



**POLSKO-JAPOŃSKA AKADEMIA
TECHNIK KOMPUTEROWYCH**

WYDZIAŁ INFORMATYKI

KATEDRA INŻYNIERII OPROGRAMOWANIA

**INŻYNIERIA OPROGRAMOWANIA, PROCESÓW BIZNESOWYCH
I BAZ DANYCH**

Adam Żurada

Nr albumu 16388

**Wpływ zastosowania Event Sourcing'u i
architektury mikroserwisowej na jakość
oprogramowania**

Praca magisterska napisana pod

kierunkiem:

dr hab. Piotr Habela

Warszawa, styczeń 2019

Streszczenie

Więszkość systemów informatycznych opartych jest na operacjach CRUD, czyli dodawaniu, odczytywaniu, aktualizacji i usuwaniu informacji w oparciu o bazę danych.

Aplikacje oparte na modelu anemicznym to takie, które używają operacji CRUD i nie są niczym innym jak jedynie oprawą graficzną bazy danych. Użytkownik owego system, władający wiedzą z zakresu baz danych, potrafiłby bezpośrednio na poziomie połączenia z bazą danych zmieniać stan danych aplikacji adekwatnie do potrzeb logiki biznesowej. Z aplikacjami opartymi na modelu anemicznym związany jest pewien problem - jakakolwiek aktualizacja czy usunięcie danych zmienia dane w sposób trwały i tracimy informację o wcześniejszej wersji danej sprzed aktualizacji czy usunięcia. Posiadanie całej historii danych w aplikacji może pozwolić nam na odtworzenie wszystkich operacji od początku, cofnięcie się w czasie do danego punktu, czy też szybkie rozpoznanie sytuacji niebezpiecznych i ochronę aplikacji przed atakami osób trzecich. Event Sourcing jest podejściem, który proponuje rozwiązanie powyższych problemów, gdzie każda operacja jest osobnym wydarzeniem w aplikacji. Jest to swoisty dziennik zachowujący całą historię danych w sposób trwały. CQRS, czyli sposób na logiczną segregację systemu na część odpowiedzialną za zmiany w systemie i część jedynie czytającą stan danych, jest bardzo często wykorzystywany w połączeniu z Event Sourcing'iem.

W ramach pracy dyplomowej została zaimplementowana w dwóch wersjach domena sklepu internetowego z kartami podarunkowymi, poprzez użycie modelu monolitycznego z anemicznym modelem danych i Event Sourcing'iem. Analiza na praktycznym przykładzie opisuje wady i zalety obu podejść.

Event Sourcing jest bardzo nowoczesnym podejściem, jeżeli użyje go się razem z CQRS. Aplikacje oparte na nim implikują oparcie rozwiązania o architekturę mikroserwisową, która daje większą elastyczność, skalowalność i łatwość w rozszerzaniu funkcjonalności, niż aplikacje monolityczne. Jednakże zastosowanie Event Sourcing'u wymaga o wiele więcej wkładu pracy i czasu, bo oparcie systemu o wydarzenia pociąga za sobą inne komplikacje i utrudnienia występujące w asynchronicznej naturze wydarzeń.

Słowa kluczowe: aplikacje monolityczne, architektura oprogramowania, Event Sourcing, jakość oprogramowania, mikroserwisy

Spis treści

| | | |
|----------|---|----------|
| 1 | Wprowadzenie | 1 |
| 1.1 | Cel pracy | 3 |
| 1.2 | Rozwiązania przyjęte w pracy | 3 |
| 1.3 | Rezultaty pracy | 3 |
| 1.4 | Organizacja pracy | 4 |
| 2 | Prezentacja omawianych technologii | 5 |
| 2.1 | Aplikacja monolityczna | 5 |
| 2.2 | Aplikacja mikroservisowa | 6 |
| 2.3 | Anemic Model | 8 |
| 2.4 | Event Sourcing | 9 |
| 2.5 | CQRS | 11 |
| 2.6 | Java | 13 |
| 2.7 | REST | 14 |
| 2.8 | API | 14 |
| 2.9 | HTTP | 15 |
| 2.10 | Spring Framework | 15 |
| 2.10.1 | Moduły Springa użyte w pracy | 15 |
| 2.11 | Hibernate | 16 |
| 2.12 | Apache Kafka | 17 |
| 2.12.1 | Architektura Kafki | 18 |
| 2.12.2 | Kafka Core APIs | 19 |
| 2.13 | Docker | 25 |
| 2.13.1 | Architektura Docker'owa | 25 |
| 2.14 | Kafka Lenses Box | 28 |
| 2.15 | Swagger | 28 |
| 2.16 | Immutables | 29 |
| 2.17 | Apache Maven | 29 |

| | | |
|----------|---|-----------|
| 3 | Implementacja aplikacji | 30 |
| 3.1 | Opis domeny | 30 |
| 3.2 | Implementacja wersji opartej na monolicie z anemicznym antyw- zorcem | 31 |
| 3.2.1 | Diagram bazy danych | 32 |
| 3.2.2 | Organizacja kodu | 33 |
| 3.2.3 | Repozytoria encji | 35 |
| 3.2.4 | Połączenie z bazą danych | 36 |
| 3.2.5 | REST'owe endpointy | 37 |
| 3.2.6 | Uruchomienie aplikacji | 41 |
| 3.3 | Implementacja wersji opartej na Event Sourcing'u | 41 |
| 3.3.1 | Event Sourcing w Apache Kafce | 42 |
| 3.3.2 | Podział na mikroserwisy wg wzorca CQRS | 44 |
| 3.3.3 | Połączenie z Apache Kafka | 45 |
| 3.4 | Struktura aplikacji apiCommand | 45 |
| 3.4.1 | Organizacja kodu | 46 |
| 3.4.2 | Walidacja zapytań | 47 |
| 3.4.3 | Sposób użycia Kafka Producer API | 49 |
| 3.4.4 | REST'owe Endpointy | 50 |
| 3.4.5 | Uruchomienie aplikacji | 52 |
| 3.5 | Struktura aplikacji apiQuery | 53 |
| 3.5.1 | Organizacja kodu | 54 |
| 3.5.2 | REST'owe endpointy | 55 |
| 3.5.3 | Uruchomienie aplikacji | 57 |
| 3.6 | Struktura aplikacji orderComponent | 58 |
| 3.6.1 | Organizacja procesorów strumieni poprzez Streams API | 59 |
| 3.6.2 | Organizacja kodu | 62 |
| 3.6.3 | Uruchomienie aplikacji | 63 |
| 4 | Wnioski w odniesieniu do stworzonych aplikacji | 65 |
| 4.1 | Wybór podejścia | 66 |
| 4.2 | Poziom skomplikowania kodu a domeny | 66 |
| 4.2.1 | Wymagania systemowe | 67 |
| 4.2.2 | Wymagania biznesowe | 68 |
| 4.2.3 | Wiedza programistyczna | 68 |
| 4.3 | Jakość oprogramowania | 69 |
| 4.3.1 | Funkcjonalna stabilność | 69 |
| 4.3.2 | Niezawodność | 70 |
| 4.3.3 | Wydajność | 71 |
| 4.3.4 | Użyteczność | 71 |
| 4.3.5 | Bezpieczeństwo | 71 |

| | | |
|----------|--------------------------|-----------|
| 4.3.6 | Kompatybilność | 72 |
| 4.3.7 | Utrzymywalność | 73 |
| 4.3.8 | Przenaszalność | 74 |
| 5 | Podsumowanie | 75 |

Rozdział 1

Wprowadzenie

Każdy projekt oparty o system informatyczny przechodzi przez fazę dokonania decyzji o wyborze architektury aplikacji. Do tej pory większość aplikacji implementowana była o strukturę monolityczną, czyli taką gdzie konstrukcja oprogramowania oparta jest na jednym, wspólnym i współdzielonym modelu danych. Tworząc aplikacje w ten sposób programiści i architekci często wpadają w pułapkę antywzorca projektowego, jakim jest anemiczny model dziedziny.

„W tym przypadku model dziedziny składa się z klas z atrybutami bez metod, nie jest więc obiektowy. Logika biznesowa przeniesiona jest do innych klas, które transformują klasy dziedziny zmieniając ich stan (...) Znaczna część metodyk tworzenia oprogramowania w Javie (w tym EJB) operuje na takim modelu. Duża część projektantów przenosi też swoje przyzwyczajenia z modelowania baz danych modelując system w ten sposób.”[2]

Aplikacja monolityczna nie tylko jest trudna do implementacji, ale także problematyczna przy dodawaniu nowych funkcjonalności. Tego typu problemy stara rozwiązać się architektura mikroservisowa, coraz częściej wybierana przez architektów i programistów, zwana często potocznie jako mikroservisy czy aplikacja w chmurze.

Mikroservisy to bardzo wygodna architektura. Wymaga całkowitej rezygnacji

z jednego, relacyjnego systemu danych dla dużej aplikacji, dzięki czemu możliwe jest niezależne, iteracyjno-przyrostowe (zwinne) implementowanie kolejnych usług. Ogólna idea to budowanie aplikacji tak, by każdy przypadek użycia stanowił praktycznie odrębny mały komponent. W efekcie mamy dużą swobodę zarządzania kolejnością ich implementacji a lokalne modyfikacje nie przenoszą się na resztę systemu. Ewentualne współdzielone komponenty to wyłącznie elementy logiki biznesowej, co nie ogranicza zbyt mocno kolejności implementowanych i wdrażanych usług aplikacyjnych. [15]

Podział aplikacji na mikroserwisy nie rozwiązuje jednakże problemów implementacyjnych. Jakakolwiek aktualizacja czy usunięcie danych zmienia stan aplikacji w sposób trwały, tym samym tracąc informację o wcześniejszego stanu sprzed aktualizacji czy usunięcia. Posiadanie całej historii danych w aplikacji, od samego początku, może pozwolić nam na odtworzenie każdej operacji użytkownika, cofnięcie się do danego punktu w czasie, czy też szybkie rozpoznanie ataku na aplikację przez osoby trzecie.

Event Sourcing jest podejściem, który proponuje rozwiązanie powyższych problemów. W tym modelu każda operacja jest osobnym wydarzeniem w aplikacji, które zostają trwale zapisane w bazie danych.

W ramach pracy dyplomowej zostały zaimplementowane i porównane dwie wersje aplikacji o tej samej funkcjonalności, oparte na modelu anemicznym i na Event Sourcing'u.

Jeżeli zasadnym jest użycie Event Sourcing'u ze względu na domenę problemową i zasoby systemowe, to zastosowanie tego podejścia, połączonego z użyciem architektury mikroservisowej zapewnia poprawę jakości oprogramowania w rozumieniu normy ISO/IEC 25010:2011.

1.1 Cel pracy

Celem niniejszej pracy było zaprezentowanie monolitycznej aplikacji z modelem anemicznym, a także Event Source'ingu jako rozwiązań architektonicznych dla aplikacji internetowych. Analiza i porównanie obu sposobów implementacji posłużyło do wyciągnięcia wniosków i pomocy w wyborze dobrego podejścia dla danej dziedziny problemowej. Porównanie opiera się na ocenie jakości oprogramowania. W celu zbadania stawianego przed pracą problemu powstał prototyp aplikacji w dwóch wersjach opartych o oba przedstawione podejścia. Jest to system aukcyjny e-commerce, gdzie można wystawiać i sprzedawać karty podarunkowe.

1.2 Rozwiązania przyjęte w pracy

Prototypy aplikacji zostały oparte na Javie w wersji 10, ze względu na zawodowe doświadczenie autora pracy w pisaniu aplikacji internetowych właśnie w tej technologii.

Wersja aplikacji monolitycznej z modelem anemicznym opiera się na Spring Framework i relacyjnej bazie SQL i Hibernate.

Wersja aplikacji oparta o Event Sourcing wykorzystuje również Spring Framework, ale do komunikacji między mikroserwisami używa Apache Kafka.

1.3 Rezultaty pracy

Głównym wynikiem pracy jest własna analiza i porównanie poszczególnych podejść do architektury systemu informatycznego na przykładzie implementacji aplikacji internetowej. W ramach tworzenia pracy powstały dwie odrębne aplikacje, których porównanie ułatwia decyzję, które podejście jest adekwatne w danej sytuacji, a także ocenia jakość oprogramowania.

1.4 Organizacja pracy

Pierwszy etap dokładnie opisuje każdą z wymienionych wyżej technologii, co stanowi wstęp teoretyczny przybliżający stos technologiczny.

Kolejny etap poświęcony jest opisowi funkcjonalności aplikacji-prototypu, które muszą być uwzględnione, a także zawiera diagram bazy danych.

Następnie autor pracy opisyje dokładną implementację prototypów, co stanowi pewnego rodzaju dokumentację obu aplikacji.

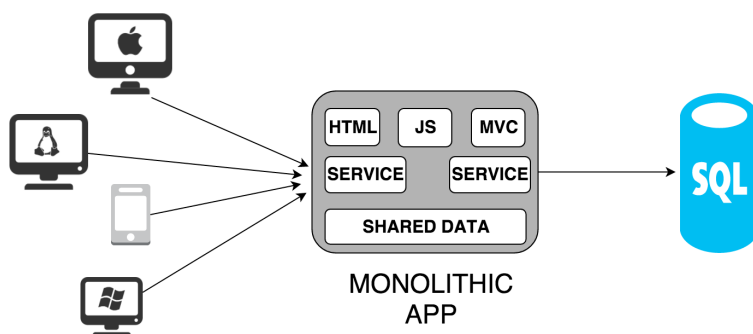
Ostatnim etapem jest szczegółowa analiza i porównanie pomiędzy monolitem z modelem anemicznym a Event Sourcing’iem. Przedstawiona jest weryfikacja jakości oprogramowania obu implementacji.

Rozdział 2

Przedstawienie omawianych technologii

Ten rozdział poświęcony jest omówieniu używanych technologii i pojęć w pozostałych rozdziałach.

2.1 Aplikacja monolityczna



Rysunek 2.1: Architektura monolityczna. Źródło: [8]

W inżynierii oprogramowania monolityczna aplikacja to taka, która została zaprojektowana bez modułowości.[22]

Aplikacja monolityczna to sposób budowania systemów informatycznych, w

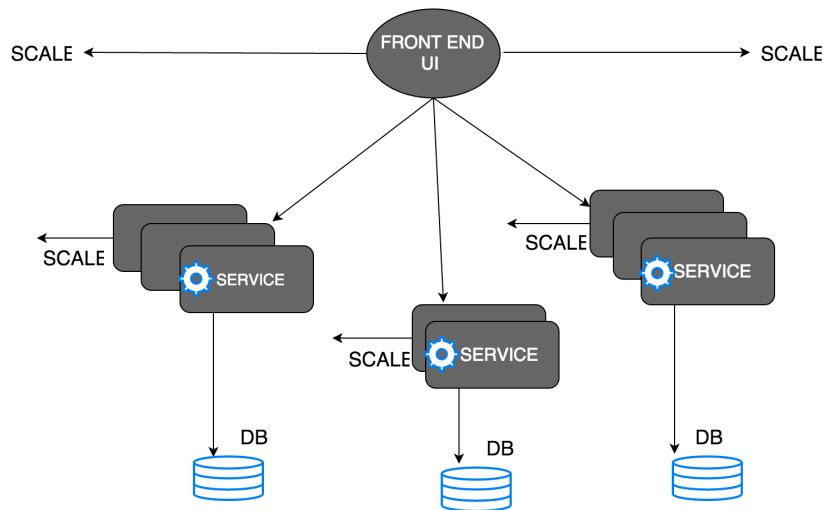
którym wszystkie komponenty działają w ramach jednej aplikacji. Dawniej popularny ze względu na brak realnych alternatyw — aplikacje działały na komputerach typu mainframe, a chmura obliczeniowa była wymysłem futurystów.

Takie podejście do tworzenia aplikacji ma zalety m.in:

- Nie dotyczą jej problemy związane z komunikacją sieciową.
- Wymiana danych pomiędzy modułami może odbywać się w pamięci.
- Nie ma problemów ze spójnością danych.

Minusem jest bardzo ograniczona możliwość skalowania — tego rodzaju aplikacje często bardzo trudno przenieść na wiele równoległych działających komputerów.[12]

2.2 Aplikacja mikroserwisowa



Rysunek 2.2: Architektura Mikroserwisowa. Źródło: [8]

Mikroserwisy to styl programowania, w którym zespolone aplikacje składają się z wielu niezależnych serwisów. Serwisy są rozdzielone i skupione na spełnianiu niewielkich zadań – mogą być one oddzielnie rozwijane, testowane, budowane

i lokowane. Koncepcja mikrousług wywodzi się z SOA – architektury zorientowanej na usługi, która swoje triumfy święciła wiele lat temu. Teraz powraca w ulepszonej postaci właśnie jako mikrousługi.[7]

Każdy mikroserwis jest stosunkowo mały, co daje wiele zalet, ale również i wad.

Do zalet [13] należą:

- Kod jest łatwiejszy do zrozumienia przez programistę.
- Stanowi mniejsze obciążenie dla zintegrowanego środowiska programistycznego.
- Pozwala na szybszy start kontenera aplikacji.
- Każdy mikroserwis można dostarczać niezależnie od innych serwisów, z różnym cyklem wydawniczym włącznie.
- Każdy mikroserwis można wytwarzać w innym zespole w innej lokalizacji nawet przy użyciu innych narzędzi, technologii i języków.
- Każdy mikroserwis działa jako odrębna aplikacja, proces, czy instancja maszyny wirtualnej, więc problemy wewnętrzne w samym kontenerze nie wpływają na działanie innych serwisów.

Do wad [13] należą:

- Komplikuje się proces wytwarzania systemu, system staje się zbiorem małych systemów, które trzeba zgrać i zsynchronizować.
- Testowanie, choć jednostkowo prostsze w układzie testowania całego złożonego systemu, jest wyraźnie trudniejsze ze znaczną rozbudową testów integracyjnych.

-
- Transakcje to potencjalny problem tejże architektury, w sytuacji kiedy transakcje rozpinają się ponad mikroserwisami.

„Zalety i wady mikroserwisów są oczywiste i trudno z nimi dyskutować, jednak warto jak zawsze zdefiniować kontekst. Jaki mamy projekt, jakie wymagania, które z wad aplikują się do naszego przypadku, które z korzyści na nas spłyną, czy w naszym przypadku to się opłaci.”[13]

2.3 Anemic Model

Anemiczny model aplikacji to antywzorzec projektowy, często występujący przy tworzeniu aplikacji monolitycznych opartych o scentralizowaną bazę danych, wykorzystujące operacje CRUD czyli dodawaniu, odczytywaniu, aktualizacji i usuwaniu informacji.

