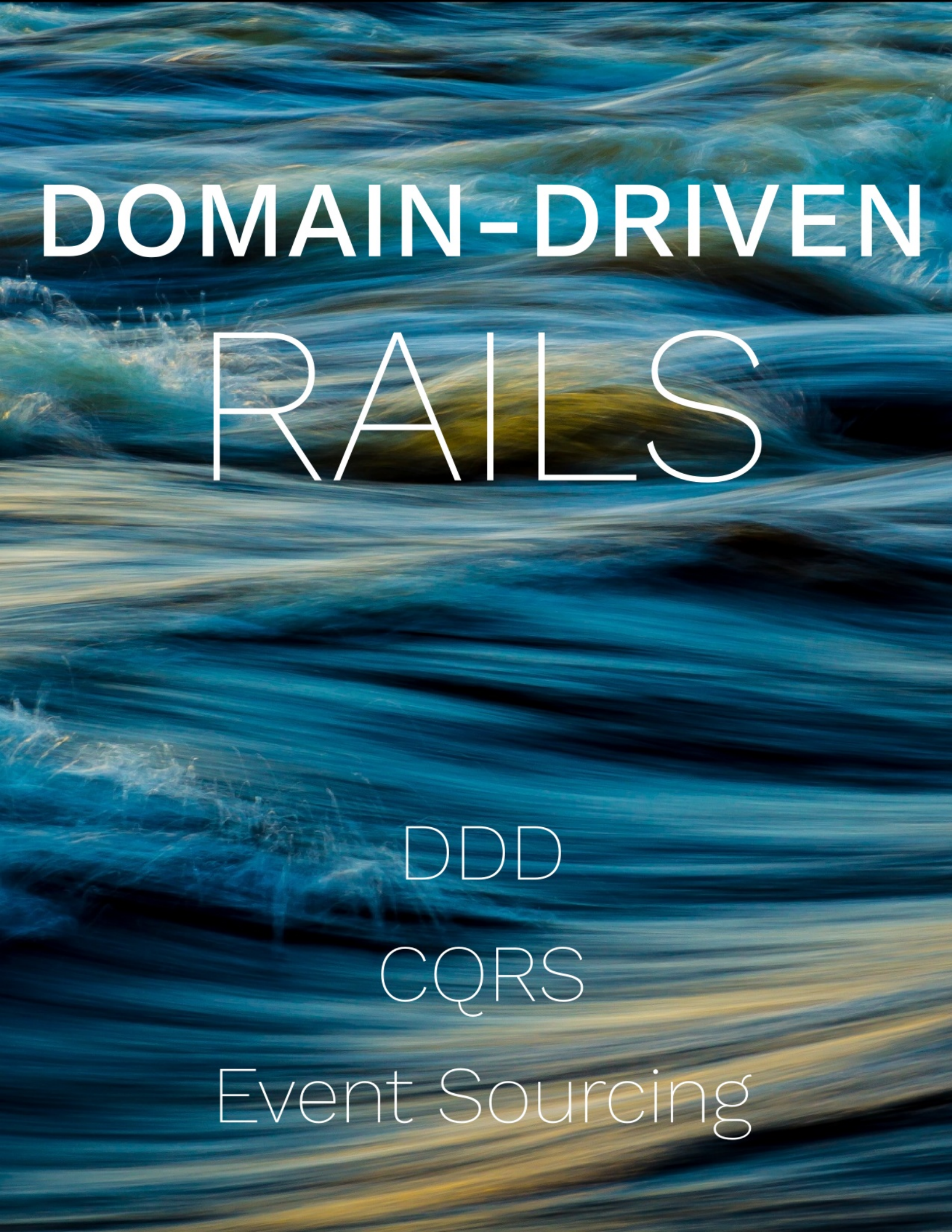

DOMAIN-DRIVEN
RAILS
DDD
CQRS
Event Sourcing

Domain-Driven Rails

DDD, CQRS, Event Sourcing

Robert Pankowecki & Arkency Team

This book is for sale at http://leanpub.com/rails-meets-ddd

This version was published on 2017-10-26

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Prologue

Two dimensions of a Rails developer's growth

Some time ago a developer contacted me through intercom. He wasn't sure about whether a bundle was right for him. I quickly jumped to a phone call (how old-school!) with him and it turned into a very interesting discussion!

One topic that was relevant here is **how to grow as a Rails developer**.

In my opinion, there are two main "dimensions" of growth. Both need to be improved **continuously**. The first direction is the "technology" part - learning languages, learning frameworks, libraries, APIs. This is quite clear, as that's what is expected from us in our daily jobs. We need to write new features to our Rails apps - we need to know Ruby, we need to understand the standard libraries of Ruby. We also need to know Rails, not just the scaffolding, but also all the details of associations, validations, controllers, routes, and view helpers. The more we know, the faster we implement features.

But what makes one programmer N times faster than another is not limited to "feature time". If you know the frameworks, it becomes easier for you to spot problems. Your debugging speed will be improved.

However, this is a double edged sword. **If you focus too much on the technology dimension, you will become too reliant on "The Framework Way"**. You will only approach problems using the idiomatic code for your framework. You may apply some tricks which are only known in this community. Heck, you may even add some "hacks" so that it's in the spirit of that technology.

- Rails Conditional validations anyone?
- Controller filters with :except anyone?
- ActiveRecord + Single Table Inheritance anyone?

There's another direction, often neglected by developers. It's the "generic" skills as a programmer:

- testing
- TDD
- refactoring
- design patterns
- DDD
- clean architecture
- DCI

- Ports & Adapters
- Aspects
- Databases/ACID/CAP
- Programming Paradigms
- Concurrency/Parallelism
- DevOps
- Communication and Soft Skills

Those are the things that make you be able to talk to **programmers from other technologies**.

Can you talk to a PHP/Symfony programmer? Can you talk to a Java/Spring/Akka/Scala/Clojure developer? Can you talk to a .NET programmer? Can you talk to a JavaScript/React.js developer?

The problem with learning in this dimension of your growth is that it seems hard to apply in your current job. It's hard to apply in your current Rails project.

It's nice to talk about DDD, but then you come back to **your 1000-line controller with complex filters algebra**! What a change!

On one hand, this direction of growing helps you long-term. You no longer need to worry what happens if Rails is really dead. You'll use your generic skills to learn other languages and frameworks.

The confidence you get by knowing that you can always switch and it's going to be easy is priceless!

I like working with Rails, I like working with React.js. And I'm not worried too much what happens if they disappear. The programming patterns that I'm using are any-technology-friendly. I can do DDD in any language. I can build more layers in any framework (well, I'm not sure about Angular ;)).

So, it's great to grow as a developer with patterns, architectures and stuff. But how can we apply it in our current projects? How can we use it with Rails?

This book tries to answer how to apply those DDD principles in your current project.

I am going to assume that this is a legacy project. Because you will more likely see benefits from applying DDD in more complicated and business-verified domains.

I am going to assume you are using ActiveRecord because it's the most popular option. You can certainly use DDD techniques with other Ruby persistence libraries (and I encourage you to) but covering all possible situations is out of scope for this book.

It would be good if you are familiar with some of these books:

- Domain-Driven Design by Eric Evans (DDD)[1]

[1] https://www.amazon.com/gp/product/0321125215/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0321125215&linkCode=as2&tag=arkency-20&linkId=0232df31187d4161a608a517d66d7a04

- Implementing Domain-Driven Design by Vaughn Vernon (IDDD)[2]
- Domain-Driven Design Distilled by Vaughn Vernon (DDDD)[3]

But even if you are not, you are still going to understand everything in this book. In such case I hope I will be an inspiration for you to read them later and learn about those concepts from the perspective of one the most influential developers in the world. DDD[4] was one of the 5 most important books for DHH so definitely you will benefit from learning it as well.

It would be impossible to summarize the vast knowledge available in this short book. I don't want to duplicate their statements. Consider reading them after *Domain-Driven Rails*.

There's a saying: "Perfect is the enemy of the good." The solutions shown in this book are in no way going to be *pure* in many cases. This is on purpose. I believe that you can get them by reading IDDD. We don't need to be perfect. Sometimes you need to know the *good enough way* to overcome your fears and start using a technique in your legacy code. Despite all the obstacles. Despite all the differences between Java and Ruby or Hibernate and Active Record.

I am going to try to show *imperfect* solutions for the problems I've encountered, and I hope they'll be relatable to you.

Robert from Arkency

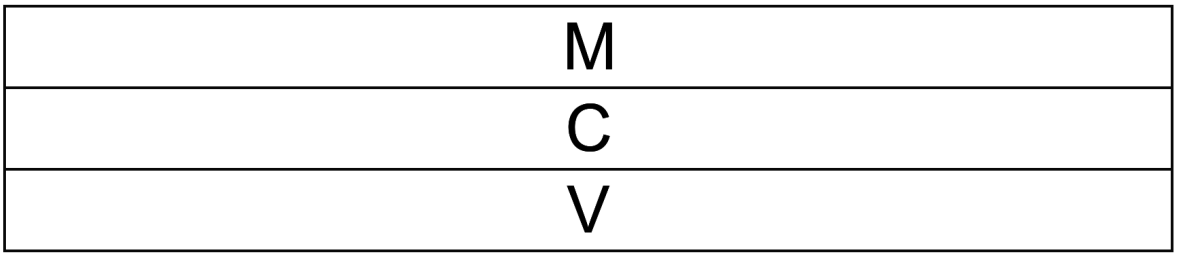
[2] https://www.amazon.com/gp/product/0321834577/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0321834577&linkCode=as2&tag=arkency-20&linkId=3155894f09101a9da242cf5cb6d9bee7

[3] https://www.amazon.com/gp/product/0134434420/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0134434420&linkCode=as2&tag=arkency-20&linkId=12c564c85da17f918d275bdc51626bde

[4] https://www.amazon.com/gp/product/0321125215/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0321125215&linkCode=as2&tag=arkency-20&linkId=0232df31187d4161a608a517d66d7a04

Bounded Contexts

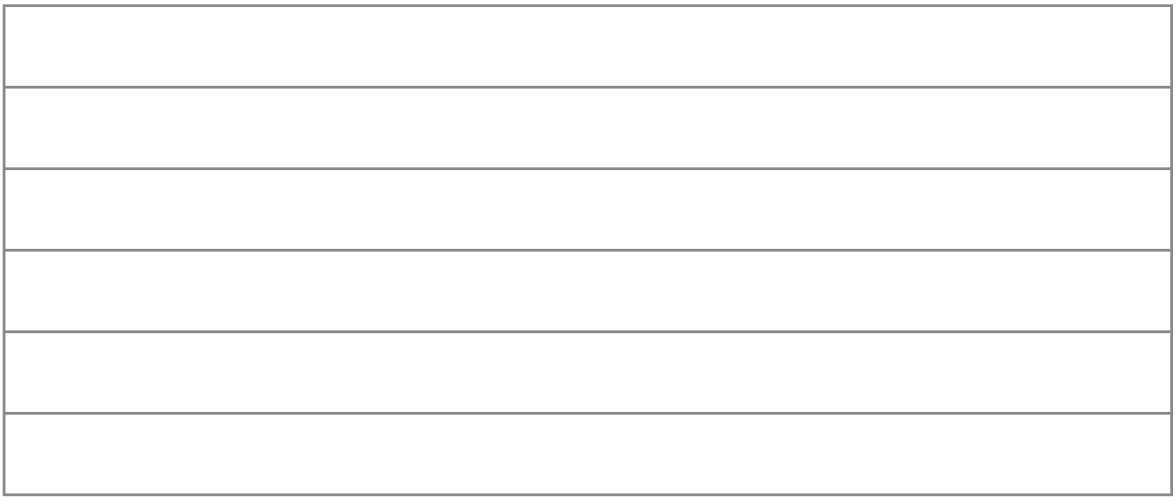
When you started working on your shiny new Rails project it probably looked like this:



Then when it started to get bigger, supporting more and more use cases you might have added new layers:

- Service objects
- Form objects
- Serializers
- Repositories
- and more…

And now your project looks more like this:



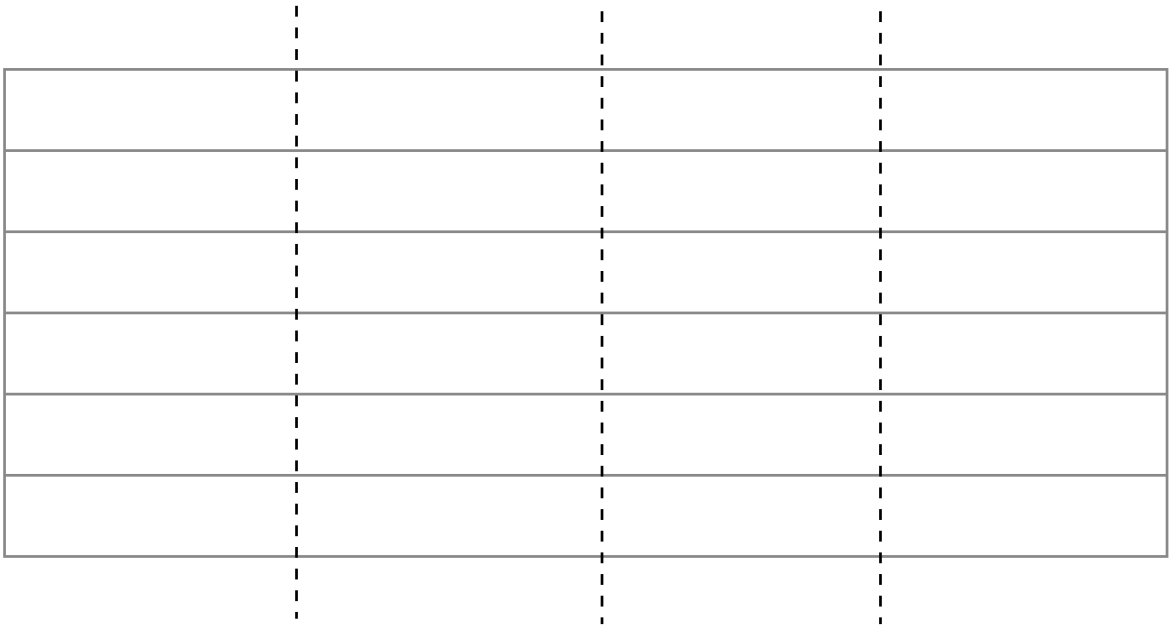
It has more layers. It is wider. It occupies more space. On disk, in memory, and in your head. Does this look familiar? Congratulations! You've got yourself a cake.

Have you ever tried to eat a whole cake at once? I guess not, because that's too much. How do you eat a cake? The same way you eat an elephant: One Bite At A Time!

There are so many books, blog posts, and frameworks out there that will tell you to add more layers to your cake (project). There are very few that will tell you to cut your cake. All those tasty layers in your project won't matter if you never cut your project into more digestible parts.

Have you ever tried to introduce a new developer into a years-old project that already has 10 devs and hundreds of tables, models, and supported use cases? It's … hard. That's because the new developer often needs to understand huge parts of the code before being able to deliver something meaningful. It's like trying to eat a whole cake at once.

How do humans deal with too big problems? We divide and conquer. We break our problems and tasks into smaller chunks. We attack them separately and combine our solutions together.



Organizations consist of divisions, divisions of departments. Rockets consist of multiple components designed to solve problems occurring at specific part of the flight.

Your code should be broken down into units as well. And horizontal layers are not good enough. We need to learn how to split projects vertically, into subsystems.

I am going to call these sub-parts of your system Bounded Contexts.

What is a bounded context?

> The *Bounded Context is a central pattern in Domain-Driven Design. It is the focus of DDD's strategic design section which is all about dealing with large models and teams. DDD deals with large models by dividing them into different Bounded Contexts and being explicit about their interrelationships.*
>
> *As you try to model a larger domain, it gets progressively harder to build a single unified model. Different groups of people will use subtly different vocabularies in different parts of a large organization. The precision of modeling suffers, often leading to a lot of confusion. Typically this confusion focuses on the central concepts of the domain.*

Quoting Martin Fowler from http://martinfowler.com/bliki/BoundedContext.html[5]

[5] http://martinfowler.com/bliki/BoundedContext.html

For more information and deeper understanding of Bounded Contexts I encourage you to read the books mentioned in the Prologue.

What does all that mean for us in practice?

When you see an Active Record class with dozens of attributes you can be almost certain that multiple Bounded Contexts are involved.

Why classes eventually reach 50 columns and hundreds of methods

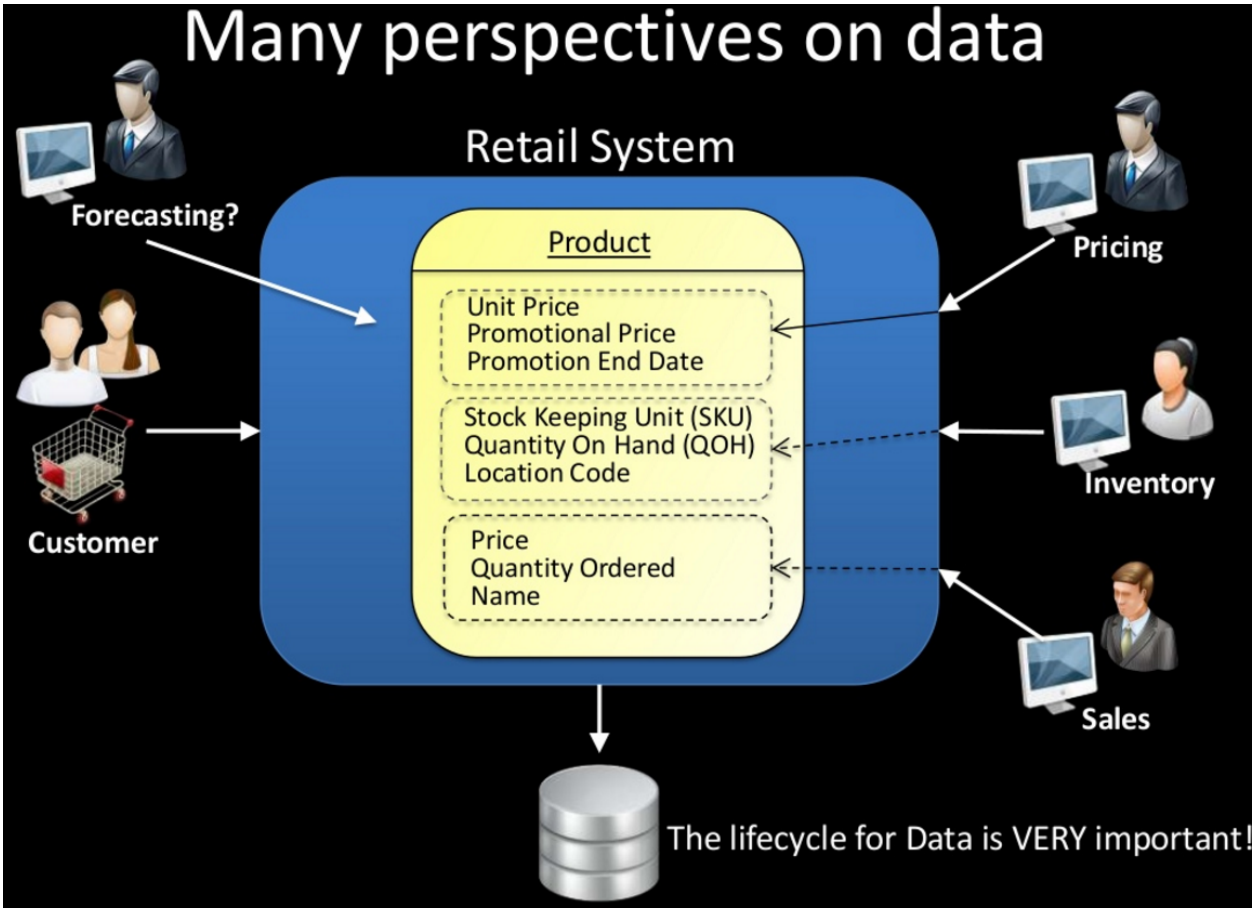
There are dozens of small problems that we can see in our code. Like diseases, they affect our applications and make them more difficult to maintain, expand and enjoy. They give us a headache and they give bugs to our customers. As a result, we (programmers) read a lot to find out more about the symptoms (code smells) and the treatments (refactoring techniques, other languages, other architectures).

One of the most common issues that I see is that classes tend to grow larger and larger. In terms of Rails ActiveRecord models, they grow new columns/attributes and methods around them. I usually see it in `User` classes, but not solely. It really depends on what your system is about. If you work on an e-commerce application it can be the `Shop` or `Product` class. If you work on an HR application, it can be `Employee`. Think about the most central classes in your system and there is a huge chance they keep growing and growing in columns and responsibilities.

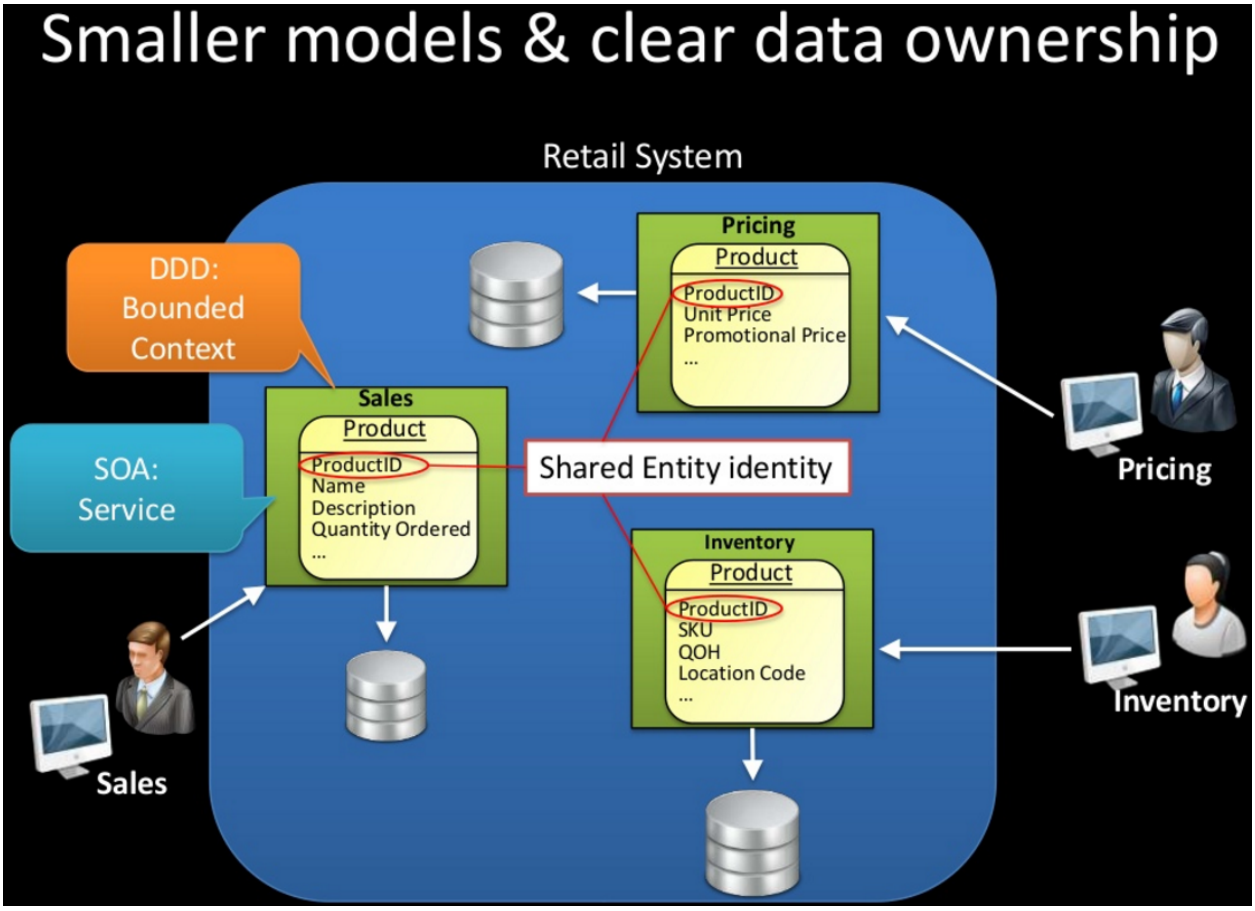
Reading about "Bounded Contexts" - one of the strategic DDD patterns - allowed me to better understand one of the potential causes of the problem. We hear a single word from many people, used in different situations and we identify the word with the same class. After all, that's what we were often taught in school. You know, the noun/verb mapping into classes/methods.

Let me show you 2 slides from one of my favorite presentations[6]. It was an eye-opener for me.

[6] https://www.slideshare.net/jeppec/soa-and-event-driven-architecture-soa-20



So… This is what we often think we have. A class (or entity). Just `Product`. Possibly with dozens of attributes and methods.



This is what we could have. `Sales::Product`, `Pricing::Product`, `Inventory::Product`. Separate classes with separate storage (just separate tables for starters), with their own attributes. There are multiple heuristics you can apply to determine how to split a class. Think which attributes change together in response to certain actions. Check who changes those attributes. For example, Quantities can change because of customer purchases but pricing is only changed by Merchants.

If you are working on discounts or promotions, which are bound to certain conditions, perhaps that in itself is reaching a level of complexity when you realize *I have a Pricing context in my application.*

When you catch yourself working on tracking inventory status; writing business rules about which delivery services provider should be used for which products; how it depends on weight or other product attributes then… Maybe you just started implementing an `Inventory` module and it owns certain information about products, which help make all those computations. So perhaps it is time to have `Inventory::Product`.

I know that understanding and putting boundaries into your application can be hard–especially when you've just started building it. There is a lot of change, a lot of new knowledge coming every day. But over time we (developers) should become more careful, more attentive to what is happening to our system. Sometimes we just add one more attribute, just one more column because we don't see how it fits into the whole system at that time. I get that.

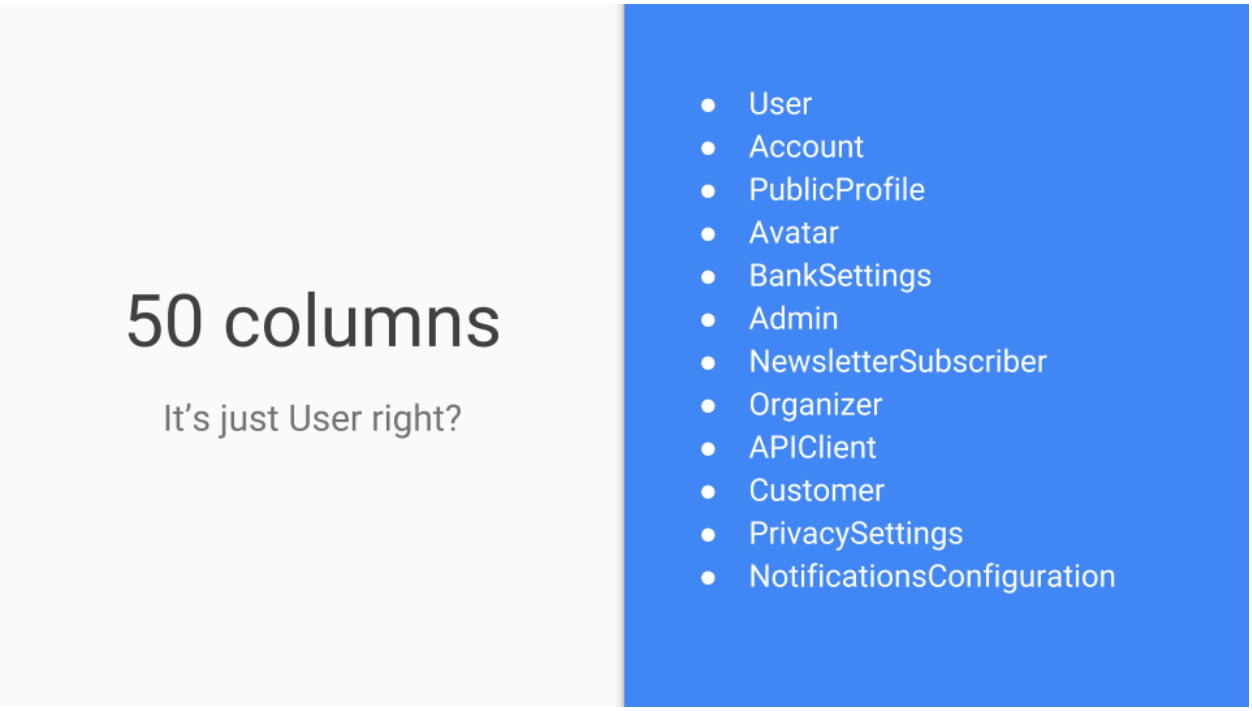
But before a class reached 50 columns, there were 40 occasions to review its design, and its responsibilities; to evaluate whether some of those attributes and methods could form a cohesive, meaningful object which could be extracted into a separate class; to wonder whether our systems is reaching a level of complexity where we should seriously think about its modularization.

This doesn't necessarily mean micro-services. You can start with namespaces in your monolith. You can start splitting those obese classes into smaller ones.

I don't recommend doing it in early stages of the project. It is easy to get things wrong unless the business which you came to already has years of experience from developing previous versions of their software. Instead, just stay vigilant. Try to notice those patterns in your application. Introduce modularization when the business is a bit more mature, but perhaps before you have a 50-columns problem :)

Case study

Go into your current Rails project and check out the schema of the `User` class (or other huge ActiveRecord models you have) in `db/schema.rb` or `db/structure.sql` file. Count the number of columns. Try to imagine which ones are actually needed together and which could be separated into smaller, different objects.



Ask yourself: what do these columns have in common? Why have they landed in the same database table and ActiveRecord model? Is it because they are updated on the same screen, like a page in your app where you go to edit user's data? It might be a huge form with many loosely correlated

inputs smashed together just because they affect the current user. In such cases it suggests that the application design suffers from being coupled directly to the UI. It is very common in CRUD applications and Rails truly excels in this domain. It makes creating a skeleton prototype very straightforward.

At a certain point however, the complexity of your domain increases and a one-to-one mapping between screen UIs and database design (and ActiveRecord models) is no longer good enough.

Think about it. Your mythical *user* does not wake up one day thinking *I am going to change my profile photo and reset my password and change my bank settings*. That does not happen in real life.

Instead, they browse the Internet, find an inspiring image, and decide to change the avatar. They only want to change the avatar.

A few days later they read on Hacker News that another service was hacked, database leaked. Another password drama. Every day on the Internet. So now they want to change their password just to be safe.

Or after a few years they changed banks and now they want to update their bank settings so they continue receiving royalties from the system.

At the beginning there may be one screen with one form and a single save button to update all of that data. Later, when things get more complex and handling the form is no longer convenient (neither for the user, nor for the developer) it can be refactored into separate, smaller, divided forms, with multiple save buttons or even with auto-save. And if that's not enough, you can split it even further and move those forms to separate screens.

So… what kind of questions can we ask ourselves to determine whether we should break a class into smaller (and better named) ones? What should we look for?

- Naming

  Start just by looking at field names. Are they coherent? From the same business domain? Is `encrypted_password` in any way connected to say `last_successful_login_date`. It probably is. But what about `newsletter_id`, `billing_address_street`, `avatar_image_url`, `admin`, `is_profile_public`, well… maybe not so much.
- More naming

  Look for the affixes (prefixes, infixes, suffixes). `address_street`, `address_city`, `bank_name`, `bank_iban`, etc. They can be strong indicators of a class waiting to be extracted.
- Coupling in updates

  Try to figure out which of these attributes are updated together–not accidentally, due to UI coupling as I mentioned before, but truly business-wise. For example, if one form allows you to change the password and avatar then are they really connected? I mean, your application might always be sending both at the same time, but are they really changed together in the real world by users? Do they expect to see/modify both of them together?

On the other hand if part of your business process reserves products - increasing the number of reserved items and decreasing the number of available items - then those two quantity attributes are connected much more strongly. It's no longer an accidental connection. It's a coherent, meaningful pair of attributes that work in harmony.

- Transactional consistency

  Look for which attributes need to be updated together, atomically in the same database transaction. In other words, look for transactional boundaries. This is similar to my previous point. Let's say the user sends you a form with 50 attributes. Do you need to process all of them in one, big, perhaps computationally expensive database transaction? Or would it be OK to split it into a few smaller commands that can be executed and processed separately?

  The problem with splitting, though, is that an error might occur or a crash (server or app) might happen. So we might end up with one command being finished and the other, subsequent commands, go unfinished. Is that a problem? Of course, it depends on what was sent in by the form. We probably don't want to have a half of an order in the DB! Conversely, if the user password is updated but their avatar and public profile are not, then it's no big deal.

  We will learn later that the classes with attributes that cooperate together and need to be updated in one transaction to protect business rules are called Aggregates.

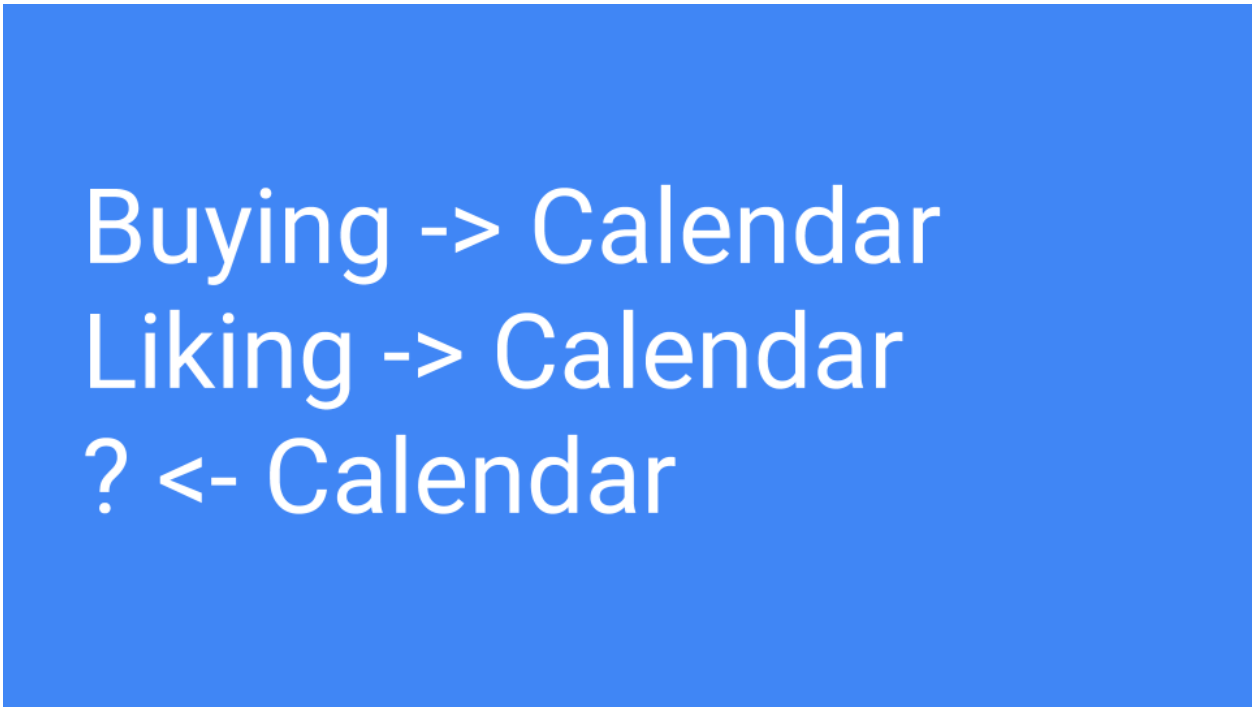
Sometimes an application evolves over time and a class gets more and more attributes added, one by one. Usually it starts because we want to add one more column, just one more little thing, and it does not fit anywhere else. So we keep adding. But later, you realize that now you have groups of attributes which together would make more sense extracted elsewhere. You can see a new class (and later a new module, new context) being born.

So I encourage you to look for hidden classes and names that you currently have in your project - those waiting to be extracted. Once you find them, maybe even many of them, you need to think how best to name them. Once you have many new names, better describing your application than `User`, you can think which of them should be grouped together in larger units (bounded contexts).

Pay attention to dependencies directions

Imagine a ticketing platform where you can buy tickets to events and "like" events. There is a calendar feature which displays events that you liked and events that you go to. Of course this can be easily implemented by querying sales and querying likes. The calendar might not need much of its own code or logic. But things can get more complicated when new features start arriving, like the option to delete an event from a calendar because some people may not be comfortable with everyone seeing what kind of events they go to.

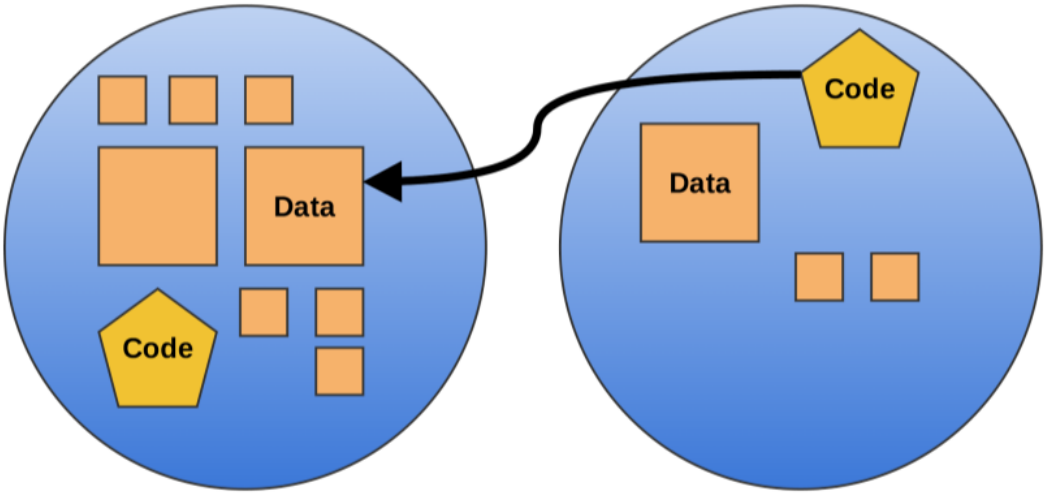
Can I delete something from the calendar without affecting Reporting / Orders / Sales, etc.? If not, then perhaps the calendar IS a separate thing.



The Calendar cares about purchases and likes, nothing in the system cares about the calendar

There are two ways you can solve these problems.

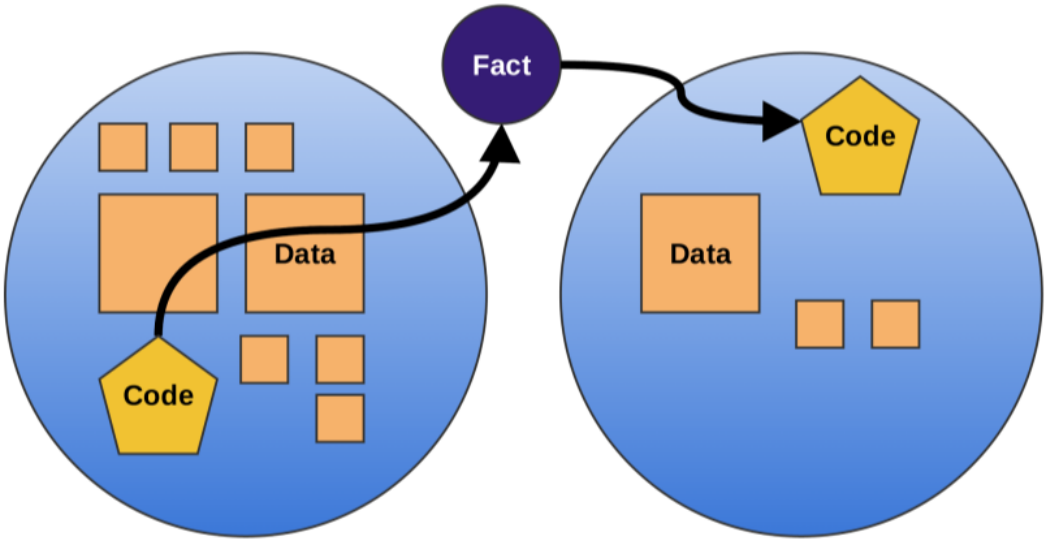
The Calendar can read things from other places - such as Orders.



One context reading data directly from another context by looking via DB

This is what often happens. Associations follow relationships to query data directly from the DB, so one part of the system (one context) gets unlimited access to data which conceptually belongs to another part of the system. We will discuss the problems with this approach in a moment.

Another possible solution is to have some data published from one bounded context, and consumed in another context.



One context publishing data and another consuming it and storing in its own storage

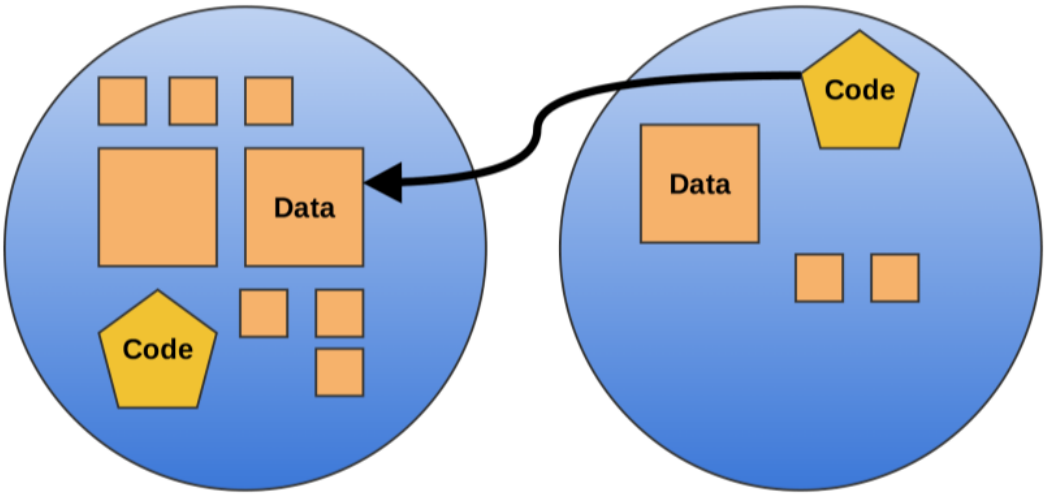
In such cases the consuming bounded context duplicates part of the data and stores it in its own storage (DB tables that it owns). Usually the consuming context needs less data than is published. For example, the Payments bounded context can notify the rest of the system that something was paid with a certain amount, and a particular credit card, using this payment provider, etc. But the consuming context, such as Orders, only cares that it was paid. It does not care how. At the cost of slight duplication of data the consuming bounded context gains independence. We will discuss this approach throughout the book.

Why can’t we easily refactor our Rails apps?

My ongoing experience with Rails is that writing code is easy but refactoring it later is hard. After a few years I have a theory as to why that happens. It may be a bit surprising.

Imagine that you are working on `Order` and `OrderLine`, and similar parts of your codebase. You are trying to do a refactoring or perhaps add a new feature. The problem is that there is a limited amount of information you can keep in your head. The more you need to keep to properly change your code the harder it is to achieve this proper state of mind. If you need less information, minor distractions do not matter much. But the harder the problem is, the more concentration you need to devote to it, the harder it is to start changing anything just because we can’t easily conceptualize all possible paths, all the codebase, all the dependencies. That is just one reason to split our apps into smaller parts (bounded contexts) that can handle most cases internally without contacting the rest of the world too often.

So imagine we have this `Orders` class in our app and we need to change something about it.



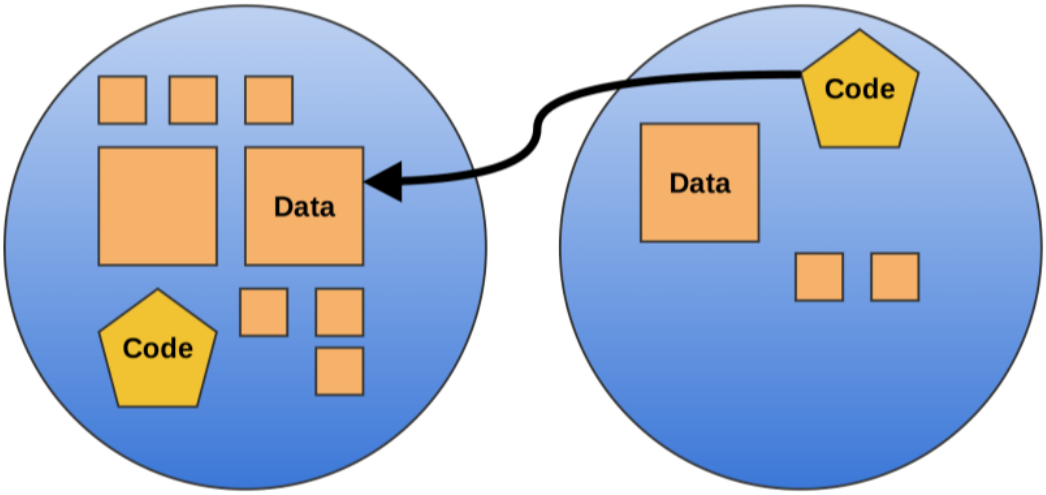
Imagine you need to change something in the left context

You might think that you only need to take care of the code (classes, methods, etc.) and data (tables, rows, storage) related to Orders because that is your focus right now. But the reality of Rails apps is usually more complicated. There is a significant chance that the data is read by other places directly. In other words, there is code somewhere in your app that directly queries the `orders` or `order_lines` tables to display something, to calculate something, to make a decision.

So you would like to change something but there are a number of other use cases that you need to keep in mind. Not only do you need to keep the left part of your app (the left context above) but you also need to keep the right side. Not there are so many things to remember that you become demotivated. The refactorings are just hard because a lot of state is global and not more localized.

A common case for that is reporting. The more reports you have which directly read from multiple tables the harder it is to change your DB schema, change your code and make refactorings or adapt your code for new features.

Because everything is coupled. **And there are few or no boundaries**. From any part of your Rails app, you can traverse to any other part of the app by doing queries inside ActiveRecord models - by following the associations' graph. **We don't have an explicit interface to expose some data** to other parts of applications. Instead, they just peer into our data and **take what they want**. Because the ActiveRecord API is **so wide** and we don't limit ourselves.



How do you expose data from one part to another via an interface?

