ZAN KAVTASKIN

Musings about Software Engineering

Saturday, 16 November 2013

# Applied Domain-Driven Design (DDD), Part 6 - Application Services

We have developed our domain, setup our infrastructure, now we need to expose our domain to Applications. This is where Application Service comes in.

Any kind of application should be able to consume your Application Service and use it, mobile, desktop or web. It's good to keep this in mind when you setup your Application Service and Distributed Interface Layer (Web Service).

**Application Service Contract (lives in a separate project):**

```
    public interface ICartService
    {
        CartDto Add(Guid customerId, CartProductDto cartProductDto);
        CartDto Remove(Guid customerId, CartProductDto cartProductDto);
        CartDto Get(Guid customerId);
        CheckOutResultDto CheckOut(Guid customerId);
    }

   //Dto's are Data Transfer Objects, they are very important as they allow you to input and get the output from Application Services without exposing the actual Domain.
    public class CartDto
    {
        public Guid CustomerId { get; set; }
        public List<CartProductDto> Products { get; set; }
        public DateTime Created { get; set; }
        public DateTime Modified { get; set; }
    }

    public class CartProductDto
    {
        public Guid ProductId { get; set; }
        public int Quantity { get; set; }
    }

    public class CheckOutResultDto
```

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 Nullable<Guid> PurchaseId { get; set; }
        public Nullable<CheckOutIssue> CheckOutIssue { get; set; }
    }
```

**Application Service Implementation:**

```
    public class CartService : ICartService
    {
        readonly IRepository<Customer> repositoryCustomer;
        readonly IRepository<Product> repositoryProduct;
        readonly IUnitOfWork unitOfWork;
        readonly ITaxDomainService taxDomainService;

        public CartService(IRepository<Customer> repositoryCustomer,
            IRepository<Product> repositoryProduct, IUnitOfWork unitOfWork, ITaxDomainService taxDomainService)
        {
            this.repositoryCustomer = repositoryCustomer;
            this.repositoryProduct = repositoryProduct;
            this.unitOfWork = unitOfWork;
            this.taxDomainService = taxDomainService;
        }

        public CartDto Add(Guid customerId, CartProductDto productDto)
        {
            CartDto cartDto = null;
            Customer customer = this.repositoryCustomer.FindById(customerId);
            Product product = this.repositoryProduct.FindById(productDto.ProductId);

            this.validateCustomer(customerId, customer);
            this.validateProduct(product.Id, product);

            decimal tax = this.taxDomainService.Calculate(customer, product);

            customer.Cart.Add(CartProduct.Create(customer.Cart, product, productDto.Quantity, tax));

            cartDto = Mapper.Map<Cart, CartDto>(customer.Cart);
            this.unitOfWork.Commit();
            return cartDto;
        }

        public CartDto Remove(Guid customerId, CartProductDto productDto)
        {
            CartDto cartDto = null;
```

**Blog Archive**

- ► 2020 (2)
- ► 2019 (1)
- ► 2018 (7)
- ► 2017 (5)
- ► 2016 (9)
- ► 2014 (3)
- ▼ 2013 (9)
  - ► Dec (3)
  - ▼ Nov (3)
    - Applied Domain-Driven Design (DDD), Part 6 - Appli...
    - Applied Domain-Driven Design (DDD), Part 4 - Infra...
    - Applied Domain-Driven Design (DDD), Part 5 - Domai...
  - ► Oct (1)
  - ► Sep (2)

```
            Customer customer = this.repositoryCustomer.FindById(customerId);
            Product product = this.repositoryProduct.FindById(productDto.ProductId);

            this.validateCustomer(customerId, customer);
            this.validateProduct(productDto.ProductId, product);

            customer.Cart.Remove(product);
            cartDto = Mapper.Map<Cart, CartDto>(customer.Cart);
            this.unitOfWork.Commit();
            return cartDto;
        }

        public CartDto Get(Guid customerId)
        {
            Customer customer = this.repositoryCustomer.FindById(customerId);
            this.validateCustomer(customerId, customer);
            return Mapper.Map<Cart, CartDto>(customer.Cart);