Aplikacje oparte na modelu anemicznym to takie, które opierają się na operacjach bazodanowych i nie są niczym innym jak jedynie oprawą graficzną dla struktury bazy danych. Użytkownik owego system opartego na tym modelu, władający wiedzą z zakresu baz danych, potrafiłby bezpośrednio na poziomie połączenia z bazą danych zmieniać stan danych aplikacji adekwatnie do potrzeb logiki biznesowej, co byłoby w wielu przypadkach równoznaczne z użyciem zaprojektowanego systemu opartego na tej bazie.

Jest to jeden z antyzworców projektowych. „W tym przypadku model dziedziny składa się z klas z atrybutami bez metod, nie jest więc obiektowy. Logika biznesowa przeniesiona jest do innych klas, które transformują klasy dziedziny zmieniając ich stan (...) Antywzorzec ten jest przedmiotem wielu dyskusji – znaczna część metodyk tworzenia oprogramowania w Javie (w tym EJB) operuje na takim modelu. Duża część projektantów przenosi też swoje przyzwyczajenia z modelowania baz danych modelując system w ten sposób.”[2]

2.4 Event Sourcing

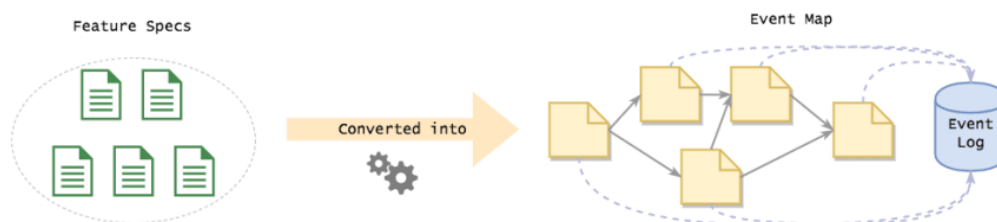
Event Sourcing opiera cały system na wydarzeniach (Event), które zapisywane są trwale do bazy danych (Sourcing).

W innych modelach, opierających się na relacyjnych lub nierelacyjnych bazach danych i operacjach CRUD, skupiamy się jedynie na stanie aktualnym aplikacji. Jakakolwiek aktualizacja czy usunięcie danych zmienia dane w sposób trwały i tracimy informację o wcześniejszej wersji danej sprzed aktualizacji czy usunięcia.

Taka informacja może być bardzo przydatna z paru powodów:

- Specyfikacja domenowa często wymaga zachowania pełnej wersji wydarzeń w systemie. Jednym z przykładów byłyby banki, gdzie każda transakcja musi być zapisana. Innym przykładem są sklepy internetowe, które chcą lepiej zrozumieć zachowanie klienta na stronie. Klient może dodać produkt do koszyka, potem go z niego usunąć i dokonać zakupu innych produktów. Aplikacja w wersji nieopartej na wydarzeniach nie zapisze stanu pośredniego, a jedynie stan końcowy z produktami kupionymi przez użytkownika. Event Sourcing pomaga na zapisanie każdej najmniejszej czynności, opisanej w wymaganiach biznesowych.
- Posiadanie całej historii danych w aplikacji może pozwolić nam na odtworzenie wszystkich operacji od początku lub cofnięcie się w czasie do danego momentu w razie potrzeby. Ta możliwość jest również charakterystyczna dla systemów opartych o Blockchain, który ma wiele wspólnego z Event Sourcing’iem pod tym względem.
- Możemy szybko rozpoznać i zabezpieczyć system przed sytuacjami niebezpiecznymi i atakami osób trzecich na dane aplikacji.

„Zamiast koncentrować się na bieżącym stanie, koncentrujesz się na zmianach, które zaszły w czasie. Jest to praktyka modelowania systemu jako sekwencji zdarzeń.”[18]

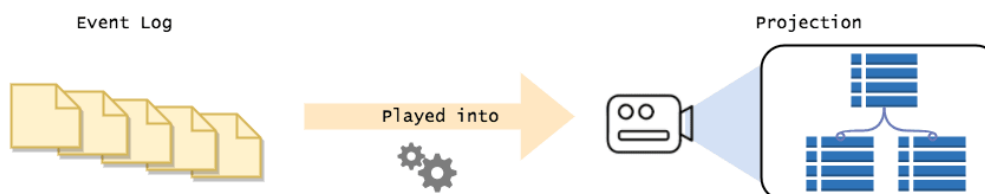


Rysunek 2.3: Zapis wydarzeń w systemie. Źródło: [18]

Każda akcja w aplikacji (ang. Feature Specs) zapisywana jest jako wydarzenie (Event) w tzw. Event Log, czyli bazie danych wydarzeń, która traktowana jest jako źródło prawdy całego systemu.

Bardzo ważną cechą Event Sourcing’u jest sposób na późniejsze przygotowanie i prezentację danych użytkownikowi systemu.

Jeśli każdy stan aplikacji pochodzi od zdarzeń, w jaki sposób możemy pobierać dane, które muszą być prezentowane użytkownikowi? Czy za każdym razem musimy pobierać wszystkie zdarzenia i budować zestaw danych?



Rysunek 2.4: Odczyt danych z wydarzeń poprzez projekcję. Źródło: [18]

„Wyniki budowane są w tle, przechowując rezultaty pośrednie w bazie danych. W ten sposób użytkownicy mogą wyszukiwać dane, a otrzymają je w dokładnie

takim kształcie, jakiego potrzebują, z minimalnym opóźnieniem. W efekcie wyniki są buforowane do późniejszego wykorzystania.” [18]

Obliczanie w tle wyników, które nas interesują bardzo usprawnia wydajność systemu. Ta operacja inaczej nazywana jest projekcją.

Dzięki temu, że zachowujemy całą historię wydarzeń, zapytania o nie możemy kreować w dowolny sposób. Możemy również tworzyć lub zmieniać na bieżąco kształt modelu odczytującego.

2.5 CQRS

CQRS to wzorzec projektowy pomagający w logicznym podziale aplikacji na część zapisującą i odczytującą dane.

„Zanim przejdziemy do bohatera pierwszoplanowego (CQRS), warto zapoznać się z konceptem, z którego bezpośrednio się on wywodzi. Mowa o CQS, czyli Command Query Separation. Został on przedstawiony w roku 1986 przez Bertranda Meyera. Widać więc wyraźnie, że wbrew powszechnemu stwierdzeniu nie jest to nic nowego. Czym zatem jest CQS? Jest to zasada, która mówi że każda metoda w systemie powinna być zaklasyfikowana do jednej z dwóch grup: Command - są to metody, które zmieniają stan aplikacji i nic nie zwracają. Query - są to metody, które coś zwracają, ale nie zmieniają stanu aplikacji.”[6]

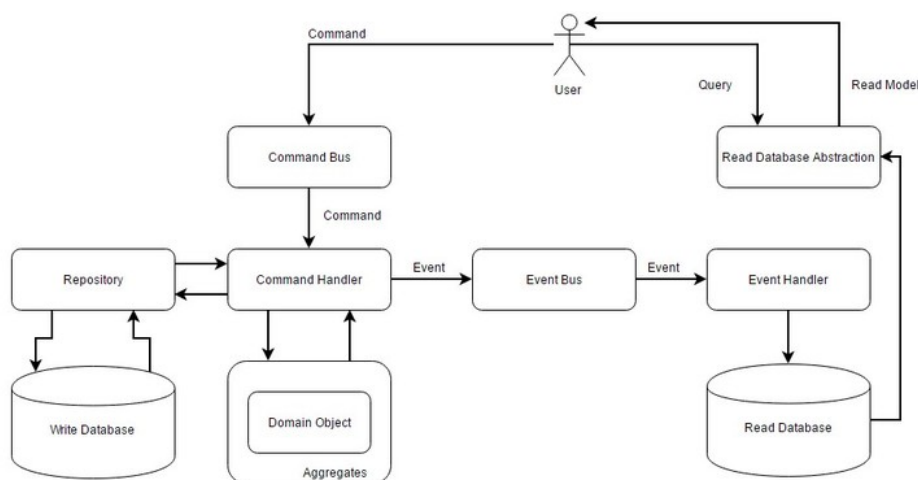
„Blisko 20 lat po narodzinach CQS, dwie wielkie osobistości tj. Greg Young oraz Udi Dahan przedstawili światu jego następcę czyli CQRS - Command Query Responsibility Segregation. Pomysł był bardzo prosty. Dlaczego dokonujemy podziału jedynie metod na te, które pobierają dane oraz na te, które zmieniają stan naszej aplikacji? Możemy przecież zaprojektować nasz system tak, aby tymi zadaniami zajmowały się osobne klasy. To jest główna różnica między dwoma podejściami.”[6]

Implementacja Event Sourcing'u może wykorzystywać CQRS, co pomaga w zachowaniu czystości kodu i podziału odpowiedzialności w kodzie źródłowym. Mimo, iż nie jest to obowiązkowe, jednakże to rozwiązanie zyskuje na popularności w ostatnich latach.

Pojęcia związane z wzorcem CQRS [6]:

- Command to obiekt, który reprezentuje intencje użytkownika systemu.
- Command Bus - ma dwa zadania. Po pierwsze zapewnia kolejkovanie wszystkich wchodzących do systemu komend. Po drugie, odszukuje odpowiedni dla danej komendy Command Handler i wywołuje na nim metodę Handle.
- Command Handler - jego zadaniem jest najpierw walidacja komendy. Następnie tworzy on lub zmienia stan obiektu domenowego.
- Domain objects (models) - są sercem naszej aplikacji. To w nich znajduje się złożoność biznesowa naszego systemu. Warto zwrócić uwagę na to, że na schemacie otacza je jeszcze jedna warstwa, czyli tzw. Aggregates. Jest to wzorzec wywodzący się z Domain-Driven-Design. W dużym uproszczeniu, agregaty mają na celu traktowanie grupy logicznie/biznesowo powiązanych ze sobą obiektów jako jedną jednostkę.
- Event - inaczej wydarzenie, jest to obiekt reprezentujący zmiany, które zaszły w systemie.
- Event Bus - ma dwa zadania. Po pierwsze zapewnia kolejkovanie wszystkich wygenerowanych w systemie zdarzeń. Po drugie, odszukuje odpowiedni dla danego zdarzenia Event Handler i wywołuje na nim metodę Handle.

- Event Handler - jego zadaniem jest zapisanie zmian do bazy danych, która służy do odczytu.
- Read Database Abstraction - jest to nic innego jak warstwa, która pośredniczy w pobieraniu danych. Sposób implementacji jest tutaj dowolny, dlatego sama nazwa na schemacie jest bardzo ogólna.



Rysunek 2.5: Graficzna reprezentacja wzorca CQRS. Źródło: [18]

2.6 Java

„Java - współbieżny, oparty na klasach, obiektowy język programowania ogólnego zastosowania[4]. Został stworzony przez grupę roboczą pod kierunkiem Jamesa Goslinga z firmy Sun Microsystems. Java jest językiem tworzenia programów źródłowych kompilowanych do kodu bajtowego, czyli postaci wykonywanej przez maszynę wirtualną. Język cechuje się silnym typowaniem. Jego podstawowe koncepcje zostały przejęte z języka Smalltalk (maszyna wirtualna, zarządzanie pamięcią) oraz z języka C++ (duża część składni i słów kluczowych).”[11]

W pracy została użyta Java w wersji 10, która wprowadza pewne usprawnienia widoczne zwłaszcza w implementacji przykładowej aplikacji w wersji opartej na Event Sourcing'u, gdzie często używana jest instrukcja "var" dla zmiennych lokalnych.

```
KTable<String, Brand> brandsKTable = brandStream.groupByKey(Serialized.with(
    Serdes.String(),
    CustomJsonSerde.of(Brand.class)))
    .reduce((brand, v1) -> v1);
```

Rysunek 2.6: Standardowa deklaracja zmiennej. Źródło: opracowanie własne

```
var brandsKTable = brandStream.groupByKey(Serialized.with(
    Serdes.String(),
    CustomJsonSerde.of(Brand.class)))
    .reduce((brand, v1) -> v1);
```

Rysunek 2.7: Deklaracja zmiennej w Javie 10. Źródło: opracowanie własne

2.7 REST

„REST – Representational State Transfer – styl architektury oprogramowania opierający się na zbiorze wcześniej określonych reguł opisujących jak definiowane są zasoby, a także umożliwiających dostęp do nich. Został on zaprezentowany przez Roya Fieldinga w 2000 roku.”[29]

2.8 API

„API – Application Programming Interface – zestaw reguł definiujący komunikację pomiędzy programami komputerowymi.

Czyli API są to reguły określające jak użytkownik może uzyskać dostęp do zasobów oraz w jakiej postaci je otrzymuje. Natomiast REST to styl architektury definiujący jak zbudowane będzie to API.”[29]

2.9 HTTP

„HTTP – Hypertext Transfer Protocol – protokół, z którego korzystasz codziennie (lub też jego wersji szyfrowanej – HTTPS) podczas przeglądania stron w sieci. Podczas tworzenia REST API do komunikacji z API wykorzystuje się metody HTTP, których łącznie jest 9. Niemniej jednak do zbudowania podstawowego API pozwalającego na odczyt, zapis, aktualizację i usuwanie danych wystarczą tylko 4 metody – GET, POST, PUT i DELETE.”[29]

2.10 Spring Framework

Spring jest platformą złożoną z wielu projektów, która dedykowana jest do tworzenia aplikacji w języku Java. Jego kluczowym elementem jest kontener wstrzykiwania zależności, jednak przez lata Spring zyskał wsparcie dla wielu technologii i stanowi dziś jeden z kluczowych elementów całego ekosystemu Javy.[14]

2.10.1 Moduły Springa użyte w pracy

Spring składa się z paru modułów, które odpowiadają za daną funkcjonalność m.in. łączenie z bazą danych i udostępnianie REST API. A autor pracy wykorzystał następujące moduły Springa:

Spring Data

„Moduł upraszczający dostęp do baz danych. Główną ideą Spring Data jest zminimalizowanie ilości powtarzalnego kodu, czyli przykładowo jeśli nasza aplika-

cja wykorzystuje JPA, potrzebujemy stworzyć repozytorium udostępniające podstawowe metody CRUD, to korzystając ze Spring Data całość sprowadza się do stworzenia jednego prostego interfejsu.” [14]

```
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;
import pl.javastart.model.Car;

@Repository
public interface CarRepository extends CrudRepository<Car, Long> {
}
```

Rysunek 2.8: Repozytorium Spring Data. Źródło: [14]

Spring MVC

„Najpopularniejszy framework MVC (Model View Controller) w Javie. Stanowi alternatywę dla JSF (JavaServer Faces) z Javy EE. Spring MVC bazuje na technologii serwletów, czyli kluczowej specyfikacji Javy EE i do działania wymaga kontenera serwletów. W najnowszej wersji konfigurację można w całości oprzeć o adnotacje.” [14] Ten moduł wykorzystywany jest w tejże pracy do wystawiania REST API.

2.11 Hibernate

Hibernate to framework do realizacji warstwy dostępu do danych. Zapewnia on translację danych pomiędzy relacyjną bazą danych a światem obiektywnym. Opiera się na wykorzystaniu opisu struktury danych za pomocą języka XML lub adnotacji w kodzie, dzięki czemu można rzutować obiekty, stosowane w obiektowych językach programowania, takich jak Java bezpośrednio na istniejące tabele bazy danych. Dodatkowo Hibernate zwiększa wydajność operacji na bazie danych

dzięki buforowaniu i minimalizacji liczby przesyłanych zapytań. Jest to projekt open source'owy. [9]

Hibernate, w połączeniu z modułem Spring Data, użyty jest do wymiany informacji z bazą danych w aplikacji w wersji monolitycznej, przedstawionej w kolejnym rozdziale.

Autor pracy użył Hibernate w połączeniu z adnotacjami w Javie, dzięki czemu nie ma potrzeby definiowania struktury obiektów w plikach o formacie XML.

```
@Entity
@Table(name = "EMPLOYEE")
public class Employee {
    @Id @GeneratedValue
    @Column(name = "id")
    private int id;
    @Column(name = "first_name")
    private String first_name;
    @Column(name = "last_name")
    private String last_name;
    @Column(name = "dept_id")
    private int dept_id;
```

Rysunek 2.9: Klasa wraz z adnotacjami Hibernate. Źródło: [10]

2.12 Apache Kafka

Apache Kafka to open sourceowy broker wiadomości napisany w Javie i Scali. Kafka w ostatnich latach zyskuje bardzo na popularności ze względu na innowacyjne rozwiązania dotyczące skalowalności i elastyczności rozwiązania.

„Cechą charakterystyczną Kafki jest jej niezawodność, wydajność i zdolność do pracy w środowisku rozproszonym. Kiedy zespół LinkedIn tworzył Kafkę, ich główną motywacją było poradzenie sobie z przetwarzaniem w czasie rzeczywistym ogromnej ilości zdarzeń. W konsekwencji powstało narzędzie, które jest idealne do wspomagania przetwarzania dynamicznie zmieniających się źródeł danych o dużej objętości i zmienności, na przykład strumieni big data.

W 2014 roku w LinkedIn komunikacja za pośrednictwem Kafki przebiegała zarówno pomiędzy klastrami, jak i centrami danych. W sumie przesyłano około 200 miliardów komunikatów dziennie, osiągając 7 milionów komunikatów na sekundę, jeśli chodzi o zapis i 35 milionów na sekundę w przypadku odczytu. Z Kafki, poza LinkedIn, korzystają między innymi Netflix, Twitter, Spotify, Cisco, czy Coursera.”[3]

2.12.1 Architektura Kafki

Kafka umożliwia przesyłanie komunikatów pomiędzy aplikacjami w systemach rozproszonych. Nadawca może przysyłać komunikaty do Kafki, a odbiorca pobierać wiadomości ze strumienia publikowanego przez Kafkę.

