Autor

Domain Driven Design krok po kroku

Część IVb: Skalowalne systemy w kontekście DDD – architektura Command-query Responsibility Segregation (stos Read)

W jaki sposób odczytywać dane w systemie pod dużym obciążeniem. Jak otworzyć system na skalowanie. Pułapki mapera relacyjno-obiektowego. Ostateczna spójność danych oraz... podróże w czasie.

W poprzednim artykule poznaliśmy ideę architektury Command-query Responsibility Segregation składającej się z dwóch stosów warstw: Read i Write. Poznaliśmy również szczegóły działania mechaniki stosu Write.

W niniejszym artykule przyjrzymy się szczegółom implementacji stosu Read, a w szczególności poznamy:

- techniki spłaszczania III Postaci Normalnej,
- stosowalność modeli nierelacyjnych,
- koncepcję kompromisu spójności danych Eventually Consistency,
- technikę persystencji zdarzeń biznesowych, będących modelem danych: Event Sourcing.

Plan serii

Niniejszy tekst jest czwartym artykułem z serii mającej na celu szczegółowe przedstawienie kompletnego zestawu technik modelowania oraz nakreślenie kompletnej architektury aplikacji wspierającej DDD.

Część I: Podstawowe Building Blocks DDD;
Część II: Zaawansowane modelowanie DDD – techniki strategiczne: konteksty i architektura zdarzeniowa;
Część III: Szczegóły implementacji aplikacji wykorzystującej DDD na platformie Java – Spring Framework;
Część IV: Skalowalne systemy w kontekście DDD – architektura CqRS;
Część V: Kompleksowe testowanie aplikacji opartej o DDD;
Część VI: Behavior Driven Development – Agile drugiej generacji.

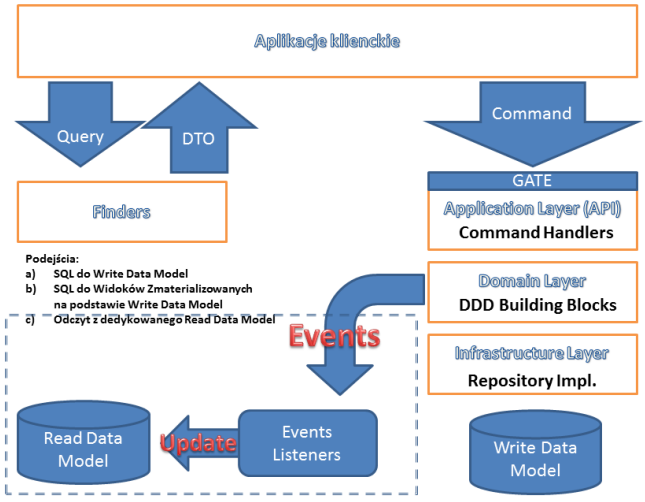
DWIE KLASY PROBLEMÓW - DWA STOSY WARSTW

Aby odświeżyć kontekst artykułu z poprzedniego miesiąca, spójrzmy na Rysunek 1, który ilustruje typowy system klasy enterprise obsługujący dwie klasy operacji:

Command (omówione w poprzednim numerze), które operują nad Building Blocks DDD- tutaj maper relacyjno-obiektowy sprawdza się doskonale

Query – odczyty stanu systemu – średnio w systemach biznesowych odczytów może być ok. 5 rzędów wielkości więcej niż zapisów

Rysunek 1. Architektura Command-query Responsibility Segregation – dedykowany stos do odczytu danych



STOS READ (QUERY)

Serwisy Stosu Read, które będziemy nazywać Finders, zwracają dane, których modele projektujemy na potrzeby konkretnych problemów (np. ekranów wyświetlających dane przekrojowe/raportowe). Są to zwykłe Data Transfer Objects (DTO), ponieważ Agregaty DDD nie nadają się do tego typu zadań z kilku powodów:

- zawierają struktury nieodpowiednie do prezentacji – spadek wydajności spowodowany pobieraniem z bazy i przesyłaniem nadmiarowych danych
- agregowane w nich obiekty mogą zostać pobrane poprzez mechanizm Lazy Loading, który w przypadku danych przekrojowych powoduje pojawienie się zjawiska „n+1 Select Problem" – drastyczny problem z wydajnością, który jest w większości przypadków niedopuszczalny
- Agregaty z definicji są hermetyczne
- posiadają metody biznesowe, które nie mają sensu w warstwie prezentacji

Dwa ostatnie problemy są charakterystyczne dla modelu zgodnych z DDD, natomiast dwa pierwsze to ogólna bolączka każdego nietrywialnego systemu.

