumny (ang. *column key*).

Bazy kolumnowe posiadają więc strukturę dwupoziomowej, zagnieżdżonej mapy. Można ją porównać do tabeli z baz relacyjnych, gdyż też jest to zbiór wierszy i kolumn, jednak z tą różnicą, że w bazach kolumnowych każdy wiersz może posiadać różną liczbę kolumn oraz typów przechowywanych wartości w zależności od potrzeb. Strukturę danych w bazach kolumnowych ilustruje rysunek 2.8.



Rys. 2.8. Struktura danych w kolumnowych bazach NoSQL

Zaletą takiego podejścia jest bardzo duża elastyczność pod względem dodawania nowych kolumn w przyszłości w związku z nieprzewidywanymi zmianami. Dodatkowo taka struktura danych umożliwia znacznie wydajniejszy odczyt poszczególnych wartości z kolumn, gdyż nie istnieje potrzeba pobierania całego wiersza i dopiero wyciągania z niego wartości konkretnej kolumny tak jak ma to miejsce w przypadku baz relacyjnych.

Najbardziej popularne kolumnowe bazy NoSQL to m.in. *Apache Cassandra*, *HBase*, *Google BigTable*, *Hypertable* oraz *Amazon DynamoDB*.



### 2.5.3. Dokumentowe

W dokumentowych bazach danych, które nazywane są również bazami zorientowanymi dokumentowo (ang. *document-oriented databases*) główną jednostką składowania informacji są właśnie dokumenty, których formaty mogą się różnić w zależności od konkretnej implementacji. Najbardziej popularnymi formatami są *XML*, *JSON*, *BSON* oraz *YAML*. Dokumenty mają strukturę drzewa, które może zawierać wartości, mapy oraz listy. Są one grupowane w tzw. kolekcje (ang. *collections*).

Dane przechowywane w dokumentach posiadają pewną strukturę, która nie jest tak restrykcyjna jak w bazach relacyjnych, dlatego są one nazywane danymi w pewnym stopniu uporządkowanymi (ang. *semi-structured data*). Wynika to z tego, że muszą one spełniać pewne kryteria względem struktury dokumentu używanego formatu. Przykładowo dokumenty *JSON* muszą być zgodne ze standardem *JSON*, jednak logiczna struktura danych jest elastyczna. Wpisy z różnymi polami mogą być przechowywane w obrębie tej samej kolekcji. Sposób przechowywanie danych w bazach dokumentowych ilustruje rysunek 2.9.

People	
{ name: "Alice", age: 20, knows: "Bob" }	{ name: "Bob", age: 25, likes: ["books", "pets"] }

**Rys. 2.9.** Struktura danych w bazach dokumentowych w formacie JSON

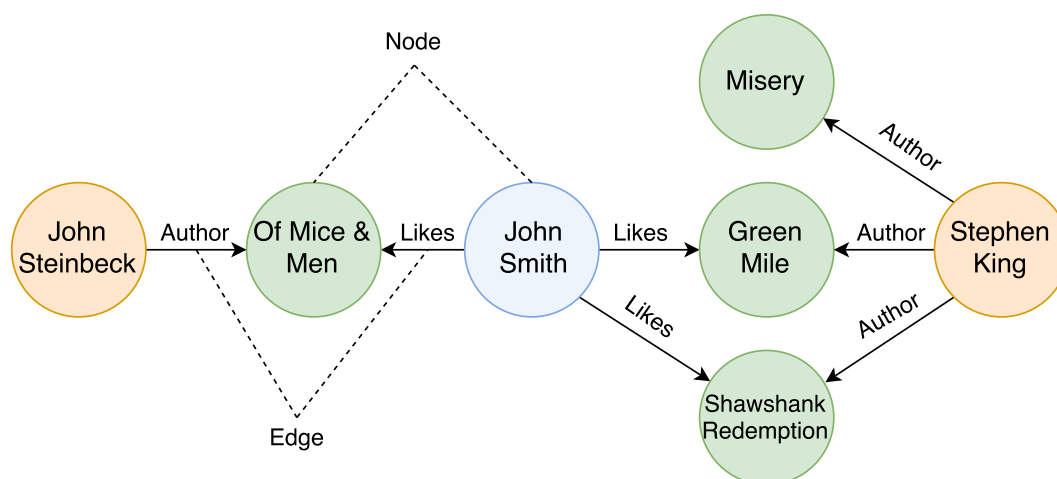
Zaletą dokumentowych baz danych jest m.in. możliwość tworzenia zapytań, które są w stanie pobierać jedynie wyznaczoną część dokumentu lub też wyszukiwać dane po polach zdefiniowanych wewnątrz struktury (nie tylko po kluczu głównym). Jest to możliwe dzięki temu, że format, w którym przechowywane są dane jest rozumiany przez system, gdyż jest on pewnego rodzaju standardem.

Najbardziej popularne bazy dokumentowe to *MongoDB*, *CouchDB* oraz *MarkLogic*.

### 2.5.4. Grafowe

Ostatnim z typów baz danych NoSQL są bazy grafowe. Znacznie różnią się one od tych, które były omawiane do tej pory, gdyż nie przechowują danych w formie agregatów. Wykorzystują one grafy, czyli strukturę danych, która jest zbiorem wierzchołków (ang. *nodes*) połączonych krawędziami (ang. *edges*)[13]. Na wierzchołkach przechowywane są dane, natomiast krawędzie reprezentują zależności pomiędzy nimi. Sposób przechowywania danych w bazach grafowych ilustruje rysunek 2.10.

Poprzednio omówione typy baz NoSQL powstały z myślą o pracy w klastrach, jednak grafowe bazy danych zostały stworzone, aby rozwiązać inny problem re-



**Rys. 2.10.** Struktura danych w grafowych bazach NoSQL

lacyjnych baz danych, a mianowicie przechowywanie danych o bardzo złożonych powiązaniach.

Umieszczenie informacji na grafie umożliwia tworzenie skomplikowanych zależności pomiędzy nimi, które mogą być równie ważne jak same dane [4]. Mogą to być na przykład preferencje użytkownika serwisu z książkami, relacje pomiędzy użytkownikami na portalu społecznościowym lub dane geograficzne.

Dane uporządkowane w ten sposób umożliwiają tworzenie zapytań, które byłyby bardzo trudne lub wręcz niemożliwe do zrealizowania w tradycyjnych bazach SQL, a nawet w innych bazach typu NoSQL. Do pracy z danymi na grafach wykorzystywane są algorytmy znane z teorii grafów, dlatego umożliwiają one rozwiązywanie problemów z wielu dziedzin. Mogą one znaleźć zastosowanie np. w systemach GPS do wyznaczania najkrótszej trasy lub w różnych systemach rekomendacji.

Przykładami tego typu baz danych są m.in. *Neo4J*, *FlockDB* oraz *Infinite Graph*.

## 2.6. Wady

Można zauważyć, że bazy danych NoSQL prezentują szerokie spektrum możliwości oraz są odpowiedzią na niedoskonałości relacyjnych baz danych. Nie oznacza to jednak, że są one pod każdym względem idealne, gdyż posiadają również kilka istotnych wad.

### 2.6.1. Brak transakcji

Brak transakcji, to największa niedogodność baz NoSQL w porównaniu do relacyjnych systemów zarządzania danymi. Wiele aplikacji wymaga zapewnienia stu-procentowej gwarancji wykonania pewnych operacji na danych, a kluczowym mechanizmem, który to umożliwia są właśnie transakcje. W systemach z taką specyfiką często lepiej jest użyć bazy relacyjnej.

Bazy NoSQL również mogą być użyte do takich celów, jednak z powodu braku transakcji odpowiedzialność za zapewnienie poprawności operacji na danych musi być zapewniona po stronie aplikacji, co może okazać się skomplikowane w imple-

mentacji.

### 2.6.2. Niedojrzałość

Stosunkowo niewielki wiek baz NoSQL w porównaniu do relacyjnych systemów skutkuje tym, że wiele implementacji baz NoSQL jest jeszcze w fazie rozwojowej. W wyniku tego może się zdarzyć, że implementacja bazy danych może posiadać jakieś błędy, niektóre funkcjonalności mogą być dopiero w trakcie tworzenia lub dokumentacja może być zbyt uboga.

Przez to, że relacyjne bazy danych znane są od wielu lat istnieje wielu ekspertów w tej dziedzinie, co przekłada się na znacznie większą szansę na znalezienie programistów, którzy będą znali technologie potrzebne do realizacji danego projektu.

### 2.6.3. Ograniczone funkcjonalności

W wyniku uproszczenia pewnych mechanizmów działania oraz braku wsparcia dla języka SQL, w większości baz NoSQL nie jest możliwe tworzenie tak złożonych zapytań, jak w bazach relacyjnych. Są one o wiele prostsze w użyciu w aplikacjach typu CRUD (ang. *Create, Read, Update, Delete*). Bardziej skomplikowane operacje na danych w wielu typach baz NoSQL wymaga wprowadzenia dodatkowej logiki w aplikacji.

### 2.6.4. Brak wsparcia technicznego

Konsekwencją otwartego kodu źródłowego większości baz NoSQL jest brak wsparcia technicznego. Wiele firm często jest w stanie zapłacić, aby mieć zapewnione wsparcie techniczne ekspertów w przypadku awarii systemu, jednak przeważnie nie jest to możliwe w przypadku większości baz danych typu NoSQL. Często są one tworzone przez małe firmy, które nie są w stanie zapewnić 24-godzinnego wsparcia technicznego tak jak firmy produkujące relacyjne bazy danych, takie jak np. *Oracle*, *IBM* czy *Microsoft*.

## 2.7. Podsumowanie

Z pewnością można stwierdzić, że bazy danych typu NoSQL nie są odpowiedzią na każdy problem, z którym spotkać się mogą systemy, których zadaniem jest przechowywanie i udostępnianie danych. Tym bardziej nie jest prawdą, że bazy NoSQL są pod każdym względem lepsze od tradycyjnych baz relacyjnych. Są one jedynie pewnego rodzaju alternatywą oraz dodatkowym narzędziem, które powinno być używane w określonych sytuacjach.

Bazy relacyjne oraz bazy NoSQL uzupełniają się nawzajem. Bardzo ważne jest dobranie odpowiedniego typu bazy do charakterystyki danego projektu. Takie podejście do tworzenia aplikacji nosi nazwę *Polyglot Persistence*.

Czasem możliwe jest również połączenie zalet obu typów baz danych i użycie ich razem w obrębie jednej aplikacji. Architektura, która umożliwia tworzenie aplikacji w ten sposób nosi nazwę CQRS i zostanie zaprezentowana w rozdziale 3.

# Rozdział 3

## Architektura aplikacji wykorzystująca oba typy baz danych

W tym rozdziale zostanie przedstawiona przykładowa aplikacja, która przewiduje wykorzystanie dwóch typów baz danych. Na jej podstawie będzie można zilustrować:

- implementację aplikacji w oparciu o architekturę CQRS,
- prosty sposób uruchamiania całego systemu - aplikacji oraz wielu instancji baz danych,
- zarządzanie bazami danych oraz ich skalowanie,
- przeprowadzanie testów wydajnościowych aplikacji.

### 3.1. Architektura CQRS

#### 3.1.1. CQS jako zaczątek CQRS

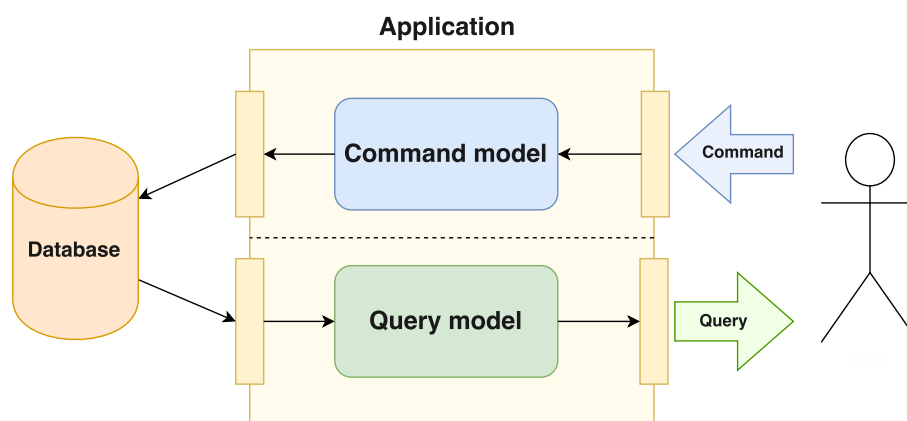
Pojęcia CQS (ang. *Command Query Separation*) zostało przedstawione przez Bertranda Meyera już w 1986 roku [6]. Jest to zasada mówiąca o tym, że w systemie powinny istnieć jedynie metody, które można zaklasyfikować do jednej z dwóch grup. Są to:

- **Komendy** (ang. *Command*) - metody zmieniające stan aplikacji, które nie zwracają żadnej wartości,
- **Kwerendy** (ang. *Query*) - metody zwracające pewną wartość, jednak nie zmieniające stanu aplikacji. Są one idempotentne (ang. *Idempotence*), czyli niezależnie od tego ile razy zostaną wykonane, zawsze zwrócą ten sam wynik. Ideę tego typu metod dobrze obrazuje zdanie: “Pytanie nie powinno zmieniać odpowiedzi”.

### 3.1.2. Klasyczny CQRS

Po blisko 20 latach od sformułowania terminu CQS, Greg Young oraz Udi Dahan przedstawili koncepcję CQRS (ang. *Command Query Responsibility Segregation*), która wywodziła się bezpośrednio z CQS [6].

Założenia są te same, jednak zmianie ulega skala problemu. W przypadku CQS mowa jest o podziale poszczególnych metod, natomiast CQRS zakłada taki podział na poziomie klas, a nawet całej aplikacji. Oznacza to, że aplikacja będzie dzielić się na dwa osobne serwisy - pierwszy odpowiedzialny za modyfikację stanu aplikacji i jego zapis (zwany *Command*) oraz drugi odpowiedzialny jedynie za odczyt jej stanu (o nazwie *Query*).



Rys. 3.1. Architektura CQRS z pojedynczą bazą danych

Aplikacja stworzona według architektury przedstawionej na rysunku 3.1 używa jednej bazy danych. Zastosowanie tej architektury pomaga w uporządkowaniu aplikacji, co przyczyni się do zmniejszenia ilości potencjalnych błędów oraz umożliwia stworzenie osobnego modelu dla obu warstw, co pozwoli w bardziej naturalny sposób odzwierciedlić logikę biznesową aplikacji. Można tu użyć zarówno bazy relacyjnej jak i NoSQL, jednak żadna z nich nie będzie w pełni przystosowana do przetwarzania zarówno operacji po stronie *Query* jak i *Command*.

Rozwiązaniem jest użycie obu typów baz danych. Podejście to nosi angielską nazwę *Polyglot persistence*.

### 3.1.3. Użycie w CQRS różnych typów baz danych

Greg Young przeanalizował obie te strony pod kątem takich kryteriów jak spójność danych (ang. *Consistency*), sposób przechowywania danych (ang. *Data storage*) oraz skalowalność (ang. *Scalability*)[5].

Wyniki tej analizy wyglądają następująco:

#### 3.1.3.1. Spójność danych

**Command:** Przetwarzanie transakcji, czyli wprowadzanie zmian do systemu jest o wiele prostsze i bardziej wiarygodne, jeżeli obsługiwane dane są zawsze spójne. W tej części aplikacji niespójne dane wymagają przewidzenia i obsłużenia wielu przypadków wynikających z braku pewności czy dane są zawsze aktualne.

**Query:** Przy odczycie danych mogą być one warunkowo spójne (*Eventual consistency* 2.4.4), gdyż pobranie nieaktualnych danych, które i tak za chwilę będą aktualne w większości przypadków nie stanowi większego problemu.

### 3.1.3.2. Sposób przechowywania danych

**Command:** Ta część aplikacji powinna zapisywać dane w ściśle określonej formie, która odpowiada sztywno zdefiniowanemu schematowi np. tabele. Jest to wymagane ze względu na to, aby zapisywane dane miały poprawną strukturę oraz nie ulegały duplikacji.

**Query:** Odczyt danych jest bardziej wydajny w sytuacji, gdy dane są przechowywane w postaci mniej uporządkowanych struktur, które nie wymagają dodatkowych operacji takich jak np. łączenie danych (ang. *join*). Dopuszcza się duplikację danych na rzecz znacznie zwiększonej wydajności.

### 3.1.3.3. Skalowalność

**Command:** W większości systemów (szczególnie webowych) strona ta przetwarza stosunkowo niewielką ilość operacji w porównaniu do operacji odczytu. W związku z tym skalowanie w tej części aplikacji przeważnie nie jest potrzebne.

**Query:** Większość systemów webowych przetwarza o wiele więcej operacji odczytu, niż zapisu. Często są to tak duże ilości, że skalowanie po tej stronie jest niezbędne.

### 3.1.3.4. Wnioski

Nie jest możliwe stworzenie jednego modelu, który będzie w stanie optymalnie przetwarzać tak różne operacje jak wyszukiwanie, odczyt oraz przetwarzanie danych i ich zapis.

