Sławomir Sobótka

Domain Driven Design krok po kroku

Część IVa: Skalowalne systemy w kontekście DDD - architektura Command-query Responsibility Segregation (stos Write)

Czy możliwe jest stworzenie systemu, który będzie charakteryzował się otwartym na rozbudowę modelem, eleganckim, testowalnym i utrzymywalnym kodem, a jednocześnie będzie przygotowany do skalowania? Czy narzędzia typu Object-relational mapper są panaceum na wszystkie problemy persystencji w systemach biznesowych? Czy baza relacyjna to zawsze najlepszy pomysł na przechowywanie danych? Na te i inne pytania odpowiemy sobie w kolejnej odsłonie naszej serii.

WSTĘP

Po omówieniu ogólnej idei DDD, technik zarówno taktycznego, jak i strategicznego modelowania oraz sposobów wykorzystania popularnych frameworków przyszedł czas na przyjrzenie się architekturze systemu jako całości. Zakładamy, że tworzymy system, co do którego postawiono następujące wymagania niefunkcjonalne:

- dostęp poprzez wiele technologii klienckich: web (w tym ajax), mobile, web service...,
- skalowanie poszczególnych modułów (w sensie wydzielonych jako produkty funkcjonalności) w celu zapewnienia wydajności,
- odporność na awarie pojedynczych modułów – system jako całość pozostaje stabilny,
- stworzenie platformy otwartej na rozszerzenia poprzez wpinanie pluginów.

W celu spełnienia tych wymagań:

- zrewidujemy tradycyjne podejście do architektur warstwowych, rozwijając je w kierunku Architektury Porst & Adapters (ramka „W sieci"),
- zastanowimy się nad kwestią spójności danych – kiedy jest ona krytyczna, a w jakich wypadkach możemy pozwolić sobie na chwilową niespójność, zyskując skalowalność,
- przyjrzymy się miejscu, jakie Object-relational Mapper zajmuje w systemie oraz kiedy jego wykorzystanie nie jest racjonalne,
- zastanowimy się nad odpowiednimi modelami danych i ich formami,
- powrócimy do zagadnienia zdarzeń omawianych w poprzednich częściach, aby wykorzystać ich pełen potencjał,
- Projekt referencyjny.

Wszystkich tych Czytelników, którzy już teraz chcieliby zapoznać się z kolejnymi zagadnieniami naszej serii, zapraszam do odwiedzenia strony projektu „DDD&CqRS Laven", której adres znajduje się w ramce „W sieci". Dostępne wersje:

- Java – Spring
- Java – EJB 3.1
- .NET - C#

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

Typowy system klasy enterprise obsługuje dwa typy operacji:

- rozkazy wykonujące pewne operacje biznesowe – co prowadzi zwykle do modyfikacji pewnej (relatywnie niedużej) ilości danych, a w konsekwencji do konieczności utrwalenia tychże zmodyfikowanych danych,
- kwerendy (w tym kontekście niekoniecznie w sensie SQL) odpytujące część danych składowanych w systemie.

Statystycznie rzecz biorąc, ilość obsługiwanych rozkazów do ilości obsługiwanych kwerend (w jakimś interwale czasu) ma się jak 1 do kilkustet (tysięcy). Innymi słowy odczytów jest o kilka rzędów wielkości więcej niż zapisów – śmiało możemy przyjąć, że dla typowego systemu klasy enterprise może być 5 rzędów wielkości.

Drugim aspektem – obok statystycznego - który różni oba typy operacji jest aspekt jakościowy modelu danych. Model danych potrzebny do wykonania operacji na modelu biznesowym oraz dane potrzebne do prezentacji wizualnej lub komunikacji z innymi modułami to zwykle inne struktury.

Z uwagi na obszerność merytoryczną, czwartą część serii podzielono na dwa osobne artykuły: niniejszy poświęcony Rozkazom oraz część, która ukaże się w przyszłym miesiącu i będzie poświęcona Kwerendom.