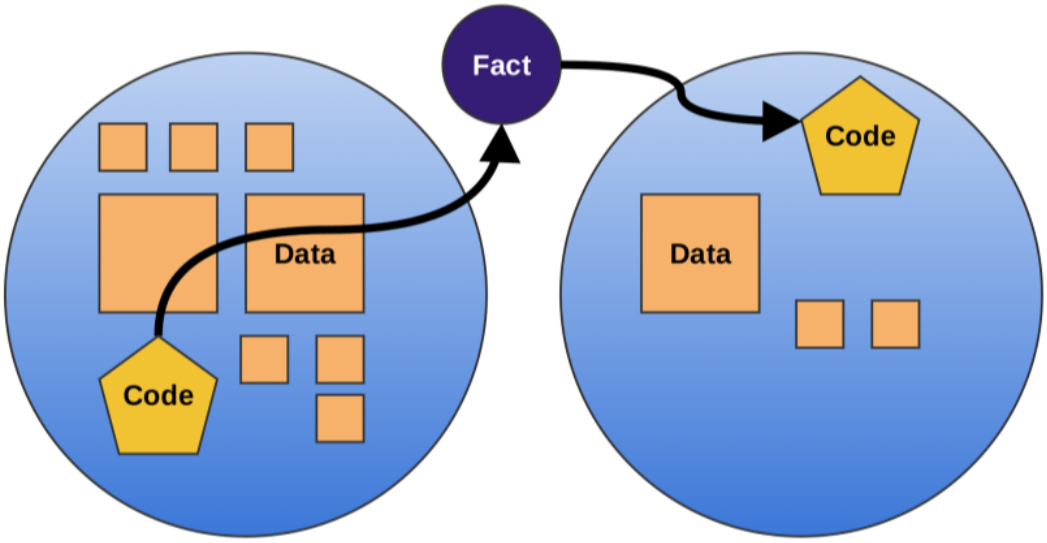
Think about it. Let's say you are changing something in the `OrderLine` class. How would you grep your codebase to find all possible places querying it directly? Relying on some columns or data being there? You need to look for `OrderLine`, `order_line`, `order_lines` and combination of it with `find`, `where`, etc. and you still might not find all places.

It would be a bit easier if we use the repository pattern and at least we know all possible queries that are executed on our tables.

To recap:

- The ActiveRecord API is wide and has tons of methods.
- We don't easily know which other parts of the app are directly reading the data we operate on and would like to change.
  - Or we know, but there are so many places doing it that it does not help
- We don't have an interface to expose that data to other parts of the application

So what's the alternative? What can be the interface connecting those two parts (two contexts) that we generally want to keep separated? I already mentioned one solution that we are going to study in this book: *published domain events* aka *facts*.



Publishing and consuming domain events

How does it work on a high level?

- one bounded context **publishes a domain event (a piece of data) when something interesting happens**
- the other part of application **consumes and stores this info in its OWN data** (its own DB tables in the simplest solution)
- typically the consuming context does not need as much data as the publishing context

For example you can have a `Payments` bounded context with multiple payment gateways from various countries, priorities between them, policies as to when they can be used (some gateways only when the amount is higher than X, other gateways only when the amount is lower than Y), surcharging rules (who pays and how much for the cost of the transactions), and the listening bounded context just cares if it was paid.

Associations

I mentioned that one of the problems which makes refactoring difficult is a lack of boundaries and unlimited traversing of associations between ActiveRecord objects. Let's have a look at a small example (somewhat contrived) which illustrates that kind of problem.

```ruby
class Order
  belongs_to :user
  has_many :order_lines

  def total
    order_lines.map(&:total).sum
  end
end

class OrderLine
  belongs_to :product

  def total
    product.price * quantity
  end
end
```

```ruby
class User
  has_many :orders

  def lifetime_value
    orders.map(&:total).sum
  end
end
```

Domain problems aside (product price is not immutable) this code is not too bad because it uses API methods defined and exposed by `Order` and `OrderLine`. But in practice it turns out to be too slow ;) So let's refactor.

```ruby
class User
  has_many :orders

  def lifetime_value
    orders.joins(
      order_lines: :product)
    ).sum("products.price * order_lines.quantity")
  end
end
```

Technically speaking the values returned will be the same. But from a **maintainability point of view** how do you feel about this change? Imagine that there are more places like this in your application.

How can you change/refactor something in `Order` when there are X other places in your code reading the data directly (`orders`, `order_lines` tables, etc.) and computing something based on them?

Imagine that you want to change the code to handle a new requirement that the product price is kept inside the order line so it is immutable and does not change the existing totals when we change a product's price. How can you, while editing the `Order` class, **easily know that there is such code** in the `User#lifetime_value` method that you should change as well? This is especially hard when you have 15 reports in your app going through 10 tables and now you can't easily change anything anywhere.

There is one more thought that I wanted to bring to your attention, that you might have never consciously thought about. **What does it have to do with `User` ... ?**

```
class User
  has_many :orders

  def lifetime_value
    orders.joins(
      order_lines: :product)
    ).sum("products.price * order_lines.quantity")
  end
end
```

It does not even take anything from user beside its ID.

The classic Rails-way teaches us to do:

```
class User
  has_many :orders
end
```

but that's just a shortcut for `Order.where(user_id: user.id)`. We added the `lifetime_value` method to `User` because it already had `has_many :orders` and that's what we as Rails developers usually do. But the whole method operates much more on `Order`, `OrderLine`, `Product` than `User` itself. It just needs to know the ID of the user that we want to compute the lifetime value for.

What could be better?

```ruby
class Order
  def self.lifetime_value_of_customer(id)
    # ...
  end
end
```

Because then the code operating on `Order` and `OrderLine` and its data in DB tables would be closer to the definition of the class. If I as a developer need to change some logic in `Order` (like the one with immutable price) there is a far higher chance that I would notice this computation that uses the affected data. So it is more likely I will notice and remember to adjust it as well when incorporating new changes. And conceptually that feature has much more to do with `Orders` than `Users`. It just needs the user's ID. That's it.

Summary of this paragraph

Associations are fantastic when you really operate on a whole graph of objects (more about that in the next chapters). They are useful, very convenient, and one of the reasons I fell in love with Rails. But on the other hand… they can strangle your DB schema and make it inflexible.

At some point you need to decide to draw the line. You need to decide that this model, although it has the identifier of another model, cannot reach it directly by going through an association. Otherwise, you will be able (without even realizing it) to go through your entire database by following the graph of associations.

Associations often make the code look like it does not have any dependencies. It is common for ActiveRecord models to have validations that reference a bunch of associations and call their

validations and so on and so on. For example Order will check OrderLines which will go into Product which will go into the ProductSupply history and other Orders' history and decide that we have enough of that product in the inventory to be reserved. But when looking over the codebase it is not clear where there is an `Inventory` bounded context involved, one that `Ordering` depends on to check the available quantity of a product.

Associations are fantastic for *reading* (especially with Rails preloading/eager loading) a whole graph of objects and presenting it on a page. But I submit they are dangerous for *writing* (changing, editing, updating, etc.) the data. You and your team need to be careful not to cross some boundaries (not to go too far in what you are updating in a batch, one transaction). It's very hard. It requires consistent knowledge and communication across the whole team.

As a half-joke, half-serious attempt, I created the `not_activerecord` gem[7] ;) Because you might be thinking *allora, I will just skip defining those associations when I don't want/need to use them.* But then another developer comes along and just adds them, thinking that you forgot about it. So I wanted a way to express that even though model A has a reference/ID of model B, it should not be able to reach it by going through an association.

```
class User < ActiveRecord::Base
  extend NotActiveRecord
  does_not_have_many :orders
end

class Order < ActiveRecord::Base
  extend NotActiveRecord
  does_not_belong_to :user
end
```

So `Order` could have a `user_id` column, but its business logic should not depend in any way on anything from `order.user`. If it needs something, it should be copied into the `order` itself. It is important to realize that we are talking about write-related logic, about handling updates and changes. We aren't talking about reading data for displaying them on a screen.

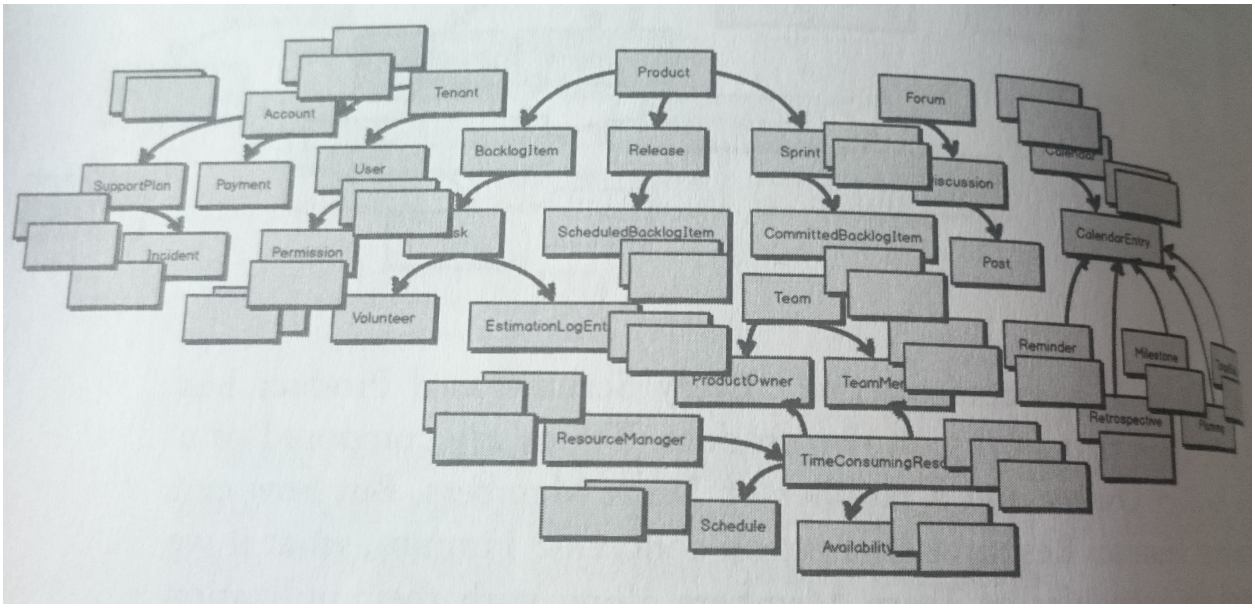
Summary

In DDD Distilled[8] there are two wonderful illustrations which greatly summarize the whole point.

Your application naturally gravitates towards Big Ball of Mud state. In this state there are no boundaries, no borders and everything can be accessed by anything.

[7] https://github.com/paneq/not_activerecord

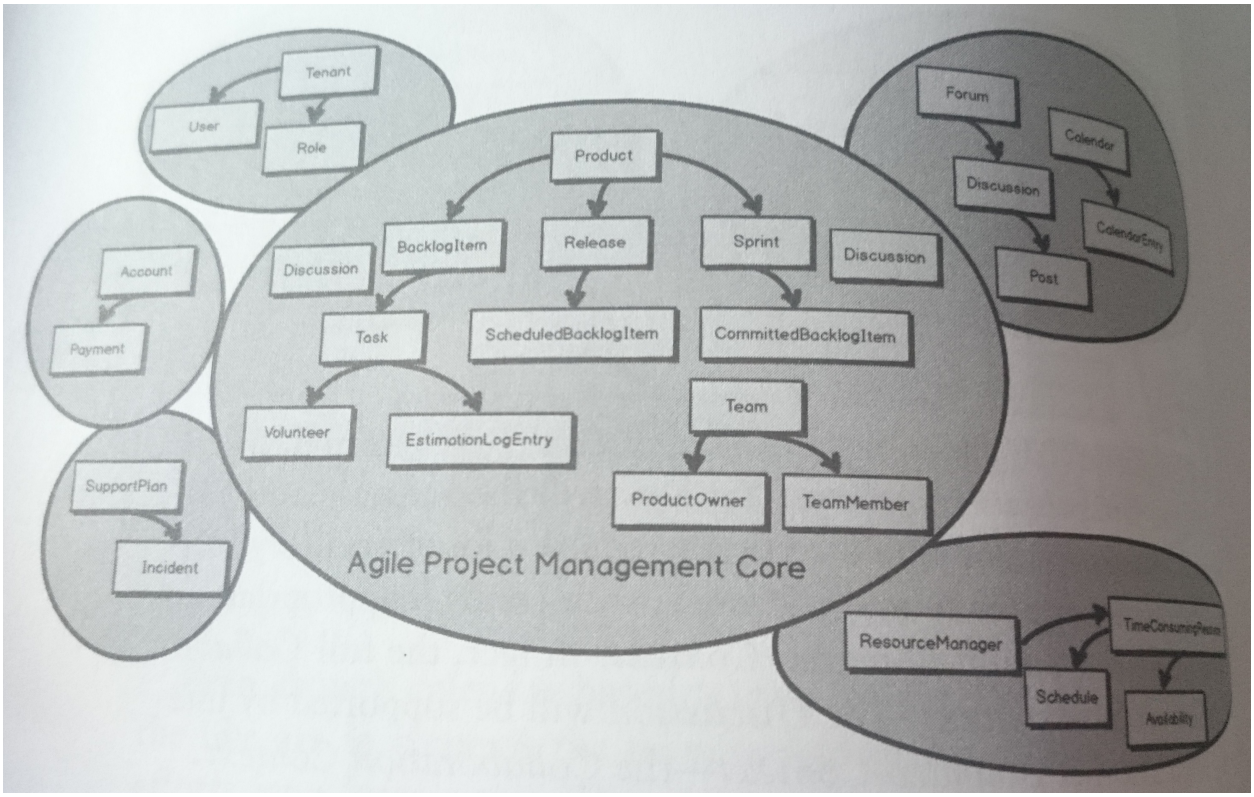
[8] https://www.amazon.com/gp/product/0134434420/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0134434420&linkCode=as2&tag=arkency-20&linkId=12c564c85da17f918d275bdc51626bde



Unbounded model of your app

The bigger our application is, the more it grows, and the more attention we need to give to properly splitting it into smaller, more manageable parts with proper boundaries. Crossing the boundary should be a bit harder than operating within the limits of it. It should give us a moment of reflection: Do I really want to do it? If so then how?

We want limits:



Bounded model of your app

Rails and Ruby give the programmer a lot of tools and flexibility. It is our responsibility to use them wisely. I believe we need to learn how to constrain our design, how to break it into smaller pieces.

Exercise

If you purchased exercises along the book, you can now try the *Bounded Contexts* task.

Questions

If you would like something to be clarified or you have some questions, email us at support@arkency.com . I can't promise an answer but will do my best to help. Maybe even I will write a blog post with an explanation.

Recommended Reading

DDD books

In the classic 3 *DDD* books that I mentioned in the introduction you will find more information about other high-level strategic design techniques such as Context Mapping and Subdomains.

In DDD[9] you can read more about approaches/relationships between contexts such as:

- Shared Kernel
- Customer/Supplier Development Teams
- Conformist
- Anticorruption Layer
- Separate Ways
- Open Host Service
- Published Language

In DDDD[10] they are described quickly as well. It also talks about the pros and cons of some technical approaches to communicate between contexts by:

- RPC with SOAP
- RESTful HTTP
- Messaging

Task based UI

> *One of the largest problems seen in "A Stereotypical Architecture" was that the intent of the user was lost. Because the client interacted by posting data-centric DTOs back and forth with the Application Server, the domain was unable to have any verbs in it. The domain had become a glorified abstraction of the data model. There were no behaviors, or rather, the behaviors existed only in the client, on pieces of paper, or in the heads of the users of the software.*

- https://cqrs.wordpress.com/documents/task-based-ui/

[9] https://www.amazon.com/gp/product/0321125215/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0321125215&linkCode=as2&tag=arkency-20&linkId=0232df31187d4161a608a517d66d7a04

[10] https://www.amazon.com/gp/product/0134434420/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0134434420&linkCode=as2&tag=arkency-20&linkId=12c564c85da17f918d275bdc51626bde

One request can be multiple commands

> *It took me many years to understand the simple truth. One HTTP request sent by a browser may include a few separate logical commands. One request does not always equal one logical command.*

- Bonus section later in the book

Martin Fowler about Bounded Contexts

> *Bounded Contexts have both unrelated concepts (such as a support ticket only existing in a customer support context) but also share concepts (such as products and customers). Different contexts may have completely different models of common concepts with mechanisms to map between these polysemic concepts for integration. Several DDD patterns explore alternative relationships between contexts.*

- Read more[11]

[11] https://martinfowler.com/bliki/BoundedContext.html

Building Blocks

In the previous chapter I talked about Bounded Contexts, a tool for higher-level, strategic design of your application. Now let's discuss the building blocks of our daily coding tasks - tools for tactical design.

Value objects

What are value objects? What does that mean? Where did this name come from?

Value here means that it is describing something.

Object means that it is not a primitive type (not a string, date, number, etc.).

```
require 'money'
Money.new(1000, "USD")
```

Properties

Value Objects describe, quantify, or measure an Entity (such as Customer, Order, Product, etc.). Think of them as *adjectives in your domain.*

```
price = Money.new(1000, "USD")
product.price = price
```

Value Objects have *No identity*, in the sense that, though you can use the word *yellow* to describe almost anything, *yellow* itself has no identity.

Since they have no identity, value objects are compared by the values they encapsulate.

```
Money.new(1000, "USD") == Money.new(1000, "USD")   #=> true
Money.new(1000, "USD") == Money.new( 100, "USD")   #=> false
Money.new(1000, "USD") == Money.new(1000, "EUR")   #=> false
Money.new(1000, "USD") != Money.new(1000, "EUR")   #=> true
```

Value Objects by design are *immutable*, i.e., if something changes, you need to create a new object rather than altering the existing one.

```
price = Money.new(1000, "USD")
product.price = price

# later

product.price = product.price + Money.new(120, "USD")
```

You should not be able to do

```
money.cents = 200_00
```

Instead, you must instantiate a new object.

```
Money.new(200_00, "USD")
```

As the currency example shows, value objects often represent compound attributes grouped together into one class.

```
currency = Money.new(1000, "USD").currency
currency.iso_code #=> "USD"
currency.name     #=> "United States Dollar"
```

```
Point.from_x_y_z(10.0, 20.0, 5.0)
```

```
Location.new(long: 51.1, lat: 17.03)
```

The benefit of using a Value Object is that, instead of passing around all those attributes separately from method to method or from object to object, you can pass around one object instead.

```
cloned_product.price = original_product.price

#instead of

cloned_product.fractional    = original_product.fractional
cloned_product.currency_code = original_product.currency_code
```

Value Objects can protect some simple business rules.

```ruby
Money.new(20_00, "CTY")
# exception => Money::Currency::UnknownCurrency: Unknown currency 'cty'
```

...but I mean really simple, like non-zero, non-empty values–situations where you know that such a measurement/value does not even make sense at all (in your application domain).

Example

2 Euro

50 Polish Zloty

`SchoolYear` Example

Let's have a look at a simple class representing a school year. The academic year usually begins in late summer or early autumn and ends during the following spring or summer, so it crosses two calendar years. In Poland it is usually noted as school year "2017/2018" for example.

So here is an example API:

```
1  school_year  = SchoolYear.current
2  sy_2013_2014 = SchoolYear.from_name("2013/2014")
3  sy_2014_2015 = SchoolYear.new(2014, 2015)
4  sy_2015_2016 = SchoolYear.from_years("2015", "2016")
5  sy_2016_2017 = SchoolYear.from_date(Date.new(2016, 11, 1))
```

A simple implementation.

```ruby
class SchoolYear < Struct.new(:start_year, :end_year)

  def initialize(start_year, end_year)
    raise ArgumentError unless Fixnum === start_year
    raise ArgumentError unless Fixnum === end_year
    raise ArgumentError unless start_year >= 0
    raise ArgumentError unless start_year+1 == end_year
    super(start_year, end_year)
  end

  def self.from_name(name)
    start_year, end_year = *name.split("/")
    from_years(start_year, end_year)
  end

  def self.from_years(start_year_string, end_year_string)
    new( Integer(start_year_string), Integer(end_year_string) )
  end

  def self.from_date(date)
    year = date.year
    if date < Date.new(year, 8, 1)
      new(year-1, year)
    else
      new(year, year+1)
    end
  end

  def self.current
    from_date(Date.current)
  end

  def name
    [start_year, end_year].join("/")
  end

  def next
    self.class.new(start_year.next, end_year.next)
  end

  def prev
    self.class.new(start_year.prev, end_year.prev)
```

```
43    end
44
45    def starts_at
46      Date.new(start_year, 8, 1)
47    end
48
49    def ends_at
50      Date.new(end_year, 7, 31)
51    end
52
53    private :start_year=, :end_year=
54  end
```

As I mentioned, Value Objects can enforce simple rules/constraints. Here, we make sure that we cannot create an incorrect `SchoolYear` such as `2012/2014`.

Be aware that the Value Objects (and the general model of your application) do not need to be universally correct. They must be OK only in the domain you are implementing. So, if you need to support many countries in which school years start and end on different days, then you would not put such logic (responsibility) in this class. However, if you only need to support one country, and the logic makes sense, it can stay inside the Value Object.

So now that you have a `SchoolYear` you can pass that object and use its helpful methods, instead of constantly passing around two separate attributes `2014` and `2015` for example.

Usage in Ruby

Using a SchoolYear in a pure Ruby class is super simple.

```
1  class Klass
2    attr_accessor :school_year # ;)
3  end
```

```
1  klass = Klass.new
2  klass.school_year = SchoolYear.new(2013, 2014)
3  klass.school_year
4  # => #<struct SchoolYear start_year=2013, end_year=2014>
```

Usage in ActiveRecord

You can serialize a Value Object with either a single column or multiple columns. Let's see both approaches.

One column serialization

Here we use the `name` such as `2015/2016` and persist it as a simple string. The getter and setter are equally simple, serializing and deserializing using a simple String and handling `nil`s. We could have a NullObject such as `EmptySchoolYear` instead, but let's leave it at that.

```
class Klass < ActiveRecord::Base
  def school_year=(sy)
    if sy.nil?
      self['school_year_name']= nil
    else
      self['school_year_name']= sy.name
    end
  end

  def school_year
    name = self['school_year_name']
    return nil if name.nil?
    SchoolYear.from_name(name)
  end

  def self.in_year(sy)
    where(school_year_name: sy.name)
  end
end

Klass.first!.school_year
Klass.new.tap{|k| k.school_year = SchoolYear.new(2015, 2015)}
Klass.in_year(SchoolYear.new(2013, 2014))
```

Multi column serialization

Of course one-column serialization does not always make sense, especially when there are many attributes and we may need to search on one.

```
class Klass < ActiveRecord::Base
  def self.in_year(sy)
    where(
      school_year_start: sy.start_year,
      school_year_end: sy.end_year
    )
  end

  def school_year=(sy)
    if sy.nil?
      self.school_year_start = self.school_year_end = nil
      return
    end
    self.school_year_start = sy.start_year
    self.school_year_end   = sy.end_year
  end

  def school_year
    return nil if school_year_start.nil? || school_year_end.nil?
    SchoolYear.from_years(school_year_start, school_year_end)
  end
end

Klass.first!.school_year
Klass.new.tap{|k| k.school_year = SchoolYear.new(2015, 2015)}
Klass.in_year( SchoolYear.new(2014, 2015) )
```

If you don't like this kind of manual attribute mapping and serialization, you can use `YAML` or `JSON` and built-in ActiveRecord features so that ActiveRecord serializes the Value Objects for you.

Other examples

- `VatRate`

  You might think that a tax rate can be a primitive type such as integer (`23`) or float (`23.0`) to describe that we need to pay 23% of a tax. But imagine a (real) case, in which there is a third situation that you need to distinguish in your app for reporting purposes, in which the computed tax is 0%, but you need to know the reason why it was 0%. You need to recognize the difference between `VatRate.new("23")`, `VatRate.new("0")`, `VatRate.new("NOT_APPLIES")`, `VatRate.new("EXEMPTION")`. It's good to have a ValueObject handling this for you so when you compute a gross price when VAT doesn't apply, it knows to use 0% for computation and `NOT_APPLIES` for the rate name.

- `GrossAmount`

  You can have a class representing the gross amount and the vat rate of a product. It can compute the net price using proper mathematical calculations and rounding algorithms defined by the government.
- `ConfigurationLevel`

  In one of our projects there are many configurations/settings which can be configured on many different levels: Shop level, Merchant level, or Country level. If there is no specific configuration for a shop, it will try on the merchant level, and if there is no configuration there, it will read a default based on the Country level. The logic is complicated because some merchants are big and they negotiate different rules/fees in their contracts on the platform. That means whenever we want to read/find the configuration value we must pass around 3 attributes: `shop_id`, `merchant_id`, `country_id`. Instead of passing around 3 attributes separately we pass them as one object `ConfigurationLevel`.
- `ActiveSupport::TimeZone`

  This is a good example from Rails code. It contains a *name*, *utc offset* and methods which will help you get `ActiveSupport::TimeWithZone` for this zone.

Read more

In DDD book[12] you can find a nice discussion about Value Objects, their properties, and exceptions to immutability.

In IDDD book[13] there is another nice discussion, about Value Objects vs. Entities (which we will discuss in a moment), how they relate to the Whole Value pattern, how they can help you normalize your data (i.e. phone number format, names, addresses, lower vs uppercase), and why it all should have Side-Effect-Free Behavior.

Links

- Martin Fowler on Value Objects[14]
- c2 wiki on Value Objects[15]
- Using ruby Range with custom classes[16]

[12] https://www.amazon.com/gp/product/0321125215/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0321125215&linkCode=as2&tag=arkency-20&linkId=0232df31187d4161a608a517d66d7a04

[13] https://www.amazon.com/gp/product/0321834577/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0321834577&linkCode=as2&tag=arkency-20&linkId=3155894f09101a9da242cf5cb6d9bee7

[14] https://martinfowler.com/bliki/ValueObject.html

[15] http://wiki.c2.com/?ValueObject

[16] http://blog.arkency.com/2014/08/using-ruby-range-with-custom-classes/

Entity

I am not going to go into great detail on entities. From my point of view they are what you would usually implement with `ActiveRecord::Base` :)

An Entity is a domain object with a unique identity. Usually their state is mutable–I say usually because I can imagine working with a domain which has certain level of immutability baked in (i.e., *Accounting*). In such cases the changes are normally made by creating a new *Invoice* or *Credit Note* or other *Correction* document.

Examples of what DDD calls entities could be:

- User
- Customer
- Organizer
- Product
- Invoice
- Order
- OrderLine

They have attributes or value objects describing them. They have IDs which make them distinguishable. And most importantly they evolve over their lifetimes: there is a life cycle of changes.

Example

When discussing *Value Objects* I presented 50 Polish Zloty and 2 Euro as examples.

50 Polish Zloty - a banknote

And usually when you implement a Point of Sale (PoS) application or e-commerce application that's a good choice. After all, you only care that you received or paid out a certain amount of money.

But imagine you are working on an application for a national bank or Bureau of Engraving and Printing[17]. The responsibility of such an institution is to print new banknotes, collect old ones, clean or recycle them, track stolen money, etc.

In such cases you would be very interested in a banknote's serial number. It would probably be described by various attributes and maybe include a unique history of what happened to it. It could have been printed, burned, washed, refurbished, stolen, collected again, etc. It has an identity (serial number), it has a life cycle (it changes over time) and it has attributes/value objects describing it (i.e. which machine printed it).

I mention this to point out that there is no single, unique, ideal model for describing the whole world which will work for all applications. Every app has its needs, its requirements, goals and a certain granularity of data it processes to function. Over-complicated models do not serve anyone, so it is your pick, as a developer, to choose whether you model something as a primitive value, a value object, an entity or an aggregate (to be discussed in a moment).

[17] https://en.wikipedia.org/wiki/Bureau_of_Engraving_and_Printing

Questions

If you would like something to be clarified or you have some questions, email us at support@arkency.com . I can't promise an answer but will do my best to help. Maybe even I will write a blog-post with an explanation.

Read more

Implementing Domain-Driven Design[18] presents a nice variety of strategies for generating entities' identities and validating domain rules inside them.

Domain-Driven Design[19] has interesting discussion about Entities and some nice examples.

[18] https://www.amazon.com/gp/product/0321834577/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0321834577&linkCode=as2&tag=arkency-20&linkId=3155894f09101a9da242cf5cb6d9bee7

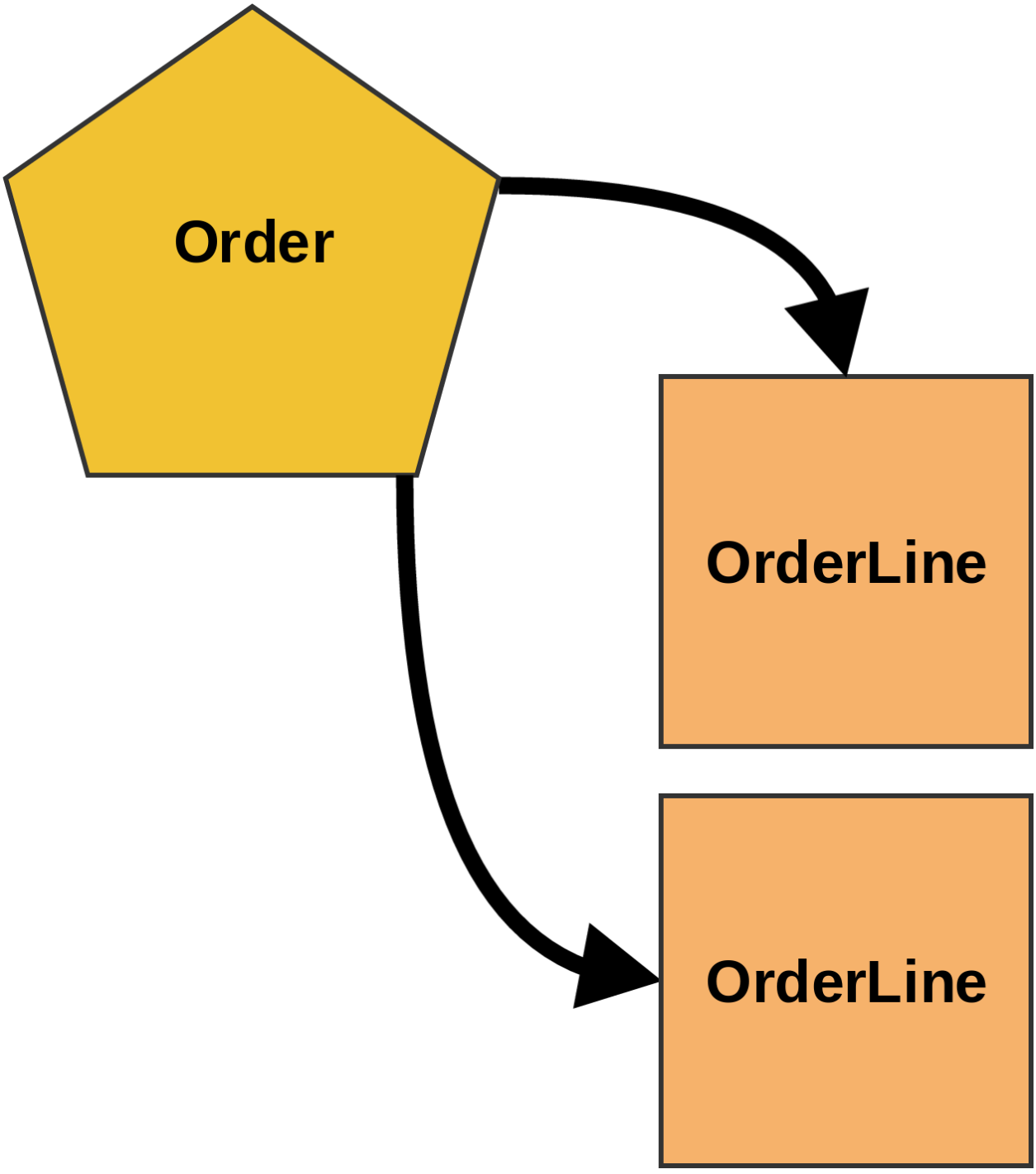
[19] https://www.amazon.com/gp/product/0321125215/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0321125215&linkCode=as2&tag=arkency-20&linkId=0232df31187d4161a608a517d66d7a04

Aggregates

This is the scariest name the DDD community could come up with, but don't let that scare you. It's just a name/tool/technique like any other.

An aggregate is just a graph of domain objects that you completely load from the database, operate on, and save back to the database. It is a set of composed objects defining a single consistent unit. When we work with aggregates we atomically persist the cluster of those objects. It means that when we change them it should either fully work and be persisted or fail and be rolled back.

The root of the objects hierarchy is called an aggregate root. Usually it is used as a conceptual name of an aggregate.



Order as Aggregate

In the case of an order, the Aggregate is a whole cluster of objects. The order has many lines, and the Order is called Aggregate Root, but to describe the whole cluster of objects we would usually just say *Order*.

The role of an aggregate is to protect business rules, to protect invariants. The kind of rules will depend on your app and your domain. For an order it can be making sure the discounts are properly applied/calculated, making sure it is not too expensive, or that taxes are added in the proper way, etc. It is a collection of business requirements that need to be coherent at the end of a transaction.

Sometimes the size of the cluster will be just one object, one entity, but that's okay. Sometimes it will be a bigger graph of objects. The concept applies to even the degenerate case of a single entity.

Why is the technique important? What do you gain?

Rails and ActiveRecord have a huge number of features which help you as a developer just sit down and write an application. However, there are a couple of problems that you will often encounter in legacy apps, downsides of the way Rails apps naturally grow.

One of the problems, which I touched on in the previous chapter, is that you have an ever-growing cluster of objects. By following an association you can traverse the graph of objects many steps down in any direction. This can be problematic when certain actions/operations/commands update unrelated objects distant from each other. As the application continues to grow and changes get to be more complex - doing more and more - there needs to be a larger graph of objects involved in performing certain operations such as adding to a basket or selling. Because the complexity keeps increasing, more objects become involved. To test something you need to create a lot of dependencies and set them up so that you can call a method and put things in motion. Lots of objects coupled together with very few boundaries between them. Instead we would like to have smaller clusters of objects, clusters that cannot so easily cross each other.

Another problem is that ActiveRecord goes way beyond the classic Object-relational mapping. By "classic" I mean allowing you to map from SQL rows to objects, operate on them, and map back. It has tons of methods that let the developer operate directly on the database. It is also a common practice to do queries or partial updates basically anywhere. As a result our legacy Rails apps often suffer from *a brain split*. Some operations, changes, validations, business rules and checks are done in memory, but some are done directly in the DB. In my opinion, that makes both reasoning about the application and testing harder.

Designing your application as a disjunctive set of aggregates can help you normalize the situation. You always (try to) read the cluster of objects from the DB, operate on it, and serialize it back to the DB.

The usual trouble... Not too big, not too small.

Trouble is that it's not always easy to design the boundaries of your aggregates properly, i.e., they are usually too big. If you want have a bit of laugh check out #youraggregatesaresobig hashtag on twitter: https://twitter.com/hashtag/youraggregatesaresobig .

When an aggregate is big, it means that to perform any kind of update you get a sizable tree of objects from the DB, which hurts performance. That itself is not always troubling if the performance is suboptimal but still good enough. But there is another problem on the horizon: contention and deadlocks.

When your aggregate performance is not great and there are many users who would like to perform an operation changing its state, the issue of contention rears its head. Because the aggregate is THE unit of consistency and is needed to protect the business rules inside it, we cannot allow two users to make changes at the same time, since that could lead to a situation in which both changes separately are allowed but together they are not. We need either optimistic or pessimistic locking (both supported by Rails) to detect and prevent such situation.

So if you:

- get an aggregate state from DB and pessimistically lock so nobody else can make changes in the same time;
- perform the necessary operation in-memory on the objects; and
- serialize the state back to DB from the in-memory objects,

and that takes say 0.2s because our aggregate is big, then you can only handle ~5 changes per second: often not good enough, even if you just have 100 customers willing to be buy something in your app :)

If however your aggregate is too small, then it won't be able to enforce some business rules because it simply won't have access to the columns or objects necessary to check it.

So when you think about how big your aggregate should be, how many relationships, how many objects it should include you need to think about:

- what business rules it needs to protect
  - which of them need to be protected within the same database transaction
  - which of them could be protected using eventual consistency and delayed a few seconds or minutes (we will talk more about it in later chapters)
- how often is it going to be changed in the worst case (how many times per second)
- how many separate users will be changing its state at the same time in the worst case

Update one aggregate in one DB transaction and update other aggregates using domain events and eventual consistency

There is a general rule of thumb in the DDD community that in one DB transaction you should only update only one aggregate. But that's just a guideline; you don't need to obey this rule. If you don't

yet feel comfortable with eventual consistency, just commit changes to two or more aggregates in a single, atomic transaction. There is nothing preventing you from doing it; sometimes it is much easier and still safe.

For me, for example, it is rarely so valuable to follow this rule very strictly in the scope of the same bounded context. I would rather just change two objects in the same transaction than go with publishing domain events, writing handlers and reacting to a domain event.

On the other hand there are situations that I've seen in multiple projects, where I would follow this rule almost religiously. This is, for example, when you receive a webhook from a payment gateway about the status of a new payment, usually successful. I would always change the payment status as one transaction and trigger all side effects, such as changing the order and delivery, in a separate transaction. I've seen too many situations where changing payment status and computations, delivery of e-goods, etc. were so coupled in one transaction that failures/exceptions in the business logic caused rollbacks in the transaction, leading to a situation where the business and the app did not know about a successful payment. It's good to decouple/split something relatively simple and irreversible, such as *this payment was paid successfully*, and something very complex, such as *now we need to deliver these files, send these emails, notify dozens of other services, and make sure one last time that no business rule was violated for this order.*

So if you want to change more objects in one transaction just be aware of potential coupling, that it might make things harder to refactor. Make sure to think for a moment whether the transaction will take too long and if it won't have too much contention from many users updating same objects.

The usual case where I don't follow this rule so strictly is when someone updates two objects which can and are only updated by the same user, so there is essentially zero contention.

The usual case where I try to follow the rule is when there is a business rule for max amount of usage, sale, views, conversions, etc., where it is shared between dozens to thousands customers, who are often buying at the same time.

How to conceptually think about them

There is a story that helped me to visualize the difference between doing everything in one transaction vs. splitting into more transactions which operate on smaller objects/aggregates.

Imagine that you develop an app for selling tickets to big events, such as football (soccer) matches, big concerts, conferences, etc. The booking process is quite complicated.

You need to make sure the seat is available in the Venue, that no other person booked or reserved a given seat or rather many seats that you want to buy.

You need to make sure that the ticket type they want to purchase is available. There can be special, limited ticket types such as *Early Bird* at a discounted price, but only for the first 100 buyers or so.

The Order needs to be saved and validated with some additional rules as well.

And let's say we also need to update some *Sales stats, organizer balance* for the conference organizer.

So this is a pretty complicated process with many business rules that need to be guaranteed.

One way to implement it, is by using one database transaction:

- open transaction
- place Order
- update Venue object and check business rules
- change Inventory object and check business rules
- update Balance
- do something more
- commit transaction

And it's OK to do it that way. In many scenarios it will work well. There is a potential problem here, however, when we enter high-throughput sales. Let's say that some parts of this transaction take 0.1 or 0.2s and in total it takes 0.6s to finish the transaction because of those complicated rules and lots of queries and updates to execute. That would be painful, but still acceptable in many cases. However, there are points of contention in this scenario: Venue, Inventory and Balance check rules which are cross-user, rules which need to be guaranteed between many orders/customers such as quantity of tickets available, or available seats.

Because of these contention points, objects that we need to lock, we can only provide 1-2 sales per second. Sometimes that's plenty, sometimes that's not nearly enough. It really depends on the business you are in. When you sell for Championships or Adele's concerts, you need to be good at it.

So what's the other way to address this scenario?

You can have many smaller transactions operating on one object/aggregate. If such transactions takes 0.2s max, then you can handle 5 sales per second, which is better. But at what cost?

In the first solution, when we used one DB transaction it might be easier to handle failures and exceptions. Let's say everything went OK when we checked Venue for available seats, but the cheaper ticket types are no longer available, because they sold out. What do we do? Rollback and forget that anything happened ;) Simple enough.

What about the second solution, with separate transactions? Well, that's more complicated and more work for a developer. You can't just rollback everything. The farther downstream we have a problem in our process, the later it fails, the more steps we need to undo manually. So if there is a problem with the Inventory, we need to publish a domain event, and later release the seats we reserved in the Venue, and also we need to change the Order status to indicate the failure. We can use solutions that we will discuss later in the book, such as Domain Event Handlers, or Sagas to manage the process, but it's definitely more manual work.

So, as usual in programming, there is no silver bullet, there are just techniques with pros and cons. The more you know about those techniques, the more different variants of code you can imagine, the more freedom you have, and the greater your chances of finding the sweet spot.

ActiveRecord-based aggregates

If you think about OOP architecture, and software design in general, the purpose of code of any size is to hide some complexity, some requirements, hide some details and expose a simpler interface for it. It doesn't matter if we talk about a whole application, service, micro-service, module, class, object, function or method. All they do is hide some complexity and expose a simpler API to get things done.

ActiveRecord-Based Aggregates are hard. The purpose of an aggregate is to protect the rules. But almost everything is public in an `ActiveRecord` class: every attribute, every association. The API is also huge. But it is still doable. Let's have a look at an example.

The basket protects the rule of unique products in the basket (without duplicates), i.e., with properly updated quantity.

Basket example

```
1  class Basket < ActiveRecord::Base
2    has_many :items, autosave: true
3
4    # Rules:
5    #
6    # - items with the same SKU are not repeated
7    # but instead their quantity is increased
8    #
9    # - basket total quantity is sum of items' qty
10   def add_item(sku, quantity)
11     changed_item   = items.find{|oi| oi.sku == sku }
12     changed_item ||= items.build(
13       sku: sku,
14       quantity: 0
15     )
16     changed_item.quantity += quantity
17     self.total_quantity += quantity
18   end
19 end
```

What is interesting here?

- We use `has_many :items, autosave: true`. This means that every changed item will be automatically persisted as well if we do `basket.save`. Records newly added to a `has_many` association are automatically persisted even without `autosave: true`. So this makes it

automatically work for just changed records. BTW, for deletion you can go with `mark_for_destruction`[20]

- We use a limited subset of the `ActiveRecord` API which operates on in-memory collections of objects. That's why there is `items.find{|oi| oi.sku == sku }` over `items.where(sku:)`, etc.
- `add_item` does not call `save` or `save!`. The code assumes saving will be called by the object outside, having a reference to the `basket` (most likely a service object/application service).

The code using this class could look like:

```
1  Basket.transaction do
2    # get the whole aggregate (cluster of objects) from DB
3    # using pessimistic locking to guarantee no conflicts
4    basket = Basket.preload(:items).lock.find(1)
5
6    # perform a business operation
7    basket.add_item("BOOK1", 2)
8
9    # persist changes back
10   basket.save!
11 end
```

Problems

As you see, doing Aggregates using ActiveRecord is possible, but it requires communicating across your team that you are writing the code this way. It basically requires self-limitation (like everything in Ruby) to not use every possible API available but restrict yourself. Because there is nothing preventing other developers from changing `items` directly without going through `Basket`.

Other developers in your team can do

```
1  basket.total_quantity = 0 # basket total_quantity is
2                            # no longer sum of item
3                            # quantities
4  basket.save!
```

or

[20] http://api.rubyonrails.org/classes/ActiveRecord/AutosaveAssociation.html#method-i-mark_for_destruction

```
basket_item = Item.find(23) # same problem as before
basket_item.quantity += 1
basket_item.save!
```

or they can use `save!(validate: false)` or `update_column()` or any other method from ActiveRecord which breaks the assumption that the aggregate protects the rules of the cluster of objects.

But I assume you and your team know what you are doing. I assume that you use Ruby every day and even though it is such a powerful language you don't shoot yourself in the foot every day.

So... It's doable but it requires limiting yourself to subsets of the ActiveRecord library in some places.

Event sourced aggregates

We are not going to talk about event sourcing here yet. It will be described in the last part of this book.

Exercise

If you purchased exercises along the book, you can now make the *Aggregates* task. It shows the previous big `Store` aggregate split into 3 smaller ones (`Store`, `Product`, `Shipment`) which communicate asynchronously using domain events. Two business rules are achieved using eventual consistency this time.

Questions

If you would like something to be clarified or you have some questions, email us at support@arkency.com . I can't promise an answer but will do my best to help. Maybe even I will write a blog-post with an explanation.

Read more

Domain-Driven Design Distilled[21] has super useful checklists and rules of thumb for designing Aggregates, for how many objects they should keep inside, how to start small and grow bigger only if needed. Also, this is the best book when it comes to illustrating them :)

[21] https://www.amazon.com/gp/product/0134434420/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0134434420&linkCode=as2&tag=arkency-20&linkId=12c564c85da17f918d275bdc51626bde

In Implementing Domain-Driven Design[22] there is a great chapter about Aggregates. It contains discussions about their size and how to gravitate towards smaller aggregates: how big should they be, which rules actually need to be protected in a single database transaction and which rules can be eventually consistent. It shows why Aggregates should reference other aggregates only by Identity (id) without having direct in-memory access to them.

Domain-Driven Design[23] nicely describes how everything that happens inside an Aggregate should go through its root to make sure all the rules are maintained. It mentions how other parts of the system should only keep references to the root of the Aggregate. Everything below is basically an implementation detail which helps protect the rules. The book includes a list of potential good reasons to break the rule of updating only one aggregate in a database transaction.

Links

- Don't create aggregate roots by Udi Dahan[24] - interesting observation that things in your system do not come out of thin air. Where do they come from then?
- Anemic Domain Model by Martin Fowler[25]

> The basic symptom of an Anemic Domain Model is that at first blush it looks like the real thing. There are objects, many named after the nouns in the domain space, and these objects are connected with the rich relationships and structure that true domain models have. The catch comes when you look at the behavior, and you realize that there is hardly any behavior on these objects, making them little more than bags of getters and setters

[22] https://www.amazon.com/gp/product/0321834577/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0321834577&linkCode=as2&tag=arkency-20&linkId=3155894f09101a9da242cf5cb6d9bee7

[23] https://www.amazon.com/gp/product/0321125215/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0321125215&linkCode=as2&tag=arkency-20&linkId=0232df31187d4161a608a517d66d7a04

[24] http://udidahan.com/2009/06/29/dont-create-aggregate-roots/

[25] https://martinfowler.com/bliki/AnemicDomainModel.html

Services

Application Service

The role of an application service is to provide all the necessary plumbing and infrastructure which makes it possible to execute the code, to *put everything in motion*. So it provides dependencies required by domain model. Application services orchestrate flow & usage of aggregates by loading and saving them to/from repositories.