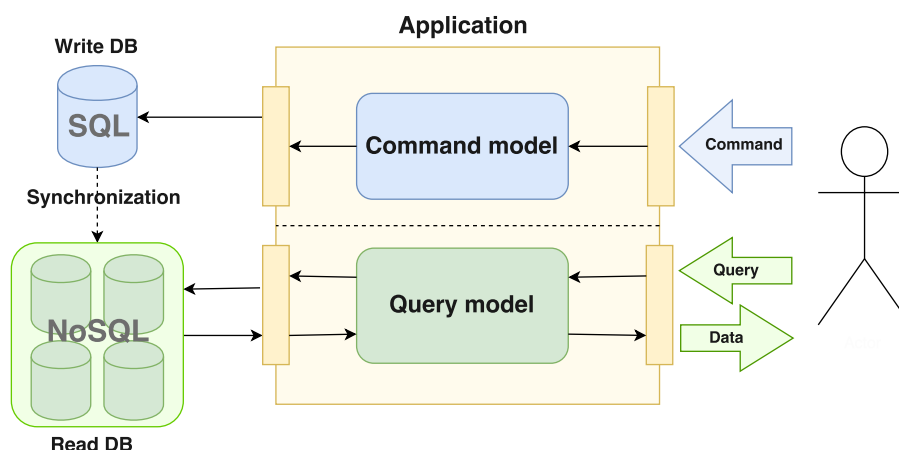
Z powyższej analizy można wywnioskować, że do części *Command* służącej do zapisu najlepiej nadaje się relacyjna baza danych, gdyż pozwala na przetwarzanie danych w transakcjach, przechowuje je w uporządkowanej strukturze oraz nie wymaga skalowalności.

Natomiast po stronie *Query* opłacalne jest użycie baz typu NoSQL, gdyż warunkowa spójność nie jest problemem, przechowywanie danych w nieuporządkowanej strukturze jest tutaj atutem oraz umożliwia łatwą skalowalność.

W architekturze zaprezentowanej na rysunku 3.2 dane są zapisywane do relacyjnej bazy danych, natomiast odczytywane z bazy NoSQL. Po każdej zmianie dane są synchronizowane pomiędzy obiema bazami danych.

Zaletą tego podejścia jest przede wszystkim możliwość niezależnego skalowania operacji odczytu i zapisu. W klasycznym podejściu do CQRS, gdzie używana jest jedna baza danych skalowaniu mogą być poddane jedynie obie strony jednocześnie. Tutaj mamy możliwość zwiększania wydajności jedynie w miejscach, gdzie rzeczywiście występuje z nią problem.

W kolejnych rozdziałach zostanie opisana przykładowa aplikacja wykorzystująca oba typy baz danych stworzona w oparciu o architekturę CQRS. Zostanie przedstawione, w jaki sposób przebiega implementacja tego typu aplikacji, jak w prosty



Rys. 3.2. Architektura CQRS z dwoma typami baz danych

sposób skalować bazy danych w celu zwiększenia wydajności oraz zostaną przeprowadzone testy wydajnościowe pokazujące zyski w porównaniu z aplikacją o klasycznej architekturze.

## 3.2. Wykorzystywane technologie

Do stworzenia przykładowej aplikacji zostaną wykorzystane następujące technologie:

- **Ubuntu 16.04** - system operacyjny (dystrybucja Linux),
- **Java 8** - obiektowy język programowania,
- **Spring Framework** oraz **Spring Boot** - biblioteki umożliwiające tworzenie aplikacji webowych,
- **Gradle** - narzędzie służące do budowania projektu oraz zarządzania zależnościami,
- **PostgreSQL** - relacyjna baza danych,
- **MongoDB** - dokumentowa baza danych typu NoSQL,
- **Docker** oraz **Docker Compose** - oprogramowanie umożliwiające tworzenie wirtualnych kontenerów oraz zarządzanie nimi,
- **Apache JMeter** - oprogramowanie do testów wydajnościowych aplikacji,
- **Git** - system kontroli wersji,
- **IntelliJ IDEA** - środowisko programistyczne.

### 3.2.1. Ubuntu

Ubuntu jest dystrybucją systemu Linux, która jest bardzo przyjazna dla użytkownika (ang. *user-friendly*). W celu uruchomienia nie wymaga skomplikowanej konfiguracji. Można ją w prosty sposób zainstalować i od razu używać.

Powodem wyboru systemu z rodziny Linux był fakt, że umożliwia on znacznie wygodniejszy dostęp do wielu narzędzi konsolowych, które często są wygodniejsze w użyciu od tych z graficznym interfejsem użytkownika. Dzięki temu możliwa jest większa automatyzacja wykonywanych działań np. poprzez tworzenie skryptów. Dodatkowo Linux jest natywnym środowiskiem uruchomieniowym dla oprogramowania *Docker*, które będzie pełniło znaczącą rolę przy uruchamianiu całego systemu. Jest on również darmowy.

### 3.2.2. Java 8

Java to język programowania obiektowego o składni wywodzącej się z C++. Jest to aktualnie najbardziej popularny język programowania [14] o szerokich zastosowaniach. W Javie można tworzyć m.in. aplikacje webowe, desktopowe lub też na platformę Android.

Java w wersji 8 wspiera elementy programowania funkcyjnego takie jak wyrażenia lambda. Niedawno (we wrześniu 2017 roku) została wydana Java 9, jednak nowe funkcjonalności, które oferuje ta wersja nie mają zastosowania w tworzonej aplikacji, dlatego użyta została Java 8.

### 3.2.3. Spring

Spring Framework to następca Javy Enterprise Edition (JEE). Aktualnie Spring jest standardem w dziedzinie tworzenia nowoczesnych aplikacji webowych w języku Java. W przeciwieństwie do JEE nie wymaga skomplikowanych konfiguracji w celu uruchomienia najprostszej aplikacji, a mimo to umożliwia zaawansowaną konfigurację projektu.

Dodatkiem do Springa jest Spring Boot, który znacznie upraszcza konfigurację startową i posiada wbudowane serwery aplikacji (Tomcat, Jetty lub Undertow). Dzięki temu możliwe jest tworzenie aplikacji, które będą zawarte w pojedynczym pliku JAR (ang. *Java ARchive*). Nie ma więc potrzeby (jak w przypadku klasycznego Springa czy JEE) ręcznego uruchamiania plików WAR (ang. *Web-application ARchive*) na serwerze aplikacyjnym.

### 3.2.4. Gradle

Gradle to narzędzie służące do automatyzacji procesu budowania projektu oraz zarządzania zależnościami. Jest to alternatywa dla m.in. Maven lub Ant.

Gradle do konfiguracji używa specjalnego dedykowanego języka (tzw. DSL - *Domain-Specific Language*) bazującego na języku Groovy, który jest o wiele bardziej czytelny, niż używany w innych narzędziach tego typu XML (ang. *Extensible Markup Language*).



Dodatkowo Gradle posiada narzędzie o nazwie Gradle Wrapper. Wrapper jest plikiem JAR i kilkoma skryptami, które są generowane i dołączane do projektu. Pozwala to na budowanie projektu bez potrzeby instalacji Gradle na komputerze. Rozwiązuje to również problem niezgodności wersji, gdyż każdy programista po pobraniu projektu od razu będzie miał tę samą wersję Gradle.

### 3.2.5. PostgreSQL

PostgreSQL to relacyjna baza danych, która zostanie użyta w aplikacji jako baza po stronie zapisu w modelu CQRS. Postgres został wybrany ze względu na to, że jest to jedna z bardziej zaawansowanych i profesjonalnych relacyjnych baz danych cechująca się dobrą wydajnością, a przy tym jest darmowa i wieloplatformowa.

Oczywiście nie jest to jedyny możliwy wybór. Można by było użyć praktycznie każdej innej relacyjnej bazy danych posiadającej mechanizmy zapewniania spójności danych takie jak m.in. transakcje. Prosta baza danych, która jest alternatywą dla PostgreSQL to np. MySQL.

### 3.2.6. MongoDB

MongoDB to dokumentowa baza danych typu NoSQL, która zostanie użyta w aplikacji jako baza po stronie odczytu w modelu CQRS. Mongo przechowuje dane w postaci dokumentów JSON, który w pewnym stopniu narzuca strukturę danych zgodną ze standardem JSON. Bazy dokumentowe zostały bardziej szczegółowo opisane w rozdziale 2.5.3.

Głównym powodem wyboru tej bazy NoSQL był mechanizm skalowania w Mongo o nazwie *sharding*, który umożliwia stosunkowo prostą konfigurację klastra serwerów. Zostanie on bardziej szczegółowo omówiony w rozdziale opisującym sposób skalowania 3.15.1.

Dodatkowo Mongo posiada rozbudowaną i utrzymywaną dokumentację techniczną wraz z wieloma poradnikami na oficjalnej stronie.

### 3.2.7. Docker

Docker jest oprogramowaniem umożliwiającym tworzenie wirtualnych izolowanych kontenerów z aplikacjami. Jest on w stanie zastąpić klasyczną wirtualizację.

Docker zamiast wirtualizacji używa tzw. konteneryzacji. Przewagą tego mechanizmu jest to, że przy uruchamianiu poszczególnych kontenerów nie ma potrzeby emulowania całej warstwy sprzętowej oraz systemu operacyjnego. Przynosi to znaczące korzyści w sytuacji, gdy istnieje potrzeba uruchomienia dużej ilości aplikacji na oddzielnych środowiskach.

Kontenery Dockera są od siebie niezależne. Posiadają odrębne obszary pamięci, własny interfejs sieciowy i przypisany do niego prywatny adres IP. Mają również przydzielone inne przestrzenie dyskowe, na których zainstalowany jest prosty obraz systemu operacyjnego (przeważnie Ubuntu, Debian lub CentOS) oraz pożądana aplikacja i wszystkie wymagane zależności. Współdzielą jedynie jądro systemu macierzystego [15].

Na takim środowisku uruchamiany jest izolowany proces z aplikacją, który jest niewidoczny z poziomu innych kontenerów. Oczywiście kontenery można skonfigurować w taki sposób, aby mogły się ze sobą komunikować np. poprzez udostępnianie poszczególnych portów lub łączenie ze sobą kontenerów (ang. *link*).

Każdy kontener może być uruchomiony przy pomocy obrazu Docker. Obrazy można tworzyć samemu za pomocą tzw. plików *Dockerfile* np. w sytuacji, gdy istnieje potrzeba uruchomienia autorskiej aplikacji w kontenerze. Jednak w przypadku większości popularnych aplikacji czy też baz danych dostępne są gotowe obrazy w darmowym rejestrze Docker. Wystarczy więc wywołać jedynie jedną komendę, aby Docker pobrał obraz, zainstalował go w nowym kontenerze, a następnie uruchomił wskazaną aplikację.

Oprogramowaniem, który wspomaga pracę z Dockerem jest Docker Compose. Narzędzie to umożliwia tworzenie plików konfiguracyjnych dla aplikacji wymagających uruchomienie wielu kontenerów Docker. Wszystkie serwisy, które muszą być uruchomione definiowane są w pliku w formacie YAML o nazwie *docker-compose.yml*. Następnie za pomocą jednego polecenia *docker-compose up* można uruchomić wszystkie zdefiniowane wcześniej kontenery ze wskazaną konfiguracją.

Tym sposobem można za pomocą jednej komendy uruchomić całe skomplikowane środowisko np. aplikację, kilka instancji baz danych, serwer SMTP itp. Zaletą tego podejścia jest to, że każdy programista do uruchomienia takiego środowiska potrzebuje mieć jedynie zainstalowane oprogramowanie Docker oraz Docker Compose, co znacznie ułatwia i przyspiesza pracę.

### 3.2.8. JMeter

Apache JMeter to darmowe narzędzie służące do wykonywania testów wydajnościowych różnych typów aplikacji, w tym szczególnie aplikacji webowych.

JMeter umożliwia symulowanie dużego obciążenia serwera odpowiadającego sytuacji, w której aplikacja wykorzystywana jest przez wielu użytkowników. Dodatkowo udostępnia również narzędzia do monitorowania wydajności aplikacji np. czasów odpowiedzi na zapytania. Pozwala również tworzyć raporty z przeprowadzonych testów w formacie zarówno tekstowym jak i graficznym np. tabele i wykresy.

JMeter można rozbudowywać o nowe funkcjonalności za pomocą wielu dostępnych pluginów.

### 3.2.9. Git

Git to tzw. system kontroli wersji (ang. *version control system*). Jest to oprogramowanie konsolowe, które umożliwia wersjonowanie kodu aplikacji poprzez przechowywanie historii wszystkich wprowadzonych zmian.

Każda zmiana w kodzie zawierająca się w logiczną całość np. dodanie nowej funkcjonalności lub poprawa błędu, powinna być zakończona stworzeniem tzw. commit'a. Każdy commit posiada przypisany hash wygenerowany algorytmem SHA-1 (ang. *Secure Hash Algorithm*), który jednoznacznie określa stan kodu w danym momencie, czyli jest to wersja kodu aplikacji.

Dzięki przechowywanej historii commit'ów Git umożliwia przywrócenie kodu

aplikacji do stanu z dowolnej utworzonej wersji. Jest to użyteczne np. w momencie, gdy w nowej wersji aplikacji występuje jakiś błąd i potrzebny jest szybki powrót do starszej, poprawnie działającej wersji.

Dodatkowo Git umożliwia zapis kodu w zdalnym repozytorium w chmurze, czyli jednocześnie automatycznie tworzona jest kopia zapasowa kodu aplikacji. Repozytorium może być używane przez wielu programistów, co wspomaga współdzielenie najnowszej wersji aplikacji w zespole deweloperskim.

Git wspiera również tworzenie tzw. gałęzi (ang. *branches*), które umożliwiają tworzenie nowych funkcjonalności na niezależnych kopiach kodu. Następnie możliwe jest scalenie zmian w kodzie z główną gałęzią o nazwie *master*. Tworzenie gałęzi pozwala kontrolować, które funkcjonalności powinny się znaleźć w głównej wersji aplikacji.

### 3.2.10. IntelliJ IDEA

IntelliJ IDEA to nowoczesne środowisko programistyczne stworzone przez firmę JetBrains służące do programowania głównie w języku Java.

IntelliJ oferuje inteligentne mechanizmy podpowiadania kodu, jego statyczną analizę pod względem poprawności, wygodne mechanizmy poruszania się po projekcie oraz śledzenia wykonywania kodu (tzw. *debugger*). Dodatkowo dostępnych jest wiele darmowych wtyczek (ang. *plugins*) umożliwiających rozbudowę środowiska o dodatkowe funkcjonalności np. przeglądanie zawartości bazy danych lub wsparcie dla Git.

Jest to aktualnie jedno z najbardziej popularnych środowisk programistycznych wykorzystywanych do profesjonalnych zastosowań [16]. IntelliJ jest dostępny za darmo w wersji Community, która posiada wszystkie najważniejsze funkcjonalności potrzebne do codziennej pracy programisty.

## 3.3. Koncepcja aplikacji

Celem pracy jest stworzenie serwisu webowego, na podstawie którego będzie możliwe zaprezentowanie architektury CQRS. Serwis ten może posiadać prosty model danych oraz niewielką liczbę funkcjonalności. Dobrym przykładem typowej aplikacji webowej jest sklep internetowy, w którym istnieją produkty i użytkownicy, którzy mogą przeglądać katalog produktów oraz składać zamówienia. Serwis odzwierciedlający uproszczony sklep internetowy będzie więc zaimplementowany i opisany w tej części pracy.

Nie musi on wykonywać skomplikowanej logiki biznesowej oraz posiadać graficznego interfejsu użytkownika, gdyż jest potrzebny jedynie w celu pokazania sposobu wykorzystania obu typów baz danych oraz metod zwiększania wydajności poprzez skalowanie.

Aplikacja zostanie zaimplementowana jako serwis REST (ang. *REpresentational State Transfer*), który będzie posiadał określone API (ang. *Application Programming Interface*) umożliwiające komunikację z aplikacją za pomocą protokołu HTTP (ang. *HyperText Transfer Protocol*). REST zostanie opisany w rozdziale 3.5.

Najważniejszym elementem systemu będą używane bazy danych oraz ich konfiguracja. Aplikacja będzie umożliwiała zmianę źródła odczytu danych za pomocą tzw. przełącznika funkcjonalności (ang. *feature switch*). Domyślnie będzie ona zachowywać się jak aplikacja wykonana w klasycznym stylu, czyli zapis oraz odczyt danych będzie kierowany do bazy relacyjnej.

Zastosowana architektura CQRS umożliwi w bardzo prosty sposób przełączenie aplikacji w tryb odczytu z bazy danych NoSQL - wystarczy zmiana wartości jednej właściwości przy uruchomieniu aplikacji. Sposób realizacji zostanie opisany w rozdziale 3.10.1.

W sytuacji, gdy aplikacja będzie wykorzystywać bazę NoSQL jako źródło odczytu danych bazy muszą się ze sobą synchronizować po każdej zmianie w bazie relacyjnej, aby w bazie NoSQL dane były aktualne. Synchronizacja zostanie opisana w rozdziale 3.12.

Samo użycie bazy NoSQL do odczytu danych powinno zwiększyć wydajność aplikacji, jednak można pójść o krok dalej. Bazy NoSQL są przystosowane do pracy w klastrach, dlatego można je w prosty sposób skalować, co pozwala na osiągnięcie jeszcze większej wydajności. Skalowanie baz danych oraz mechanizm skalowania bazy MongoDB o nazwie *sharding* zostaną opisane w rozdziale 3.15.

Konfiguracja baz danych oraz uruchamianie całego środowiska będzie odbywało się automatycznie za pomocą utworzonych skryptów powłoki Bash oraz narzędzi Docker i Docker Compose. Zostanie to przedstawione w rozdziale 3.14.

