ZAN KAVTASKIN

Musings about Software Engineering

Saturday, 16 November 2013

Applied Domain-Driven Design (DDD), Part 4 - Infrastructure

If you come from database centric development (where database is the heart of the application) then this is going to be hard for you. In domain-driven design database and general data sources are not important, your application is persistence ignorant.

Put your infrastructure interfaces in to Domain Model Layer. Your domain will use them to get data, it doesn't need to care how, it just knows there is an interface exposed and it will use it. This simplifies things and allows you to focus on your actual Domain rather worrying about what database you will be using, where data is coming from, etc.

Infrastructure Contracts:

```
public interface IEmailDispatcher
{
    void Dispatch(MailMessage mailMessage);
}

public interface INewsletterSubscriber
{
    void Subscribe(Customer customer);
}

//this lives in the core library and you can inherit from it and extend it e.g. ICustomerRepository : IRepository<Customer> then you can add some custom methods to your new interfa
public interface IRepository<TEntity>
    where TEntity : IDomainEntity
{
    TEntity FindById(Guid id);
    TEntity FindOne(ISpecification<TEntity> spec);
    IEnumerable<TEntity> Find(ISpecification<TEntity> spec);
    void Add(TEntity entity);
    void Remove(TEntity entity);
}
```

Infrastructure Implementation (lives in the Infrastructure Layer):

```
public class NHibernateRepository<TEntity> : IRepository<TEntity>
        where TEntity : IDomainEntity
    {
```

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

Blog Archive

- 2020 (2)
- 2019 (1)
- 2018 (7)
- 2017 (5)

```
    public TEntity FindById(Guid id)
    {
        throw new NotImplementedException();
    }

    public TEntity FindOne(ISpecification<TEntity> spec)
    {
        throw new NotImplementedException();
    }

    public IEnumerable<TEntity> Find(ISpecification<TEntity> spec)
    {
        throw new NotImplementedException();
    }

    public void Add(TEntity entity)
    {
        throw new NotImplementedException();
    }

    public void Remove(TEntity entity)
    {
        throw new NotImplementedException();
    }
}

public class SmtpEmailDispatcher : IEmailDispatcher
{
    public void Dispatch(MailMessage mailMessage)
    {
        throw new NotImplementedException();
    }
}

public class WSNewsletterSubscriber : INewsletterSubscriber
{
    public void Subscribe(Customer customer)
    {
        throw new NotImplementedException();
    }
}
```

Example usage:

```
public class CustomerCreatedHandle : Handles<CustomerCreated>
{
    readonly INewsletterSubscriber newsletterSubscriber;
    readonly IEmailDispatcher emailDispatcher;

    public CustomerCreatedHandle(INewsletterSubscriber newsletterSubscriber, IEmailDispatcher emailDispatcher)
    {
        this.newsletterSubscriber = newsletterSubscriber;
```


- ▶ 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)

```
            this.emailDispatcher = emailDispatcher;
        }

        public void Handle(CustomerCreated args)
        {
            //example #1 calling an interface email dispatcher this can have differnet kind of implementation depending on context, e.g
            // smtp = SmtpEmailDispatcher (current), exchange = ExchangeEmailDispatcher, msmq = MsmqEmailDispatcher, etc... let infrastructure worry about it
            this.emailDispatcher.Dispatch(new MailMessage());

            //example #2 calling an interface newsletter subscriber  this can differnet kind of implementation e.g
            // web service = WSNewsletterSubscriber (current), msmq = MsmqNewsletterSubscriber, Sql = SqlNewsletterSubscriber, etc... let infrastructure worry about it

            this.newsletterSubscriber.Subscribe(args.Customer);
        }
    }
```

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

Summary:

- Infrastructure contains implementation classes that actually talks to the infrastructure IO, Sql, Msmq, etc.
- Domain is the heart of the application not the Infrastructure (this can be hard to grasp if you come from DBA background).
- Infrastructure is not important in Domain-design design, it facilitates the application development doesn't lead it.
- Infrastructure should not contain any domain logic, all domain logic should be in the domain. (i guarantee that when you first start out, you will put logic in there without knowing it)

Tips:

- When it comes to repositories try and just use a generic repository and stay away from custom implementations as much as possible i.e. IRepository<Customer> = good, ICustomerRepository = bad (it's never that simple, but a good general rule to work with).
- When you first start out with infrastructure implementations, force your self not to put any if statements in to it. This will help your mind adjust to leaving all logic out of the Infrastructure Layer.
- Take your time and try and understand what persistence ignorance really means, also try and research polyglot persistence this will expand your understanding.