Tematy

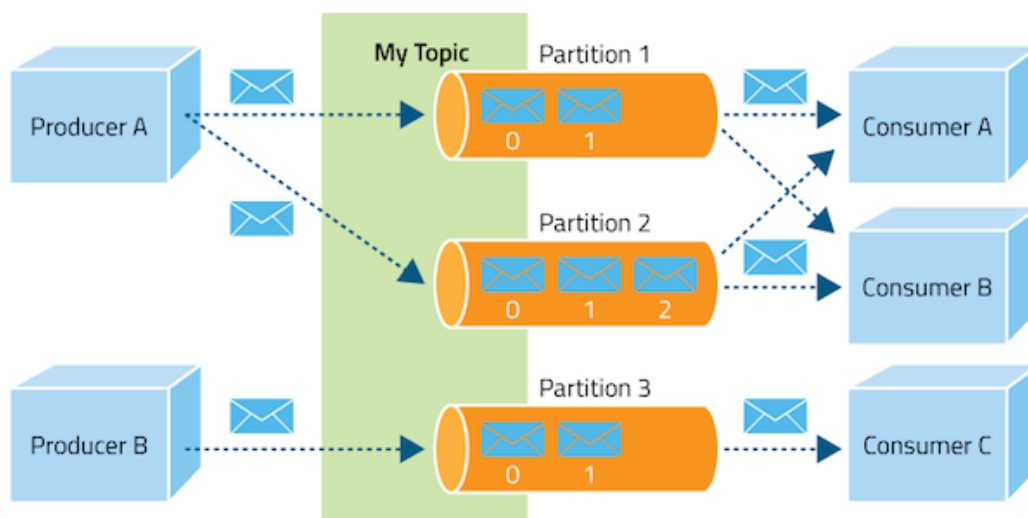
Komunikaty pogrupowane są w tzw. tematy (ang. topic). Nadawca przesyła komunikaty z określonego tematu, a odbiorca otrzymuje za pośrednictwem Kafki wszystkie komunikaty z określonego tematu, które mogą pochodzić nawet od wielu nadawców. Każdy wysłany przez dowolnego nadawcę komunikat z danego tematu trafi do każdego odbiorcy, który nasłuchuje tego tematu. Całość odbywa się w środowisku rozproszonym, czyli opratym o mikroserwisy.[3]

Partycje

„Komunikaty z danego tematu dopisywane są do tzw. partycji (ang. partition). Partycja to pewien rejestr, uporządkowana sekwencja komunikatów, która nie zmienia się, oprócz tego, że nowe komunikaty mogą zostać dopisane na koniec tej sekwencji, a stare – na przykład starsze niż dwa dni – są zapominane. Aby pobrać odpowiednią sekwencję komunikatów, odbiorcy muszą jedynie znać swoją pozycję w rejestrze – indeks ostatnio odczytanego komunikatu.

Pojedyncza partycja musi w całości zmieścić się na jednym serwerze, musi być możliwe obsłużenie jej przez jednego brokera. Z innej strony, jeśli masz jedną partycję z jednego tematu, do zapisu i odczytu komunikatów będzie wykorzystywany tylko jeden broker. Większa liczba partycji pozwala na wykorzystanie większej liczby brokerów w celu zrównoleglenia zapisu i odczytu komunikatów, a tym samym na zwiększenie wydajności klastra. Twórcy gwarantują wydajne działanie Kafki nawet dla 10 000 partycji – maksymalnie dla takiej ilości przeprowadzają testy wydajnościowe.”[3]

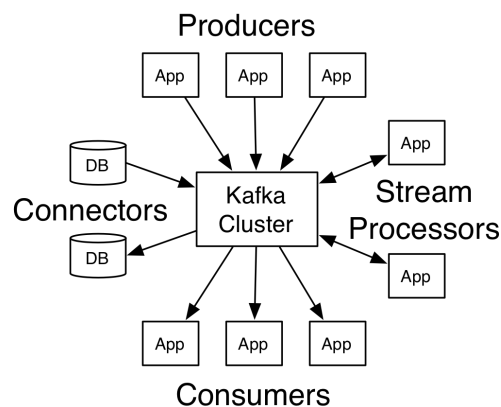
Partycje Kafki są replikowalne i ich kopie mogą znajdować się na wielu serwerach. Dla każdej partycji istnieje jeden serwer, który jest tzw. liderem (ang. leader) i który obsługuje wszystkie operacje odczytu i zapisu danej partycji. Dla każdej partycji mogą istnieć także serwery (ang. followers), które jedynie replikują dane od lidera.[3]



Rysunek 2.10: Relacja między tematem a partycją w Apache Kafka. Źródło: [4]

2.12.2 Kafka Core APIs

Kafka ma cztery podstawowe API [19]:



Rysunek 2.11: Apache Kafka API. Źródło: [4]

- Producer API umożliwia aplikacji publikowanie strumienia rekordów do jednego lub więcej tematów Kafki.
- Consumer API pozwala aplikacji subskrybować jeden lub więcej tematów i przetwarzać strumień wygenerowanych rekordów.
- Streams API pozwala aplikacji działać jako procesor strumieniowy, pochłaniając strumień wejściowy z jednego lub więcej tematów i generując strumień wyjściowy do jednego lub większej liczby tematów wyjściowych, skutecznie przekształcając strumienie wejściowe w strumienie wyjściowe.
- Connector API umożliwia budowanie i uruchamianie producentów lub konsumentów wielokrotnego użytku, łączących tematy Kafki z istniejącymi aplikacjami lub systemami danych. Na przykład łącznik do relacyjnej bazy danych może przechwytywać każdą zmianę w tabeli.

W niniejszej pracy zostały wykorzystane Producer API i Streams API, które zostały opisane poniżej.

Producer API

Jego główną funkcją jest mapowanie każdej wiadomości na partycję tematu i wysyłanie wydarzenia do lidera tej partycji. Najpierw zostaje obliczona partycja na podstawie klucza, gdzie wydarzenie ma zostać umieszczone. Partycje dostarczane z Kafką gwarantują, że wszystkie wiadomości z tym samym niepustym kluczem zostaną wysłane na tę samą partycję. Jeśli nie podano klucza, partycja jest wybierana w sposób zbalansowany, aby zapewnić równomierną dystrybucję między partycjami tematów.[20]

```
var message = ImmutableOrderComponentEvent.builder().body(body)
    .header(ImmutableOrderComponentHeader.builder().action(actionEnum)
        .orderId(orderId)
        .entityId(entityId)
        .userId(userId)
        .creationTimestamp(Instant.now())
        .type(type)
        .build()).build();
Future<RecordMetadata> send = producer.send(
    new ProducerRecord<>(topic, entityId, message));
send.get(timeout: 5, TimeUnit.SECONDS);
```

Rysunek 2.12: Przykładowe Producer API w Javie. Źródło: opracowanie własne

Streams API

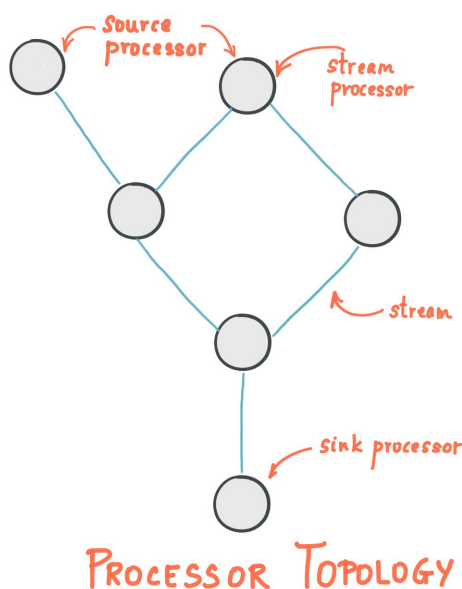
Kafka Streams jest biblioteką klienta do przetwarzania i analizy danych przechowywanych w Kafce. Opiera się na ważnych koncepcjach przetwarzania strumienia, takich jak odpowiednie rozróżnianie czasu zdarzenia i czasu przetwarzania, obsługa okien i proste, ale wydajne zarządzanie i sprawdzanie w czasie rzeczywistym stanu aplikacji.[16]

Kafka posiada wbudowaną możliwość operacji na strumieniach zdarzeń (ang. Streams) i możemy definiować dowolną ilość procesorów (ang. Stream Processors).

Strumień jest najważniejszą abstrakcją dostarczaną przez Streams API: reprezentuje nieograniczony, stale aktualizowany zbiór danych. Strumień jest uporządkowaną, powtarzalną i odporną na uszkodzenia sekwencją niezmiennych rekordów danych, w której rekord danych jest definiowany jako para klucz-wartość.

Aplikacja do przetwarzania strumienia to dowolny program korzystający z biblioteki Streams API. Definiuje swoją logikę obliczeniową przez jedną lub więcej topologii procesora, gdzie topologia jest definicją łączącą procesory strumieniowe.

Procesor strumienia jest węzłem w topologii procesora; reprezentuje on etap przetwarzania w celu transformacji danych w strumieniach przez otrzymywanie jednego rekordu wejściowego w tym samym czasie od jego przyszlých procesorów w topologii, stosując do niego swoją operację, i może następnie wytwarzać jeden lub większą liczbę rekordów wyjściowych w swoich procesorach końcowych.[16]



Rysunek 2.13: Topologia procesora w Apache Kafka. Źródło: [16]

```
var cardStream = transformedStream.mapValues
    ((s, processingWrapper) -> processingWrapper.getCard())
    .filter(((s, card) -> nonNull(card)))
    .peek((key, value) -> logger.info("card: {}: {}", key, value));
cardStream.to(s: "card", CustomJsonSerde.produce(Card.class));
```

Rysunek 2.14: Przykładowe Stream API w Javie. Źródło: opracowanie własne

Dualność strumieni i tabel w Kafce

Podczas implementowania przypadków użycia strumienia w praktyce zazwyczaj jest potrzeba użycia zarówno strumieni, jak i baz danych. Przykładem, który jest bardzo popularny w praktyce, jest aplikacja e-commerce, która wzbogaca przychodzący strumień transakcji klientów o najnowsze informacje o klientach z tabeli bazy danych. Innymi słowy, strumienie są wszędzie, jak i bazy danych.

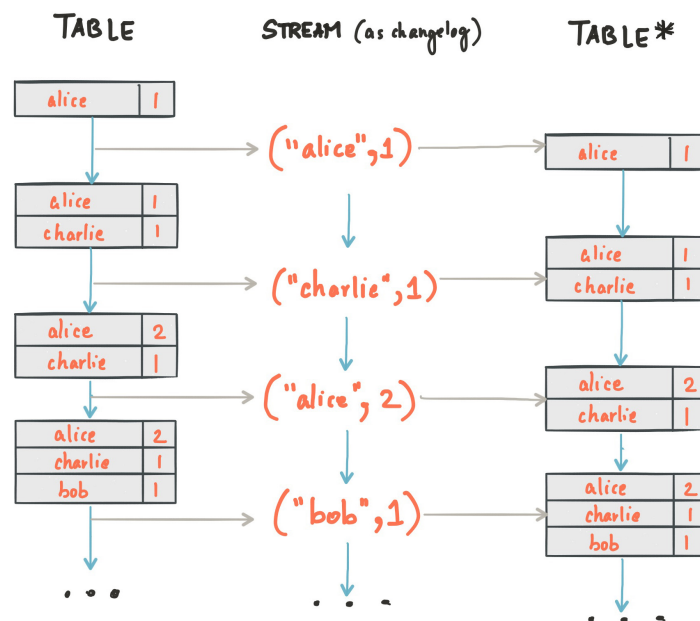
Każda technologia przetwarzania strumieniowego musi zatem zapewniać wysokiej klasy obsługę strumieni i tabel. Kafka's Streams API zapewnia taką funkcjonalność poprzez podstawowe abstrakcje obu pojęć. Interesującą obserwacją jest to, że istnieje ścisły związek między strumieniami i tabelami, tak zwana dwoistość strumieniowo-tabelowa. Kafka wykorzystuje tę dwoistość na wiele sposobów: na przykład, aby uczynić aplikacje elastycznymi, aby wspierać przetwarzanie stanowe z tolerancją na błędy lub aby uruchamiać interaktywne zapytania wobec najnowszych wyników przetwarzania aplikacji. Poza swoim wewnętrznym wykorzystaniem API Kafka Streams umożliwia deweloperom wykorzystanie tej dwoistości w swoich własnych aplikacjach. [27]

Dualizm tabeli i strumieni opisuje bliskie relacje między strumieniami i tabelami: [27]

- Strumień jako tabela - Strumień można uznać za dziennik zmian danej tabeli, w której każdy rekord danych w strumieniu przechwytyje zmianę stanu tabeli. Strumień można łatwo przekształcić w tabelę, odtwarzając

listę zmian od początku do końca, aby zrekonstruować tabelę. Podobnie agregowanie rekordów danych w strumieniu zwróci tabelę. Na przykład możemy obliczyć całkowitą liczbę odsłon użytkownika według strumienia wejściowego zdarzeń odsłon strony, a wynikiem będzie tabela, przy czym kluczem tabeli będzie użytkownik, a wartością będzie odpowiadać liczbie odsłon strony.

- Tabela jako strumień: tabelę można uznać za migawkę ostatniej wartości dla każdego klucza w strumieniu (rekordy danych strumienia są parami klucz-wartość). Tabela jest zatem zakamuflowanym strumieniem i można go łatwo przekształcić w "prawdziwy strumień" przez iterowanie po każdym wpisie klucz-wartość w tabeli.



Rysunek 2.15: Dualizm strumieni i tabel. Źródło: [27]

2.13 Docker

Docker to narzędzie zaprojektowane w celu ułatwienia tworzenia, wdrażania i uruchamiania aplikacji przy użyciu kontenerów. Kontenery umożliwiają programistom skompletowanie aplikacji z wszystkimi potrzebnymi częściami, takimi jak biblioteki i inne zależności, i wysłanie jej jako jednej paczki. Dzięki temu, dzięki pojemnikowi, twórca może mieć pewność, że aplikacja będzie działała na dowolnym innym komputerze, niezależnie od dowolnych dostosowanych ustawień, które może mieć maszyna, która może różnić się od maszyny używanej do pisania i testowania kodu.

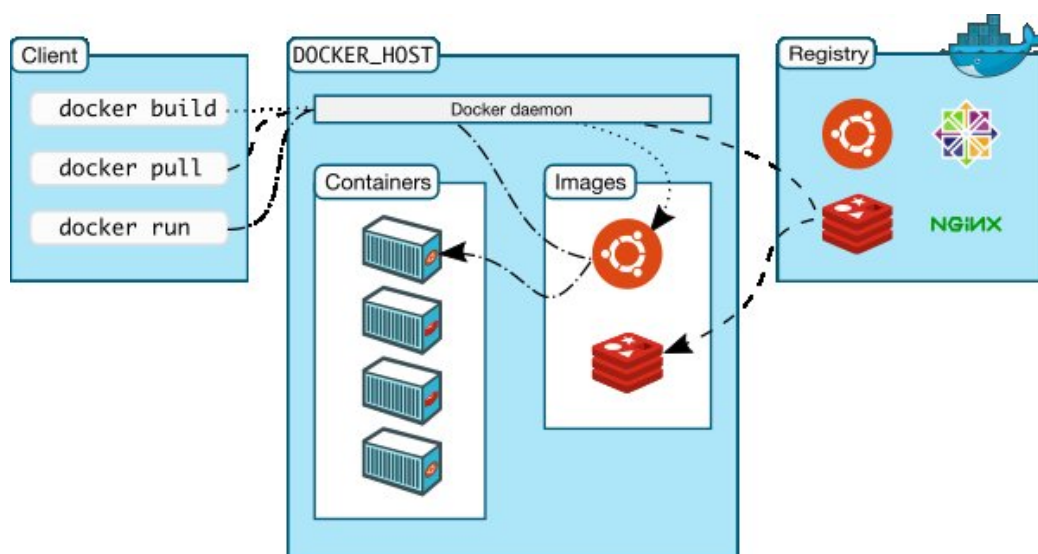
W pewnym sensie Docker przypomina trochę maszynę wirtualną. Jednak w przeciwieństwie do maszyny wirtualnej, zamiast tworzyć cały wirtualny system operacyjny, Docker pozwala aplikacjom używać tego samego jądra systemu Linux, co system, na którym działają, i wymaga jedynie aplikacji dostarczanych z rzeczami, które nie są jeszcze uruchomione na komputerze hosta. Daje to znaczny wzrost wydajności i zmniejsza rozmiar aplikacji. [23]

2.13.1 Architektura Docker'owa

Docker używa architektury klient-serwer. Klient Docker rozmawia z daemonem Docker, który zajmuje się budowaniem, uruchamianiem i dystrybucją kontenerów Docker. Klient i daemon Docker mogą działać w tym samym systemie lub można podłączyć klienta Docker do zdalnego demona Docker. Klient i daemon Docker komunikują się za pomocą REST API lub interfejs sieciowy. [17]

Docker Daemon

Docker Daemon nasłuchuje żądań interfejsu Docker API i zarządza obiektami Docker, takimi jak obrazy, kontenery, sieci i woluminy. Daemon może również komunikować się z innymi daemonami w celu zarządzania usługami Docker. [17]



Rysunek 2.16: Architektura Docker'owa. Źródło: [17]

Klient Docker'owy

Klient Docker'owy to podstawowy sposób interakcji wielu użytkowników Docker z Dockerem. Podczas korzystania z poleceń, takich jak uruchamianie dokera, klient wysyła te polecenia do dockerd, który je wykonuje. Polecenie „docker” używa interfejsu API Docker. Klient Docker'owy może komunikować się z więcej niż jednym deamonem. [17]

Rejestry Docker'owe

Rejestr Docker'owy przechowuje obrazy Docker'owe. Docker Hub i Docker Cloud są publicznymi rejestrami, z których każdy może korzystać, a Docker jest domyślnie skonfigurowany do wyszukiwania obrazów w Docker Hub. Możesz nawet uruchomić swój prywatny rejestr.

Podczas korzystania z poleceń otwierania okna dokowanego lub poleceń dokowania wymagane obrazy są pobierane ze skonfigurowanego rejestru. Po użyciu polecenia „push” obraz zostanie przekazany do skonfigurowanego rejestru.

Sklep Docker pozwala kupować i sprzedawać obrazy Docker lub dystrybuować je za darmo. Na przykład można kupić obraz Docker zawierający aplikację lub usługę od dostawcy oprogramowania i użyć obrazu do wdrożenia aplikacji w środowiskach deweloperskich, testowych i produkcyjnych.

Obiekty Docker’owe

Podczas korzystania z Docker tworzysz i używasz obrazów, kontenerów, sieci, woluminów, wtyczek i innych obiektów.

Obraz jest szablonem tylko do odczytu z instrukcjami dotyczącymi tworzenia kontenera Docker’owego. Często obraz jest oparty na innym obrazie, z dodatkową dostosowaną konfiguracją do danych potrzeb.

