ZAN KAVTASKIN

Musings about Software Engineering

Wednesday, 1 June 2016

Applied Domain-Driven Design (DDD), Part 7 - Read Model

When I first started using DDD I came across a really messy situation. I had my aggregate root and it linked it self to another child aggregate root. Everything worked really well. Shortly after everything was written new requirement came through, I had to expose counts and sums of data based on different filters. This was very painful, I ended up modifying my aggregate roots to try and provide these additional properties. This approach did not perform, for each aggregate root, it was loading another aggregate root with entities and summing them. I've played around with NHibernate mapping files and I've managed to make it performant. By this point I've optimized NHibernate mapping files and my aggregate roots were polluted with query methods. I really didn't like this approach. Shortly after I've came up with another idea, how about we create an immutable model that maps directly to the SQL view and we let the infrastructure handle the mapping? This way our aggregate roots will remain unaffected and we will get much better performance through SQL querying! This is when I have discovered the read model.

In this article we are going to explore how we can end up in this messy situation and why you should use the read model for data mash up and summarisation.

Let's recap how our fictional domain model looks like (omitted to show properties only):

```
public class Customer : IDomainEntity
{
    private List<purchase> purchases = new List<purchase>();

    public virtual Guid Id { get; protected set; }
    public virtual string FirstName { get; protected set; }
    public virtual string LastName { get; protected set; }
    public virtual string Email { get; protected set; }

    public virtual ReadOnlyCollection<purchase> Purchases { get { return this.purchases.AsReadOnly(); } }
}

public class Purchase
{
    private List<purchasedproduct> purchasedProducts = new List<purchasedproduct>();
```

Search This Blog

Search

About Me

Zan Kavtaskin

Nottingham, United Kingdom

I am a Software Director, Architect and Engineer. I work at MHR as a Software Delivery Director and I have also written software for companies such as Experian, Emirates and Royal Mail.

View my complete profile

Popular Posts

- Applied Domain-Driven Design (DDD), Part 1 - Basics
- Applied Domain-Driven Design (DDD), Part 0 - Requirements and Modelling
- Applied Domain-Driven Design (DDD), Part 2 - Domain Events
- Applied Domain-Driven Design (DDD), Part 3 - Specification Pattern
- Applied Domain-Driven Design (DDD), Part 4 - Infrastructure
- Applied Domain-Driven Design (DDD), Part 6 - Application Services
- Applied Domain-Driven Design (DDD), Part 5 - Domain Service
- Applied Domain-Driven Design (DDD), Part 7 - Read Model
- Applied Domain-Driven Design (DDD) - Event Logging & Sourcing For Auditing
- Unit Of Work Abstraction For NHibernate or Entity Framework C# Example

```
        public Guid Id { get; protected set; }
        public ReadOnlyCollection<purchasedproduct> Products
        {
            get { return purchasedProducts.AsReadOnly(); }
        }
        public DateTime Created { get; protected set; }
        public Customer Customer { get; protected set; }
        public decimal TotalCost { get; protected set; }
    }

    public class PurchasedProduct
    {
        public Purchase Purchase { get; protected set; }
        public Product Product { get; protected set; }
        public int Quantity { get; protected set; }
    }
```

Please notice the deep relationship between Customer, Purchase and Purchased Product.

New Requirement
Back office team has just come up with a brand new requirement. They need to get a list of customers that have made purchases, they want to see how much they have spent overall and how many products they have purchased. They are going to contact these customers, thank them for their custom, ask them few questions and give them discount vouchers.

Here is the DTO that we will need to populate and return back via API:

```
    public class CustomerPurchaseHistoryDto
    {
        public Guid CustomerId { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string Email { get; set; }
        public int TotalPurchases { get; set; }
        public int TotalProductsPurchased { get; set; }
        public decimal TotalCost { get; set; }
    }
```

#Approach 1 - Domain Model DTO Projection

```
        public List<CustomerPurchaseHistoryDto> GetAllCustomerPurchaseHistory()
        {
            IEnumerable<Customer> customers =
                 this.customerRepository.Find(new CustomerPurchasedNProductsSpec(1));

            List<CustomerPurchaseHistoryDto> customersPurchaseHistory =
                new List<CustomerPurchaseHistoryDto>();
```