Tak stworzona aplikacja umożliwi porównanie wydajności zastosowanego podejścia poprzez przeprowadzenie testów wydajnościowych. Sposób przeprowadzenia testów oraz uzyskane wyniki zostaną omówione w rozdziale 3.16.

## 3.4. Konfiguracja projektu - Gradle

Podstawowy projekt aplikacji bazującej na Spring można w bardzo prosty sposób wygenerować w momencie tworzenia projektu w środowisku IntelliJ za pomocą narzędzia *Spring Initializr*. Jako narzędzie do zarządzania zależnościami można wybrać Maven lub Gradle. Do wykonania tego projektu został użyty Gradle.

Zapewnia on podstawowe zadania (ang. *task*), które służą m.in. do kompilacji, uruchamiania testów, tworzenia pliku JAR z aplikacją oraz jej uruchamiania. Można je wywołać używając wygenerowanego w projekcie Gradle Wrappera za pomocą komendy w konsoli: `./gradlew <listaZadań>`.

W celu zbudowania projektu (kompilacja, uruchomienie testów i stworzenie JAR) należy wykonać `./gradlew build`. Do uruchomienia aplikacji SpringBoot służy natomiast komenda `./gradlew bootRun`.

Gradle zarządza również zależnościami oraz konfiguracją za pomocą pliku *build.gradle*. Dzięki temu żadna z bibliotek nie musi być na stałe dołączona do projektu - są one dynamicznie pobierane ze wskazanego repozytorium. Domyślnym repozytorium jest *mavenCentral*. Jest to największe repozytorium zawierające praktycznie wszystkie dostępne w Internecie biblioteki.

Do tego projektu potrzebne będą następujące biblioteki:

- **spring-boot-starter-web** - tworzenia aplikacji webowych w SpringBoot,

- **spring-boot-starter-data-jpa** - komunikacja z relacyjną bazą danych,
- **spring-boot-starter-data-mongodb** - komunikacja z bazą MongoDB,
- **gradle-docker** - tworzenie obrazu Docker z aplikacją,
- **springfox-swagger** - automatyczne tworzenie dokumentacji API,
- **liquibase** - narzędzie do zarządzania schematem relacyjnej bazy danych,
- **postgresql** - sterowniki do komunikacji z bazą PostgreSQL,
- **lombok** - automatyczna generacja kodu za pomocą adnotacji Spring,
- **spring-boot-starter-test** - tworzenie testów Spring,
- **assertj** - asercje wykorzystywane w testach jednostkowych,
- **mockito** - tworzenie tzw. *mock*'ów w testach jednostkowych.

Zarządzanie zależnościami odbywa się poprzez sekcję *dependencies* w pliku *build.gradle*:

**Listing 3.1.** build.gradle

```
1  buildscript {
2      ext {
3          springBootVersion = '1.5.7.RELEASE'
4      }
5      repositories {
6          mavenCentral()
7      }
8      dependencies {
9          classpath("org.springframework.boot:spring-boot-gradle-plugin:${springBootVersion}")
10         classpath("se.transmode.gradle:gradle-docker:1.2")
11         classpath("org.liquibase:liquibase-core:3.4.1")
12         classpath("org.liquibase:liquibase-gradle-plugin:1.1.1")
13         classpath("org.postgresql:postgresql:42.1.4")
14     }
15 }
16
17 dependencies {
18     // SPRING
19     compile('org.springframework.boot:spring-boot-starter-web:1.5.8.RELEASE')
20     compile('org.springframework.boot:spring-boot-starter-data-jpa')
21     compile('org.springframework.boot:spring-boot-starter-data-mongodb')
22
23     // SWAGGER
24     compile('io.springfox:springfox-swagger2:2.7.0')
25     compile('io.springfox:springfox-swagger-ui:2.7.0')
26
27     // POSTGRES
28     compile('org.liquibase:liquibase-core')
29     runtime('org.postgresql:postgresql:42.1.4')
30
31     compileOnly('org.projectlombok:lombok:1.16.18')
32
33     // TEST
34     testCompile('org.springframework.boot:spring-boot-starter-test')
35     testCompile('org.assertj:assertj-core:3.8.0')
36     testCompile('org.mockito:mockito-core:2.12.0')
37 }
```

W sekcji *dependencies* znajdują się biblioteki wraz z ich wersjami, które zostaną automatycznie pobrane w momencie pierwszego uruchomienia projektu. Dzięki temu nigdy nie nastąpią problemy z kompilacją kodu aplikacji, gdyż zawsze będą pobrane odpowiednie wersje bibliotek.

## 3.5. REST

Aplikacja została wykonana jako serwis REST (ang. *REpresentational State Transfer*). REST jest swego rodzaju wzorcem, który narzuca dobre praktyki odnośnie tworzenia aplikacji rozproszonych. Bazuje on na bezstanowej komunikacji klient-serwer za pomocą protokołu HTTP (ang. *HyperText Transfer Protocol*). Jest to alternatywa dla bardziej złożonych, rzadziej używanych mechanizmów takich jak CORBA (ang. *Common Object Request Broker Architecture*), RPC (ang. *Remote Procedure Call*) czy też SOAP (ang. *Simple Object Access Protocol*) [17].

Podjęcie tego typu stosuje się w nowoczesnych systemach mających architekturę tzw. mikroservisów lub mikrousług (ang. *microservices*). Dzięki temu, że wykorzystywany jest protokół HTTP aplikacje, które się ze sobą komunikują mogą być stworzone przy użyciu różnych technologii oraz języków programowania.

Przykładowo zaimplementowana aplikacja sklepu internetowego nie posiada interfejsu użytkownika, jednak udostępnia ona możliwość korzystania z niej za pomocą API (ang. *Application Programming Interface*). Daje to możliwość napisania widoku w zupełnie innej technologii np. HTML/CSS oraz PHP lub JavaScript itp. Aplikacja z graficznym interfejsem użytkownika mogła by korzystać z serwisu sklepu internetowego wysyłając do niego żądania HTTP i prezentować otrzymane dane użytkownikowi w formie strony internetowej. Dodatkowo może wtedy istnieć wiele widoków np. dodatkowy w postaci aplikacji mobilnej korzystający z tego samego serwisu sklepu.

### 3.5.1. Protokół HTTP

#### 3.5.1.1. Metody

Klient wysyłający zapytania HTTP do serwera najczęściej korzysta z dwóch metod GET oraz POST. GET jest najbardziej popularną metodą HTTP, gdyż używana jest przeważnie do pobierania zasobów. Przykładowo, gdy wpiszemy w przeglądarce adres strony internetowej, to tak na prawdę wysyłamy do serwera żądanie HTTP z metodą GET o pobranie zasobów - w tym przypadku strony internetowej. Drugą najbardziej popularną metodą HTTP jest POST, którego używa się przeważnie do zapisu danych.

Teoretycznie można by używać jedynie tych dwóch metod do każdej operacji na danych np. również do usuwania lub edycji, jednak byłoby to bardzo nieczytelne. REST zakłada wykorzystanie innych metod HTTP.

Do tych najczęściej używanych należą:

- **GET** - pobieranie stanu zasobu,
- **POST** - tworzenie zasobu,
- **DELETE** - usunięcie zasobu,
- **PUT** - aktualizacja całego zasobu (zastąpienie go innym),
- **PATCH** - aktualizacja jakiejś części istniejącego zasobu.

Istnieją również inne metody HTTP takie jak HEAD, OPTIONS, CONNECT i TRACE, jednak są one o wiele rzadziej używane.

### 3.5.1.2. Kody statusów

Każda odpowiedź na zapytanie HTTP oprócz danych zwraca również kod statusu. Na podstawie wartości kodów możliwe jest określenie typu zwróconej odpowiedzi.

Podstawowy podział kodów HTTP, to:

- **1xx** - kody informacyjne,
  - **2xx** - kody powodzenia,
  - **3xx** - kody przekierowania,
  - **4xx** - kody błędu klienta,
  - **5xx** - kody błędu serwera,
- gdzie  $x$  oznacza określoną cyfrę.

Najczęściej używane kody statusów to:

- **200 (OK)** - operacja udana,
- **201 (Created)** - zasób został utworzony,
- **400 (Bad Request)** - niepoprawne zapytanie,
- **404 (Not Found)** - zasób nie został znaleziony,
- **500 (Internal Server Error)** - błąd wewnętrzny serwera,
- **503 (Service Unavailable)** - serwis niedostępny.

### 3.5.2. Dokumentacja API

API aplikacji, czyli udostępnione dla klienta możliwe do wykonania operacje na aplikacji często wymaga dokumentacji. Czasami nie do końca wiadomo, co dana metoda robi, jakie parametry przyjmuje, czy jakie dane zwraca.

Z pomocą przychodzi tutaj narzędzie o nazwie Swagger, które automatycznie generuje dokumentację API aplikacji Spring na podstawie kodu. Dodatkowo generowany jest również widok w postaci strony HTML, który umożliwia wysyłanie zapytań. Dzięki temu użytkownik chcący korzystać z aplikacji, nie ma potrzeby ręcznego tworzenia zapytań HTTP np. przez narzędzie *Postman*. API aplikacji oraz jego dokumentacja zostaną przedstawione w rozdziale 3.7.

## 3.6. Model danych

Model danych nie jest zbyt rozbudowany, gdyż ma on służyć jedynie jako przykład. Odzwierciedlenie takiego modelu w bazie danych będzie wymagało jednak stworzenia kilku relacji.

Jako przykład wykorzystany został relacyjny model danych, gdyż to właśnie relacje często powodują spadki wydajności przy odczycie danych, gdyż niezbędne jest łączenie danych za pomocą tzw. *JOIN*ów.

Na model danych aplikacji składa się kilka elementów takich jak produkt, użytkownik, zamówienie oraz pozycje w zamówieniu (rys. 3.3). Dzięki dodanym adnotacjom Spring i bibliotece Spring Data JPA schemat danych w bazie relacyjnej (tabele, relacje, klucze itp.) zostanie automatycznie utworzony na podstawie zdefiniowanych klas.

### 3.6.1. Produkt

Produkt jest reprezentowany w kodzie aplikacji jako klasa o nazwie *Product*. Przechowuje on główne informacje o produkcie takie jak jego unikalny identyfikator, nazwę, cenę, opis oraz ilość.

**Listing 3.2.** Product.java

```
1 @Entity(name = "products")
2 @Getter
3 @Setter
4 @NoArgsConstructor
5 @AllArgsConstructor
6 public class Product {
7
8     @Id
9     @GeneratedValue(generator = "uuid")
10    @GenericGenerator(name = "uuid", strategy = "uuid2")
11    @Column(name = "id")
12    @NotNull
13    private UUID uuid;
14
15    @NotNull
16    private String name;
17
18    @NotNull
19    private BigDecimal price;
20
21    @NotNull
22    private String description;
23 }
```

W klasie zostały umieszczone adnotacje Spring - nazwy poprzedzone znakiem @. Służą one do konfiguracji klasy.

Pierwsza adnotacja *@Entity* oznacza klasę jako encję, czyli obiekt, który będzie używany do przechowywania danych pochodzących z bazy danych. Encja tej klasy jest mapowana na dane przechowywane w tabeli *products*, gdyż tak właśnie została ona skonfigurowana przy użyciu adnotacji *@Entity* za pomocą właściwości *name = "products"*.

Dodatkowe adnotacje takie jak *@Getter*, *@Setter*, *@NoArgsConstructor* oraz *@AllArgsConstructor* pochodzą z biblioteki Lombok i umożliwiają automatyczne tworzenie powtarzalnego kodu takiego jak *getter*y, *setter*y oraz *konstruktory*. Dzięki



temu nie ma potrzeby pisania dodatkowych linijek kodu wewnątrz klasy. Nie zawsze generowanie wszystkich getterów i setterów do pól jest dobrym rozwiązaniem, jednak w przypadku encji jest to wymagane, gdyż są one używane przez bibliotekę do mapowania relacyjno-obiektowego.

Jako unikalny identyfikator obiektu został zastosowany typ UUID (ang. *Universally Unique Identifier*). Jest to 128-bitowa, generowana losowo wartość w postaci hexadecymalnej. Dzięki zastosowanemu algorytmowi generowania i ogromnej ilości kombinacji UUID zapewnia unikalność każdego wygenerowanego identyfikatora. W systemach rozproszonych tego typu identyfikatory zapewniają unikalność, niż wykorzystanie sekwencji, gdyż identyfikatory stworzone na podstawie sekwencji są unikalne jedynie w obrębie jednej bazy danych [18].

UUID jest automatycznie generowany dzięki użyciu adnotacji *@GeneratedValue* oraz *@GenericGenerator*. Odatkowo pole *uuid* jest oznaczane jako klucz główny tabeli za pomocą adnotacji *@Id*.

Ostatnim użytym typem adnotacji jest *@NotNull*, która zapewnia, że pole zostanie zainicjalizowane jakąś wartością, czyli nie przyjmie wartości *null*.

### 3.6.2. Użytkownik

Użytkownik sklepu jest odzwierciedlony w kodzie aplikacji jako klasa *Customer*. Przechowuje ona informacje o identyfikatorze użytkownika, jego nazwie, adresie, stanie konta oraz wszystkich jego zamówieniach.

**Listing 3.3.** Customer.java

```
1 @Entity(name = "customers")
2 @Getter
3 @Setter
4 @NoArgsConstructor
5 @AllArgsConstructor
6 public class Customer {
7
8     @Id
9     @GeneratedValue(generator = "uuid")
10    @GenericGenerator(name = "uuid", strategy = "uuid2")
11    @Column(name = "id")
12    @NotNull
13    private UUID uuid;
14
15    @NotNull
16    private String name;
17
18    @Embedded
19    @NotNull
20    private Address address;
21
22    @NotNull
23    private BigDecimal balance;
24
25    @OneToMany(mappedBy = "customer", cascade = CascadeType.ALL)
26    private List<Order> orders;
27 }
```

Klasa *Customer* wygląda podobnie do klasy *Product* pod względem konfiguracji. Posiada jednak dwie dodatkowe adnotacje.

Pierwsza z nich to *@Embedded* na polu *address*. Adres użytkownika jest reprezentowany przez więcej, niż jedno pole, dlatego został stworzony specjalny obiekt klasy *Address*. Dzięki tej adnotacji nie ma potrzeby tworzenia dodatkowej tabeli

przechowującej adres - wszystkie pola obiektu *Address* zostaną dodane do tabeli *customers*.

**Listing 3.4.** Address.java

---

```

1  @Getter
2  @Setter
3  @NoArgsConstructor
4  @AllArgsConstructor
5  @Embeddable
6  public class Address {
7
8      @Column(nullable = false)
9      private String street;
10
11     @Column(nullable = false)
12     private String city;
13
14     @Column(nullable = false)
15     private String postCode;
16 }
```

---

Klasa *Address* oznaczona jest adnotacją *@Embeddable*. Oznacza to, że nie musi istnieć osobna tabela do przechowywania adresów - może być on wbudowany (ang. *embedded*) w inną tabelę.

Druga dodatkowa adnotacja użyta w klasie *Customer*, to *@OneToMany* na polu *orders*. Mówi ona o tym, że jest to relacja jeden do wielu (ang. *one to many*) z encją typu *Orders*. Oznacza to, że użytkownik będzie mógł posiadać wiele zamówień przechowywanych w innej tabeli.

### 3.6.3. Zamówienie

Kolejnym elementem modelu jest zamówienie, które jest przypisane do konkretnego użytkownika. Klasa reprezentująca zamówienie o nazwie *Order* zawierająca unikalny identyfikator zamówienia, datę jego wykonania, typ płatności, listę pozycji wchodzących w jego skład oraz informacje o użytkowniku, który dokonał zamówienia.

**Listing 3.5.** Order.java

---

```

1  @Entity(name = "orders")
2  @Getter
3  @Setter
4  @NoArgsConstructor
5  @AllArgsConstructor
6  public class Order {
7
8      @Id
9      @GeneratedValue(generator = "uuid")
10     @GenericGenerator(name = "uuid", strategy = "uuid2")
11     @Column(name = "id")
12     @NotNull
13     private UUID uuid;
14
15     @OneToMany(mappedBy = "order", cascade = CascadeType.ALL)
16     @NotNull
17     private List<OrderItem> orderItems;
18
19     @NotNull
20     private LocalDateTime dateTime;
21
22     @Enumerated(EnumType.STRING)
23     @NotNull
24     private PaymentType paymentType;
```

---

```
25
26     @ManyToOne
27     @JoinColumn(name = "customer_uuid")
28     @NotNull
29     private Customer customer;
30 }
```

Zamówienie jest powiązane z jego elementami za pomocą relacji jeden do wielu (*@OneToMany*) oraz z użytkownikiem relacją wiele do jednego (*@ManyToOne*), gdyż użytkownik może posiadać wiele zamówień. Adnotacja *@JoinColumn* mówi o tym na podstawie której kolumny tworzona jest relacja.

