

Wsparcie projektowania i implementacji aplikacji opartych o architekturę CQRS

Michał Aniserowicz

Warszawa, 19.03.2015

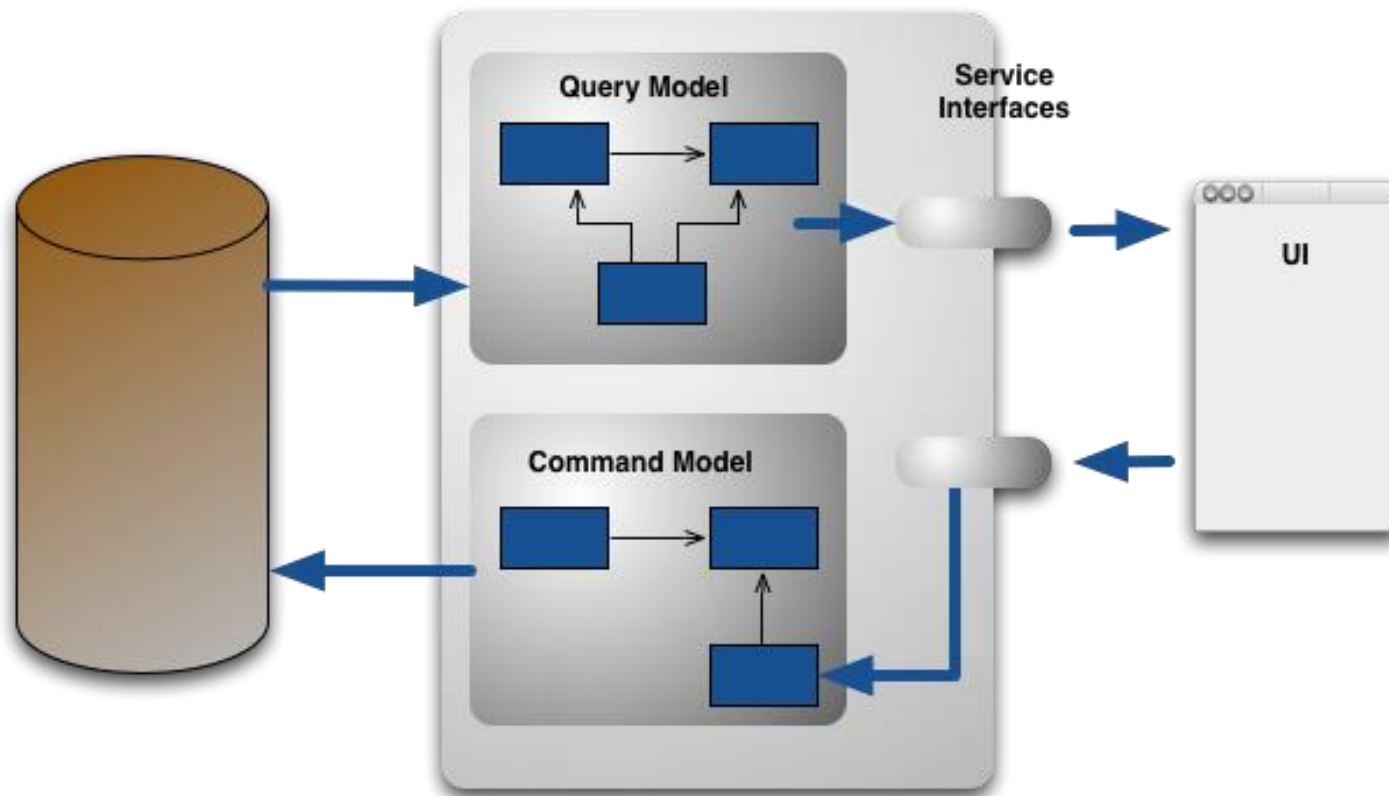
Agenda

- Architektura CQRS
- Problem
- Zaproponowane rozwiązanie
- Przykładowa aplikacja
- Opis dziedziny aplikacji
- Działanie generatora
- Podsumowanie

Architektura CQRS

- CQRS = Command Query Responsibility Segregation.
- Zakłada podział wszystkich działań w aplikacji na dwa rodzaje:
 - zapytanie (*query*) - działanie wiążące się z odczytaniem danych z bazy danych (lub innego źródła);
 - komenda (*command*) - działanie wiążące się z modyfikacją danych.
- Działania te są rozłączne – działają na dwóch osobnych modelach danych aplikacji:
 - model zapytań (*Query Model*) - model przeznaczony do odczytu danych;
 - model komend (*Command Model*) - model przeznaczony do modyfikacji danych.