Można tworzyć własne obrazy lub korzystać tylko z tych stworzonych przez innych i opublikowanych w rejestrze. Aby zbudować własny obraz, należy utworzyć plik Dockerfile z prostą składnią do definiowania kroków potrzebnych do utworzenia obrazu i uruchomienia go. Każda instrukcja w pliku Dockerfile tworzy warstwę na obrazie. Gdy zmienisz plik Dockerfile i odbudujesz obraz, odbudowane zostaną tylko te warstwy, które się zmieniły. Jest to część tego, co sprawia, że obrazy są tak lekkie, małe i szybkie, w porównaniu do innych technologii wirtualizacji.

Kontener to działająca instancja obrazu. Za pomocą interfejsu API lub interfejsu CLI można tworzyć, uruchamiać, zatrzymywać, przenosić i usuwać kontenery. Możesz podłączyć kontener do jednej lub wielu sieci, dołączyć do niego pamięć masową, a nawet utworzyć nowy obraz w oparciu o jego aktualny stan.

Domyślnie kontener jest stosunkowo dobrze izolowany od innych kontenerów i komputera-hosta. Możesz kontrolować sposób odizolowania sieci, kontenera lub innych podstawowych podsystemów od innych kontenerów lub hosta. [17]

2.14 Kafka Lenses Box

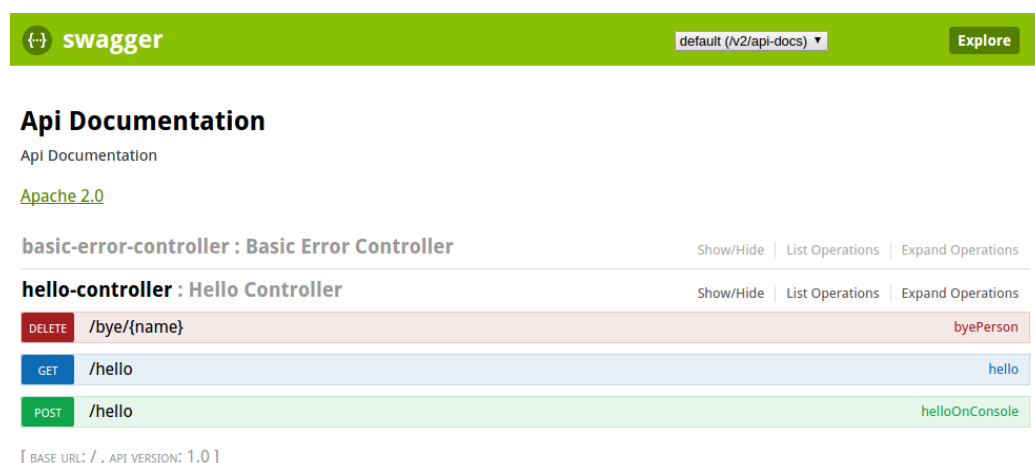
Dla łatwości implementacji Event Sourcing'u, autor pracy opiera się na gotowym obrazie Docker'owym Apache Kafki, który nazywa się Lenses Box.

Lenses Box to obraz Docker'owy zawierający pełną instalację Apache Kafki wraz ze wszystkimi jego odpowiednimi komponentami. Wszystko to jest automatycznie skonfigurowane i jedynym wymaganiem jest zainstalowanie Dockera. Zawiera Lenses, brokera Kafki, Kafka Connect, 25+ Kafka Connectors z obsługą SQL, narzędzia CLI. [21]

2.15 Swagger

Aplikacje zaimplementowane na potrzeby tej pracy posiadają jedynie back-end aplikacji bez interfejsu użytkownika napisanego w technologiach front-endowych.

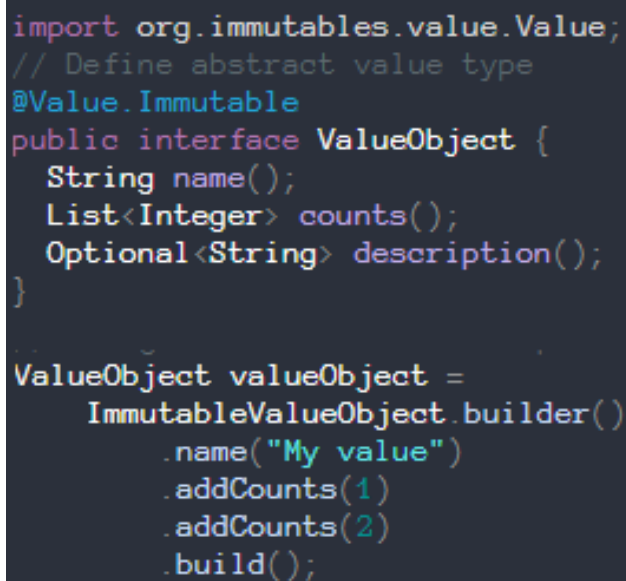
Prostym interfejsem, generowanym na podstawie endpoint'ów RESTowych, jest Swagger, który stwarza automatycznie możliwość wysyłania każdego typu zapytania HTTP wraz z zawartością JSON.



Rysunek 2.17: Przykładowy interfejs Swagger'owy. Źródło: [25]

2.16 Immutables

Immutables to biblioteka do generowania za pomocą adnotacji niezmiennych obiektów w Javie, czyli takich, których stan nie może zostać zmieniony przez cały okres życia obiektu.



```
import org.immutables.value.Value;
// Define abstract value type
@Value.Immutable
public interface ValueObject {
    String name();
    List<Integer> counts();
    Optional<String> description();
}

ValueObject valueObject =
    ImmutableValueObject.builder()
        .name("My value")
        .addCounts(1)
        .addCounts(2)
        .build();
```

Rysunek 2.18: Definicja i tworzenie obiektu niezmiennego. Źródło: [24]

2.17 Apache Maven

„Apache Maven - narzędzie automatyzujące budowę oprogramowania na platformę Java. Poszczególne funkcje Mavena realizowane są poprzez wtyczki, które są automatycznie pobierane przy ich pierwszym wykorzystaniu. Plik określający sposób budowy aplikacji nosi nazwę POM-u (ang. Project Object Model).”[28]

Apache Maven został wykorzystany jako narzędzie do budowania aplikacji w ramach ten pracy. Apache Maven wykorzystuje komendę „mvn spring-boot:run”, która buduje i łączy aplikację poprzez wbudowany kontener Tomcat.

Rozdział 3

Implementacja aplikacji

Rozdział poświęcony jest opisowi aplikacji, które zostały stworzone na potrzeby pracy. Na podstawie ich implementacji wysuwane są wnioski w kolejnym rozdziale.

3.1 Opis domeny

Domena problemowa, którą wybrał autor pracy do zaimplementowania, jest bardzo prosta ideowo i biznesowo. Jest to aplikacja internetowa, w której można wystawiać i sprzedawać karty podarunkowe.

Istnieją trzy rodzaje użytkowników:

- Kupujący,
- Sprzedający,
- Administrator.

Główną encją, na której przeprowadzane są operacje w systemie to karta podarunkowa. Każda karta podarunkowa musi mieć markę (ang. brand), cenę sprzedaży, wartość rynkową, datę ważności i kod do wykorzystania w sklepie. Karta może być fizyczna lub elektroniczna. Na podstawie ceny wyznacza się przedział cenowy (ang. price range) w której się znajduje.

Operacje, które mogą być wykonywane w systemie to:

- Zaawansowane wyszukiwanie z filtrami i sortowaniem branż, wraz z ich aktualną maksymalną zniżką. Operacja przeznaczona dla kupującego.
- Zaawansowane wyszukiwanie z filtrami i sortowaniem listy dostępnych kart wg marek, do których należą. Operacja przeznaczona również dla kupującego.
- Lista i filtrowanie wszystkich kart i transakcji dla administratora.
- Weryfikacja karty przez administratora. Dopiero po weryfikacji karta pojawia się w systemie sprzedaży.
- Operacje zamówienia (określana jako „checkout”) i zapłaty (określana jako „pay”) karty przez kupującego. Sprzedający może taką operację widzieć w systemie odpytując listę transakcji, w których bierze udział.
- Operacja wysyłki karty przez sprzedającego do kupującego.
- Lista wszystkich transakcji dla zarówno kupującego jak i sprzedającego, w których biorą udział.
- Możliwość edycji i usuwania marki i karty przez administratora.
- Możliwość edycji i usuwania karty, którą sprzedający umieścił w systemie.

3.2 Implementacja wersji opartej na monolicie z anemicznym antywzorcem

Pierwszym i najbardziej naturalnym podejściem do zaimplementowania domeny problemowej opisanej powyżej jest stworzenie aplikacji monolitycznej opartej na

relacyjnej bazy danych. Wszystkie operacje w systemie mają odzwierciedlenie na encji bazy danych jaką jest karta podarunkowa.

Takie podejście jest dla większości programistów pierwszym wyborem, jeżeli myśli się o szybkiej implementacji systemu i walidacji pomysłu jako MVP (ang. minimal value product). Większość frameworków w różnych językach programowania opiera się właśnie na tym modelu.

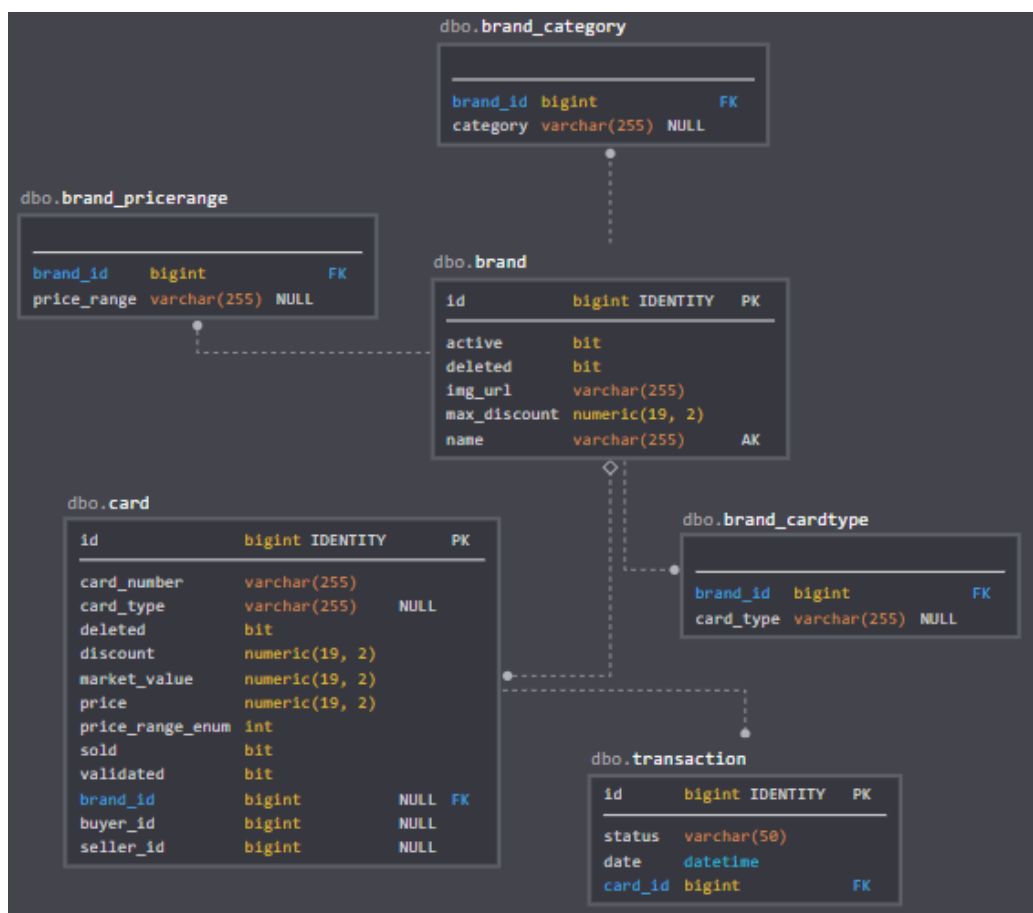
Dodatkowo, aplikacja została napisana w sposób anemiczny, czyli jest to tak naprawdę zastąpienie manualnego pisania zapytań bazodanowych poprzez dodatkową warstwę aplikacji, która jest pewnego rodzaju interfejsem dla użytkownika do wykonywania operacji.

Implementacja opiera się na Javie w wersji 10 i Spring Framework. Użyte zostały moduły Spring Boot i Spring Data.

3.2.1 Diagram bazy danych

Diagram przedstawia relacje między encjami, które opisują zakładaną domenę problemową. Na tej strukturze bazodanowej opiera się omawiana implementacja aplikacji. Głównymi encjami jest marka (ang. brand), karta (ang. card) i transakcja (ang. transaction) i użytkownik (ang. user).

Operacje zakupu i wysyłki kart opierają się na encji transakcji, zawierająca aktualny stan (np. zainicjowana).



Rysunek 3.1: Diagram bazy danych. Źródło: opracowanie własne

3.2.2 Organizacja kodu

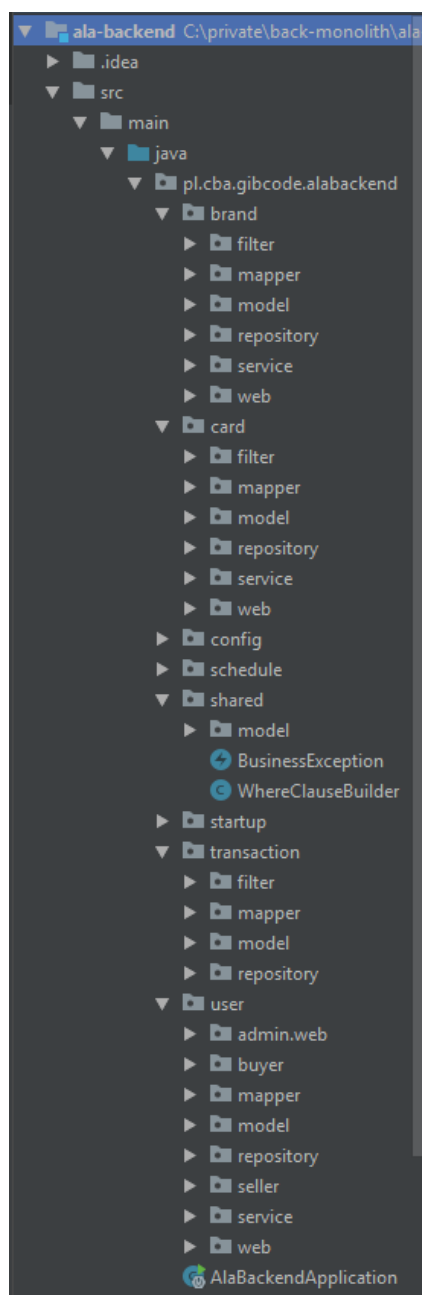
Kod został logicznie podzielony na pakiety (ang. packages), gdzie większość pakietów znajdujących się w pl.cba.gibcode.alabackend odpowiada za typ encji i wszystkie operacje z nią związane. Możemy wyróżnić następujące pakiety w lokalizacji pl.cba.gibcode.alabackend:

- brand - wszystkie operacje na marce karty,
- card - wszystkie operacje dotyczące kart,
- transaction - wszystkie operacje na transakcjach,

-
- user - wszystkie operacje na użytkownikach,
 - config - wszystkie klasy konfiguracyjne (np. konfiguracja Swaggera)
 - shared - wspólny kod zawierający wszystkie modele w aplikacji, które są używane w wielu miejscach na raz,
 - schedule - scheduler, który w równomiernym przedziale czasowym wymusza czyszczenie kart, które zostały kupione, ale nie zostały zapłacone w danym okresie czasu,
 - startup - skrypt tworzący przykładową populację danych w bazie danych podczas włączania aplikacji.

Każda z ww. pakietów związanych z daną encją (tj. brand, card, transaction i user) posiada zbliżoną strukturę, składającą się z logicznego podziału:

- filter - wszystkie klasy tworzące filtry do zapytań wg kryteriów tworzonych po stronie REST Api,
- mapper - pakiet ze wszystkimi mapowaniami między obiektami encji, a obiektami, które są odpowiedzią lub zapytaniem REST Api,
- model - definicje klas wszystkich modeli dotyczących danej encji,
- repository - repozytoria encji używające moduł Spring Data,
- service - tzw. serwisy, które dokonują zmian na encjach, zmieniają ich stan, który zapisują poprzez repozytoria,
- web - miejsce na kontrolery REST Api używające Spring MVC Rest Controllers.