Blog Archive
► 2020 (2)
► 2019 (1)
► 2018 (7)
► 2017 (5)
▼ 2016 (9)
► Dec (2)
► Aug (3)
▼ Jun (1)
Applied Domain-Driven Design (DDD), Part 7 - Read ...
► May (2)
► Feb (1)
► 2014 (3)
► 2013 (9)

```
            foreach (Customer customer in customers)
            {
                CustomerPurchaseHistoryDto customerPurchaseHistory = new CustomerPurchaseHistoryDto();
                customerPurchaseHistory.CustomerId = customer.Id;
                customerPurchaseHistory.FirstName = customer.FirstName;
                customerPurchaseHistory.LastName = customer.LastName;
                customerPurchaseHistory.Email = customer.Email;
                customerPurchaseHistory.TotalPurchases = customer.Purchases.Count;
                customerPurchaseHistory.TotalProductsPurchased =
                    customer.Purchases.Sum(purchase => purchase.Products.Sum(product => product.Quantity));
                customerPurchaseHistory.TotalCost = customer.Purchases.Sum(purchase => purchase.TotalCost);
                customersPurchaseHistory.Add(customerPurchaseHistory);

            }
            return customersPurchaseHistory;
        }
```

With this approach we have to get every customer, for that customer get their purchases, for that purchase get the products that were actually purchased and then sum it all up (lines 16-19). That's a lot of lazy loading. You could fine tune your NHibernate mapping so that it gets all of this data using joins in one go. However that will mean you will be getting unnecessary child data when you are interested only in the parent data (Customer). Also what if your domain-model is not exposing some of the data that you would like summarise? Now you have to add extra properties to your aggregate roots to make this work. Messy.

#Approach 2 - Infrastructure Read Model Projection

```
    /*Read only model, I don't think read models should have "readmodel" suffix.
    We don't suffix Customer, we don't write CustomerDomainModel or CustomerModel we just write Customer.
    We do this because it's part of the ubiquitous language, same goes for the CustomerPurchaseHistory.
    I've added this suffix here just to make things more obvious. */
    public class CustomerPurchaseHistoryReadModel
    {
        public Guid CustomerId { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int TotalPurchases { get; set; }
        public int TotalProductsPurchased { get; set; }
        public decimal TotalCost { get; set; }
    }

    public List<CustomerPurchaseHistoryDto> GetAllCustomerPurchaseHistory()
    {
        IEnumerable<CustomerPurchaseHistoryReadModel> customersPurchaseHistory =
                this.customerRepository.GetCustomerPurchaseHistory();

        return AutoMapper.Mapper.Map<IEnumerable<CustomerPurchaseHistoryReadModel>, List<CustomerPurchaseHistoryDto>>(customersPurchaseHistory);
    }
```

```
interface ICustomerRepository : IRepository<Customer>
{
    IEnumerable<CustomerPurchaseHistoryReadModel> GetCustomersPurchaseHistory();
}

public class CustomerNHRepository : ICustomerRepository
{
    public IEnumerable<CustomerPurchaseHistoryReadModel> GetCustomersPurchaseHistory()
    {
        //Here you either call a SQL view, do HQL joins, etc.
        throw new NotImplementedException();
    }
}
```

In this example we have created CustomerPurchaseHistoryReadModel which is identical to CustomerPurchaseHistoryDto, which means I can keep things simple and just use AutoMapper to do one to one mapping. I've extended IRepository by creating new interface ICustomerRepository and added custom method GetCustomersPurchaseHistory(). Now I need to fill in CustomerNHRepository.GetCustomersPurchaseHistory() method. As we are now in the infrastructure layer we can just write some custom HQL or query a SQL view.

Would like to see full working example?
Browse "Domain-Driven Design Example" Repository On Github

Summary:

- Don't use your entities and aggregate roots for properties mush up or summarisation. Create read models where these properties are required.
- Infrastructure layer should take care of the mapping. For example, use HQL to project data on to your read model.
- Reads models are just that, read only models. This is why they are performant and this is why they should have no methods on them and just properties (they are immutable).

Useful links:

- Read model as tactical pattern in domain-drive design - Lev Gorodinski

**Note: Code in this article is not production ready and is used for prototyping purposes only. If you have suggestions or feedback please do comment.*

Rate this blog post (11 Votes)

Posted by Zan Kavtaskin
Labels: domain-driven design, modelling, software engineering

2 comments:

for ict 99 11 November 2017 at 22:37

great

Reply

Nguyen Quang 2 July 2019 at 00:56

very useful

Reply

Enter your comment...

Comment as: ahm7dkhalifa@ ▼ Sign out

Publish Preview ☐ Notify me

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)