Architektura CQRS: działanie



Architektura CQRS: zalety

- Skomplikowana dziedzina aplikacji może być podzielona na dwie prostsze dziedziny, co ułatwia jej zrozumienie.
- Zapytania i komendy mogą być wykonywane równolegle, co poprawia wydajność aplikacji.
- Zapytania są wykonywane na specjalnie przygotowanych dla nich danych, co ma bardzo pozytywny wpływ na wydajność.
- Architektura CQRS dobrze współgra z:
 - wzorcem EventSourcing:
 - śledzenie i zapisywanie wszystkich zdarzeń w systemie,
 - możliwość cofnięcia stanu systemu do dowolnego punktu w czasie;
 - bazami danych typu NoSQL (np. Cassandra):
 - szybki odczyt danych, promowanie denormalizacji,
 - odporność na błędy, duża wydajność dla dużych ilości danych.

Architektura CQRS: wady

- Aby każde zapytanie mogło być obsłużone jak najszybciej, model zapytań musi być w dużym stopniu zdenormalizowany.
- Zarządzanie dwoma modelami tej samej dziedziny może sprawiać problemy.
- Jako że oba modele reprezentują tę samą dziedzinę, w systemie pojawia się duplikacja.
- Synchronizacja obu modeli może być kłopotliwa.

Problem

- Aby każde zapytanie mogło być obsłużone jak najszybciej, model zapytań musi być w dużym stopniu zdenormalizowany.
 - Zarządzanie dwoma modelami tej samej dziedziny może sprawiać problemy.
 - Jako że oba modele reprezentują tę samą dziedzinę, w systemie pojawia się duplikacja.
-
- Jak zarządzać dwoma modelami tej samej dziedziny?

Zaproponowane rozwiązanie

- Koncepcja: zapisywanie dziedziny systemu w postaci pojedynczego opisu, który będzie zawierał informacje na temat obu modeli.
- Implementacja (weryfikacja koncepcji): stworzenie generatora, który na podstawie opisu wygeneruje kod implementujący oba modele.
- Wymagania:
 - opis dziedziny powinien być zrozumiały dla osób niebędących programistami;
 - generator powinien generować jak najwięcej artefaktów systemu związanych z jego dziedziną.

Przykładowa aplikacja

- Platforma blogowa.
- Komendy pozwalające użytkownikowi:
 - opublikować wpis na swoim blogu;
 - polubić wpis innego użytkownika;
 - skomentować wpis innego użytkownika.
- Zapytania na potrzeby:
 - wyświetlenia danego bloga, tzn. wszystkich wpisów jego autora wraz z liczbą ich komentarzy;
 - wyświetlenia danego wpisu wraz z jego komentarzami i “polubieniami”;
 - wyświetlenia profilu użytkownika wraz z lubianymi przez niego wpisami.

Opis dziedziny aplikacji: JSON

```
{
  "Name": "User",
  "IsView": true,
  "Fields": [
    {
      "Name": "UserName",
      "Type": "text",
      "IsKey": true,
      "PresentInViews": [ { "Name": "PostLike", "IsKey": true }, { "Name": "UserLike", "IsKey": true },
                        { "Name": "Post", "IsSearchable": true }, { "Name": "Comment" } ],
      "PresentInEvents": ["PostPublishedEvent", "PostLikedEvent", "PostCommentedEvent"]
    },
    {
      "Name": "FirstName",
      "Type": "text",
      "PresentInViews": [ { "Name": "UserLike" } ]
    },
    {
      "Name": "LastName",
      "Type": "text",
      "PresentInViews": [ { "Name": "UserLike" } ]
    }
  ]
}
...
```

Opis dziedziny aplikacji: DSL

Events:

Post published: @User(UserName) of @User:UserName can publish a @Post(PostID[id]) with @Post:Title and @Post:Content.

Post liked: @User of @User:UserName can like a @Post by its @Post:PostID.

Post commented: @User of @User:UserName can write a @Comment(CommentID[id]) of @Comment:Content under a post of ID @Comment:PostID or another comment of ID @Comment:ParentCommentID[id].

=====

Views:

I can display @Post list with @Post:@User:UserName, @Post:Title, @Post:Content and @Post:CommentsNumber[int].

I can display @Post details with @Post:@User:UserName, @Post:Title, @Post:Content, @Post:@Comment:Content, @Post:@Comment:@User:UserName, @Post:@User:FirstName and @Post:@User:LastName.