Rysunek 3.2: Struktura aplikacji monolitycznej. Źródło: opracowanie własne

3.2.3 Repozytoria encji

Repozytoria encji oparte są na Spring Data i są fasadami umożliwiającymi odczyt i zapis danych w bazie danych. Dla każdej encji istnieje osobne repozytorium. Do

zdefiniowanych repozytoriów należą:

- BrandRepository
- CardRepository
- TransactionRepository
- UserRepository
- UserDetailsRepository

3.2.4 Połączenie z bazą danych

Aplikacja opiera się na Hibernate jako ORM (ang. object-role modeling), gdzie sama implementacja bazy danych jest dowolona (może być to serwer mySql, PostgreSQL, czy jakikolwiek inny relacyjny silnik bazodanowy). W aktualnej wersji projektu domyślnie aplikacja opiera się na bazie danych w pamięci H2. Za każdym resetowaniem aplikacji, baza danych zostaje zupełnie czyszczona i odtwarzana ponownie. W aplikacji autor pracy zamieścił automatyczne skrypty, które wstępnie ładują przykładowe dane do bazy danych, na których można wykonywać następnie operacje.

```

public class ApplicationStartup implements ApplicationListener<ApplicationReadyEvent> {

    private static final String IMG_BASE_URL = "../../assets/img/";
    private final BrandRepository brandRepository;

    /**
     * This event is executed as late as conceivably possible to indicate that
     * the application is ready to service requests.
     */
    @Override
    @Transactional
    public void onApplicationEvent(final ApplicationReadyEvent event) {
        log.info("Loading test data");
        brandRepository.deleteAll();
        brandRepository.flush();
        createBrand( name: "Adidas", imgUrl: IMG_BASE_URL + "adidas-gc-taxon.1.png",
            Arrays.asList(CardTypeEnum.ELECTRONIC, CardTypeEnum.PHYSICAL),
            Collections.singletonList(CategoryEnum.FASHION),
            Arrays.asList(PriceRangeEnum.ZERO_FIFTY, PriceRangeEnum.FIFTY_HUNDRED));
        createBrand( name: "Airbnb", imgUrl: IMG_BASE_URL + "airbnb-taxon.png",
            Arrays.asList(CardTypeEnum.ELECTRONIC, CardTypeEnum.PHYSICAL),
            Collections.singletonList(CategoryEnum.SERVICES),
            Arrays.asList(PriceRangeEnum.FIFTY_HUNDRED));
        createBrand( name: "Aldo", imgUrl: IMG_BASE_URL + "Aldo.png",
            Arrays.asList(CardTypeEnum.ELECTRONIC, CardTypeEnum.PHYSICAL),
            Collections.singletonList(CategoryEnum.FASHION),
            Arrays.asList(PriceRangeEnum.ZERO_FIFTY));
        createBrand( name: "BP", imgUrl: IMG_BASE_URL + "bp_taxon.png",
            Arrays.asList(CardTypeEnum.ELECTRONIC),
            Collections.singletonList(CategoryEnum.SERVICES),
            Arrays.asList(PriceRangeEnum.FIFTY_HUNDRED, PriceRangeEnum.HUNDRED_FIVEHUNDRED));
        createBrand( name: "Burger King", imgUrl: IMG_BASE_URL + "burger-king-gift-card.png",
            Arrays.asList(CardTypeEnum.ELECTRONIC, CardTypeEnum.PHYSICAL),
            Collections.singletonList(CategoryEnum.RESTAURANTS),

```

Rysunek 3.3: Fragment klasy ładującej dane przy włączaniu aplikacji. Źródło: opracowanie własne

3.2.5 REST'owe endpointy

Każda z funkcjonalności opisana w domenie aplikacji jest dostępna przez REST Api udostępniane z aplikacji. Funkcjonalności zostały pogrupowane wg encji. Dla każdej encji stworzono kontroler z nią powiązany.

| | |
|--------------------------|-------------------|
| admin-controller | Admin Controller |
| brand-controller | Brand Controller |
| buyer-controller | Buyer Controller |
| card-controller | Card Controller |
| seller-controller | Seller Controller |
| user-controller | User Controller |

Rysunek 3.4: Lista kontrolerów w aplikacji. Źródło: opracowanie własne

Kontroler administratora

Kontroler dla administratora składa się z następującej listy endpointów:

- admin/brands z metodą HTTP GET - lista wszystkich marek dla administratora,
- admin/brands z metodą HTTP PUT - edycja marki,
- admin/brands z metodą HTTP DELETE - usunięcie marki,
- admin/cards z metodą HTTP GET - lista wszystkich kart,
- admin/cards z metodą HTTP DELETE - usunięcie danej karty,
- admin/cards/validate - walidowanie danej karty,
- admin/users - lista wszystkich użytkowników.

| admin-controller Admin Controller | | |
|-----------------------------------|---------------------------|--------------------------|
| POST | /api/admin/brands | Get all brands for admin |
| PUT | /api/admin/brands | Update brand by admin |
| DELETE | /api/admin/brands | Delete brand by admin |
| POST | /api/admin/cards | Get all cards for admin |
| DELETE | /api/admin/cards | Delete card for admin |
| POST | /api/admin/cards/validate | Validate card for admin |
| GET | /api/admin/users | Get all users for admin |

Rysunek 3.5: Endpointy kontrolera administratora. Źródło: opracowanie własne

Kontroler marki

Kontroler marek (ang. brands) składa się z następującej listy endpointów:

- brands/create - tworzenie nowej marki,
- brands/list - lista wszystkich dostępnych marek (takie, które zawierają choć jedną kartę na sprzedaż).

| brand-controller Brand Controller | |
|-----------------------------------|-------------------------------------|
| POST | /api/brands/create Create new brand |
| POST | /api/brands/list Get all brands |

Rysunek 3.6: Endpointy kontrolera marek. Źródło: opracowanie własne

Kontroler kupującego

Kontroler kupującego (ang. buyer) składa się z następującej listy endpointów:

- buyer/checkout - podsumowanie zakupu karty (jeszcze przed zapłatą)
- buyer/pay - kupujący płaci za utworzoną transakcję przy buyer/checkout
- buyer/transactions - lista wszystkich transakcji kupującego

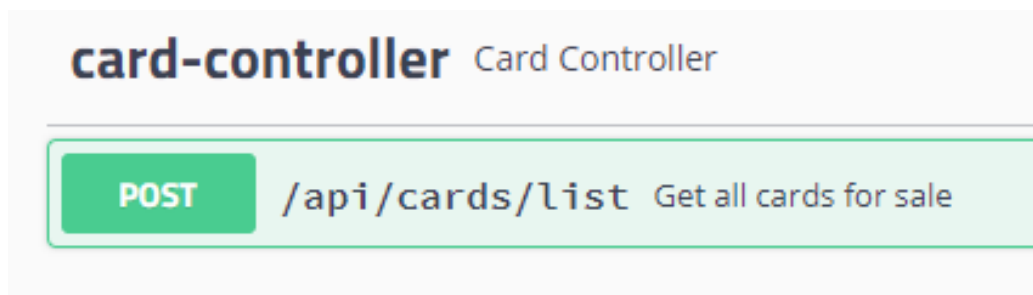
| buyer-controller Buyer Controller | |
|-----------------------------------|--|
| POST | /api/buyer/checkout Checkout a card as a buyer |
| POST | /api/buyer/pay Pay a card as a buyer |
| GET | /api/buyer/transactions Get all transactions for a buyer |

Rysunek 3.7: Endpointy kontrolera kupującego. Źródło: opracowanie własne

Kontroler kart

Kontroler kart składa się z jednego endpointu:

- `cards/list` - lista kart dostępnych do sprzedaży



Rysunek 3.8: Endpointy kontrolera kart. Źródło: opracowanie własne

3.2.6 Uruchomienie aplikacji

Aby uruchomić aplikację trzeba zainstalować Apache Maven w wersji 3. Po tym, w głównym folderze z kodem źródłowym należy podać komendę w terminalu „mvn spring-boot:run”. Ta komenda spowoduje, że aplikacja zostanie zbudowana i wystartuje na wbudowanym serwerze aplikacyjnym Tomcat na porcie 8080.

Aplikacja po uruchomieniu powinna być dostępna na `http://localhost:8080/alakarta/swagger-ui.html`. Jest to strona z interfejsem udostępnionym przez Swaggera, gdzie można wykonywać interaktywne zapytanie do udostępnionego REST Api.

3.3 Implementacja wersji opartej na Event Sourcing’u

Coraz bardziej modnym podejściem staje się oparcie aplikacji o Event Sourcing. Z autopsji autora tekstu i jego kariery programistycznej wynika, że coraz częściej

wybierane jest właśnie to podejście, zwłaszcza w aplikacjach bankowych, gdzie każda transakcja musi zostać zapisana dla zachowania całej historii aplikacji.

Nie dziwi też fakt, iż takie rozwiązania stają się coraz bardziej popularne, zwłaszcza, że jesteśmy w czasach rozkwitu technologii opartych o Blockchain. Idea zapisywania historii wszystkich wydarzeń jest zbliżona do Blockchain.

Event Sourcing z natury opiera się na mikroserwisach i tak też jest w wypadku implementacji obranej domeny problemowej.

3.3.1 Event Sourcing w Apache Kafce

Event Sourcing został zaimplementowany za pomocą Apache Kafka. Amerykańska firma Confluent, najbliższy partner i firma konsultingowa technologii Apache Kafka, poleca użycie tej technologii do Event Sourcing'u, pisząc:

„Event Sourcing i Apache Kafka są powiązane. Event Sourcing obejmuje utrzymywanie niezmiennych sekwencji zdarzeń, które mogą subskrybować wiele aplikacji. Kafka to wysokowydajny, o niskiej latencji, skalowalny i trwały dziennik, który jest używany przez tysiące firm na całym świecie i jest testowany na skalę bitewną. W związku z tym Kafka jest naturalnym kręgosłupem do przechowywania wydarzeń przy przechodzeniu do architektury aplikacji opartej na pozyskiwaniu zdarzeń.” [1]

Persystencja wydarzeń

Wszystkie wydarzenia w aplikacji zostają zapisane przez Kafkę do plików systemowych, dzięki czemu nie ma potrzeby integracji z jakąkolwiek inną bazą danych jak np. w przypadku aplikacji monolitycznej.

Wydarzenie jako Order

Każde wydarzenie w aplikacji zostało przyporządkowane danej akcji i typowi wydarzenia, które może się powieść lub nie zostać zrealizowane. Dlatego każde wydarzenie jest jednocześnie traktowane jako swoiste zamówienie (ang. order), które ma swój identyfikator UUID.

Zobrazuje to przykład kupna karty. Z jednej strony stan danych zostanie zmieniony podczas realizacji zamówienia, a z drugiej możemy sprawdzić jako kupujący, czy wszystko powiodło się. Sprzedający mógł wycofać w międzyczasie kartę z systemu i zamówienie mogło zostać przerwane, o czym kupujący zostanie powiadomiony jak tylko wysłane zostanie zapytanie o dany „order”.

Każde wydarzenie ma strukturę obiektową podzieloną na nagłówek i główne ciało. W nagłówku znajduje się informacja nt. akcji jaką ma wydarzenie wykonać i jej typ. W ciele głównym wydarzenia znajdują się szczegóły zamówienia, gdzie np. przy tworzeniu nowej karty znajdują się tu jej wszystkie potrzebne szczegóły.

Zdefiniowano dwa typy wydarzeń, tj. zamówienia dot. kart i marek.

Wszystkie możliwe akcje zamówień dot. marek to:

- Tworzenie marki
- Aktualizacja marki
- Usunięcie marki

Lista akcji zamówień dot. kart:

- Tworzenie karty
- Aktualizacja karty
- Usunięcie karty

-
- Walidacja karty
 - Podsumowanie zamówienia karty przed płatnością
 - Opłacenie zamówionej karty
 - Wysłanie karty

Maszyna stanów

Dzięki traktowaniu każdego wydarzenia jako zamówienia posiadającego akcję i typ, możemy sprecyzować stan w jakim znajduje się aktualnie dana encja przed i po realizacji zamówienia. Jest to typowy automat skończony (ang. finite state machine), czyli abstrakcyjny, matematyczny, iteracyjny model zachowania systemu dynamicznego oparty na tablicy dyskretnych przejść między jego kolejnymi stanami. [5]

Posiadanie informacji o stanie encji pozwala na wprowadzenie logiki do walidacji akcji, które są wykonywane na encji. Dobrze zobrazuje to sytuacja w której karta nie może zostać kupiona, jeżeli nie jest zweryfikowana uprzednio przez administratora. Innym przykładem byłaby sytuacja, gdzie karta została już kupiona. W tej sytuacji możliwy następujący stan to jedynie wysyłka karty. Karta na tym etapie nie może już zostać usunięta ani kupiona przez innego użytkownika.

3.3.2 Podział na mikroserwisy wg wzorca CQRS

Aplikacja została podzielona na trzy mikroserwisy zgodnie z założeniami wzorca CQRS. Jeden mikroservis odpowiedzialny jest za dokonywanie zmian w systemie, a inny za czytanie stanu danych. Oba posiadają dostępne REST-owe endpointy HTTP. Trzecim mikroserwisem jest silnik zamówień, nie posiadający dostępnego REST Api, a który posiada logikę biznesową i jest implementacją wspomnianej już maszyny stanów.

-
- „apiCommand” posiadający wystawiony REST Api, przeznaczony do wysyłania komend (ang. command), dokonujących zmian w danych aplikacji,
 - „apiQuery” posiadający wystawiony REST Api, przeznaczony do wysyłania zapytań (ang. query), bez możliwości edycji danych,
 - „orderComponent” odpowiedzialny za przyjmowanie zamówień (ang. order) od apiCommand i stałe przetwarzanie strumieni danych, które następnie mogą być odpytywane przez apiQuery.

3.3.3 Połączenie z Apache Kafka

Wszystkie wymienione wyżej mikroserwisy komunikują się między sobą za pomocą połączenia z serwerem Apache Kafka, wspólnym dla każdej aplikacji. Kafka jest wykorzystana jednocześnie jako baza danych i jako audyt wszystkich wydarzeń, zaszłych w systemie. Mikroservis orderComponent jako jedyny komunikuje się tylko i wyłącznie poprzez Kafkę z pozostałymi mikroservisami w środowisku. Jednakże jest to komunikacja jednostronna, gdzie zarówno apiCommand jak i apiQuery wysyłają wydarzenia dla orderComponent, u którego nie ma potrzeby wysyłki odpowiedzi lub innych zapytań poprzez wydarzenia w drugą stronę.

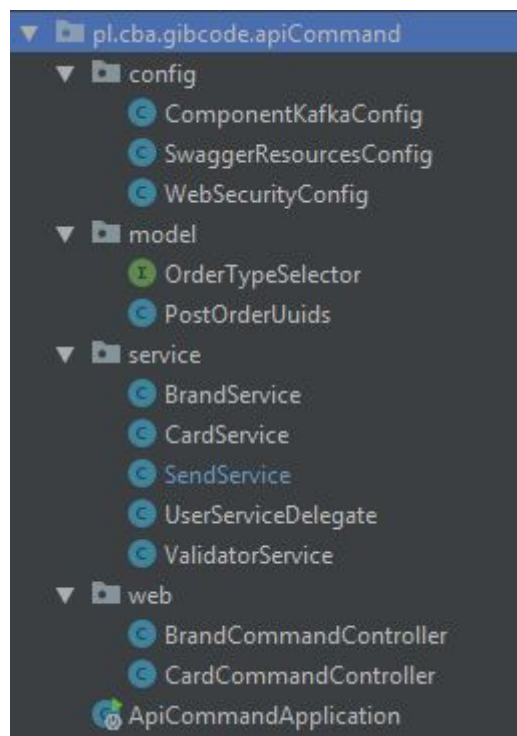
3.4 Struktura aplikacji apiCommand

Aplikacja apiCommand odpowiada za interakcję z interfejsem użytkownika poprzez przyjmowanie zapytań HTTP. Wystawiony jest własny REST Api. Po przyjęciu zapytania HTTP, po uprzednim zwalidowaniu, zostaje stworzone wydarzenie w aplikacji i wysyłane do serwera Apache Kafka. Każde takie wydarzenie jest odbierane przez orderComponent. Zgodnie z ideą CQRS, ten mikroservis odpowiada za wysyłanie komend, dokonujących zmian w danych całego systemu.

3.4.1 Organizacja kodu

Każdy z trzech mikroserwisów posiada podobną strukturę, gdzie:

- „config” zawiera wszystkie pliki konfiguracyjne,
- „model” zawiera wszystkie modele obiektów używane tylko i wyłącznie przez ten mikroserwis. Pozostałe modele wspólne są w osobnym module `modelLibrary`,
- „service” zawiera tzw. serwisy, czyli klasy posiadające logikę biznesową np. usuwanie karty z systemu,
- "web" to pakiet z kontrolerami Springowymi udostępniającymi zasoby przez REST Api.



Rysunek 3.9: Organizacja kodu w apiCommand. Źródło: opracowanie własne

3.4.2 Walidacja zapytań

Każde zapytanie, które zostaje odebrane przez mikroserwis zostaje zwalidowane, zanim zostanie stworzone wydarzenie przekazywane do brokera Kafka. Walidacja zostaje zrealizowana poprzez wyciągnięcie informacji z Apache Kafka o danej encji karty lub marki. Zapytanie skierowane do Apache Kafka jest oparte na wydobyciu danego obiektu na podstawie klucza encji, jakim jest jej ciąg znaków UUID. Realizacja takiego zapytania jest umożliwiona dzięki wykorzystaniu API udostępnionego przez Kafka Streams. Został użyty `ReadOnlyKeyValueStore` z pakietu `org.apache.kafka.streams.state`, który umożliwia zapytanie o wydarzenia z serwera Kafki na podstawie ich kluczy, zakresu kluczy jeżeli są one tworzone w sposób sekwencyjny a także prośba o wszystkie wydarzenia i ich szacowana ilość.

Dzięki dualności strumieni i tabel w Kafce, co zostało opisanej w drugim rozdziale, `ReadOnlyKeyValueStore` automatycznie przygotowuje tabelę przedstawiającą aktualny a zarazem ostatni stan encji, którą możemy dostać za pomocą klucza jaki został nadany przy tworzeniu wydarzenia.

```
package org.apache.kafka.streams.state;

public interface ReadOnlyKeyValueStore<K, V> {
    V get(K var1);

    KeyValueIterator<K, V> range(K var1, K var2);

    KeyValueIterator<K, V> all();

    long approximateNumEntries();
}
```

Rysunek 3.10: `ReadOnlyKeyValueStore`. Źródło: opracowanie własne

Cała walidacja zostaje przeprowadzona w klasie `ValidatorService` w paczce `pl.cba.gibcode.apiCommand.service`.

```

private final Provider<ReadOnlyKeyValueStore<String, Brand>> brandStoreProvider;
private final Provider<ReadOnlyKeyValueStore<String, Card>> cardStoreProvider;

public ValidatorService(
    Provider<ReadOnlyKeyValueStore<String, Brand>> brandStoreProvider,
    Provider<ReadOnlyKeyValueStore<String, Card>> cardStoreProvider) {
    this.brandStoreProvider = brandStoreProvider;
    this.cardStoreProvider = cardStoreProvider;
}

public void validateBrand(String entityId) {
    Brand brand = brandStoreProvider.get().get(entityId);
    if (brand == null || brand.getDeleted()) {
        throw new BusinessException(String.format("Brand not found with entityId %s", entityId));
    }
}

public void validateCard(String entityId) {
    Card card = cardStoreProvider.get().get(entityId);
    if (card == null || card.getDeleted()) {
        throw new BusinessException(String.format("Card not found with entityId %s", entityId));
    }
}

public void validateNewBrandCreation(String entityId) {
    Brand brand = brandStoreProvider.get().get(entityId);
    if (nonNull(brand) && !brand.getDeleted())
        throw new BusinessException(String.format("Brand already found in Order with entityId %s", entityId));
}

```

Rysunek 3.11: ValidatorService odpowiedzialny za walidację nowych wydarzeń przed wysłaniem do Kafki. Źródło: opracowanie własne

3.4.3 Sposób użycia Kafka Producer API

Po zwalidowaniu żądania pochodzącego z REST Api, apiCommand komunikuje się z brokerem za pomocą Kafka Producer API. Wiadomość, traktowana jako tworzące się nowe wydarzenie, zostaje wysłana, co jest równoznaczne z późniejszym stworzeniem zamówienia w systemie. Zamówienie zostaje wysłane przez Kafkę na temat „orderComponentEvent”, który subskrybowany jest przez mikroserwis o nazwie orderComponent. Klasa SendService z pakietu pl.cba.gibcode.apiCommand.service odpowiedzialna jest za przygotowanie wydarzenia i wysłanie go przez Kafka Producer API. Klasa ma metody postOrder do tworzenia nowego zamówienia i putOrder do aktualizacji istniejącej karty lub marki.


```

@Service
@RequiredArgsConstructor
public class SendService {

    private static final Logger logger = LoggerFactory.getLogger(SendService.class);

    private final Producer<String, OrderComponentEvent> producer;

    public PostOrderUids postOrder(String entityId, OrderComponentBody body, ActionEnum actionEnum, Long userId, OrderType orderType) {
        var orderUuid = UUID.randomUUID();
        sendMessage(orderUuid, entityId, body, actionEnum, userId, orderType);
        return new PostOrderUids(entityId, orderUuid);
    }

    public UUID putOrder(String entityId, OrderComponentBody body, ActionEnum actionEnum, Long userId, OrderType orderType) {
        var orderUuid = UUID.randomUUID();
        sendMessage(orderUuid, entityId, body, actionEnum, userId, orderType);
        return orderUuid;
    }

    private void sendMessage(UUID orderUuid, String entityId, OrderComponentBody body, ActionEnum actionEnum, Long userId, OrderType type) {
        try {
            var message = ImmutableOrderComponentEvent.builder().body(body)
                .header(ImmutableOrderComponentHeader.builder().action(actionEnum)
                    .orderUuid(orderUuid)
                    .entityId(entityId)
                    .userId(userId)
                    .creationTimestamp(Instant.now())
                    .type(type)
                    .build()).build();
            Future<RecordMetadata> send = producer.send(
                new ProducerRecord<>("topic:orderComponentEvent", entityId,message));
            send.get(Timeout.of(5, TimeUnit.SECONDS));
            logger.info("Sent message {}", message);
        } catch (InterruptedException | ExecutionException | TimeoutException e) {
            throw new ServiceUnavailableException("asdf");
        }
    }
}

```

Rysunek 3.12: SendService. Źródło: opracowanie własne

3.4.4 REST'owe Endpointy

Każda funkcjonalność, tak jak w przypadku aplikacji w wersji monolitycznej, jest udostępniana przez REST Api i interfejs Swagger. Ten mikroservis udostępnia wszystkie funkcjonalności związane z operacjami zmieniającymi stan danych w aplikacji.