POTRZEBA SEPARACJI

Projektując model danych, zwykle wybieramy podejście Relacyjne i skupiamy się na wsparciu dla modelu domenowego - czyli dążymy do Trzeciej Postaci Normalnej. Postać ta, ze względu na brak redundancji, jest optymalna z punktu widzenia modyfikacji danych, zatem pożyteczna dla operacji typu Rozkaz.

Natomiast postać ta w przypadku operacji typu Kwerenda skutkuje pojawianiem się iloczynów kartezjańskich (JOIN w SQL). Dodatkowo Agregaty mogą zawierać dane zupełnie nieistotne w kontekście danej Kwerendy.

Stosowalność Object-relational Mapper

Świat relacyjny na obiektowy mapujemy po to, aby:

- Pobierać w wygodny sposób obiekty biznesowe - wraz z wygodnymi mechanizmami typu Lazy Loading,
- Wykonywać na nich operacje biznesowe zmieniające ich stan - możemy tutaj tworzyć zarówno anemiczne encje modyfikowane przez serwisy, jak również projektować prawdziwe obiekty modelujące reguły i niezmienniki biznesowe (styl Domain Driven Design),
- Utrwalać stan obiektów biznesowych - stan, który zmienił się w poprzednim kroku (korzystając z wygodnych mechanizmów wykrywania "brudzenia" i mechanizmu kaskadowego zapisu całych grafów obiektów).

Jeżeli stosujemy ORM (np. Java Persistence API) do tych klas problemów, to używamy odpowiedniego „młotka" do odpowiedniej klasy problemu. Czyli pobieramy kilka obiektów biznesowych (Agregatów), zmieniamy jego stan, zapisujemy go.

Pisząc „zmieniam stan", nie mam na myśli "edytuję, podpinając pod formularz". Mam na myśli logikę aplikacji (modelującą Use Case/User Story), która modyfikuje mój obiekt biznesowy (uruchamiając jego metody biznesowe lub settery, jeżeli jest on anemiczny). Na Listingu 1 widzimy przykład z poprzedniej części serii – przypomnijmy: Order i Invoice to persystentne Agregaty:

Listing 1. Serwis Aplikacyjny `pl.com.bottega.erp.sales.application.services.Purchase Service` operujący na kilku persystentnych Agregatach

```
public class PurchaseService{
//...
public void approveOrder(Long orderId) {
      Order order = orderRepository.load(orderId);

      //sample: Specification Design Pattern
      Specification<Order> orderSpecification =
         generateSpecification(systemUser);
      if (!orderSpecification.isSatisfiedBy(order))
         throw new OrderOperationException("Order does not
            meet specification", order.getEntityId());

      order.submit();

      Invoice invoice = invoicingService.issuance(order,
                generateTaxPolicy(systemUser));

      invoiceRepository.save(invoice);
      orderRepository.save(order);
   }
}
```

Jeżeli natomiast chcemy wyświetlić na ekranie dane, np. dane przekrojowe w postaci tabelki (wszyscy wiemy, że dla klienta biznesowego najważniejsze są tabelki, w których można przestawiać kolejność ich kolumn), to narzędzie typu ORM nie jest najlepszym rozwiązaniem tego problemu. W tym wypadku na każdym etapie postępujemy nieracjonalnie:

- Pobieramy z ORM listę obiektów (zamapowanych na całe tabelki w bazie), gdy potrzebuję na ekranie jedynie kilku kolumn z każdej tabelki (dla bazy nie robi to różnicy, ale w przypadku komunikacji sieciowej zaczniemy odczuwać skutki wydajnościowe tej decyzji),
- Mam możliwość korzystania z mechanizmu Lazy Loadingu, który nie ma sensu dla operacji typu "pobierz dane do wyświetlenia" i prowadzi do dramatycznego problemu z wydajnością „N+1 Select Problem",
- Silnik mapera wykonuje niepotrzebne operacje związane ze wsparciem dla Lazy Loading i Dirty Checking, które nie będą wykorzystywane,
- Zdradzam model biznesowy warstwie prezentacji. Być może w prostych aplikacjach z prezentacją w technologii webowej (ta sama maszyna pobiera i prezentuje dane) nie jest to problem - dodatkowo zyskujemy produktywność w pracy. Ale jeżeli klienty są zdalne (np. Android)? Zdradzanie modelu domenowego wiąże się z drastycznym spadkiem bezpieczeństwa (wsteczna inżynieria) oraz z koniecznością koordynacji prac zespołów pracujących nad "klientem" i "serwerem", i zapewnianiem kompatybilności starszych wersji klientów. Co prawda jest to kwestia oczywista, ale w „materiałach ewangelizacyjnych" popularnych platform korporacyjnych można znaleźć nawoływanie do takich „uproszczeń",
- Pracujemy na puli połączeń zestawionej w trybie READ-WRITE, gdy wystarczający jest tryb READ – większość baz danych optymalizuje działanie w tym trybie.