Code

```
class OrderExpirationService
  # ...

  def call(order_number)
    order_repository.transaction do
      order = order_repository.find(order_number, lock: true) # Load
      order.expire # business operation
      order_repository.save(order) # save back
    end
  end
end
```

The event-sourced version looks pretty similar. I won't discuss it now, but we will come back to this topic in later chapters.

```
class OrderExpirationService
  # ...

  def call(order_number)
    stream = "Order$#{order_number}"
    order = Order.new(number: order_number)
    order.load(stream, event_store: @store)  # load
    order.expire # business operation
    order.store(stream, event_store: @store) # store back
  end
end
```

If you migrate from Rails-way towards more Domain-Driven approach and you don't have repositories yet but you continue using `ActiveRecord` this will look similar to:

```ruby
class OrderExpirationService
  def call(order_number)
    Order.transaction do
      order = Order.lock.find(order_number) # Load
      order.expire # business operation
      order.save! # save back
    end
  end
end
```

One of the simplest rules that you can follow to make your code more testable and easier to refactor in the future is to avoid calling `save!` (or `save`) from your models.

Don't:

```ruby
class Order < ApplicationRecord
  def expire
    # verify preconditions
    self.state = "expired"
    # other logic
    save!
  end
end
```

it is not a responsibility of the domain object to save itself. It makes testing harder, slower and does not allow you to easily compose multiple operations without constantly saving to DB.

So even if you use ActiveRecord, just don't call `save!` from within the class. Only the application service should do it.

Other responsibilities

The repository is often just one of the dependencies that our code need. Others can be

- adapters
- message bus
- event store
- domain services

```ruby
class OrdersAddProductService
  def call(order_number, product_id, quantity)
    prices_adapter = ProductPricesApiAdapter.new
    order_repository.transaction do
      order = order_repository.find(order_number, lock: true)
      order.add_product(
        prices_adapter,
        product_id,
        quantity
      )
      order_repository.save(order)
    end
  end
end
```

Can the application service read more than one object?

I believe it can. But the other object (`Product`) should come from the same bounded context as the object we are updating (`Order`).

```ruby
class OrdersAddProductService
  def call(order_number, product_id, quantity)
    Order.transaction do
      order = Order.lock.find(order_number)
      product = Product.find(product_id)
      order.add_product(product, quantity)
      order.save!
    end
  end
end
```

There are opinions saying that it application services should not be doing it and it should only read and update one object. I am not convinced, however.

There is also a fraction claiming that the 2nd object should not be another aggregate but rather a read-model. I believe it can be an aggregate from the same bounded context.

Can the application service update more than one object?

It is not recommended as it increases coupling between objects (aggregates) that are being updated at the same time. Potential issues to consider:

- the operation takes longer
- the objects have a different lifecycle (one can be updated rarely and by one person, the other can be updated multiple times per second by various users). So the high-throughput nature of one object can cause deadlocks and prevent the whole operation from happening. Or the fact that the operation is longer and both objects remain locked in DB can lower the throughput of the object which is more often edited.
- the objects might be persisted in different DBs so the change is not transactional anyway

What should the application service receive as an argument?

I believe it's best if the service receives a command. The command can be implemented using your preferred solution - pure ruby, dry-rb, virtus, active model, whatever.

```
1  class AddProductToOrderCommand
2    attr_accessor :order_number,
3                  :product_id,
4                  :quantity
5  end
6
7  class OrdersAddProductService
8    def call(command)
9      command.validate!
10     Order.transaction do
11       order = Order.lock.find(command.order_number)
12       product = Product.find(command.product_id)
13       order.add_product(product, command.quantity)
14       order.save!
15     end
16   end
17 end
```

Having commands can be beneficial if you want to easily see what's supposed to be provided. It is more explicit, and it makes it more visible. The more attributes are provided by a form or API the more likely having this layer can be valuable. Also, the more complicated/nested/repeated the attributes are, the more grateful you will be for having commands.

Commands can perform simple validations that don't require any business knowledge. Usually, this is just a trivial thing like making sure a value is present, properly formatted, included in a list of allowed values, etc.

The interesting aspect of defining a closed (not allowing every possible value) structure is that it also increases security and basically acts similar to `strong_parameters`.

Where should the command be instantiated?

I think the controller fits nicely to the responsibility. It has access to all the data from HTTP layer such as currently logged user id (from a session or a token), routing parameters, form post parameters etc. And from all that, it can build a command by passing the necessary values.

Should the command be implemented in the same layer as application service or a higher layer (above it)

Ideally, I believe the layer which contains application service should define the interface of the command and a higher layer could implement it. What's the difference?

Command implemented in a higher layer

If we implemented the command in a higher layer (ie controller) then it could use objects it already has access to such as cookies, session, params etc.

```
class Controller < ApplicationController
  class MyCommand
    def initialize(session, params)
      @session = session
      @params = params
    end

    def user_id
      session.fetch(:user_id)
    end

    def product_id
      params.fetch(:product).fetch(:id)
    end

    def command_name
      "MyCommand"
    end
  end

  def create
    cmd = MyCommand.new(session, params)
    ApplicationService.new.call(cmd)
    head :ok
  end
end
```

Because of Ruby's duck-typing mechanism, this can work but, due to the lack of interfaces, it is not easy for the `ApplicationService` to describe the exact format it expects the data from the command. For simple commands that's not a big issue, but the bigger they get and the more nested attributes they include the harder it is.

One thing I considered as a substitute for interface was... linters :) Such as rack linter[26] or this[27]. In other words shared examples distributed as parts of the lower layer (implementing an Application Service) that could be executed in the tests of the higher layer (controller).

```
RSpec.describe Controller::MyCommand do
  include_examples "ApplicationService::MyCommand"
end
```

in case Test Unit frameworks this would be just a module with test methods to include:

```
class TestMeme < Minitest::Test
  include ApplicationService::MyCommandLint

  def setup
    @command = Controller::MyCommand.new
  end
end
```

But I didn't find this separation worthy and I never went that way. I would in a language with interfaces where defining the structure without implementation is pretty fast.

Command implemented in the same layer as the service

In this version, the controller must copy the necessary data to a provided implementation of the command, which might be less convenient when there are more arguments.

[26] http://www.rubydoc.info/gems/rack/Rack/Lint

[27] https://github.com/RailsEventStore/rails_event_store/blob/master/ruby_event_store/lib/ruby_event_store/spec/event_repository_lint.rb

```ruby
class Controller < ApplicationController
  def create
    cmd = MyCommand.new
    cmd.user_id = session.fetch(:user_id)
    cmd.product_id = params.fetch(:product).fetch(:id)
    ApplicationService.new.call(cmd)
    head :ok
  end
end
```

or

```ruby
class Controller < ApplicationController
  def create
    cmd = MyCommand.new(
      user_id: session.fetch(:user_id)
      product_id: params.fetch(:product).fetch(:id)
    )
    ApplicationService.new.call(cmd)
    head :ok
  end
end
```

Can the application service handle more than 1 operation?

I think it can and I would even say that it's often beneficial to group many operations together in one Application Service instead of scattering them across multiple classes. Because usually, the use-cases need the same dependencies to finish and have a similar workflow.

```ruby
class OrdersService
  def expire(order_number)
    with_order(order_number) do |order|
      # ...
    end
  end

  def add_product(order_number, product_id, quantity)
    with_order(order_number) do |order|
      # ...
    end
  end
```

```
13
14    private
15
16    def with_order(number)
17      order_repository.transaction do
18        order = order_repository.find(number, lock: true)
19        yield order
20        order_repository.save(order)
21      end
22    end
23  end
```

If you go with commands, you can even hide all those internal methods as private.

```
1   class OrdersService
2     def call(command)
3       case command
4       when ExpireOrderCommand
5         expire(command)
6       when AddProductToOrderCommand
7         add_product(command)
8       else
9         raise ArgumentError
10      end
11    end
12
13    private
14
15    def expire(command)
16      with_order(command.order_number) do |order|
17        # ...
18      end
19    end
20
21    def add_product(command)
22      with_order(command.order_number) do |order|
23        # ...
24      end
25    end
26
27    def with_order(number)
28      order_repository.transaction do
```

```
29        order = order_repository.find(number, lock: true)
30        yield order
31        order_repository.save(order)
32      end
33    end
34  end
```

What should the command contain?

- The id/uuid of related entities which should be changed
- The data necessary for performing those changes
- Verified id of the user performing current action
  - `current_user_id` coming from the controller
    - usually based on session, cookies or tokens
- The user id on behalf of whom the action is executed in case that's not the same as the user performing current action
  - Scenario: An Admin executes an operation on behalf of an user who called customer support and asked for help.

What's the difference between Application Service and Command Handler

I believe there is none, really. It's just the names comes from 2 different communities (DDD vs CQRS) but they represent a similar concept. However, Command Handler is more likely to handle just one command :)

What should the Application Service return?

Ideally, nothing. The reason most Application Services return anything is that the higher layer needs the ID of the created record. But if you go with client-side generated (JS fronted or in a controller) UUIDs then the only thing the controller needs to know is whether the operation succeeded. This can be expressed with domain events and/or exceptions.

Should the Application Service contain business logic?

Nope. That should stay in your Aggregates and Domain Services.

Can the application service handle more than 1 operation at the same time?

The important part of this question is **at the same time**.

Yes, although I am not yet sure what's the best approach for it. I consider a separate command vs a commands container for a batch of smaller commands.

Handling many smaller operations can be useful when there are multiple clients with various needs or when you need to handle less granular (compared to standard UI operations) updates provided via CSV/XLS/XLSX/XML files.

As an example: Imagine that you have `ConfirmCommand` and `SetDeliveryMethodCommand` commands that are usually used in your system when actions are performed on the UI. But sometimes you get a batch file to process and for one row in that file you might need to set a delivery method for an order and confirm it as well. What are your options?

A separate command

```
1  class SetDeliveryMethodAndConfirmCommand
2    attr_accessor :order_number,
3                  :delivery_method
4  end
5
6
7  class OrdersService
8    def call(command)
9      case command
10     when SetDeliveryMethodAndConfirmCommand
11       delivery_and_confirm(command)
12     when ConfirmCommand
13       confirm(command)
14     when SetDeliveryMethodCommand
15       set_delivery(command)
16     # ...
17     else
18       raise ArgumentError
19     end
20   end
21
22   private
23
24   def delivery_and_confirm(command)
25     with_order(command.order_number) do |order|
```

```
26        # 2 operations on the same object
27        order.set_delivery_method(command.delivery_method)
28        order.confirm
29      end
30    end
31
32    def confirm(command)
33      # ...
34    end
35
36    def set_delivery(command)
37      # ...
38    end
39
40    def with_order(number)
41      order_repository.transaction do
42        order = order_repository.find(number, lock: true)
43        yield order
44        order_repository.save(order)
45      end
46    end
47  end
```

If the use-case is fairly specific you might implement an additional command. When the application service processes the command, it might perform multiple operations on the domain object (usually an aggregate). Because that aggregate protects its own invariants and business rules we can do it safely and without any concerns.

General commands container

If the use-case is quite general, you can implement an object to group together a bunch of commands.

```
1   class BatchOfCommands
2     attr_reader :commands
3
4     def initialize
5       @commands = []
6     end
7
8     def add_command(cmd)
9       commands << cmd
10      self
```

```
  end
end


class OrdersService
  def call(command)
    case command
    when BatchOfCommands
      batch(command.commands)
    when ConfirmCommand
      confirm(command)
    when SetDeliveryMethodCommand
      set_delivery(command)
    # ...
    else
      raise ArgumentError
    end
  end

  private

  def batch(commands)
    commands.each{|cmd| call(cmd) }
  end

  # ...
end

batch = BatchOfCommands.new
batch.
  add_command(SetDeliveryMethodCommand.new(...)).
  add_command(ConfirmCommand.new(...))

OrdersService.new.call(batch)
```

However, this naive approach can lead to a poor performance (reading and writing the same object multiple times) and it does not guarantee processing all commands transactionally. It will be up to your implementation to balance transactionality and performance and choose the model which best covers your application requirements (including non-functional ones). Generally, you can choose from:

- one transaction per one operation

    - simplest to implement
    - worst performance
    - short locks on objects
  - one transaction per one object
    - balanced performance and lock time
    - guaranteed a whole record processed or skipped (in case of crash in the middle of the process)
  - one transaction per the whole batch
    - very long lock on many objects which might affect many other operations occurring at the same time
    - guaranteed all or none records processed

Here is an example of how the balanced *one transaction per one object* approach can be implemented by grouping commands related to the same object.

```
1  class OrdersService
2    def call(command)
3      case command
4      when BatchOfCommands
5        batch(command.commands)
6      when ConfirmCommand
7        confirm(command)
8      when SetDeliveryMethodCommand
9        set_delivery(command)
10     # ...
11     else
12       raise ArgumentError
13     end
14   end
15
16   private
17
18   def batch(commands)
19     groupped_commands = commands.group_by(&:order_number)
20     groupped_commands.each do |order_number, order_commands|
21       with_order(number) do
22         order_commands.each{|cmd| call(cmd) }
23       end
24     end
25   end
26
27   def confirm(command)
```

```
28      with_order(command.order_number) do |order|
29        order.confirm
30      end
31    end
32
33    def with_order(number)
34      if @order && @order.number == number
35        yield @order
36      elsif @order && @order.number != number
37        raise "not supported"
38      else
39        begin
40          order_repository.transaction do
41            @order = order_repository.find(number, lock: true)
42            yield @order
43            order_repository.save(@order)
44          end
45        ensure
46          @order = nil
47        end
48      end
49    end
50
51  end
52
53  batch = BatchOfCommands.new
54  batch.
55    add_command(SetDeliveryMethodCommand.new(...)).
56    add_command(ConfirmCommand.new(...))
57
58  OrdersService.new.call(batch)
```

No matter which approach, you go with, it can be beneficial when transforming your UI from a bunch of fields sent together into more Task Based UI.

Domain service

Domain services can be strange creatures. They are for keeping the segment of domain logic that does not belong to any aggregate. Usually: calculators, managers, policies, etc.

Here we have an example of a service responsible for calculating various invoice-related dates. It takes into consideration country of purchase, working days, concert date and calculates the last day

an invoice can get paid (due date), when a credit note would be issued (stating the invoice was not paid), when reserved tickets would be released, and generates reminders that you need to pay the invoice.

```
class InvoiceCalendar
  class Terms < Struct.new(
      :invoice_due_at,
      :credit_note,
      :release,
      :optional_reminder1,
      :optional_reminder2
    )
  end

  FREE_DAY = Proc.new do |country_code, date|
    date.saturday? ||
    date.sunday?   ||
    date.holiday?(country_code, :observed)
  end

  def initialize(free_day: FREE_DAY)
    @free_day = free_day
  end

  def call(current_date:, concert_date:, country_code:)
    invoice_due_at = invoice_due_at(country_code, current_date, concert_date)

    credit_note = add_working_days(country_code, invoice_due_at, 1)
    release = add_working_days(country_code, credit_note, 1)

    optional_reminder2 = subtract_working_days(country_code, invoice_due_at, 3)
    optional_reminder2 = nil unless current_date + 2 <= optional_reminder2

    optional_reminder1 = subtract_working_days(country_code, invoice_due_at, 7)
    optional_reminder1 = nil unless current_date + 2 <= optional_reminder1

    Terms.new(
      invoice_due_at,
      credit_note,
      release,
      optional_reminder1,
      optional_reminder2
    )
```

```ruby
  end

  private

  def invoice_due_at(country_code, current_date, concert_date)
    due_date_candidate1 = add_working_days(country_code, current_date + 13, 1)
    due_date_candidate2 = subtract_working_days(country_code, concert_date, 3)
    [due_date_candidate1, due_date_candidate2].min
  end

  def subtract_working_days(country_code, date, days)
    days.times do
      loop do
        date -= 1
        break unless free_day?(country_code, date)
      end
    end
    date
  end

  def add_working_days(country_code, date, days)
    days.times do
      loop do
        date += 1
        break unless free_day?(country_code, date)
      end
    end
    date
  end

  def free_day?(country_code, date)
    @free_day.call(country_code, date)
  end
end
```

All of that is business logic around Invoicing. But it does not make much sense for the `Invoice` class to know about all this logic. Especially if you realize that changing it, having more days here, or less days there does not affect what's on the Invoice, how you pay it, how you recognize whether it was paid.

Notice that this calculator is **stateless and side-effect free**. It just takes 3 values as input `current_date:, concert_date:, country_code:` and returns 5 values as output (although grouped together into one object). That means I can easily test it without creating any invoice, any concert on database,

etc.

`InvoiceCalendar` takes the algorithm for detecting a free day as a dependency, which makes testing the logic easier.

```
RSpec.describe InvoiceCalendar do
  specify "when concert is far from now, due at is 14 days later" do
    calendar = InvoiceCalendar.new(
      free_day: ->(country_code, date){ date.day % 3 > 0}
    )
    result = calendar.call(
      current_date: Date.new(2016, 1,  1),
      concert_date: Date.new(2016, 1, 25),
      country_code: :uk,
    )
    expect(result.invoice_due_at).to eq(Date.new(2016, 1, 15))
    expect(result.credit_note).to    eq(Date.new(2016, 1, 18))
    expect(result.release).to        eq(Date.new(2016, 1, 21))
  end
```

It's nice to look out for business logic in your app that can be tested with an isolation. Many DDD techniques help you separate mutations from side-effect free code. In fact DDD is very functional programming friendly.

Questions

If you would like something to be clarified or you have some questions, email us at support@arkency.com . I can't promise an answer but will do my best to help. Maybe even I will write a blog-post with an explanation.

Exercise

If you purchased exercises along the book, you can now make the *Services* task.

Links

- Services in Domain-Driven Design[28]

[28] http://gorodinski.com/blog/2012/04/14/services-in-domain-driven-design-ddd/

Application services declare dependencies on infrastructural services required to execute domain logic. Command handlers are a flavor of application services which focus on handling a single command typically in a CQRS architecture.

- Command Handlers[29]

  An example of converting an Application Service into Command Handler in C#
- Domain services vs Application services[30]

  Interesting read about the border between *Domain services vs Application services*. It's all about logic and business domain. About whether it's there or not. But how do you spot business logic? Nice examples and lovely explanations.

[29] https://buildplease.com/pages/fpc-10/

[30] http://enterprisecraftsmanship.com/2016/09/08/domain-services-vs-application-services/

Command bus

When I learned about *Service Objects*, and found out how much they can help by separating the HTTP layer from the Application/Domain layer, I went through a phase where I created a new class for every operation, like:

- ApproveOrder
- ExpireOrder
- CancelOrder

etc.

After some time I realized there are a couple of problems with this approach.

- A lot of logic was repeated between those classes (unless you use inheritance or mixins as a workaround).

  This is typically infrastructure code like loading/saving to DB with proper transactions and locking.
- It can easily lead to an anemic domain model.

  It is tempting to put the business rules in services instead of the model. Then when you want to find out what rules you actually protect, you need to check multiple files.
- It feels like the whole solution is not cohesive.

So I started to lean towards services that can handle multiple commands (operations/actions/however you name them).

Commands

Commands are structures that contain all the attributes necessary to perform an operation in your system and describe the intention.

Usually we don't have explicit commands in our Rails apps. Usually they are just a `Hash` or `ActiveSupport::HashWithIndifferentAccess` (available as `params` in controllers).

But having commands can be beneficial if you want to easily see what's supposed to be provided. It is more explicit, and it makes it more visible. The more attributes are provided by a form or API the more likely having this layer can be valuable. Also, the more complicated/nested/repeated the attributes are, the more grateful you will be for having commands.

Commands can perform simple validations that don't require any business knowledge. Usually this is just a trivial thing like making sure a value is present, properly formatted, included in a list of allowed values, etc. You can even use active_model validations for it.

The interesting aspect of defining a closed (not allowing every possible value) structure is that it also increases security and basically acts similar to `strong_parameters`.

command.rb

```ruby
require 'active_model'

module Command
  ValidationError = Class.new(StandardError)

  def self.included(base)
    base.include ActiveModel::Model
    base.include ActiveModel::Validations
    base.include ActiveModel::Conversion
  end

  def validate!
    raise ValidationError, errors unless valid?
  end
end
```

This is the simplest point you can start from.

If you know Ruby well, then you are probably aware that technically this solution should work equally well:

```ruby
module Command
  include ActiveModel::Model
  include ActiveModel::Validations
  include ActiveModel::Conversion
end
```

... but unfortunately it does not, due to some built-in assumptions of ActiveModel. But Rails is constantly evolving and dropping its old habits to follow Ruby philosophy, so hopefully it will be fixed soon.

Defining a command

To define a command you can use the `Command` module presented and/or sprinkle it with your hand-picked choice of gems for type-casting, validations, serializations/deserializations such as `virtus` or `dry-validation` or `dry-types`. We did not release any gems for defining commands, so you can just use the ecosystem of libraries that you know and like. If you are a fan of Rails, ActiveModel, etc. just use it. If you are a dry-rb aficionado, nothing stops you from using it. I can show you some examples.

Manual type-casting with Ruby

```
class ExpireOrderCommand
  include Command

  attr_accessor :order_number
  validates_presence_of :order_number
end
```

You know that `ExpireOrderCommand` takes one attribute and that it needs to be present. But is it a number, or a string or what? It would be nice to know.

```
class ExpireOrderCommand
  include Command

  attr_reader :order_number
  validates_presence_of :order_number


  def order_number=(int)
    @order_number = Integer(int)
  end
end
```

You can use `Integer()` or `to_i` or `to_int` depending on the semantics you prefer or what you'd like to have happen in the event of an error.

Virtus.model

```
class LikeProductsCommand
  include Command
  include Virtus.model

  attribute :product_ids, [Integer]
  attribute :value,       Boolean

  validates :product_ids, presence: true
end
```

It is nice if your library can handle nested collections or objects.

```
  class OrderLine
    include Virtus.model
    attribute :quantity, Integer
    attribute :product_type_id, Integer
    attribute :price, BigDecimal
    attribute :seats, [Seat]
  end

  class DeliveryAddress
    include Virtus.model
    include ActiveModel::Validations

    attribute :first_name,   String
    attribute :last_name,    String
    attribute :street_line,  String
    attribute :street_line2, String
    attribute :zip_code,     String
    attribute :city_name,    String
    attribute :country_code, String

    validates :first_name,
              :last_name,
              :street_line,
              :zip_code,
              :city_name,
              :country_code,
              presence: true
  end

  class Input
```

```
31      include Virtus.model
32      include ActiveModel::Validations
33
34      attribute :payment_method,   String
35      attribute :delivery_address, DeliveryAddress
36      attribute :order_lines,      [OrderLine]
37    end
```

Virtus.value_object

```
class ChangePricingCommand
  include Virtus.value_object
  values do
    attribute :product_id, Integer
    attribute :price,    GrossAmount
    attribute :kickback, BigDecimal, default: 0.0
  end
end
```

dry-rb

I am not super familiar with the dry-rb ecosystem but I am pretty sure that `dry-types`, `dry-struct` and `dry-validations` provide a nice experience in building command objects as well.

```
module Types
  include Dry::Types.module

  ComicPreference = Strict::String.enum("Marvel", "DC")
end

class RegisterReader < Dry::Struct
  attribute :name, Types::Strict::String
  attribute :age, Types::Strict::Int
  attribute :prefernce, Types::ComicPreference
end
```

Instantiating a command in a controller

Most of those libraries will let you easily (or with minor modification) instantiate the command from the `params` in a controller.

```
class ProductLikesController < ApplicationController
  def create
    cmd = LikeProductsCommand.new(params)
    LikesService.new.call(cmd)
    head :no_content
  end
end
```

However, over time I learned that certain kinds of repetition in code are not harmful and don't require more thinking, work or maintenance. Yes, it is more boilerplate but it is grep-able boilerplate that can help in finding usages, being explicit and understanding the flow of data.

It's up to you. With commands you are already one level higher in documenting the code, because you list all the attributes expected.

```
class ConcertsIntegrationsController < ApplicationController
  def enable
    cmd = SetupConcertCommand.new(
      api_key:    params[:api_key],
      concert_id: params[:concert_id]
    )
    cmd.validate!
    service.call(cmd)
    head :no_content
  rescue DuplicatedLabel => e
    ErrorReporting.notify(e)
    render json: {
       errors: [{ status: 422, title: e.message }]
    }, status: :unprocessable_entity
  end
end
```

Also, I don't really care that much if you do command validation (validating simplest things) inside or outside a service. My preference leans toward doing it inside the service.

Now that we have covered defining the commands we can go back to using the command bus.

Command bus inside a service

The first strategy revolves around having an Application Service (_aka Service Object) that helps you handle a certain aggregate and its operations, accepting multiple commands. We could use different method names, but instead the public API consists of only one method: `#call`. We will determine what needs to be done based on the Command class.

```
require 'arkency/command_bus'

class OrdersService
  def initialize(store:)
    @store = store

    @command_bus = Arkency::CommandBus.new
    { SubmitOrderCommand  => method(:submit),
      ExpireOrderCommand  => method(:expire),
      CancelOrderCommand  => method(:cancel),
      ShipOrderCommand    => method(:ship),
    }.map{|klass, handler| @command_bus.register(klass, handler)}
  end

  def call(*commands)
    commands.each do |cmd|
      @command_bus.call(cmd)
    end
  end

  private

  def submit(cmd)
    # ...
  end

  def expire(cmd)
    cmd.validate!
    stream = "Order$#{cmd.order_number}"
    order = Order.new(number: cmd.order_number)
    order.load(stream, event_store: @store)
    order.expire
    order.store(stream, event_store: @store)
  end

  def cancel(cmd)
    # ...
  end

  def ship(cmd)
    # ...
  end
```

```
43 end
```

As you can see, instead of `SubmitOrderService`, `CancelOrderService`, `ExpireOrderService`, etc., we now have just one service `OrdersService` that can handle all commands related to Orders such as `SubmitOrderCommand`, `ExpireOrderCommand`, `CancelOrderCommand` etc.

The consumer of this code does not need to worry what method it should call as there is only `#call` (pun intended) available as a public API. The intention was already expressed by instantiating the object of a proper command class. That's enough to know what needs to be done.

arkency-command_bus handlers

In our gem[31] any object that responds to `call` can be a handler. This turns out to be an extremely flexible solution.

In the case of stateless services you can register just one instance.

```
1 class FooService
2   def call(cmd)
3     Foo.create!(name: cmd.name, boo: "boo")
4   end
5 end
```

```
1 command_bus.register(FooCommand, FooService.new)
```

`FooService` is *stateless* because it does not preserve any instance variables between multiple method calls (between processing many commands).

For safety, if you want to be sure that the service continues properly operating in a stateless way, you can `freeze` it.

```
1 command_bus.register(FooCommand, FooService.new.freeze)
```

This method is available in Ruby on any object. If you try to change an instance variable after #freeze, you get an exception.

[31] https://github.com/arkency/command_bus

```
class Element
  def name=(name)
    @name = name
  end
end

fire = Element.new
fire.name = "fire"

fire.freeze
fire.name = "water"
#=> RuntimeError: can't modify frozen Element
```

Or you can make the service `freeze` itself when it is ready.

```
class FooService
  def initialize(dependency)
    @dependency = dependency # assuming stateless as well
    freeze
  end
end
```

```
command_bus.register(FooCommand, FooService.new)
```

That way, when you edit `FooService` you can see that callers expect it to be stateless and safe for passing multiple commands to the same instance.

When a stateless services does not have a `call` method it is not a problem.

If you don't want to or can't have a `call` method, it is not a problem. Because in ruby you can easily obtain a reference to an object's method.

```
class BarService
  def bark(cmd)
    # ...
  end
end
```

```
command_bus.register(BarCommand, BarService.new.method(:bark))
```

The same `freeze` advice applies as before :)

```
command_bus.register(BarCommand, BarService.new.freeze.method(:bark))
```

In the case of a stateful service we need a new instance for every command.

A service is *statefull* when it preserves any instance variable between multiple method calls (between processing many commands). In such case we need a new instance for every command to avoid potential issues when processing command B could use an instance variable memoized by a previously processed command A.

```
class BazService
  def initialize(dependency)
    @dependency = dependency
  end

  def bazinga(cmd)
    @dependency.surprise += 1 # stateful
    Baz.create!(attributes(cmd))
    @dependency.boom(attributes(cmd))
  end

  private

  # stateful
  def attributes(cmd)
    @attributes ||= {
      wow: cmd.wow,
      pow: cmd.powpow
    }
  end
end
```

We simply wrap creating and calling a new instance in a proc.

```
command_bus.register(
  BazCommand,
  -> (baz_cmd) { BazService.new(dependency).bazinga(baz_cmd) }
)
```

It doesn't matter if the service method is named `call` or not, as we already invoke it manually in the proc.

This gives use the necessary safety net we need. Every new command will be processed by a new, clean instance of a `BazService`.

Command bus outside a service

You can also use a command bus for **delivering commands to various, different services**. This can be useful if you don't want your controllers to know exactly which service they are calling. It will be described later as a technique for **simplifying your sagas' (process managers) implementation**.

```
require 'arkency/command_bus'

config.to_prepare do
  config.command_bus = bus = Arkency::CommandBus.new

  bus.register(FooCommand, FooService.new)
  bus.register(BarCommand, BarService.new.method(:bark))
  bus.register(BazCommand,
    -> (baz_cmd) { BazService.new(dependency).bazinga(baz_cmd) }
  )
end
```

In Rails development mode, when you change a registered class, it is reloaded, and a new class with same name is constructed. To workaround this problem we use `to_prepare`[32] callback which is executed before every code reload in development, and once in production. Notice that this will trigger autoloading all the command classes and also `FooService` and `BarService`, but not `BazService`, which is lazily invoked only when a `BazCommand` arrives.

When you configure which commands should be delivered to which services, you can invoke the proper service to process an incoming command from any place in your app using the command bus.

[32] http://api.rubyonrails.org/classes/Rails/Railtie/Configuration.html#method-i-to_prepare

```
Rails.configuration.command_bus.call(FooCommand.new)
```

This can be in a controller:

```
class ProductLikesController < ApplicationController
  def create
    cmd = LikeProductsCommand.new(params)
    Rails.configuration.command_bus.call(cmd)
    head :no_content
  end
end
```

Or in a Saga (more about them later):

```
def perform(serialized_event)
  event = YAML.load(serialized_event)
  State.get_by_order_number(event.data.fetch(:order_number)) do |state|
    case event
      when Orders::OrderShipped
        state.data[:shipped] = true
      when Payments::PaymentAuthorized
        state.data[:transaction] = event.data.fetch(:transaction_identifier)
      else
        raise ArgumentError
    end

    if !state.data[:completed] && state.data[:shipped] && state.data[:transactio\
n]
      Rails.configuration.command_bus.call(CapturePaymentCommand.new(
        order_number: event.data.fetch(:order_number),
        transaction_identifier: state.data[:transaction],
      ))
      state.data[:completed] = true
    end

  end
end
```

1-1 mapping

You might be wondering why **one command** can be handled by **one service** only. Why can't we register two or more?

But if you think about it, it's logical. The command must succeed or fail. If we had multiple services being called for a single command and one succeeded and one failed, what would that mean for the caller object who issued the command? Did it fail or not? We need to know and react accordingly.

That's why there can be only one handler (one service) registered for the same command.

command bus as routing layer but on application level

If you haven't noticed yet, the command bus is conceptually quite similar to Rails routing but on different layer. Obviously they also use different syntax, but that's a minor difference. The basic pattern is the same.

The Rails HTTP router maps every HTTP request to a single controller that can handle it–usually based on the path, but sometimes also based on the domain, parameters and other constraints. Notice that the HTTP request must return a single HTTP response with just one HTTP code such as 200 or 404. It must succeed or fail. There can be one and only one controller that can handle an HTTP request, process it and return an answer. However, it's possible for many different requests to be handled by the same controller, and the controller does not care.

```
1  Rails.application.routes.draw do
2    root to: 'orders#index'
3    resources :orders, only: %i(index)
4  end
```

In such cases, `OrdersController#index` can handle `/` and `/orders` requests.

The command bus, when used outside services (above them) can be much the same. It maps a command to a single handler (usually a service) that can handle it. In this case the mapping would be based on the class of the command (mapping based on command content is not supported, and so far we haven't needed it). Processing the command must either succeed or fail. There can be one and only one service that can handle the command, process it and return an answer. However, it's possible for many different commands to be handled by the same service, and the service may not care.

```
1  bus = Arkency::CommandBus.new
2  service = Service.new
3  bus.register(FooCommand, service)
4  bus.register(BarCommand, service)
```

Are there asynchronous commands?

I have often struggled with this question as well. Here is what I think right now.

The commands are always synchronous. They must succeed or fail. However, it is just a matter of naming. If the results of your command are delivered asynchronously to you, just name it appropriately. Instead of naming it `DoIt`, name it `ScheduleDoingIt`. Now if the command succeeds, you know that it is not *doing it* that went ok, but merely *scheduling to do it* that worked. After all, it is still possible that the *scheduling* itself fails (especially in case of networking problems). So even such a trivial command must succeed or fail.

If the name is right, everything is clear as to what worked and what didn't. Asynchronously, you will receive a notification if *doing it* worked or not, but when the command worked, at least you know immediately that *scheduling to do it* went just fine.

Sidenote

Don't confuse the Command object with a Command pattern[33]. In our solution the command does not know anything about how it is going to be executed.

Questions

If you would like something to be clarified or you have some questions, email us at support@arkency.com . I can't promise an answer but will do my best to help. Maybe even I will write a blog-post with an explanation.

Read more

- CQRS Documents by Greg Young[34]

  This free mini-book has a chapter devoted to Task Based UI and it explains how introducing commands such as "Complete a Sale", "Approve a Purchase Order", "Submit a Loan Application" makes it possible to not miss the user's intention. And the intention is often lost if we just send bunch of attributes from DB to View and back.

[33] https://en.wikipedia.org/wiki/Command_pattern
[34] https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf

Domain Events

Domain events are structures describing **what happened** in your app. They are a gateway drug to enable lots of other techniques. Used correctly, they can be beneficial and valuable on their own for debugging (*what the heck happened in my app, why did it behave like that…*) and audit logging (*who the hell changed this to that and caused all those problems that we have right now*).

The anatomy of Domain Events

Almost 2 years and over 16 million domain events ago I started the process of "switching the mindset". I had no production experience with Event Sourcing (BTW it still is used only in some parts of the application, but that's a topic for another chapter). I had only a limited experience with Domain Driven Design (mainly knowing the tactical patterns). Since then, a lot has changed.

Start from the middle

I've started introducing new concepts in our project's code base from the middle. Not with Domain Driven Design, not with Event Sourcing or CQRS (Command Query Responsibility Segregation). It all started by just publishing Domain Events.

Trials & errors

The more Domain Events we "published" the better my understanding of our domain became. Also, I started to better understand the core concepts of Domain Driven Design, with terms like *bounded context*, *context map* and others from strategic patterns of DDD making more sense and becoming more and more important. Of course, I've made a few mistakes, some of them still bite us because of decisions made almost 2 years ago ;)

Our first ever published domain event is:

```
RailsEventStore::Client.new.read_all_streams_forward(:head, 1)
=> ProductItemEvents::ProductItemSold
     @event_id="74eb88c0-8b97-4f27-9234-ed390f72287c",
     @metadata={:timestamp=>2014-11-12 22:20:24 UTC},
     @data={:order_id=>23456, :product_item_id=>123456,
            :attributes=>{
               "id"=>123456, "order_id"=>23456, "product_type_id"=>98765,
               "price"=>50, "barcode"=>"1234567890",
               "scanned_at"=>nil, "serialized_type"=>nil,
               "order_line_id"=>3456789, "code_id"=>nil,
               "updated_at"=>2014-11-12 22:20:24 UTC, "created_at"=>2014-11-12 2\
2:20:24 UTC}}
```

Here is a set of rules/things to consider when you build your domain events. Each of them is based on a mistake I've made ;)

Rule #1: Naming is hard - really hard

In my old talk[35] I've presented at dev's user group meetup I've defined a domain event as:

- Something that has already happened, and therefore...
- Should be named in past tense...
- ... and in business language (Ubiquitous Language)
- Represents a state change,
- Something that will never change (the record is written once and never edited afterwards)

The name of a domain event is extremely important. It is the "definition" of an event for others. It brings a lot of value when defined right, and it can be misleading when it doesn't capture the exact business change.

In the example event above, the name of the domain event is `ProductItemSold`. This name is not the ideal one. The application domain is not selling some products but selling tickets for events (actually that's huge simplification but it does not matter here). We do not sell products. We sell tickets. This domain event should be named `TicketSold`. Yeah, sure we could also sell some other products but then it should be a different domain event.

Rule #2: don't be CRUDy

There are very few domains where something is really created. Every time I see a `UserCreated` domain event I feel that is not the case. The user might have registered (`UserRegisteredFromEmail`, `UserJoinedFromFacebook`), the user might have been imported (`UserImportedByAdmin`), I don't

[35] http://praglowski.com/presentations/cqrses/#/14

know the case when we really create a user (he or she exists already ;P). Don't stop when your domain expert tells you that something is created (updated or deleted). It is usually something more, something that has real business meaning.

Prefer intention revealing names: `UserChangedPassword` over `UserUpdated`.

And one more thing: don't talk CRUD to your domain expert/customer. When she/he starts to talk CRUD, you are in serious trouble.

see on twitter[36]

Rule #3: your event is not your entity

You might have spotted the `attributes` in the data argument of our first domain event. This is something I dislike most about that domain event. Why? Because it creates a coupling between our domain event & our database schema. That kind of coupling is bad. Especially when you try to build a loosely coupled, event-driven architecture for your application. The attributes of a domain event are their contract. It is not something you could nor should easily change. There could be parts of the system that rely on that contract. Changing it is always a trouble. Avoid that by applying Rule #4.

Rule #4: be explicit

The `serialized_type`? Is this a business term? Really? Or does the business care about the `scanned_at` when a ticket has been just sold? I don't. And all event handlers for this event do not care. That's

[36] https://twitter.com/andrzejkrzywda/status/650272559733321728

just pure garbage. It holds no meaningful information there. It just messes with your domain event contract, making it less usable and more complicated. Explicit definition of your domain event's attributes will not only let you avoid those unintentional attributes in the domain event schema but force's you to think what really should be included in the event's data.

```
1 => TicketSold
2      @event_id="74eb88c0-8b97-4f27-9234-ed390f72287c",
3      @metadata={:timestamp=>2014-11-12 22:20:24 UTC},
4      @data={:barcode=>"1234567890",
5             :order_id=>23456, :order_line_id=>3456789,
6             :ticket_type_id=>98765,
7             :price=>{Price value object here}}
```

This is the modified version of my first domain event. Much cleaner. All important data is explicit. Clearly defined contracts on what to expect. Maybe some more refactoring could be applied here (TicketType value object & OrderSummary value object that will encapsulate the ids of other aggregates). Also, an important attribute was revealed here: the ticket's barcode.

Rule #5: natural is better

With the explicit definition of domain event schema, it is easier to notice that we do not need to rely on database's id of a ticket (`product_item_id`) because we already have a natural key to use - the `barcode`. Why is the barcode better? Natural keys are part of the ubiquitous language, are the identifications of the objects you & your domain expert will understand and will use when you talk about it. Natural keys also will be used in most cases on your application UI (if not you should rethink your user experience). When you want to print the ticket, you will use the barcode as identification. When you validate the ticket near the venue entrance, you scan the barcode. When some guest has troubles with his ticket your support team asks for the barcode (...or order number, or guest name if you're doing it right ;)). The barcode is the identification on the ticket. The database record's id is not. Don't let your database leak through your domain.

Rule #6: time is a modeling factor

> "Modeling events forces temporal focus"

said Greg Young at DDD Europe 2016[37]

How do things correlate over time, what happens when this happens before this becomes a real domain problem. Very often our understanding of the domain is very naive. If you don't include time as a modelling factor your model might not reflect what is happening in the real world.

[37] http://youtube.com/watch?v=LDW0QWie21s

Rule #7: when in doubt

TALK TO YOUR DOMAIN EXPERT / BUSINESS

What do we do with Domain Events?

- **persist** in database / event store

  This can be your standard SQL database (in the case of `rails_event_store`) for example or a dedicated database for storing events (such as https://eventstore.org/).

  Persisting is what enables debugging and auditing and the main difference between this solution and classic in-memory pub/sub libraries such as wisper.
- **publish it for subscribers** using a pub/sub mechanism

  Interested subscribers (called handlers) receive the domain event and can synchronously or asynchronously react to them.

We will discuss reacting to domain events more in further chapters and see what kind of techniques they enable.

Domain Events Schema Definitions

When we started a few years ago, publishing Domain Events in our applications, we were newbies in the DDD world. I consider the experiment to be very successful but some lessons had to be learned the hard way.

At the very beginning, we were just publishing events. We didn't think much about consuming them. We haven't yet considered them a very powerful mechanism for communication between the application subsystems (called Bounded Contexts in DDD world). And we didn't think much about how those events would evolve in the future.

Nowadays, one of our events has 18 handlers. And I believe this number will continue growing.

We also started using domain events in many smaller test i.e. tests for one class or one sub-system.

So at some point in time, it became necessary that what we publish in code, what we expect in tests and what we use to set up a state in tests, has precisely the same interface. All those events should contain the same attribute names and the same types of values in them.

For that, I used `classy_hash` gem which raises useful exceptions when things don't match.

```
class PaymentNeedsMoreTime < RailsEventStore::Event
  SCHEMA = {
    order_id:             Integer,
    payment_id:           Integer,
    payment_gateway_name: String,
    seconds_needed:       Integer,
  }

  def self.strict(data, **attr)
    ClassyHash.validate(data, SCHEMA, true)
    new(data, **attr)
  end
end
```

I tried an approach in which the event schema is validated in a constructor phase (`new/initialize`) but later decided against it. In a few very rare cases we might be OK with an event which is not completely full (not all attributes are present). When we get historical events from event store, we don't want (or need) to verify the schema as well.

So instead when you want to verify the schema (in 97% of cases) you should just use the `strict` method to create the event instead of `new`.

```
stream_name = "payment_#{payment.id}"
event = PaymentNeedsMoreTime.strict(
  order_id: payment.order_id,
  payment_id: payment.id,
  payment_gateway_name: "v8",
  seconds_needed: 30*60*60,
)
client.publish_event(event, stream_name)
```

`classy_hash` supports nullable keys (value is nil), optional keys (value not present), multiple choices, regular expressions, ranges, lambda validations, nested arrays, and hashes.

I know some people who use `dry-types` for defining events' schema and they were happy with that library as well.

With the 220 domain events that we already publish, with every new one that I add, I remember to define its schema. That way **it's much easier for every other team member to know what they can expect in those events just by looking at their definition**.

There is a big difference if you read this code:

```
class OrderShipped < RubyEventStore::Event
end
```

and you have no idea what's inside such domain events.

Compare it to reading this:

```
class OrderShipped < RubyEventStore::Event
  SCHEMA = {
    order_number: String,
    customer_id:  Integer,
  }.freeze

  def self.strict(data:)
    ClassyHash.validate(data, SCHEMA, true)
    new(data: data)
  end
end
```

Now you have a much better understanding what you can expect from this domain event. You know the attributes that are there and the attributes which are not.

In many DDD books, you may learn a technique that your bounded context can deserialize the message under a different name and have different accessors; for example in CRM context `customer_id` could be `lead_id`. However, if you use `YAML` for events serialization (default, that I will be showing in examples) that is not possible because YAML is highly coupled to class names and classes. But it is possible with other formats such as JSON (other bounded contexts can read JSON and create an instance of a different struct if needed).

One action/command/request can trigger multiple events

It took me quite to some time to understand and appreciate this. Often when there is one HTTP request or one command it can lead to several domain events being published. Maybe the code just did multiple things. Especially when the logic is more complex.

Good indicators for you to consider splitting into smaller events are: nils, empty arrays and other collections, zeros, etc. It doesn't mean that you always need to split but pay attention when you have many attributes like that.

It might be that instead of `FooHappened` with a bunch of empty data, you might need two events. `FooHappened` and `BarHappened`. And sometimes `Bar` does not happen and you don't publish such domain event at all.

In other words, instead of:

```
FooHappened.strict(data: {
  one: 1,
  two: "two",
  three: 3.0,
  four: nil,
  five: [],
  six: {},
})
```

you have

```
FooHappened.strict(data: {
  one: 1,
  two: "two",
  three: 3.0,
})
```

and instead of

```
FooHappened.strict(data: {
  one: 1,
  two: "two",
  three: 3.0,
  four: "4",
  five: [5],
  six: {6:6},
})
```

you have

```
FooHappened.strict(data: {
  one: 1,
  two: "two",
  three: 3.0,
})

BarHappened.strict(data: {
  four: "4",
  five: [5],
  six: {6:6},
})
```

Such approach might be more intention revealing, more clear.

Example 1

Here is an example from one of our projects.

1. `BarcodeScanned`
2. `RulesValidatedPositively`
3. `EnteredVenue`

When you come to a stadium with a ticket or season pass, we scan and recognize a barcode. That's one thing. Then business rules are evaluated (whether you can enter today, using this gateway, etc.). Assuming everything is OK, that's the 2nd thing which happened. And you enter the venue so that's the 3rd thing.

But if you come with a Coca-Cola can and put it to the scanner, the barcode will also be scanned and recognized. But you won't enter the venue :)

And it can happen that the rules can be verified on a different device and during sync we just know that you entered a venue. This is what happened. What an offline device decided to do.

Distinguishing between more granular domain events can help you. Different handlers subscribe to them and react. So the handler which keeps track of how many people are inside a venue does not need to react to `BarcodeScanned` and check inside attributes as to whether that was a successful scan or not. Instead, it can subscribe to more intentional, smaller event such as `EnteredVenue`.

Example 2

Here is another example.