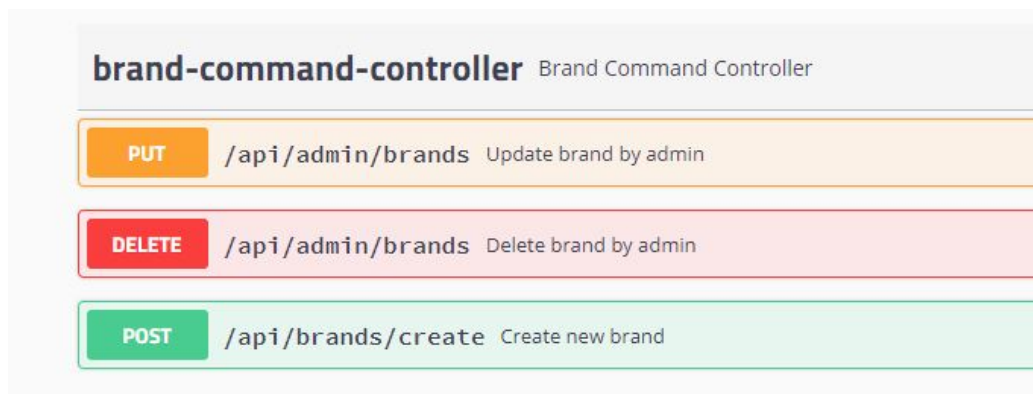
| | |
|---------------------------------|--------------------------|
| brand-command-controller | Brand Command Controller |
| card-command-controller | Card Command Controller |

Rysunek 3.13: Lista kontrolerów w mikroservisie apiCommand. Źródło: opracowanie własne

Kontroler Marki

Kontroler wykonujący operacje komend na markach składa się z endpointów:

- admin/brands jako HTTP PUT - aktualizacja marki przez administratora,
- admin/brands jako HTTP DELETE - usuwanie marki przez administratora,
- brands/create - tworzenie nowej marki,



Rysunek 3.14: Endpointy kontrolera marki. Źródło: opracowanie własne

Kontroler Kart

Kontroler wykonujący operacje komend na kartach składa się z endpointów:

- admin/cards jako HTTP DELETE - usuwanie karty z systemu przez administratora,
- admin/cards/validate - walidowanie karty przez administratora,
- buyer/checkout - dokonanie zamówienia przez kupującego,
- buyer/pay - dokonanie płatności przez kupującego,
- seller/card jako HTTP POST - tworzenie karty przez sprzedającego,

- seller/card jako HTTP PUT - aktualizacja karty przez sprzedającego,
- seller/card jako HTTP DELETE - usuwanie karty przez sprzedającego,
- seller/card/send - wysyłanie karty przez sprzedającego,

| card-command-controller Card Command Controller | |
|---|---|
| DELETE | /api/admin/cards Delete card for admin |
| POST | /api/admin/cards/validate Validate card for admin |
| POST | /api/buyer/checkout Checkout a card as a buyer |
| POST | /api/buyer/pay Pay a card as a buyer |
| POST | /api/seller/card Create a card as a seller |
| PUT | /api/seller/card Update a card as a seller |
| DELETE | /api/seller/card Delete a card as a seller |
| PUT | /api/seller/card/send Send the card |

Rysunek 3.15: Endpointy kontrolera kart. Źródło: opracowanie własne

3.4.5 Uruchomienie aplikacji

Aby uruchomić aplikację trzeba mieć zainstalowany, tak jak w przypadku aplikacji monolitycznej, Apache Maven w wersji 3. Następnie serwer Kafki musi zostać włączony i być dostępny pod adresem podanym w pliku application.properties, a domyślnie jest nim localhost:9092. Użyte zostało w tym celu Kafka Lenses ze

strony `https://docs.lenses.io/install_setup/download.html` jako obraz dockerowy. Po tym, w głównym folderze z kodem źródłowym należy podać komendę w terminalu „`mvn spring-boot:run`”. Ta komenda spowoduje, że aplikacja zostanie zbudowana i wystartuje na wbudowanym serwerze aplikacyjnym Tomcat na porcie 1000.

Aplikacja po uruchomieniu powinna być dostępna na `http://localhost:1000/ala-karta/swagger-ui.html`. Jest to strona z interfejsem udostępnionym przez Swaggera, gdzie można wykonywać interaktywne zapytanie do udostępnionego REST Api.

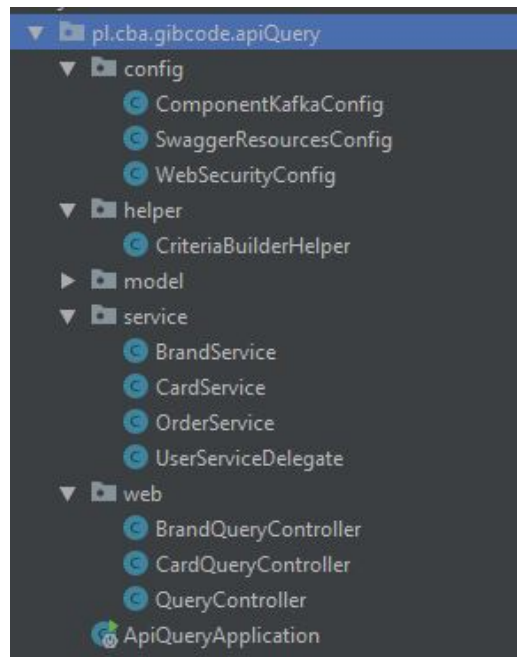
3.5 Struktura aplikacji apiQuery

Aplikacja apiQuery, podobnie jak apiCommand, odpowiada za interakcję z interfejsem użytkownika poprzez przyjmowanie zapytań HTTP. Wystawiony jest własny REST Api. Ten mikroserwis odpowiada tylko i wyłącznie za odczytywanie stanu danych w systemie. Posiada definicję wszystkich operacji, które nie zmieniają stanu encji ani nie wykonują żadnych akcji na nich poza zczytywaniem. Po przyjęciu żądania HTTP aplikacja korzysta z ReadOnlyKeyValueStore w celu wyciągania informacji o encjach. Udostępniono możliwość filtrowania wyników a także podział danych na strony, gdzie zwracana jest tylko strona odpowiedzi (domyślnie 20 wpisów), co zmniejsza obciążenie sieci i poprawia szybkość przetwarzania żądania, a także pomaga w dogodnym zaprojektowaniu stronicowania wyników w interfejsie użytkownika. Dane, które można zażądać przez apiQuery dotyczą stanu encji kart i marek a także jest możliwość zapytania o dane zamówienie (np. akcja usunięcia karty) czy zostało przeprowadzone pozytywnie i z jaką encją jest powiązane.

3.5.1 Organizacja kodu

Kod, tak jak w przypadku apiCommand ma logiczną strukturę składającą się z:

- „config” zawiera wszystkie pliki konfiguracyjne,
- „model” zawiera wszystkie modele obiektów używane tylko i wyłącznie przez ten mikroserwis. Pozostałe modele wspólne są w osobnym module modelLibrary,
- „helper” posiada klasę CriteriaBuilderHelper pomagającą tworzyć filtrowanie kart i marek,
- „service” zawiera tzw. serwisy, czyli klasy posiadające logikę biznesową np. usuwanie karty z systemu,
- "web" to pakiet z kontrolerami Springowymi udostępniającymi zasoby przez REST Api.



Rysunek 3.16: Organizacja kodu w apiQuery. Źródło: opracowanie własne

3.5.2 REST'owe endpointy

Każda funkcjonalność, tak jak w przypadku aplikacji w wersji monolitycznej, jest udostępniana przez REST Api i interfejs Swagger. Ten mikroserwis udostępnia wszystkie funkcjonalności związane z operacjami tylko i wyłącznie odczytującymi stan danych w aplikacji.

Kontroler marek

Kontroler wykonujący zapytania zczytujące stan encji marek. Posiada możliwość filtrowania i stronicowania wyników.

- admin/brands - lista wszystkich marek, w tym marek nieposiadających żadnej karty, czyli nieaktywnych w sprzedaży,
- brands/list - lista wszystkich aktywnych marek, których karty są do sprzedaży.

Kontroler kart

Kontroler wykonujący zapytania zczytujące stan encji kart. Posiada możliwość filtrowania i stronicowania wyników.

- admin/cards - lista wszystkich kart, w tym kart niezwalidowanych przez administatora, usuniętych lub uprzednio sprzedanych a także wysłanych,
- cards/list - lista wszystkich aktywnych kart będących dostępnymi w sprzedaży.

Query kontroler

Ogólny kontroler posiadający endpointy do sprawdzania stanów transakcji wykonywanych w systemie, a także informacji o aktualnym stanie danego zamówienia związanego z konkretnym wydarzeniem w systemie. Każda transakcja rozpoczyna się w momencie potwierdzenia złożenia zamówienia karty (ang. checkout) przez kupującego.

- admin/transactions - lista wszystkich transakcji dla administratora,
- buyer/transactions - lista wszystkich transakcji dotycząca danego kupującego,
- orders - na podstawie orderUUID zwracane dane zamówienie (ang. order) i jego status. Mowa o zamówieniu jako wyniku wydarzenia w systemie, a nie o czynności potwierdzającej zamówienie (ang. checkout) karty,
- seller/transactions - lista wszystkich transakcji dotycząca danego sprzedającego.

| | | |
|--|--------------------------|---------------------------------|
| brand-query-controller Brand Query Controller | | |
| POST | /api/admin/brands | Get all brands for admin |
| POST | /api/brands/list | Get all brands for sale |
| card-query-controller Card Query Controller | | |
| POST | /api/admin/cards | Get all cards for admin |
| POST | /api/cards/list | Get all cards for sale |
| query-controller Query Controller | | |
| POST | /api/admin/transactions | Get all transactions for admin |
| POST | /api/buyer/transactions | Get all transactions for buyer |
| POST | /api/orders | Gets order state |
| POST | /api/seller/transactions | Get all transactions for seller |

Rysunek 3.17: Lista kontrolerów i endpointów w mikroserwisie apiQuery. Źródło: opracowanie własne

3.5.3 Uruchomienie aplikacji

Aby uruchomić aplikację trzeba mieć zainstalowany, tak jak w przypadku aplikacji monolitycznej, Apache Maven w wersji 3 i uruchomiony serwer Apache Kafka. Po tym, w głównym folderze z kodem źródłowym należy podać komendę w terminalu „mvn spring-boot:run”. Ta komenda spowoduje, że aplikacja zostanie zbudowana i wystartuje na wbudowanym serwerze aplikacyjnym Tomcat na porcie 1002.

Aplikacja po uruchomieniu powinna być dostępna na `http://localhost:1002/ala-karta/swagger-ui.html`. Jest to strona z interfejsem udostępnionym przez Swaggera, gdzie można wykonywać interkaktywne zapytanie do udostępnionego REST Api.

3.6 Struktura aplikacji orderComponent

Mikroserwis orderComponent jest odpowiedzialny za przetwarzanie wydarzeń i tworzenie na ich podstawie zamówień w systemie (ang. order) związanych z daną akcją przesyłaną w nagłówku wiadomości. Aplikacja komunikuje się tylko i wyłącznie za pomocą wydarzeń poprzez broker Kafki. Po otrzymaniu wiadomości poprzez strumień Kafki dokonywane są wszystkie operacje na encjach, a także ich walidacja. Dla przykładu karta już sprzedana nie może zostać sprzedana ponownie. Aplikacja jest implementacją maszyny stanów, gdzie logika biznesowa jest oparta o możliwe stany encji w jakich mogą się znajdować. Jest weryfikowana, czy akcja na danej encji jest wykonywalna na podstawie definicji możliwych przejść między stanami encji w połączeniu z listą zezwolonych akcji.

Implementacja maszyny stanów znajduje się w klasie StateMachine w pakiecie `pl.cba.gibcode.orderComponent.command.service`.

```

public class StateMachine {

    public static List<ActionEnum> getPossibleTransitions(EntityStateEnum entityStateEnum) {
        switch(entityStateEnum) {
            case BRAND_DELETED:
                return List.of(CREATE_BRAND);
            case BRAND_CREATED:
                return List.of(UPDATE_BRAND, DELETE_BRAND);
            case BRAND_UPDATED:
                return List.of(UPDATE_BRAND, DELETE_BRAND);
            case CARD_DELETED:
                return List.of(CREATE_BRAND);
            case CARD_SENT:
                return List.of();
            case CARD_PAID:
                return List.of(SEND_CARD);
            case CARD_CHECKED_OUT:
                return List.of(PAY_CARD);
            case CARD_UPDATED:
                return List.of(UPDATE_CARD, VALIDATE_CARD, DELETE_CARD);
            case CARD_CREATED:
                return List.of(UPDATE_CARD, VALIDATE_CARD, DELETE_CARD);
            case CARD_VALIDATED:
                return List.of(UPDATE_CARD, CHECKOUT_CARD, DELETE_CARD);
            default:
                return List.of();
        }
    }
}

```

Rysunek 3.18: Maszyna stanów. Źródło: opracowanie własne

Kolejną ważną cechą tej aplikacji jest ekstensywne wykorzystanie Kafka Streams API co jest opisane w kolejnej sekcji.

3.6.1 Organizacja procesorów strumieni poprzez Streams API

Główny rdzeń całego systemu opiera się na wykorzystaniu Kafka Streams API. Logika przetwarzania strumieni nie jest skomplikowana i znajduje się w dwóch klasach, zgodnie z założeniem CQRS gdzie jedna klasa to gałąź przetwarzania strumieni dla komend a druga dla kwerend tylko do odczytu.

Topologia komend

Cała definicja topologii strumieni opartych o Kafka Streams API dotyczących komend w systemie znajduje się w StreamingConfig w pakiecie pl.cba.gibcode.orderComponent.command

Aplikacja nasłuchuje na temat „orderComponentTopic” i reaguje przy otrzymaniu wiadomości. Każda wiadomość jest traktowana jako wydarzenie w systemie.

```
KStream<String, OrderComponentEvent> orderComponentStream = streamsBuilder
    .stream(topic: "orderComponentEvent", CustomJsonSerde.consume(OrderComponentEvent.class));
```

Rysunek 3.19: Subskrypcja tematu orderEventTopic jako początek przetwarzania strumienia. Źródło: opracowanie własne

Wiadomość, którą otrzymujemy w strumieniu transformujemy przez logikę biznesową, gdzie rezultatem tego przetwarzania jest nowa informacja, którą wysyłamy na tematy „order”, „card” i „brand”. Te tematy zawierają wszystkie informacje o encjach z nimi powiązanymi. Temat „order” to zamówienie w systemie (nie mówimy o potwierdzenia zamówienia karty tylko o zamówieniu jako rezultacie jakiegokolwiek wydarzenia w aplikacji). Temat „card” dla kart i „brand” dla marek.

```
var transformedStream =
    orderComponentStream
        .peek((key, value) -> logger.info("orderComponentEvent: {}: {}", key, value))
        .selectKey((k, v) -> v.getHeader().getEntityId())
        .transformValues(Transformer::new,
            ORDER_COMPONENT_EVENT_STORE, ORDER_STORE, BRAND_STORE, CARD_STORE);

var orderStream = transformedStream.mapValues((s, processingWrapper) -> processingWrapper.getResponse())
    .selectKey((s, order) -> order.getOrderUuid().toString())
    .peek((key, value) -> logger.info("order: {}: {}", key, value));
orderStream.to(S: "order", CustomJsonSerde.produce(Order.class));

var brandStream = transformedStream.mapValues((s, processingWrapper) -> processingWrapper.getBrand())
    .filter((s, brand) -> nonNull(brand))
    .peek((key, value) -> logger.info("brand: {}: {}", key, value));
brandStream.to(S: "brand", CustomJsonSerde.produce(Brand.class));

var cardStream = transformedStream.mapValues
    ((s, processingWrapper) -> processingWrapper.getCard())
    .filter((s, card) -> nonNull(card))
    .peek((key, value) -> logger.info("card: {}: {}", key, value));
cardStream.to(S: "card", CustomJsonSerde.produce(Card.class));
```

Rysunek 3.20: Transformacja i wysyłanie informacji na pojedynczy temat per encja. Źródło: opracowanie własne