### 3.6.4. Element zamówienia

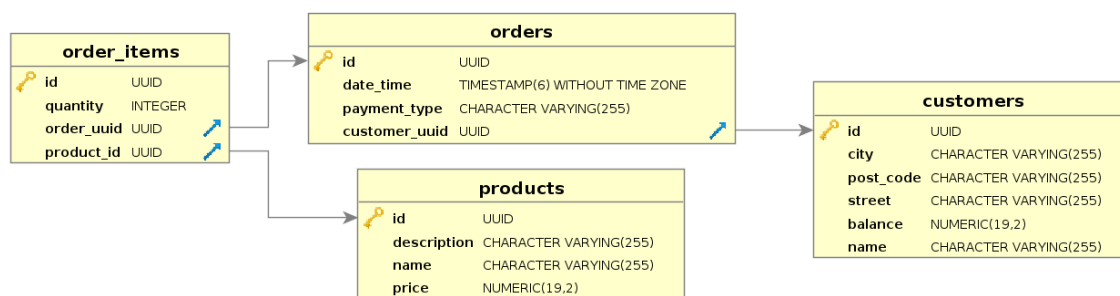
Klasa *OrderItem* reprezentuje każdy z elementów, który wchodzi w skład zamówienia. Przechowuje ona informacje o konkretnym produkcie (relacja jeden-do-jednego) oraz ilość zamówionych produktów.

Listing 3.6. OrderItem.java

```
1 @Entity(name = "orderItems")
2 @Getter
3 @Setter
4 @NoArgsConstructor
5 @AllArgsConstructor
6 public class OrderItem {
7
8     @Id
9     @GeneratedValue(generator = "uuid")
10    @GenericGenerator(name = "uuid", strategy = "uuid2")
11    @Column(name = "id")
12    @NotNull
13    private UUID uuid;
14
15    @NotNull
16    private Integer quantity;
17
18    @NotNull
19    @OneToOne
20    private Product product;
21
22    @ManyToOne
23    @JoinColumn(name = "order_uuid")
24    @NotNull
25    private Order order;
26 }
```

### 3.6.5. Schemat danych

Na rysunku 3.3 przedstawiony jest schemat ilustrujący relacje oraz model danych. Został on wygenerowany za pomocą programu *DbVisualizer* na podstawie istniejącej struktury bazy.



Rys. 3.3. Struktura relacyjnej bazy danych

## 3.7. API

API umożliwia użytkownikowi korzystanie z aplikacji za pomocą zapytań HTTP.

Funkcjonalności, jakie udostępnia aplikacja są operacjami na modelu danych takie jak dodawanie, wyszukiwanie, usuwanie oraz modyfikacja. W szczególności jest to na przykład dodanie użytkownika, dodanie produktu lub jego wyszukanie, stworzenie zamówienia itp.

Spis wszystkich dostępnych akcji jest widoczny na widoku z dokumentacją API (rysunek 3.7.2). Została ona automatycznie wygenerowana za pomocą narzędzia Swagger, którego konfiguracja została opisana w rozdziale 3.7.2.

Akcje są pogrupowane pod względem typu danych, którego dotyczą. Przykładowo wszystkie operacje dotyczące produktów sklepowych znajdują się w sekcji *products-rest-controller*. Oznacza to, że wszystkie one znajdują się w jednej klasie REST kontrolera Spring, który umożliwia właśnie udostępnienie API aplikacji.

### 3.7.1. Kontrolery REST

Spring umożliwia tworzenie kontrolerów REST, które definiują jakie akcje będą obsługiwane po przyjsciu odpowiednich zapytań HTTP.

Jako przykład zostanie przedstawiony kontroler dotyczący produktów oraz dwie akcje: dodanie oraz pobranie jednego produktu. Wszystkie inne kontrolery są zaimplementowane w analogiczny sposób.

W projekcie aplikacji klasa reprezentująca kontroler służący do obsługi zapytań HTTP dotyczących akcji na produktach nosi nazwę *ProductsRestController* i jest przedstawiona na listingu 3.7

Listing 3.7. ProductsRestController.java


```

1 @RestController
2 @RequestMapping(value = "/products",
3     produces = MediaType.APPLICATION_JSON_VALUE,
4     consumes = MediaType.APPLICATION_JSON_VALUE)
5 public class ProductsRestController {
6
7     private final ProductsWriteApplicationService writeApplicationService;
8     private final ProductsReadApplicationService readApplicationService;
9
10    public ProductsRestController(
11        final ProductsWriteApplicationService writeApplicationService,
12        final ProductsReadApplicationService readApplicationService) {
13        this.writeApplicationService = writeApplicationService;
14        this.readApplicationService = readApplicationService;
  
```

```

15     }
16
17     @RequestMapping(method = RequestMethod.POST)
18     public ResponseEntity<?> createProduct(@RequestBody final CreateProductRequest
19         request) {
20         final UUID uuid = writeApplicationService.create(request.asCommand());
21
22         return new ResponseEntity<>(new ObjectCreatedResponse(uuid),
23             HttpStatus.CREATED);
24     }
25
26     @RequestMapping(value = "/{uuid}", method = RequestMethod.GET)
27     public ResponseEntity<?> getProduct(@PathVariable final UUID uuid) {
28         final ProductReadModel product = readApplicationService.getProduct(new
29             GetByIdQuery(uuid));
30         return new ResponseEntity<>(product, HttpStatus.OK);
31     }
32
33     @ResponseStatus(HttpStatus.NOT_FOUND)
34     @ExceptionHandler(ProductNotFoundException.class)
35     @ResponseBody
36     public ResponseEntity handle(final ProductNotFoundException exception) {
37         return ErrorResponse
38             .prepare("Product not found", exception.getMessage(), HttpStatus.NOT_FOUND);
39     }
40 }

```

 swagger

default (/v2/api-docs) ▾

Explore

# Api Documentation

Api Documentation

[Apache 2.0](#)

basic-error-controller : Basic Error Controller

Show/Hide | List Operations | Expand Operations

customers-rest-controller : Customers Rest Controller

Show/Hide | List Operations | Expand Operations

orders-rest-controller : Orders Rest Controller

Show/Hide | List Operations | Expand Operations

products-rest-controller : Products Rest Controller

Show/Hide | List Operations | Expand Operations

GET

/customers

getCustomers

POST

/customers

createCustomer

DELETE

/customers/{uuid}

deleteCustomer

GET

/customers/{uuid}

getCustomer

PUT

/customers/{uuid}

updateCustomer

POST

/orders

createOrder

GET

/products

getProducts

POST

/products

createProduct

DELETE

/products/{uuid}

deleteProduct

GET

/products/{uuid}

getProduct

PUT

/products/{uuid}

updateProduct

[ BASE URL: / , API VERSION: 1.0 ]

Rys. 3.4. Spis dostępnych operacji na API

Najważniejsze elementy listingu 3.7 dotyczące klasy to:

- **linia 1** - adnotacja *@RestController* oznacza klasę jako kontroler, który będzie umożliwiał odbieranie metod HTTP,
- **linia 2** - adnotacja *@RequestMapping* tworzy ścieżkę bazową, pod którą dostępne będą metody danego kontrolera. Wartość podawana jest przez atrybut *value*. Atrybut *produces* oraz *consumes* mówią o formacie danych, które będą zwracane oraz pobierane. W obu przypadkach jest to format JSON,
- **linia 7 i 8** - deklaracja dwóch serwisów, które umożliwią zapis oraz odczyt danych. Serwisy bardziej szczegółowo zostały opisane w rozdziale 3.8.
- **linia 10** - konstruktor, który jako parametry przyjmuje serwisy. Jest to mechanizm tzw. wstrzykiwania zależności (ang. *dependency injection*), który umożliwia zainicjalizowanie serwisu odpowiednią wartością w zależności od konfiguracji aplikacji np. serwis do odczytu danych z bazy relacyjnej lub NoSQL. Więcej o wstrzykiwaniu zależności w rozdziale 3.10.2.

Najważniejsze elementy listingu 3.7 dotyczące metody tworzenia produktu (*createProduct*) to:

- **linia 17** - adnotacja *@RequestMapping* dotycząca konkretnej akcji. Parametr *method* określa metodę HTTP, która jest obsługiwana przez daną funkcję,
- **linia 18** - definicja metody służącej do dodawania produktu. Jako parametr przyjmuje obiekt zapytania (ang. *request*), który oznaczony jest adnotacją *@RequestBody*. Umożliwi to wysyłanie parametrów zapytania jako obiekt w formacie JSON,
- **linia 19** - użycie serwisu, który stworzy produkt na podstawie obiektu zapytania,
- **linia 21** - zwrócenie odpowiedzi jako obiekt *ResponseEntity* zawierający dane oraz kod statusu odpowiedzi.

Metodę służącą do pobierania produktu można zaimplementować analogicznie. Różni się ona jedynie kilkoma szczegółami.

W kontrolerach dodatkowo implementuje się mechanizmy reagujące na błędy ze strony użytkownika. Najważniejsze elementy listingu 3.7 dotyczące obsługi błędów to:

- **linia 30** - adnotacja *@ResponseStatus* definiuje jaki kod odpowiedzi zostanie zwrócony,
- **linia 31** - adnotacja *@ExceptionHandler* konfiguruje jaki typ wyjątku powinien być obsługiwany przez metodę,
- **linia 33** - definicja metody *handle* umożliwiającej obsługę błędów,
- **linia 34** - zwracany jest obiekt zawierający informację o błędzie oraz odpowiedni kod statusu.

### 3.7.2. Konfiguracja Swagger

Konfiguracja aplikacji Spring odbywa się przeważnie przez implementację klas konfiguracyjnych oznaczonych adnotacją *@Configure*.

W celu użycia narzędzia Swagger należy zaimplementować jedynie jedną klasę konfiguracyjną przedstawioną na listingu 3.8.

**Listing 3.8.** SwaggerConfiguration.java

```
1 @Configuration
2 @EnableSwagger2
3 public class SwaggerConfiguration {
4
5     @Bean
6     public Docket api() {
7         return new Docket(DocumentationType.SWAGGER_2)
8             .select()
9             .paths(PathSelectors.any())
10            .build();
11    }
12 }
```

Klasa ta aktywuje narzędzie Swagger oraz tworzy obiekt z konfiguracją, który będzie widoczny w całym kontekście Springa. Jest to tzw. *Bean*.

Po dodaniu konfiguracji wszystkie metody z kontrolerów będą dodawane do widoku z dokumentacją API dostępną pod adresem *http://{adres-hosta-aplikacji}/swagger-ui.html*.

### 3.7.3. Wysyłanie zapytań do aplikacji

Dzięki bibliotece Swagger ręczne wysyłanie zapytań jest bardzo proste. Po rozwinięciu danej akcji ukaże się dodatkowy opis API. Możliwe jest dodanie opisu akcji, wyświetlany jest model zapytania oraz model odpowiedzi wraz z przykładowymi wartościami.

Na samym dole rysunku 3.5 znajduje się przycisk *Try it out!*, który pozwala na wysłanie zapytania do aplikacji ze wskazanymi parametrami. Odpowiedź zostanie wyświetlona poniżej. Zostało to ukazane na rysunku 3.6.

POST

/products

createProduct

**Implementation Notes**  
Umożliwia dodanie nowego produktu do bazy danych sklepu.

**Response Class (Status 200)**  
OK

Model | Example Value

```
{
  "uuid": "string"
}
```

Response Content Type

**Parameters**

Parameter	Value	Description	Parameter Type	Data Type
request	<pre>{   "name": "Komputer PC",   "price": 2000,   "description": "Opis komputera" }</pre> <p>Parameter content type: <input type="text" value="application/json"/></p>	request	body	Model   Example Value <pre>{   "name": "string",   "price": 0,   "description": "string" }</pre>

**Response Messages**

HTTP Status Code	Reason	Response Model	Headers
201	Created		
401	Unauthorized		
403	Forbidden		
404	Not Found		

Rys. 3.5. Opis akcji

Response Body

```
{
  "uuid": "7392132b-f1be-45cd-8d81-35bde8861ee1"
}
```

Response Code

201

Rys. 3.6. Odpowiedź aplikacji

## 3.8. Wykonywanie logiki biznesowej

Wszelkie operacje na modelu danych wykonują tzw. serwisy. Klasy serwisów są w Springu oznaczane adnotacją `@Service`. Umożliwia to automatyczne utworzenie obiektu (tzw. *Bean*), który widoczny będzie w całym kontekście Springa.

Serwisy przetwarzają dane, a następnie utrwalają ich stan w bazie danych za pomocą tzw. repozytoriów, które zostaną opisane w rozdziale 3.9. Jako przykład zostanie zaprezentowany serwis służący do zapisu danych dotyczących produktów



widoczny na listingu 3.9.

**Listing 3.9.** ProductsWriteApplicationService.java

```
1 @Service
2 public class ProductsWriteApplicationService {
3
4     private final ProductWriteRepository writeRepository;
5     private final ApplicationEventPublisher eventPublisher;
6
7     public ProductsWriteApplicationService(
8         final ProductWriteRepository writeRepository,
9         final ApplicationEventPublisher eventPublisher) {
10         this.writeRepository = writeRepository;
11         this.eventPublisher = eventPublisher;
12     }
13
14     @Transactional
15     public UUID create(final CreateProductCommand command) {
16         final Product savedEntity = saveProduct(command);
17         eventPublisher.publishEvent(ProductCreatedEvent.from(savedEntity));
18         return savedEntity.getUuid();
19     }
20
21     private Product saveProduct(final CreateProductCommand command) {
22         final Product entity = Product.fromCommand(command);
23         return writeRepository.save(entity);
24     }
25 }
```

Klasa *ProductsWriteApplicationService* wykorzystuje dwie zależności, które są jej dostarczone poprzez konstruktor. Pierwszą z nich jest obiekt repozytorium umożliwiający zapis danych do bazy - klasa *ProductsWriteRepository*, która została ręcznie zaimplementowana. Natomiast drugą zależnością jest gotowy obiekt klasy *ApplicationEventPublisher*, który jest udostępniany przez Spring i służy do publikacji zdarzeń (ang. *event*). Tutaj obiekt ten jest używany do zgłoszenia zdarzenia świadczącego o stworzeniu nowego produktu. Mechanizm zdarzeń zostanie opisany w rozdziale 3.12, gdyż został on użyty do synchronizacji danych z bazą NoSQL.

Klasa posiada jedną metodę publiczną *create*, która na podstawie przekazanego obiektu komendy *CreateProductCommand* utworzy obiekt produktu i zapisze go w bazie danych. Jest ona również oznaczona adnotacją *@Transactional*, która zapewnia wykonanie wszystkich operacji w ramach jednej transakcji.

Co prawda transakcyjność w tym serwisie nie jest wymagana, gdyż wykonywany jest jedynie jeden zapis do bazy. Zalecane jest jednak wykonywanie w transakcji każdej metody modyfikującej stan w serwisach aplikacyjnych. Transakcje są niezbędne przy wykonywaniu kilku operacji modyfikacji danych, które razem tworzą jedną logiczną całość. Dzięki transakcji w przypadku wystąpienia błędu w trakcie wprowadzania zmian, zostaną one wycofane. Zapewnia to, że dane będą znajdować się zawsze w spójnym stanie.

### 3.8.1. Komendy oraz kwerendy

Zgodnie z założeniami architektury CQRS istnieją osobne serwisy do zapisu oraz odczytu danych. Posiadają one metody, które do zapisu przyjmują obiekty komend, natomiast do odczytu są to obiekty kwerend.

Analizując nagłówek metody tworzącej produkt można zauważyć, że jako parametr przyjmuje ona właśnie obiekt komendy:

**Listing 3.10.** Nagłówek metody tworzącej produkt

---

```
1 public UUID create(final CreateProductCommand command)
```

---

Obiekt tej klasy służy do przechowywania informacji, na podstawie których będzie możliwe utworzenie właściwego obiektu produktu.

**Listing 3.11.** CreateProductCommand

---

```
1 public class CreateProductCommand implements Command {
2
3     private final String name;
4     private final BigDecimal price;
5     private final String description;
6     // ...
7 }
```

---

Obiekty kwerend są bardzo podobne, z tą różnicą, że używa się ich jako argumenty funkcji służących do odczytu danych. Zawierają one informacje na temat kryteriów wyszukiwania. Przykład nagłówka metody pobierającej dane o produkcie na podstawie jego identyfikatora widoczny jest na listingu 3.12.

**Listing 3.12.** Nagłówek metody służącej do pobierania danych produktu

---

```
1 public ProductReadModel getItem(final GetByIdQuery query)
```

---

Obiekt kwerendy jedynie przechowuje dane, które zostaną użyte jako kryterium wyszukiwania.

**Listing 3.13.** GetByIdQuery

---

```
1 public class GetByIdQuery implements Query {
2
3     private final UUID uuid;
4
5     // ...
6 }
```

---

## 3.9. Komunikacja z bazą danych

Za połączenie z bazą danych oraz wykonywanie na niej operacji odpowiadają tzw. repozytoria (ang. *repository*). W Springu są one oznaczane adnotacją *@Repository*.

Dzięki bibliotece *spring-boot-starter-data-jpa* dostępne są gotowe generyczne klasy repozytoriów, które mają zaimplementowane wszystkie podstawowe operacje na danych takie jak ich zapis, odczyt, wyszukiwanie po polu itp. Wystarczy jedynie stworzyć pusty interfejs rozszerzający *CrudRepository* z odpowiednim typem. Dzięki temu możliwe jest wykonywanie operacji na danych bez potrzeby pisania zapytań SQL.

Przykładem wykorzystania gotowych implementacji jest klasa służąca do pobierania informacji o produkcie *ProductReadRepository*. Jest to interfejs, który rozszerza *CrudRepository*. Dzięki temu dziedziczy po nim takie metody jak np. *findAll* oraz *findOne*.