WARSTWY SĄ DOBRE, ALE DWA STOSY WARSTW SĄ JESZCZE LEPSZE

W dotychczasowych artykułach opieraliśmy architekturę aplikacji na stylu warstwowym, wprowadzając warstwy:

- prezentacji,
- logiki aplikacji,
- logiki domenowej (z ew. podziałem na 4 poziomy modelu: Capability, Operations, Policy, Decission Support),
- Infrastruktury.

Architektura Command-query Responsibility Segregation przedstawiona na Rysunku 1 rozdziela odpowiedzialność kodu, na dwa wyraźnie stosy warstw. Stos „Write" zawiera opisane powyżej warstwy i obsługuje omawiane wcześniej polecenia typu Command – operacja biznesowa i zmiana stanu systemu. Drugi stos „Read" jest mniej złożony i zawiera serwisy wyszukujące dane.

Separacja stosów nie dotyczy jedynie kodu. Problemy wydajnościowe opisane w sekcji „Potrzeba separacji" mogą (ale nie muszą – o czym w dalszej części) wymóc decyzję o rozdziale modelu danych. Model odpowiedni do operacji biznesowych zwykle nie jest odpowiedni do szybkiego serwowania danych np. na potrzeby prezentacji.

STOS WRITE (COMMAND)

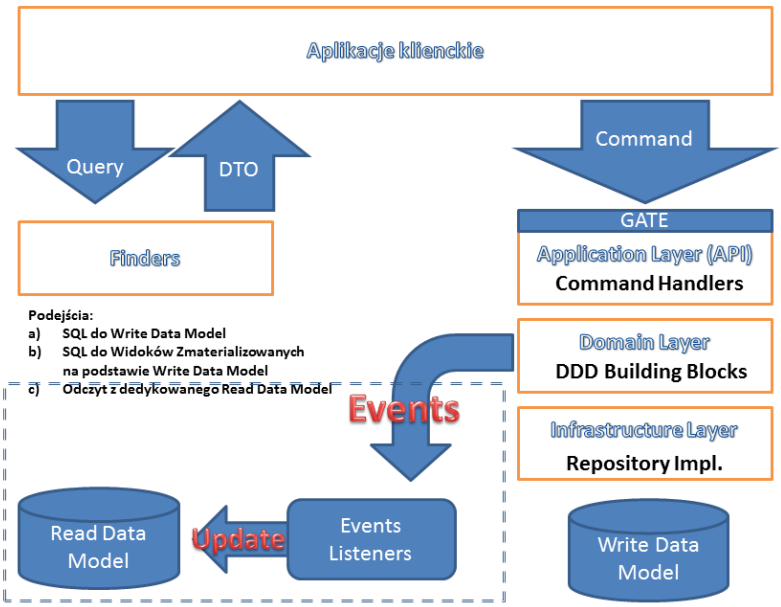
Stos „Write" niejawnie omawialiśmy w poprzedni częściach – były to nasze serwisy aplikacyjne. Natomiast w niniejszym artykule – bazując na założeniach i wiedzy z poprzednich części – wprowadzimy pewne rozszerzenia.

Stos „Write" możemy tworzyć w dwóch stylach:

- „klasycznym" w postaci serwisów,
- opartym na dostosowanym do architektury klient-serwer Wzorcu Projektowym Command.