1. `InvoiceRequested`
2. `InvoiceGenerated`
3. `CreditNoteDeliveryScheduled`
4. `InvoicePaymentReminderScheduled`

When someone asks for payment with an invoice (and bank transfer) they just send us one command. But we do a few things. We register the fact they requested an invoice, we generate the invoice and we schedule two jobs for the future.

At a certain moment we send the customer a reminder that they should pay the invoice, in case they didn't. And we also schedule automatic credit note delivery in case they didn't pay in the required time. But if there is very little time to pay (like a day or two), we don't schedule the reminder at all.

Publishing events on failures

A few years ago I wouldn't care that much about failures. Maybe there would be failed Sidekiq/Resque job, or exception in Honeybadger and that would be the only trace.

Nowadays I often treat them as a regular, expected part of the application and domain I work on. In some contexts this can be valuable and important - an error rate can be an important metric.

As an example, we have a process manager (described in bonus chapter later) which switches payment gateways if one is failing.

So here are some examples of domain events, which are all about failures:

- `StartRemoteTransactionFailed`

  Some payment gateways that we work with require an API request to create the transaction on their side first. This is done synchronously and can be very problematic when not working correctly. It's good to know and be able to react to such failures.
- `PaymentFailed`

  A bit different than the previous one. The technical infrastructure worked but the client probably did not have enough money. But it could be also that there were problems with 3D-Secure or payment authorization.
- `InvoiceDeliveryFailed` / `InvoiceDeliveryRescheduled`

  Obviously, we want our customer to pay their invoices. But it is hard to expect it if delivering the email containing it fails. In such case, it is good to know when it was rescheduled to.
- `ErrorProcessingBigBankFile`

  There is a BigBank™, which sends us improperly formatted CSV files (I know, facepalm...) that we download and process daily. Most of the time the file is good enough to automatically unmangle it. But sometimes it is messed up beyond our code capabilities. This requires developer attention, business and customer support awareness as it delays some automatic processes that happen on the platform. Sounds important enough to deserve a domain event.
- `SalesforceEvents`
  - `AccountCreateFailed`
  - `OpportunityUpdateFailed`
  - `OpportunityOwnerNotFound`

  In one project our integration with Salesforce can be sometimes be a bit ... fragile. The way they handle permissions for APIs is still a bit of mystery for me. We have domain events published when the integration is not working correctly with the errors included. Based on those cryptic errors sales people can change Salesforce data and fix the integration without a developer's help.
- `OrderCannotBeCompleted`

  Even when a payment is successful, when we try to complete the order and ship all the related virtual goods, errors can still occur (due to bugs or more likely final business rules being

violated). It's a very important domain event because we have the customer's money but we can't ship the products. Later you will learn how you can compensate in such situations.

- `OrderEmailDeliveryFailed`

  Even when you got paid, completed all the virtual products you want to deliver, the customer might still not get them because sending an email can fail.

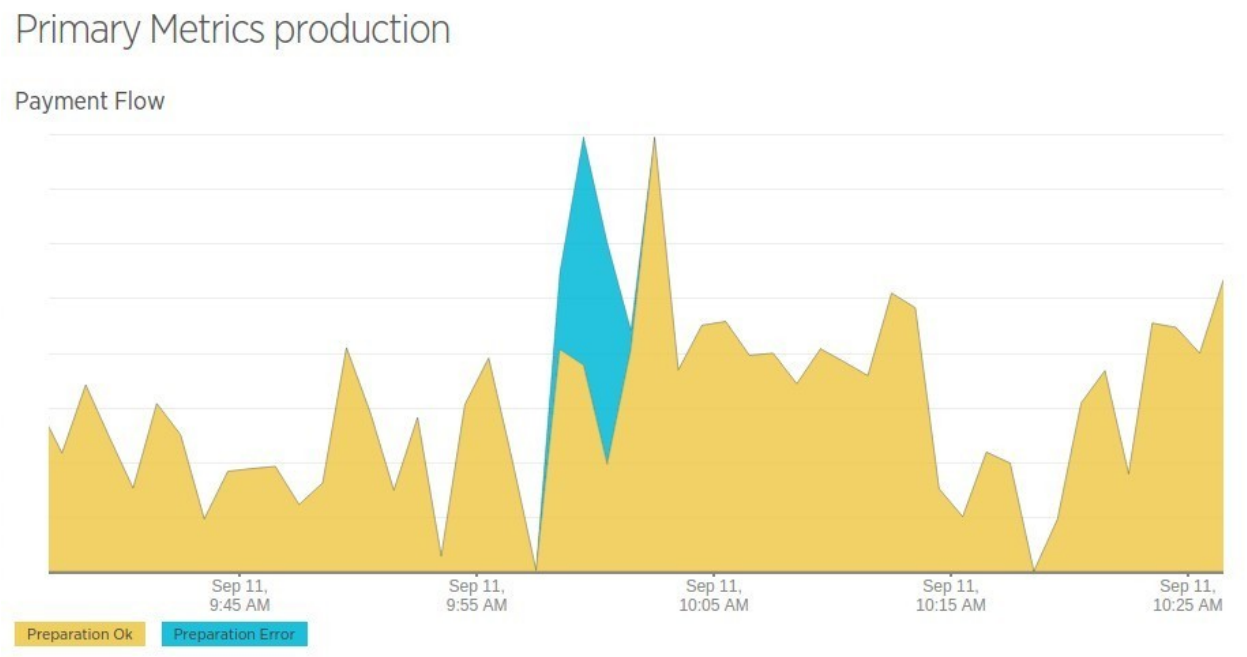
- `Postal::UploadFailed`

  Similar to the last one, but related to the integration with postal API. When a customer wants tickets printed and shipped by a real postman, it is important to know and react when the integration fails.

TIMESTAMP	FACT TYPE	USER ID	EVENT ID	MESSAGE
2016-09-18 07:05:33 UTC	SalesforceEvents::OpportunityCreateFailed		153445 [H]	INACTIVE_OWNER_OR_USER: operation performed with inactive user [005b0000000ERWd] as owner of opportunity
2016-09-18 07:05:33 UTC	SalesforceEvents::OpportunityCreateFailed		153444 [H]	INACTIVE_OWNER_OR_USER: operation performed with inactive user [005b0000000ERWd] as owner of opportunity
2016-09-18 07:05:32 UTC	SalesforceEvents::OpportunityCreateFailed		153443 [H]	INACTIVE_OWNER_OR_USER: operation performed with inactive user [005b0000000ERWd] as owner of opportunity

Integration errors panel

As I mentioned, based on such domain failures we can show a list of recent failures. It might seem cryptic but people using Salesforce can find the problem using the mentioned IDs and fix it.



Metrics based on domain events

If you have a metric tracker such as get.chillout.io[38] for example you can combine them together and send metrics based on domain events. In this graph, we see how the integration with a payment gateway works.

Where should I publish my domain events?

There is no single good answer. You will have to find out which style suites best you and your team.

In a model

```
1  class Basket < ActiveRecord::Base
2    has_many :items, autosave: true
3
4    def add_item(sku, quantity)
5      changed_item   = items.find{|oi| oi.sku == sku }
6      changed_item ||= items.build(
7        sku: sku,
8        quantity: 0
9      )
10     changed_item.quantity += quantity
11     self.total_quantity += quantity
12     event_store.publish_event(ItemAddedToBasket.strict(data: {
13       basket_id: id,
14       sku: sku,
15       quantity: quantity
16     }), stream_name: "Basket$#{id}")
17   end
18
19   private
20
21   def event_store
22     Rails.application.config.event_store
23   end
24 end
```

There is a downside to that approach, however. Calling `#add_item` changes in-memory state of `Order` (that's good) and immediately publishes `OrderItemAdded` domain event, even if we don't save the changes (that's bad because we can have a domain event in a DB, which did not happen and we might have notified some subscribers/handlers as well).

So let me show a slightly different approach.

[38] http://get.chillout.io

Model + callback

```
class Basket < ActiveRecord::Base
  has_many :items, autosave: true

  def unpublished_events
    @unpublished_events ||= []
  end

  after_save do
    unpublished_events.each do |ev|
      event_store.publish_event(ev, stream_name: "Basket$#{id}")
    end
  end

  def add_item(sku, quantity)
    changed_item   = items.find{|oi| oi.sku == sku }
    changed_item ||= items.build(
      sku: sku,
      quantity: 0
    )
    changed_item.quantity += quantity
    self.total_quantity += quantity

    self.unpublished_events << ItemAddedToBasket.strict(
      basket_id: id,
      sku: sku,
      quantity: quantity
    )
  end

  private

  def event_store
    Rails.application.config.event_store
  end
end
```

In this approach domain events are published only after `save` is called. I know it might look a bit unusual but in the chapter about event sourcing we will further explain the role of `unpublished_events`.

One nice aspect of this approach is that after invoking `#add_item` method, all the changes are only done in-memory and there are no side effects. That makes testing easier.

Service

It's OK to publish your domain events from a Service if that's the easiest layer for you to add it in your current architecture.

```
class BasketsService
  def initialize(event_store)
    @event_store = event_store
  end

  def add_item(basket_id:, sku:, quantity:)
    ActiveRecord::Base.transaction do
      basket = Basket.lock.find(basket_id)
      item = basket.add_item(sku, quantity)
      basket.save!
      @event_store.publish_event(ItemAddedToBasket.new(data: {
        basket_id: b.id,
        sku: item.sku,
        quantity: item.quantity}),

        stream_name: "Basket$#{basket.id}"
      )
    end
  end
end
```

Controller

I would not recommend it, but as a last resort and hack you can publish your domain events from controllers. Please consider it a technical debt. This code basically screams for extracting into a service as a first refactoring step.

```
class BasketItemsController < ApplicationController

  def create
    ActiveRecord::Base.transaction do
      basket = Basket.lock.find(params[:order_id])
      item = basket.add_item(params[:order_item])
      basket.save!
      event_store.publish_event(ItemAddedToBasket.new(data: {
        basket_id: basket.id,
        sku: item.sku,
```

```
11          quantity: item.quantity}),
12
13          stream_name: "Basket$#{basket.id}"
14        )
15      end
16
17      head :ok
18    end
19
20  end
```

Stream name

You might have noticed that when we publish an event we also provide a `stream_name`.

```
event_store.publish_event(
  ItemAddedToBasket.new(data: {
    basket_id: b.id,
    sku: item.sku,
    quantity: item.quantity}
  ),
  stream_name: "Basket-#{basket.id}"
)
```

Usually, the stream name is the UUID of an entity or a concatenation of an object's class name and its ID.

```
Product$1
Product-123
21aa8e11-e2ea-4082-9ce6-39ff294c5e57
products-21aa8e11-e2ea-4082-9ce6-39ff294c5e57
```

Providing distinct stream names allows us later to query for events in a particular stream. This makes it easier to debug how a certain entity or an aggregate reached a quirky state. Instead of querying all potential events that ever happened (and verifying in their data the ID of an object which we are interested in) you can easily get all events from a specific stream such as `Product-123`.

```
event_store.read_stream_events_forward("Product-123")
```

In Rails Event Store right now an event can live in only one stream. But we are working on making it possible to link events to many streams[39].

The concept of a stream is useful for grouping/partitioning related events together. Usually, this is based on the subject, which was affected by a certain change.

But you can probably easily imagine also different groupings. Streams based on:

- time
  - daily stream of events which happened in that day - `day-2018-02-03`
  - weekly - `week-2018-44`
  - etc
- geo-location
  - all events related to a certain country - `country-Canada`
  - or a certain city - `city-New York`
- user/system who triggered given action - `requestor-3467120`
- long business process - `reservation-ANQYHC`

They all can be useful in certain scenarios. If you want to notice irregularities, outstanding users which might be trying to attack your system then maybe having a log of all changes caused by a single user is useful for your system to analyze.

Questions

If you would like something to be clarified or you have some questions, email us at support@arkency.com . I can't promise an answer but will do my best to help. Maybe even I will write a blog-post with an explanation.

Exercise

If you purchased exercises along the book, you can now make the *Domain Events* task.

Read more

Domain-Driven Design Distilled[40] has examples of domain events and discusses which properties they should include.

[39] https://github.com/RailsEventStore/rails_event_store/issues/134

[40] https://www.amazon.com/gp/product/0134434420/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0134434420&linkCode=as2&tag=arkency-20&linkId=12c564c85da17f918d275bdc51626bde

In Implementing Domain-Driven Design[41] you can find which words uttered by the business you should pay attention to because they suggest a domain event. It discusses naming events, event enrichment and more.

Links

- What do you mean by "Event-Driven"? by Martin Fowler[42]

  > The biggest outcome (...) was recognizing that when people talk about "events", they actually mean some quite different things. (...) This note is a brief summary of the main ones we identified.

  It lists techniques such as:
  - Event Notification
  - Event-Carried State Transfer
  - Event-Sourcing
  - CQRS

  And problems such as when **an event is used as a passive-aggressive command**.
- Martin Fowler on Domain Events[43]

  > Events are about something happening at a point in time, so it's natural for events to contain time information. When doing this it's important to consider two Time Point that could be stored with the event: the time the event occurred in the world and the time the event was noticed . These Time Points correspond to the notions of actual and record time.

[41] https://www.amazon.com/gp/product/0321834577/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0321834577&linkCode=as2&tag=arkency-20&linkId=3155894f09101a9da242cf5cb6d9bee7

[42] https://martinfowler.com/articles/201701-event-driven.html

[43] http://martinfowler.com/eaaDev/DomainEvent.html

Consuming domain events with event handlers

Now that you've learned about publishing domain events, you need to start using them in your application for decoupling code and maintaining boundaries.

There are two possibilities here:

1. **Reacting in the same bounded context**

   That would usually mean updating a different aggregate, i.e., one aggregate publishes a domain event describing what happened and another one reacts to it.

   For me, personally, this is not the most interesting case. I am usually not that worried about updating two aggregates in the same bounded context in one transaction, in one service, without going through domain events and event handlers.
2. **Reacting in different bounded contexts**

   This is the more interesting case: the domain event is used as a message crossing the boundary of a bounded context, and the handler is used to trigger a transition, a state change or a reaction in a separate part of the application.

Event handler examples

- **Order reacting to Payment being paid**

  After a few projects which involved handling transactions and money there is one thing I learned for sure. Getting the money and handling orders, on the one hand, and handling their delivery on the other, are completely different processes with different requirements. You can get money for orders that expired or were canceled. You can get the money but not be able to deliver the product. There are thousands of reasons to keep Payments as a separate context in your app.

  Usually, the complexity of reacting to payment is much greater than just marking payment as paid. In all the projects that I have worked on, the logic of the Payment being paid itself was minimal: just an update here or there, nothing big. But the logic of the whole system that now needs to react to it? 10 times bigger. It does not matter if it was about granting access to movies, delivering tickets or selling pillows. The business logic is more complex, more prone to failure in other parts of the system. And you can't risk a rollback of the information that the payment was successful.

  The obvious best solution for those situations is to keep the logic separate. Payment was paid, cool. Now let the rest of the system react to it.

- **Scanner (when refunded -> revoke access)**

  In one project, a ticketing application, we have a Scanner bounded context. It might sound strange, because how complicated could it be to scan a barcode? Remember that it was `scanned_at`, and that's it, right? That's what we thought as well, but it turned out to be a more complex beast. You learn that you need to support leaving the venue (for a smoke or breath of fresh air) and coming back. There are also multi-day events, when every day you may return and need to be scanned again. There are venues with multiple gates and sometimes you must use a certain gate to enter the venue, e.g., in a stadium you might be required to use the gate closest to your seat. So technically you may enter, but not from this side of the stadium. And there are venues with poor Internet connections that need to support offline scanning and synchronization in the background. When you combine all of that, you have a bounded context :)

  In this system, when a ticket/order gets refunded, the scanner context reacts to it and revokes access for certain barcodes.
- **Unbook seats when placing order failed**

  In one part of the code, we react to an error when placing an order by unbooking seats that were previously reserved for it.
- **Pushing to segment.io, getvero, salesforce and other data sinks**

  In the world of multiple services available via API which help us build better software, oftentimes when something important happens we need to notify them with some payload.
- **Sending metrics when payment paid**

  It's also nice to send metrics (i.e. to chillout[44]) so admins, CEOs, CTOs and others can see what's going on in our system and so DevOps can react in case of an emergency.
- **mobile/sms notifications**
- **sending emails when user imported**
- **read models**

  (will be covered in next chapter)

In sum, when we hear about event handlers, our first thought should be about **SIDE-EFFECTS**.

Services and Side Effects

Here is something that I noticed in Rails apps which introduced Service Objects and then grew over time. The services often take a bunch of objects as dependencies. However, they are often not directly related to the core of an action. The service does not require co-operation with these dependencies to fulfill their jobs. They are there just to trigger the side-effects of the actions being executed. These dependencies often make the action more complicated, harder to understand.

[44] get.chillout.io

```ruby
class UserService
  def initialize(a, b, c, d)
    @a, @b, @c, @d = a, b, c, d
  end

  def register
    register_user

    @a.user_registered(user)
    @b.notify_about_user(user.id)
    @c.greet_user(user.email)
    @d.new_user_happened(user.login)
  end
end
```

So here is an exemplary, simplified service for user registration. It takes a bunch of dependencies (`@a`, `@b`...) but they are only being called after the user is registered. They are not being used to decide whether a new user can register or not.

Many parts of our system need to know that something happened, but they don't participate in that thing happening.

Here are some additional questions worth asking:

- **What if A, B, C, or D fails? Should we roll the transaction back?**

  If so, then you might have better consistency (assuming the dependencies do not contact other databases or APIs). But it might also mean that a small problem or bug, in a less-important part of your system prevents the user from registering.

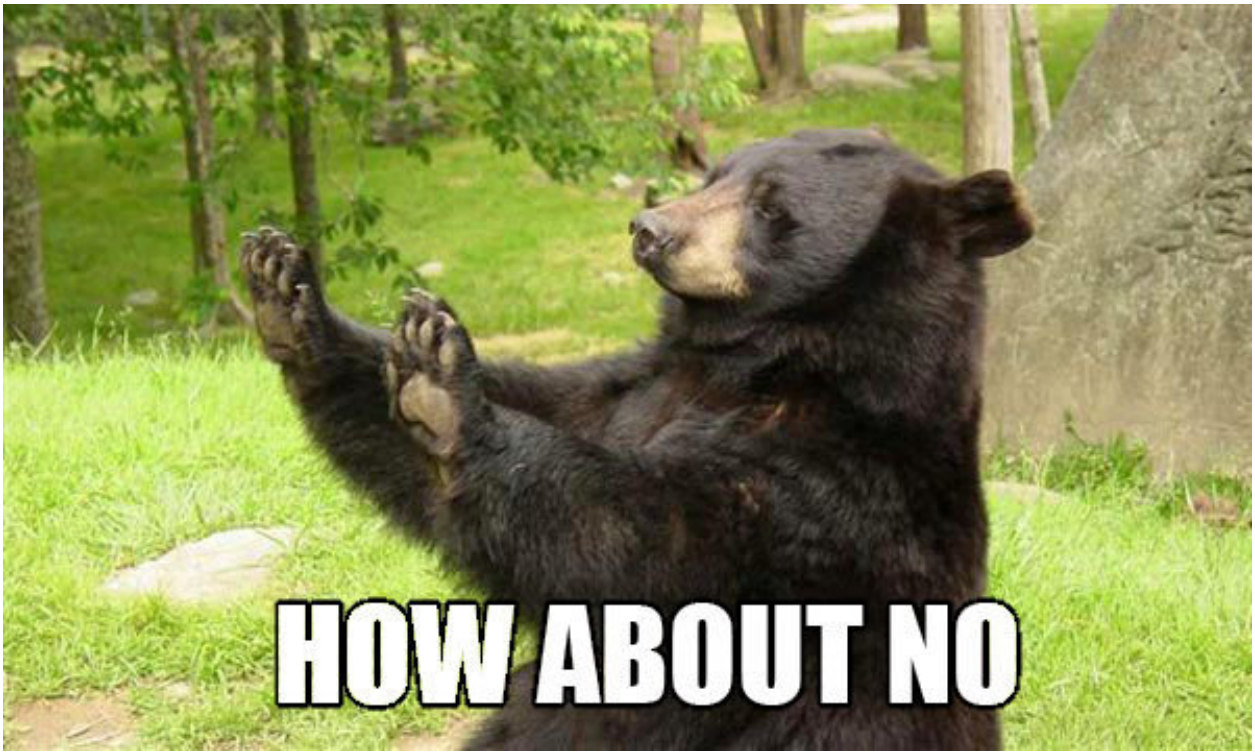
  Often it might be better if the user registers, A, B, C react properly and are in a consistent state but when D fails, only D remains affected and there is small inconsistency in our system in one of its parts. Developers can get notified with an exception tracker about the problem in D and fix it manually or automatically (maybe via retrying) without blocking the (more important) user registration.
- **Do we want to wait for A, B, C, D? Do we need to?**

  All of those things take time. Perhaps it's better to show the registration result faster to the customer.

So what's the alternative? You can publish a domain event and have those interested parts react to it.

But ActiveRecord callbacks...

You might be tempted to say *but Robert, I can use ActiveRecord callbacks for that...* Hmm...



Some time ago my coworker wrote an article about ActiveRecord callbacks being the biggest code smell in Rails apps[45]; among other reasons, they can easily get out of control. The article was posted on Reddit and a very interesting comment appeared there[46]:

Imagine an important model, containing business-vital data that changes very rarely, but does so due to automated business logic in a number of separate places in your controllers.

Now, you want to send some kind of alert/notification when this data changes (email, text message, entry in a different table, etc.), because it is a big, rare, change that several people should know about.

Do you:

A. Opt to allow the Model to send the email every time it is changed, thus encapsulating the core functionality of "notify when changes" to the actual place where the change occurs?

Or

B. Insert a separate call in every spot where you see a change of that specific Model file in your controllers?

[45] https://medium.com/planet-arkency/the-biggest-rails-code-smell-you-should-avoid-to-keep-your-app-healthy-a61fd75ab2d3#.q537fl3g5
[46] https://www.reddit.com/r/ruby/comments/4hr125/the_biggest_rails_code_smell_you_should_avoid_to/

I would opt for A, as it is a more robust solution, future-proof, and most to-the-point. It also reduces the risk of future programmer error, at the small cost of giving the model file one additional responsibility, which is notifying an external service when it changes.

The author brings very interesting and very good points to the table. I, myself, used a callback just like that a few months ago…

```
class Order < ActiveRecord::Base
  after_commit do |order|
    Resque.enqueue(IndexOrderJob,
      order.id,
      order.shop_id,
      order.buyer_name,
      order.buyer_email,
      order.state,
      order.created_at.utc.iso8601
    )
  end
end
```

…to schedule indexing in an ElasticSearch database. It was the fastest solution to our problem. But I did it knowing that it does not bring us any further in terms of improving our codebase. However, I knew that at the same time we were doing other things which would help us get rid of that code later.

So despite the undeniable usefulness of those callbacks, let's talk about a couple of problems with them.

They are not easy to get right

Imagine very similar code such as:

```
class Order < ActiveRecord::Base
  after_save do |order|
    Elasticsearch::Model.client.index(
      id: id,
      body: {
        id:              id.to_s,
        shop_id:         shop_id,
        buyer_name:      buyer_name,
        email:           buyer_email,
        state:           state,
        created_at:      created_at
```

```
12      })
13    end
14  end
```

At first sight, everything looks all right. However **if the transaction gets rolled back** (for example, saving an Order can be part of a bigger transaction that you open manually) **you would have indexed incorrect state in the second database**. You can either live with that or switch to `after_commit`.

But then, what happens if we get an exception from Elastic? It would bubble up and rollback our DB transaction as well. You can think of it as a good thing (we won't have inconsistent DBs, there is nothing in Elastic and there is nothing in the SQL DB) or a bad thing (an error in the less important DB prevented someone from placing an order and us from earning money).

So let's switch to `after_commit` which might be better suited to these particular needs. After all, the documentation says:

These callbacks are useful for interacting with other systems since you will be guaranteed that the callback is only executed when the database is in a permanent state. For example `after_commit` is a good spot to put in a hook to clearing a cache since clearing it from within a transaction could trigger the cache to be regenerated before the database is updated

So in other words. `after_commit` is a safer choice if you use those hook to integrate with 3rd party systems/APIs/DBs. `after_save` and `after_update` are good enough if the side-effects are stored in the SQL DB as well.

```
1   class Order < ActiveRecord::Base
2     after_commit do |order|
3       Elasticsearch::Model.client.index(
4         id: id,
5         body: {
6           id:              id.to_s,
7           shop_id:         shop_id,
8           buyer_name:      buyer_name,
9           email:           buyer_email,
10          state:           state,
11          created_at:      created_at
12      })
13    end
14  end
```

So we know to use `after_commit`. Now, probably most of our tests are transactional, meaning they are executed in a DB transaction because that is the fastest way to run them. Because of that, those hooks won't be fired in your tests. This can also be a good thing because we bothered with a feature

that might be only of interest to very few tests. But it could also be a bad thing if there are a lot of use-cases in which you need those data stored in Elastic for testing. You will either have to switch to a non-transactional way of running tests, use the `test_after_commit` gem[47], or upgrade to Rails 5[48].

Historically (read "in legacy Rails apps"), exceptions from `after_commit` callbacks were swallowed and only logged in the logger, because what can you do when everything is already committed? That's been fixed since Rails 4.2[49], though your stack trace might not be as good as you are used to.

So we know that most of the technical problems can be dealt with one way or the other and you need to be aware of them. The exceptions are what's most problematic and you need to handle them somehow.

They increase coupling

Here is my gut feeling when it comes to Rails and most of its problems. There are not enough technical layers in it by default. We have views (not interesting at all in this discussion), controllers and models. So by default, the only choice you have when you want to trigger a side-effect of an action is between controller and model. That's where we can put our code. But both choices have problems.

If you put your side-effects (API calls, caching, 2nd DB integration, mailing) in controllers you might have a problem with testing it properly, for two reasons. First, controllers are tightly coupled with the HTTP interface, so to trigger them you need to use the HTTP layer in tests to communicate with them. Second, instantiating your controllers and calling their methods is not easily done directly in tests. They are managed by the framework.

On the other hand, if you put the side-effects into your models, you end up with a different problem. It's hard to test the domain models without those other integrations (obviously) because they are hardcoded there. So you must either live with slower tests or mock/stub them all the time in tests.

That's why there are so many blog posts about Service Objects in the Rails community. As the complexity of an app rises, people want a place to put *after save* effects like sending an email or notifying a 3rd party API about something interesting. In other communities and architectures, those parts of code would be called Transaction Scripts[50] or Appplication/Domain/Infrastructure Services[51]. But by default, these are missing in Rails, and that's why everyone (who needs them) is re-inventing services based on blog posts, or using gems (there are at least a few) or new frameworks (hanami[52], trailblazer[53]) which don't forget about this layer.

[47] https://github.com/grosser/test_after_commit
[48] https://github.com/rails/rails/pull/18458
[49] https://github.com/rails/rails/pull/14488
[50] http://martinfowler.com/eaaCatalog/transactionScript.html
[51] http://gorodinski.com/blog/2012/04/14/services-in-domain-driven-design-ddd/
[52] http://hanamirb.org/
[53] https://github.com/apotonick/trailblazer

They miss the intention

When your callback is called you know that the data has changed but you don't know why. Was the Order placed by the user? Was it placed by a POS operator which is a different process? Was it paid, refunded, canceled? We don't know–or we may, based on some `state` attribute, which in many cases is an anti-pattern as well. Sometimes it is not a problem that you don't know this because you just send some data in your callback. Other times it can be problem.

Imagine we want to send a new User a welcome email when they are registered via an API call from mobile, or by using a different endpoint in a web browser, or by joining from Facebook…but not when they are imported to our system because a new merchant decided to move their business with their customers to our platform. In 3 situations out of 4 we want a given side effect (sending an email) and in one case we don't want it. To handle that, we need to know the intention of what happened. `after_create` is just not good enough.

Domain Events + Handlers instead

What I recommend, instead of using Active Record callbacks, is publishing domain events such as `UserRegisteredViaEmail`, `UserJoinedFromFacebook`, `UserImported`, `OrderPaid` and so on… and having handlers subscribed to them that can react to what happened. You can use one the many PubSub gems for that (ie. `whisper`[54]) or the `rails_event_store`[55] gem if you also want to have them saved in the database and thus available for future inspection, debugging or logging.

Over time, your application gets to where, whenever something changes, an event gets published, and you don't need to look for places changing the given model because you know all of them.

P.S.

It only gets worse in Rails 5[56]

TLDR:

- Rails is missing some technical layers
- side-effects in controllers make testing hard because of the HTTP layer
- side-effects in your models
  - hard to test models without those hardcoded integrations
  - slower tests
  - mock/stub those effects all the time in tests
- callbacks increase coupling
- often the part that wants to react to something is in a different bounded context of the system
  - and by writing the callback inside a class you couple them together

[54] https://github.com/krisleech/wisper

[55] http://railseventstore.arkency.com/docs/publish.html

[56] https://www.reddit.com/r/ruby/comments/4j3097/rails_5_activerecord_suppress_a_step_too_far/

Disable some Event handlers in your integration/acceptance tests

Have you ever experienced slow tests in a Rails application because they are always sending emails, resizing images, contacting APIs? Or were you annoyed that you constantly need to stub things here and there so that they don't make external requests?

Here is an idea. Disable some of those handlers completely in your tests, except in places where you actually need them or test them:

- PDF generation
- Image scaling
- Video transformations
- Elastic Search
- Sending Emails
- Sink APIs
- Metrics
- Communicating via FTP with postal service
- Generating heavy-read models
  - cute maps in our example
  - graphs
- etc.

Maybe even consider which handlers to enable (whitelist approach) instead of which handlers to disable (blacklist).

Event handlers

When you decide to introduce event handlers, you have two options: they can be either synchronous or asynchronous.

Sync event handlers

Sync event handlers give you decoupling in space (different code/place executes the logic) but not in time (it happens synchronously, immediately after an event was published).

That means that the event handler's code is executed in the same thread and in the same DB transaction (if it was started).

If you don't feel comfortable with eventual consistency (asynchronicity) you might want to start with sync handlers that you feel safer with.

Here is how you can subscribe a handler (`OrderList::OrderSubmittedHandler`) to an event (`Orders::OrderSubmitted`) using `rails_event_store` gem.

```
Rails.application.config.event_store.tap do |es|
  es.subscribe(
    ->(event) {
      OrderList::OrderSubmittedHandler.new.call(event)
    } ,
    [Orders::OrderSubmitted]
  )
end
```

Exceptions

Because sync handlers are executed in the same open DB transaction, if there is an exception inside them, and you don't catch it, it can rollback the whole operation, including what caused the domain event and previous sync handlers.

That might be something you want, or it might be something you don't want. If you don't want it, then explicitly handle exceptions in handlers. We usually log them and send them to an exception tracker so that developers can intervene when necessary.

```
module OrderList
  class OrderSubmittedHandler
    def call(ev)
      # code ...
    rescue => ev
      ErrorNotifier.notify(ev)
    end
  end
end
```

Async event handlers

Async event handlers give you decoupling in space (logic executed in a different code/place) and in time (it does not happen in the same moment but later): the event handler's code is executed in a separate thread/process and in a separate DB transaction (if you open one).

To use Async event handlers you need to have the infrastructure. You need to have a message queue (MQ) available in your app. Rails ActiveJob and its adapters (delayed_job, sidekiq, resque, etc.) can be used as one as well if that's sufficient for your current needs. And if you use the `delayed_job` or `que` gems you don't even need a separate DB, so the infrastructure does not get more complex.

```
Rails.application.config.event_store.tap do |es|
  es.subscribe(
    ->(event){
      Discounts::Saga.perform_later( YAML.dump(event) )
    }, [Orders::OrderShipped]
  )
end
```

This is an example of the `Discounts::Saga` async handler subscribing to `Orders::OrderShipped` domain events.

`YAML` was used here for the serialization format as it is available and works out of box for most types and structures, but you can go with anything you want: `JSON`, `XML`, binary protocols. This is just an example of the simplest thing that can work.

Similarly with the message bus. You can go with Kafka, RabbitMQ and other options, but using the underlying SQL database and Rails ActiveJob can be simple enough for you at the beginning.

Handling rollbacks and crashes

The more databases and messaging solutions you have the harder it becomes keeping everything in sync and working in case of downtimes and problems. Let's evaluate the potential issues.

SQL-based Message Queue

This is the easiest option that I strongly recommend at the beginning of your journey into the world of asynchronicity, eventual consistency and async handlers: a message bus with a queue in the same SQL database as your app. It really simplifies error handling on many levels. For that you can use the DelayedJob (500 jobs per second) or Que (claims 9,000 jobs per second[57]) gems and their adapters in Active Job.

Imagine these handlers:

```
Rails.application.config.event_store.tap do |es|
  es.subscribe(
    ->(event){
      WelcomeInSystem.perform_later( YAML.dump(event) )
    }, [UserRegisteredFromEmail]
  )
  es.subscribe(
    ->(event){
      GrantFreeCredits.perform_later( YAML.dump(event) )
```

[57] https://github.com/chanks/queue-shootout

```
    }, [UserRegisteredFromEmail, UserJoinedFromFacebook]
  )
end
```

And when there is a crash or rollback after we published a domain event.

```
ActiveRecord::Base.transaction do
  User.create!(...)
  event_store.publish( UserRegisteredFromEmail.strict(...) )

  # ...

  raise ActiveRecord::Rollback
  # or server crash
end
```

What happens in such a situation? In the case of using an SQL DB the situation is quite simple.

- user creation is rolled back
- stored domain events are rolled back (in the default case of the `rails_event_store` gem and using active record for storing those events)
- `WelcomeInSystem` and `GrantFreeCredits` handlers (which are just ActiveJob background jobs saved in the SQL db) are also rolled back.

TL;DR: You have a clean state, like nothing happened.

NO-SQL-based Message Queue

What would happen in such situation if we used a different solution/adapter? What if we used, for example, Redis + Sidekiq. The code looks identical, but the ActiveJob configuration is different.

Handlers (this time backed by Redis and Sidekiq):

```
Rails.application.config.event_store.tap do |es|
  es.subscribe(
    ->(event){
      WelcomeInSystem.perform_later( YAML.dump(event) )
    }, [UserRegisteredFromEmail]
  )
  es.subscribe(
    ->(event){
      GrantFreeCredits.perform_later( YAML.dump(event) )
    }, [UserRegisteredFromEmail, UserJoinedFromFacebook]
  )
end
```

Situation in which there is a crash or rollback after we published a domain event.

```
ActiveRecord::Base.transaction do
  User.create!(...)
  event_store.publish( UserRegisteredFromEmail.strict(...) )

  # ...

  raise ActiveRecord::Rollback
  # or server crash
end
```

What happens in such a situation?

- user creation (SQL) is rolled back
- stored domain events are rolled back (in the default case of the `rails_event_store` gem and using active record for storing those events)
- `WelcomeInSystem` and `GrantFreeCredits` handlers (which are just ActiveJob background jobs saved in Redis) **are not rolled back**.

In other words you can have handlers scheduled and executed for something that you rolled back. Not good. How can we workaround the problem? Same way I described when integrating Rails classic callbacks with multiple DBs integrations. By executing our scheduling after commit. I will describe this strategy in a moment.

Depending on how often crashes and rollbacks occur in your system, I would say this is 99% fine. If publishing domain events is the last thing you do, then you should be OK most of the time. Unfortunately, in legacy applications when you start introducing domain events, you might not be always in such comfortable situation.

Async after commit

This strategy brings us closer to a reliable, good integration between our 2 databases. We use duck-typing and existing Rails features to schedule our async handlers (background jobs in Sidekiq/Redis) only once the transaction has been committed.

Notice that this is not a problem with domain events or `rails_event_store` gem. If you manually schedule background jobs to Redis within DB transactions, you also need to be careful in case of rollback (or more often, in case of the background job being executed faster than DB transaction is committed so it can't find the relevant records yet). This is a general problem of properly integrating separate databases.

```
class AfterCommitJob
  def initialize(handler, payload)
    @handler = handler
    @payload    = payload
  end

  def has_transactional_callbacks?
    true
  end

  def before_committed!
  end

  def committed!(*_, **__)
    @handler.perform_later(@payload)
  end

  def rolledback!(*_, **__)
    logger.warn("Transaction rolledback! - async handlers skipped")
  end

  def logger
    Rails.logger
  end
end
```

```
Rails.application.config.event_store.tap do |es|
  es.subscribe(
    ->(event){
      serialized_event = YAML.dump(event)
      if ApplicationRecord.connection.transaction_open?
        ApplicationRecord.connection.current_transaction.add_record(
          AfterCommitJob.new(WelcomeInSystem, serialized_event)
        )
      else
        WelcomeInSystem.perform_later(serialized_event)
      end
    }, [UserRegisteredFromEmail]
  )
end
```

Situation in which there is a crash or rollback after we published a domain event.

```
ActiveRecord::Base.transaction do
  User.create!(...)
  event_store.publish( UserRegisteredFromEmail.strict(...) )

  # ...

  raise ActiveRecord::Rollback
  # or server crash
end
```

What happens in such situation?

- user creation (SQL) is rolled back
- stored domain events are rolled back (in the default case of `rails_event_store` gem and using active record for storing those events)
- `WelcomeInSystem` and `GrantFreeCredits` handlers (which are just ActiveJob background jobs saved in redis db) **are never scheduled**.

Based on my experience I would say this is 99.9% OK. There are still cases where it can fail. Did I mention that integrating 2 databases 100% correctly is hard and that I recommend using just an SQL db? :)

This can fail if:

- the application server crashes after it committed data to the SQL DB but before it scheduled all handlers to sidekiq/redis. It's a small amount of time, but still possible.

- the SQL DB is working but Redis is unavailable. Such cases can be worked around by saving the scheduled job in the SQL DB and having a process which periodically moves such scheduled handlers jobs from the SQL DB to sidekiq/redis. Doh…

SQL MQ -> NO-SQL MQ

Ok, so what is a 100% good solution?

Besides the already mentioned solution of using only an SQL transactional DB there is one that I know about: save everything in SQL and have a separate (monitored) process move the data from SQL to external message queues such as Redis, Kafka, RabbitMQ with retries and proper idempotency.

Do you need it? It depends. Maybe 99.9% is good enough for you and you can detect and handle the 0.1% manually. The cost of going from 99.9% to 99.99% is always higher and means more complexity than going from 99% to 99.9%.

You can read more about this later in bonus chapter: Reliable notifications between two apps or microservices.

Eventual consistency

The more we try to do a single transaction the more likely it will fail, the longer it will take and the more records it can lock, leading to potential deadlocks and contention.

Big transactions can cause problems and failures even from something as simple as bugs in our code. The bigger the transaction (i.e., the longer, the more we already did), the more painful it is to roll it back and lose all the data and computation our app did. At the end we might be left with nothing, not even remembering what the customer wanted to order or whether the transaction succeeded or not. We don't want everything to fail. Often we can't even have everything fail, like the payment status because it is a crucial piece of information that we should preserve.

When we split our app into multiple aggregates we need to decide:

- the size of the aggregates, what data to keep inside and which business rules will be enforced synchronously, in DB transactions. These should be the most important rules in your app.
- what rules can be guaranteed by using eventual consistency. They will be temporarily broken, but eventually, with time, they will become consistent and protected.

Remember when I said the more important parts should not fail because of problems in less important parts of the system. This often happens when we keep adding more and more logic to a single action wrapped in a big, long DB transaction, updating tons of records. Side-effects which operate on data can also be moved to separate event handlers.

We don't want our transactions to cross multiple bounded contexts (even if that's possible because they operate on the same DB) because we want to leave an open option to decouple them later into a separate microservice if they grow in size and logic.

Let's see an example of how we can split the act of updating multiple records between many transactions and use domain events to achieve eventual consistency and synchronization.

Example

There are multiple ways to implement communication between two separate microservices in your application. Messaging is often the most recommended way of doing this. However, you probably heard that *"You must be this tall to use microservices"*[58]. In other words, your project must be a certain size and your organization must be mature enough (in terms of DevOps, monitoring, etc.) to split your app into separately running and deployed processes. **But that doesn't mean you can't benefit from decoupling individual subsystems in your application earlier**.

Those subsystems in your application are called Bounded Contexts.

Now, imagine that you have two bounded contexts in your application:

- **Season Passes [SP]** - which takes care of managing season passes for football clubs.
- **Identity & Access [I&A]** - which takes care of authentication, registrations and permissions.

And the use-case that we will be working with is described as:

When a Season Pass Holder is imported, create an account for her/him and send a welcome e-mail with password-setting instructions.

The usual way to achieve this would be to wrap everything in a transaction, create a user, create a season pass and commit it. **But we already decided to split our application into two separate parts**. And we don't want to operate on both of them directly. They have their individual responsibilities, and we want to keep them decoupled.

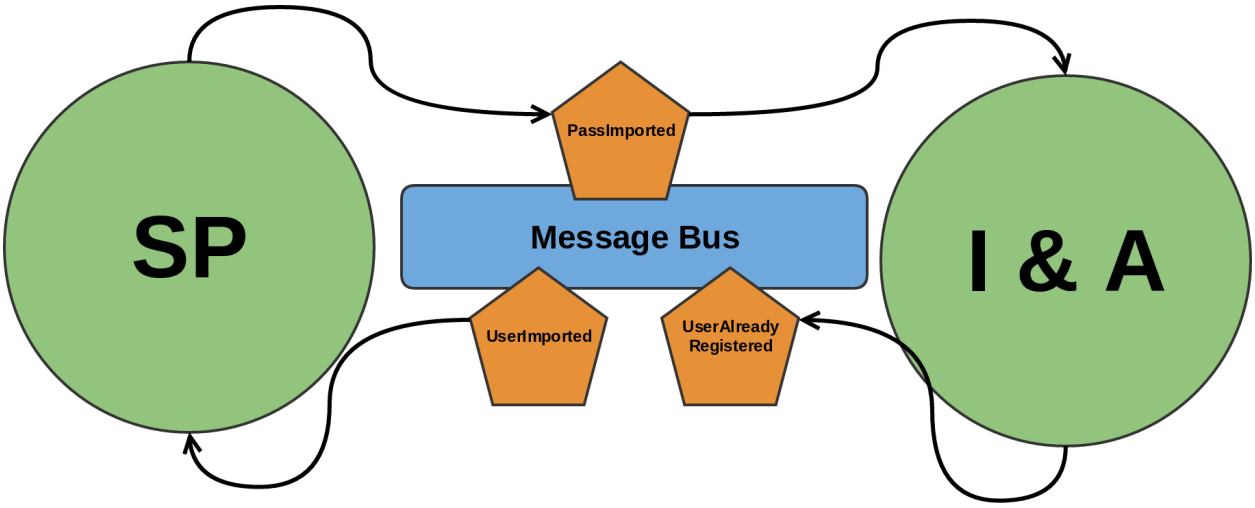
Evented way

So here is what we could do instead:

- [SP] create a season pass (without a reference to `user_id`)
- [SP] publish a domain event that season pass was imported
- [I&A] react to that domain event in the Identity & Access part of our application by creating the user account
- [I&A] in case of success, publish a domain event that User was imported (`UserImported`)

[58] http://martinfowler.com/bliki/MicroservicePrerequisites.html

  - or if the user is already present on our platform, publish `UserAlreadyRegistered` domain event.
- [SP] react to the `UserImported` or `UserAlreadyRegistered` domain event and update the `user_id` of the newly-created `SeasonPass` to the ID of the user.



It certainly sounds (and is) more complicated compared to our default solution. We should only apply this tactic where the benefits outweigh the costs.

I assume that if you decided to separate some sub-systems of your applications into **indepedent**, **decoupled units**, **you already weighed the pros and cons**. So now we are talking only about the execution.

About the cost

You might be thinking that there is a big infrastructure cost in communicating via domain events, i.e., you need to set up some message bus and think about event serialization. But the chances are **things are easier than you expect them to be**.

You can start small and straightforward, later changing to more complex solutions when the need appears. Chances are you already have all the components required for it, but you never thought of them in such a way.

Do you use Rails 5 Active Job, or Resque or Sidekiq or DelayedJob or any similar tooling, for scheduling background jobs? Good, you can **use them as the message bus for asynchronous communication between two parts of your application**. With `#retry_job`[59] you can even think of it as *at least one delivery* in case of failures.

In the beginning, the parts of your application (sub-systems, bounded-contexts) don't need to be deployed as separate applications (microservices). They don't need a separate message bus such

[59] http://api.rubyonrails.org/v5.0.0.0.1/classes/ActiveJob/Enqueuing.html#method-i-retry_job

as RabbitMQ or Apache Kafka. At the start, all you need is code which assumes **asynchronous communication** (also embracing eventual consistency) and that uses the tools you have at your disposal.

Also, you don't need any fancy serializer at the beginning such as message pack or protobuf. YAML or JSON are sufficient when you keep communicating asynchronously within the same codebase (just different parts of it).

Show me the code

Storing and publishing a domain event

We are going to use `rails_event_store`[60], but you could achieve the same results using any other pub-sub (e.g. wisper + wisper-sidekiq extension). `rails_event_store` has the advantage that your domain events will be persisted in a database.

Domain event definition. This will be published when the Season Pass is imported:

```
1  class Season::PassImported < RubyEventStore::Event
2    SCHEMA = {
3      id: Integer,
4      barcode: String,
5      first_name: String,
6      last_name: String,
7      email: String,
8    }.freeze
9
10   def self.strict(data:)
11     ClassyHash.validate_strict(data, SCHEMA)
12     new(data: data)
13   end
14 end
```

Domain event handlers/callbacks. These are how we put messages on our background queue, treating it as a simple message bus. We use the Rails 5 API here.

[60] https://github.com/RailsEventStore/rails_event_store

```
Rails.application.config.event_store.tap do |es|
  es.subscribe(->(event) do
    IdentityAndAccess::RegisterSeasonPassHolder.perform_later(YAML.dump(event))
  end, [Season::PassImported])

  es.subscribe(->(event) do
    Season::AssignUserIdToHolder.perform_later(YAML.dump(event))
  end, [IdentityAndAccess::UserImported, IdentityAndAccess::UserAlreadyRegistere\
d])
end
```

Imagine that this part of the code (somewhere) is responsible for importing season passes. It saves the pass and publishes `PassImported` event.

```
ActiveRecord::Base.transaction do
  pass = Season::Pass.create!(...)
  event_store.publish_event(Season::PassImported.strict(data: {
    id: pass.id,
    barcode: pass.barcode,
    first_name: pass.holder.first_name,
    last_name: pass.holder.last_name,
    email: pass.holder.email,
  }), stream_name: "pass$#{pass.id}")
end
```

When the `event_store` saves and publishes the `Season::PassImported` event, it will also be queued for processing by the `IdentityAndAccess::RegisterSeasonPassHolder` async handler.