**Listing 3.14.** ProductReadRepository

---

```
1 @Repository
2 public interface ProductReadRepository extends CrudRepository<Product, UUID> {
3
4 }
```

---

Możliwe jest również tworzenie metod wykorzystujących zapytania SQL używając adnotacji *@Query*. Należy ją umieścić nad nagłówkiem metody i jako parametr podać ciąg znaków z zapytaniem. Jest to lekko zmodyfikowany SQL, który umożliwia na przykład używanie nazw klas (które są encjami) zamiast nazw tabel. Zostanie to automatycznie skonwertowane na odpowiadającą encji nazwę tabeli w bazie danych.

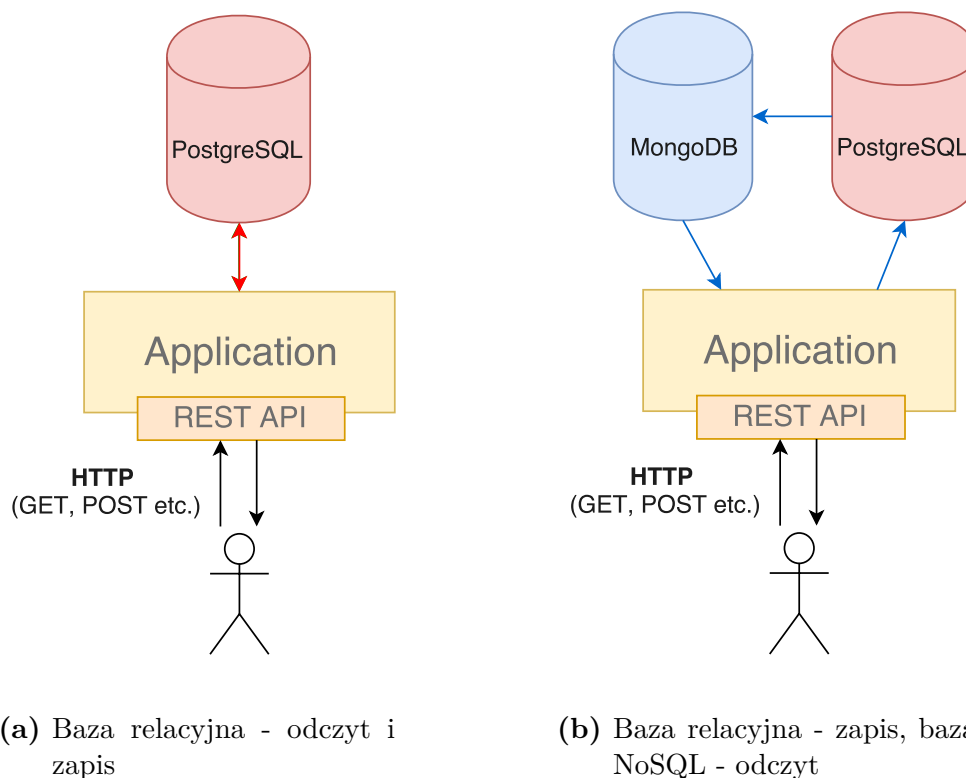
**Listing 3.15.** Przykład wykorzystania adnotacji *@Query*

```
1 @Query("SELECT product FROM Product product WHERE product.uuid = ?")
2 Product findProductByUuid(String uuid);
```

## 3.10. Zmiana źródła odczytu danych

Zgodnie z założeniami aplikacja może pracować w dwóch trybach:

- wszystkie operacja zapisu oraz odczytu danych kierowane są do bazy relacyjnej,
- operacje zapisu danych wykonywane są na bazie relacyjnej, następnie zmiany wprowadzane są do bazy NoSQL (synchronizacja) i odczyty danych są realizowane jedynie z bazy NoSQL.



**Rys. 3.7.** Tryby pracy aplikacji

Możliwe jest używanie jednej z tych dwóch opcji w zależności od konfiguracji aplikacji. Jest to zrealizowane za pomocą tzw. przełącznika funkcjonalności (ang. *feature switch*).

### 3.10.1. Przełącznik funkcjonalności

Przełącznik funkcjonalności umożliwia aktywację/deaktywację lub zmianę danej funkcjonalności, dla której jest wykonany. W tym wypadku przy uruchomieniu aplikacji będzie można wybrać skąd mają być odczytywane dane, czyli z bazy relacyjnej lub NoSQL.

W projekcie Spring domyślnie w katalogu z zasobami */resources* znajduje się plik *application.properties*. Przechowuje on właściwości, które używane są do konfiguracji projektu. Można w nim definiować własne właściwości, które mogą zostać użyte np. do wykonania przełącznika funkcjonalności.

Do pliku *application.properties* została dodana właściwość *read.source*. Będzie ona przyjmować dwie wartości: *nosql* lub *rdbms*, które odpowiednio będą oznaczały, że źródłem odczytu danych ma być baza NoSQL lub baza relacyjna (RDBMS).

---

#### Listing 3.16. application.properties

---

```
1 read.source=nosql
```

---

Tak zdefiniowana właściwość może zostać użyta do podjęcia decyzji jakiego serwisu do pobierania danych z bazy użyć. Użycie odpowiedniego serwisu będzie możliwe dzięki mechanizmowi polimorfizmu (ang. *polymorphism*) oraz wstrzykiwania zależności (ang. *dependency injection*).

### 3.10.2. Wstrzykiwanie zależności oraz polimorfizm

Mechanizm wstrzykiwania zależności polega na przekazaniu do obiektu przez konstruktor wszystkich jego zależności, czyli wszystkich innych obiektów, które będzie on wykorzystywał. Przykładem wykorzystania tego mechanizmu jest klasa *ProductsRestController*, która korzysta z dwóch serwisów aplikacyjnych.

---

#### Listing 3.17. ProductsRestController

---

```
1 public class ProductsRestController {
2
3     private final ProductsWriteApplicationService writeApplicationService;
4     private final ProductsReadApplicationService readApplicationService;
5
6     public ProductsRestController(
7         final ProductsWriteApplicationService writeApplicationService,
8         final ProductsReadApplicationService readApplicationService) {
9         this.writeApplicationService = writeApplicationService;
10        this.readApplicationService = readApplicationService;
11    }
12
13    // ...
14 }
```

---

Obie zależności są przekazane przez konstruktor. Każdy z obiektów może być klasą lub interfejsem. Użycie interfejsu przy wstrzykiwaniu zależności pozwala na zastosowanie polimorfizmu, czyli wyboru jego implementacji w zależności od kontekstu. Dodatkowo takie podejście znacząco zwiększa tzw. testowalność klasy (ang. *testability*).

Przykładem jest interfejs *ProductsReadApplicationService*, który definiuje wymagane metody dla klas, które będą go rozszerzać. Jest to jeden z parametrów konstruktora omawianego kontrolera.

**Listing 3.18.** ProductsReadApplicationService

---

```

1 public interface ProductsReadApplicationService {
2
3     List<ProductsReadModel> getProducts();
4
5     ProductsReadModel getProduct(GetByIdQuery query);
6 }

```

---

W projekcie istnieją dwie klasy rozszerzające ten interfejs: *ProductsReadFromNoSqlApplicationService* do wykonywania odczytów z bazy NoSQL oraz *ProductsReadFromRdbmsApplicationService* do wykonywania odczytów z relacyjnej bazy danych. Każda z tych klas używa odpowiedniego repozytorium dla danego typu bazy. Implementacja interfejsu jest wybierana na podstawie ustawionej wartości właściwości *read.source*. Jest to zrealizowane za pomocą adnotacji *@ConditionalOnProperty*.

Na listingu 3.19 oraz 3.20 przedstawione są obie klasy serwisów (implementacja metod na listingu została celowo pominięta, aby zwiększyć czytelność).

**Listing 3.19.** ProductsReadFromRdbmsApplicationService

---

```

1 @Service
2 @ConditionalOnProperty(name = "read.source", havingValue = "rdbms")
3 public class ProductsReadFromRdbmsApplicationService implements
4     ProductsReadApplicationService {
5
6     private final ProductReadFromRdbmsRepository productReadRepository;
7
8     public ProductsReadFromRdbmsApplicationService(
9         final ProductReadFromRdbmsRepository productReadRepository) {
10         this.productReadRepository = productReadRepository;
11     }
12
13     @Override
14     public List<ProductReadModel> getProducts() {
15         // implementacja
16     }
17
18     @Override
19     public ProductReadModel getProduct(final GetByIdQuery query) {
20         // implementacja
21     }
22 }

```

---

**Listing 3.20.** ProductsReadFromNoSqlApplicationService

---

```

1 @Service
2 @ConditionalOnProperty(name = "read.source", havingValue = "nosql", matchIfMissing =
3     true)
4 public class ProductsReadFromNoSqlApplicationService implements
5     ProductsReadApplicationService {
6
7     private final ProductReadFromNoSqlRepository productReadRepository;
8
9     public ProductsReadFromNoSqlApplicationService(
10         final ProductReadFromNoSqlRepository productReadRepository) {
11         this.productReadRepository = productReadRepository;
12     }
13
14     @Override
15     public List<ProductReadModel> getProducts() {
16         // implementacja
17     }
18
19     @Override
20     public ProductReadModel getProduct(final GetByIdQuery query) {
21         // implementacja
22     }
23 }

```

---

Można zauważyć, że każdy serwis korzysta z innego repozytorium, więc zapisuje dane do innej bazy. Kluczowym elementem jest adnotacja *@ConditionalOnProperty*, która na podstawie wartości właściwości *read.source* wybiera odpowiednią implementację interfejsu.

Wystarczy więc przed uruchomieniem aplikacji ustawić *read.source=nosql*, aby odczytywać dane z bazy NoSQL lub *read.source=rdbms*, aby dane były odczytywane z bazy relacyjnej.

## 3.11. Model danych do odczytu

W CQRS model danych przechowywany w bazie NoSQL często określany jest angielskim określeniem *read model*, czyli model do odczytu. Struktura danych modelu do odczytu może się znacznie różnić od tego, który zapisywany jest w bazie relacyjnej. Może istnieć również wiele modeli, które na przykład będą przystosowane do różnych widoków.

Jednak główną zaletą stosowania modelu do odczytu jest możliwość zapisania danych bez użycia relacji. Wykorzystywana baza MongoDB jest bazą dokumentową, która przechowuje dane w formacie JSON. Format ten umożliwia odwzorowanie rzeczywistej struktury obiektu w bazie danych, gdyż możliwe jest tworzenie zagnieżdżeń. Dzięki temu zyskuje się na wydajności, gdyż dane do pobrania są przechowywane w docelowej postaci. Nie ma więc potrzeby wykonywania żadnych operacji na danych w przeciwieństwie do baz relacyjnych (łączenie danych połączonych relacjami). Dane są od razu gotowe do odczytu, więc wystarczy je jedynie pobrać.

W przykładowej aplikacji sklepu dane pogrupowane są w bazie na dwie kolekcje. Jedna z nich przechowuje informacje o produktach, natomiast druga o użytkownikach wraz z ich zamówieniami.

### 3.11.1. Kolekcja produktów

Pierwsza kolekcja o nazwie *products* przechowuje informacje o produktach. Struktura danych jest bardzo podobna do struktury tabeli, gdyż jest to prosty obiekt z kilkoma polami. Przykładowy dokument produktu przechowywany jest w bazie w następujący sposób:

**Listing 3.21.** Dokument z kolekcji *products*

```
1 {  
2   "_id" : "d829ca68-2efa-43a3-a197-20553b98ffff",  
3   "_class" : "edu.uz.inz.port.adapter.repository.read.ProductReadModel",  
4   "name" : "Komputer PC",  
5   "price" : "2000",  
6   "description" : "Opis komputera"  
7 }
```

Dokumenty tworzone są poprzez serializację obiektów do formatu JSON. Z wartości pola *"\_class"* można odczytać, z jakiego typu obiektu powstał dokument. Na listingu 3.21 przedstawiony jest obiekt klasy *ProductReadModel* poddany serializacji do JSON.

### 3.11.2. Kolekcja użytkowników

Druga stworzona w bazie kolekcja nosi nazwę *customers* i przechowuje informacje o użytkowniku wraz z jego zamówieniami. Zamówienia nie zawierają w sobie jedynie identyfikatora elementu zamówienia tak jak w przypadku bazy relacyjnej - są one w nim bezpośrednio zawarte wraz z obiektami produktów. W wyniku tego w ramach jednego dokumentu przechowywany jest kompletny zestaw danych na temat danego użytkownika, czyli informacje o nim, jego zamówienia oraz zamówione produkty.

**Listing 3.22.** Dokument z kolekcji *customers*

```
1 {  
2   "_id" : "51d35b06-e881-42c8-a9b7-e0cadfbaf886",  
3   "_class" : "edu.uz.inz.port.adapter.repository.read.CustomerReadModel",  
4   "name" : "Krystian Dziędziola",  
5   "address" : {  
6     "street" : "Ulica 1/23",  
7     "city" : "Zielona Góra",  
8     "postCode" : "12-345"  
9   },  
10  "balance" : "5000",  
11  "orders" : [  
12    {  
13      "uuid" : "83135bd2-a1a0-4788-8aad-527f11615433",  
14      "dateTime" : "2017-12-11T20:05:00.196",  
15      "items" : [  
16        {  
17          "uuid" : "33107f0f-2365-4ae4-9ba2-c8b28098e471",  
18          "quantity" : 2,  
19          "product" : {  
20            "_id" : "d829ca68-2efa-43a3-a197-20553b98ffff",  
21            "name" : "Komputer PC",  
22            "price" : "2000.00",  
23            "description" : "Opis komputera"  
24          }  
25        }  
26      ],  
27      "paymentType" : "CASH"  
28    }  
29  ]  
30 }
```

Na listingu 3.22 ukazany jest obiekt *CustomerReadModel* poddany serializacji do JSON i zapisany w bazie danych. Zawiera on w sobie również obiekt adresu typu *AddressReadModel* oraz listę zamówień typu *OrderReadModel*. Zamówienia przechowują również informacje o elementach zamówienia (*OrderItemReadModel*), a te z kolei o produktach jako obiekt typu *ProductReadModel*.

Można zauważyć, że zapisywanie obiektu produktu w zamówieniu jest z pewnością duplikowaniem danych, gdyż informacje o produktach znajdują się już w innej kolekcji *products*. Duplikacja danych jest tutaj niezbędna w celu uzyskania większej wydajności. Przeważnie ilość danych nie jest problemem, gdyż przestrzeń dyskowa nie jest zbyt droгим zasobem. Kluczowym elementem jest często wydajność systemu, więc duplikowanie danych na rzecz lepszej wydajności jest w większości przypadków całkowicie akceptowalna.

## 3.12. Synchronizacja danych

Każda zmiana danych wprowadzona do bazy relacyjnej musi nieść za sobą aktualizację danych w bazie NoSQL. W przeciwnym razie pobierane dane (*read model*)

będą niezgodne z rzeczywistym stanem systemu. W celu realizacji synchronizacji danych pomiędzy obiema bazami został wykorzystany mechanizm publikacji zdarzeń dostarczony przez Spring.

Za każdym razem, gdy dane ulegają zmianie, powoduje to zgłoszenie odpowiedniego zdarzenia (*event*), w którym zawarte są informacje o zmianach. Przykładowo, gdy zostaje dodany nowy użytkownik zgłaszane jest zdarzenie mówiące o stworzeniu użytkownika. Jest to realizowane w serwisie aplikacyjnym o nazwie *CustomersWriteApplicationService* przedstawionym na listingu 3.23.

### Listing 3.23. CustomersWriteApplicationService

---

```

1  @Service
2  public class CustomersWriteApplicationService {
3
4      private final CustomerWriteRepository customerWriteRepository;
5      private final ApplicationEventPublisher eventPublisher;
6
7      // ...
8
9      @Transactional
10     public UUID create(final CreateCustomerCommand command) {
11         final Customer customer = saveCustomer(command, address);
12         eventPublisher.publishEvent(CustomerCreatedEvent.from(customer));
13         return customer.getUuid();
14     }
15
16     // ...
17 }

```

---

Kluczowym elementem jest tu linia 12, w której wywoływana jest metoda *publishEvent* na obiekcie typu *ApplicationEventPublisher*. Jest to mechanizm dostarczony przez Spring, który może zostać użyty do zgłaszania zdarzeń. Możliwe jest publikowanie swoich własnych zdarzeń jak na przykład zdarzenie o utworzeniu użytkownika zdefiniowane jako klasa *CustomerCreatedEvent*.

### Listing 3.24. CustomerCreatedEvent

---

```

1  public class CustomerCreatedEvent implements DomainEvent {
2
3      private final UUID uuid;
4      private final String name;
5      private final String street;
6      private final String city;
7      private final String postCode;
8      private final BigDecimal balance;
9
10     // ...
11
12     public static CustomerCreatedEvent from(final Customer customer) {
13         // implementacja
14     }
15
16     public CustomerReadModel asReadModel() {
17         // implementacja
18     }
19 }

```

---

Zdarzenie to posiada wszystkie niezbędne dane, aby taki sam użytkownik został dodany również do bazy służącej do odczytu. Dodatkowo ma zaimplementowane metody pomocnicze służące do stworzenia zdarzenia z modelu oraz przekształcenie go do modelu służącego do odczytu.

