ZAN KAVTASKIN

Musings about Software Engineering

Monday, 29 August 2016

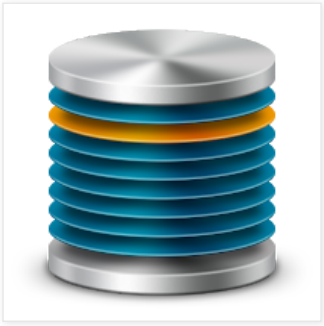
Applied Domain-Driven Design (DDD) - Event Logging & Sourcing For Auditing

In this article I am going to explore the use of Event Logging and Sourcing as a solution for domain auditing. This article is not going to explore how to use Event Sourcing to obtain the current model state.

What is Event Logging?

In my previous article I've explored domain events. In that article they were synchronous unpersisted events. Aggregate root or Service would just raise an event and a handler would handle it. In this article we are going to change that, we are going to persist these domain events.

What is Event Sourcing?



"Append-only store to record the full series of events that describe actions taken on data in a domain, rather than storing just the current state, so that the store can be used to materialize the domain objects. This pattern can simplify tasks in complex domains by avoiding the requirement to synchronize the data model and the business domain; improve performance, scalability, and responsiveness; provide consistency for transactional data; and maintain full audit trails and history that may enable compensating actions." - Event Sourcing Pattern Microsoft

Requirements domain Event Logging and Sourcing can fulfil:

- As a technical support member of staff I would like to be able to view audit log so that I can find out what my customers did i.e. did they get themselves in to a mess or is it that our software is buggy?
- As a system admin I would like to be able to view the audit log so that I can find out what my users are doing i.e. someone is not sure why something was changed, software admin needs to double check what happened.
- As a security analyst I would like to view audit log so that I can find out who has committed fraud.
- As a business expert I would like to find out how long it has taken someone to go through a process so that we can optimise it.
- As a security analyst I would like audit log to be immutable so that no one can tamper with it
- As a software engineer I would like to see what user has done so that I can re-produce their steps and debug the application.
- As a software engineer I would like persisted domain events to be forwarded to the queue as we can't have 2 phase commit in the Cloud.

Why not just use CQRS with Event Sourcing?

As it was mentioned by Udi, CQRS is a pattern that should be used where data changes are competitive or collaborative. A lot of systems don't fall in to this category, even if they do, you would only use CQRS potentially with Event Sourcing (CQRS != Event Sourcing) for a part of the application and not everywhere. This means you can't have automatic audit for your entire system by using CQRS with Event Sourcing.

Event Sourcing is all about storing events and then sourcing them to derive the current model state.
If you don't need "undo" and "replay" functionality, and if you don't need to meet super high scalability non-functional requirements (which most likely you don't) why over-engineer?

This proposed solution is just logging events to get some of the benefits that Event Sourcing provides without the deriving the current model state. However, it will still be sourcing the events to obtain the audit log.

Why is this a good solution for auditing?

Your domain is rich and full of domain events (domain event is something that has happened, it's an immutable fact and you are just broadcasting it). It's also written using ubiquitous language. Because it describes what has happened and what was changed it's a great candidate to meet your auditing, troubleshooting, debugging and 2 phase commit Cloud requirements.

Pros:

- It's fairly easy to create audit read model from domain events
- Domain events provide business context of what has happened and what has changed
- Reference data (Mr, Dr, etc) is stored in the same place so you can provide full audit read model
- Events can be written away to append only store
- Only useful event data is stored

Cons:

- Every request (command) must result in domain event and you need to flatten it, it's more development work
- Requires testing

Blog Archive

- ► 2020 (2)
- ► 2019 (1)
- ► 2018 (7)
- ► 2017 (5)
- ▼ 2016 (9)
  - ► Dec (2)
  - ▼ Aug (3)
    - Applied Domain-Driven Design (DDD) - Event Logging...
    - Creating Custom Key Store Provider for SQL Always ...
    - How to use TypeScript with FlotCharts or any other...
  - ► Jun (1)
  - ► May (2)
  - ► Feb (1)
- ► 2014 (3)
- ► 2013 (9)

- Duplication of data. One dataset for current state. Second dataset for events. There might be mismatch due to bugs and changes.

What about "proof of correctness"?

Udi, has already discussed this here (scroll down to the "proof of correctness").

I recommend that you keep your storage transaction logs, it doesn't give you proof of correctness however it gives you extra protection. If someone bypasses your application and tampers with your data in the database at least it will be logged and you will be able to do something about it.