Reacting to the PassImported event

These are the events that will be published by the Identity and Access bounded context:

```ruby
class IdentityAndAccess::UserImported < RubyEventStore::Event
  SCHEMA = {
    id: Integer,
    email: String,
  }.freeze

  def self.strict(data:)
    ClassyHash.validate_strict(data, SCHEMA)
    new(data: data)
  end
end

class IdentityAndAccess::UserAlreadyRegistered < RubyEventStore::Event
  SCHEMA = {
    id: Integer,
    email: String,
  }.freeze

  def self.strict(data:)
    ClassyHash.validate_strict(data, SCHEMA)
    new(data: data)
  end
end
```

This is how the Identity And Access context reacts to the fact that the Season Pass was imported.

```ruby
module IdentityAndAccess
  class RegisterSeasonPassHolder < ApplicationJob
    queue_as :default

    def perform(serialized_event)
      event = YAML.load(serialized_event)
      ActiveRecord::Base.transaction do
        user = User.create!(email: event.data.email)
        event_store.publish_event(UserImported.strict(data: {
          id: user.id,
          email: user.email,
        }), stream_name: "user$#{user.id}")
      end
    rescue User::EmailTaken => exception
```

```
15        event_store.publish_event(UserAlreadyRegistered.strict(data: {
16          id: exception.user_id,
17          email: exception.email,
18        }), stream_name: "user$#{exception.user_id}")
19      end
20    end
21  end
```

Reacting to the UserAlreadyRegistered/UserImported event

Reminder about how when an event is published we schedule something to happen in another part of the app.

```
1  Rails.application.config.event_store.tap do |es|
2    es.subscribe(->(event) do
3      IdentityAndAccess::RegisterSeasonPassHolder.perform_later(YAML.dump(event))
4    end, [Season::PassImported])
5
6    es.subscribe(->(event) do
7      Season::AssignUserIdToHolder.perform_later(YAML.dump(event))
8    end, [IdentityAndAccess::UserImported, IdentityAndAccess::UserAlreadyRegistere\
9  d])
10 end
```

The Season Pass Bounded Context reacts to either `UserImported` or `UserAlreadyRegistered` by saving the reference to `user_id`. It does not have direct access to the `User` class. It just holds a reference.

```
1  module Season
2    class AssignUserIdToHolder < ApplicationJob
3      queue_as :default
4
5      def perform(serialized_event)
6        event = YAML.load(serialized_event)
7        ActiveRecord::Base.transaction do
8          Pass.all_with_holder_email!(event.data.email).each do |pass|
9            pass.set_holder_user_id(event.data.id)
10           event_store.publish_event(Season::PassHolderUserAssigned.strict(data: {
11             pass_id: pass.id,
12             user_id: pass.holder.user_id,
```

```
13            }), stream_name: "pass$#{pass.id}")
14          end
15        end
16      end
17
18    end
19  end
```

When your needs grow

Now imagine that the needs of Identity And Access grow a lot. We would like to **extract it as a small application (microservice) and scale it separately**. Maybe deploy many more instances than the rest of our app needs? Maybe ship it with JRuby instead of MRI? Perhaps expose it to other applications that could use it for authentication and managing users as well? Can it be done?

Yes. Switch to a serious message bus that can be used between separate apps, and use a proper serialization format (not YAML, because YAML is connected to class names and you won't have identical class names between two separate apps).

Your code already assumes asynchronous communication between Season Passes and Identity And Access so you are **safe to do so**.

How is that related to business rules and eventual consistency?

The business rule that was temporally violated was that *the season pass should be connected to a user in our system*. For a brief moment it was not. We didn't know who the season pass belonged to. But once the whole cycle finished, everything was fine.

Reminder

Here's a reminder about how to add sync and async handlers (powered by ActiveJob) to your Rails apps using the `rails_event_store` gem.

```
Rails.application.config.event_store.tap do |es|
  # sync handler
  es.subscribe(
    -> (event) {
      OrderList::OrderSubmittedHandler.new.call(event)
    }, [Orders::OrderSubmitted]
  )

  # async handler
  es.subscribe(
    -> (event){
      Discounts::Saga.perform_later(YAML.dump(event))
    }, [Orders::OrderShipped]
  )
end
```

Questions

If you would like something to be clarified or you have some questions, email us at support@arkency.com . I can't promise an answer but will do my best to help. Maybe even I will write a blog post with an explanation.

Exercise

If you purchased exercises along with the book, you can now work on the *Event Handlers / Eventual Consistency* task.

Read more

Domain-Driven Design Distilled[61] has 4 Aggregate Rules of Thumb. One of them is *Update other Aggregates using eventual consistency* which is nicely described and illustrated. Also it contains tips when eventual consistency seems scary to you.

In Implementing Domain-Driven Design[62] there are nice chapters about spreading events to remote bounded contexts and about messaging infrastructure consistency. I also really enjoyed reading about what latencies between the state becomes consistent can the business often tolerate (usually much higher than what developers imagine).

[61] https://www.amazon.com/gp/product/0134434420/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0134434420&linkCode=as2&tag=arkency-20&linkId=12c564c85da17f918d275bdc51626bde

[62] https://www.amazon.com/gp/product/0321834577/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0321834577&linkCode=as2&tag=arkency-20&linkId=3155894f09101a9da242cf5cb6d9bee7

Links

- Evented Rails: Decoupling domains in Rails with Wisper pub/sub events[63]
- Reactive Messaging Patterns with the Actor Model[64]
- Microservice Prerequisites[65]

[63] http://www.g9labs.com/2016/06/23/rails-pub-slash-sub-with-wisper-and-sidekiq/

[64] https://www.amazon.com/gp/product/0133846830/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0133846830&linkCode=as2&tag=arkency-20&linkId=4baa3ace73e2b99eff9ae949347850de

[65] http://martinfowler.com/bliki/MicroservicePrerequisites.html

CQRS & Read models

Command–Query Separation (CQS)

First there was Command–Query Separation (CQS).

> Every method should either be a command that performs an action, or a query that returns data to the caller, but not both.

Notice that when it says *command* it does not mean the *command objects* that we discussed earlier. It means a conceptual distinction.

As Martin Fowler said:

- Queries: Return a result and do not change the observable state of the system (are free of side effects).
- Commands (modifiers/mutators): Change the state of a system but do not return a value.

Notice that this is just a convention, a guideline. There are tons of classes in standard libraries of many languages (including Ruby) that do not follow this principle.

Let's see an example of a class following CQS principle.

Following CQS example

```
1   class CreationsContainer
2     def initialize
3       @container = Hash.new {|hash, key| hash[key] = 0}
4     end
5
6     def increment!(class_name, value=1)
7       key = key(class_name)
8       @container[key] += value
9       nil
10    end
11
12    def [](name)
13      key = key(name)
```

```
14        @container[key]
15      end
16
17      def each(&proc)
18        @container.each(&proc)
19      end
20
21      def resource_keys
22        @container.keys
23      end
24
25      def merge(other_container)
26        for key in other_container.resource_keys
27          increment!(key, other_container[key])
28        end
29        nil
30      end
31
32      def ==(other)
33        self.class == other.class &&
34        @container == other.instance_variable_get(:@container)
35      end
36
37      def empty?
38        @container.empty?
39      end
40
41      private
42
43      def key(name)
44        name.to_sym
45      end
46    end
```

```
c = CreationsContainer.new

## COMMANDS

c.increment!("One")
# => nil

c.increment!("Two", 3)
# => nil

c.increment!("One", 1)
# => nil

## QUERIES

c["One"]
# => 2
c["Two"]
# => 3

## COMMAND
c.merge(c)
# => nil

## QUERIES

c["One"]
# => 4

c["Two"]
# => 6

c.empty?
# => false
```

It is common for Ruby methods to return `self` instead of `nil` in case of some commands. I don't think it violates the principle. It is just convenient.

```
a = []
a << 2
# => [2]
```

You might have a problem classifying some Ruby methods as commands or queries such as `map`.

```
a = [2,3]
a.map(&:to_s)
 => ["2", "3"]
```

After all, it sounds like a command, you are telling Ruby to do something, not query something. But technically it *returns a result and does not change the observable state of the system (is free of side effects)* as queries do. The internal state of the array `a` did not change. Instead we got a new array.

In contrast `map!` changes the internal state of an Array, after calling it, the array holds different objects.

```
a = [2,3]
a.map!(&:to_s)
 => ["2", "3"]
```

But for convenience it returns `self` instead of `nil` so `a` is changed, and `a` is also returned.

Breaking CQS example

```
p = Prime.new

p.next
# => 2

p.next
# => 3

p.next
# => 5
```

Almost every kind of iterator I know breaks this principe by having a method which changes internal state, and returns a result. Following it would require having two methods, which would most likely be inconvenient.

```ruby
p = Prime.new

p.next
p.current
# => 2

p.next
p.current
# => 3

p.next
p.current
# => 5
```

And there are many more methods like it in Ruby library.

```ruby
a = [:a, :b]

a.delete(:a)
# => :a

a.delete(:a)
# => nil
```

Often because they use result instead of exceptions to indicate the success or failure of requested command.

Multi-threading

Command–Query Separation (CQS) is not very useful when working with structures which need to be thread-safe and perform changes with mutexes around core logic.

```ruby
require 'thread'

queue = Queue.new

producer = Thread.new do
  5.times do |i|
    queue << i
    puts "#{i} produced"
  end
end
```

```ruby

consumer = Thread.new do
  5.times do |i|
    value = queue.pop
    puts "consumed #{value}"
  end
end
```

In this example `Queue#pop` changes the internal queue state (pops 1 element) and returns it as well. It would be much more cumbersome to follow CQS principle here.

Implementation for the reference.

```ruby
def pop(non_block=false)
  Thread.handle_interrupt(StandardError => :on_blocking) do
    @mutex.synchronize do
      while true
        if @que.empty?
          if non_block
            raise ThreadError, "queue empty"
          else
            begin
              @num_waiting += 1
              @cond.wait @mutex
            ensure
              @num_waiting -= 1
            end
          end
        else
          return @que.shift
        end
      end
    end
  end
end
```

CQS summary

Command–Query Separation (CQS) is a more low-level principe, which you can try adhere to classes. For more higher-level application architecture there is Command-Query Responsibility Segregation (CQRS) principle, which we will discuss now.

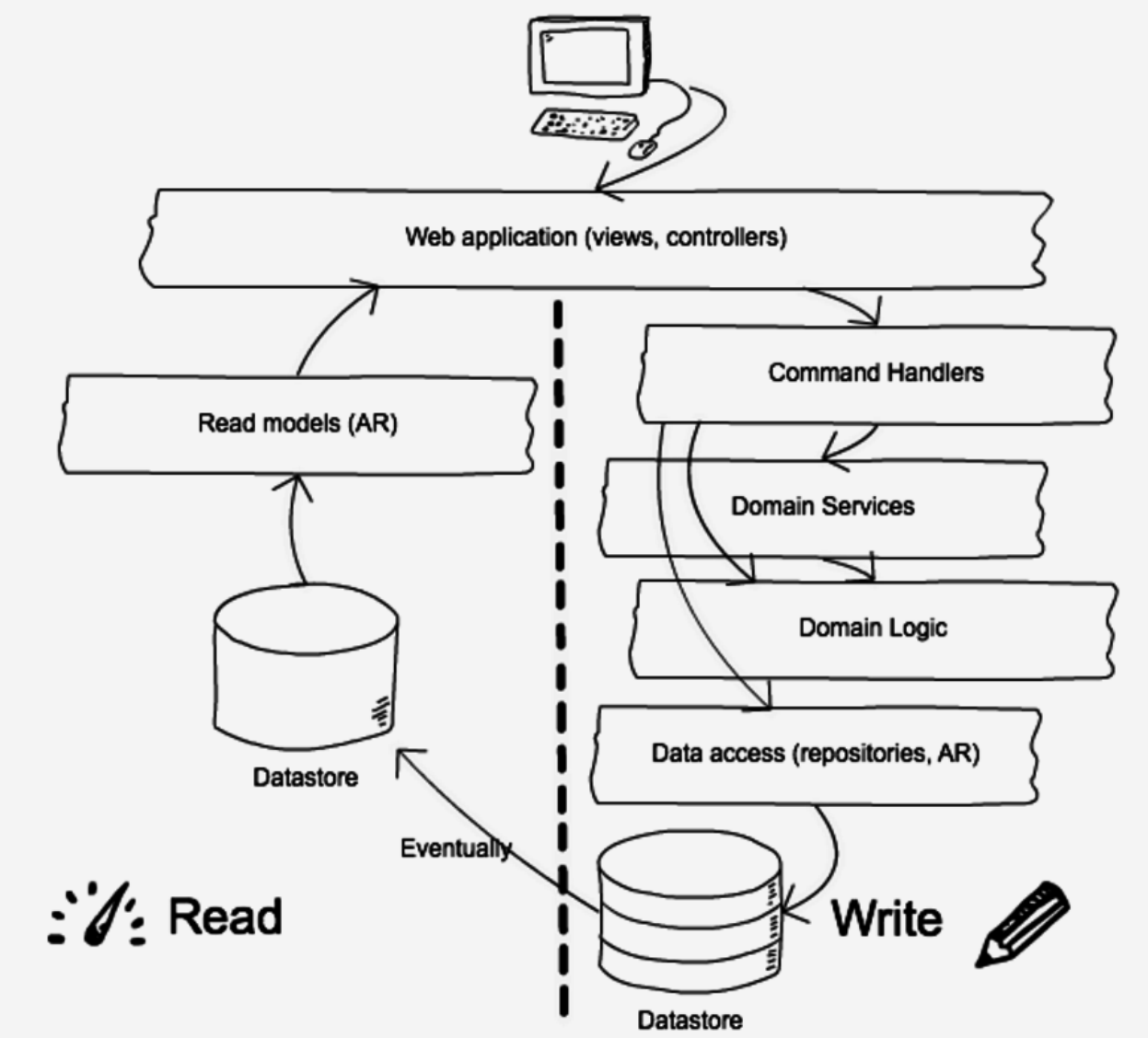
There is some symmetry between them, so I wanted to show you CQS (with its pros and cons) first. It's often easier to imagine a technique when you can compare it with something similar but from different layer/level.

Command-Query Responsibility Segregation (CQRS)

As your Rails application keeps growing there is usually increasing amount of ways for displaying same data, connecting them on various levels and presenting. There are more screens, more reports.

The usual way of implementing those requirements means more sophisticated DB queries which go through multiple tables, columns, models. It is not a problem usually, but sometimes your model needs to evolve and then all those places which read data relaying on the DB schema can prevent easy changes in the parts which write data and protect business rules.

The alternative is to follow CQRS principle for some parts of your app.



In such case you have

- Write model - optimized for protecting business rules and making decisions
- Read model - optimized for displaying to users, for reads
  - as most apps have 90% reads and 10% writes
  - as most people browse not and only sometimes buy/comment/etc

Be aware that this is highly application-dependent. There are applications and usecases out there which are very write-heavy and should follow different principles.

Pros of CQRS

- Improve read operations performance
  - Data store of read models optimized for reads
    That read-model can be ElasticSearch or InfluxDB or any store which suits best for the needs.
  - Scalable independently
    You can scale the infrastructure required for handling 9X reads differently than the ones for writes.
- Simplicity & ease of changes in read models
  - In worst case throw away & rebuild
    By iterating and re-applying historical domain events you might build your read-model from scratch instead of migrating data in DB.
- De-cluttering the domain
  - Not exposing public attributes for reads (requires event sourced aggregates)
  - Easier to tests (writes only)

Cons of CQRS

- Mindset switch

  This technique is not widely used so your developers will have to get familiar with it and learn.
- More infrastructure needed

  More databases, more devops, orchestration and monitoring is needed to handle it.
- Write operations might take longer

  In case of synchronously updating write and read model, it might take longer.

Good read model

- Read optimized
  - Denormalized data
    Usually read model will contain data from multiple tables and there will be duplication.
  - Tailor made
    Good read model is tailor made for a single purpose, usually a single screen.
- Re-createable
  - Throw away & rebuild from domain data
    Best read models are fully created from domain events, without querying other parts of the system or external APIs. It's not always easy to achieve, especially at the beginning when you don't publish enough events). It is not required, but it makes them easier to write, test and recreate from scratch based on historically saved events.
- Consider
  - Domain events
    As with every technique in this book, you can use it separately and you can use it in combination with each other. Read-models don't require publishing domain events but it can certainly be more convenient for you to build them with it.
  - Eventual consistency
    If your event handlers building domain events are asynchronous, there will be a delay between write models and read models in your app. There are ways to workaround it, by using data from commands or write-models for displaying on screen immediately after a successful change.

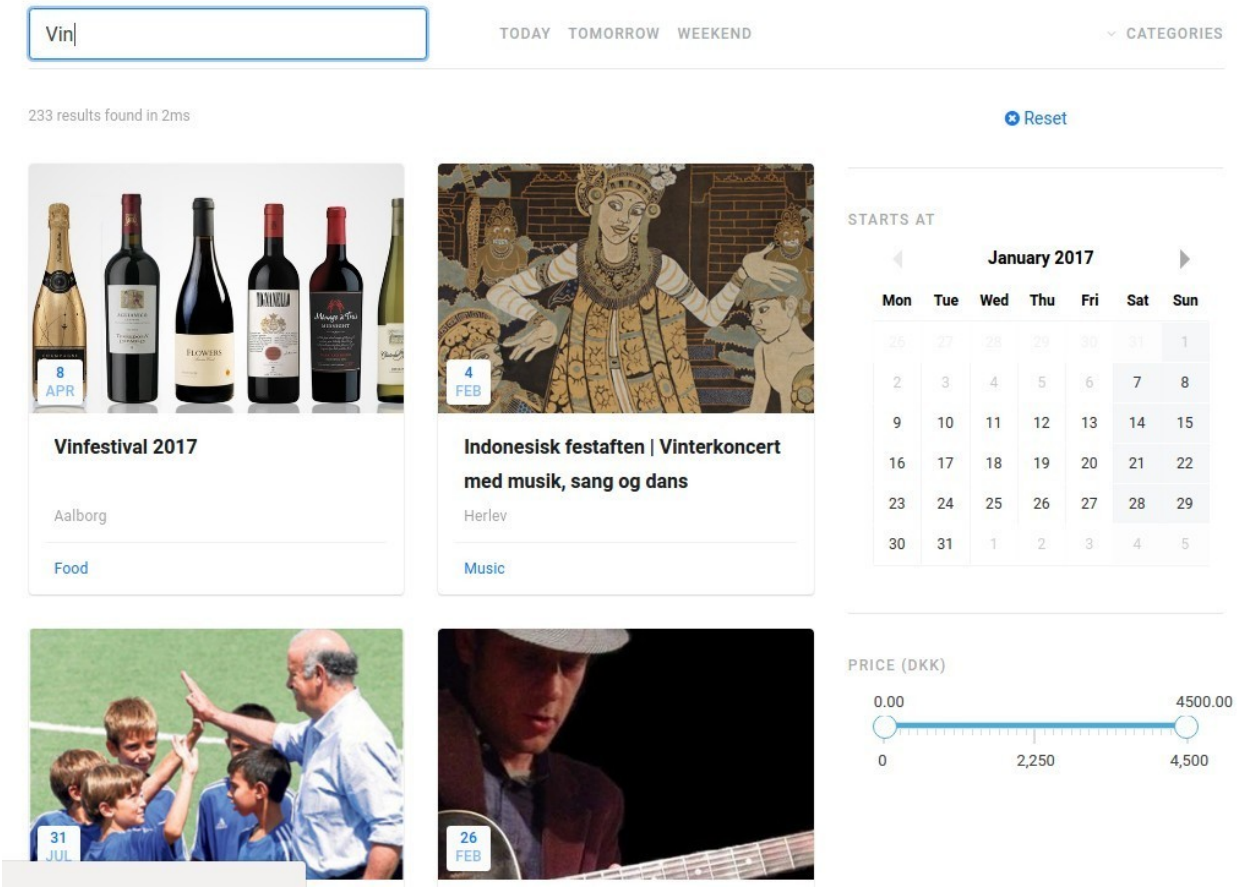
Not all about performance

I want you to know that using read-models is not all about performance. That can be just one reason you might want to go with them. I believe that in certain cases there just an easier way to deliver a feature, compared to writing complex queries spanning multiple tables and bounded-contexts.

Examples

Search

Let's have a look at a simple screen.

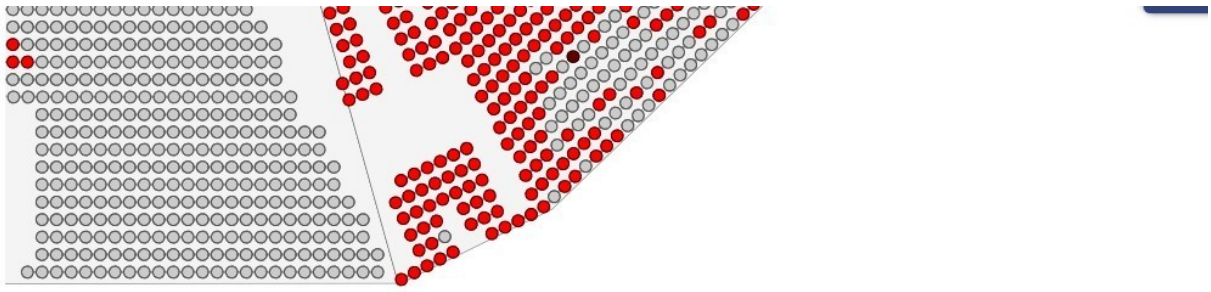


It's a screen for searching events (conferences, concerts, festivals, parties, not domain events) based on provided text, price range and dates. Events are complex beasts which multiple tickets, various prices, business rules around what is can be sold and what not, discounts, coupon codes, dynamic pricing campaigns, locations and more…

But this screen just needs to know (name, dates, picture, city, category, minimal price). It would be also super nice if it was ultra-fast with displaying search results while typing.

For this purpose it was building using Algolia, a hosted elastic-search + a collection of React.js components for building search interfaces.

Venue Overview



TYPE	LABEL / QUANTITY	NAME	EVENT	STARTS AT	CATEGORY	SEATING CATEGORY
	CARLSBERG SVINGET, TÅRN 31/32-29-21					
Season Pass	Carlsberg Svinget, Tårn 31/32-29-21	Mikkel	Sæsonkortsalg 2016/17 (Brøndby IF)	18 September 2016 23:45	FRIIS (C) - V130	Seat
Season Pass	Carlsberg Svinget, Tårn 31/32-29-21	Mikkel	Brøndby IF - Silkeborg	27 November 2016 18:00	FRIIS (C) - V130	Seat
Season Pass	Carlsberg Svinget, Tårn 31/32-29-21	Mikkel	Brøndby IF - FC Nordsjælland	5 March 2017 15:00	FRIIS (C) - V130	Seat

In one project a customer wanted to have an overview of a stadium across the whole season and all its matches. So that when a potential partner or supporter comes up and they need to find a place for dozens or hundreds people from one company to seat around during subsequent matches, they could easily find them. When there is not enough seats they can easily check if the reserved seats nearby are taken for one, upcoming match (so they could be available for a partner later) or whether they are taken by a season pass holder and reserved for all remaining matches. In any case, they can contact the customer offer them the possibility to swap seats so that they can offer the partner a large area of nearby seats next to each other.

Instead of querying many places of the system to present the situation, there is read model involved. It reacts to many domain events happening across many matches, to build this overview.

Building read models without domain events

As I mentioned, it is possible to build a read-model without publishing and reacting to domain events. All you need to do, is update the read-model right after changing the write-model.

```ruby
class CustomersService
  def register_customer(attributes)
    # write-model
    customer = Customer.register(attributes)

    # read-model
    Elasticsearch::Model.client.index({
      index: 'admin_customers',
      type:  'admin_customer',
      id: customer.id,
      body: {
        email: customer.email,
        country: customer.country.
    }})
  end
end
```

Building read models with domain events and handlers

```ruby
Rails.application.config.event_store.tap do |es|
  es.subscribe(->(ev) {
    OrderList::OrderSubmittedHandler.new.call(ev)
  }, [Orders::OrderSubmitted])
end

module OrderList
  class OrderSubmittedHandler
    def call(ev)
      Order.create!(
        number:         ev.data[:order_number],
        items_count:    ev.data[:items_count],
        net_value:      ev.data[:net_total],
        vat_amount:     ev.data[:vat_total],
        gross_value:    ev.data[:gross_total],
        state:          'submitted'
      )
    end
  end
end
```

Databases involved

Be aware that there is infinite amount of database combinations which can be used for read and write-model.

- SQL (write) + SQL (read)
- SQL (write) + ElasticSearch (read)
- SQL (write) + InfluxDB (read)
- SQL (write) + neo4j (read)
- EventStore.org (write) + SQL (read)
- etc etc…

The difference is conceptual, and does not require actually involving a separate DB, although it can be beneficial when they fit better to the usecases required.

Write model (aggregates) is the part of code which is optimized for handling changes and protecting business rules. Making sure that this change can be applied and is applied in a right way.

Read model is reacting to those changes and is not involved usually in making those decisions. It cannot reject a change. It does not decide.

Not even in DB

Frankly the read-model does not even need to be store in a database. It can be in-memory, it can be an XML file, CSV file, Excel sheet, HTML document, markdown document.

One vs Another

When you ask yourself whether you should use read model or not, you are weighting two sides. There is this vs that in almost every aspect of it.

- complex queries vs building separate model

  Do you prefer writing complex queries (usually in SQL) to extract necessary data when there is time to display it, or do you prefer building a separate model.

  Building a proper read-model can be challenging, but so does extracting everything from many places in the app.

  In my experience mistakes/bugs in queries are usually quiet and hard to notice. They don't crash the app, they will rather return an incorrect result or set of data. Whereas bugs in read-model builders cause exceptions. You are more likely to notice them when changing data, at the moment that something occurred, rather than hours or days later when a complex query is executed.

- doing more on read vs doing more on writes

  If you read certain data a lot do you prefer a complex query when it is read or complex write when something changes. Nowadays I lean towards doing more on change. I prefer to have easier reads, less complex reads. If there is complexity, let's deal with it when the complexity occurs, when there is a change in our system. At that time we usually have more data about what's going on exactly in the system. What's the context, what kind of change is that.

I think this is similar to *pull* vs *push*. Do you prefer to push changes to interested parties, or do you prefer when they pull them.

Read models vs (russian-doll) caching

Let's do a small comparison of building read models vs caching pages

Caching

- Caching is usually based on
  - time
  - and/or keys/attributes

Let's say *this is a view of a customer so let's cache it based on it's updated_at and/or other attributes*. But you don't know what is changing those attributes displayed on a page and when in your code. You rely on a convention that when they are changed the time attributes are updated as well.

However I noticed that when pages grow bigger and more complex you often don't know if you included all necessary arguments as cache key. This happens because you display data based on dozens of objects, which call methods, which load additional records via associations and so on... At the end we display data based on a tree of objects. Often different views will use slightly different objects to compute and display the necessary data. That's why it is not always easy to propagate changes up the tree of objects to keep updating `updated_at` and to expire a cache.

```
1  class Something < ActiveRecord::Base
2    has_many :others
3
4    def computed_value
5      others.map(&:this).max
6    end
7  end
```

If your view displays `something.computed_value` then maybe you are going to remember to call `something.touch` when you change any of the `others` because it is just one association. It is

not easy to be sure, and not easy to get it right. `Other` can be updated in many ways, directly (`Other.find(1).update_attributes(...)`), indirectly (`something.others.update_all(...)`). And finally, the returned value depends also on what's in the collection itself. Which records are there are which not. So deleting or re-assigning one of `Other` can cause the `computed_value` to return a different number as well.

There are certain places where caching works fantastically but *There are only two hard things in Computer Science: cache invalidation and naming things.*

Read models

Read models on the other hand are

- explicit consumers of changing events

You keep an explicit list of domain events that can trigger a read model change. You know which handlers you subscribed to which published domain events.

This of course is not the cure. You can still forget to subscribe your read model to some domain events (you didn't think they matter for what is displayed). This can especially happen when you implement a new feature, add a new domain event and don't remember that there is a view interested in that new event. That something should change there, on this view, because this new thing can now happen in your system.

The difference however is that you can easily verify it. There is a list of domain events, if the list is missing a event which should be there, you can see it.

Wisdom

- **CQRS is not a top level architecture**
- Just a way to implement part of the system
- No silver bullet
- Use when it solves more problems than it introduces

Questions

If you would like something to be clarified or you have some questions, email us at support@arkency.com . I can't promise an answer but will do my best to help. Maybe even I will write a blog-post with an explanation.

Exercise

If you purchased exercises along the book, you can now make the *Read models* task. We use published domain events to build data necessary to display a graph of daily product reservations.

Read More

Links

- Martin Fowler about CQS[66]

  > The really valuable idea in this principle is that it's extremely handy if you can clearly separate methods that change state from those that don't. This is because you can use queries in many situations with much more confidence, introducing them anywhere, changing their order. You have to be more careful with modifiers.

- Martin Fowler about CQRS[67]

  > As our needs become more sophisticated we steadily move away from that model. We may want to look at the information in a different way to the record store, perhaps collapsing multiple records into one, or forming virtual records by combining information for different places. On the update side we may find validation rules that only allow certain combinations of data to be stored, or may even infer data to be stored that's different from that we provide.

- Caching with Rails: An Overview[68]

  Introduction to caching with Rails.

[66] https://martinfowler.com/bliki/CommandQuerySeparation.html
[67] https://martinfowler.com/bliki/CQRS.html
[68] http://edgeguides.rubyonrails.org/caching_with_rails.html

Sagas / Process managers

It's nice to have Bounded Contexts which are independent. However there are processes / workflows / use cases which span around the whole app and all contexts. What can co-ordinate tasks which touch multiple parts of our system? The answer is Sagas.

I want to show them and explain their usability using examples first and code later.

Send PDF via Postal

Imagine that in our ticketing system we have we four domain events being published.

1. `PostalAddedToOrder`

   Someone added a postal addon to order. It means that they want the PDFs with their tickets for a party printed out and shipped via postal. Apparently, in XXI century, there are still customers willing to have it like that. Maybe they find such delivery convenient to use, maybe they don't want to take their phone to the party; maybe they don't have a printer. Doesn't matter. What matters is that this addon was added.
2. `PostalAddressFilledOut`

   At some point in time, they need to fill out the address where they want the tickets to be delivered.
3. `PdfGenerated`

   At some point in time, there is a PDF generated with the tickets for a party.
4. `PaymentPaid`

   From Payments Bounded Context we get an information that given payment for an order was paid.

When all of those events (related to a particular order) happen we want to trigger a command:

- ⇒ `SendPdfViaPostal`

Here is an interesting fact. The order of those events does not matter.

In our system, we considered asking customers for the address before payment, at the same time they added a postal addon but we decided against it. We were worried that filling out those address details would lower the conversion. So in the end, we decided to ask for address after successful payment when customers already invested emotionally and financially. But if we ever decide to

change it and move in a different place in the process, the code would still work. It does not matter if `PostalAddressFilledOut` occurs before or after `PaymentPaid`

Similarly, we don't know if `PdfGenerated` is going to occur before or after `PostalAddressFilledOut`. If the customer uses a previously provided address or an autocomplete feature from a browser, they can fill out the form very quickly (in seconds) which can be faster than generating PDFs where there is a small delay in processing them. On the other hand on a daily basis PDFs are generated almost instantly, and the customer can use an entire minute or more to fill out all the address details.

Knowing all that, I would like you to think how you would implement it? Using callbacks? Or cron jobs?

Payment paid too late

Imagine a scenario like this:

1. **`OrderPlaced`**
2. **`OrderExpired`**

   Because some concerts are very popular in ticketing systems, there is usually a limited amount of time during which you need to finish booking. Imagine that a customer was not quick enough (went looking for a credit card which worked and had enough money, was distracted, switched tabs in a browser, went for lunch, anything…). Anyway, our system expired the order and released reserved seats and certain quantities in an inventory.
3. **`PaymentPaid`**

   But then, sometime later, we got a notification from an external payment system that the Payment for given Order was paid.

   The order will not trigger delivering product when it is paid; it will ignore the payment, but we can't keep the customer's money.

- ⇒ **`ReleasePayment`**

  So we have to trigger a command to release the reserved amount without really charging the customer.

  From the point of *Ordering* the operation/process was unsuccessful, the Order expired. From the point of *Payments*, it was successful and paid. Our bounded contexts diverged, and we need to compensate.

Piping events to a saga

```ruby
Rails.application.config.event_store.tap do |es|
  es.subscribe(->(event) do
    Payments::CapturingSaga.perform_later(YAML.dump(event))
  end, [Orders::OrderShipped, Payments::PaymentAuthorized])
end
```

Here we've chosen to serialize the messages with YAML, but it can be anything else.

For Message Queuing this example uses Rails ActiveJob API (and its underlying adapter) but you can switch to anything else.

after_commit

As mentioned in Async after commit chapter you might want to consider scheduling side-effects (sagas included) only after a successful commit.

```ruby
Rails.application.config.event_store.tap do |es|
  es.subscribe(->(event) do
    only_after_commit do
      Payments::CapturingSaga.perform_later(YAML.dump(event))
    end
  end, [Orders::OrderShipped, Payments::PaymentAuthorized])
end
```

Initializing the state of a saga

```ruby
class CapturingSaga < ApplicationJob

  class State < ActiveRecord::Base
    self.table_name = "capturing_sagas_state"
    serialize :data

    def self.get_by_order_number(order_number)
      transaction do
        lock.find_or_create_by(order_number: order_number).tap do |s|
          s.data ||= {auth: false, shipped: false, completed: false}
          yield s
          s.save!
        end
      end
    end
```

```
16    end
17
18    def perform(serialized_event)
19      event = YAML.load(serialized_event)
20      State.get_by_order_number(event.data.fetch(:order_number)){ ... }
21    end
22
23  end
```

Because saga will be asynchronously processing domain events from many bounded contexts, which can happen at any time, there is potential for race conditions, which we want to avoid. For that, we are going to use two SQL database locking mechanisms.

- pessimistic locking[69]

  with `lock` which will use `SELECT ... FOR UPDATE` SQL statement to guarantee that only one transaction will load the state of a saga for processing a domain event.

  However, `lock` useful only when the saga's state is already in the database which means after at least one processed domain event. Otherwise `SELECT ... FOR UPDATE` returns no rows and there is nothing to lock on.

  But for that case, we use our 2nd weapon against race conditions.
- unique index in database on `order_number` column.

  In the case when the state of a saga was not yet saved (the saga was never triggered so far by any domain event) and two domain events occurring at almost the same time the 1st mechanism is not sufficient. For that, we use a unique index in the SQL database which will raise an exception if two processes try to save two different records with the state of a saga.

It is assumed here that the underlying adapter from ActiveJob will retry processing failed jobs when any of those race conditions occur.

Let's break the code down:

serialize

```
1  class State < ActiveRecord::Base
2    self.table_name = "capturing_sagas_state"
3    serialize :data
```

We are going to store the state of a saga as serialized `Hash` or `Array` in `data`. We could use separate columns in the database, but I noticed it usually does not bring too much benefit.

get_by_order_number

[69] http://api.rubyonrails.org/classes/ActiveRecord/Locking/Pessimistic.html

```
State.get_by_order_number(order_number)
```

The domain events need to contain an attribute that we are going to use to correlate them together. After all, domain events from different orders should use different sagas. In this example, we assume that it is going to be an order number.

find_or_create_by

```
transaction do
  lock.find_or_create_by(order_number: order_number).tap do |s|
  end
end
```

This is the code that is responsible for safely processing domain events one by one, in a serial, linear manner. It uses locking to prevent race conditions.

Domain events coming to saga might be in any order, but our saga must decide on some order and cannot process the facts simultaneously.

yield

```
s.data ||= {auth: false, shipped: false, completed: false}
yield s
s.save!
```

Here we initialize the defaults for the saga if it has not yet processed any domain event. Then we call (`yield`) a block which will handle implement the business logic. We will see it in a moment. After the business logic is processed, we will store the state back in DB.

Processing an event by a saga

```
def perform(serialized_event)
  event = YAML.load(serialized_event)
  State.get_by_order_number(event.data.fetch(:order_number)) do |state|
    case event
      when Orders::OrderShipped
        state.data[:shipped] = true
      when Payments::PaymentAuthorized
        state.data[:auth] = true
      else
        raise ArgumentError
```

```
11      end
12      if trigger_command?(state)
13        PaymentsService.new.call(
14          CapturePaymentCommand.new(order_number: state.order_number)
15        )
16        state.data[:completed] = true
17      end
18    end
19  end
20
21  private
22
23  def trigger_command?(state)
24    state.data[:auth] && state.data[:shipped] && !state.data[:completed]
25  end
```

There are 4 phases here:

- Deserialize the domain event.
- Get the state of a saga based on an attribute from the event.
- Process the event and update the state of a saga. Store all the data that you are going to need to trigger a command.
- If certain conditions have been met, we trigger a command and remember that. This is useful in case we might receive a duplicate of the message with domain event. It makes our saga idempotent.

triggering with command bus

Sometimes the service that you would like to call with a command can have many dependencies and might be non-trivial to instantiate. Also building it does not sound like a good responsibility for a saga. The saga should only be concerned with having a proper logic to trigger the command. In other words, we want to decouple what should be done from what knows how to do it, which service. This simplifies your handlers and sagas.

```
def perform(serialized_event)
  event = YAML.load(serialized_event)
  State.get_by_order_number(event.data.fetch(:order_number)) do |state|
    case event
      when Orders::OrderShipped
        state.data[:shipped] = true
      when Payments::PaymentAuthorized
        state.data[:auth] = true
      else
        raise ArgumentError
    end
    if trigger_command?(state)
      command_bus.call(
        CapturePaymentCommand.new(order_number: state.order_number)
      )
      state.data[:completed] = true
    end
  end
end

private

def trigger_command?(state)
  state.data[:auth] && state.data[:shipped] && !state.data[:completed]
end
```

Testing Sagas

Testing sagas is relatively simple. The events are the input, and the triggered command is the output that we expect (to happen or not to happen).

Feed saga with events

I like when a saga is only responsible for detecting the right situation to trigger a command and doing it. I enjoy testing my sagas with complete isolation from the rest of the services, aggregates and so on… I use artificial events constructed manually to feed the saga and to see how it reacts.

```
RSpec.describe Payments::Saga do
  subject(:saga) do
    saga = Payments::Saga.new
  end

  specify "capture when authorized and shipped" do
    allow(command_bus).to receive(:call)

    saga.perform(shipped(order_number: "111", customer_id: 4499))
    saga.perform(authorized(order_number: "111", transaction_identifier: "777"))

    # ...
  end

  private

  def command_bus
    Rails.configuration.command_bus
  end

  def authorized(data)
    YAML.dump(Payments::PaymentAuthorized.strict(data: data))
  end

  def shipped(data)
    YAML.dump(Orders::OrderShipped.strict(data: data))
  end
end
```

Check if command triggered with proper data

It's good to check if the triggered command is the one we expect. It will usually have attributes based on the data we received in events. Checking the class might also be very important, especially in the case when our saga can trigger different commands in different situations (as an example this saga could sometimes trigger capturing money from a transaction/payment and sometimes trigger releasing it).

```
  specify "capture when authorized and shipped" do
    allow(command_bus).to receive(:call)

    saga.perform(shipped(order_number: "111", customer_id: 4499))
    saga.perform(authorized(order_number: "111", transaction_identifier: "777"))

    expect(command_bus).to have_received(:call) do |x|
      expect(x.class).to eq(Payments::CapturePaymentCommand)
      expect(x.order_number).to eq("111")
      expect(x.transaction_identifier).to eq("777")
    end
    expect(command_bus).to have_received(:call).once
  end
```

Check different order of events

Because the domain events from various Bounded Contexts can happen at any moment, arrive with various delays and be processed in a non-deterministic order it is a good idea to ensure that every order will lead to the same deterministic result at the end. Remember that thanks to locking our saga processes the incoming domain events one by one, linearly. So the messages are processed exclusively and in some order. We just don't know which order, but that's not a problem.

Let me present what I used to be doing and how I changed my approach. The example will be based on a saga but it applies to any solution that you want to test for order independence.

```
RSpec.describe Payments::Saga do
  subject(:saga) do
    saga = Payments::Saga.new
  end

  specify do
    [
      shipped(order_number: "111", customer_id: 4499),
      authorized(order_number: "111", transaction_identifier: "777"),
    ].shuffle.each{|ev| saga.perform(ev) }
    # leanpub-start-end

    expect(command_bus).to have_received(:call) do |x|
      expect(x.class).to eq(Payments::CapturePaymentCommand)
      expect(x.order_number).to eq("111")
      expect(x.transaction_identifier).to eq("777")
    end
  end
```

This solution, however, has major drawbacks

- It does not test all possibilities
- Failures are not easily reproducible

It will eventually test all possibilities. Given enough runs on CI. And you can reproduce it if you pass the `--seed` attribute.

But generally, it does not make our job easier. And it might miss some bugs until it is executed enough times.

It was rightfully questioned by my coworker. We can do better.

We should strive to test all possible cases. It's boring to go manually through all of them. With even more possible inputs the number goes high very quickly. And it might be error prone. So let's generate all of them with the little help of `#permutation` method.

```
RSpec.describe Payments::Saga do
  subject(:saga) do
    saga = Payments::Saga.new
  end

  def self.authorized(data)
    YAML.dump(Payments::PaymentAuthorized.strict(data: data))
  end

  def self.shipped(data)
    YAML.dump(Orders::OrderShipped.strict(data: data))
  end

  [
    shipped(order_number: "111", customer_id: 4499),
    authorized(order_number: "111", transaction_identifier: "777"),
  ].permutation.each do |ev1, ev2|
# leanpub-start-end
    specify "triggers capturing when #{[ev1.class, ev2.class].to_sentence}"
      saga.perform(ev1)

      allow(command_bus).to receive(:call)
      saga.perform(ev2)

      expect(command_bus).to have_received(:call) do |x|
        expect(x.class).to eq(Payments::CapturePaymentCommand)
        expect(x.order_number).to eq("111")
```

```
28          expect(x.transaction_identifier).to eq("777")
29        end
30      end
31    end
32  end
```

Notice that we had to promote `authorized` and `shipped` methods to be class methods as they are now used outside of `specify` to define all spec, rather than inside.

Caveats:

- The more cases you generate, the faster they should run individually
- There is obviously a certain limit after which doing this does not make sense anymore. Maybe in such case fuzzy testing or moving it outside the main build is a better solution.

Check idempotency & handle events duplication

If the message queue that you use gives **at-least-once delivery** guarantees your sagas should be idempotent and handle duplicated events. To test that just duplicate at least one event and check that the command bus is still called only once.

```
1   specify "capture when authorized and shipped - idempotent" do
2     allow(command_bus).to receive(:call)
3
4     saga.perform(shipped(order_number: "111", customer_id: 4499))
5     saga.perform(authorized(order_number: "111", transaction_identifier: "777"))
6     saga.perform(shipped(order_number: "111", customer_id: 4499))
7
8     expect(command_bus).to have_received(:call) do |x|
9       expect(x.class).to eq(Payments::CapturePaymentCommand)
10      expect(x.order_number).to eq("111")
11      expect(x.transaction_identifier).to eq("777")
12    end
13    expect(command_bus).to have_received(:call).once
14  end
```

You can use a similar trick to generate all possible permutations where you have all required events to trigger a command plus one of them is additionally duplicated in a random order. Here is how I am doing it in one project.

```
[
  calculated,
  payment_succeeded,
  order_completed,
].repeated_permutation(4).select{|a| a.uniq.size == 3}.each do |f1,f2,f3,f4|
  specify "triggered once when dup #{[f1,f2,f3,f4].map(&:class).to_sentence}" do
    expect(command_bus).to receive(:call).with(RegisterPaymentFee.new(
      transaction_id: 5,
      currency: 'DKK',
      amount: -0.5.to_d
    )).once
    saga.call(f1)
    saga.call(f2)
    saga.call(f3)
    saga.call(f4)
  end
end
```

The array of events required to trigger a command has three elements:

```
  [
    calculated,
    payment_succeeded,
    order_completed,
  ]
```

We want to generate all situations where there is one of them duplicated, so there are 4 in total. Example:

```
  [
    calculated,
    payment_succeeded,
    order_completed,
    payment_succeeded,
  ]
```

We do it using `repeated_permutation(4)`. We want all combinations when there are 4 events. The three necessary ones and 1 duplicated:

```
  [
    calculated,
    payment_succeeded,
    order_completed,
  ].repeated_permutation(4)
```

But this will also generate arrays with all four events being the same, or not all necessary events being there to trigger a command. They are not interesting to us, so we can reject them. Let's select only those permutations when all three necessary, different events are there:

```
  [
    calculated,
    payment_succeeded,
    order_completed,
  ].repeated_permutation(4).select{|a| a.uniq.size == 3}
```

This should help you visualize which cases we generate:

```
[1,2,3].repeated_permutation(4).select{|a| a.uniq.size == 3}.to_a

# => [[1, 1, 2, 3], [1, 1, 3, 2], [1, 2, 1, 3], [1, 2, 2, 3],
# [1, 2, 3, 1], [1, 2, 3, 2], [1, 2, 3, 3], [1, 3, 1, 2], [1, 3, 2, 1],
# [1, 3, 2, 2], [1, 3, 2, 3], [1, 3, 3, 2], [2, 1, 1, 3], [2, 1, 2, 3],
# [2, 1, 3, 1], [2, 1, 3, 2], [2, 1, 3, 3], [2, 2, 1, 3], [2, 2, 3, 1],
# [2, 3, 1, 1], [2, 3, 1, 2], [2, 3, 1, 3], [2, 3, 2, 1], [2, 3, 3, 1],
# [3, 1, 1, 2], [3, 1, 2, 1], [3, 1, 2, 2], [3, 1, 2, 3], [3, 1, 3, 2],
# [3, 2, 1, 1], [3, 2, 1, 2], [3, 2, 1, 3], [3, 2, 2, 1], [3, 2, 3, 1],
# [3, 3, 1, 2], [3, 3, 2, 1]]
```

You might be thinking that's ridiculous to generate 36 test cases and you don't need all of that. I used to think that way as well for some time until I did it and 2 out of 36 cases were failing. If every build just sampled one random permutation it would fail in 5% of cases. Most likely for some time, you would keep rebuilding random builds on CI, thinking it is a strange random and only after some time you would spot a pattern and maybe try to reproduce with identical seed.