W celu odebrania zgłoszonego zdarzenia musi istnieć dodatkowy komponent dedykowany do nasłuchiwanie na zdarzenia oraz reagowania na nie określonym za-



chowaniem. Została zaimplementowana klasa *ReadModelOnDomainEventUpdater*, która realizuje tę funkcjonalność. Obsługuje ona wszystkie typy zdarzeń zgłoszone w systemie. Na listingu 3.25 zostanie zaprezentowana jedynie część dotycząca tworzenia nowego użytkownika.

**Listing 3.25.** ReadModelOnDomainEventUpdater

```
1 @Component
2 public class ReadModelOnDomainEventUpdater {
3
4     private final CustomerReadModelUpdateRepository customerRepository;
5     // ...
6
7     @EventListener
8     public void handle(final CustomerCreatedEvent event) {
9         customerRepository.save(event.asReadModel());
10    }
11
12    // ...
13 }
```

Obsługa zdarzenia w Springu jest bardzo prosta. Wystarczy stworzyć metodę, która będzie oznaczona adnotacją *@EventListener* oraz będzie przyjmować jako argument obiekt danego zdarzenia. Po zgłoszeniu zdarzenia danego typu zostanie ono przechwycone przez tę metodę. Wystarczy tylko wykonać odpowiednie operacje w zależności otrzymanego zdarzenia.

Reakcją na otrzymanie zdarzenia o stworzeniu użytkownika jest utworzenie takiego samego użytkownika w bazie służącej do odczytu. Zostało do tego użyte repozytorium wykonujące operacje aktualizacji na bazie MongoDB o nazwie *CustomerReadModelUpdateRepository*.

**Listing 3.26.** CustomerReadModelUpdateRepository

```
1 @Repository
2 public interface CustomerReadModelUpdateRepository extends
3     MongoRepository<CustomerReadModel, String> {
4 }
```

Na listingu 3.26 przedstawione jest repozytorium rozszerzające generyczny interfejs *MongoRepository*, który posiada zaimplementowane podstawowe operacje na danych (podobnie jak *CrudRepository* dla baz relacyjnych).

W metodzie obsługującej zdarzenie wystarczy jedynie przekształcić je w obiekt modelu służącego do odczytu wywołując *event.asReadModel()*, a następnie zapisać otrzymane dane korzystając z metody *save* udostępnionej przez repozytorium.

Dzięki tym operacjom każdy nowo dodany użytkownik będzie również zapisany w bazie NoSQL. Inne operacje takie jak np. dodanie produktu, stworzenie zamówienia itp. zostały zaimplementowane w analogiczny sposób.

### 3.13. Tworzenie obrazu Docker z aplikacją

Chcąc uruchomić swoją aplikację w kontenerze Docker należy najpierw przygotować jej obraz. W tym celu należy stworzyć plik *Dockerfile*, na podstawie którego zbudowany zostanie obraz. Obraz kontenera Docker z aplikacją można stworzyć w kilku prostych krokach ukazanych na listingu 3.28.

---

**Listing 3.27.** Dockerfile

---

```
1 FROM openjdk:8-jdk-alpine
2 ADD target/inz-0.0.1-SNAPSHOT.jar app.jar
3 ENTRYPOINT exec java -jar /app.jar
```

---

W tym przypadku Dockerfile składa się z trzech następujących linii:

- **FROM** - wskazuje obraz bazowy dla kontenera. W tym przypadku jest to *openjdk:8-jdk-alpine*, który zawiera JDK (ang. *Java Development Kit*), czyli narzędzia potrzebne do tworzenia i uruchamiania aplikacji Java,
- **ADD** - dodanie do obrazu pliku JAR (*inz-0.0.1-SNAPSHOT.jar*) z aplikacją i zapisanie go pod nazwą *app.jar*,
- **ENTRYPOINT** - komendy, które zostaną wykonane przy starcie kontenera. W tym przypadku jest to jedynie uruchomienie pliku JAR z aplikacją poleceniem *java -jar*.

### 3.13.1. Budowanie obrazu z linii komend

Obraz kontenera można zbudować programem Docker z linii komend wykorzystując przygotowany plik Dockerfile. W tym celu należy najpierw zbudować projekt, aby stworzony został plik JAR z aplikacją. Następnie należy przejść do folderu, w którym stworzony został plik Dockerfile i wykonać polecenia z listingu 3.28.

---

**Listing 3.28.** Budowanie oraz uruchomienie kontenera

---

```
1 # w katalogu projektu
2 ./gradlew clean build
3
4 # w katalogu z plikiem Dockerfile
5 docker build . -t aplikacja_inz
6
7 # uruchomienie kontenera
8 docker run -p 8080:8080 aplikacja_inz
```

---

Komenda *docker build* to zbudowanie obrazu o nazwie *aplikacja\_inz*, który następnie może być uruchomiony poleceniem *docker run*. Przy uruchomieniu należy również pamiętać o udostępnieniu portu, który umożliwi komunikację z aplikacją, gdyż kontenery Docker są domyślnie odizolowane.

W celu udostępnienia portów należy podczas uruchamiania dodać mapowanie za pomocą przełącznika *-p port\_zewnetrzny:port\_wewnetrzny*. W tym przypadku port, na którym uruchomiona jest aplikacja, czyli 8080 zostanie udostępniony na zewnątrz również jako port 8080.

### 3.13.2. Budowanie obrazu za pomocą Gradle

Budowanie obrazu z linii poleceń co prawda nie jest trudne, jednak wymaga kilku manualnych operacji lub stworzenia odpowiedniego skryptu. Istnieje jednak lepsze rozwiązanie, które zakłada dołączenie procedury budowania obrazu aplikacji do cyklu budowania projektu za pomocą Gradle.

Potrzebny jest jedynie dodatek *gradle-docker* do Gradle oraz kilka linii konfiguracji w pliku *build.gradle*.

**Listing 3.29.** Konfiguracja Docker w Gradle

```
1 buildscript {
2
3 // ...
4     dependencies {
5         classpath("se.transmode.gradle:gradle-docker:1.2")
6         // ...
7     }
8 }
9
10 apply plugin: 'docker'
11
12 // ...
13
14 // ### DOCKER CONFIGURATION ###
15 task buildDocker(type: Docker, dependsOn: build) {
16     applicationName = jar.baseName
17     dockerfile = file("${projectDir}/src/main/docker/Dockerfile")
18     doFirst {
19         copy {
20             from jar
21             into "${stageDir}/target"
22         }
23     }
24 }
```

Na listingu 3.29 w linii 15 tworzone jest specjalne zadanie (*task*) Gradle, które pozwoli na automatyczne zbudowanie obrazu Docker z aplikacją. Zadanie *buildDocker* jest zależne (*dependsOn*) od zadania o nazwie *build*, które służy do budowania projektu. Oznacza to, że gdy zostanie wykonane zadanie *buildDocker*, to przed nim będzie uruchomione również zadanie *build*, które zbuduje projekt w tym również stworzy plik JAR z aplikacją.

Dodatkowe opcje konfiguracyjne to podanie nazwy obrazu (*applicationName*), wskazanie ścieżki do pliku Dockerfile (*dockerfile*) oraz skopiowanie pliku JAR do katalogu *target* znajdującego się w plikach tymczasowych Gradle.

Teraz wystarczy wykonać jedynie polecenie `./gradlew buildDocker` i obraz zostanie automatycznie zbudowany oraz dodany do lokalnego rejestru Docker, skąd będzie mógł być pobrany i uruchomiony.

## 3.14. Środowisko uruchomieniowe

Uruchamiana aplikacja w podstawowej wersji będzie potrzebować dwóch instancji baz danych:

- jedna instancja **PostgreSQL**,
- jedna instancja **MongoDB**.

Dodatkowo aplikacja sama w sobie powinna być automatycznie uruchamiana w kontenerze Docker. Dzięki programowi Docker bazy danych nie muszą być manualnie instalowane na komputerze, a uruchomienie całego środowiska będzie się ograniczało jedynie do użycia jednej komendy.

Istnieje możliwość napisania ręcznie odpowiednich skryptów, jednak w znacznie prostszy sposób można to zrealizować za pomocą oprogramowania dedykowanego dla Docker o nazwie Docker Compose.

Docker Compose umożliwi stworzenie pliku w formacie YAML, w którym zostaną wypisane wszystkie potrzebne kontenery oraz zależności pomiędzy nimi. Umożliwi to uruchomienie całego systemu, czyli aplikacji oraz obu baz danych jednym poleceniem.

**Listing 3.30.** docker-compose.yml

```
1 version: '2'
2 services:
3   inz-app:
4     image: edu.uz/inz:0.0.1-SNAPSHOT
5     ports:
6       - "8080:8080"
7     depends_on:
8       - postgresql
9       - mongodb
10
11  postgresql:
12    image: postgres
13    ports:
14      - "5432:5432"
15    environment:
16      - POSTGRES_PASSWORD=postgres
17      - POSTGRES_USER=postgres
18      - POSTGRES_DB=inz
19
20  mongodb:
21    image: mongo
22    ports:
23      - "27017:27017"
24    environment:
25      - MONGODB_USER=mongo
26      - MONGODB_PASS=mongo
27      - MONGODB_DATABASE=inz
```

W pliku z listingu 3.30 zostały zdefiniowane trzy serwisy (*services*), które posiadają dodane odpowiednie konfiguracje:

- **inz-app** - kontener z aplikacją uruchomiony ze zbudowanego obrazu o nazwie *edu.uz/inz:0.0.1-SNAPSHOT*. Dodatkowo przy uruchamianiu zostanie udostępniony na zewnątrz port 8080. Serwis aplikacji jest zależny (*depends\_on*) od dwóch serwisów baz danych. Oznacza to, że zostanie on uruchomiony dopiero wtedy, gdy uruchomione zostaną oba kontenery z bazami danych,
- **postgresql** - kontener zawierający bazę danych PostgreSQL, który zostanie uruchomiony z obrazu *postgres* pobranego z repozytorium Docker. Dodatkowo ustawione zostało mapowanie portów oraz nazwa użytkownika, hasło oraz nazwa bazy danych,
- **mongodb** - serwis bazy MongoDB. Kontener będzie uruchomiony z obrazu *mongo* pobranego z oficjalnego repozytorium. Dodatkowo zostało skonfigurowane mapowanie portów oraz zmienne środowiskowe dla bazy danych takie jak użytkownik, hasło oraz nazwa bazy danych.

Mając tak przygotowany plik wystarczy jedynie przejść do katalogu, w którym się on znajduje i wywołać polecenie *docker-compose up*. Spowoduje to utworzenie i uruchomienie kontenerów dla każdego z serwisów. Po kilku sekundach aplikacja powinna być uruchomiona, oraz połączona z dwoma bazami danych i gotowa do użytku. Informacje o uruchomionych kontenerach można wyświetlić używając polecenia *docker ps*.

## 3.15. Skalowanie

Ostatnim ważnym etapem projektowania przykładowej aplikacji pokazującej sposób wykorzystania obu typów baz danych jest konfiguracja skalowania bazy służącej do odczytu. Samo zastosowanie bazy NoSQL powinno wpłynąć pozytywnie na wydajność (zostanie to sprawdzone w testach wydajnościowych opisanych w rozdziale 3.16). Jednak w celu zwiększenia wydajności samej bazy NoSQL można wykonać jej skalowanie.

Dzięki budowie i sposobie działania baz nierelecyjnych są one przystosowane do pracy w klastrach i umożliwiają partycjonowanie danych. MongoDB posiada mechanizm skalowania o nazwie *sharding*.

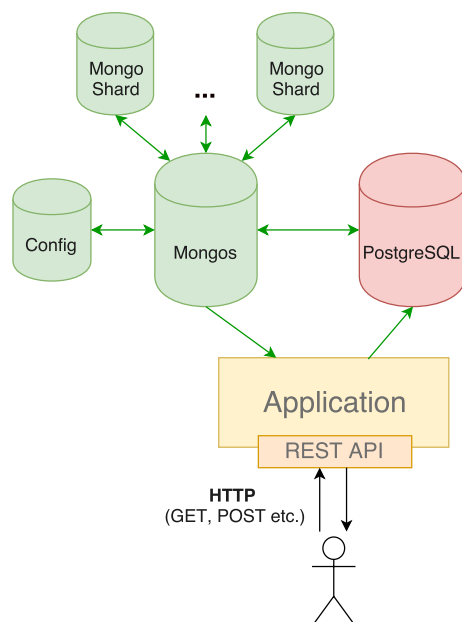
### 3.15.1. Idea shardingu

Sharding to metoda skalowania, która zakłada partycjonowanie danych. Oznacza to, że dane ulegają podzieleniu i przechowywane są na odrębnych instancjach serwerów bazodanowych.

Klaster baz MongoDB składa się z kilku następujących elementów [7]:

- **shard** - instancja procesu *mongod* (*mongo daemon*). *Mongod* to główny proces bazy MongoDB, czyli serwer baz danych. Jest on uruchamiany jako proces działający w tle *daemon*. Każdy shard może być pojedynczą lub kilkoma zreplikowanymi instancjami (ang. *replica set*). To właśnie na shardach są przechowywane dane podzielone na kawałki (ang. *chunk*),
- **mongos** - proces służący do kierowania ruchu do odpowiedniego sharda (ang. *routing*) oraz odpowiadający za równoważenie obciążenia (ang. *load balancing*). Wymagany jest przynajmniej jeden *Mongos*, jednak może ich istnieć kilka,
- **config server** - serwer konfiguracyjny przechowujący metadane dotyczące klastra, które opisują m.in. które porcje danych (*chunks*) znajdują się na każdym z shardów. Od wersji MongoDB 3.4 serwer konfiguracyjny musi być uruchomiony jako *replica set*.

Architekturę całego systemu po zastosowaniu skalowania za pomocą shardingu ilustruje rysunek 3.8



Rys. 3.8. Architektura systemu po zastosowaniu shardingu

### 3.15.2. Strategie shardingu

Strategie shardingu to kryterium, na podstawie którego dane będą trafiać do odpowiedniego sharda. W MongoDB istnieją dwie strategie shardingu. Każda z nich jako kryterium podziału danych wykorzystuje tzw. klucz shardingu (ang. *shard key*).

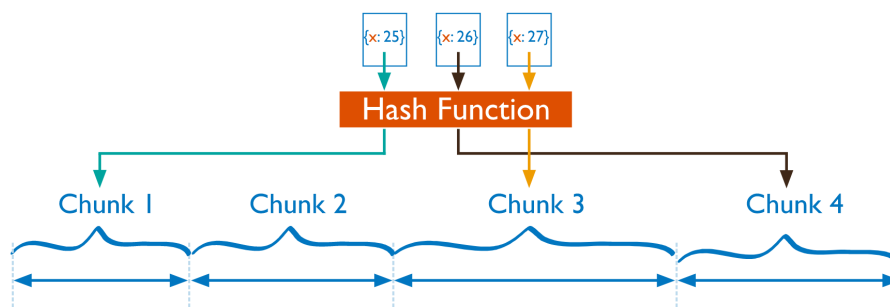
*Shard key* jest to pole posiadające index (ang. *indexed field*) lub kilka pól, na które założony jest wspólny index (ang. *indexed compound field*), które musi istnieć w każdym dokumencie danej kolekcji. Każdy shard posiada dynamicznie zdefiniowany zakres wartości kluczy shardingu, na podstawie których dane będą partycjonowane [1].

Przy wyborze klucza shardingu należy mieć na uwadze, że wartość pola, do którego przypisany jest klucz nie może się zmieniać oraz po podziale danych nie jest możliwa zmiana klucza na inny. Wybór odpowiedniego klucza shardingu powinien być dobrze przemyślany, gdyż ma on największy wpływ na wydajność całego klastra.

#### 3.15.2.1. Hashed sharding

Pierwszą strategią shardingu jest tzw. *hashed sharding*, czyli dystrybucja danych na podstawie wartości klucza poddanej działaniu funkcji skrótu (hashującej). MongoDB automatycznie przelicza hashe kluczy, więc nie musi być to zaimplementowane w aplikacji. Działanie tej strategii ilustruje rysunek 3.9.

*Hashed sharding* zapewnia jednolite rozłożenie danych pomiędzy wszystkimi shardami, gdyż nawet klucze o podobnych wartościach poddane działaniu funkcji skrótu będą miały znacznie inne wartości. Ta strategia shardingu ma zastosowanie, gdy kluczem jest pole, którego wartość zmienia się w przewidywalny sposób np. identyfikator obiektu tworzony na podstawie sekwencji. Zakładając, że pierwszy shard przechowywałby dane, których klucz znajduje się w przedziale  $\langle 1, 1000 \rangle$ , to pierwsze 1000 dokumentów trafiłoby do tej samej instancji bazy, a reszta pozostałaby



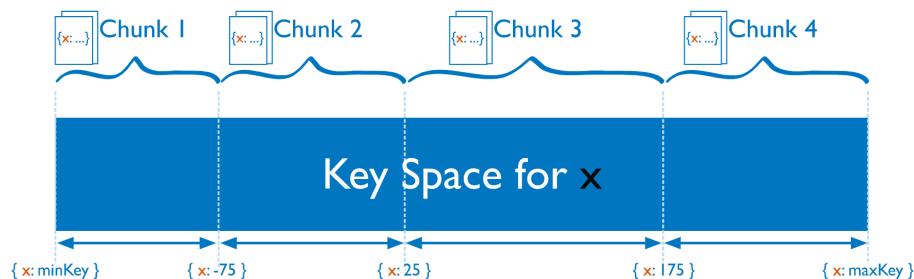
**Rys. 3.9.** Podział danych na podstawie skrótu wartości klucza [1]

pusta.

