The Power of Events

Ashic Mahtab, SPA2014, Sunday 29th June, 2014 1:00pm to 7:00pm

About Ashic

Software builder – interested in distributed systems, solving problems, and data science.

Fan of Haskell, Erlang, F#, but most clients prefer to pay him to write C#.

Contact: ashic@heartysoft.com

Agenda

- A bit about DDD, CQRS and Event Sourcing
- Beyond notifications – events in modelling
- Events used for testing
- Events used for storage (in place of key-value or SQL databases)
- The Bigger Picture e.g. integration of bounded contexts, scalability etc. – there be dragons

Domain Driven Design

What it is, what it isn't and when to apply it…

There is no real definition of DDD. But there is an eponymous book. Ignore the first 11 chapters of the book. All these are concerned with one specific aspect of bounded concepts. Chapter 11 onwards is still relevant.

Ubiquitous Language – meant to be common to BAs, developers, architects, testers and users.

DDD is a divide-and-conquer approach. The domain is decomposed into subdomains. Each one has its distinct problem-space. Some subdomains are called "generic" in that they can be used across multiple spaces (e.g. HR). Within this bounded context, everyone will have the same concepts, which are used to build the solution.

Theoretically, each bounded context is contained within a specific domain, subdomain or generic subdomain. In practice, it's more likely to extend across parts of several.

Simple principle: a database serves only a single bounded context. Schemas etc. belong to the bounded context, and so does the universal language of that context. Other BCs can access it only through the externally exposed API. E.g. in insurance, you have claims, sales, actuarials… "Policy" means a different thing to Sales than it does to Claims. This is why it is futile to try to store policy entities in a single "central" policy table. The solution for Claims has to evolve in one direction that has implications for the database schema and might break the Sales solution.

Events exported beyond a bounded context usually require a context-specific adapter to make them accessible to another bounded context.

The book has advice about both implementation and strategic DDD. Alan Kay's original object orientation model scaled up, essentially, but with stricter naming conventions.

The book says that DDD is not the use of specific technologies. Even the book states that it is only appropriate to solve about 10% of problems. In most cases, the problem is not complex enough. Or perhaps the problem is non-core business – e.g. CRM, Payroll. Just buy the solution from a specialist supplier. Or perhaps you'll learn more by creating a proof of concept prototype in a couple of days.

CQRS

Command-Query-Responsibility Separation

Event Sourcing

Both CQRS and Event Sourcing are notionally orthogonal to DDD. E.g. a commit-log can be an event source.

A common problem

Stereotypical architecture with layers:

- UI
- Controller
- Service
- Domain
- Repository
- Database

The layers are typically bound together with Data Transfer Objects (DTOs). Ashic has seen up to 17 layers of abstraction between database and UI! The top layer had 16 dependencies because the abstraction was visible all the way up the stack.

The domain layer exists to maintain the integrity of the data by applying rules. In typical applications, research has shown that some 80% of transactions are read-only, so don't really need the integrity constraints. Why not optimise by letting the UI access the database directly for reads? Or provide just a thin abstraction layer.

Implementing separate paths for querying and updating is one form of CQRS. In many applications, this gives you all the optimisation you might need.

A second aspect is reducing the content of each record to the essentials for each model – e.g. querying needs more fields than integrity assurance. Dedicated query models allow you to draw boundaries around them and limit the complexity. You can get advantages by separating the parts of the schema required to support each operation into different tables, even different databases.

Another form of CQRS is to separate a domain from its data storage using a message queue. This is called FORM 2. Very dangerous: if the programmer makes an error allowing one database to diverge from another, e.g. sending some messages more than once or not at all, there is no "source of truth" from which to determine "what really happened". Despite that it is actually the most common form implemented!

Form 3: UI connects to Domain, Domain fires off events. The event store publishes the events to the interested parties. The event store becomes the single source of truth.

Events in Modelling

Tiny steps – in today's exercise, just implement a tiny bounded context.

Event storming

Modelling a solution using verbs and nouns can lead to awkward and rigid class hierarchies, double- and even quadruple-dispatching. Systems become loaded with "valuable names" such as Person or Repository, which appear to have higher levels of abstraction than RavenDBRepository, for example – though actually they are the same, it's just that the SQLServer implementation came first.

Requirements from the business are always changing, which leads to friction.

Event storming is an approach to resolve both problems. Get all stakeholders in a room with an "infinite" modelling space. For every use case, take a sticky note and place it on the wall. These are effectively our commands in CQRS. Things that happen in reaction to those incoming commands are events – more sticky notes (perhaps in a contrasting colour). Trace the flow of events as far as it goes through the system.

Next step is to draw boundaries by examining the flows of information.

Then the next step is to add payloads to the data flows.

Then look at the commands and events within a given bounded context and identify the payload information shared across multiple commands – this becomes your aggregate boundary. We'll talk about aggregates later.

Modelling Exercise

Commercial book lending library. It emerges that there are (at least) five domains: inventory, customer relations, store front, promotions, billing…

Events generated in one can become commands to another domain. To forward these commands to the relevant domain(s), use either

- Process manager
- Routing slip

BizTalk is actually very good – but the facility to execute arbitrary code from within a node is heavily abused and makes BizTalk solutions inflexible. Separate routing logic from everything else.