PODEJŚCIA DO SEPARACJI READ MODEL

W celu obejścia/uniknięcia powyższych problemów stosuje się techniki opisane poniżej. Część z nich przedstawiono jedynie w celu uzupełnienia obrazu całości, nie zalecając ich stosowania.

Przepakowanie encji

W architekturach opartych na rozwiązaniach ORM widujemy podejścia, w których wybrane dane z encji zwracanych przez bibliotekę/framework ORM zostają przepakowane do DTO. Tego typu podejście, owszem redukuje ilość danych przesyłanych do zdalnych klientów, ale wciąż na serwerze aplikacyjnym dochodzi do „rozbłysków" pamięci spowodowanych pobieraniem zbędnych danych.

HQL szyty na miarę

Niektóre implementacje ORM (takie jak Hibernate) pozwalają pobrać dane, które wskażemy w zapytaniu. Służą do tego konstrukcje typu:

```
SELECT NEW pakiet.MyDTO(order.number, order.client.name)
FROM Order order
```

Są to konstrukcje charakterystyczne dla języka danego mapera – w tym wypadku konstrukcja przypomina „uruchomienie" konstruktora DTO w zapytaniu.

Wydajnościowo osiągniemy zdecydowanie lepsze wyniki niż w poprzednim podejściu, jednak możemy zastanowić się nad sensownością uruchamiania tych zapytań poprzez połączenia zestawione w trybie READ-WRITE, skoro wystarczające jest połączenie w trybie READ (chodzi tu o tryb działania połączenia do bazy danych, a nie o nazwy stosów w CqRS).

Tego typu obiekty DTO nie są obarczone mechanizmem śledzenia zmian (Dirty Checking), mechanizmem Lazy Loadingu ani wsparcia dla zapisu kaskadowego. W stosie Read są to mechanizmy zbędne, ponieważ z założenia planujemy jedynie odczyt danych. Modyfikacja następuje jedynie w stosie Write.

SQL szyty na miarę

Jeżeli już decydujemy się na pobieranie poszczególnych kolumn z bazy, to dlaczego nie użyć czystego SQL zamiast konstrukcji języka zapytań mapera? Dodatkowo skoro wiemy, że w pewnych miejscach pobieramy dane jedynie do odczytu, to być może warto zestawić osobną pulę połączeń z bazą - połączeń read-only.

W takim wypadku warto użyć lekkiego mapera typu myBatis, który mapuje Result Set na paczki danych (DTO), a nie na obiekty biznesowe służące do wykonywania operacji biznesowych!

Widoki zmaterializowane

Separację możemy poprowadzić jeszcze "głębiej" i dokonać projekcji modelu domenowego, który utrzymujemy w III postaci normalnej do postaci płaskiej, odpowiedniej do odczytu. W tym celu możemy zastosować Widoki Zmaterializowane.

Projektując Widoki, zmieniamy myślenie w stylu „co będziemy zapisywać" na myślenie w stylu „co będziemy odczytywać". Pozwala to na projektowanie dedykowanych do szybkich odczytów struktur. Można traktować je jako projekcje modeli domenowych na potrzeby wydajnych odczytów. Być może potrzebujemy więcej niż jednej projekcji danego modelu domenowego. Pozwoli to uniknąć klauzul JOIN, które pojawiają się w nieturlany sposób, gdy odczytujemy dane z Modelu Domenowego.

Dedykowany Read Model odświeżany zdarzeniami

Być może powyższe rozwiązania wciąż nie przynoszą zadowalających wyników wydajnościowych. Możemy wówczas zaprojektować osobne modele do odczytu przygotowane do skalowania poziomego. W naszym przykładzie problematyczny może okazać się odczyt katalogu produktów, do tego odczyt w różnych ujęciach. Rozwiązaniem mogą okazać się osobne modele oparte np. o Shredding danych.

Natychmiast pojawia się pytanie o odświeżanie danych w tego typu modelach. Co w sytuacji, gdy w Write Modelu (modelu domenowym) nastąpi np. zmiana danych produktu? W poprzednich częściach mówiliśmy o zdarzeniach domenowych, które są emitowane przez Agregaty w „ważnych momentach ich życia". Zdarzenia były stosowane w celu:

- zapewniania asynchroniczności - odłożenia na później niekrytycznych operacji, czyli zwiększenia responsywności systemu,
- otwarcia systemu na pluginy,
- decouplingu Bounded Contexts.