To podejście ma również swoje minusy. W przypadku, gdy istnieje przechowywanie potrzeba danych, których wartość klucza (niepoddana działaniu funkcji skrótu) znajduje się w jakimś konkretnym przedziale, wtedy zapytanie najprawdopodobniej będzie wysyłane do wielu instancji bazy danych. W tym przypadku lepszym podejściem jest wybór dystrybucji danych na podstawie zakresu, czyli tzw. *ranged sharding*.

### 3.15.2.2. Ranged sharding

Drugą strategią shardingu jest tzw. *ranged sharding*, czyli podział danych na podstawie zakresu, w którym znajduje się wartość klucza shardingu. Wynika z tego, że dokumenty, które posiadają podobne wartości klucza będą umieszczone w tej samej instancji bazy. Ideę tej strategii ilustruje rysunek 3.10.



**Rys. 3.10.** Podział danych na podstawie zakresu wartości klucza [1]

W tej strategii wybór klucza jest również bardzo ważny, gdyż nieodpowiedni klucz będzie skutkował w nierównomiernym podziale danych pomiędzy wszystkie shardy. Przynosi ona efekty w momencie, gdy istnieje potrzeba pobrania danych o kluczu z danego przedziału, gdyż istnieje wtedy bardzo duże prawdopodobieństwo, że wszystkie te dane znajdować się będą na tym samym shardzie. Pozwoli to uniknąć przeszukiwania wszystkich innych dokumentów.

### 3.15.3. Przygotowanie środowiska do shardingu

Do tej pory w całym systemie sklepu uruchomione były jedynie 3 kontenery: aplikacja, baza PostgreSQL oraz baza MongoDB. W celu konfiguracji skalowania za

pomocą shardingu potrzebne będą dodatkowe kontenery z instancjami bazy MongoDB, które będą służyły różnym celom.

Minimalne wymagania to:

- 1 kontener - mongos (*router, load ballancer*),
- 1 kontener - serwer konfiguracyjny,
- 2 kontenery - shardy.

Jest to minimalna ilość kontenerów potrzebnych do konfiguracji shardingu. Musi istnieć więcej niż jeden shard. W przeciwnym wypadku dane byłyby przetrzymywane w ten sam sposób jak w pojedynczej instancji bazy. Mongos oraz serwer konfiguracyjny nie byłyby wtedy potrzebne, gdyż byłoby wiadomo, że zapytania mogą być kierowane zawsze do konkretnej, pojedynczej instancji bazy, gdyż tam właśnie znajdują się wszystkie dane.

W celu utworzenia dodatkowych kontenerów zostanie również użyte narzędzie Docker Compose. Dodanie nowych kontenerów odbywa się poprzez dodanie kolejnych serwisów do pliku *docker-compose.yml*.

**Listing 3.31.** docker-compose.yml

```
1 version: '2'
2 services:
3   inz-app:
4     image: edu.uz/inz:0.0.1-SNAPSHOT
5     ports:
6       - "8080:8080"
7     depends_on:
8       - postgresql
9       - mongos
10
11  postgresql:
12    image: postgres
13    ports:
14      - "5432:5432"
15    environment:
16      - POSTGRES_PASSWORD=postgres
17      - POSTGRES_USER=postgres
18      - POSTGRES_DB=inz
19
20  mongos:
21    image: mongo
22    command: mongos --config /etc/mongos.yml --port 27017
23    volumes:
24      - './mongos.yml:/etc/mongos.yml'
25    ports:
26      - '27017:27017'
27
28  mongo-config:
29    image: mongo
30    command: mongod --config /etc/mongo-config.yml --port 27018
31    volumes:
32      - './mongo-config.yml:/etc/mongo-config.yml'
33    ports:
34      - '27018:27018'
35
36  mongo-shard-1:
37    image: mongo
38    command: mongod --config /etc/mongo-shard-1.yml --port 27021
39    volumes:
40      - './mongo-shard-1.yml:/etc/mongo-shard-1.yml'
41    ports:
42      - '27021:27021'
43
```



```
44 mongo-shard-2:
45   image: mongo
46   command: mongod --config /etc/mongo-shard-2.yml --port 27022
47   volumes:
48     - './mongo-shard-2.yml:/etc/mongo-shard-2.yml'
49   ports:
50     - '27022:27022'
```

---

Na listingu 3.31 została zaprezentowana kompletna konfiguracja podstawowych serwisów potrzebnych do wykonania shardingu. Serwis aplikacji oraz serwis bazy PostgreSQL zostały pozostawione bez zmian. Nowością są 4 dodatkowe serwisy tworzone z obrazu o nazwie *mongo*. Do każdego z nich został również przypisany osobny port.

Warto zauważyć, że *mongos* pełniący rolę routera, do którego będą kierowane wszystkie zapytania jest uruchomiony na porcie 27017. Jest to standardowy port, na którym uruchamiana jest baza MongoDB. Został on również użyty w poprzedniej konfiguracji z jednym kontenerem bazy (listing 3.30). Dzięki temu nie są wymagane żadne zmiany konfiguracyjne w aplikacji, gdyż sposób komunikacji pozostaje ten sam - wszystkie operacje odczytu kierowane są do jednej instancji bazy Mongo (*mongos*) jak na rysunku 3.8.

Każdy z serwisów Mongo musi zostać odpowiednio skonfigurowany, aby mógł pełnić wyznaczoną rolę. Odbywa się to poprzez uruchomienie procesu bazy danych w odpowiednim trybie używając załączonych do kontenerów plików konfiguracyjnych.

Wpis definiujący serwis *mongos* wygląda następująco:

---

**Listing 3.32.** Konfiguracja serwisu *mongos*

---

```
1 mongos:
2   image: mongo
3   command: mongos --config /etc/mongos.yml --port 27017
4   volumes:
5     - './mongos.yml:/etc/mongos.yml'
6   ports:
7     - '27017:27017'
```

---

Kluczowym elementem konfiguracji przedstawionej na listingu 3.32 są dwa atrybuty:

- **command** - definicja komend, które mają zostać uruchomione przy starcie kontenera. Zdefiniowana komenda w każdym serwisie Mongo służy do uruchomienia procesu bazy danych z odpowiednią konfiguracją pobraną z pliku (przełącznik *--config*) na odpowiednim porcie (*--port*),
- **volumes** - definicja folderów lub plików, które będą dołączone do kontenera. W przypadku serwisów Mongo są to pliki konfiguracyjne używane do uruchomienia ich w określonym trybie.

Pliki konfiguracji uruchamiania serwisów zostały przedstawione na listingach 3.33, 3.34 oraz 3.35

---

**Listing 3.33.** Konfiguracja uruchamiania *mongos*

---

```
1 sharding:
2   configDB: mongo-config/mongo-config:27018
```

---

W konfiguracji mongos należy jedynie wskazać adres serwera konfiguracyjnego. Definiuje się go w atrybucie *configDB* w formacie *<replSetName>/<host>:<port>*, gdzie *replSetName* jest nazwą *replica set* zdefiniowaną w pliku 3.34, *host* to adres kontenera (może być to jego nazwa) oraz *port* to oczywiście port, na którym uruchomiona jest usługa.

---

**Listing 3.34.** Konfiguracja uruchamiania serwera konfiguracyjnego

---

```
1 sharding:
2   clusterRole: configsvr
3 replication:
4   replSetName: mongo-config
```

---



---

**Listing 3.35.** Konfiguracja uruchamiania pojedynczego sharda

---

```
1 sharding:
2   clusterRole: shardsvr
3 replication:
4   replSetName: mongo-shard-1
```

---

W przypadku serwera konfiguracyjnego oraz poszczególnych shardów wystarczy zdefiniować rolę jaką pełnić będzie serwis (*configvr* lub *shardsvr*) oraz przypisać nazwę zbioru replik (*replica set*).

Po przygotowaniu wszystkich plików należy wykonać jedynie polecenie *docker-compose up*, aby uruchomić całe środowisko, na którym następnie będzie można skonfigurować sposób rozdzielania danych pomiędzy poszczególne shardy.

W celu zwiększenia liczby shardów należy jedynie dodać kolejne serwisy w pliku *docker-compose.yml* i uwzględnić je w późniejszej konfiguracji. W ten sposób można rozbudować klaster o dowolną liczbę shardów tym samym zwiększając jego wydajność.

### 3.15.4. Konfiguracja shardingu

Po uruchomieniu całego środowiska należy jedynie skonfigurować sharding przy użyciu wybranej strategii. W związku z tym, że konfiguracji należy dokonać na każdym z elementów klastra ręczna konfiguracja mogłaby być czasochłonna. W tym celu został przygotowany skrypt *bash*, który połączy się z każdym kontenerem i wykona na nim operacje konfiguracyjne zawarte w osobnych skryptach *JavaScript* obsługiwanych przez MongoDB.

Konfiguracja shardingu będzie się więc sprowadza do wywołania jednego skryptu *setup-sharding.sh* zamieszonego w listingu 3.36.

---

**Listing 3.36.** setup-sharding.sh

---

```
1 #!/bin/sh
2 host_ip=$(route get 1 | awk '{print $NF;exit}')
3
4 # Configure config server
5 mongo --host $host_ip --port 27018 < scripts/configure-config-server.js
6
7 # Configure shard-1 server
8 mongo --host $host_ip --port 27021 < scripts/configure-shard-1-server.js
9
10 # Configure shard-2 server
11 mongo --host $host_ip --port 27022 < scripts/configure-shard-2-server.js
12
13 # Configure mongos server
14 sleep 3
```

```
15 mongo --host $host_ip --port 27017 < scripts/configure-mongos-server.js
```

Skrypt *setup-sharding.sh* pobiera adres zewnętrzny adres IP komputera i zapisuje go w zmiennej *host\_ip*. Jest to adres, który służy jako brama dla kontenerów Docker. Nawiązując połączenie z tym adresem oraz odpowiednim portem uzyskamy połączenie z kontenerem.

W pierwszym kroku skrypt łączy się z serwerem konfiguracyjnym za pomocą konsolowego klienta MongoDB o nazwie *mongo*, a następnie w kontenerze wywołuje komendy konfiguracyjne zawarte w pliku *configure-config-server.js* zaprezentowanym na listingu 3.37.

**Listing 3.37.** *configure-config-server.js*

```
1 rs.initiate(  
2   {  
3     _id: "mongo-config",  
4     configsvr: true,  
5     members: [  
6       {_id: 0, host: "mongo-config:27018"}  
7     ]  
8   }  
9 );  
10  
11 exit;
```

Konfiguracja polega jedynie na zdefiniowaniu zestawu replik (*replica set*) serwera konfiguracyjnego. W tym wypadku serwer konfiguracyjny będzie uruchomiony jako pojedyncza instancja znajdująca się pod adresem *mongo-config:27018*. W celach testowych nie jest wymagana replikacja serwera, jednak w środowiskach produkcyjnych warto uruchamiać zreplikowane serwery w celu zwiększenia niezawodności.

Kolejny krok dotyczy konfiguracji poszczególnych shardów, która jest bardzo podobna do serwera konfiguracyjnego. Wystarczy jedynie zdefiniować zestaw replik dla każdego sharda. Przykład skryptu konfiguracyjnego dla pierwszego sharda znajduje się na listingu 3.38.

**Listing 3.38.** *configure-shard-1-server.js*

```
1 rs.initiate(  
2   {  
3     _id: "mongo-shard-1",  
4     members: [  
5       {_id: 0, host: "mongo-shard-1:27021"}  
6     ]  
7   }  
8 );  
9  
10 exit;
```

Najistotniejszy element konfiguracji odbywa się na serwerze *mongos*. To właśnie tam konfiguruje się na ilu shardach będą zapisywać się dane oraz jakie strategie shardowania oraz klucze zostaną użyte.

Skrypt z przykładowym sposobem konfiguracji shardingu na serwerze *mongos* znajduje się na listingu 3.39.

**Listing 3.39.** *configure-mongos-server.js*

```
1 sh.addShard("mongo-shard-1/mongo-shard-1:27021");  
2 sh.addShard("mongo-shard-2/mongo-shard-2:27022");  
3  
4 sh.enableSharding("inz");  
5 sh.shardCollection("inz.products", {_id: "hashed"});
```

```
6 sh.shardCollection("inz.customers", {_id: "hashed"});  
7  
8 exit;
```

W skrypcie *configure-mongos-server.js* zostało zaprezentowane w jaki sposób skonfigurować sharding zapisujący dane na dwóch shardach. Dodawanie poszczególnych shardów odbywa się za pomocą komendy *sh.addShard()*. Sharding musi zostać aktywowany dla konkretnej bazy danych poleceniem *sh.enableSharding()*. Po wykonaniu tych kroków możliwe jest ustawienie shardingu na konkretnej kolekcji używając określonego klucza i strategii hashowania. W tym przykładzie shardingowi zostały poddane kolekcje *products* oraz *customers* z bazy *inz*. W obu przypadkach kluczem jest pole *\_id*, a strategią shardingu jest dystrybucja na podstawie wartości hasha (*hashed sharding*).

## 3.16. Testy wydajnościowe

Ostatnim etapem jest przeprowadzenie testów wydajnościowych zaproponowanego rozwiązania za pomocą narzędzia *Apache JMeter*. Umożliwią one pokazanie w jaki sposób wzrasta wydajność aplikacji stosując zaprezentowane rozwiązanie. Testom zostanie poddana jedynie część aplikacji służąca do odczytu danych, gdyż zapis zawsze będzie wykonywany na bazie relacyjnej.

Testy wydajnościowe zostaną podzielone na kilka etapów. Przede wszystkim zostanie sprawdzona wydajność dotycząca wyszukiwania produktów oraz wyszukiwania użytkowników wraz ze wszystkimi ich danymi.

W przypadku produktów sytuacja jest bardzo prosta, gdyż jest to tylko pojedyncza tabela z danymi. Drugi przypadek testowy jest bardziej złożony, gdyż odczyt danych o użytkownikach wraz z ich danymi wymaga zebrania danych z kilku źródeł. W relacyjnej bazie danych są to tabele przechowujące informacje o użytkowniku, jego zamówieniach, pozycjach zamówienia oraz produktach. Wymaga to wykonanie większej ilości operacji wyszukiwania oraz łączenia danych. W tej sytuacji baza NoSQL powinna osiągnąć znacznie większą wydajność. Dodatkowo zostanie pokazane jak skalowanie bazy NoSQL dodatkowo wpływa na wydajność całej aplikacji.

### 3.16.1. Plan testów

Testy zostały przeprowadzone w następujących etapach:

- Zapis przykładowych danych do bazy relacyjnej - dzięki synchronizacji dane te trafiają również do bazy NoSQL,
- Wykonanie testów wydajnościowych na aplikacji uruchomionej w trybie odczytu z bazy relacyjnej (przełącznik *read.source=rdbms*),
- Przełączenie aplikacji w tryb odczytu z bazy NoSQL (*read.source=nosql*) przy uruchomionej jednej instancji bazy MongoDB i wykonanie tego samego zestawu testów.

- Uruchomienie klastra MongoDB składającego się z routera, serwera konfiguracyjnego oraz 4 instancji przechowujących dane i ponowne przeprowadzenie testów wydajnościowych,
- Porównanie otrzymanych wyników.

### 3.16.2. Przygotowanie danych

W celu przeprowadzenia testów wydajnościowych obie bazy powinny posiadać znaczącą ilość danych. Został stworzony skrypt, który umożliwił dodanie określonej ilości zróżnicowanych danych.

Dane, którymi została zainicjalizowana baza danych to:

- **100 000** produktów,
- **100 000** użytkowników, z których każdy posiada 1-3 zamówień. W każdym z zamówień znajduje się 1-3 produktów.

Podczas zapisu danych do bazy wszystkie identyfikatory zostały zapisane do dwóch osobnych plików tekstowych - odpowiednio dla produktów i użytkowników. Pliki te posłużą jako dane testowe, aby za każdym razem poszukiwany był inny wpis. Jest to o wiele lepsze rozwiązanie, niż wyszukiwanie cały czas jednego wpisu, gdyż baza danych może podjąć próbę optymalizacji takiego zapytania, co dałoby niemiarodajne wyniki.

Po dodaniu danych do bazy mają one następujące rozmiary:

- **PostgreSQL** - 107 MB
- **MongoDB** - 153,7 MB

Można zauważyć, iż na skutek duplikacji danych oraz narzutu pamięciowego z uwagi na przechowywanie danych w dokumentach JSON baza danych MongoDB zajmuje około 50 więcej przestrzeni dyskowej.

### 3.16.3. Testy odczytu z bazy relacyjnej

W początkowym etapie przetestowana została wydajność serwisu, który wszystkie operacje odczytu wykonuje na relacyjnej bazie danych PostgreSQL. Wyniki wydajności wyszukiwania produktów oraz użytkowników zaprezentowane zostały w odpowiednio w tabelach 3.1 oraz 3.2.

**Tab. 3.1.** Czasy odpowiedzi bazy relacyjnej - wyszukiwanie produktu

Ilość wątków	Ilość zapytań	Czas odpowiedzi (ms)		
		Średni	Min	Max
10	100 000	2	0	23
10	100 000	2	0	43
20	100 000	4	1	41
20	100 000	4	0	42
40	100 000	8	0	108
40	100 000	8	0	61
80	100 000	17	0	264
80	100 000	15	0	169

Średnie czasy odpowiedzi zapytań o proste dane (produkty - tab. 3.1) są satysfakcjonujące nawet przy dużym obciążeniu, gdyż wynoszą maksymalnie 15 ms.