I can display user profile with @User:UserName, @User:FirstName, @User:LastName and @User:@Post:Title.

Działanie generatora

PostPublishedEvent
UserName
Title
Content

PostLikedEvent
UserName
PostID

PostCommentedEvent
PostID
UserName
ParentCommentID
Content

User
UserName
FirstName
LastName

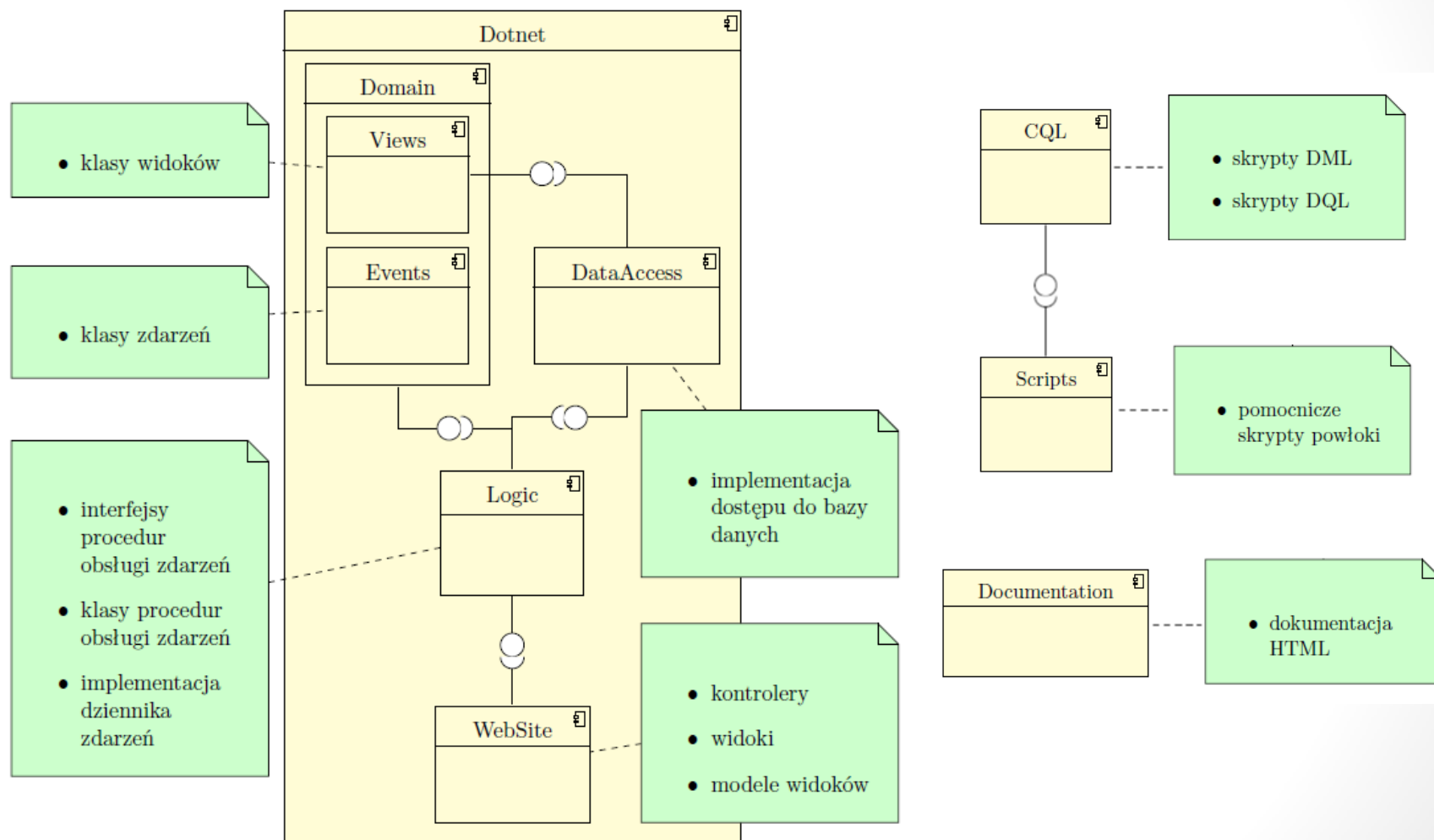
Post
PostID
UserName
Title
Content
CommentsNumber