Transformacja przy użyciu zasad opisanych w logice biznesowej znajduje się

w prywatnej klasie Transformer w obiekcie StreamingConfig. Jest tam m.in. walidowane wydarzenie w powiązaniu z daną akcją i typem, opierające się na logice maszyny stanów.

```
if (nonNull(orderStore.get(key)) && !StateMachine
    .getPossibleTransitions(orderStore.get(key).getEntityState())
    .contains(event.getHeader().getAction())) {
    var possibleTransitions = StateMachine.getPossibleTransitions(orderStore.get(key).getEntityState());
    processingWrapper.setResponse(ModelBuilderUtils.buildFailureOrder(event,
        reason: "Event transition incorrect:" + orderStore.get(key).getEntityState() + " -> " + event
            .getHeader().getAction() + ", possible are: " + possibleTransitions,
            orderStore.get(key).getEntityState()));
    return processingWrapper;
}
```

Rysunek 3.21: Walidacja możliwych przejść na podstawie maszyny stanów. Źródło: opracowanie własne

Ostatnim krokiem w przetwarzaniu strumieni jest przygotowanie informacji dostępnych dla mikroserwisu apiQuery. Dane, które są przygotowane dla kwerend, są obliczane na bieżąco przy każdym wydarzeniu. Wszystkie inne tematy znajdujące się w topologii Kafki zostają również wzbudzone i przetwarzane ponownie. Jednym z przykładów użycia jest zapytanie o wszystkie karty na podstawie marki. Rezultat jest zapisywany na osobnym temacie, który jest w tym wypadku nazwany „queryCardsByBrand”.

```

var queryCardsByBrand = brandKTable
    .leftJoin(
        cardAggregation,
        (brand, cardWrapperAggregator) -> {
            logger.info("Left join {}, {}", brand, cardWrapperAggregator);
            var cardsByBrand = ImmutableQueryCardsByBrand.builder().isDeleted(false);
            if(brand.getDeleted()) {
                cardsByBrand.isAvailable(false);
                cardsByBrand.isDeleted(true);
                return (QueryCardsByBrand) cardsByBrand.build();
            }
            if(isNull(cardWrapperAggregator) || cardWrapperAggregator.getEntries().isEmpty()) {
                cardsByBrand.isAvailable(false);
            } else {
                cardsByBrand.isAvailable(true);
                cardsByBrand.cards(cardWrapperAggregator.getEntries().values().stream()
                    .map(CardByBrandAggregationEntry::getCard).collect(
                        Collectors.toList()));
                cardsByBrand.addAllPriceRanges(
                    cardWrapperAggregator
                        .getEntries().map<String, CardByBrandAggregationEntry>
                        .values().collection<CardByBrandAggregationEntry>
                        .stream().map(CardByBrandAggregationEntry::getPriceRangeEnum).distinct().collect(
                            Collectors.toList()));
            }
            return (QueryCardsByBrand) cardsByBrand.build();
        }).toStream();

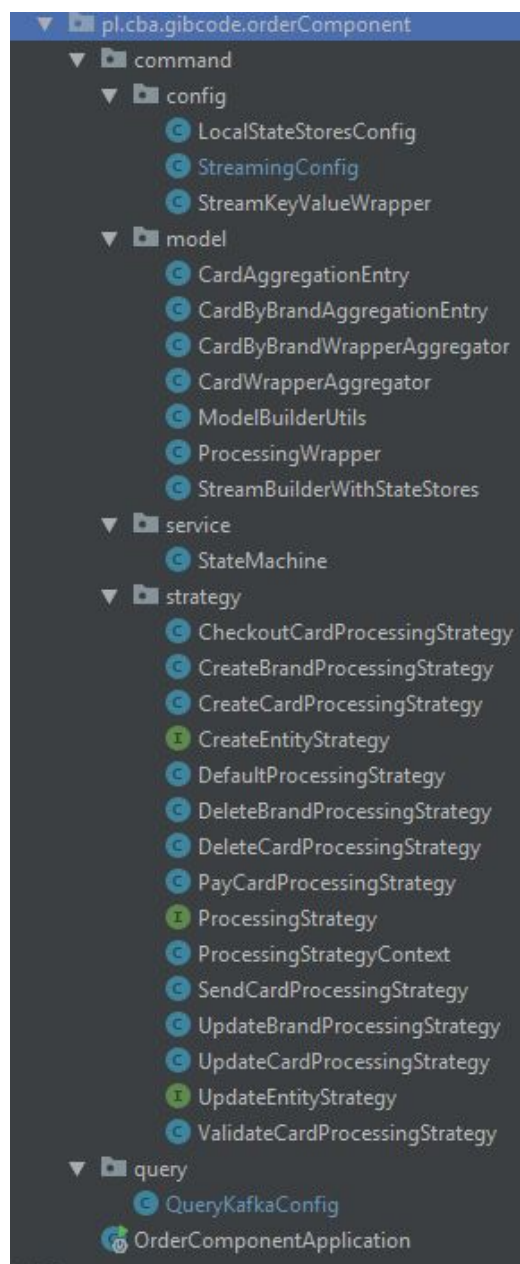
queryCardsByBrand.to( S: "queryCardsByBrand", CustomJsonSerde.produce(QueryCardsByBrand.class));

```

Rysunek 3.22: Agregacja listy kart na podstawie nazwy marki. Źródło: opracowanie własne

3.6.2 Organizacja kodu

Podział kodu jest bardzo podobny do pozostałych mikroserwisów. Zgodnie z ideą CQRS pakiety zostały podzielone na część „command” i część „query”. Dodatkowym pakietem jest „strategy” gdzie zdefiniowane są wszystkie możliwe operacje w systemie.



Rysunek 3.23: Organizacja kodu w orderComponent. Źródło: opracowanie własne

3.6.3 Uruchomienie aplikacji

Aby uruchomić aplikację trzeba mieć zainstalowany Apache Maven w wersji 3 i uruchomiony serwer Apache Kafka.

Po tym, w głównym folderze z kodem źródłowym należy podać komendę w terminalu „mvn spring-boot:run”. Ta komenda spowoduje, że aplikacja zostanie zbudowana i wystartuje na wbudowanym serwerze aplikacyjnym Tomcat na porcie 1001. Aplikacja nie ma żadnego interfejsu ani REST Api, w przeciwieństwie do apiQuery czy apiCommand.

Włączenie aplikacji po raz pierwszy przy połączeniu z brokerem Kafki wywołuje utworzenie tematów automatycznie.

| Topic |
|---------------------|
| brandsByCategories |
| buyerOrder |
| card |
| sellerOrder |
| queryBrand |
| brand |
| queryCardsByBrand |
| order |
| joinedCardWithOrder |

Rysunek 3.24: Lista tematów w interfejsie Kafka Lenses. Źródło: opracowanie własne

Rozdział 4

Wnioski w odniesieniu do stworzonych aplikacji

W tym rozdziale autor tekstu prezentuje własne wnioski w odniesieniu do stworzonych aplikacji przedstawiających dwa przeciwstawne podejścia do architektury systemów informatycznych. Pierwszym podejściem jest użycie struktury monolitycznej, czyli takiej gdzie konstrukcja oprogramowania oparta jest na jednym, wspólnym i współdzielonym modelu danych. Przeciwstawnym jest wzorzec Event Sourcing w połączeniu ze wzorcem CQRS i podejściem mikroserwisowym. Co jest warte uwagi, to to, iż każdy z obu sposobów na tworzenie architektury systemów jest poprawny i oba spotykane są na co dzień w pracy programisty. Jedynie kontekst ich użycia jest kluczowy. Przez kontekst rozumiany jest typ aplikacji, skomplikowość domeny biznesowej, możliwości i wymagania projektowe. Dlatego autor pracy stara się przedstawić własne wnioski kiedy i w jakim kontekście dany wybór jest właściwy. W dalszej części rozdziału zostaje porównana większość cech jakości oprogramowania obu podejść. Ażeby jeszcze wyostrzyć kontrast różnic między oba podejściami, użyty został model anemiczny przy tworzeniu aplikacji monolitycznej.

4.1 Wybór podejścia

Na początkowym etapie tworzenia systemu informatycznego najważniejszym wyborem jest decyzja o podejściu architektonicznym. Decyzja pociąga za sobą bardzo dużo konsekwencji projektowych. Zmienia się czas realizacji projektu, koszty, a także inne wyzwania jeżeli chodzi o aspekty związane z bezpieczeństwem rozwiązania. Znacząco różnie może też wyglądać późniejszy etap wdrożenia aplikacji i wymagania co do potrzebnych zasobów.

4.2 Poziom skomplikowania kodu a domeny

Pierwszą rzeczą nad którą warto się skupić jest poziom skomplikowania kodu w obu podejściach. Zdecydowanie mniej kodu źródłowego, mniej zewnętrznych zależności jest przy implementowaniu aplikacji monolitycznej.

Aplikację w wydaniu Event Sourcing’owym autor pracy pisał o wiele dłużej i wymagała stworzenie o wiele więcej kodu w pierwszej fazie implementacji. Wysznuwany został pewien empiryczny wniosek, że mimo, iż zwłaszcza na początku jest o wiele więcej kodu źródłowego do zaimplementowania, to potem bardzo logicznie i łatwo można system rozwijać o nowe funkcjonalności, co nie jest takie proste przy aplikacji monolitycznej w zaawansowanym stadium tworzenia systemu.

Aplikacje oparte na mikroserwisach w połączeniu z Event Sourcing’iem są bardziej naturalnym wyborem przy bardzo skomplikowanej domenie biznesowej. Przy prostym sklepie internetowym gdzie domena jest bardzo prosta a encji jest niewiele, użycie Event Sourcing’u jest niekoniecznie optymalnym wyborem, bo po pierwsze domena nie będzie się w żaden sposób rozwijała i nie jest domeną nową i nieznaną. Implementacja modelu bazodanowego sklepu internetowego jest bardzo prosta nawet do znalezienia w Internecie i wystarczy ją zmodyfikować

i zaaplikować do własnych potrzeb.

Tym bardziej skomplikowana jest domena tym większa szansa, że powinniśmy ją podzielić na mikroserwisy i coraz bardziej optymalnym staje się również użycie Event Sourcing'u jako sposobu na persystencję danych i komunikację między aplikacjami.

Inwestycja w Event Sourcing i szkolenie w tej dziedzinie zespołu programistów zacznie się zwracać przy skomplikowanej domenie biznesowej, jeżeli chodzi o poziom skomplikowania kodu.

4.2.1 Wymagania systemowe

Oczywistością jest, iż Event Sourcing będzie wymagał o wiele więcej zasobów sprzętowych ze względu na jego charakter mikroserwisowy i potrzebę działania brokera aplikacji Apache Kafka. Kafka zapisuje wszystkie dane w systemie plików i potrzebne jest o wiele więcej miejsca na dysku twardym niż w przypadku drugiego podejścia. Mamy o wiele więcej danych, bo wszystkie zmiany w systemie są zapisane i możliwe jest ustawienie strategii nieusuwania żadnych historycznych danych, przez co bardzo szybko rośnie ilość wymaganego miejsca na dysku twardym. Dodatkowo Kafka opiera się na bazie danych RocksDB zapisująca dane w pamięci, przez co wymagania co do pamięci RAM również wzrastają z napływem coraz więcej ilości danych. Podczas implementacji aplikacji autor pracy zauważył, że bardzo dużo danych zostało zduplikowanych wielokrotnie. Informacja, która została wysłana jako wiadomość na broker Kafki na temat `orderComponentEvent` zostaje potem przetransformowana i przeniesiona na inne tematy. Z taką sytuacją autor pracy spotyka się również w pracy zawodowej przy implementacji tego typu systemów, co uważa za naturalną konsekwencję użycia Kafka Streams API, gdzie zasadność używania strumieni jest wtedy kiedy strumień przekazuje wiadomość na kolejny strumień, znajdujący swój początek na

innym temacie wiadomości.

Jeżeli mamy ograniczone zasoby sprzętowe to zdecydowanie bezpieczniejszym rozwiązaniem jest użycie podejścia monolitycznego, tym bardziej, że przy wdrożeniu produkcyjnym Kafkę zaleca się konfigurować w klastrze i o wielu wątkach, co zwiększa jeszcze bardziej wymogi sprzętowe. Trzeba jednak mieć na uwadze, żeby nie wpaść w pułapkę zastawioną przez Anemic Model.

4.2.2 Wymagania biznesowe

Event Sourcing ze swojej natury implikuje zapis większej ilości danych, co w niektórych branżach i ich specyficznych domenach biznesowych jest wymagane. Posiadanie historii wszystkich zdarzeń w systemie pozwala na wysyłanie ich do hurtowni danych i tworzenie na ich podstawie statystyk i prognoz potrzebnych w funkcjonowaniu danego biznesu. Atawistyczna możliwość zapisywania wszystkich wiadomości w systemie pociąga za sobą również możliwości podniesienia kontroli nad systemem jak i jego bezpieczeństwa. Można wykryć nieprawidłowości w prosty sposób. Istotne też jest, że mamy szansę na odtworzenie całego systemu lub cofnięcie się do danego momentu w czasie, co np. może być bardzo przydatne w świecie bankowości. Przy braku Event Sourcing'u jakakolwiek aktualizacja czy usunięcie danych zmienia stan aplikacji w sposób trwały, tym samym tracąc informację o wcześniejszego stanu sprzed aktualizacji czy usunięcia.

4.2.3 Wiedza programistyczna

Bardzo dużym problemem jest zmiana sposobu myślenia dla programisty przy tworzeniu aplikacji opartych o wydarzenia i Event Sourcing. Jest to zupełnie nowatorski sposób pisania aplikacji, który dla wielu, mimo wieloletniego doświadczenia w programowaniu aplikacji, jest czymś zupełnie nowym. Dlatego też potrzebna jest inwestycja dodatkowego czasu i wysiłku na wyszkolenie zespołu pro-

gramistów w tej kwestii. Jeżeli projekt jest krótki i żaden z programistów w zespole nie miał styczności ani z Event Sourcingiem ani z Apache Kafką, osobiście nie polecałbym przejścia na ten typ architektury, jako, że technologia ta ma dosyć wysoki próg wejścia i lepiej mieć w zespole mentora z tej dziedziny.

4.3 Jakość oprogramowania

Decyzje związane z wyborem podejścia i architektury systemu rzutuje na jakość oprogramowania, czyli tak naprawdę ostateczne powodzenie projektu lub porażkę.

Według międzynarodowego standardu ISO/IEC 25010:2011 jakość oprogramowania to stabilność funkcjonalna, wydajność działania, kompatybilność, użyteczność, niezawodność, bezpieczeństwo, utrzymywalność i przenośność wytworzonego oprogramowania.[26]

W świecie projektowania architektury aplikacji prężnie rozwijają się nowe podejścia, które powstały na przestrzeni ostatnich kilku lat. Czy Event Sourcing jako jeden z nich ma wysoką jakość oprogramowania?

4.3.1 Funkcjonalna stabilność

Posiadanie większej ilości mikroserwisów ma dwie strony medalu. Trzeba o wiele więcej wysiłku w zarządzaniu każdą z tych aplikacji, tym bardziej w środowisku produkcyjnym, gdzie jest działających wiele instancji jednego mikroserwisu. Jednakże, aplikacja monolityczna, jeżeli zostanie wyłączona, to system zupełnie przestaje odpowiadać, przez co wszystkie operacje w systemie jednocześnie zostaną zablokowane, w tym procesowanie aktualnych żądań. Takiej sytuacji nie ma przy zastosowaniu mikroserwisów, bowiem zawsze chociaż część komponentów będzie w stanie przetwarzać żądania.

Jednakże dopiero zastosowanie Event Sourcing'u w połączeniu z mikroserwisami daje nam jak największą funkcjonalną stabilność, bowiem jeżeli któryś z komponentów nie będzie działał, to i tak broker Kafka będzie posiadał wszystkie wiadomości, które mają być przetworzone przez dany komponent i jak awaria zostanie naprawiona, żadna informacja nie zostanie pominięta, a wszystkie zaległe wiadomości zostaną odebrane i przetworzone przez dany mikroserwis.

Niebezpieczeństwa związane z wydarzeniami

Funkcjonalna stabilność w asynchronicznych rozwiązaniach opartych o wydarzenia może zostać zagrożona, jeżeli nie będziemy świadomi pewnych niebezpieczeństw, które implikuje natura takich systemów. Autor tekstu zauważył podczas testów empirycznych, że może wystąpić sytuacja wiadomości-duplikatów w sytuacji jak użytkownik kliknie kontrolkę w interfejsie wielokrotnie. Taka sytuacja musi być obsługiwana i zaplanowana. Kolejną sytuacją jest to, że dany mikroserwis może otrzymać parę wiadomości na raz ze względu na opóźnienie lub brak dostępności mikroserwisu. Taki scenariusz może spowodować, że wydarzenia będą nieprzetworzone w kolejności w jakiej zostały dostarczone do Kafki, a na to nie mamy wpływu lub niektóre wiadomości będą operować na danych zdezaktualizowanych. Aplikacja orderComponent została zabezpieczona od takiego scenariusza. Niewykrycie takich sytuacji w testach i brak implementacji ze względu na przez brak doświadczenia z tego typu systemami może jednoznacznie pogorszyć jakość oprogramowania.

4.3.2 Niezawodność

Oba podejścia są jednakowo niezawodne, ponieważ to wszystko zależy od późniejszej konfiguracji systemu. Aplikacje oparte na Kafce jednakże wymagają o wiele czasu i specjalistycznej wiedzy, żeby system uczynić niezawodnym. Jeżeli

projekt nie posiada w zespole specjalisty od wdrażania Kafki produkcyjnie, na pewno bardziej niezawodnym będzie stworzenie aplikacji monolitycznej, której zasady wdrażania są bliższe o wiele większej liczbie osób.

4.3.3 Wydajność

Autor pracy skupił się na aspektach implementacyjnych obu aplikacji, dlatego nie przedstawia dokładnych danych porównujących wydajność zapytań dla tej samej operacji w obu systemach. Byłoby to porównanie nieobiektywne, bowiem oba systemy mają odrobną naturę. W monolicie wszystkie żądania są przetwarzane synchronicznie a w Event Sourcingu opieramy się na eventach, które z natury są asynchroniczne. Nawet bez posiadania mocnego sprzętu Kafka jest w stanie obsługiwać dane o dużej objętości. Ponadto, jest w stanie obsługiwać przepustowość tysięcy wiadomości na sekundę. Dlatego sam broker nie jest punktem, który pogarszałby wydajność. Raczej sama implementacja mikroservisów i przepustowość łącza może wpłynąć na ogólną wydajność systemu.