        }

        public CheckOutResultDto CheckOut(Guid customerId)
        {
            CheckOutResultDto checkOutResultDto = null;
            Customer customer = this.repositoryCustomer.FindById(customerId);
            this.validateCustomer(customerId, customer);

            Nullable<CheckOutIssue> checkOutIssue = customer.Cart.IsCheckOutReady();

            if (!checkOutIssue.HasValue)
            {
                Purchase purchase = customer.Cart.Checkout();
                checkOutResultDto = Mapper.Map<Purchase, CheckOutResultDto>(purchase);
                this.unitOfWork.Commit();
            }

            return checkOutResultDto;
        }

        //this is just an example, don't hard code strings like this, use reference data or error codes
    private void validateCustomer(Guid customerId, Customer customer)
        {
            if (customer == null)
                throw new Exception(String.Format("Customer was not found with this Id: {0}", customerId));
        }

        private void validateProduct(Guid productId, Product product)
        {
            if (product == null)
                throw new Exception(String.Format("Product was not found with this Id: {0}", productId));
        }
    }
```

**Example usage:**

```
            this.cartService.Add(
                this.customer.Id,
                new CartProductDto()
                {
                    ProductId = viewModel.ProductId,
                    Quantity = 2
                }
            );

            CheckOutResultDto checkoutResult = this.cartService.CheckOut(this.customer.id);
```

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

**Summary:**

- Application Service is a gateway in to your Domain Model Layer via Dto's (Data Transfer Objects)
- Application Service should not encapsulate any domain logic, it should be really thin
- Application Service method should do only one thing and do it well with one region of the domain, don't mix it to "make it more performance efficient for the Application that's consuming it".
- To access Application Service you expose interface and Dto's for inputs and outputs (it's important not to expose your Domain Entity in a raw format, Dto is a proxy and it protects your domain)
- Presenter (mobile app, desktop or web), should call different services to get data it needs and manipulate it to suit the UI. This might seem inefficient, or wasteful at first. You will realise that actually it's just as fast (if not faster), easier to test and maintain.

**Tips:**

- Use AutoMapper to map your Domain Entity to Dto's, don't waste your time with manual mapping. It clutters your implementation code and maintenance becomes a nightmare.
- Don't think of screens when it comes to exposing Application Services, it's an API, think how a mobile app would access it, or how external potential customers would use it.
- Realise that you will end up writing Application Services that suit your UI. This is only natural as this is what you been doing for a while. It will take a few goes before you change your thinking.
- Application Service can be consumed directly if you don't need distribution i.e. your MVC app will just reference Application Service directly, you can then just try and catch errors in your Controller.
- Application Service can be exposed via Web Service (Distributed Interface Layer). This further abstraction give you ability to "try and catch" errors so they can be exposed in a friendlier manner. Additionally it allows you to future proof your application e.g. versioning.

**Useful links:**

- SOA.com Service-oriented architecture explained in 7 steps

**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 (18 Votes)

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

## 10 comments:

**Unknown** 27 February 2016 at 16:53

I saw several validation of product and customer in your application service. But, would you put product availability (quantity) validation in here as well? Or would you have the purchase entity to do the validation?

Second, how do you update product quantity availabilities? Is it good idea to create event handler of PurchaseCreated and reduce product quantity there?

Reply

Replies

**Zan Kavtaskin** 24 April 2016 at 07:41

Hi, thanks for the comment!

1. Application Service & Entity Validation
Application service should be super thin and it should not contain any logic. So in this particular case I would probably put quantity check inside the cart class (see https://github.com/zkavtaskin/Domain-Driven-Design-Example/blob/master/eCommerce/DomainModelLayer/Carts/Entity/Cart.cs , methods such as a Cart.Checkout could be a great place for it.