**Tab. 3.2.** Czasy odpowiedzi bazy relacyjnej - wyszukiwanie użytkownika

Ilość wątków	Ilość zapytań	Czas odpowiedzi (ms)		
		Średni	Min	Max
10	50 000	236	34	694
10	50 000	249	31	736
20	50 000	377	74	825
20	50 000	382	72	913
40	100 000	802	27	2588
40	100 000	777	17	1540
80	100 000	1459	244	2890
80	100 000	1484	341	3624

Można jednak zauważyć, że w przypadku zapytań o bardziej złożone dane (informacje o użytkowniku - tab. 3.2) średnie czasy odpowiedzi znacznie się zwiększają. Dochodzą nawet do 1,5 sekundy przy większym obciążeniu.

#### 3.16.4. Testy odczytu z bazy NoSQL

Następnym etapem jest przetestowanie wydajności serwisu, który wszystkie operacje odczytu wykonuje na nierelacyjnej bazie danych MongoDB. Wyniki wydajności wyszukiwania produktów oraz użytkowników zaprezentowane zostały w odpowiednio w tabelach 3.3 oraz 3.4.

**Tab. 3.3.** Czasy odpowiedzi bazy NoSQL - wyszukiwanie produktu

Ilość wątków	Ilość zapytań	Czas odpowiedzi (ms)		
		Średni	Min	Max
10	100 000	1	0	19
10	100 000	1	0	12
20	100 000	3	0	30
20	100 000	3	0	23
40	100 000	6	0	90
40	100 000	5	0	100
80	100 000	11	0	226
80	100 000	11	0	355

Wyszukiwanie produktów w bazie NoSQL jest nieznacznie bardziej wydajne, niż w przypadku bazy relacyjnej. W tym przypadku różnice nie są zbyt duże, gdyż aplikacja używając bazy relacyjnej również zwracała odpowiedzi w akceptowalnym czasie.

**Tab. 3.4.** Czasy odpowiedzi bazy NoSQL - wyszukiwanie użytkownika

Ilość wątków	Ilość zapytań	Czas odpowiedzi (ms)		
		Średni	Min	Max
10	100 000	1	0	23
10	100 000	2	0	23
20	100 000	4	0	58
20	100 000	3	0	32
40	100 000	7	0	148
40	100 000	7	0	130
80	100 000	14	0	268
80	100 000	15	0	356

Dużą różnicę w średnich czasach odpowiedzi można zauważyć, gdy porówna się wyniki wyszukiwania informacji o użytkowniku. Aplikacja używająca bazy NoSQL przy największym obciążeniu zwracała odpowiedź na zapytanie średnio w 15 ms. Jest to blisko 100 razy szybciej, niż w przypadku bazy relacyjnej.

### 3.16.5. Testy odczytu z klastra baz NoSQL

Wyniki uzyskane podczas testowania aplikacji używającej bazę danych NoSQL są satysfakcjonujące. Jednak w przypadku, gdy dane urosły by do ogromnych rozmiarów oraz obciążenie aplikacji znacznie wzrosło, nawet baza danych NoSQL mogłaby napotkać problemy wydajnościowe.

Aplikacja została skonfigurowana tak, aby korzystała z klastra baz MongoDB składającego się z 4 shardów. Otrzymane wyniki prezentują tabele 3.5 oraz 3.6.

**Tab. 3.5.** Czasy odpowiedzi klastra baz NoSQL - wyszukiwanie produktu

Ilość wątków	Ilość zapytań	Czas odpowiedzi (ms)		
		Średni	Min	Max
40	100 000	1	0	15
40	100 000	1	0	21
80	100 000	1	0	63
80	100 000	1	0	55

**Tab. 3.6.** Czasy odpowiedzi klastra baz NoSQL - wyszukiwanie produktu

Ilość wątków	Ilość zapytań	Czas odpowiedzi (ms)		
		Średni	Min	Max
40	100 000	2	0	53
40	100 000	2	0	51
80	100 000	3	0	183
80	100 000	4	0	221

Zgodnie z oczekiwaniami zastosowanie partycjonowania danych w klastrze baz NoSQL znacznie podniosło wydajność aplikacji. Przy najwyższym testowanym obciążeniu aplikacja wyszukiwała informacji o użytkowniku średnio w 3 ms, co jest bardzo dobrym wynikiem przy takiej ilości danych.

### 3.16.6. Podsumowanie

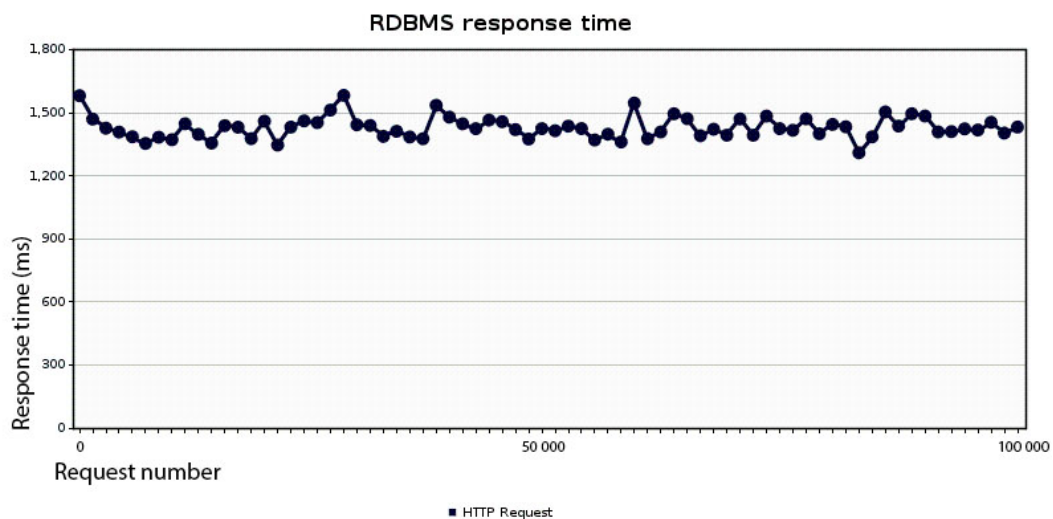
Otrzymane wyniki wskazują wyraźnie, że relacyjne bazy danych przechowujące dużą ilość informacji napotykały na pewne problemy wydajnościowe. Stosując architekturę CQRS możliwe jest wykorzystanie bazy NoSQL do wykonywania odczytów danych, co znacznie poprawia wydajność całej aplikacji. Dodatkowo wykorzystując mechanizm skalowania bazy MongoDB o nazwie *sharding* można dodatkowo zwiększyć wydajność wykonywanych operacji odczytu.

Porównanie wydajności pod względem średnich czasów odpowiedzi przy najwyższym testowanym obciążeniu (80 wątków) dla RDBMS, NoSQL oraz klastra NoSQL zostały zaprezentowane odpowiednio na rysunkach 3.11, 3.12 oraz 3.13.

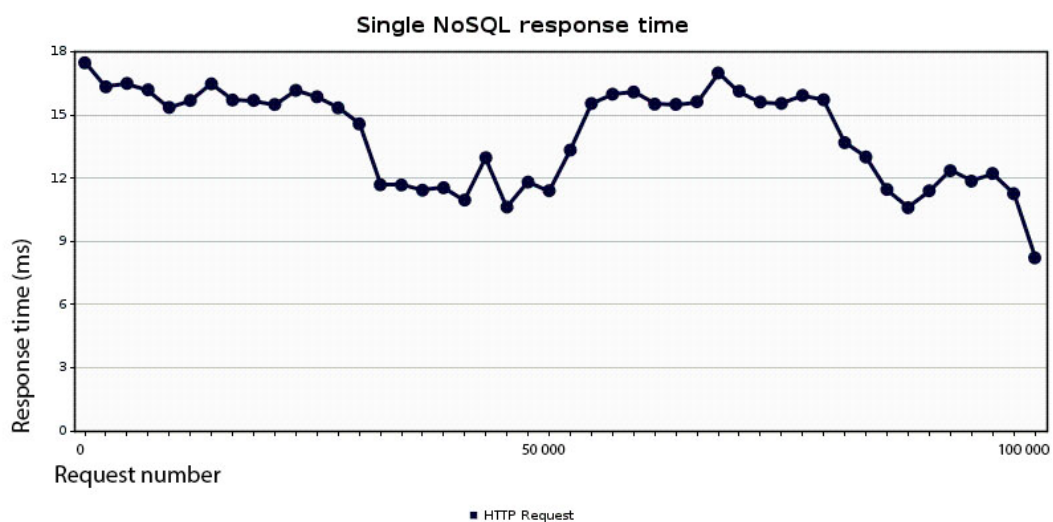
Na zaprezentowanych wykresach widać, że średni czas odpowiedzi bazy relacyjnej wynosi ok. **1500 ms** (1,5 s). Używając bazy NoSQL do odczytu danych można zwiększyć wydajność 100-krotnie, gdyż średni czas odpowiedzi w takiej konfiguracji wynosi jedynie **15 ms**. Zastosowanie klastra baz z 4 instancjami shardów pozwala na dodatkowe prawie 5-krotne zwiększenie wydajności względem pojedynczego serwera bazy NoSQL - średni czas odpowiedzi ok. **3 ms**, czyli jest to ok. 500 razy szybciej, niż baza relacyjna.

Warto jednak zauważyć, że wyniki uzyskane przy pojedynczej instancji bazy NoSQL są już zadowalające dla takiej ilości danych oraz przy założonym obciążeniu. Zastosowanie klastra w tym wypadku co prawda zwiększa wydajność aplikacji, jednak wprowadza dodatkowe problemy administracyjne związane z zarządzaniem wieloma kontenerami. Decyzja o użyciu klastra baz danych musi być więc odpowiednio uzasadniona oraz przemyślana. Należy ją stosować jedynie w przypadku,

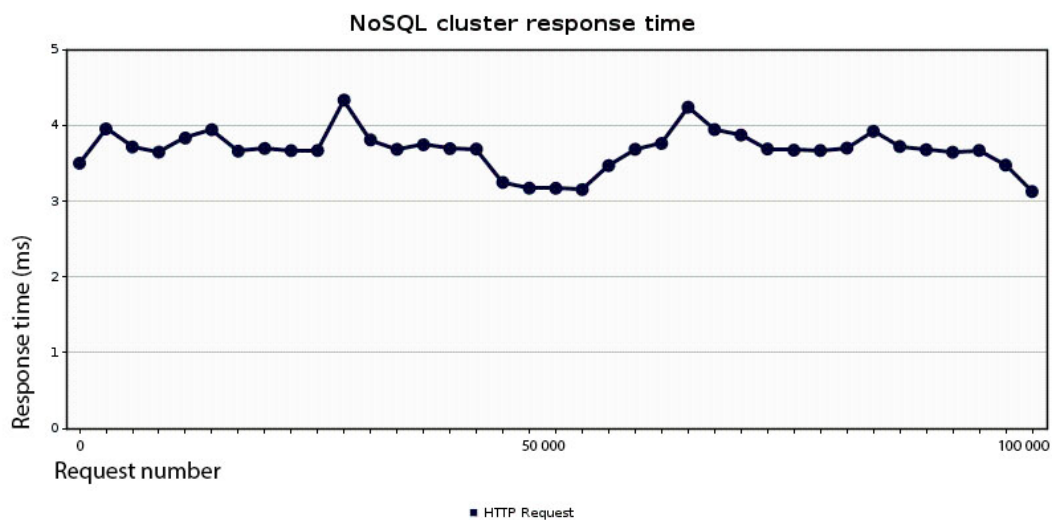




Rys. 3.11. Wydajność relacyjnej bazy danych



Rys. 3.12. Wydajność pojedynczej bazy NoSQL



Rys. 3.13. Wydajność klastra baz NoSQL

gdy zastosowanie pojedynczej instancji bazy danych NoSQL staje się niewydatne - o wiele większa ilość danych i znacznie większe obciążenie.

# Rozdział 4

## Zakończenie

Z pewnością można stwierdzić, że bazy danych typu NoSQL znajdują zastosowanie we współczesnych aplikacjach. Przykładem jest przedstawione użycie tych baz wraz z bazami relacyjnymi w obrębie jednej aplikacji stworzonej zgodnie z architekturą CQRS. Skutkiem tego podejścia jest znaczące zwiększenie wydajności aplikacji, której czasy odpowiedzi przy większej ilości danych oraz większym obciążeniu stają się nieakceptowalne.

Tradycyjne bazy danych są dalej bardzo popularne i bardzo często wykorzystywane, jednak nie są one już jedynym możliwym wyborem, jeżeli chodzi o decyzję odnośnie tego jakiej bazy danych użyć w projekcie. Omawiane bazy NoSQL przede wszystkim oferują zupełnie inne właściwości w porównaniu do tradycyjnych baz relacyjnych. Można powiedzieć, że jest to narzędzie z tej samej kategorii, jednak służące do trochę innych celów. Dla każdego systemu z osobna należy najpierw przeanalizować problem, który ma rozwiązywać, a następnie dobrać odpowiedni typ bazy danych.

Na podstawie tych spostrzeżeń można dojść do wniosku, że wybór jednego typu bazy danych dla każdego projektu nie jest prawidłowym podejściem. Nie można bowiem stwierdzić, że jeden typ baz danych jest pod każdym względem lepszy od drugiego. Bazy NoSQL nie są więc rozwiązaniem każdego problemu. Są jednak dobrą alternatywą dla tradycyjnych baz relacyjnych.

# Dodatek A

## Płyta DVD

Do tekstu pracy załączona została płyta DVD z następującą zawartością:

- plik **/praca-dyplomowa.pdf** - tekst pracy dyplomowej,
- katalog **/projekt/** - zawiera wszystkie pliki wykonanego projektu,
- katalog **/oprogramowanie/** - zawiera oprogramowanie wymagane do uruchomienia projektu. W szczególności są to:
  - katalog **/oprogramowanie/apache-jmeter/** - program Apache JMeter do wykonywania testów wydajnościowych,
  - katalog **/oprogramowanie/docker/** - program Docker do tworzenia kontenerów,
  - katalog **/oprogramowanie/docker-compose/** - program Docker Compose do automatyzacji uruchamiania wielu kontenerów,
  - katalog **/oprogramowanie/intellij-idea/** - środowisko programistyczne IntelliJ IDEA umożliwiające otworzenie projektu.

# Bibliografia

- [1] MongoDB. *Shard keys*.  
<https://docs.mongodb.com/manual/core/sharding-shard-key/>.
- [2] Martin Fowler Pramod J. Sadalage. *MySQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Pearson Education, New Yersey, 2013.
- [3] Christof Strauch. *NoSQL Databases*, Czerwiec 2010.  
<http://www.christof-strauch.de/nosql dbs.pdf>.
- [4] Johannes Zollmann. *NoSQL Databases*. [https://sewiki.iai.uni-bonn.de/\\_media/teaching/labs/xp/2012b/seminar/10-nosql.pdf](https://sewiki.iai.uni-bonn.de/_media/teaching/labs/xp/2012b/seminar/10-nosql.pdf).
- [5] Greg Young. *CQRS Documents*.  
[https://cQRS.files.wordpress.com/2010/11/cQRS\\_documents.pdf](https://cQRS.files.wordpress.com/2010/11/cQRS_documents.pdf).
- [6] Dariusz Pawlukiewicz. *CQRS i Event Sourcing - czyli łatwa droga do skalowalności naszych systemów*.  
[https://bulldogjob.pl/articles/122-cQRS-i-event-sourcing-czyli-latwa-droga-do-skalowalnosci-naszch-systemow\\_](https://bulldogjob.pl/articles/122-cQRS-i-event-sourcing-czyli-latwa-droga-do-skalowalnosci-naszch-systemow_).
- [7] MongoDB. *Sharding*. <https://docs.mongodb.com/manual/sharding/>.
- [8] Eric Evans. *NoSQL: What's in a name?*, Październik 2009.  
[http://blog.sym-link.com/2009/10/30/nosql\\_whats\\_in\\_a\\_name.html](http://blog.sym-link.com/2009/10/30/nosql_whats_in_a_name.html).
- [9] Eric Evans. *Domain-Driven Design*. Addison-Wesley, 2004.
- [10] Computerworld. *Słownik pojęć z zakresu IT*. <http://www.computerworld.pl/sownik/termin/41175/impedance.mismatch.html>.
- [11] Wikipedia. *ACID*. <https://en.wikipedia.org/wiki/ACID>.
- [12] Eric Brewer. *Towards Robust Distributed Systems*. <https://people.eecs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf>.
- [13] Wikipedia. *Graph*.  
[https://en.wikipedia.org/wiki/Graph\\_\(discrete\\_mathematics\)](https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)).
- [14] PYPL. *PYPL PopularitY of Programming Language*.  
<http://pypl.github.io/PYPL.html>.
- [15] Wojciech Paciorkowski. *Docker dla programistów, co to jest?*  
<https://sii.pl/blog/docker-dla-programistow-co-to-jest/>.

- [16] Baeldung. *Java in 2017 Survey*. <http://www.baeldung.com/java-in-2017>.
- [17] Restfulapi. *What is REST?* <https://restfulapi.net/>.
- [18] PostgreSQL. *UUID*.  
<https://www.postgresql.org/docs/9.2/static/datatype-uuid.html>.