Check separate business processes do not intertwine

It's good to check that events about unrelated business processes don't trigger the command. For example when you have payments but for a different order.

```
  specify "no capture when separate orders authorized and shipped" do
    expect(command_bus).not_to receive(:call)

    saga.perform(shipped(order_number: "123", customer_id: 4499))
    saga.perform(authorized(order_number: "999", transaction_identifier: "777"))
  end
```

Check when saga should not be triggered

Just to be sure, check that when not all the necessary events happen, the command is not sent to the command bus.

```
  specify "no capture when only shipped" do
    expect(command_bus).not_to receive(:call)

    saga.perform(shipped(order_number: "123", customer_id: 4499))
  end
```

When you have many events, you can generate such list with `permutation` as well.

```
  [
    calculated,
    payment_succeeded,
    order_completed,
  ].permutation(2).each do |f1,f2|
    specify "not triggered when only #{[f1, f2].map(&:class).to_sentence}" do
      expect(command_bus).not_to receive(:call)
      saga.call(f1)
      saga.call(f2)
    end
  end
```

Questions

If you would like something to be clarified or you have some questions, email us at support@arkency.com . I can't promise an answer but will do my best to help. Maybe even I will write a blog-post with an explanation.

Exercise

If you purchased exercises along the book, you can now make the *Sagas* task.

Read More

Links

- original Saga pattern
  - Clemens Vasters on Sagas[70]
    Originally the Saga pattern was named to describe handling failures in long business processes split into multiple transactions. As usually with patterns in programming, there are now many things called with the same name depending on which community and in which context uses the name. This blog-post shows the typical “Rent a car, hotel, flight” scenario.
  - Clarifying the Saga pattern[71]

    > A Saga is a distribution of multiple workflows across multiple systems, each providing a path (fork) of compensating actions in the event that any of the steps in the workflow fails.
    >
    > The Process Manager pattern from the Enterprise Integration Patterns book by Gregor Hohpe is a workflow pattern and more closely resembles the implementations we are seeing in the community that are incorrectly being labeled Sagas.

- Saga - Process Manager
  - A Saga on Sagas from Microsoft Developer Network[72]
    Great article with really nice images. It contains and you should and should not use Process Manager. It also contains a nice clarification of the usage of both words in different communities.

    > The term saga is commonly used in discussions of CQRS to refer to a piece of code that coordinates and routes messages between bounded contexts and aggregates. However, for the purposes of this guidance, we prefer to use the term process manager to refer to this type of code artifact
    >
    > There is a well-known, pre-existing definition of the term saga that has a different meaning from the one generally understood in relation to CQRS.
    >
    > The term process manager is a better description of the role performed by this type of code artifact.
    >
    > Although the term saga is often used in the context of the CQRS pattern, it has a pre-existing definition. We have chosen to use the term process manager in this guidance to avoid confusion with this pre-existing definition.
    >
    > Typically, you would expect to see a process manager routing messages between aggregates within a bounded context, and you would expect to see

[70] http://vasters.com/clemensv/2012/09/01/Sagas.aspx
[71] https://web.archive.org/web/20161205130022/http://kellabyte.com/2012/05/30/clarifying-the-saga-pattern/
[72] https://msdn.microsoft.com/en-us/library/jj591569.aspx

> a saga managing a long-running business process that spans multiple bounded contexts.

  - Udi Dahan on Saga Persistence[73]
    Describes the common e-commerce example:

    > We accept orders, bill the customer, and then ship them the product.

    It was my biggest inspiration for the implementation that I showed in this chapter.

- Other
  - A Functional Foundation for CQRS/ES by Mathias Verraes[74]
    Great article showing the lifecycle of a CQRS-based app and how aggregates, read-models, sagas and commands play various roles. Contains amazing (but very simple) diagram which blew my mind, the first time.
    Check also http://verraes.net/2013/12/fighting-bottlenecks-with-cqrs/ for more great diagrams.
  - What is the difference between a saga, a process manager? - on stack overflow[75]
    Depending on the answer you choose, you can learn that:

    > They aren't mutually exclusive

    or

    > Process Manager is a state machine, Saga is not.

    My comment: `/shrug`. After reading this I stopped worrying if the name is right or not. It is important that it works and helps your project.

[73] http://udidahan.com/2009/04/20/saga-persistence-and-event-driven-architectures/
[74] http://verraes.net/2014/05/functional-foundation-for-cqrs-event-sourcing/
[75] http://stackoverflow.com/questions/15528015/what-is-the-difference-between-a-saga-a-process-manager-and-a-document-based-ap

Event sourcing

The typical application architecture only stores the current state of domain objects in a database. Throughout this book, I have shown you how you can enrich and improve the architecture. You can achieve this by publishing domain events, introducing event handlers and read-models. Yet, there are couple of problems with this **Current State + Events** architecture.

- **You have two models**

  You have a model based on events and a model representing the current state. You **save!** the current state to a database and you **publish** events.

```
1   class BasketsService
2     def initialize(event_store)
3       @event_store = event_store
4     end
5
6     def add_item(basket_id:, sku:, quantity:)
7       ActiveRecord::Base.transaction do
8         basket = Basket.lock.find(basket_id)
9         item = basket.add_item(sku, quantity)
10        basket.save!
11        @event_store.publish_event(ItemAddedToBasket.new(data: {
12          basket_id: b.id,
13          sku: item.sku,
14          quantity: item.quantity}),
15
16          stream_name: "Basket$#{basket.id}"
17        )
18      end
19    end
20  end
```

  Whenever you change one, you also need to change the other. Keeping them in sync can be tedious. In my experience, developers usually either test the changes via the current state OR via domain events. But rarely through both of them. And that sometimes leaves the room for bugs which are hard to spot. For example, when an event is missing a changed attribute. So it's not easy to guarantee that those two models (current state and events) don't fall out of sync.

Two models mean more work for you. You need to **test and maintain** the code for changing current state and the code for publishing domain events. By definition, it can't be less work than maintaining one model.

Also, I have noticed that when I had to maintain both, I still focused much more on the current state data and model. I wanted to be event-driven, I used those events to communicate between decoupled parts of my application, but... When there was a bug in my code I often fixed the current state but I rarely fixed the domain events. That would usually be more work and not always so beneficial if you don't read historical events. But that also creates a chicken-egg situation in your project. If you don't fix problematic domain events then you can't rely on them. Even when you would like to create a new read-model based on historical domain events. In other words, I noticed that they tend to be a 2nd class citizen in such a codebase. *Your mileage may vary.*

- **There is no guarantee that a change to a domain object generates an event and saves/publishes it**

  It's not so hard to imagine a scenario in which someone wants to fix data and runs:

```
1  Supplier.find_each |s|
2    c.company_code = company_code.upcase
3    c.save!
4  end
```

  ...and forgets/skips publishing a domain event. As a result, no integration, no read-model subscribed to any domain event knows about it. Thus your models start to diverge.

  This does not even need to be in a migration fixing your data. It might be a simple oversight in your application code. You have 13 possible commands which change a domain object's state. And your team forgot to add proper event publishing in the single rarest situation.

Event sourcing solves these problems. It guarantees that all changes to the domain objects are initiated by the events. Period! We are going to ditch the separate current state model completely and only save events.

How

Let's see an example implementation of an Event Sourced entity. We'll use `rails_event_store` and `aggregate_root` gems. I've looked at a few implementations of Event Sourcing in Ruby, Java + Akka, Haskell and they are all very similar. I don't want you to focus too much on the tooling. But rather on the concepts which you can use in any language and any framework.

Here is the general algorithm for changing the state of an entity in an event-sourced solution.

1. Read old events

2. Apply historical events to change state
3. Call a method/command
    - Verify invariants
    - Generate and apply new events to change state
4. Persist new events
    - Optionally publish to MQ

Let's see how it looks in an example.

Read old events from database

Remember that when we say database it does not imply a SQL database. Event sourcing can be implemented basically on top of any DB. `rails_event_store` by default uses the SQL DB but there is a MongoDB driver as well.

```
1 product_id = "21aa8e11-e2ea-4082-9ce6-39ff294c5e57"
2 stream_name = "Product$#{product_id}"
3 events = event_store.read_stream_events_forward(stream_name)
```

In event sourced solutions your events are usually partitioned in *stream*. A stream is nothing more but a named collection of events. We don't usually want to read the whole story of our application. But rather the parts of that story which relate to our domain object, such as a given Product.

Usually, the stream name is the UUID of an entity or a concatenation of an object's class name and its ID.

```
1 Product$1
2 Product-123
3 21aa8e11-e2ea-4082-9ce6-39ff294c5e57
4 products-21aa8e11-e2ea-4082-9ce6-39ff294c5e57
```

The convention that you choose does not matter much, but it can be quite cumbersome to change it later.

So let's say we got back such events:

```
events = [
  ProductRegistered.new(data: {
    store_id: 3,
    sku: "Apple",
  }),
  ProductSupplied.new(data: {
    store_id: 3,
    sku: "Apple",
    quantity: 4,
  }),
]
```

Apply historical events to change state

Now we are going to use those historical domain events to re-create the current state of the entity.

```
product = Product.new
events.each do |event|
  product.apply(event)
end
version = events.size - 1
```

As you can see, we started with a new product with a clean state (or a default state). And we apply all the historical events one by one from oldest to newest. What does this `apply` do? How does it work? Very simply:

```
class Product
  def apply(event)
    case event
    when ProductRegistered
      registered(event)
    when ProductSupplied
      supplied(event)
    else
      raise "Missing handler for #{event}"
    end
  end

  def register(store_id:, sku:)
    apply(pr = ProductRegistered.new(data: {
      store_id: store_id,
      sku: sku,
```

```
    }))
    return [pr]
  end

  def supply(quantity)
    apply(ps = ProductSupplied.new(data: {
      store_id: @store_id,
      sku: @sku,
      quantity: quantity,
    }))
    return [ps]
  end

  private

  def registered(event)
    @store_id = event.data.fetch(:store_id)
    @sku = event.data.fetch(:sku)
    @quantity_available = 0
    @quantity_reserved = 0
    @quantity_shipped = 0
  end

  def supplied(event)
    @quantity_available += event.data.fetch(:quantity)
  end
end
```

`Product` uses event's class to determine to which private method should it delegate the logic of updating its internal state.

The first event

```
  ProductRegistered.new(data: {
    store_id: 3,
    sku: "Apple",
  }),
```

will be passed to the `registered` method:

```
  def registered(event)
    @store_id = event.data.fetch(:store_id)
    @sku = event.data.fetch(:sku)
    @quantity_available = 0
    @quantity_reserved  = 0
    @quantity_shipped   = 0
  end
```

as a result the `Product` updates a bunch of instance variables.

For the 2nd event

```
  ProductSupplied.new(data: {
    store_id: 3,
    sku: "Apple",
    quantity: 4,
  }),
```

we call `supplied` and pass the event as well.

```
  def supplied(event)
    @quantity_available += event.data.fetch(:quantity)
  end
```

As you can see, that increases the amount of the available quantity of the Product. It happens because the product was supplied to our magazine.

Usually, the mapping between an event and how to update the internal state can be based only on event's class (or event's name). That's what I've presented. In some more complicated cases, you might use the event's version, or its metadata, or its data if that's what you need.

Now that we have applied our historical events, we re-created the current state of the Product.

```
#<Product:0x0000000362ae50
  @store_id=3,
  @sku="Apple",
  @quantity_available=4,
  @quantity_reserved=0,
  @quantity_shipped=0
>
```

We can try to update its state.

Call a method/command

```
unpublished_events = product.reserve(quantity: 3, order_number: "K/123/10/2017")
```

```
class Product
  QuantityNotAvailable = Class.new(StandardError)

  def reserve(quantity:, order_number:)
    unless @quantity_available >= quantity
      raise QuantityNotAvailable
    end
    apply(pr = ProductReserved.new(data: {
      store_id: @store_id,
      sku: @sku,
      quantity: quantity,
      order_number: order_number,
    }))
    return [pr]
  end

  def apply(event)
    case event
    when ProductRegistered
      registered(event)
    when ProductSupplied
      supplied(event)
    when ProductReserved
      reserved(event)
    else
      raise "Missing handler for #{event}"
    end
  end

  # ...

  private

  def reserved(event)
    quantity = event.data.fetch(:quantity)
    @quantity_available -= quantity
    @quantity_reserved  += quantity
  end
end
```

This might look unusual so let's see what's going on.

We call a domain method and get back a list of domain events generated by the entity. They are not yet published.

```
unpublished_events = product.reserve(quantity: 3, order_number: "K/123/10/2017")
```

How does it work under the hood?

First, the `Product#reserve` method checks business constraints (the rules that it protects). In this case, we want to make sure we can't reserve more items of a product than there are available.

```
class Product
  QuantityNotAvailable = Class.new(StandardError)

  def reserve(quantity:, order_number:)
    unless @quantity_available >= quantity
      raise QuantityNotAvailable
    end
    # ...
  end
end
```

In such a case, we raise an exception. Sometimes, instead of raising an exception, we generate a domain event about failure. It might not change the object's internal state.

If everything is OK, we change the object's state by applying a newly created event (or many events).

```
  def reserve(quantity:, order_number:)
    unless @quantity_available >= quantity
      raise QuantityNotAvailable
    end
    apply(pr = ProductReserved.new(data: {
      store_id: @store_id,
      sku: @sku,
      quantity: quantity,
      order_number: order_number,
    }))
    return [pr]
  end
```

The fact that we use `apply` to change the internal state guarantees that the next time we recreate the state using historical domain events we are going to use the very same logic. This is what prevents accidental mistakes and reduces code duplication.

```
  def apply(event)
    case event
    when ProductRegistered
      registered(event)
    when ProductSupplied
      supplied(event)
    when ProductReserved
      reserved(event)
    else
      raise "Missing handler for #{event}"
    end
  end

  private

  def reserved(event)
    quantity = event.data.fetch(:quantity)
    @quantity_available -= quantity
    @quantity_reserved  += quantity
  end
```

It may feel a bit unnatural at first and it requires some time to get used to. It is important to use the same logic to change the object's state when you are changing it for the first time and when later you try to recreate the current state by applying historical events. That's why, instead of changing instance variables directly in `reserve`, we call `apply` with new domain events.

Persist new events

Those events that we generated are not yet stored or published. So far, they only exist in memory. Now is the moment for publishing them.

```
event_store.publish_events(
  unpublished_events,
  stream_name: stream_name,
  expected_version: version
)

# optionally, usually not needed
version += unpublished_events.size
unpublished_events = []
```

I will elaborate a bit more about `expected_version` in a separate sub-chapter.

Full code

Let's see the whole code without any interruption from me this time.

Domain events:

```
class ProductRegistered < RubyEventStore::Event
end

class ProductSupplied < RubyEventStore::Event
end

class ProductReserved < RubyEventStore::Event
end
```

`Product` model which allows to supply and reserve quantities of it.

```
class Product
  QuantityNotAvailable = Class.new(StandardError)

  def register(store_id:, sku:)
    apply(pr = ProductRegistered.new(data: {
      store_id: store_id,
      sku: sku,
    }))
    return [pr]
  end

  def supply(quantity)
    apply(ps = ProductSupplied.new(data: {
      store_id: @store_id,
      sku: @sku,
      quantity: quantity,
    }))
    return [ps]
  end

  def reserve(quantity:, order_number:)
    unless @quantity_available >= quantity
      raise QuantityNotAvailable
    end
    apply(pr = ProductReserved.new(data: {
      store_id: @store_id,
```

```
27        sku: @sku,
28        quantity: quantity,
29        order_number: order_number,
30      }))
31      return [pr]
32    end
33
34    def apply(event)
35      case event
36      when ProductRegistered
37        registered(event)
38      when ProductSupplied
39        supplied(event)
40      when ProductReserved
41        reserved(event)
42      else
43        raise "Missing handler for #{event}"
44      end
45    end
46
47    private
48
49    def registered(event)
50      @store_id = event.data.fetch(:store_id)
51      @sku = event.data.fetch(:sku)
52      @quantity_available = 0
53      @quantity_reserved  = 0
54      @quantity_shipped   = 0
55    end
56
57    def supplied(event)
58      @quantity_available += event.data.fetch(:quantity)
59    end
60
61    def reserved(event)
62      quantity = event.data.fetch(:quantity)
63      @quantity_available -= quantity
64      @quantity_reserved  += quantity
65    end
66  end
```

Service:

```ruby
def reserve_product(product_id, quantity, order_number)
  stream_name = "Product$#{product_id}"
  events = event_store.read_stream_events_forward(stream_name)
  product = Product.new
  events.each do |event|
    product.apply(event)
  end
  version = events.size - 1

  unpublished_events = product.reserve(
    quantity: quantity,
    order_number: order_number
  )

  event_store.publish_events(
    unpublished_events,
    stream_name: stream_name,
    expected_version: version
  )
end
```

Small discussion

Some developers disagree with the practice of changing the aggregate's state before persisting the events. As you can see in the code below, calling `reserve` changes the internal state of the aggregate. Change happens by applying `ProductReserved` via `reserved` method.

```ruby
class Product
  QuantityNotAvailable = Class.new(StandardError)

  def reserve(quantity:, order_number:)
    unless @quantity_available >= quantity
      raise QuantityNotAvailable
    end
    apply(pr = ProductReserved.new(data: {
      store_id: @store_id,
      sku: @sku,
      quantity: quantity,
      order_number: order_number,
    }))
    return [pr]
  end
```

```
16
17    private
18
19    # ... other methods
20
21    def reserved(event)
22      quantity = event.data.fetch(:quantity)
23      @quantity_available -= quantity
24      @quantity_reserved  += quantity
25    end
26  end
```

In a service there is moment in time before we persist the generated events.

```
1   # ...
2   unpublished_events = product.reserve(
3     quantity: quantity,
4     order_number: order_number
5   )
6
7   # product's internal state changed
8
9   event_store.publish_events(
10    unpublished_events,
11    stream_name: stream_name,
12    expected_version: version
13  )
14  # only here we are sure that persisting unpublished_events worked.
```

I don't see a problem in applying changes from unpublished events to an in-memory representation of the aggregate. I don't see a problem with that because I know those events are going to be persisted in the very next step.

If persisting does not work there will be an exception and nobody is going to use the aggregate (`product`) for anything anyway. Even if persisting works correctly, developers are not supposed to be using this object for anything afterward, because they should be updating only 1 aggregate at the same time.

AggregateRoot

Remembering the `version` of an aggregate can be tedious. Especially how many events constituted it at the moment we recreated its state. And the `unpublished_events` generated by the `reserve` command. And then passing all that back to `published_events`. That's why we created the `aggregate_root` gem. It simplifies it a bit and offers more friendly API.

```
product = Product.new
product.load("Product$#{product_id}")
product.reserve(
  quantity: quantity,
  order_number: order_number
)
product.store
```

It keeps the name of the loaded stream, unpublished events, and a version inside the model. This means you have fewer data to pass around and get a more ActiveRecord-like feeling. It comes for the price of a bit more coupling with the persistence layer. As a result, you also don't need to return those unpublished events (but you still can, for testing).

```
class Product
  include AggregateRoot

  def register(store_id:, sku:)
    apply(ProductRegistered.new(data: {
      store_id: store_id,
      sku: sku,
    }))
  end

  def supply(quantity)
    apply(ProductSupplied.new(data: {
      store_id: @store_id,
      sku: @sku,
      quantity: quantity,
    }))
  end

  def reserve(quantity:, order_number:)
    unless @quantity_available >= quantity
      raise QuantityNotAvailable
    end
    apply(ProductReserved.new(data: {
      store_id: @store_id,
      sku: @sku,
      quantity: quantity,
      order_number: order_number,
    }))
  end
```

```
30
31    private
32
33    def apply_strategy
34      ->(_me, event) do
35        case event
36        when ProductRegistered
37          registered(event)
38        when ProductSupplied
39          supplied(event)
40        when ProductReserved
41          reserved(event)
42        else
43          raise "Missing handler for #{event}"
44        end
45      end
46    end
47
48    def registered(event)
49      @store_id = event.data.fetch(:store_id)
50      @sku = event.data.fetch(:sku)
51      @quantity_available = 0
52      @quantity_reserved  = 0
53      @quantity_shipped   = 0
54    end
55
56    def supplied(event)
57      @quantity_available += event.data.fetch(:quantity)
58    end
59
60    def reserved(event)
61      quantity = event.data.fetch(:quantity)
62      @quantity_available -= quantity
63      @quantity_reserved  += quantity
64    end
65  end
```

The implementation of AggregateRoot is very small so I can even include it here.

```ruby
module AggregateRoot
  def apply(*events)
    events.each do |event|
      apply_strategy.(self, event)
      unpublished_events << event
    end
  end

  def load(stream_name, event_store: default_event_store)
    @loaded_from_stream_name = stream_name
    events = event_store.read_stream_events_forward(stream_name)
    events.each do |event|
      apply(event)
    end
    @version = events.size - 1
    @unpublished_events = []
    self
  end

  def store(stream_name = loaded_from_stream_name, event_store: default_event_st\
ore)
    event_store.publish_events(
      unpublished_events,
      stream_name: stream_name,
      expected_version: version
    )
    @version += unpublished_events.size
    @unpublished_events = []
  end

  private
  attr_reader :loaded_from_stream_name

  def unpublished_events
    @unpublished_events ||= []
  end

  def version
    @version ||= -1
  end

  def apply_strategy
```

```
43      DefaultApplyStrategy.new
44    end
45
46    def default_event_store
47      AggregateRoot.configuration.default_event_store
48    end
49  end
```

Eventide

The same logic in the Eventide[76] framework would look very similar.

```
1   category :product
2
3   handle Reserve do |reserve|
4     product_id = reserve.product_id
5
6     product, version = store.fetch(product_id, include: :version)
7     position = reserve.metadata.global_position
8
9     stream_name = stream_name(product_id)
10    unless product.available?(reserve.quantity)
11      reservation_rejected = ReservationRejected.follow(reserve)
12      reservation_rejected.time = clock.iso8601
13      reservation_rejected.transaction_position = position
14      write.(reservation_rejected, stream_name)
15      return
16    end
17
18    reserved = Reserved.follow(reserve)
19    reserved.processed_time = clock.iso8601
20    reserved.transaction_position = position
21    write.(reserved, stream_name, expected_version: version)
22  end
```

That's because the algorithm in every framework is very similar:

1. Read old events from the database from a stream
2. Apply historical events to a fresh object to change it's state
3. Perform business logic

[76] https://eventide-project.org/

- Verify invariants
- Generate new events to change state

4. Persist new events in the database

Handling race conditions on writes in Event Sourcing

When you update an entity in any framework it's possible that someone else updated it in the meantime. The 3 best-known strategies for handling the problem are:

1. **Ignore it**

   That's the usual strategy. I'm judging from the code I've seen around when reviewing legacy Rails app. Either the developers don't know there is a potential issue, or they don't have time or the skills to fix it. It may be that the cost of potential issues is so low that we don't bother handling the situation. Usually, that means that last write wins and we could lose some conflicting changes.

   Ignoring race conditions for event sourced entities is not a viable strategy. It can lead to violating the business rules that we were supposed to protect. In the case of `Product` it would be reserving a quantity which is no longer available. Also, with event sourcing things cannot happen (to the same entity) at exactly the same time. We need to have a deterministic order in which events occurred for our entity. And that order is not determined by time. But rather by the monotonically growing position of events in a stream.

2. **Pessimistic lock**

   Another solution is to use a pessimistic locking. Wrap the whole read-change-write operation in a lock. This prevents anyone else from doing simultaneous changes. For that, often the mechanism that the underlying DB provides is used. In SQL that can be as simple as using `SELECT LOCK FOR UPDATE` or advisory locks available in Postgres or Mysql.

   If your database supports it, you can use such a solution.

```
1  # Mysql
2  product_id = "a3531171-0d57-4239-a248-9fe5080e3f3c"
3  unless ActiveRecord::Base.connection.execute(
4    "SELECT GET_LOCK('#{product_id}', 10);"
5  ).each.to_a == [[1]]
6    raise "Lock failed"
7  end
8  # Logic goes here...
9  ActiveRecord::Base.connection.execute("SELECT RELEASE_LOCK('#{product_id}');")
```

```ruby
# Postgres
product_id = "a3531171-0d57-4239-a248-9fe5080e3f3c"
ActiveRecord::Base.transaction do
  ActiveRecord::Base.connection.execute(
    "SELECT pg_advisory_xact_lock(#{product_id.hash})"
  )
  # logic goes here
end
```

However, pessimistic locking is not the approach, I would recommend here.

3. **Optimistic lock**

   An optimistic lock is a different approach. It relies on the database's ability to atomically update whole rows/documents. The application remembers the version of an entity that was read from the database. And this version is passed back when saving changes. If there was a race condition and a different thread/processes updated the same entity in the meantime, the DB knows about it and. It will reject the application changes. Such optimistic locking strategies can work even with databases which don't have transactions but have UPSERT functionality.

   Bear in mind that optimistic locking may (deeply under the hood) use pessimistic locking anyway. You might not see it because a `rails_event_store` adapter for your DB does the job for you. Or it might be even deeper, implemented in the code of a database engine that needs to handle multiple simultaneous connections. However, from the perspective of an application, it does not matter.

   If a conflict is detected and your thread/process lost, it will get an exception.

```ruby
begin
  product = Product.new
  product.load("Product$#{product_id}")
  product.reserve(
    quantity: quantity,
    order_number: order_number
  )
  product.store
rescue RailsEventStore::WrongExpectedEventVersion
  # someone changed the product in the meantime
  # between we did `load` and `store`
end
```

What can you do when such conflict occurs?

1. **Propagate this error up to UX layer and display to the user**

In such case, the user can figure out what changed in the meantime and repeat or discard the action.

2. **Retry the command**

   For that you would need to:

   - recreate the current state based on historical events again. Including those events stored a moment ago by a conflicting process/thread
   - invoke a command and verify invariants again
   - generate new domain events
   - again try to save them at later positions in the stream

   In its simplest form, that would look like:

```
1  begin
2    product = Product.new
3    product.load("Product$#{product_id}")
4    product.reserve(
5      quantity: quantity,
6      order_number: order_number
7    )
8    product.store
9  rescue RailsEventStore::WrongExpectedEventVersion
10   product = Product.new
11   product.load("Product$#{product_id}")
12   product.reserve(
13     quantity: quantity,
14     order_number: order_number
15   )
16   product.store
17 end
```

   But in a high-throughput scenario that can fail again, so you might want to try doing it up to N times.

   There is, however, a potential problem if calling the command is not idempotent. That can happen when the object collaborates with an external API (via adapter usually).

```
        product = Product.new
        product.load("Product$#{product_id}")
        product.reserve(
          quantity: quantity,
          order_number: order_number,
          api_adapter: ExternalInventoryApiAdater.new
        )
        product.store
```

Triggering the command again could make another HTTP request. You might want to avoid that. In such a case you might prefer the next solution.

3. **Detect non-conflicting changes**

   Imagine that our `Product` in inventory was concurrently changed in 2 places. One tried to reserve a certain amount of the product and another tried to supply. Verifying business invariants for reservation succeeded. But when we tried to store events we got a race condition error. Obviously, if there was enough quantity to reserve before supplying the product, there should be also enough of it after supplying. In other words, those two changes are not conflicting.

   Yet, if we had two reservations happening at the same time we cannot ensure with ease that they are not conflicting. There could be 1 unit of the product available. Both reservations could try to reserve exactly 1 unit. Invariants verification in memory can succeed. But because the changes occur concurrently at the same time and they are conflicting, only one reservation will be saved. The second reservation cannot because it would lead to a negative amount of product in stock. That's what our system is supposed to prevent.

   As you can see, there are concurrent changes that we can allow to occur and there are those that we cannot allow.

   Here is an example of how you could try to detect if `unpublished_events` generated by one process conflict with `published_events` which succeeded in being published in the meantime by another process.

```
module AutoConflictResolution
  def store(
    stream_name = loaded_from_stream_name,
    event_store: default_event_store
  )
    super
  rescue RailsEventStore::WrongExpectedEventVersion
    unpublished_events = unpublished_events()
    tried_version = version()
    self.load(stream_name, event_store: event_store)
    published_events = loaded_events[tried_version+1..version()]
```

```
12        if not_conflicting(unpublished_events, published_events)
13          @unpublished_events = unpublished_events
14          store(stream_name: stream_name, event_store: event_store)
15        end
16      end
17    end
18
19    class Product
20      include AggregateRoot
21      include AutoConflictResolution
22
23      private
24
25      NOT_CONFLICTING_PAIRS = Set.new([
26        [ProductSupplied, ProductReserved],
27        [ProductReserved, ProductSupplied],
28        [ProductSupplied, ProductSupplied],
29      ])
30
31      def not_conflicting(unpublished_events, published_events)
32        unpublished_events.flat_map do |ev1|
33          published_events.flat_map do |ev2|
34            NOT_CONFLICTING_PAIRS.include?([ev1.class, ev2.class])
35          end
36        end.all?
37      end
38
39    end
```

Retrying the command (when possible) and verifying all invariants again seems to me like a simpler and safer approach. I can imagine subtle bugs occurring due to allowed conflicting changes which should be excluded. Make sure to test heavily.

BTW, Compare optimistic locking in aggregates with domain events vs ActiveRecord's optimistic locking[77] in which you don't easily know the intention and scope of conflicting changes:

[77] http://api.rubyonrails.org/classes/ActiveRecord/Locking/Optimistic.html

```
p1 = Person.find(1)
p2 = Person.find(1)

p1.first_name = "Michael"
p1.save!

p2.last_name = "Dope"
p2.save! # Raises an ActiveRecord::StaleObjectError
```

To figure out if both changes can be concurrently allowed you would need to manually diff both conflicting objects and compare attributes. That might be easy and possible for `Person` but much harder for complex `Product` or `Order` objects.

Testing Event Sourced aggregates

Testing event sourced aggregates is all about checking whether we generate or publish the right kind of events. I am going to present a solution using `rspec` and `rails_event_store-rspec` gems. They will helps us achieve it using a bunch of helpful matchers. But the general formula applies no matter what toolset you use.

Test events generated in-memory by an object

The first and most simple way of testing your aggregates is by doing it completely in-memory. Then there's no need to load or save events into any DB.

Let's try to test our `Product` and its business rules. Here is the code we will be testing:

```
require 'ruby_event_store'
require 'aggregate_root'

class ProductRegistered < RubyEventStore::Event
end

class ProductSupplied < RubyEventStore::Event
end

class ProductReserved < RubyEventStore::Event
end

class Product
  include AggregateRoot

```

```
16    QuantityNotAvailable = Class.new(StandardError)
17    NotRegistered = Class.new(StandardError)
18    AlreadyRegistered = Class.new(StandardError)
19
20    def register(store_id:, sku:)
21      raise AlreadyRegistered if @store_id
22      apply(ProductRegistered.new(data: {
23        store_id: store_id,
24        sku: sku,
25      }))
26    end
27
28    def supply(quantity)
29      raise NotRegistered unless @store_id
30      apply(ProductSupplied.new(data: {
31        store_id: @store_id,
32        sku: @sku,
33        quantity: quantity,
34      }))
35    end
36
37    def reserve(quantity:, order_number:)
38      raise NotRegistered unless @store_id
39      unless @quantity_available >= quantity
40        raise QuantityNotAvailable
41      end
42      apply(ProductReserved.new(data: {
43        store_id: @store_id,
44        sku: @sku,
45        quantity: quantity,
46        order_number: order_number,
47      }))
48    end
49
50    private
51
52    def apply_strategy
53      ->(_me, event) do
54        case event
55        when ProductRegistered
56          registered(event)
57        when ProductSupplied
```

```
58          supplied(event)
59        when ProductReserved
60          reserved(event)
61        else
62          raise "Missing handler for #{event}"
63        end
64      end
65    end
66
67    def registered(event)
68      @store_id = event.data.fetch(:store_id)
69      @sku = event.data.fetch(:sku)
70      @quantity_available = 0
71      @quantity_reserved  = 0
72      @quantity_shipped   = 0
73    end
74
75    def supplied(event)
76      @quantity_available += event.data.fetch(:quantity)
77    end
78
79    def reserved(event)
80      quantity = event.data.fetch(:quantity)
81      @quantity_available -= quantity
82      @quantity_reserved  += quantity
83    end
84  end
```

The lifecycle of a product is that it is registered first, then supplied, and then reserved usually. So let's test those methods in that order.

```
1   require 'rails_event_store/rspec'
2
3   RSpec.describe Product do
4     subject(:product) { Product.new }
5
6     specify "product registered" do
7       # execution
8       events = product.register(store_id: 1, sku: "BOOK")
9
10      # validation
11      expect(events).to match([
```

```
12        an_event(ProductRegistered).with_data(
13          store_id: 1,
14          sku: "BOOK",
15        ).strict
16      ])
17    end
18  end
```

When testing ActiveRecord models we usually care about what they return. Very often we also check what's been stored in a database.

When it comes to event sourced objects we care about:

- whether they can protect the business rules (in our case the rule you can reserve only as much as available)
- the domain events they generated

Registration generates a very simple event. We check it by using the matchers provided via our gem.

```
an_event(ProductRegistered).with_data(
  store_id: 1,
  sku: "BOOK",
).strict
```

The `strict` modifier makes sure that the event `data` does not contain other keys, but only those mentioned in `with_data`.

An event like this:

```
ProductRegistered.new(data: {
  store_id: store_id,
  sku: sku,
  something: "more",
})
```

would fail this check because it contains an unexpected `something: "more"` part.

Of course, we can't register the same product twice so there is:

```
specify "product can be registered only once" do
  # setup
  product.register(store_id: 1, sku: "BOOK")

  expect do
    product.register(store_id: 1, sku: "BOOK")   # execution
  end.to raise_error(Product::AlreadyRegistered) # validation
end
```

Let's go into supplying a product.

```
specify "product supplied" do
  product.register(store_id: 1, sku: "BOOK") # setup

  events = product.supply(30)                # execution
  expect(events).to match([                  # validation
    an_event(ProductSupplied).with_data(
      store_id: 1,
      sku: "BOOK",
      quantity: 30,
    ).strict
  ])
end
```

There are two schools of building a test state for an event-sourced object:

- using events to reach a state
- using commands to reach a state

I don't like using events to setup test state. It's not easy to guarantee that you would be applying correct events in a correct order to reach a state that could possibly occur in your system. But such a test could look like this:

```
  specify "product supplied" do
    # setup
    product.apply([
      ProductRegistered.new(data: {
        store_id: store_id,
        sku: sku,
        something: "more",
    })])

    # execution
    events = product.supply(30)

    # validation
    expect(events).to match([
      an_event(ProductSupplied).with_data(
        store_id: 1,
        sku: "BOOK",
        quantity: 30,
      ).strict
    ])
  end
```

There are different ways to make sure that the events used to build the current entity state are proper. One would be to use the same events that are in other tests used in a validation step. But that still leaves the possibility of an incorrect order. Or missing events and building states that could not occur in a real-life situation. That's why I prefer building the state with commands. If an invalid state is reached, the command is going to raise an exception. This will prevent you from continuing.

So here we use `register` to build an initial state. Then we call `supply` and we verify that the generated events contain all that's needed.

```
  specify "product supplied" do
    product.register(store_id: 1, sku: "BOOK") # setup

    events = product.supply(30)                # execution
    expect(events).to match([                  # validation
      an_event(ProductSupplied).with_data(
        store_id: 1,
        sku: "BOOK",
        quantity: 30,
      ).strict
    ])
  end
```

As I mentioned in the Domain Events chapter they should usually contain

- IDs (`sku` here)
- tenant ids (`store_id`)
- command data (`quantity`)

And here are two very similar tests. They check whether we can or cannot reserve a certain quantity of a product.

```
1   specify "reserve available quantity" do
2     product.register(store_id: 1, sku: "BOOK")
3     product.supply(10)
4     events = product.reserve(quantity: 10, order_number: "2018/10/KBVAU")
5
6     expect(events).to match([
7       an_event(ProductReserved).with_data(
8         store_id: 1,
9         sku: "BOOK",
10        quantity: 10,
11        order_number: "2018/10/KBVAU",
12      ).strict
13    ])
14  end
15
16  specify "reserve unavailable quantity" do
17    product.register(store_id: 1, sku: "BOOK")
18    product.supply(10)
19    product.reserve(quantity: 5, order_number: "2018/10/KBVAU")
20
21    expect do
22      product.reserve(quantity: 6, order_number: "2033/06/ABCNO")
23    end.to raise_error(Product::QuantityNotAvailable)
24  end
```

Test publishing on DB/stream via Application Service

The equivalent of checking if a database was updated by an ActiveRecord model would be to check if domain events were persisted. This is usually done in a service, so let's test one.

```
class ProductService
  def supply(store_id:, sku:, quantity:)
    product = Product.new
    product.load("Product-#{store_id}-#{sku}")
    product.supply(quantity)
    product.store
  end

  def register(store_id:, sku:)
    product = Product.new
    product.register(store_id: store_id, sku: sku)
    product.store("Product-#{store_id}-#{sku}")
  end
end
```

```
require 'rails_event_store/rspec'

RSpec.describe ProductService do
  specify "supply publishes an event" do
    service = ProductService.new
    service.register(store_id:1, sku: "BOOK")
    service.supply(store_id:1, sku: "BOOK", quantity: 30)

    expect(event_store).to have_published(
      an_event(ProductSupplied).with_data(
        store_id: 1,
        sku: "BOOK",
        quantity: 30,
      ).strict
    ).in_stream("Product-1-BOOK")
  end

  private

  def event_store
    RailsEventStore.event_repository
  end
end
```

`have_published(...).in_stream("Product-1-BOOK")` is another matcher provided by the `rails_event_store-rspec` gem that will read the last events from the DB and verify if they include the ones specified.

Event Sourcing with EventStore DB (eventstore.org)

Using SQL-based EventStore can be beneficial to your project. Especially a legacy one where introducing more databases is often not really feasible. Yet, using an SQL-based DB for storing events has some downsides and limitations as well. It's not that easy to use such SQL-based EventStore as a Queue (the subject of the next chapter). It's not easy to design the DB schema and code in a way that would exclude all possible race conditions and not affect the application's performance. Especially when the library tries to support multiple use cases like storing events for technical logs, storing events for event sourcing and multiple access patterns such as publishing events within a SQL DB transaction and without a transaction.

For greenfield applications which want to use the Event Sourcing pattern I would recommend considering EventStore DB (https://eventstore.org/). It's sometimes called *Greg Young's Event Store* (*GES* in short) to distinguish from other solutions with *Event Store* in the name.

EventStore has a native HTTP interface based on the AtomPub protocol which is plenty fast enough for the majority of use cases. For high-performance use, there are native drivers for .NET, Akka and Erlang.

It's designed as a transactionless append-only store with Event Sourcing in mind.

There are, however, only a few gems for it in Ruby world and none of them seem to be very actively maintained. In the following code, I will use the `http_event_store` gem simply because I am most familiar with it.

Let's start with our `Product` class from an Inventory domain, that we've already seen in this book. The logic remains basically unchanged and storage agnostic.

Domain logic and events

```
1  class DomainEvent
2    def initialize(data, event_id = SecureRandom.uuid)
3      @data = data
4      @event_id = event_id
5    end
6
7    attr_reader :data, :event_id
8  end
9
10 class ProductRegistered < DomainEvent
11 end
12
13 class ProductSupplied < DomainEvent
14 end
15
```

```
16  class ProductReserved < DomainEvent
17  end
18
19  class Product
20    QuantityNotAvailable = Class.new(StandardError)
21
22    def register(store_id:, sku:)
23      apply(pr = ProductRegistered.new({
24        store_id: store_id,
25        sku: sku,
26      }))
27      return [pr]
28    end
29
30    def supply(quantity)
31      apply(ps = ProductSupplied.new({
32        store_id: @store_id,
33        sku: @sku,
34        quantity: quantity,
35      }))
36      return [ps]
37    end
38
39    def reserve(quantity:, order_number:)
40      unless @quantity_available >= quantity
41        raise QuantityNotAvailable
42      end
43      apply(pr = ProductReserved.new({
44        store_id: @store_id,
45        sku: @sku,
46        quantity: quantity,
47        order_number: order_number,
48      }))
49      return [pr]
50    end
51
52    def apply(event)
53      {
54        ProductRegistered => method(:registered),
55        ProductSupplied   => method(:supplied),
56        ProductReserved   => method(:reserved),
57      }.fetch(event.class).call(event)
```

```
  end

  private

  def registered(event)
    @store_id = event.data.fetch(:store_id)
    @sku = event.data.fetch(:sku)
    @quantity_available = 0
    @quantity_reserved  = 0
    @quantity_shipped   = 0
  end

  def supplied(event)
    @quantity_available += event.data.fetch(:quantity)
  end

  def reserved(event)
    quantity = event.data.fetch(:quantity)
    @quantity_available -= quantity
    @quantity_reserved  += quantity
  end
end
```

There are two underlying operations that we need to implement event sourcing with GES.

Restoring object's state based on historical events

```
require 'http_event_store'
require 'active_support/core_ext/string'
require 'active_support/core_ext/hash'

def client
  @client ||= HttpEventStore::Connection.new
end

def restore_product(stream_name)
  events  = client.read_all_events_forward(stream_name).map do |ev|
    ev.type.constantize.new(ev.data.symbolize_keys, ev.event_id)
  end
  product = Product.new
  events.each do |event|
    product.apply(event)
```

```
16    end
17    version = events.size - 1
18    return product, version
19  end
```

This method reads all historical events for a stream of events related to one Product. As we've seen with previous solutions, we create a fresh `Product` instance. Then we apply all the events to establish the current state.

`read_all_events_forward` returns data structures. We manually use the event's types to map to instances of a proper class with the help of `constantize`. This is not required at all. You could write `Product`'s code to return simple structures and `apply`. Method `apply` to operate based on an attribute such as `event_type` instead of based on `event.class`. Up to you.

Appending new events

To append new events to a product's stream we map them to a simple structure expected by the gem being used. `version` is the position in the stream that we used to construct current state. It's set after reading events in `restore_product`. GES uses it to detect concurrency conflicts. It's done in a fashion like that presented in the previous sub-chapter.

```
1  def update_product(stream_name, unpublished_events, version)
2    client.append_to_stream(stream_name, unpublished_events.map do |ev|
3      {
4        event_type: ev.class.name,
5        data:       ev.data,
6        event_id:   ev.event_id,
7      }
8    end, version)
9  end
```

Registering a product

```
def register_product(store_id, sku)
  stream_name = "Product-#{store_id}-#{sku}"
  product = Product.new
  unpublished_events = product.register(
    store_id: store_id,
    sku: sku,
  )

  update_product(stream_name, unpublished_events, -1)
end
```

Here we pass `-1` as the version because we expect that the product does not exist yet. Notice that the stream name includes `store_id` (which is a tenant Id) and `sku`. `sku` is unique only in the scope of a particular store.

Supplying / Reserving a product

```
def reserve_product(store_id, sku, quantity, order_number)
  stream_name      = "Product-#{store_id}-#{sku}"
  product, version = restore_product(stream_name)

  unpublished_events = product.reserve(
    quantity: quantity,
    order_number: order_number
  )

  update_product(stream_name, unpublished_events, version)
end

def supply_product(store_id, sku, quantity)
  stream_name      = "Product-#{store_id}-#{sku}"
  product, version = restore_product(stream_name)

  unpublished_events = product.supply(quantity)

  update_product(stream_name, unpublished_events, version)
end
```

Both operations look almost identical. And even though we switched from SQL-based RailsEventStore to EventStore DB (https://eventstore.org), the general pattern is equivalent. Also, the code is almost the same.

And you can use it like that:

```
store_id = 1
sku = "WORKSHOP"

register_product(store_id, sku)
supply_product(store_id, sku, 30)
reserve_product(store_id, sku, 22, "2017/10/X895N")

reserve_product(store_id, sku, 9, "2017/10/K0123") # exception, not enough quant\
ity
```

Event Store as a Queue

One of the interesting side-effects of using Event Stores is that we can use them as a Queue for exchanging data between systems/micro-services/bounded contexts.

Usually, when we want to integrate two systems we add a stateful message broker such as RabbitMQ. The application server processes some changes, updates a local database and sends events to the broker. This can lead to some bugs or rollbacks when there are connectivity problems with the MQ. The DB and MQ are two separate systems and that always leads to potential troubles. Anyway, the general idea is that the application server sends events to the MQ. Then they are dispatched asynchronously to clients. Clients acknowledge processed messages and the MQ remembers that.

Event Stores allow a different integration pattern. In this pattern the client remembers its position in the stream of data it reads and keeps asking for more. As a result, the Event Store does not need to send the events or remember which were send to whom. The consuming client remembers its own progress and position in a stream of events.

Problems with using SQL-based EventStores as a Queue

Bear in mind that using SQL-based EventStores as a Queue can lead to subtle problems. They might be hard to detect. We've looked into a few of such SQL-based stores and, basically, all of them had some race condition issues. I am still researching how to solve this problem for Rails Event Store and SQL adapter[78].

The problem comes from the fact that using SQL-based EventStores as a Queue requires monotonically increasing positions of domain events in a table/stream. Generally you want the consumer to get events 1-10, process them, save local progress, get events 11-20, process them, save local progress etc etc.

But, in SQL-based solutions it is possible that some events are not yet committed. It happens because of a longer transaction or they were rolled back. As a result, the client could get events: 1,2,3,4,7,8,9,10.

[78] https://github.com/RailsEventStore/rails_event_store/issues/106

If the consumer ignores such gaps and processes the events and remembers that it is at position 10, then it will never get events 5,6. That can lead to inconsistent state between systems.