Dla projektantów posiłkujących się frameworkiem Spring (lub innym kontenerem wspierającym Aspect Oriented Programming) wybór stylu ma wymiar głównie estetyczny, ponieważ przy pomocy AOP można osiągnąć w klasycznym podejściu wszystko to, co oferuje podejście oparte o styl Command.

Jeżeli natomiast nie mamy dostępu do AOP, wówczas styl Commad daje nam znaczną przewagę dzięki możliwości silnego odwracania kontroli – co zobaczymy już niebawem...



Rysunek 1. Architektura Command-query Responsibility Segregation – dwa stosy warstw. Komponent Gate został w szczegółach przedstawiony na Rysunku 2

Styl Serwisowy

Styl Serwisowy omawialiśmy w poprzednich częściach serii, dla przypomnienia odsyłam do Listingu 1. Transakcyjność i bezpieczeństwo zostały wprowadzone poprzez mechanizmy AOP dzięki zastosowaniu Adnotacji interpretowanych przez Spring Framework.

Styl Command & Command Handler

Alternatywą dla każdej z metod Serwisowych jest para: Command i Command Handler. Jak widać na Listingach 2 i 3 styl ten polega na rozdzieleniu klasycznego Wzorca Projektowego Command na 2 klasy. Command – przesyłany przez Tier Kliencki zawiera jedynie parametry żądania. Natomiast odpowiadający mu Command Handler zawiera logikę obsługującą Command. Dzięki temu w odróżnieniu od klasycznego Wzorce Command, aplikacje klienckie nie mają dostępu do kodu logiki aplikacyjnej – ma to oczywisty wpływ na bezpieczeństwo (dekompilacja) i ogólny coupling modułów.

Ogólne zasady tworzenia Command Handlerów pozostają takie same jak tworzenia metod Serwisów Aplikacyjnych omawianych w poprzednich częściach. Przedstawiony na Listingu 3 Command Handler jest odpowiednikiem Serwisu Aplikacyjnego z Listingu 1.

Listing 2. Command `pl.com.bottega.erp.sales.application.commands.SubmitOrderCommand` – polecenie zatwierdzenia zamówienia niosące parametry z warstwy prezentacji

```
@SuppressWarnings("serial")
@Command(unique=true)
public class SubmitOrderCommand implements Serializable{

    private final Long orderId;

    public SubmitOrderCommand(Long orderId) {
        this.orderId = orderId;
    }

    public Long getOrderId() {
        return orderId;
    }

    @Override
    public boolean equals(Object obj) {
        if (obj instanceof SubmitOrderCommand) {
            SubmitOrderCommand command = (SubmitOrderCommand) obj;
            return orderId.equals(command.orderId);
        }

        return false;
    }

    @Override
    public int hashCode() {
        return orderId.hashCode();
    }

}
```

Wspólny punkt – miejsce na odwracanie kontroli

Aplikacje klienckie komunikując się z Serwerem, wysyłają Command na wspólny punktu dostępowy, przedstawiony na Listingu 4 oraz zilustrowany na Rysunku 2.

Wspólny punkt dostępowy pozwala na wprowadzenie interesujących i pożytecznych operacji dodatkowych wykonywalnych podczas obsługi polecenia. Przykładowo:

- Sprawdzenie, czy dany Command jest duplikatem (przykładowo atak DOS) – jeżeli tak, to następuje jego odrzucenie. Jako duplikat traktujemy Command oznaczony odpowiednią adnotacją oraz przechowywany w rejestrze ostatnio obsługiwanych poleceń (ostatnio, czyli w czasie x milisekund lub w obszarze x MB pamięci lub w ilości x). Przykładowo dodanie kilkakrotnie tego samego produktu do zamówienia nie będzie traktowane jako duplikat (pozwólmy klientom wydawać pieniądze), ale zatwierdzenie tego samego zamówienia więcej niż raz jest niepożądane. Dzięki mechanizmowi historii poleceń możemy odrzucać niepożądane duplikaty bez potrzeby sięgania do bazy po dane biznesowe.
- Sprawdzenie, czy dany Command jest oznaczony jako asynchroniczny – jeżeli tak, to wówczas odkładamy jego wykonanie np. do Kolejki.