Domain event logging implementation example

I am going to take my previous article and build upon it. I've introduced in the past this interface:

```
public interface IDomainEvent {}
```

IDomainEvent interface was used like this:

```
public class CustomerCheckedOut : IDomainEvent
{
    public Purchase Purchase { get; set; }
}
```

We are going to change IDomainEvent to DomainEvent:

```
public abstract class DomainEvent
{
    public string Type { get { return this.GetType().Name; } }

    public DateTime Created { get; private set; }

    public Dictionary<string, Object> Args { get; private set; }

    public DomainEvent()
    {
        this.Created = DateTime.Now;
        this.Args = new Dictionary<string, Object>();
    }
```

```
        public abstract void Flatten();
    }
```

This new DomainEvent will:

1. Give you a timestamp for when domain event was created
2. Get the domain event name
3. Force events to flatten its payloads
4. Stores important arguments against the event

Here is example implementation:

```
    public class CustomerCheckedOut : DomainEvent
    {
        public Purchase Purchase { get; set; }

        public override void Flatten()
        {
            this.Args.Add("CustomerId", this.Purchase.Customer.Id);
            this.Args.Add("PurchaseId", this.Purchase.Id);
            this.Args.Add("TotalCost", this.Purchase.TotalCost);
            this.Args.Add("TotalTax", this.Purchase.TotalTax);
            this.Args.Add("NumberOfProducts", this.Purchase.Products.Count);
        }
    }
```

Flatten method is used to capture important arguments against the event. How you flatten really depends on your requirements. For example if you want to store information for audit purposes, then above flatten might be good enough. If you want to store events so that you can "undo" or "replay" you might want to store more information.

Why have Flatten method at all? Why not serialise and store the entire "Purchase" object? This object might have many value objects hanging of it, it might also have an access to another aggregate root. You will end up storing a lot of redundant data, it will be harder to keep track of versions (if your object shape changes, which it will) and it will be harder to query. This is why Flatten method is important, it strips away all of the noise.

We don't want to handle all event flattening and persisting manually. To simplify and automate the event handling process I've introduced generic event handler:

```
    public class DomainEventHandle<TDomainEvent> : Handles<TDomainEvent>
        where TDomainEvent : DomainEvent
    {
        IDomainEventRepository domainEventRepository;

        public DomainEventHandle(IDomainEventRepository domainEventRepository)
        {
```

```
            this.domainEventRepository = domainEventRepository;
        }

        public void Handle(TDomainEvent args)
        {
            args.Flatten();
            this.domainEventRepository.Add(args);
        }
    }
```

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

Extending this to meet additional security and operational requirements

You can take this few steps further and create a correlation id for the entire web request. This way you will be able to correlate IIS W3C logs, event logs and database logs. Find out how you can achieve this here.

**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 (10 Votes)

Posted by Zan Kavtaskin
Labels: Audit, domain-driven design, Event Sourcing, Security, software engineering

12 comments:

Pavan 13 March 2017 at 00:34

Dear Zan,

I've downloaded the example code from Github. Could you please let me know where you are saving the Domain Events? Can we save it to the log tables of SQL Server? How about the audit fields like Modified Date, Modified By, Created Date, Created By? I'm now having these fields in a class called AuditEntity.

Could you please throw some light on these Domain Events..

Thanks,
Pavan

Reply

Replies

Zan Kavtaskin 18 March 2017 at 16:46

Hello Pavan,

Thank you for the question!

I've created class called "DomainEventHandle", you can find it here:
https://github.com/zkavtaskin/Domain-Driven-Design-Example/blob/master/eCommerce/Helpers/Domain/DomainEventHandle.cs

This class is registered using dependency injection here:
https://github.com/zkavtaskin/Domain-Driven-Design-Example/blob/master/eCommerce.WebService/App_Start/Installers/DomainModelLayerInstall.cs

What happens is:
DomainEvents.Raise is invoked.

.Raise method resolves everything that handles CustomerCreated event.

In this case there are 2 handles, CustomerCreatedHandle and DomainEventHandle, these two handles get resolved and called.

DomainEventHandle flattens the event and saves it to some storage (this can be any storage).

DomainEvent abstract class can be found here:
https://github.com/zkavtaskin/Domain-Driven-Design-Example/blob/master/eCommerce/Helpers/Domain/DomainEvent.cs

You can save the following metadata against the event:
Created (date)
CreatedBy (you will need to figure out how you are going flow the relevant identity down)

You can't save the following:
Modified by
Modified (date)

Why? Because events are immutable, they have happened and they can't be changed.