There are some workarounds to this problem but they have their own pitfalls. One solution that is usually mentioned is to linearize ALL events and the whole system. That means using advisory locks or SERIALIZABLE transaction levels so as to always have only 1 application thread appending new events at any given moment. Yet introducing such a change in legacy applications with lots of traffic might be close to impossible.

The other solution requires detecting a gap during a read. Then figuring out whether it was due to a rollback (and we can safely skip some numbers) or due to as-yet-uncommitted transactions (so we might want to wait). We might also try to fill in the gaps in the stream in case of rollback with *fake events*.

The problem is complex and has many sides and as I mentioned I keep researching it[79] and discussing with other devs.

Doing it in EventStore DB

The EventStore DB (eventstore.org) does not have this problem. It does not have the concept of a transaction. But you are guaranteed that events you sent together will be saved or rejected atomically. And it can safely and quickly linearize all events, even if they are sent at the same time. EventStore DB will decide whether they happened in order P1,P2,P3,N1,N2 or N1,N2,P1,P2,P3 when you send them from processes N & P.

EventStore DB (eventstore.org, GES) also allows for Persistent Subscriptions and competing consumers, where the subscription state is stored in the EventStore DB and it acts as MQ. But we are not going to discuss that now. We will see how the EventStore can be used as a Queue.

HTTP Polling

Let's see how we can use EventStore DB (eventstore.org, GES) as a Queue by implementing a Catch-Up Subscriber. Our process is going to iterate over a stream of all (`$all`) events stored in GES and build a read model. Based on inventory product reservations and releases we are going to build a SQL-based read model. The read model is supposed to store how much of given product was reserved per day. This will be helpful in graphs that we display for users of our inventory system.

The part responsible for writing events is going to be identical as we've seen a moment ago. We have a `Product` creating domain events.

[79] https://github.com/RailsEventStore/rails_event_store/issues/106

```
class Product
  QuantityNotAvailable = Class.new(StandardError)

  def register(store_id:, sku:)
    apply(pr = ProductRegistered.new({
      store_id: store_id,
      sku: sku,
    }))
    return [pr]
  end

  def supply(quantity)
    apply(ps = ProductSupplied.new({
      store_id: @store_id,
      sku: @sku,
      quantity: quantity,
    }))
    return [ps]
  end

  def reserve(quantity:, order_number:)
    unless @quantity_available >= quantity
      raise QuantityNotAvailable
    end
    apply(pr = ProductReserved.new({
      store_id: @store_id,
      sku: @sku,
      quantity: quantity,
      order_number: order_number,
    }))
    return [pr]
  end

  def apply(event)
    {
      ProductRegistered => method(:registered),
      ProductSupplied   => method(:supplied),
      ProductReserved   => method(:reserved),
    }.fetch(event.class).call(event)
  end

  private
```

```
43
44    def registered(event)
45      @store_id = event.data.fetch(:store_id)
46      @sku = event.data.fetch(:sku)
47      @quantity_available = 0
48      @quantity_reserved  = 0
49      @quantity_shipped   = 0
50    end
51
52    def supplied(event)
53      @quantity_available += event.data.fetch(:quantity)
54    end
55
56    def reserved(event)
57      quantity = event.data.fetch(:quantity)
58      @quantity_available -= quantity
59      @quantity_reserved  += quantity
60    end
61  end
```

And services responsible for storing those events in GES.

```
1   require 'http_event_store'
2   require 'active_support/core_ext/string'
3   require 'active_support/core_ext/hash'
4
5   def reserve_product(store_id, sku, quantity, order_number)
6     stream_name      = "Product-#{store_id}-#{sku}"
7     product, version = restore_product(stream_name)
8
9     unpublished_events = product.reserve(
10      quantity: quantity,
11      order_number: order_number
12    )
13
14    update_product(stream_name, unpublished_events, version)
15  end
16
17  private
18
19  def restore_product(stream_name)
20    events  = client.read_all_events_forward(stream_name).map do |ev|
```

```
    ev.type.constantize.new(ev.data.symbolize_keys, ev.event_id)
  end
  product = Product.new
  events.each do |event|
    product.apply(event)
  end
  version = events.size - 1
  return product, version
end

def update_product(stream_name, unpublished_events, version)
  client.append_to_stream(stream_name, unpublished_events.map do |ev|
    {
      event_type: ev.class.name,
      data:       ev.data,
      event_id:   ev.event_id,
    }
  end, version)
end
```

There is a separate process that will constantly poll GES for all events, stored by all products.

Here is a DB schema for our read model. In the `inventory_daily_reservations` table we are going to store records for how much of a product was reserved in given day. In `catch_up_progress` we are going to store our progress.

```
require 'open-uri'
require 'net/http'
require 'json'
require 'active_record'

ENV['DATABASE_URL'] ||= "sqlite3::memory:"

ActiveRecord::Base.establish_connection(ENV['DATABASE_URL'])
ActiveRecord::Schema.define do
  self.verbose = true
  create_table :catch_up_progress do |t|
    t.string  "name",              null: false
    t.string  "page",              null: false
  end
  add_index :catch_up_progress,
    :name,
    unique: true
```

```
18
19    create_table :inventory_daily_reservations do |t|
20      t.integer "store_id",           null: false
21      t.string  "sku",                null: false
22      t.date    "date",               null: false
23      t.integer "total_reserved",     null: false, default: 0
24    end
25    add_index :inventory_daily_reservations,
26      [:store_id, :sku, :date],
27      unique: true,
28      name: :index_inventory_daily_reservations
29  end
30
31  class CatchUpSubscriptionProgress < ActiveRecord::Base
32    self.table_name = "catch_up_progress"
33  end
34
35  module Inventory
36    class DailyReservation < ActiveRecord::Base
37      self.table_name = "inventory_daily_reservations"
38    end
39  end
```

Let's see how the process would work if we had to do it manually.

We start by querying a special stream called `$all`. As we've seen previously, events regarding different Aggregate roots are stored in different streams such as

- `"Product-1-WORKSHOP"`
- `"Product-1-BOOK"`
- `"Product-2-BURGER"`

and we use those events from such streams to reproduce the current state of those 3 products.

But in GES you can use `$all` to iterate over events from all streams.

```
(master) > curl -i 'http://127.0.0.1:2113/streams/$all' -u admin:changeit
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Access-Control-Expose-Headers: Location, ES-Position, ES-CurrentVersion
Cache-Control: max-age=0, no-cache, must-revalidate
Vary: Accept
ETag: "28456548;1415427280"
Content-Type: application/vnd.eventstore.streamdesc+json; charset=utf-8
Server: Mono-HTTPAPI/1.0
Date: Tue, 12 Sep 2017 14:16:16 GMT
Content-Length: 12236
Keep-Alive: timeout=15,max=100

{
  "title": "All events",
  "id": "http://127.0.0.1:2113/streams/%24all",
  "updated": "2017-09-12T14:16:05.783672Z",
  "author": {
    "name": "EventStore"
  },
  "headOfStream": false,
  "links": [
    {
      "uri": "http://127.0.0.1:2113/streams/%24all",
      "relation": "self"
    },
    {
      "uri": "http://127.0.0.1:2113/streams/%24all/head/backward/20",
      "relation": "first"
    },
    {
      "uri": "http://127.0.0.1:2113/streams/%24all/0000000000000000000000000000\
000/forward/20",
      "relation": "last"
    },
    {
      "uri": "http://127.0.0.1:2113/streams/%24all/0000000001ADF7580000000001ADF\
758/backward/20",
      "relation": "next"
    },
    {
      "uri": "http://127.0.0.1:2113/streams/%24all/0000000001B26FB90000000001B26\
```

```
43 FB9/forward/20",
44       "relation": "previous"
45     },
46     {
47       "uri": "http://127.0.0.1:2113/streams/%24all/metadata",
48       "relation": "metadata"
49     }
50   ],
51   ...
```

At `http://127.0.0.1:2113/streams/$all` we can see that the `last` page is under

```
1 http://127.0.0.1:2113/streams/%24all/0000000000000000000000000000000000/forward/20
```

What we are going to do is start from the `last` page and iterate by following `previous` pages until we reach the newest pages.

```
1     {
2       "uri": "http://127.0.0.1:2113/streams/%24all/0000000000000000000000000000000000\
3 000/forward/20",
4       "relation": "last"
5     },
```

If that `last`/`previous` nomenclature sounds counter-intuitive for you, don't worry. It's just as strange for me and many other developers as well. What's important is that we iterate from the oldest to the newest events by following `pages` of them.

Our next step would be to check out the events in

```
1 http://127.0.0.1:2113/streams/%24all/0000000000000000000000000000000000/forward/20
```

That looks like this:

```
curl -i 'http://127.0.0.1:2113/streams/%24all/0000000000000000000000000000000/f\
orward/3?embed=body' -u admin:changeit

HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Access-Control-Expose-Headers: Location, ES-Position, ES-CurrentVersion
Cache-Control: max-age=31536000, private
Vary: Accept
Content-Type: application/vnd.eventstore.streamdesc+json; charset=utf-8
Server: Mono-HTTPAPI/1.0
Date: Tue, 12 Sep 2017 14:24:08 GMT
Content-Length: 3405
Keep-Alive: timeout=15,max=100

{
  "title": "All events",
  "id": "http://127.0.0.1:2113/streams/%24all",
  "updated": "2017-09-11T10:10:00.120984Z",
  "author": {
    "name": "EventStore"
  },
  "headOfStream": false,
  "links": [
    {
      "uri": "http://127.0.0.1:2113/streams/%24all",
      "relation": "self"
    },
    {
      "uri": "http://127.0.0.1:2113/streams/%24all/head/backward/3",
      "relation": "first"
    },
    {
      "uri": "http://127.0.0.1:2113/streams/%24all/0000000000000282000000000000\
000/forward/3",
      "relation": "previous"
    },
    {
      "uri": "http://127.0.0.1:2113/streams/%24all/metadata",
      "relation": "metadata"
    }
  ],
  "entries": [
```

```
    {
      "eventId": "a9cb9d39-526e-4f8e-a904-1b51116c657b",
      "eventType": "$metadata",
      "eventNumber": 0,
      "data": "{\"$acl\":{\"$w\":\"$admins\",\"$d\":\"$admins\",\"$mw\":\"$admin\
s\"}}",
      "streamId": "$$$user-admin",
      "isJson": true,
      "isMetaData": false,
      "isLinkMetaData": false,
      "positionEventNumber": 0,
      "positionStreamId": "$$$user-admin",
      "title": "0@$$$user-admin",
      "id": "http://127.0.0.1:2113/streams/%24%24%24user-admin/0",
      "updated": "2017-09-11T10:10:00.120984Z",
      "author": {
        "name": "EventStore"
      },
      "summary": "$metadata",
      "links": [
        {
          "uri": "http://127.0.0.1:2113/streams/%24%24%24user-admin/0",
          "relation": "edit"
        },
        {
          "uri": "http://127.0.0.1:2113/streams/%24%24%24user-admin/0",
          "relation": "alternate"
        }
      ]
    },
    {
      "eventId": "65648759-878a-4427-a6b9-caca0181f82d",
      "eventType": "$metadata",
      "eventNumber": 0,
      "data": "{\"$acl\":{\"$w\":\"$admins\",\"$d\":\"$admins\",\"$mw\":\"$admin\
s\"}}",
      "streamId": "$$$user-ops",
      "isJson": true,
      "isMetaData": false,
      "isLinkMetaData": false,
      "positionEventNumber": 0,
      "positionStreamId": "$$$user-ops",
```

```
      "title": "0@$$$user-ops",
      "id": "http://127.0.0.1:2113/streams/%24%24%24user-ops/0",
      "updated": "2017-09-11T10:10:00.120923Z",
      "author": {
        "name": "EventStore"
      },
      "summary": "$metadata",
      "links": [
        {
          "uri": "http://127.0.0.1:2113/streams/%24%24%24user-ops/0",
          "relation": "edit"
        },
        {
          "uri": "http://127.0.0.1:2113/streams/%24%24%24user-ops/0",
          "relation": "alternate"
        }
      ]
    }
    ...
  ]
}
```

There are two interesting parts here.

The URL of a `previous` page. That's the next link that we are going to follow.

```
"links": [
  {
    "uri": "http://127.0.0.1:2113/streams/%24all/0000000000000282000000000000000\
0/forward/3",
    "relation": "previous"
  },
```

And `entries` which contain the events and their `data`.

```
"entries": [
  {
    "eventId": "a9cb9d39-526e-4f8e-a904-1b51116c657b",
    "eventType": "$metadata",
    "eventNumber": 0,
    "data": "{\"$acl\":{\"$w\":\"$admins\",\"$d\":\"$admins\",\"$mw\":\"$admins\\
"}}",
    "streamId": "$$$user-admin",
```

Event `data` can be any binary or textual format so parsing it is on our side. With GES, you can use any serialization format you want. For simplicity, we will work with JSON.

Some of those events are going to be internal events coming from GES but we can skip them. We recognize them based on `$` sign at the beginning of their name.

And that's how it works.

You start from the `last` page, get 20 events (you can use more if you change the URL accordingly) along with the URL of the next page to query for the next 20 events.

What happens if you eventually catch up with all events?

The page with events will contain more than 1 event but at most the max amount you asked for (20 by default). When you follow the `previous` URL, the previous page is going to be empty. Normally, you should continue fetching it until at least 1 event appears. However GES implements HTTP Long Polling and you can provide "ES-LongPoll" HTTP header specifying how long you are willing to wait for a response, a timeout. If any new event is written during that time, you will receive a new page as soon as possible. Otherwise you are going to get the same empty page again. What it means is that once your process catches up you don't need to call `sleep` by yourself. You can continue just doing HTTP requests and the HTTP library you use will wait. Because you are served an HTTP response with new events as soon as they appear there is no artificial delay here. It would occur if you added `sleep` in your code. The sleep happens as a side effect of the networking layer and HTTP Polling.

Here is a very simple `FeedReader` that infinitely yields all events you stored and the next page you should check for new events.

```
class FeedReader
  def run(page)
    loop do
      data = open(
        page+"?embed=body",
        {
          http_basic_authentication: ["admin", "changeit"],
          "Accept" => 'application/json',
          "ES-LongPoll" => "35",
```

```
        }
      ).read
      json = JSON.parse(data)
      link = json['links'].find{|l| l['relation'] == 'previous' }
      next if json['entries'].empty?
      entries = json['entries'].reject do |entry|
        entry['eventType'].start_with?("$")
      end.reverse
      page = link.fetch('uri')
      yield entries, page
    end
  end
end
```

We are going to use this reader in a moment. Now let's see how we handle those events.

```
module Inventory
  class ReadEvent < Struct.new(:id, :type, :datetime, :stream, :data)
  end

  class DailyReservationsCatchUp
    def run(feed_reader)
      progress = CatchUpSubscriptionProgress.find_or_initialize_by(
        name: "DailyReservationsCatchUp"
      )
      progress.page ||= "http://localhost:2113/streams/%24all/0000000000000000000\
00000000000000/forward/20"
      feed_reader.run(progress.page) do |entries, page|
        progress.page = page
        batch(entries, progress)
      end
    end

    def batch(entries, progress)
      ActiveRecord::Base.transaction do
        entries.each do |entry|
          perform(ReadEvent.new(
            entry['eventId'],
            entry['eventType'],
            Time.parse(entry['updated']),
            entry['streamId'],
            JSON.load(entry['data']),
```

```
          ))
        end
        progress.save!
      end
    end

    def perform(event)
      value = case event.type
        when 'ProductReserved'
          event.data.fetch('quantity')
        when 'ProductReleased'
          -event.data.fetch('quantity')
        else
          return
      end
      daily = DailyReservation.lock.find_or_create_by!(
        sku:      event.data.fetch('sku'),
        store_id: event.data.fetch('store_id'),
        date:     event.datetime.to_date,
      )
      DailyReservation.update_counters(daily.id, total_reserved: value)
    end
  end
end
```

To run the whole process together call:

```
Inventory::DailyReservationsCatchUp.new.run(FeedReader.new)
```

How does it work?

```
def run(feed_reader)
  progress = CatchUpSubscriptionProgress.find_or_initialize_by(
    name: "DailyReservationsCatchUp"
  )
  progress.page ||= "http://localhost:2113/streams/%24all/0000000000000000000000\
0000000000/forward/20"
  feed_reader.run(progress.page) do |entries, page|
    progress.page = page
    batch(entries, progress)
  end
end
```

The `run` method starts by reading where we finished previously. It's determined from the URL of the next unprocessed page and provides that as a starting point to `feed_reader`. `FeedReader` will download the page, parse it, and provide us with events it contained. Also we'll get the `URL` of next `previous` page URL we should visit.

Then we call `batch(entries, progress)` and try to process all the events we received.

```
def batch(entries, progress)
  ActiveRecord::Base.transaction do
    entries.each do |entry|
      perform(ReadEvent.new(
        entry['eventId'],
        entry['eventType'],
        Time.parse(entry['updated']),
        entry['streamId'],
        JSON.load(entry['data']),
      ))
    end
    progress.save!
  end
end
```

We try to transactionally process all the events and store our progress. That means if there is an exception, crash or another problem, we will discard our work. Also we won't store the next page URL. That means when the process starts again it will resume work where it finished. And for every domain event we update our read model.

```
def perform(event)
  value = case event.type
    when 'ProductReserved'
      event.data.fetch('quantity')
    when 'ProductReleased'
      -event.data.fetch('quantity')
    else
      return
  end
  daily = DailyReservation.lock.find_or_create_by!(
    sku:      event.data.fetch('sku'),
    store_id: event.data.fetch('store_id'),
    date:     event.datetime.to_date,
  )
  DailyReservation.update_counters(daily.id, total_reserved: value)
end
```

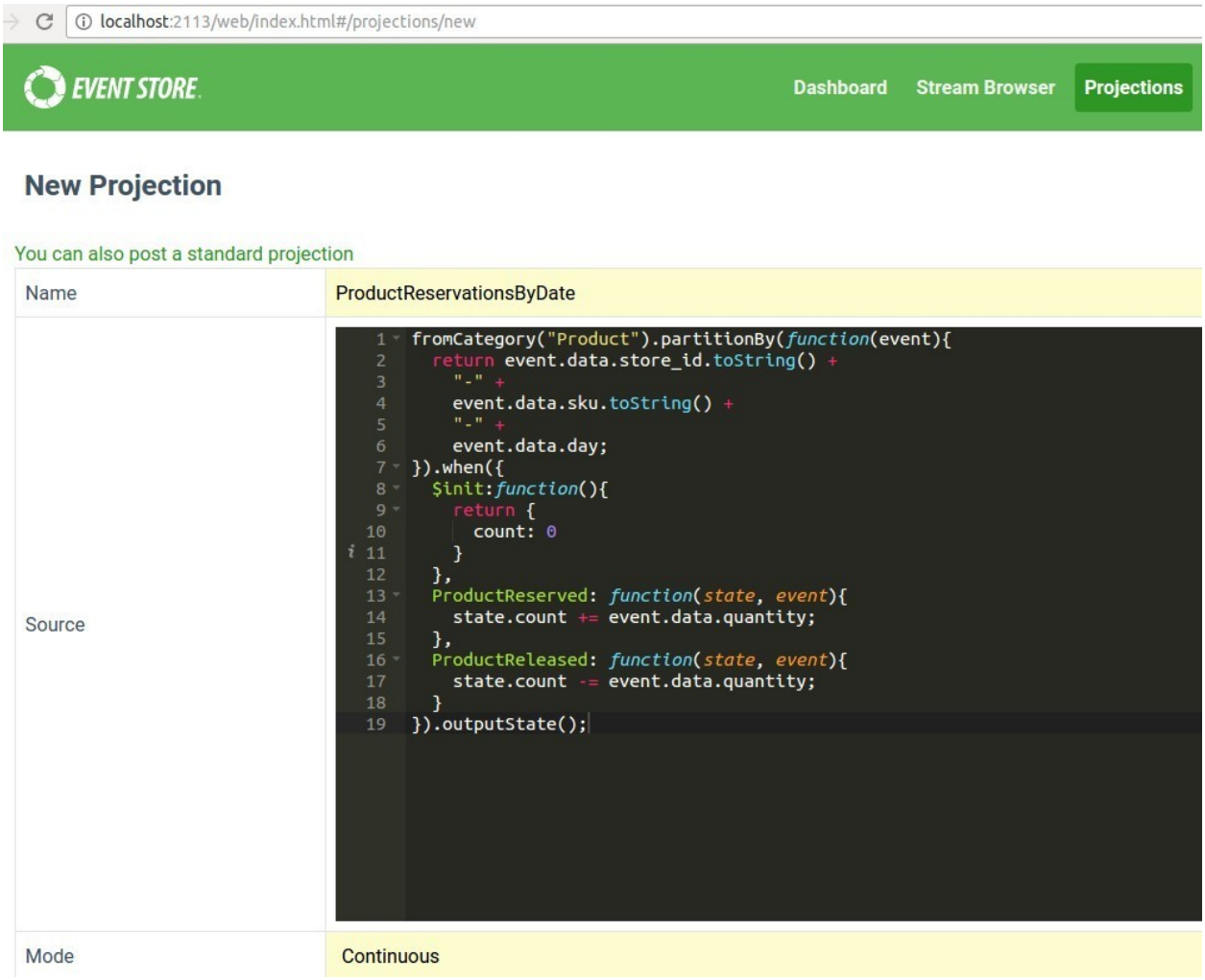
In this way, you implement a simple, catch-up subscription which can be stopped at any moment and restarted later. If the logic of a read model that you want to create changes you deploy new code and process all the events from the beginning of your system again. Once the subscription is up to date and processes freshly incoming data, you can expose its result to end users and discard the old version.

Sidenote: `$all` is a special stream in GES which also requires more permissions. You can enable `$by_category` system projection. It will link events from all `Product-*` streams into one, big `$ce-Product` stream. You can iterate through this stream in an identical way. (Incidentally, we are going to use it in a moment.)

EventStore Projection

The EventStore DB (eventstore.org, GES) has the ability to execute projections. Projections are written in JavaScript, stored in GES and executed when new events are added to streams.

Projections can either be supplied to Event Store directly via the HTTP API. You can also use the Administration User Interface. It provides a section for writing projections.



New Projection UI

As an example, we are going to write a `ProductReservationsByDate` projection. It's similar to what we implemented a moment ago using SQL DB, via reading events.

For my example to work, I enabled the `$by_category` system projection. It links events from all `Product-*` streams into one, big `$ce-Product` stream.

Let's see the code of our new `ProductReservationsByDate` projection and analyze it step-by-step.

```
fromCategory("Product").partitionBy(function(event){
  return event.data.store_id.toString() +
    "-" +
    event.data.sku.toString() +
    "-" +
    event.data.day;
}).when({
  $init:function(){
    return {
      count: 0
    }
  },
  ProductReserved: function(state, event){
    state.count += event.data.quantity;
  },
  ProductReleased: function(state, event){
    state.count -= event.data.quantity;
  }
}).outputState();
```

We start with:

```
fromCategory("Product")
```

`fromCategory(category)` selects events from the `$ce-{category}` stream. So in our case, that's going to be `$ce-Product`. As I mentioned, it is a special stream in which events related to all our products are going to appear as well.

As you've seen previously, events regarding different Aggregate roots are stored in different streams such as

- `"Product-1-WORKSHOP"`
- `"Product-1-BOOK"`
- `"Product-2-BURGER"`

...and all those events are also be going to be copied into a `$ce-Product` stream that we will be reading in our projection. The copying itself is done by a system projection named `$by_category` that is available but stopped by default. I've enabled it in the admin panel.

Let's go back to our `ProductReservationsByDate` projection.

```
fromCategory("Product").partitionBy(function(event){
  return event.data.store_id.toString() +
    "-" +
    event.data.sku.toString() +
    "-" +
    event.data.day;
}).when({
  $init:function(){
    return {
      count: 0
    }
  },
  ProductReserved: function(state, event){
    state.count += event.data.quantity;
  },
  ProductReleased: function(state, event){
    state.count -= event.data.quantity;
  }
}).outputState();
```

The next part is:

```
partitionBy(function(event){
  return event.data.store_id.toString() +
    "-" +
    event.data.sku.toString() +
    "-" +
    event.data.day;
})
```

We want our projection to compute how much of a given product was reserved in a given day. With thousands of possible products and thousands of possible days, the state could grow quite big. And most likely you don't ever need the whole state. So we are going to partition it by

- store_id (tenant id)
- sku (product id)
- day (date)

and that's what our `partitionBy` does by producing a string such as `1-WORKSHOP-2017-09-15`.

I had to add a `day` attribute to the event's data. Unfortunatelly, the event timestamp is not automatically exposed in projections.

```
class Product
  def reserve(quantity:, order_number:)
    unless @quantity_available >= quantity
      raise QuantityNotAvailable
    end
    apply(pr = ProductReserved.new({
      store_id: @store_id,
      sku: @sku,
      quantity: quantity,
      order_number: order_number,
      day: Date.current.iso8601,
    }))
    return [pr]
  end
```

And finally the real business logic of our read model.

```
when({
  $init:function(){
    return {
      count: 0
    }
  },
  ProductReserved: function(state, event){
    state.count += event.data.quantity;
  },
  ProductReleased: function(state, event){
    state.count -= event.data.quantity;
  }
})
```

This is the logic which counts how much of a product was reserved.

We have `$init` to initialize the starting state. Based on event type we provide functions to update the state. These functions have access to the event's data and the current state.

The last part is `outputState()`. It will append events with the computed state into a stream called `$projections-ProductReservationsByDate-result`.

Now that we have our continuous projection running, we can query its state using the HTTP API.

```
curl -i http://localhost:2113/projection/ProductReservationsByDate/state?partiti\
on=2-BOOK-2017-09-14
{"count":22}
```

```
curl -i http://localhost:2113/projection/ProductReservationsByDate/state?partiti\
on=1-WORKSHOP-2017-09-15
{"count":7}
```

Why Event Sourcing basically requires CQRS and Read Models

Event sourcing is a nice technique with certain benefits. But it has a big limitation. As there is no concept of easily available *current state*, you can't get an answer to a query such as *give me all products with available quantity lower than 10.*

You could read `Product-1` stream of events for `Product` with `ID=1`. Then use them to rebuild the current state of this one product and get an answer to whether it has less than 10 available quantity. But to find all such products, you would need to iterate over all `Product-*` streams, and process all domain events stored for all products. That would be costly and take a lot of time.

All that use-cases that you see in your daily job get a little harder:

- Show me last 10 registered users
- Find customers by emails or address
- What's the total amount of all transaction from this month
- What's the Life Time Value of a customer
- Search all products with the text *blue pillow*

and so on, and so on...

Why?

Because when an Entity/Aggregate is event-sourced, there is only one method you can ask the repository about the object. And that's `find_by_id` (or `find` or `load`, the exact name does not matter). That's it.

You know the `Id` from somewhere ie: other entity has a reference to it, or from UI, or from API, or from a request. And you can do:

```
id = params[:id]
Product.new.load("Product-#{id}")
```

The code will know what stream of events it should read (ie. `Product-1`), those events will be applied on a `Product` instance and we will re-build the current state of one product. That's it.

So what's the solution to all those before-mentioned use-cases? Read models.

If you run e-commerce app and `Product` is event sourced, you are going to need to have a read-model of Products. In case you want to display a list of products so that customers can browse them and call commands like `AddToBasket`. This read model can be in Elastic Search or in SQL or in any DB you want. That's up to specific requirements.

How does the process of building a read model work in steps?

1. When you update the product, you do it by saving new domain events.
2. Event handlers are triggered
    - They can be triggered by a message queue that you pushed events into, after they had been stored
        - In simplest case that can be implemented using ActiveJob, in more complex scenarios it can be Kafka, Rabbit or Amazon SQS.
    - Or you have a separate process (a projection) constantly iterating over saved domain events and picking them up for processing.
        - This is very simple when you use EventStore DB[80] for saving domain events.
3. The event handler updates the read model accordingly based on what happened, what domain event it is processing.

As an example.

`ProductRegistered` event can cause adding a new element to ActiveRecord-backed read model `ProductList`.

```
ProductList.create!(
  id: event.data[:product_id],
  name: event.data[:name],
  price: BigDecimal.new(event.data[:price]),
)
```

`ProductPriceChanged` event can cause updating the price on the list.

[80] https://eventstore.org/

```
ProductList.
  find_by!(id: event.data[:product_id]).
  update_attributes!(
    price: BigDecimal.new(event.data[:price]),
  )
```

etc etc.

And then when you want to display *10 most expensive products* you can do it based on the read side of your application, based on the `ProductList` read-model.

```
ProductList.order("price DESC").limit(10)
```

The write-side of your application, the event-sourced `Product` class is about making changes, keeping track of them, and protecting business rules. It's the side responsible for publishing `ProductRegistered` or `ProductPriceChanged`.

Sidenote on Functional & Immutable approach

I've shown you examples of Event Sourcing implementations. They use objects and mutate their internal state while applying events. But nothing stops you from going more functional-style. You can always return a new, immutable object after applying an event. Also, if you like primitive structures such as Hash, go ahead and use them instead of objects. That is going to work as well. It's just not my style, do I didn't present it.

Why use event sourcing

- *It is not a new concept*

  A lot of domains in the real world work like that. Your bank statement, for example, is not the current state - it is a log of domain events. (If you are still not convinced talk to your accountant ;)
- *Temporal aspect*

  By replaying an event we could get the state of an aggregate for any moment in time. That could greatly help us to understand our domain and why things change. Not to mention helping to debug really nasty errors.
- *No ORM mapping*

  There is no coupling between the representation of current state in the domain and in storage. We are avoiding impedance mismatch between the object oriented and relational worlds.

- *Append-only*

  An append-only model storing events is a far easier model to scale. By having a read model we could have best of both worlds. A read side optimised for fast queries, and a write side highly optimised for writes. Since there is no delete here, it could be really fast writes.
- *Intention*

  Beside the "hard" data we also store the user's intentions. The order of events stored could be used to analyze what the user was really doing.
- *Audit log for free*

  And this time the audit log really has all the changes (remember there is no change of state if there is an event for that).

> Every database on the planet sucks. And they all suck it their own unique original ways.

For me the biggest advantage is that I can have different data models generated based on domain events stored in Event Store. Having an event log allows us to define new models, appropriate for new business requirements. Such models could not only be tables in a relational database. They could be anything. A graph data model to store relations between contractors in your system with an easy way to find how the are connected to each other. A document database. Or a static HTML page if you are building the newest and fastest (and of course most popular) blogging platform.

> As the events represent every action the system has undertaken, any possible model describing the system can be built from the events.

You might not know future requirements for your application. But having an event log you could build a new model that hopefully will satisfy emerging business requirements. And one more thing: it won't be that hard. No long migrations, no trying to guess when something has changed. Just replay all your events and build the new model based on the data stored in them.

Versioning in an Event Sourced System

I am not going to touch on this subject as it is Ruby/Rails agnostic. Gregory Young did a great job in Versioning in an Event Sourced System[81]. You can buy it for $5 or read online for free. This book is 70% complete.

Sidenote

Some code presented in this chapter comes from unreleased version of RailsEventStore. It's available on `master` branch and should be released soon.

[81] https://leanpub.com/esversioning/

Read More

- Versioning in an Event Sourced System[82] - ebook by Gregory Young

  Have you had troubles with versioning an Event Sourced system? Just getting into a new Event Sourced system for the first time? This book is the definitive guide of how to handle versioning over long periods of time.
- CQRS Documents by Greg Young[83]

  This free mini-book clearly describes the stereotypical architecture of an application. Its pros (such as simplicity and available tooling) and cons (not suitable for DDD, often leading to Anemic Model). Over the course of the book, you find out how this architecture can be improved in incremental steps. Attempting to limit or remove cost, adding business value at each additional step. You learn the problems with typical UIs and how a Task Based User Interface prevents losing the intention of a change. It shows how a typical architecture can by transformed more towards EventSourcing. And how EventSourcing plays well with CQRS and why. Great read, recommended!
- Scaling Event Sourcing for Netflix Downloads[84]

  > Phillipa Avery and Robert Reta describe how Netflix successfully launched their Download feature with the use of a Cassandra-backed event sourcing architecture. They describe their event store implementation and cover what they learned along the way, including what they could have done better. Finally, they review some improvements and extensions that they are planning to address going forward.

- Greg Young on Why use Event Sourcing[85]

  Greg nicely presents the potential problems of using an architecture where you update current state and publish domain events.

  > There are however some issues that exist with using something that stores a snapshot of the current state. The largest issue is that you have introduced two models to your data, an event model and a model representing current state.
  >
  > One must create the code to save the current state of the objects and one must write the code to generate and publish the events. No matter how you go about these tasks, it cannot possibly be easier than only publishing events.

  And how the Event Sourcing approach is simpler because the events and state don't diverge easily, so there is no synchronization.

  It contains a good introduction to Why use Event Sourcing and ends with a simple list of pros and cons compared to a typical architecture of an app which only stores current state.

[82] https://leanpub.com/esversioning/
[83] https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf
[84] https://www.infoq.com/presentations/netflix-scale-event-sourcing
[85] http://codebetter.com/gregyoung/2010/02/20/why-use-event-sourcing/

- Greg Young on CQRS and Event Sourcing[86]

  Greg explains why Event Sourcing and CQRS really go hand in hand and why those two patterns enjoy a symbiotic relationship.
- Event Sourcing in Akka[87]
- Martin Fowler on Event Sourcing[88]

  > Event Sourcing ensures that all changes to application state are stored as a sequence of events. Not just can we query these events, we can also use the event log to reconstruct past states, and as a foundation to automatically adjust the state to cope with retroactive changes.

- EventStore DB[89]

  > The open-source, functional database with Complex Event Processing in JavaScript.

- Pros and Cons of Event/Command Sourcing[90]
- Proposal to introduce AutoConflictResolution into AggregateRoot gem[91]
- EventStore - TCP vs HTTP api[92]

  > TCP will be faster
  >
  > In addition, the number of writes per second which can be supported is often dramatically higher when using TCP. At the time of writing, standard Event Store applicances can service around 2000 writes/second over HTTP compared to 15,000-20,000/second over TCP! This might be a deciding factor if you are in a high-performance environment.
  >
  > AtomPub is more scalable for large numbers of subscribers.

- Custom projections in EventStore[93]

  Documents the options, selectors, filters, transformations, functions and handler's API.
- System projections in EventStore[94]

  Documents projections available out of box in GES, such as the `$by_category` which we used in one example.
- UPSERT in postgresql[95]

[86] http://codebetter.com/gregyoung/2010/02/13/cqrs-and-event-sourcing/
[87] http://doc.akka.io/docs/akka/2.5.3/scala/persistence.html#event-sourcing
[88] http://martinfowler.com/eaaDev/EventSourcing.html
[89] https://eventstore.org
[90] http://ookami86.github.io/event-sourcing-in-practice/#considering-event-sourcing/01-pros-and-cons-of-event-sourcing.md
[91] https://github.com/RailsEventStore/rails_event_store/issues/107
[92] https://eventstore.org/docs/introduction/4.0.0/which-api/
[93] https://eventstore.org/docs/projections/4.0.0/user-defined-projections/
[94] https://eventstore.org/docs/projections/4.0.0/system-projections/
[95] https://wiki.postgresql.org/wiki/UPSERT

- UPSERT in MongoDB[96]

[96] https://docs.mongodb.com/manual/reference/method/db.collection.update/#upsert-option

Bonus chapters

These bonus chapters are not the main part of our book, we are certain reading them can be beneficial to you as well. Usually they come from our weekly blog-posts.

If you don’t want to miss free knowledge from us, subscriber at http://arkency.com/newsletter for our newsletter.

One simple trick to make Event Sourcing click

Event Sourcing is like having two methods when previously there was one. There — I've said it.

But it isn't my idea at all.

It was Greg that used it first, in a bit different context. When explaining CQRS[97] he used this exact words:

> Starting with CQRS, CQRS is simply the **creation of two objects where there was previously only one**. The separation occurs based upon whether the methods are a command or a query (the same definition that is used by Meyer in Command and Query Separation, a command is any method that mutates state and a query is any method that returns a value).

You can have quite a similar statement on event-sourced aggregate root. The separation occurs based upon whether the method:

- corresponds to an **action** we want to take on an aggregate — protects **business rules** and tells what domain event happened if those were met
- maps **consequences of the domain event** that happened to internal state representation (against which business rules are executed)

Not convinced yet? Let the examples speak.

Stereotypical aggregate without Event Sourcing

Below is a typical aggregate root. In the scope of the example there are only two actions you can take — via public **register** and **supply** methods.

[97] http://codebetter.com/gregyoung/2010/02/16/cqrs-task-based-uis-event-sourcing-agh/

```ruby
class Product
  CannotSupply = Class.new(StandardError)
  AlreadyRegistered = Class.new(StandardError)

  def initialize(store_id: nil, sku: nil, quantity_available: 0)
    @store_id = store_id
    @sku = sku
    @quantity_available = quantity_available
  end

  def register(store_id:, sku:, event_store:)
    raise AlreadyRegistered if @store_id
    @store_id = store_id
    @sku = sku

    event_store.publish_event(ProductRegistered.new(data: {
      store_id: @store_id,
      sku: @sku,
    }))
  end

  def supply(quantity, event_store:)
    raise CannotSupply unless @store_id && @sku

    @quantity_available += quantity

    event_store.publish_event(ProductSupplied.new(data: {
      store_id: @store_id,
      sku: @sku,
      quantity: quantity,
    }))
  end
end
```

Aggregate with Event Sourcing

In event sourcing it is the domain events that are our source of truth. They state what happened. What we need to do is to make them a bit more useful and convenient for decision making. This is the **sourcing** part.

```
class Product
  CannotSupply = Class.new(StandardError)
  AlreadyRegistered = Class.new(StandardError)

  def initialize(store_id: nil, sku: nil, quantity_available: 0)
    @store_id = store_id
    @sku = sku
    @quantity_available = quantity_available
  end

  def register(store_id:, sku:, event_store:)
    raise AlreadyRegistered if @store_id

    event = ProductRegistered.new(data: {
      store_id: store_id,
      sku: sku,
    })

    event_store.publish_event(event)
    registered(event)
  end

  def supply(quantity, event_store:)
    raise CannotSupply unless @store_id && @sku

    event = ProductSupplied.new(data: {
      store_id: @store_id,
      sku: @sku,
      quantity: quantity,
    })

    event_store.publish_event(event)
    supplied(event)
  end

  private

  def supplied(event)
    @quantity_available += event.data.fetch(:quantity)
  end

  def registered(event)
```

```
43      @sku = event.data.fetch(:sku)
44      @store_id = event.data.fetch(:store_id)
45    end
46  end
```

In this step we've drawn the line between making a statement that something happened (being possible to happen first) and what side effects does it have. Notice private **registered** and **supplied** methods.

Why make such effort and introduce indirection? The reason is simple — if the events are source of truth, we could not only shape internal state for current actions we take but also for the ones that happened in the past.

Instead of loading current state stored in a database, we can take collection of events that happened in scope of this aggregate — in its stream.

```
1   class Product
2     CannotSupply = Class.new(StandardError)
3     AlreadyRegistered = Class.new(StandardError)
4
5     def initialize(store_id: nil, sku: nil, event_store:)
6       stream_name = "Product$#{store_id}-#{sku}"
7       events = event_store.read_all_events_forward(stream_name)
8       events.each do |event|
9         case event
10        when ProductRegistered then registered(event)
11        when ProductSupplied then supplied(event)
12        end
13      end
14    end
15
16    def register(store_id:, sku:, event_store:)
17      raise AlreadyRegistered if @store_id
18
19      event = ProductRegistered.new(data: {
20        store_id: store_id,
21        sku: sku,
22      })
23
24      event_store.publish_event(event)
25      registered(event)
26    end
27
```

```
28    def supply(quantity, event_store:)
29      raise CannotSupply unless @store_id && @sku
30
31      event = ProductSupplied.new(data: {
32        store_id: @store_id,
33        sku: @sku,
34        quantity: quantity,
35      })
36
37      event_store.publish_event(event)
38      supplied(event)
39    end
40
41    private
42
43    def supplied(event)
44      @quantity_available += event.data.fetch(:quantity)
45    end
46
47    def registered(event)
48      @sku = event.data.fetch(:sku)
49      @store_id = event.data.fetch(:store_id)
50    end
51  end
```

At this point you may have figured out that `event_store` dependency that we constantly pass as an argument belongs more to the infrastructure layer than to a domain and business.

What if something above passed a list of events first so we could rebuild the state? After an aggregate action happened we could provide a list of domain events to be published (`unpublished_events`):

```
1   class Product
2     CannotSupply = Class.new(StandardError)
3     AlreadyRegistered = Class.new(StandardError)
4
5     attr_reader :unpublished_events
6
7     def initialize(events)
8       @unpublished_events = []
9       events.each { |event| dispatch(event) }
10    end
11
12    def register(store_id:, sku:)
```

```
13      raise AlreadyRegistered if @store_id
14
15      apply(ProductRegistered.new(data: {
16        store_id: store_id,
17        sku: sku,
18      }))
19    end
20
21    def supply(quantity)
22      raise CannotSupply unless @store_id && @sku
23
24      apply(ProductSupplied.new(data: {
25        store_id: @store_id,
26        sku: @sku,
27        quantity: quantity,
28      }))
29    end
30
31    private
32
33    def apply(event)
34      dispatch(event)
35      @unpublished_events << event
36    end
37
38    def dispatch(event)
39      case event
40      when ProductRegistered then registered(event)
41      when ProductSupplied then supplied(event)
42      end
43    end
44
45    def supplied(event)
46      @quantity_available += event.data.fetch(:quantity)
47    end
48
49    def registered(event)
50      @sku = event.data.fetch(:sku)
51      @store_id = event.data.fetch(:store_id)
52    end
53  end
```

More or less this reminds the aggregate_root[98] gem that is aimed to assist you with event sourced aggregates.

The rule of **having two methods when there was previously one** however still holds.

- **The public method** (such as `supply`) corresponds to an **action** we want to take on an aggregate — protects **business rules** and tells what domain event happened if those rules were met.
- **The private method** (such as `supplied`) maps **consequences of the domain event** that happened to the internal state representation.

[98] https://github.com/RailsEventStore/rails_event_store/tree/master/aggregate_root

Physical separation in Rails apps

I've been just following an interesting discussion within our Rails/DDD community[99]. The particular topic which triggered me was a debate what kind of separation is possible within a Rails app which aims to be more in the DDD spirit.

I remember, back in the school, I was taught about the difference between the physical separation of components vs conceptual separation. This was a part of a software engineering class I took. I hope I got it right.

If we translate it into Rails terminology + some bits of DDD, we want to have the conceptual separation at the level of Bounded Context. A Bounded Context is a world in its own. It has its own "model" (don't confuse it with the ActiveRecord models) which represents the domain.

Your typical Rails app can have 5-50 different bounded contexts, for things like Inventory, Invoicing, Ordering, Catalog. If you also go CQRS, then some of your read models (or collections of them) can also be considered bounded contexts, but that's more debatable.

Anyway, once you have the bounded contexts (some people like to call them modules or components), that's a conceptual concept. Often you implement it with Ruby namespaces/modules.

But what options do you have for a physical separation?

Directories

You can just create a new directory for the bounded context files and keep the separation at this level. It can be at the top-level directory of your Rails app or at the level of `lib/`. This usually works fine, however you need to play a bit with Rails autoloading paths to make it work perfectly.

This is probably the default option we choose for the Rails apps at Arkency.

Gems

Another level is a gem. It can be still within the same app directory/repo, but you keep them in separate directories, but declare the dependency at the Gemfile level.

Gems + repos

The same as above, but you also separate the repos. This can create some more separation, but also brings new problems, like frequent jumps between repos to make one feature.

[99] http://blog.arkency.com/domain-driven-rails/

Rails engines

This is a way, probably most in the spirit of The Rails Way. Which is both good and bad. Technically it can be a rational choice. However, it's a tough sell, as usually people who jump into Rails+DDD are not so keen to rely on Rails mechanisms too heavily. Also, many people may think of that separation as at the controllers level, which doesn't have to be the case.

BTW, splitting at the web/controllers level is an interesting technique of splitting your app/infra layer, but it's less relevant to the "domain" discussions". I like to split the admin web app, as it's usually a separate set of controllers/UI. The same with API. But still, this split is rarely a good split for your domain, that's only the infra layer.

Anyway, Rails engines can be perfectly used as a physical separation of the domain bounded contexts. If you don't mind the dependency on Rails (for the physical separation mechanism) here, then that's a good option.

It's worth noting that gems and engines were chosen as the physical separation by our friends from Pivotal, they call it Component-Based Rails Applications[100].

Microservices

This approach relies on having multiple nodes/microservices for our app. Each one can be a Rails application on its own. It can be that one microservice per bounded context, but it doesn't need to be like that. In my current project, we have 6 microservices, but >15 bounded contexts.

I wasn't a big fan of microservices, as they bring a lot of infrastructure overhead. My opinion has changed after I worked more heavily with a Heroku-based setup. The tooling nowadays has improved and a lot is offered by the platform providers.

It's worth noting that you can separate the code among several repos. However, you can also keep them in one monorepo. With a heroku-based setup, it seems to be simpler to keep them separated, but one repo should also be possible.

Microservices also allow another separation - at the programming language level. You can write each microservice in different languages, if it makes sense for you. It's an option not possible in previous approaches.

Serverless aka Function as a Service

This is a relatively new option and probably not mostly considered. Especially that the current serverless providers don't support Ruby out of the box. Serverless is quite a revolution happening and they can change a lot in regards to the physical separation.

[100] http://shageman.github.io/cbra.info/

What's possible with serverless is to not only separate physically bounded contexts, but also the smaller building blocks, like aggregates, read models, process managers (sagas).

I'm not yet experienced enough with how to use it with Rails, but I'm excited about this option. As with microservices, this gives an option to use a different programming language, but at a lower scale. While, I'd be scared to implement a whole app in Haskell, I'm super ok, if we implement one read module or one aggregate in Haskell. In the worst case, we can rewrite those 200 LOC. Another big deal with serverless is the fact that they handle the HTTP layer for you. Does it mean "good bye Rails"? I'm not sure yet, but possibly it can reduce the HTTP-layer of our codebases to a minimum.

Summary

The nice thing with a DDD-based architecture of your application is that it mostly works with whichever physical separation you choose. It's worth noting that those physical mechanisms can change over time. You can start with a Rails engine, then turn it into a microservice and then split it into several serverless functions.