Listing 3. Command `pl.com.bottega.erp.sales.application.commands.handlers.SubmitOrderCommandHandler` – polecenie zatwierdzenia zamówienia niosące parametry z warstwy prezentacji

```
@CommandHandlerAnnotation
public class SubmitOrderCommandHandler implements CommandHandler<SubmitOrderCommand, Void> {

   @Inject
   private OrderRepository orderRepository;

   @Inject
   private InvoiceRepository invoiceRepository;

   @Inject
   private InvoicingService invoicingService;

   @Inject
   private SystemUser systemUser;

   @Override
   public Void handle(SubmitOrderCommand command) {
      Order order = orderRepository.load(command.getOrderId());

      Specification<Order> orderSpecification = generateSpecification(systemUser);
      if (! orderSpecification.isSatisfiedBy(order))
         throw new OrderOperationException("Order does not meet specification", order.getEntityId());

      //Domain logic
      order.submit();
      //Domain service
      Invoice invoice = invoicingService.issuance(order, generateTaxPolicy(systemUser));

      orderRepository.save(order);
      invoiceRepository.save(invoice);

      return null;
   }
```

Listing 4. Command pl.com.bottega.cqrs.command.impl.StandardGate – wspólny punkt dostępowy

```
@Component
public class StandardGate implements Gate {

   @Inject
   private RunEnvironment runEnvironment;

   private GateHistory gateHistory = new GateHistory();

   @Override
   public Object dispatch(Object command){
      if (! gateHistory.register(command)){
         //TODO log.info(duplicate)
         return null;//skip duplicate
      }

      if (isAsynchronous(command)){
         //TODO add to the queue. Queue should send this command to the RunEnvironment
         return null;
      }

      return runEnvironment.run(command);
   }

   private boolean isAsynchronous(Object command) {
      if (! command.getClass().isAnnotationPresent(Command.class))
         return false;

      Command commandAnnotation = command.getClass().getAnnotation(Command.class);
      return commandAnnotation.asynchronous();
   }
}
```

Listing 5. Środowisko uruchomieniowe pl.com.bottega.cqrs.command.impl.RunEnvironment pozwalające na wprowadzenie dodatkowych mechanizmów Odwracania Kontroli

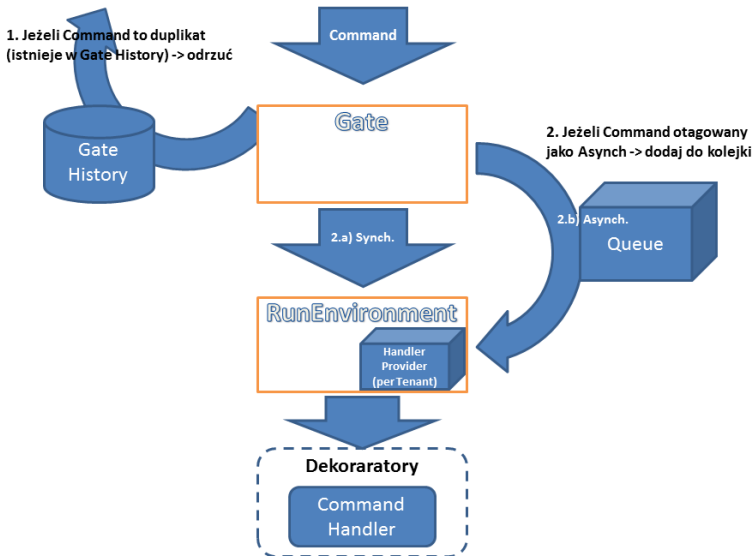
```
@Component
public class RunEnvironment {

   public interface HandlersProvider{
      CommandHandler<Object, Object> getHandler(Object command);
   }

   @Inject
   private HandlersProvider handlersProfiver;

   public Object run(Object command) {
      CommandHandler<Object, Object> handler = handlersProfiver.getHandler(command);
      //You can add Your own capabilities here: dependency injection, security, transaction management, logging,
profiling, spying, storing commands, etc
      Object result = handler.handle(command);
      //You can add Your own capabilities here
      return result;
   }
}
```

Rysunek 2. Komponent Gate stanowiący wspólny punkt dostępowy – obsługa Command wysyłanych z aplikacji klienckich



Przyjrzyjmy się teraz Listingowi 5, który przedstawia środowisko uruchomienia polecenia.