Useful links:

- Onion architecture (i really don't like the name) showcases well how you need to start thinking about infrastructure layer - Jeffrey Palermo
- Polyglot Persistence - Martin Fowler

**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 (22 Votes)

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

13 comments:

Unknown 3 March 2016 at 06:42

How do I do if it take some time to get answer from the infrastructure?
I have to get some data from a FTP server and it take some seconds up to some minutes.

Reply

Replies

Zan Kavtaskin 24 April 2016 at 07:11

Hi, thanks for the comment!

FTP is just a way of getting data, I would treat it the same as a web service. As it's infrastructure I would also abstract it away.

If your application needs to read this large dataset then you have 3 options (off the top of my head):
1. Make data smaller
2. Reduce latency time
3. Accept the time it takes

1. Make data smaller
To do this you can do 3 things, extract less data (get just what you need), page it (get a small chunk of it at the time) or compress it.

```
public interface ISomeData
{
IEnumerable GetSomeData();

IEnumerable GetSomeData(int pageNumber, int numberOfResults);
}

public class FTPSomeData : ISomeData
{
public IEnumerable GetSomeData()
{
//de-compress data here
//de-serialize here
}

public IEnumerable GetSomeData(int pageNumber, int numberOfResults)
{
//de-compress data here
//de-serialize here
//page
}
}
```

To make read processes faster you could also pre-process this data and make it read friendly by pre-paging it, reformatting it so that there is less data to get or compress it and let the CPU do more work.

2. Reduce latency time
You could take all of the data from the FTP server and cache it on the local application server so that there is less latency. This might be a big one if the FTP server is on the other side of the world.

3. Accept the time it takes
If the dataset is very large, then it's large. Increase the timeout, put a nice loading bar on to the screen and make users wait.

I recommend options 1 and 2, however it depends on your situation.

The key things is that your domain layer should not know how FTP does it's magic. Hence FTPSomeData should be inside the infrastructure layer.

I hope this helps and sorry for the late reply.
Let me know if this has answered your question and what you have decided to do.

Unknown 25 April 2016 at 02:44

Thanks for your reply.

Unfortunately, the FTP server is an old system that I'm not able to change. So option 1 is unfortunately no alternative.

I try to cache the data I download to accelerate repeated downloads. It is not a big deal if the data presented is little outdated, but when the data is changed little all the time, I must download new data when requested.

One idea I've been thinking about is that when a user request a particular data, it is returned directly from the cached data (if any). At the same time, a download from the FTP server is initialized, and when the data has been retrieved;the user

interface is updated. I think it is acceptably to the user under the conditions given that it works so. But the question is how to implement it.

Reply

youtubeline 18 April 2016 at 23:22

CustomerCreated class has a clean Handle method code with used interfaces. But MailMessage will have multiple parameters like From, To[], mail body, mail title. I will have email rules that send email to system admins. And these stuations will be dirty Handle() code. How can we fix this?

Reply

Replies

Zan Kavtaskin 24 April 2016 at 06:19

Hi, thanks for the comment!

You need to keep your Handle clean i.e. it should not care about To, From, etc. It needs to care about your domain objects only. So you would pass Customer, Invoice and other objects in to it and then inside the handle you would actually extract the required data from the objects and prepare your MailMessage, so it would look something like this:

```
public void Handle(CustomerCreated args)
{
MailMessage msg = new MailMessage();
msg.To = args.Customer.Email;
msg.From = //my company from address config
msg.Body = String.Format("Hello {0} {1}, to active your email click here...",
args.Customer.FirstName, args.Customer.LastName);

this.emailDispatcher.Dispatch(msg);
}
```

Let me know if this makes sense, if it doesn't I will update my domain-driven design Github repository (https://github.com/zkavtaskin/Domain-Driven-Design-Example) with an example.

Reply

Unknown 20 December 2016 at 19:00

For those using EF, here's a generic Entity Framework Repository example (and general writeup on repositories): http://deviq.com/repository-pattern/

Also for caching, keep caching responsibility separate from your persistence logic by using the CachedRepository pattern: http://ardalis.com/introducing-the-cachedrepository-pattern

Reply

Trung 26 July 2017 at 00:50

Dear Zan,
Thank you very much for your post. It helps me a lot.
However, I don't see the implementation of IRepository in your code. Could you explain why it works?
Thank you very much!

Reply

Replies

Trung 26 July 2017 at 19:40

I mean I don't see the implementation of Irepository in your application.