While I was writing this I realised that DomainEventHandle is not the best name for the class, better name would be something like DomainEventPersistenceHandle.

I hope this helps.

Reply

Anonymous 6 April 2017 at 12:44

Hi Zan! How do you recommend to set up OccuredOn property on Domain Event? I don't want to initialize this property directly in the event with DateTime.UtcNow. I want to take the time value from ISystemClock implementation. How would you suggest to do it?

Reply

Replies

Zan Kavtaskin 11 April 2017 at 13:15

Hi,

Thank you for the question!

I would use dependency injection and inject ISystemClock into the constructor, so it will look something like this:

```
public class SomethingHandle : Handles
{
readonly ISystemClock systemClock;

public CustomerCreatedHandle(ISystemClock systemClock)
{
this.systemClock = systemClock;
}

public void Handle(Something args)
{
//Use systemClock here.
}
}
```

Reply

Anonymous 11 April 2017 at 23:29

And then you should pass systemClock as a parameter to any Method (even to constructor) on the aggregate? i.e inside Handle we will have smth like:

```
var user = userRepository.FindById(userId);
user.ChangeAddress(newAddress, systemClock);
...
```

Did I get you correct? Don't you find this design brittle and cumbersome? Thanks!

Reply

Replies

Zan Kavtaskin 18 April 2017 at 05:27

Hi Dmitri,

Thank you for the question!

Well you have few options:
1. Use service locator inside the method to look up ISystemClock. This is not great as it will mess with your testing.
2. Use static method to get the time information, this StackOverflow Q&A explains it well http://stackoverflow.com/questions/2425721/unit-testing-datetime-now. This is my favorite option. As you can test and you don't have to worry about the DI.
3. Use method overloading, which is what you are referring to. I agree, I don't like this option, I don't mind it if DI resolves it for you, but if you have to manually pass it in it just seems wrong.

Reply

Unknown 24 June 2017 at 14:50

How you would deal if there is a requirement do not introduce changes if data was not really changed? I understand that presentation layer should check that email should be different before submitting to the service. But, if submitting more complex objects where property A and B were changed, but C was not (and C is another depended aggregate root). In this case only A and B should be submitted into repository. So, on domain level (commands, and events handlers) you would check if there are any changes before handing it over to repositories? And raising "changed" events only if data was changed?

Reply

Replies

Zan Kavtaskin 27 June 2017 at 09:18

Hi Denis,

This is a good question. I can't think of an elegant solution right now however, here are some high-level ideas:
1. Use ORM to track changes for you, NHibernate example: http://gunnarpeipman.com/2013/07/implementing-simple-change-tracking-using-nhibernate/, EntityFramework example: http://www.entityframeworktutorial.net/change-tracking-in-entity-framework.aspx.
2. Create a background process that squashes duplicate data.
3. Wrap your entity in a proxy class so that changes are tracked automatically.
4. Manually take a snapshot of the object before the change, when you raise the event provide before and after objects and let handler work it out.

Reply

for ict 99 5 February 2018 at 20:56

great

Reply

Ranjit 13 March 2018 at 19:22

Hi Zan,

Is is possible to use 'Serilog' in Domain Driven Design for Domain Model class using Domain event and domain handler

Reply

Vin 10 January 2019 at 02:00

Hi Mr. Kavtaskin

I'm new to learning DDD and repositories. It looks confusing but thanks to your blog.

Say you have a Repository

as in your code we have:

```
public class Product : IAggregateRoot
{
private List returns = new List();


public virtual Guid Id { get; protected set; }
public virtual string Name { get; protected set; }
```

.....

and then we have static factory methods like

```
public static Product Create(string name, int quantity, decimal cost, ProductCode productCode)
{
return Create(Guid.NewGuid(), name, quantity, cost, productCode);
}
```

.....

I wonder how this will be compatible with let's say Entity Framework because even there is default constructor, the fields are protected set. If I'm implementing something else aside from EF but uses reflection. Do I need an extra layer of DTO to map the domain object from EF or custom implementation? thanks

Reply

Norbert 23 March 2020 at 06:46

Absolutely pragmatic, just the solution to any mid sized enterprise application!
A good compromise between CQRS and DDD in a way every developer is capable of interpreting fast enough to make adequate changes.

Reply

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)