2. Outside of aggregate root data updates
Your cart should not have access to all of the products in the system, so you can't really do anything there. So yes I would recommend that you raise a domain event and have a handler update the quantity as soon as the purchase was made.

I hope this helps and sorry for the late reply. Let me know if this has answered your question!

**Reply**

**Unknown** 5 January 2017 at 10:46

Will Presentation layer be responsible for initializing repositories before they are passed to CartService? I thought Presentation layer should be unaware of persistence/infrastructure, no? What is the best practice?

Reply

Replies

**Zan Kavtaskin** 6 January 2017 at 13:43

Hi Sergei,

Thank you for reading.

Dependency Injection will initialize your repositories. Please take a look at this:

Presentation
https://github.com/zkavtaskin/Domain-Driven-Design-Example/blob/master/eCommerce.WebService/Controllers/ProductController.cs

Service
https://github.com/zkavtaskin/Domain-Driven-Design-Example/blob/master/eCommerce/ApplicationLayer/Products/ProductService.cs

Dependency Injection Installation
https://github.com/zkavtaskin/Domain-Driven-Design-Example/blob/master/eCommerce.WebService/App_Start/BootstrapConfig.cs

**Reply**

**Unknown** 27 June 2017 at 06:21

What about the infrastructure validation like length and uniqueness, where would I validate it?

Reply

Replies

**Zan Kavtaskin** 5 July 2017 at 07:42

Hello Lucas,

I think that that uniqueness and length of the data is not an infrastructure concern, it’s part of the domain.

For example, here is the Customer.cs class:
https://github.com/zkavtaskin/Domain-Driven-Design-Example/blob/master/eCommerce/DomainModelLayer/Customers/Customer.cs

When you create a customer, you could just check the length of the property.

Also, if you creating a customer and they must have a unique email address you could use this approach (see Add method):
https://github.com/zkavtaskin/Domain-Driven-Design-Example/blob/master/eCommerce/ApplicationLayer/Customers/CustomerService.cs

You could say that this is business logic and it should be part of the domain, and I would agree with that. This logic could reside inside a domain service or a factory method.

From data consistency point of view there is nothing wrong with applying length and uniqueness constraints at the infrastructure level. As you already have this validation in the domain, infrastructure should never throw an error.

I hope this helps!

Reply

**Unknown** 23 July 2018 at 00:53

well I don't really understand why CartService is application service instead of domain service. with medthod validateCustomer() and validateProduct(), these are not business logic?

Reply

**Unknown** 5 September 2018 at 12:46

Hi Zan,

A couple of questions about the application service vs domain layer.

You state: Application Service should not encapsulate any domain logic, it should be really thin.

Let's take a specific example from your code.

When a customer is created, where should the logic for checking the input go?

I see in the domain layer, in the static create method you check if firstname/lastname/email are empty and throw an exception if that is the case.

But in the application layer, in the customerservice you check if the email is already registered before creating the customer.

Is there a rule or best practice of how and where to place these validations? If the customerservice checks for the existence of the email, why not validate firstname/lastname/email and so forth? If it all should be done in the domain layer, then why is it in the application layer?

The other question is that the create method in the domain layer is throwing an exception when the firstname is not provided (for example). This means that the application service has to use a try/catch to control execution flow? Is there a better way to handle this other then just throwing an exception?

Thanks in advance.

Reply

**Unknown** 7 September 2018 at 06:30

Hi Zan,

Great writeup. One question about application service and validation. You say that application layer should be thin and have no if statements. You are using an if statement though to check in customer exists in the application layer, instead of the domain.

What is the best practice for this? Should that be in the application or the domain layer?

Thanks

Reply

**Unknown** 12 September 2018 at 13:26

Is this page still active or has it been abandoned, because I've asked a few questions, and no answers yet?

Reply

Enter your comment...

Comment as: ahm7dkhalifa@ Sign out

Publish Preview Notify me

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)