Zan Kavtaskin 27 July 2017 at 15:36

Hello Trung,

Thank you for the question, unfortunately I am going to provide a lengthy answer.

Simple example of this implementation can be found here:
https://github.com/zkavtaskin/UnitOfWork

Interface is defined here:
https://github.com/zkavtaskin/UnitOfWork/blob/master/UnitOfWork/RepositoryOfTEntity.cs

NHibernate implementation can be found here:
https://github.com/zkavtaskin/UnitOfWork/blob/master/UnitOfWork/NH/NHRepositoryOfTEntity.cs

Here NHRepository is constructed:
https://github.com/zkavtaskin/UnitOfWork/blob/master/Application/Program.cs

This is the key line:
new CustomerService(uowNH, new NHRepository(uowNH));

and CustomerService is using NHRepository(uowNH) here as IRepoistory:
https://github.com/zkavtaskin/UnitOfWork/blob/master/Application/CustomerService.cs

If you are feeling comfortable with the above, you can take a look at this repository:
https://github.com/zkavtaskin/Domain-Driven-Design-Example it is a bit more confusing as it's using dependency injection to construct classes and it's using decorator pattern, dependency injection is registered here:
https://github.com/zkavtaskin/Domain-Driven-Design-Example/blob/master/eCommerce.WebService/App_Start/Installers/InfrastructureLayerInstall.cs

These two lines are important:
container.Register(Component.For(typeof(IRepository<>),
typeof(MemoryRepository<>)).ImplementedBy(typeof(MemoryRepository<>)).LifestyleSingleton());

container.Register(Component.For().ImplementedBy().LifestyleSingleton());

First line constructs MemoryRepository with IRepository interface and then this is injected into StubDataCustomerRepository which can be found here:
https://github.com/zkavtaskin/Domain-Driven-Design-Example/blob/master/eCommerce/InfrastructureLayer/StubDataCustomerRepository.cs

Second line constructs StubDataCustomerRepository.cs with ICustomerRepository and IRepository interface, this can be found here:
https://github.com/zkavtaskin/Domain-Driven-Design-Example/blob/master/eCommerce/DomainModelLayer/Customers/ICustomerRepository.cs

It might seem confusing as you have two classes that implement the same interface but you have two different implementations and one of them is injected into the other. This is the decorator pattern in action. I've used decorator pattern as I needed the StubDataCustomerRepository to add a fake customer into the memory repository.

I hope this helps!

Reply

Unknown 26 July 2017 at 11:20

See my links, Trung, for IRepository implementations.

Reply

Trung 27 July 2017 at 20:50

Thank you Zan and Steve for swift reply. Now I understand how they are implemented. It really helps!

Trung.

Reply

Unknown 11 February 2018 at 17:03

Zan,

I have a question about the performance in DDD apps versus database-centered apps. It appears to me (I could be wrong) that by placing the app logic in the domain vs database (i.e stored procedures) that in DDD apps there are more calls to the database for the equivalent of what is usually handled by one stored procedure.

This isn't made obvious with simple examples, but it appears to me that in more complex apps there is no way a DDD approach could come close to the performance of what stored procedures give us. After all you're still spending a lot of time tweaking how the ORM is going to generate the right sql, which defies the purpose of persistence ignorance.

Am I misconstruing the issue?

Reply

Replies

Zan Kavtaskin 19 February 2018 at 05:37

Hello,

Great question.

You are correct, when it comes to the command stack, by calling a store procedure with few parameters and letting store procedure do all of the data manipulation would be faster. Mainly because there is no latency between application server and the database. However, from my experience you will face maintainability and testing problems pretty quickly. As your store procedures start to depend on each other and as you couple to schema it will be harder and harder to change. This is OK for small applications, however when you are building large scale applications, personally I would not use this approach.

Your aggregate roots should be small so the amount of data loading is done would be limited. Also, ORM can be tuned to get data in one go, but again, there is still latency, serialisation, etc. Tuning ORM is part of the infrastructure layer so your domain would still be persistence ignorant.

This is not as much of an issue when it comes to the read stack, with the read stack you could use Read Model:
http://www.zankavtaskin.com/2016/06/applied-domain-driven-design-ddd-part-7.html

As a rule of thumb, I like to keep "premature optimisation is root of all evil" quote in my mind. From my experience the latency is negligible (in most cases) and would not impact the overall application performance. While application is waiting for few milliseconds for the response from the database server, your application server will be serving other requests. This is even less of an issue if you are using async command/read stack.

Reply

Enter your comment...

Comment as: ahm7dkhalifa@ ▾ Sign out

Publish Preview Notify me

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)