4.3.4 Użyteczność

Użyteczność dla użytkownika końcowego w obu podejściach jest taka sama, ponieważ autor pracy dołożył wszelkich starań, żeby funkcjonalność obu implementacji systemu była bardzo zbliżona do siebie. Dla programisty jest o wiele większy próg wejścia do wersji systemu opartego na Event Sourcing'u, ze względu na potrzebę poznania i zrozumienia zupełnie odmiennego podejścia niż w przypadku aplikacji monolitycznej.

4.3.5 Bezpieczeństwo

Bezpieczeństwo systemu jest o wiele bardziej zagrożone w przypadku architektury mikroservisowej. Każda z aplikacji powinna mieć szyfrowanie SSL podczas

wysyłki lub odbiorze wiadomości między sobą, a także enkrypcja i dekrypcja wiadomości na tematach Kafki. Również pojawia się temat autoryzacji i autentykacji danych mikroserwisów do danych tematów. Kafka natywnie wspiera różne poziomy bezpieczeństwa, ale faktem jest, że większa ilość mikroserwisów stwarza większe niebezpieczeństwo, którego czasami nie jesteśmy w stanie przewidzieć podczas implementacji systemu.

Z kolei bezpieczeństwo danych jest lepiej zapewniane przez Event Sourcing ze względu na fakt, że posiadamy więcej danych na których podstawie możemy odtworzyć cały system przy pomyłkowym zastąpieniu istniejących danych lub cofnąć się w czasie do danego momentu w przypadku nieautoryzowanej operacji w systemie lub innego ataku osób trzecich. Możemy również te dane rozumieć z perspektywy ich zmian, tworząc za pomocą Kafka Streams API interaktywne zapytania.

4.3.6 Kompatybilność

Oba podejścia są podobnie kompatybilne i wszystkie elementy danego systemu ściśle i w naturalny sposób współdziałają ze sobą. Auto pracy stwierdza, iż w obu podejściach dosyć dużym problemem jest zmiana modelu encji np. przez wprowadzenie lub usunięcie jakiegoś pola encji (mówię tu o wersjonowaniu encji). W aplikacji monolitycznej nie jest to proste do zrealizowania w sytuacji, kiedy mamy już w danej tabeli bazy danych wiele wierszy. Tak samo nie jest sytuacja ułatwiona w przypadku zmiany modelu wydarzenia w systemie opartego na Event Sourcing'u, gdzie Kafka od razu zacznie wysyłać błędy o problemach z serializacją i deserializacją danych.

4.3.7 Utrzymywalność

Utrzymywalność aplikacji monolitycznych jest na pewno mniej skomplikowana dla zespołu wsparcia. Po wdrożeniu aplikacji opartej na Event Sourcing’u, błędy jakie mogą wystąpić w danych są o wiele trudniejsze do wykrycia, bo nie jesteśmy w stanie prosto wyciągnąć informacji z serwera Kafki, tak jak w przypadku zapytań SQL w systemie opartym o bazy relacyjne. Również kwestia ręcznej podmiiany danych, jaka umożliwiona jest przez komendy SQL nie jest aplikowalna do świata opartego na wydarzeniach Kafki.

Nie ma również dobrze rozwiniętych na tę chwilę narzędzi do monitorowania systemu opartego na Kafce, których to dla aplikacji monolitycznych jest dostępnych bardzo wiele.

Musimy jednak pamiętać, że dalszy rozwój aplikacji i wgrywanie poprawek jest o wiele bardziej skomplikowane przy monolicie. Kod staje się coraz bardziej niejasny i zagnieżdżony. Modularność, jaką dostarcza podzielenie systemu na mikroserwisy, pomaga we wgrywaniu poprawek bezpośrednio do danego mikroserwisu, co czyni tę procedurę łatwą i zwinną, bo jedynie ta konkretna część systemu jest dotknięta zmianami a nie całe środowisko. Dodatkowym skomplikowaniem w utrzymaniu aplikacji, która opiera się o model anemiczny, jest jeszcze większa możliwość naruszenia innej części aplikacji, zupełnie niezwiązanej ze zmianą którą dokonujemy w kodzie. Dzieje się to ze względu na ścisłą relację między kodem a bazą danych, gdzie często nazwy metod nie są wystarczająco bliskie nazewnictwu funkcjonalności domeny biznesowej.

Testowalność rozwiązań opartych o asynchroniczne wydarzenia, z obserwacji autora pracy, jest o wiele trudniejsze, wymaga napisania o wiele większej ilości dodatkowych komponentów do testowania, opartych o protokół dostarczony przez Kafkę.

4.3.8 Przenaszalność

Kafka została zaprojektowana w taki sposób, że może być skalowalna bez przestojów poprzez dodawanie nowych węzłów serwera w klastrze. Ponadto opiera się na idei replikacji i partycjonowania, co pozwala rozszerzać instalację Kafki o kolejne serwery bez potrzeby ponownego pełnego restartowania całego brokera. Kafka Streams API wspiera wielowątkowość, co pomaga w skalowalności systemu w przypadku dodawania nowych partycji Kafki. Ta funkcjonalność wpisuje się doskonale w naturę mikroservisów, gdzie skalowalność systemu jest o wiele bardziej ułatwiona. Każdy mikroservis niezależnie można replikować lub zwiększać i zmniejszać ilość jego instancji.

W aplikacji monolitycznej jesteśmy zdani tylko i wyłącznie na skalowanie ilości instancji, gdzie zawsze cała aplikacja ze wszystkimi jej modułami funkcjonalnymi (a nie tylko jej część jak w przypadku mikroservisów) jest uruchamiana w klastrze.

Łatwość adaptacyjna mikroservisów pozwala na ich zmianę i szybkie wprowadzenie zmian tylko w obrębie danego mikroservisu, podczas gdy zrestartowanie aplikacji monolitycznej trwa o wiele dłużej.

Rozdział 5

Podsumowanie

W ramach pracy dyplomowej została zaimplementowana domena sklepu internetowego z kartami podarunkowymi w dwóch wersjach, tj. przy użyciu modeli monolitycznego z anemicznym modelem danych i Event Sourcing'u połączonego z CQRS.

Oba sposoby na tworzenie architektury systemów są poprawne i oba spotykane na co dzień w pracy programisty. Jedynie kontekst ich użycia jest kluczowy i musi być wzięty pod uwagę przy podejmowaniu decyzji o architekturze systemu. Kontekst rozumiany jest jako typ aplikacji, skomplikowość domeny biznesowej, możliwości i wymagania projektowe. Każda dziedzina problemowa będzie miała inne wymagania i to zależnie od nich decyzja na dane podejście musi być indywidualnie podjęta.

Jednakże zrozumienie jakie oba podejścia mają cechy charakterystyczne może bardzo pomóc przy decyzji. Ażeby można było lepiej zrozumieć wady i zalety obu podejść, autor pracy oparł swoje wnioski o jakość oprogramowania zgodną ze standardem ISO/IEC 25010:2011, gdzie jakość oprogramowania określana jest przez stabilność funkcjonalna, wydajność działania, kompatybilność, użyteczność, niezawodność, bezpieczeństwo, utrzymywalność i przenośność wytworzonego oprogramowania.[26]

Analiza jakości oprogramowania oparta na tym praktycznym przykładzie wykazała wady i zalety obu podejść.

Aplikacje oparte na Event Sourcing'u naturalnie implikują oparcie rozwiązania o architekturę mikroserwisową, która daje większą elastyczność, skalowalność i łatwość w rozszerzaniu funkcjonalności, niż aplikacje monolityczne i anemiczne. Jednakże zastosowanie Event Sourcing'u wymaga o wiele więcej wkładu pracy i czasu, bo oparcie systemu o wydarzenia pociąga za sobą inne komplikacje i utrudnienia występujące w asynchronicznej naturze wydarzeń. Wymaga poznania nowego podejścia, które często w ogóle nie jest znane dla programistów nawet z wieloletnim doświadczeniem. Jeżeli dziedzina problemowa jest dosyć skomplikowana i wymaga zapisu wszystkich zdarzeń w systemie, to najlepszym rozwiązaniem jest inwestycja czasu na poznanie i wdrożenie Event Sourcing'u. Wielką zaletą jest też przenaszalność systemu opartego na Event Sourcing'u ze względu na jego bardzo elastyczną skalowalność dzięki architekturze mikroserwisowej i sposobie działania Apache Kafka.

Model monolityczny lepiej sprawdza się przy domenie prostej do implementacji, a także przy ograniczonych zasobach systemowych. Jest on również prostszy w utrzymaniu, ze względu na mniejsze skomplikowanie wykonywania operacji na danych produkcyjnych i dostępie narzędzi do monitorowania całego systemu, które są ogólnodostępne i niewymagające implementacji własnych rozwiązań.

Przy tworzeniu aplikacji monolitycznych musi jednakże zawsze być zachowana świadomość niebezpieczeństwa, jakie wynika z anemicznego modelu danych i jej głębszych problemów w utrzymaniu.

Jeżeli zasadnym jest użycie Event Sourcing'u ze względu na domenę problemową i zasoby systemowe, to zastosowanie tego podejścia, połączonego z użyciem architektury mikroserwisowej zapewnia poprawę jakości oprogramowania w rozumieniu normy ISO/IEC 25010:2011.

Bibliografia

- [1] *Event sourcing, CQRS, stream processing and Apache Kafka: What's the connection?*, Dostęp: 2018-11-20, tłum. własne.
<https://www.confluent.io/blog/event-sourcing-cqrs-stream-processing-apache-kafka-whats-connection/>.
- [2] *Antywzorzec projektowy*, Dostęp: 2018-11-27. https://pl.wikipedia.org/wiki/Antywzorzec_textunderscoreprojektowy.
- [3] *Apache Kafka*, Dostęp: 2018-11-27. <http://dzikowski.github.io/2014/12/07/kafka/>.
- [4] *Apache Kafka Overview*, Dostęp: 2018-11-27. <https://www.cloudera.com/documentation/kafka/1-2-x/topics/kafka.html>.
- [5] *Automat skończony*, Dostęp: 2018-11-27. https://pl.wikipedia.org/wiki/Automat_skonczony.
- [6] *CQRS i Event Sourcing*, Dostęp: 2018-11-27. https://bulldogjob.pl/articles/122-cqrs-i-event-sourcing-czyli-latwa-droga-do-skalowalnosci-naszyc-systemow_.

-
- [7] *Czym są mikrousługi?*, Dostęp: 2018-11-27. <http://www.ictshop.pl/czym-sa-mikrouslugi/>.
- [8] *Difference between monolithic and microservices based architecture*, Dostęp: 2018-11-27. <https://www.slashroot.in/difference-between-monolithic-and-microservices-based-architecture>.
- [9] *Hibernate*, Dostęp: 2018-11-27. <https://pl.wikipedia.org/wiki/Hibernate>.
- [10] *Hibernate Tutorial For Beginners With Examples*, Dostęp: 2018-11-27. <https://examples.javacodegeeks.com/enterprise-java/hibernate/hibernate-tutorial-beginners-examples/>.
- [11] *Java*, Dostęp: 2018-11-27. <https://pl.wikipedia.org/wiki/Java>.
- [12] *main, 10 października 2016*, Dostęp: 2018-11-27. <https://kobietydokodu.pl/main-10-pazdziernika-2016/>.
- [13] *Mikro jest piękne*, Dostęp: 2018-11-27. <http://blog.streamsoft.pl/mikro-jest-piekne>.
- [14] *Spring*, Dostęp: 2018-11-27. <https://javastart.pl/baza-wiedzy/baza-wiedzy/frameworki/spring>.
- [15] *Słyszalesz o mikrousługach?*, Dostęp: 2018-11-27. <https://it-consulting.pl/autoinstalator/wordpress/tag/mikroserwisy>.

-
- [16] *Core Concepts*, Dostęp: 2018-11-27, tłum. własne. <https://kafka.apache.org/21/documentation/streams/core-concepts>.
- [17] *Docker overview*, Dostęp: 2018-11-27, tłum. własne. <https://docs.docker.com/engine/docker-overview/#docker-architecture>.
- [18] *Event Sourcing what it is and why it's awesome*, Dostęp: 2018-11-27, tłum. własne. <https://dev.to/barryosull/event-sourcing-what-it-is-and-why-its-awesome>.
- [19] *Introduction*, Dostęp: 2018-11-27, tłum. własne. <https://kafka.apache.org/intro>.
- [20] *Kafka Java Producer*, Dostęp: 2018-11-27, tłum. własne. <https://docs.confluent.io/current/clients/producer.html>.
- [21] *Lenses Box*, Dostęp: 2018-11-27, tłum. własne. <https://www.landoop.com/lenses-box/>.
- [22] *Monolithic application*, Dostęp: 2018-11-27, tłum. własne. https://en.wikipedia.org/wiki/Monolithic_application.
- [23] *What is Docker?*, Dostęp: 2018-11-27, tłum. własne. <https://opensource.com/resources/what-docker>.
- [24] *Immutableables*, Dostęp: 2018-12-20. <https://immutableables.github.io/>.
- [25] *Swagger – interaktywna dokumentacja API*, Dostęp: 2018-12-20. <https://www.simplecoding.pl/swagger/>.

-
- [26] *Charakterystyki jakości oprogramowania - ISO 25010*, Dostęp: 2019-01-26. <http://testerzy.pl/baza-wiedzy/charakterystyki-jakosci-oprogramowania-iso-25010>.
- [27] *Streams Concept*, Dostęp: 2019-01-26, tłum. własne. <https://docs.confluent.io/current/streams/concepts.html>.
- [28] *Apache Maven*, Dostęp: 2019-01-30. https://pl.wikipedia.org/wiki/Apache_Maven.
- [29] *Wstęp do REST API*, Dostęp: 2019-01-30. <https://devszczepaniak.pl/wstep-do-rest-api/>.

Spis rysunków

| | | |
|------|---|----|
| 2.1 | Architektura monolityczna. Źródło: [8] | 5 |
| 2.2 | Architektura Mikroserwisowa. Źródło: [8] | 6 |
| 2.3 | Zapis wydarzeń w systemie. Źródło: [18] | 10 |
| 2.4 | Odczyt danych z wydarzeń poprzez projekcję. Źródło: [18] | 10 |
| 2.5 | Graficzna reprezentacja wzorca CQRS. Źródło: [18] | 13 |
| 2.6 | Standardowa deklaracja zmiennej. Źródło: opracowanie własne | 14 |
| 2.7 | Deklaracja zmiennej w Javie 10. Źródło: opracowanie własne | 14 |
| 2.8 | Repozytorium Spring Data. Źródło: [14] | 16 |
| 2.9 | Klasa wraz z adnotacjami Hibernate. Źródło: [10] | 17 |
| 2.10 | Relacja między tematem a partycją w Apache Kafka. Źródło: [4] | 19 |
| 2.11 | Apache Kafka API. Źródło: [4] | 20 |
| 2.12 | Przykładowe Producer API w Javie. Źródło: opracowanie własne | 21 |
| 2.13 | Topologia procesora w Apache Kafka. Źródło: [16] | 22 |
| 2.14 | Przykładowe Stream API w Javie. Źródło: opracowanie własne | 23 |
| 2.15 | Dualizm strumieni i tabel. Źródło: [27] | 24 |
| 2.16 | Architektura Docker'owa. Źródło: [17] | 26 |
| 2.17 | Przykładowy interfejs Swagger'owy. Źródło: [25] | 28 |
| 2.18 | Definicja i tworzenie obiektu niezmiennego. Źródło: [24] | 29 |
| 3.1 | Diagram bazy danych. Źródło: opracowanie własne | 33 |
| 3.2 | Struktura aplikacji monolitycznej. Źródło: opracowanie własne | 35 |

| | | |
|------|--|----|
| 3.3 | Fragment klasy ładującej dane przy włączaniu aplikacji. Źródło: opracowanie własne | 37 |
| 3.4 | Lista kontrolerów w aplikacji. Źródło: opracowanie własne . . . | 38 |
| 3.5 | Endpointy kontrolera administratora. Źródło: opracowanie wła- sne | 39 |
| 3.6 | Endpointy kontrolera marek. Źródło: opracowanie własne | 40 |
| 3.7 | Endpointy kontrolera kupującego. Źródło: opracowanie własne . | 40 |
| 3.8 | Endpointy kontrolera kart. Źródło: opracowanie własne | 41 |
| 3.9 | Organizacja kodu w apiCommand. Źródło: opracowanie własne . | 47 |
| 3.10 | ReadOnlyKeyValueStore. Źródło: opracowanie własne | 48 |
| 3.11 | ValidatorService odpowiedzialny za walidację nowych wydarzeń przed wysłaniem do Kafki. Źródło: opracowanie własne | 49 |
| 3.12 | SendService. Źródło: opracowanie własne | 50 |
| 3.13 | Lista kontrolerów w mikrosерwisie apiCommand. Źródło: opra- cowanie własne | 50 |
| 3.14 | Endpointy kontrolera marki. Źródło: opracowanie własne | 51 |
| 3.15 | Endpointy kontrolera kart. Źródło: opracowanie własne | 52 |
| 3.16 | Organizacja kodu w apiQuery. Źródło: opracowanie własne . . . | 55 |
| 3.17 | Lista kontrolerów i endpointów w mikrosерwisie apiQuery. Źró- dło: opracowanie własne | 57 |
| 3.18 | Maszyna stanów. Źródło: opracowanie własne | 59 |
| 3.19 | Subskrypcja tematu orderEventTopic jako początek przetwarza- nia strumieni. Źródło: opracowanie własne | 60 |
| 3.20 | Transformacja i wysyłanie informacji na pojedynczy temat per encja. Źródło: opracowanie własne | 60 |
| 3.21 | Walidacja możliwych przejść na podstawie maszyny stanów. Źró- dło: opracowanie własne | 61 |

| | |
|--|----|
| 3.22 Agregacja listy kart na podstawie nazwy marki. Źródło: opracowanie własne | 62 |
| 3.23 Organizacja kodu w orderComponent. Źródło: opracowanie własne | 63 |
| 3.24 Lista tematów w interfejsie Kafka Lenses. Źródło: opracowanie własne | 64 |

Indeks

Anemic Model, 1, 3, 4, 8, 32, 68, 73, 76

aplikacja monolityczna, 1, 3–5, 8, 17,
31, 42, 50, 52, 55, 57, 65, 66,
69, 71–74, 76

CQRS, 1, 11, 12, 44, 45, 59, 62, 65, 75

CRUD, 1, 8, 9, 16

Event Sourcing, 1, 3, 4, 9, 10, 12, 14,
28, 41, 42, 65–72, 75, 76

Spring, 3, 15–17, 32, 34, 35, 46