Events in Testing

Formulating a test case often involves specifying the preceding series of events (the "given"), a command or incoming event (the "when") and a series of events raised (the "then").

(Andy Longshaw): Pre- and post-conditions are events, while the "when" is a command.

This series of events and commands can be represented by pure data. They can be read in and run to execute the test – but they can also be rendered to form documentation.

See example in https://gist.github.com/ashic/f622d06d5b1a0fd46e6a

The events and commands are represented as objects in the test. So for example we have AccountOverdrawn.

Note that it's important not to pollute the Account object with events or the event handling with Account objects. The AccountHandler class decouples the two.

It is vitally important that the public interface of the Account class cannot modify the state of the object. Instead, the state is updated in private UpdateFrom() methods. These are invoked by the aggregate itself raising events, e.g. Apply().

The other classes are immutable value objects that represent events and commands.

Test frameworks / libraries used:

- dokimi
  - Testing DSL
  - Code generator – creates SpecSuite and SpecSuiteExtractor
  - You write framework class (a test class) based on nUnitSpecificationTest
- Infura
  - Conventions – e.g. UpdateFrom (reflection is cached for efficiency)
- Res
  - A simple, efficient event store that can be run on "traditional" storage (for when it really must be sql...)

You can override the Wireup section of each test spec in a subclass to run the same test case in a production environment, for example. The default uses an in-memory data store.

Experiences with message-based testing

Business people tend to come to an implementation level when discussing requirements – by abstracting away from databases, object models etc. it keeps the discussion at the right level of abstraction.

You can describe scenarios without running them. The test specifications are rendered as a Word document. You need to invoke dokimi.exe --help to see the options. It's in dokimi/debug/bin.

What tends to get missed is entire use cases, not erroneous or missing steps in the process.

Event Sourcing

Aggregate

In the model where a domain generates events to an event store, which then distributes them to various other domains, including a UI which can send commands back to the domain to update its state, the domain state is determined by the preceding sequence of events.

In general, an "aggregate" is one in which certain internal objects must be transactionally consistent at all times. For example, an account object must always safeguard that the balance is non-negative: so just the ID and balance are part of the aggregate. For e-mail sending, you need first name, last name and address to be grouped with ID into an aggregate. An aggregate is always persisted as a single operation.

When transferring money from one account to another, the two account balances need to be transactionally consistent – so create another TransferBalance aggregate. The penalty for using a simple transaction manager is that you then have to wrap every other access to the account object

in a transaction as well. Also, the rollback needed if the composite transaction fails loses a lot of business context.

The aggregate is responsible for safeguarding integrity within its boundary – ideally no external object should even know what is inside that boundary.

Principle

Events generated from an aggregate are the only things used to "hydrate" the aggregate on the next command. Public getters and setters are strongly discouraged.

The AccountHandler receives a command such as DebitAccount. To determine whether sufficient funds are in the account, the account handler creates a new (uninitialized) Account object and plays into it the events previously generated from the original Account. E.g. if previously initialised with an initial balance of £300, followed by a debit of £100, a new debit of £250 would be rejected by the business rules in the public API of Account.

To keep this efficient, the transaction history can be truncated by means of a snapshot.

The balance could simply be stored in the database as was traditional. But again, we would have no single source of truth to determine what really happened. Also, we don't need to create a database schema for each domain – the event schema is very generic. So we're trading off flexibility for performance (potentially – there are caching tricks that get the performance up again).

Design guideline: if you see an aggregate with an unbounded event stream, it usually indicates the need for decomposition to smaller aggregates. In the case of a bank account, this is a quarterly account, for example. Another example is an overarching concept that supports temporal queries encapsulating a pure CRUD object.

Hydration of a domain object can be controlled by the object itself.

Example: locking an account after three failed attempts to log in. What if someone decides to reduce the number to 2? The SQL database would need to have columns for number of attempts, current state (locked or not) etc. And some rows could end up in an "inconsistent" state – 2 failed attempts made and yet not locked.

NB this only applies to aggregates! Ordinary CRUD data can be stored/retrieved as normal.

Comment: Cassandra is becoming very popular because it can be very efficient at storing both relational and non-relational data.

Exercise

Event-processing that stores something in SQLServer.

1. Clone from http://github.com/heartysoft/res (you can use "cinst git" to install git in Windows)
2. Open the solution res.SqlStorage
3. Right-click the project res.SqlStorage and choose "publish"
4. Click "Edit"
5. Enter the database name, e.g. ".\sqlexpress" if you have installed SQL Server Express with default options
6. Change database name to "Res"

7. Press OK and then Publish

Create a new solution as a Console Application. This will use the database created before.
The code for the example is largely copied from the earlier exercise project.

Ashic promised a fully elaborated demo on github soon.

Other Frameworks

You can just write hydration and dehydration yourself. Resonant is a good event store – but if you have to use SQL Server then Res is very good. Gregstore (sp?)