Odpowiedzialność Środowiska Uruchomieniowego jest prosta: dopasować do Command odpowiedni ComamndHandler i wykonać go.

Dzięki scentralizowanemu punktowi uruchamiania logiki aplikacji mamy możliwość dokonania całkowitego odwrócenia kontroli, bez użycia kontenera typu Spring:

- Rozpoczęcie transakcji przed uruchomianiem Hadlera oraz jej zatwierdzenie albo wycofanie w zależności od powodzenia działania Handlera.
- Sprawdzenie uprawnień do wykonania Handlera.
- Wstrzyknięcie zależności do Handlera.
- Logowanie poczynań klientów.
- Profilowanie działania systematu.
- Zwrócenie Handlera odpowiedniego dla pracującego w systemie klienta – zagadnienie wersjonowania API i systemy wspierające multi-tenant.
- Dekorowanie Handlerów – wzorzec Dekoratora.

Transakcyjność vs wydajność

Jedną z głównych zasad projektowania skalowalnych systemów z wykorzystaniem DDD jest zapisywanie podczas transakcji jednego Agregatu. Odczytać (w celu wykonania operacji biznesowej) możemy oczywiście kilka Agregatów, ale zapisać – tylko jeden.

Jest to oczywiście jedynie wskazówka, którą możemy złamać w uzasadnionych przypadkach, ale zastanówmy się nad konsekwencją jej stosowania.

Jeżeli zapisujemy w jednej transakcji kilka Agregatów, to nie możemy sobie pozwolić na ich utrwalanie w osobnych bazach danych. Zakładając oczywiście, że transakcje rozproszone (Double Phase Commit) nie jest tym, czego potrzebuje architekt projektujący wysokowydajny i skalowalny system.

Zdarzenia

Co zatem zrobić, jeżeli musimy zapisać więcej niż jeden Agregat podczas obsługi Command? Jeżeli wynika to po prostu z wymagań. Rozwiązaniem mogą być Zdarzenia Domenowe, które omawialiśmy w poprzedniej części naszej serii. Przypomnijmy, że Agregaty emitują zdarzenia niosące informacje o istotnych w cyklu życia Agregatów „wydarzeniach biznesowych". Zdarzenia do tej pory stosowaliśmy w celu:

- komunikacji pomiędzy osobnymi Bounded Context
- odłożeniu wykonania do kolejki operacji, których natychmiastowe wykonanie nie jest krytyczne
- otwarciu systemu na Pluginy (którymi są Listenery Zdarzeń, jak również Polityki DDD)

Zdarzenia będą nam również potrzebne, aby odświeżać dane w dane do odczytu w drugim stosie naszej Architektury CqRS. Zdarzeń możemy również używać jako Behwioralnego Modelu Danych i zastąpić nimi Relacyjną Bazę w Stosie Write. Zdarzenia mogą być analizowane przez systemy CEP (Complex Events Processing) oraz służyć jako wektory uczące dla Sztucznych Sieci Neuronowych. Ale o tym w kolejnej odsłonie serii.

W sieci

- oficjalna strona DDD: http://domaindrivendesign.org
- wstępny artykuł poświęcony DDD: http://bottega.com.pl/pdf/materialy/sdj-ddd.pdf
- przykładowy projekt – Java (Spring i EJB): http://code.google.com/p/ddd-cqrs-sample/
- przykładowy projekt .NET: http://cqrssample.codeplex.com
- Architektura CqRS http://martinfowler.com/bliki/CQRS.html
- Architektura Porst and Adapters: http://c2.com/cgi/wiki?PortsAndAdaptersArchitecture

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).