Na tym etapie możemy użyć tych samych zdarzeń w celu odświeżenia Read Modelu. Dokładnie rzecz ujmując: wprowadzamy techniczne Listenery zdarzeń domenowych, których zadaniem będzie uaktualnianie Read Modeli. Przykład tego typu Listenera ilustruje Listing 1.

Listing 1. Listener pl.com.bottega.erp.sales.presentation.listeners.ProductEventsListener, którego zadaniem jest nasłuchiwanie zdarzenia domenowego i uaktualnianie dedykowanego modelu do odczytu

```
@EventListeners
public class ProductEventsListener {

   @PersistenceContext
   private EntityManager entityManager;

   @EventListener(asynchronous=true)
   public void handle(ProductAddedToOrderEvent event){
      entityManager.persist(new OrderedProduct(
         event.getProductid(),
         event.getClientId(),
         event.getQuantity(),
         new Date()));
   }
}
```

NoSQL

W pewnych zastosowaniach warto rozważyć wykorzystanie NoSQL do implementacji Read Model.

Wyobraźmy sobie następujący przykład: na stronie z podglądem szczegółów produktu wyświetlamy obok ceny, zdjęcia i opisu produktu również sugestię zakupu innych produktów – takich, które zostały zakupione przez znajomych-znajomych zalogowanego użytkownika, którzy należą do grafu jego przyjaciół w module „Social" naszego systemu. Stworzenie modelu grafu w bazie relacyjnej nie jest szczególnie problematyczne, natomiast odczyt takiej struktury może być już nieco złożony (szczególnie czasowo).

Warto wówczas rozważyć Read Model sugerowanych produktów zaimplementowany w bazie grafowej (np. Neo4j), która pozwala na wykonywanie zapytań typu „select ścieżka from węzeł A to B".

W początkowej fazie projektu możemy potraktować taki graf jako zewnętrzny index dla naszych danych domenowych. Czyli graf przechowuje strukturę zakupionych produktów, a węzły grafu jedynie numery produktów. Zatem aby odczytać dane o produktach, należy pobrać je z 2 źródeł: graf zawierający jedynie numery, a następnie posiadając numery, dane o produktach z bazy produktów. Takie rozwiązanie będzie oczywiście niewydajne od pewnej skali, zatem możemy zdecydować się na duplikację da-

nych o produktach i umieszczenie ich również w bazie grafowej.

W razie zmian danych produktu w Domain Modelu pojawią się zdarzenia domenowe, które zostaną wychwycone przez jednego z Listenerów i odświeżą graf w Read Modelu.

OSTATECZNA SPÓJNOŚĆ DANYCH

Oczywiście we wszystkich opisanych powyżej przypadkach mamy do czynienia z chwilową niespójnością danych. Pomiędzy emisją zdarzeń z Domain Modeli a ich obsługą w Read Modelu mamy czas, gdy dane nie są spójne.

Zakładając, że silnik będący nośnikiem zdarzeń zapewnia ich trwałość, możemy mieć pewność, że dane ostatecznie będą spójne (Eventually Consistent).

Należy wówczas zadać sobie pytanie o wymagania niefunkcjonalne: jak system powinien zachować się w wypadku wykrycia niespójnych danych. Jak biznes radzi sobie w tym momencie z niespójnymi danymi.

Rozważmy przykład: w module magazynowym mamy zarejestrowaną 1 sztukę pewnego towaru. W module sprzedaży następuje zakup tego towaru. Pojawia się zdarzenie. Jeden z Listenerów będzie uaktualniał katalog produktów, a drugi Listener (w module magazynowym) zajmie się rezerwacją produktu. Co powinno się stać, gdy dwa zamówienia na ten sam produkt pojawią się niemal równocześnie, co doprowadzi do stanu magazynowego o wartości -1?

Jak w takich sytuacjach zachowuje się biznes aktualnie – bez systemu IT? Czy jest to akceptowalne? Być może nie, a być może tak... Czy z drugiej strony akceptowalne jest z biznesowego punktu widzenia zachowanie systemu, polegające na odrzuceniu drugiego zamówienia? Być może w takich sytuacjach biznes chce po prostu powiadomienia o brakach na magazynie i operator zajmie się sprowadzeniem/produkcją towaru...?