Comment
CommentID
PostID
UserName
ParentCommentID
Content

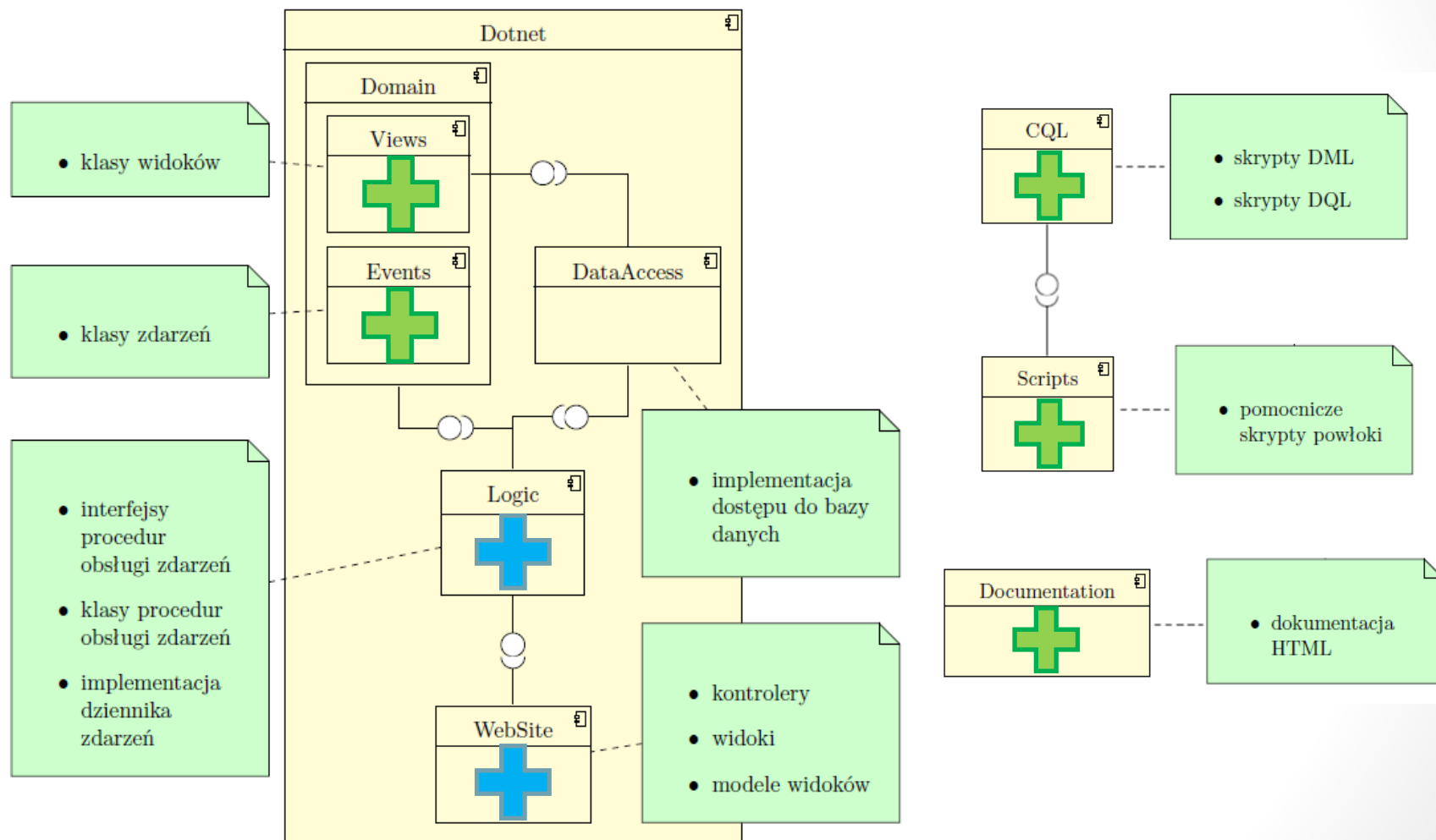
UserLike
UserName
PostID
PostTitle

PostLike
PostID
UserName
FirstName
LastName

Działanie generatora



Działanie generatora



Podsumowanie

- Rozwiązaniem problemu rozłącznych modeli dziedziny w CQRS jest generowanie tych modeli na podstawie pojedynczego opisu dziedziny.
- Język DSL umożliwia zaangażowanie osób niebędących programistami w proces tworzenia systemu.
- Opracowany generator generuje znaczną część artefaktów systemu.
- Stworzenie generatora potrafiącego wygenerować aplikacje każdego rodzaju wydaje się być niemożliwe – jednak opracowany generator można łatwo dostosować do potrzeb docelowej aplikacji.

Dziękuję za uwagę