Zwykle możemy sobie pozwolić na Eventually Consistency, gdy w różne etapy tego samego procesu są zaangażowani różnie użytkownicy.

Powyższe rozważania mogą budzić kontrowersyjne odczucia, ale przy pewnej skali jedyne, co możemy zrobić, to dokonać kompromisów w celu zapewniania wymaganej wydajności. Kluczowe jest zrozumienie modelu biznesowego i jego zachowań w sytuacjach wykrycia niespójności.

DUPLIKACJA DANYCH TO NIE DUPLIKACJA LOGIKI

W podejściu CqRS decydujemy się na duplikowanie danych w celu zwiększenia wydajności całego systemu. Może to prowadzić do pojawienia się chwilowej niespójności zduplikowanych danych.

Należy zwrócić jednak uwagę, że czym innym jest duplikacja danych, a czym innym jest duplikacja logicznych punktów decyzyjnych. Stos Read (być może wielokrotnie zduplikowany) służy jedynie do odczytu danych, natomiast wszelkie operacje biznesowe są wykonywane przez stos Write, którym stan Modelu Domenowego jest tak zwanym „źródłem prawdy".

Przykładowo: w Write Modelu może dojść do dezaktywacji produktu w katalogu. Zanim zmiana ta pojawi się w Read Modelu, niektórzy użytkownicy mogą wyrazić chęć zakupu tego produktu. Jednak zakup zawsze przechodzi przez Write Model, w którym to dane są aktualne. Tak więc w gestii Logiki Aplikacji lub Logiki Domenowej w Write Modelu jest podjęcie stosownych działań w momencie wykrycia, że Command odnosi się do nieaktywnego produktu. Być może Domena odrzuci taki produkt, a być może Aplikacja zasugeruje jego zamiennik...?

EVENT SOURCING

Zdarzeń możemy również użyć jako modelu danych naszych Agregatów w Write Modelu. Technika, w której modelem danych Agregatów (zamiast np. bazy relacyjnej) jest zapisana sekwencja zdarzeń domenowych, nazywa się Event Sourcing. ES może być stosowany niezależnie od CqRS, jak również CqRS może być stosowany bez ES.

Zainteresowanych techniką ES odsyłamy do ramki „W sieci", a w ramach niniejszego tekstu ograniczymy się do naszkicowania zastosowań tego typu modelu danych.

Kiedy Event Sourcing ma zastosowanie

Składowanie w magazynie danych sekwencji zdarzeń, jakie zaszły w Agregatach DDD, pozwala na zadanie następującego pytania do Repozytorium: jaki był stan Agregatu w momencie X. Spowoduje to odczytanie sekwencji zdarzeń dla danego Agregatu do danego momentu w czasie i zaaplikowanie ich do zwracanego przez Repozytorium obiektu.

Dzięki temu możemy pobrać kilka instancji danego agregatu z różnych momentów w czasie i dokonać pożytecznych „obserwacji" na temat tego, co zaszło w biznesie.

Odpowiednio podzielone sekwencje zdarzeń mogą stanowić wektory uczące dla Sztucznych Sieci Neuronowych.

W sekwencjach zdarzeń możemy wykrywać wzorce przy pomocy silników CEP (Complex Events Processing), które pozwalają nakładać filtry typu: „sekwencja zdarzeń typu X, spełniających pewne kryteria, która zaszła w oknie czasowym Y lub w oknie ilościowym Z".

- oficjalna strona DDD http://domaindrivendesign.org
- wstępny artykuł poświęcony DDD http://bottega.com.pl/pdf/materialy/sdj-ddd.pdf
- przykładowy projekt: http://code.google.com/p/ddd-cqrs-sample/
- Architektura CqRS http://martinfowler.com/bliki/CQRS.html
- Event Sourcing http://martinfowler.com/eaaDev/EventSourcing.html

Sławomir Sobótka slawomir.sobotka@bottega.com.pl

Programujący architekt aplikacji specjalizujący się w technologiach Java i efektywnym wykorzystaniu zdobyczy inżynierii oprogramowania. Trener i doradca w firmie Bottega IT Solutions. W wolnych chwilach działa w community jako: prezes Stowarzyszenia Software Engineering Professionals Polska (http://ssepp.pl), publicysta w prasie branżowej i blogger (http://art-of-software.blogspot.com).