Bounded contexts as gems vs directories

There has been a very interesting discussion on #ruby-rails-ddd community slack channel[101]. The topic circulated around bounded contexts and introducing certain "component" solutions to separate them.

There are various approaches to achieve such separation — Rails Engines and CBRA[102] were listed among them. It was, however, the mention of "unbuilt gems" that reminded me of something.

Gem as a code boundary

Back in the day, we had an approach in Arkency in which distinct code areas were extracted to gems. They had the typical gem structure with `lib/` and `spec/` and top-level `gemspec` file.

```
# top-level app directory

scanner/
├── lib
│   ├── scanner
│   │   ├── event.rb
│   │   ├── event_db.rb
│   │   ├── domain_events.rb
│   │   ├── scan_tickets_command.rb
│   │   ├── scanner_service.rb
│   │   ├── ticket.rb
│   │   ├── ticket_db.rb
│   │   └── version.rb
│   └── scanner.rb
├── scanner.gemspec
└── spec
    ├── scan_tickets_command_spec.rb
    ├── scan_tickets_flow_spec.rb
    └── spec_helper.rb
```

[101] http://blog.arkency.com/domain-driven-rails/
[102] http://shageman.github.io/cbra.info/

```ruby
# scanner/scanner.gemspec

lib = File.expand_path('../lib', __FILE__)
$LOAD_PATH.unshift(lib) unless $LOAD_PATH.include?(lib)
require 'scanner/version'

Gem::Specification.new do |spec|
  spec.name          = 'scanner'
  spec.version       = Scanner::VERSION

  # even more stuff here

  spec.add_dependency 'event_store'
end
```

Each gem had its own namespace, reflected in gem's name. In example `Scanner` with `scanner`. It held code related to the scanning context, from the service level to the domain. You were able to run specs related to this particular area in separation. In fact, at that time we were just starting to use the term Bounded Context.

```ruby
# scanner/lib/scanner.rb

module Scanner
end

require 'scanner/version'
require 'scanner/domain_events'
require 'scanner/scanner_service'
require 'scanner/scan_tickets_command'
require 'scanner/ticket'
require 'scanner/ticket_db'
require 'scanner/event'
require 'scanner/event_db'
```

Yet these gems were not like the others. We did not push them to RubyGems obviously. Neither did we store them on private gem server. They lived among Rails app, at the very top level in the code repository. They're referenced in `Gemfile` using `path:`, like you'd do with vendored dependencies.

```
# Gemfile

gem 'scanner', path: 'scanner'
```

That way much of the versioning/pushing hassle was out of the radar. They could change simultaneously with the app that used them (starting in the controllers calling services from gems). Yet they organized cohesive concept in one place. Quite idyllic, isn't it? Well, there was only one problem…

Rails autoloading and you

Code packaged as gem suffers from Rails code reload mechanism. I just does not reload when you `reload!`. While that rarely bothers you with the dependencies distributed from RubyGems that you'd never change locally, it is an issue for "unbuilt" gems.

Struggle with Rails autoload is real. If you keep losing battles with it — go read the guide[103] thoroughly. That was also the reason we disregarded the gem approach.

Code component without gemspec

The solution we're happy with now does not differ drastically from having vendored gems. There's no `gemspec` but the namespace and directory structure from a gem stay. The gem entry in `Gemfile` is gone. Any runtime dependencies this gem had, go into `Gemfile` directly now.

What differs is that we no longer have `require` to load files. Instead, we use autoload-friendly `require_dependency`.

```
# scanner/lib/scanner.rb

module Scanner
end

require_dependency 'scanner/version'
require_dependency 'scanner/domain_events'
require_dependency 'scanner/scanner_service'
require_dependency 'scanner/scan_tickets_command'
require_dependency 'scanner/ticket'
require_dependency 'scanner/ticket_db'
require_dependency 'scanner/event'
require_dependency 'scanner/event_db'
```

[103] http://guides.rubyonrails.org/autoloading_and_reloading_constants.html

With that approach, you also have to make sure that Rails is aware to autoload code from the path[104] your Bounded Context lives in.

```
# config/application.rb

config.paths.add 'scanner/lib', eager_load: true
```

And that's mostly it!

As an example you put `Scanner::Ticket` into `scanner/lib/scanner/ticket.rb` as:

```
# scanner/lib/scanner/ticket.rb

module Scaner
  class Ticket
  end
end
```

Dealing with test files

If you wish to painlessly run spec files in the isolated directory there are certain steps to take.

First, the spec helper should be responsible to correctly load the code.

```
# scanner/spec/spec_helper.rb

require_relative '../lib/scanner'
```

Then the test files should require it appropriately.

```
# scanner/spec/scan_tickets_command_spec.rb

require_relative 'spec_helper'

module Scanner
  RSpec.describe ScanTicketsCommand do
    # whatever it takes to gain confidence in code ;)
  end
end
```

Last but not least — it would be a pity to forget to run specs along the whole application test suite on CI. For this scenario we tend to put following code in app `spec/` directory:

[104] http://blog.arkency.com/2014/11/dont-forget-about-eager-load-when-extending-autoload/

```ruby
# spec/scanner_spec.rb

path = Rails.root.join('scanner/spec')
Dir.glob("#{path}/**/*_spec.rb") do |file|
  require file
end
```

Summary

The solution picture above is definitely not the only viable option. It has worked for me and my colleagues thus far. No matter which one you're using — deliberate design with bounded contexts is a win[105].

In Component-Based Rails Applications ebook[106] you can find solutions how to make the code in gems work with code reloading.

[105] https://twitter.com/owickstrom/status/889819275820756992

[106] https://leanpub.com/cbra

How we save money by using DDD and Process Managers in our Rails app

It's obvious that e-commerce businesses depend on an ability to sell goods or services. If an e-commerce platform is unable to sell, even temporarily, it becomes useless. It suffers from short-term revenue loss, but also it appears as untrustworthy to both its customers and merchants.

Unfortunately only the biggest players can afford to become an independent payment service provider. All other platforms have to depend on external providers and these providers have their own problems: infrastructure failures, errors in code, DDOS attacks, etc. To ensure the ability to operate a platform needs at least two independent payment providers. There are numerous interesting challenges related to having more than one payment provider, but let's focus on one of them only - how to detect that a gateway is failing and be able to use another one instead.

One simple yet effective solution is to use a process manager that reacts to a set of business events, maintains its internal state and makes a decision to switch to the backup gateway when some condition is met.

Let's assume the system publishes these business events:

- `PaymentSucceeded` - published after a payment ends with a success
- `PaymentFailed` - published after payment is declined
- `RemoteTransactionCreateFailed` - in some of the gateways it's required to register a payment first, only then can a customer be redirected to a payment page

Each of these business events include `payment_provider_identifier` in their data so the information is available as to which provider was related to the event. These types of events allow us to detect if there is a problem with the payment gateway that is currently in use. A process manager can be used to react to these events and keep track of the number of failures. If this number exceeds a certain threshold a command is issued to switch to the backup provider:

```
module Payments
  class SwitchToFallbackProvider
    include CommandBusInjector

    class State < ActiveRecord::Base
      self.table_name = "switch_to_fallback_provider"

      def self.purge(payment_provider_identifier)
        where(
          payment_provider_identifier: payment_provider_identifier
        ).update_all(failed_payments_count: 0)
      end

      def self.failures_count(payment_provider_identifier)
        find_by(payment_provider_identifier: payment_provider_identifier).
          failed_payments_count
      end
    end

    def initialize(fallback_configuration:)
      @fallback_configuration = fallback_configuration
    end

    def call(fact)
      data = fact.data.symbolize_keys
      return if fallback_configuration.nil?

      case fact
        when PaymentEvents::PaymentSucceeded
          purge_state(data)
        when PaymentEvents::PaymentFailed, PaymentEvents::CreateRemoteTransactio\
nFailed
          payment_failed(data)
          state = process_state(data, fallback_configuration)
          if is_critical?(state)
            send_command(data)
            purge_state(data)
          end
      end
    end

    private
```

```
43
44      attr_reader :fallback_configuration
45
46      def process_state(data, fallback_configuration)
47        count = State.failures_count(data[:payment_provider_identifier])
48        count > fallback_configuration.max_failed_payments_count ? "critical" : "n\
49  ormal"
50      end
51
52      def purge_state(data)
53        State.purge(data[:payment_provider_identifier])
54      end
55
56      def payment_failed(data)
57        record = State.lock.find_or_create_by(
58          payment_provider_identifier: data[:payment_provider_identifier]
59        )
60        State.increment_counter(:failed_payments_count, record.id)
61      end
62
63      def is_critical?(state)
64        state == "critical"
65      end
66
67      def send_command(data)
68        command = Payments::SwitchToFallbackPaymentProvider.new(
69          payment_provider_identifier: data[:payment_provider_identifier],
70        )
71        command_bus.(command)
72      end
73    end
74  end
```

Let's trace what the process manager does when one of the events it's subscribed to happens.

After a failure, when `PaymentFailed` or `CreateRemoteTransactionFailed` occurs the process manager finds a related `State` record that keeps a current number of failures for the provider and increases the counter. Next it checks for the current state. If the number of failures defined in configuration wasn't exceeded the state is not critical so nothing more happens. If the state is critical a command is sent to the command bus that triggers provider switch. A command handler takes care of it and also notifies appropriate people about the fact. Then the counter is reset.

When there was a successful payment and `PaymentSucceeded` was published it just finds the state record and resets the counter.

To sum up: if too many problems pile up it makes a decision to switch to a backup provider.

Some may argue that a failed payment isn't actually a failure, but experience with payment gateways shows that it's a common problem when a provider seems to work normally but all or most of the payments are declined because for example, an underlying acquirer is having issues.

Of course, this solution is prone to false-positives. It can happen that a group of customers have their payments declined because lack of funds, etc. and no successful payment happens in the meantime. The process manager records it as a problem with payment provider and triggers a switch. However, in platforms with heavy traffic it's very unlikely. And even if it does happen it's still better than the inability to sell goods for a period of time before someone notices and manually selects a backup gateway.

One request can be multiple commands

It took me years to understand one simple truth. One HTTP request sent by a browser may include a number of separate logical commands. One request does not always equal one logical command.

If you are already past the *mind-blown* phase like me, it may even sound obvious to you. But it was a bumpy road for me to find enlightenment.

So why do we send multiple commands in one HTTP request instead of multiple separate requests?

- because of the limitations of non-scripted (no JS) browser form model
- because of how we build UIs

Before I elaborate let me tell you that it is not inherently bad that we have multiple commands in one request as long as we are aware of it. When we do it consciously and weigh the pros and cons. We can always compensate for it on the backend side.

Native browser form limitations

Browsers always send all fields (except for unchecked checkboxes, but Rails works around that with hidden inputs) even if they were not changed. This is a blessing when you just want to do a simple DB update. But it makes understanding user intent much harder sometimes because we would need to compare previous and new values (doable and even easy but it requires more effort).

Because we always see all provided attributes, we (developers) don't think much about whether the user intended to do only X without touching the rest of the fields at all. And maybe 90% of time users only change X and that the X action is quite important and should be a separate, dedicated, explicit command and corresponding explicit UI interface.

An example could be a "publish" checkbox. Perhaps publishing (and unpublishing) is so important that it deserves a dedicated "publish" button and `PublishCommand` in your domain. Of course, as always in programming, it depends on many factors.

How we build UIs

We often build our UI as a long list of inputs with a single "Save" button at the end. Especially when it comes to less frequently used parts of our applications. An example could be a page for updating your user *settings* where you can change things such as:

- avatar photo
- cover photo
- email
- password
- notification settings
- your personal page path or URL or nickname
- birthday
- privacy settings
- and sometimes many more things as well…

This is often just a long form with a "Save" button.

But not a single person wakes up in the morning thinking "I am going to change my avatar, and cover and email and password and privacy settings!".

It's much more likely they were browsing the Internet, found something inspiring and decided *hmm let's change my cover photo*. Or they were reading Hacker News and Reddit and heard about yet another password leak and decided to change their passwords on many websites. Or they got angry with a push notification and decided to turn it off. Or they decided to get rid of that silly, childish nickname they have been using for years and become more professional so they changed that.

But they don't come to this page to change everything. We just built such a UI for them because those things don't fit well anywhere else so we present them together on a massive "settings" page.

What to do about it?

I think the solution is to go more granular.

If there is a large form in your app consider splitting it into something smaller and more manageable.

The first step I try is to break the form down into multiple separate ones, each one with its own "Save" button.

Instead of 10 inputs + Save I have for example:

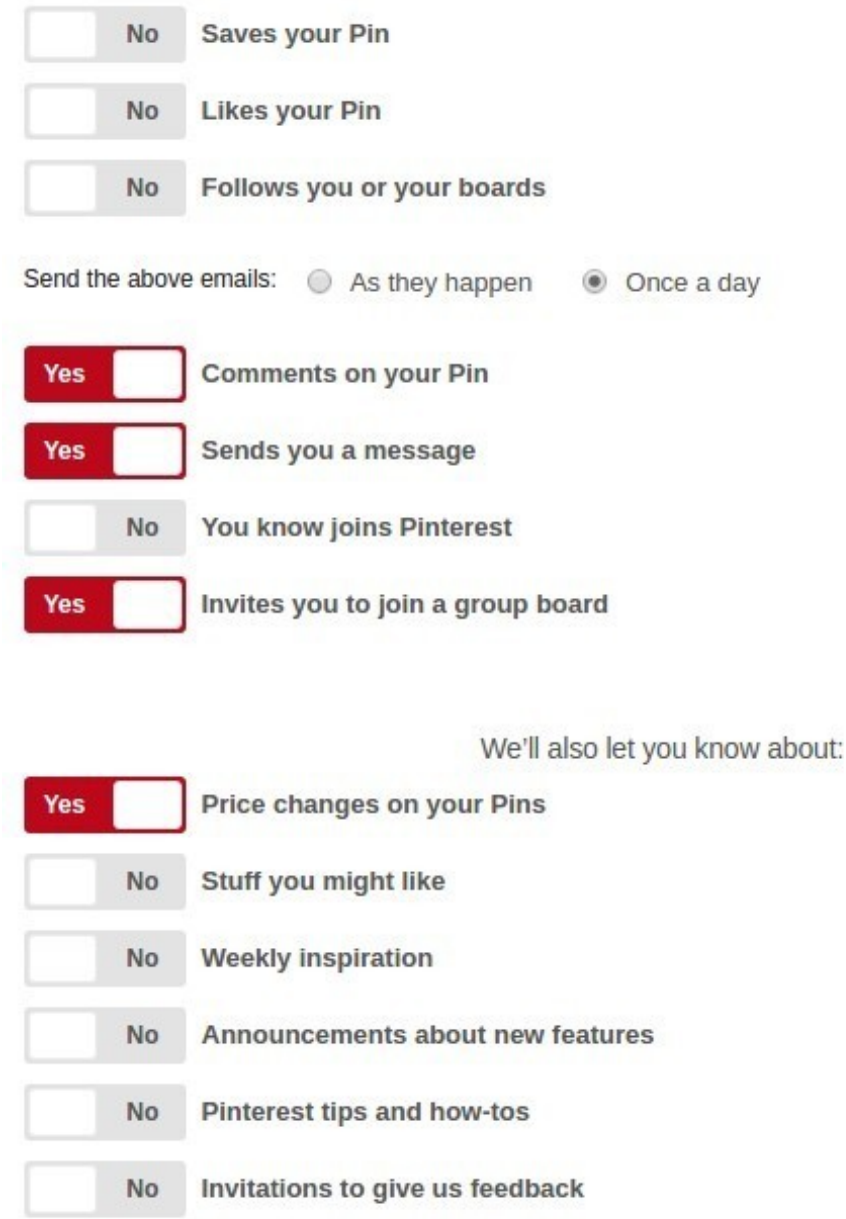
- 3 inputs + Save + divider
- 4 inputs + Save + divider
- 3 inputs + Save + divider

That way everything is still listed on one page, but the user can now update smaller, coherent, meaningful parts without thinking about the rest. The UI suggests (with dividers and grouping) what I am about to update. *Today I read an article about how SEO is important, so I am updating only the SEO settings of a product.*

The next step is to use JavaScript to improve the usability and intentionality of what the user wants to achieve even further.

For example, if there are fields which don't depend on anything else, they are completely separate and the cost of change (or revert of the decision) is minuscule. Perhaps we can automatically save the change directly when the user enters it.

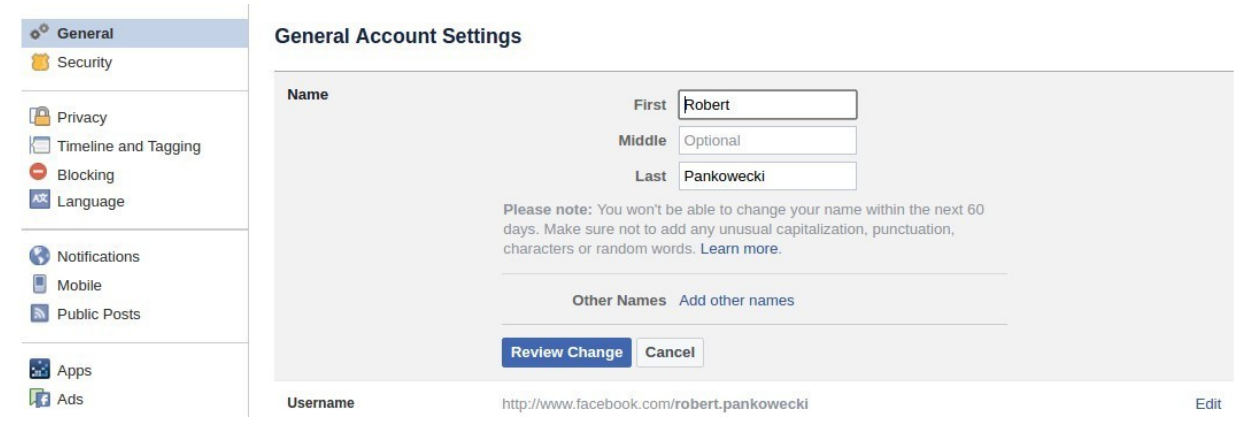
Examples



If setting a new value does not cause huge side-effects and is trivial for a user to revert, does it really require a “Save” button?

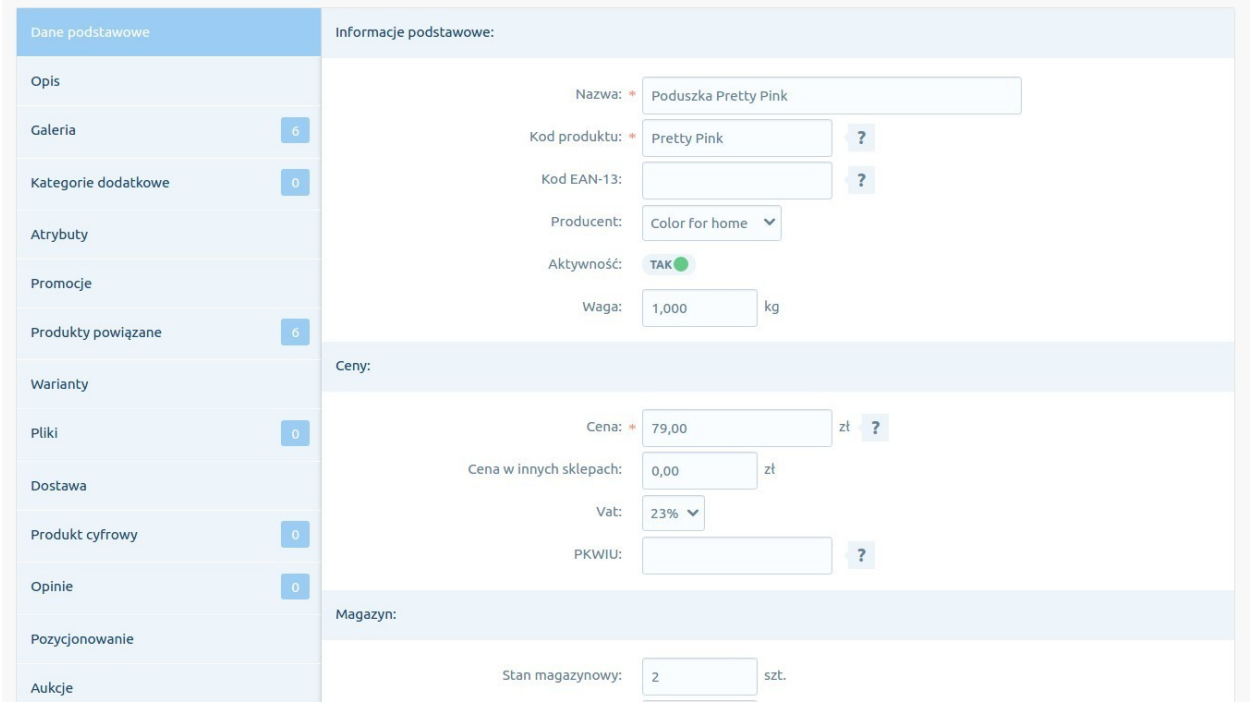
Or maybe we can send one request which translates to `DisableNotificationFor.new("saved_pin")` command?

<hr>



Grouping allows the user to better signal their intention and only update the specific field they need to change right now. They came to your app to perform a certain task.

<hr>



UI for changing a product in a shop. Options grouped in 14 logical categories.

Conclusion

Just because we received 20 different attributes from one form does not mean we need to construct one command with 20 attributes and pass it to a single Service Object. We might construct separate commands for groups of attributes and pass them further, even to different Service Objects.

Read more

- Task based UI[107]

[107] https://cqrs.wordpress.com/documents/task-based-ui/

Extract side-effects into domain event handlers

Introduction

As our applications grow over time, more and more things need to happen when certain situations occur in our system. They are often not directly related to the core of an action. They don't require co-operation to fulfill the action. They are side-effects of the action being executed. These dependencies often make the action more complicated and harder to understand.

For example aa a user registers in our ticketing system, we should assign to him/her the tickets, which have been given by a friend. But assigning those tickets is not a core to the registering user process. Therefore we might better handle such assignments in a "handler" reacting to a domain event when the user registered.

You can think of this technique as pub-sub on steroids. Because compared to classic pub-sub techniques you also gain the benefit that domain events are saved to a database and you can later review them for debugging.

Prerequisites

1. Install and configure `rails_event_store` gem.

Algorithm

1. Create new domain event class describing what happened.
2. Start publishing domain events after the change happened but before side-effects are called.
3. Create an empty handler
4. Register the handler to react to the domain event
5. Move the code of side-effect into the handler
6. (optional) Repeat for other side-effects.

Example

We are going to start with a classic controller doing quite a lot.

```ruby
class SignupsController < ApplicationController
  def new
    @user = User.new
  end

  def create
    @user = User.new(signup_params)

    respond_to do |format|
      if @user.save
        UserMailer.welcome_email(@user.email).deliver_now
        tracker.event("User Registered")
        if params[:subscribe_me]
          NewsletterSubscribeJob.perform_later(user.email, mailinglist_id)
        end
        LandingPage.find_or_create_by(name: params[:landing_page]).
          conversions.
          create!(
            user_id: @user.id
        ) if params[:landing_page]
        format.html { redirect_to @user, notice: 'Signup successful.' }
      else
        format.html { render new_signup_path }
      end
    end
  end

  private

  def signup_params
    params.require(:user).permit(:name, :email, :password)
  end

  def mailing_list_id
    current_country.mailinglists.default.id
  end
end
```

Besides classic registration (aka sign up) it also handles ...

- sending emails
- subscribing to a newsletter

- tracking landing pages for SEO stats
- and tracking analytics in JavaScript with analytical events recorded in the controller.

The important factor here is that all those factors do not change whether the user successfully registered or not. They are all side-effects which happen after a successful registration.

Sometimes you have dependencies in a process which are true collaborators. Meaning that you can't extract them, nor get rid of them, because they help to decide whether an action can succeed or not. Here that's not the case.

Let's introduce the domain event that we will be publishing when the user registers.

```
class UserRegisteredWithEmail < RubyEventStore::Event
  SCHEMA = {
    country_id: Integer,
    user_id: Integer,
    email:  String,
    newsletter_subscription: [FalseClass, TrueClass],
    landing_page: [NilClass, String],
  }.freeze

  def self.strict(data:)
    ClassyHash.validate(data, SCHEMA, true)
    new(data: data)
  end
end
```

It contains all the data necessary to convert all our side-effects into handlers. But we will just go with one. The domain event has a certain schema which makes it easy for handlers to know what they can expect when reacting to a certain domain event.

Now we can start publishing this event after something interesting happens. In our case that means when the user registers.

```
class SignupsController < ApplicationController
  def create
    @user = User.new(signup_params)

    respond_to do |format|
      if @user.save
        event_store.publish(UserRegisteredWithEmail.new(data:{
          country_id: current_country.id,
          user_id: @user.id,
```

```
10            email:  @user.email,
11            newsletter_subscription: !!params[:subscribe_me],
12            landing_page: params[:landing_page],
13          }), stream_name: "User$#{@user.id}")
14          UserMailer.welcome_email(@user.email).deliver_now
15          tracker.event("User Registered")
16          if params[:subscribe_me]
17            NewsletterSubscribeJob.perform_later(user.email, mailinglist_id)
18          end
19          LandingPage.find_or_create_by(name: params[:landing_page]).
20            conversions.
21            create!(
22              user_id: @user.id
23          ) if params[:landing_page]
24          format.html { redirect_to @user, notice: 'Signup successful.' }
25        else
26          format.html { render new_signup_path }
27        end
28      end
29    end
30
31    private
32
33    def signup_params
34      params.require(:user).permit(:name, :email, :password)
35    end
36
37    def mailing_list_id
38      current_country.mailinglists.default.id
39    end
40
41    def event_store
42      Rails.configuration.event_store
43    end
44  end
```

So far this will just save those domain events serialized in database. We can later use it for debugging, if we ever have such need.

Let's now create an empty handler.

```
class DeliverWelcomeEmail
  def perform(event)
  end
end
```

and subscribe that handler to be immediatelly called when `UserRegisteredWithEmail` occurs.

```
Rails.application.config.event_store.tap do |es|
  es.subscribe(->(event){
    DeliverWelcomeEmail.new.perform(event)
  }, [UserRegisteredWithEmail])
end
```

At this point the handler is called, but it is not doing anything so this is still a safe refactoring. Let's now move one of the side-effects into the handler...

```
class DeliverWelcomeEmail
  def perform(event)
    email = event.data.fetch(:email)
    UserMailer.welcome_email(email).deliver_now
  end
end
```

...and out of the controller...

```
class SignupsController < ApplicationController
  def create
    @user = User.new(signup_params)

    respond_to do |format|
      if @user.save
        event_store.publish(UserRegisteredWithEmail.new(data:{
          country_id: current_country.id,
          user_id: @user.id,
          email:  @user.email,
          newsletter_subscription: !!params[:subscribe_me],
          landing_page: params[:landing_page],
        }))
        ~~UserMailer.welcome_email(@user.email).deliver_now~~
        tracker.event("User Registered")
        if params[:subscribe_me]
```

```
17            NewsletterSubscribeJob.perform_later(user.email, mailinglist_id)
18          end
19          LandingPage.find_or_create_by(name: params[:landing_page]).
20            conversions.
21            create!(
22              user_id: @user.id
23          ) if params[:landing_page]
24          format.html { redirect_to @user, notice: 'Signup successful.' }
25        else
26          format.html { render new_signup_path }
27        end
28      end
29    end
30
31    # ...
32  end
```

Based on the other attributes provided in domain event you could easily extract the rest of the side-effect into other handlers.

```
1  class TrackLandingPageConversions
2    def perform(event)
3      landing_page = event.data.fetch(:landing_page)
4      return unless landing_page
5      LandingPage.find_or_create_by(name: landing_page).
6        conversions.
7        create!(
8          user_id: event.data.fetch(:user_id)
9      ) if params[:landing_page]
10   end
11 end
```

```
1  class SubscribeToNewsletter
2    def perform(event)
3      agreement = event.data.fetch(:subscribe_me)
4      return unless agreement
5      list_id = Country.find(event.data.fetch(:country_id)).
6        mailinglists.
7        default.
8        id
9      NewsletterSubscribeJob.perform_later(
```

```
      event.data.fetch(:email),
      list_id
    )
  end
end
```

and end up with a much smaller controller class:

```
class SignupsController < ApplicationController
  def create
    @user = User.new(signup_params)

    respond_to do |format|
      if @user.save
        event_store.publish(UserRegisteredWithEmail.new(data:{
          country_id: current_country.id,
          user_id: @user.id,
          email:  @user.email,
          newsletter_subscription: !!params[:subscribe_me],
          landing_page: params[:landing_page],
        }))
        format.html { redirect_to @user, notice: 'Signup successful.' }
      else
        format.html { render new_signup_path }
      end
    end
  end

  # ...
end
```

Benefits

Clean responsibilities

Your controller or service object is only interested in doing one thing. It's dependencies are only objects which help to finish the given action, such as registration. The myriad of side-effects are handled outside it.

Audit log

For free you get an audit log detailing what happened in your application. You can easily query the table storing domain events in your DB and get meaningful descriptions. Contrary to other solutions such as `papertrail` which store serialized state, with domain events you store a meaningful description of the change of state. So you don't need to look into diffs between states to figure out what happened in between.

Ability to disable in tests

In our code we disabled some of those side-effects which don't change much of the business process and provide very little value. Sending emails, generating PDFs and many other are simply disabled by not having certain handlers react to certain events in a test environment.

Easy to extend with another side-effects

When another part of the application needs to react when a user registers you don't even need to change the service or controller. You just add another handler subscribing to already existing domain events.

Warnings

Be aware if the order of calling side-effects matters or not.

If order matters try extracting them in the same order they are called. Otherwise you are free to proceed in any order you want.

Consider exception handling logic

Decide whether exceptions occurring in handler should prevent the whole process from finishing or not. Usually there is no reason to rollback everything. It is usually better to catch the exception, send it to a tracker and swallow without presenting it to the user. Especially in a case when the side-effects are not critical in any way to the system. Partial degradation is often better than a whole process not working.

```ruby
class DeliverWelcomeEmail
  def perform(event)
    email = event.data.fetch(:email)
    UserMailer.welcome_email(email).deliver_now
  rescue => e
    Honeybadger.notify(e)
  end
end
```

Keep things which require setting cookies or session in the controller.

Don't extract them to a Service Object or a Handler. In our case that would be code responsible for recording events which are later sent via JavaScript to Google Analytics, etc.

```ruby
tracker.event("User Registered")
```

Code tied to HTTP flow should remain in the controller.

You can perform those refactorings in a very similar manner after extracting Service Object as well.

I showed how to extract handlers from Controllers. But if you first extract most of this code to a Service Object, you can later extract side-effects into handlers. The procedure would be almost identical.

It's best to keep recording changes and publishing domain events in one db transaction

Since domain events are saved in the DB, for consistency prefer:

```ruby
ActiveRecord::Base.transaction do
  User.create!(...)
  event_store.publish(...)
end
```

over

```
User.create!(...)
event_store.publish(...)
```

Async handlers

It's possible to have asynchronous handlers (with some trade-offs) as well. Just serialize/deserialize the domain event (ie. using `YAML.dump` & `YAML.load` or `JSON`) in proper places.

```
Rails.application.config.event_store.tap do |es|
  es.subscribe(->(event){
    NewsletterSubscribeJob.perform_later( YAML.dump(event) )
  }, [UserRegisteredWithEmail])
end
```

```
class NewsletterSubscribeJob < ApplicationJob
  def perform(serialized_event)
    event = YAML.load(serialized_event)
    # ...
  end
end
```

In our code we use a little module included in handlers which make them work with normal domain events or serialized ones without the need to do it explicitly.

Links to resources:

- http://railseventstore.arkency.com/index.html
- https://github.com/RailsEventStore/rails_event_store/tree/master/aggregate_root#resources
- http://blog.arkency.com/2016/09/minimal-decoupled-subsystems-of-your-rails-app/
- http://blog.arkency.com/2016/05/domain-events-over-active-record-callbacks/

Reliable notifications between two apps or microservices

img: reliable-messaging-notifications-between-two-apps-micoservices-api/queue.png —

Let's say you have 2 systems or microservices (or processes). And one of them needs to be notified when something happened in another one. You might think it is not difficult until you start thinking about networking, reliability, and consistency.

Let's briefly examine some patterns to achieve this and what they typically bring to the table.

Direct communication (v1)

1. System A does something in a SQL transaction, which is committed.
2. System A contacts system B directly via API after the transaction is committed.

It all works nicely until system B is down and non-responsive. In such cases, it won't be notified about what happened in B, so we have a discrepancy. Assuming we have some kind of error reporting (and it worked at that moment) a developer can be notified about the problem and try to fix it manually later.

This, however, could be easily resolved, couldn't it? Let's just contact system B inside the DB transaction, instead of outside.

Direct communication (v2)

1. System A does something in a SQL transaction
2. System A contacts system B directly via API (still inside the DB transaction)
3. System A commits DB transaction.

Some developers believe this is a perfect solution, but they forget about one corner case that can still occur. Imagine that system B received your message (HTTP request) but you didn't receive a response (because networking is not completely reliable). In such cases there will most likely be an exception triggered in system A. It will rollback a DB transaction and pretend that nothing happened. But system B assumes it did happen. So we have a discrepancy again.

This situation might not happen just because the response did not get back. There are other cases where the final effect is the same. HTTP request was sent, but an application process was killed, or

server turned off. Or there was a bug in the code (if there is such code) between sending the request and committing the DB transaction).

I believe however that all those situations combined are less likely than server B just being unavailable. So probably this is superior to v1. But still not perfect.

Using external queue

1. System A does something in a SQL transaction
2. System A saves info in an external queuing system
3. System A commits DB transaction.
4. a) System A pulls jobs from queuing system and sends them to system B. Jobs can be retried in case of failure.

   or

   b) System B retrieves jobs from the queuing system and processes them. Jobs can be retried in case of failure.

Here we introduced an external queuing system such as Kafka, RabbitMQ or Redis. I call it external because the storage mechanism is using a different database then the application itself (which assumes SQL DB).

Also, depending on the situation, it might be your system (but another process, like a background worker solution) is taking jobs from the queue and pushing them further. Or it might be that another micro-service (system B) takes the jobs and processes them.

Notice that by introducing a queueing system in the middle with retry capability we changed the semantics from at-most-once delivery to at-least-once delivery.

It's still not all roses, however. We don't contact a separate system directly now, but we contact a separate database. With exactly the same potential pitfalls. What if we rollback after pushing to the queue? What if we pushed to the queue, but we didn't receive a confirmation and rolled-back in SQL? All the same situations can happen. Because we assume those servers running queues are closer to us, we also assume the likelihood of such problems happening is much lower. But still not zero. In my system, it happened 10 times in one month.

The assumption that both DBs are very close to each other is not always valid in the modern world anymore. If you use hosted Redis or hosted X there is a big chance they are going to be in the same region, but not necessarily the same availability zone.

To summarize: Thanks to retries we are safe from system B failures but we can still encounter problems on our side.

Using an internal queue

Ultimately the only safe solution is to use only one database - the same SQL database.

1. System A does something in a SQL transaction
2. System A saves info in an internal queuing system running hosted on the same SQL DB
3. System A commits DB transaction.
4. a) System A (another thread or process) takes jobs from the internal queuing system and sends them to system B.

   or

   b) System A (another thread or process) takes jobs from the internal queuing system and moves them to the external queuing system, where system B takes them from.

In this case, we save jobs info about what we want to notify external system about in the same SQL DB we store application state in. We can safely commit or rollback both of them together.

Then we either have background workers pulling from the same DB (internal queue) and communicating with system B or pushing those jobs to the external queue such as Kafka or RabbitMQ (one reason for that is there might be more systems than just B interested in this data).

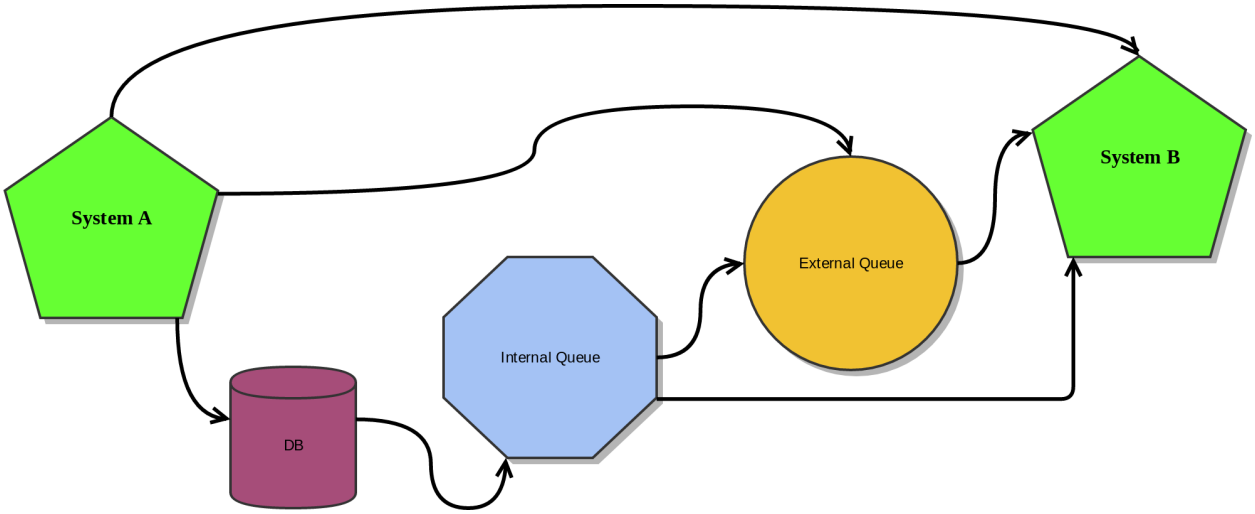
I am tempted to say that this gives you 100% reliability - probably not true and I am just missing a case where it can fail :)

Anyway, this is most likely the safest solution. But it requires more monitoring. Not only you need to watch system B, for the external queue, but now you also need to watch the thread or process moving data from the internal to the external queue.

Summary

How do you solve those problems in your system? Which solution did you go with?

I think some apps just ignore them and handle such issues manually (or not at all), because they are not crucial. But many systems I work on handle monetary transactions, so I am always cautious when thinking about such problems.



As you can see there are many ways system A can notify B about something (notice that we are talking about notifications, where A is not immediately interested in a response from B, just that it got the message and they are both in sync about the state of the world). You can do it directly, you can introduce external queues, you can have internal queues in the same DB or you can even go with both queues if you find it worthy of the cost of DevOps.

Links

Examples of internal queues:

- https://github.com/collectiveidea/delayed_job
- https://github.com/chanks/que

Examples of external queues:

- https://github.com/mperham/sidekiq
- https://kafka.apache.org/ + https://github.com/karafka/karafka
- https://www.rabbitmq.com/ + https://github.com/ruby-amqp/bunny

Other:

- Patterns for dealing with uncertainty[108]
- You Cannot Have Exactly-Once Delivery[109]
- Exactly-once Semantics are Possible: Here's How Kafka Does it[110]
  - With sequence numbers you can simulate exactly-once but only if the position is written atomically with whatever the consumer does.[111]

[108] /2016/12/techniques-for-dealing-with-uncertainity/
[109] http://bravenewgeek.com/you-cannot-have-exactly-once-delivery/
[110] https://www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it/
[111] https://twitter.com/gregyoung/status/881576308605673473

  - HN discussion[112]
- Delivering Billions of Messages Exactly Once[113]
  - HN discussion[114]

[112] https://news.ycombinator.com/item?id=14670801
[113] https://segment.com/blog/exactly-once-delivery/
[114] https://news.ycombinator.com/item?id=14664405

My first 10 minutes with Eventide

Recently I find out about Eventide project and it looked really nice to me based on the initial documentation so I wanted have a quick look inside. Here is what I discovered in the first few minutes without knowing anything about Eventide so far.

What's Eventide?

> Microservices, Autonomous Services, Service-Oriented Architecture, and Event Sourcing Toolkit for Ruby with Support for Event Store and Postgres

Sounds good to me!

- https://eventide-project.org/
- https://github.com/eventide-examples/account-component

File structure

My first thought was that there is quite an interesting structure of directories:

```
lib/
├── account
│   ├── client
│   │   ├── controls.rb
│   │   └── namespace.rb
│   └── client.rb
├── account_component
│   ├── account.rb
│   ├── commands
│   │   ├── command.rb
│   │   └── withdraw.rb
│   ├── consumers
│   │   ├── commands
│   │   │   └── transactions.rb
│   │   ├── commands.rb
│   │   └── events.rb
│   ├── controls
│   │   ├── account.rb
```

```
19  |   |   ├── commands
20  |   |   |   ├── close.rb
21  |   |   |   ├── deposit.rb
22  |   |   |   ├── open.rb
23  |   |   |   └── withdraw.rb
24  |   |   ├── customer.rb
25  |   |   ├── events
26  |   |   |   ├── closed.rb
27  |   |   |   ├── deposited.rb
28  |   |   |   ├── opened.rb
29  |   |   |   ├── withdrawal_rejected.rb
30  |   |   |   └── withdrawn.rb
31  |   |   ├── id.rb
32  |   |   ├── message.rb
33  |   |   ├── money.rb
34  |   |   ├── position.rb
35  |   |   ├── replies
36  |   |   |   └── record_withdrawal.rb
37  |   |   ├── stream_name.rb
38  |   |   ├── time.rb
39  |   |   └── version.rb
40  |   ├── controls.rb
41  |   ├── handlers
42  |   |   ├── commands
43  |   |   |   └── transactions.rb
44  |   |   ├── commands.rb
45  |   |   └── events.rb
46  |   ├── load.rb
47  |   ├── messages
48  |   |   ├── commands
49  |   |   |   ├── close.rb
50  |   |   |   ├── deposit.rb
51  |   |   |   ├── open.rb
52  |   |   |   └── withdraw.rb
53  |   |   ├── events
54  |   |   |   ├── closed.rb
55  |   |   |   ├── deposited.rb
56  |   |   |   ├── opened.rb
57  |   |   |   ├── withdrawal_rejected.rb
58  |   |   |   └── withdrawn.rb
59  |   |   └── replies
60  |   |       └── record_withdrawal.rb
```

```
61 |   ├── projection.rb
62 |   ├── start.rb
63 |   └── store.rb
64 └── account_component.rb
```

I was not sure where to look first to find the business logic but from the names you can quickly figure out what the project is about.

I looked at `start.rb` but that didn't tell me much:

```
1  module AccountComponent
2    module Start
3      def self.call
4        Consumers::Commands.start('account:command')
5        Consumers::Commands::Transactions.start('accountTransaction')
6        Consumers::Events.start('account')
7      end
8    end
9  end
```

I could not easily navigate to an interesting place in RubyMine so I just started poking around.

Commands

The first place that I felt on known ground was around files in `lib/account_component/messages/-commands/` which include commands such as:

```
1  #!ruby
2  # lib/account_component/messages/commands/deposit.rb
3  module AccountComponent
4    module Messages
5      module Commands
6        class Deposit
7          include Messaging::Message
8
9          attribute :deposit_id, String
10         attribute :account_id, String
11         attribute :amount, Numeric
12         attribute :time, String
13       end
14     end
15   end
16 end
```

Commands are for telling services/handlers what to do. They're just data structures. That's important. So we have our input. Let's see where it is coming into.

Handlers

In `lib/account_component/handlers/commands.rb` and `lib/account_component/handlers/commands/transactions.rb` you can find handlers and the logic for processing those commands. I won't show the whole code. It's pretty interesting. Just the most important snippets.

```
#!ruby

  category :account

  handle Open do |open|
    account_id = open.account_id

    account, version = store.fetch(account_id, include: :version)

    if account.open?
      logger.info(tag: :ignored) { "Command ignored (Command: #{open.message_typ\
e}, Account ID: #{account_id}, Customer ID: #{open.customer_id})" }
      return
    end

    time = clock.iso8601

    opened = Opened.follow(open)
    opened.processed_time = time

    stream_name = stream_name(account_id)

    write.(opened, stream_name, expected_version: version)
  end
```

What I recognized immediately was:

- getting account in its current version based on historically stored domain events.

```ruby
#!ruby
account, version = store.fetch(account_id, include: :version)
```

- saving new domain events in the account stream using optimistic concurrency (we expect the version has not changed since we got it last time)

```ruby
#!ruby
write.(opened, stream_name, expected_version: version)
```

The other interesting parts are:

- What seems to me like preserving idempotent behavior. Don't do anything if asked to `Open` account when the account is already `open?`

```ruby
#!ruby
if account.open?
  return
end
```

- building a new domain event with all the data

```ruby
#!ruby
opened = Opened.follow(open)
opened.processed_time = time
```

But at that point I could not easily navigate to `follow` method to check its implementation. I will probably find out later how it works.

Events

Anyway `Opened` is a domain event. Let's see it:

```ruby
#!ruby
# lib/account_component/messages/events/opened.rb
module AccountComponent
  module Messages
    module Events
      class Opened
        include Messaging::Message

        attribute :account_id, String
        attribute :customer_id, String
        attribute :time, String
        attribute :processed_time, String
      end
    end
  end
end
```

> Events are written to streams. All of the events for a given account are written to that account's stream. If the account ID is 123, the account's stream name is account-123, and all events for the account with ID 123 are written to that stream.

Classic thing if you already learned about event sourcing basics.

Handlers again

Here is an interesting thing:

> A handler might also respond (or react) to events by other services, or it might respond to events published by its own service (when a service calls itself).

```ruby
#!ruby
# lib/account_component/handlers/events.rb

  handle Withdrawn do |withdrawn|
    return unless withdrawn.metadata.reply?

    record_withdrawal = RecordWithdrawal.follow(withdrawn, exclude: [
      :transaction_position,
      :processed_time
    ])
```

```
11
12      time = clock.iso8601
13      record_withdrawal.processed_time = time
14
15      write.reply(record_withdrawal)
16    end
```

Replies

Events and commands are messages in EventIDE. Apparently there is also one more class of messages: Replies.

```
1   #!ruby
2   # lib/account_component/messages/replies/record_withdrawal.rb
3   module AccountComponent
4     module Messages
5       module Replies
6         class RecordWithdrawal
7           include Messaging::Message
8
9           attribute :withdrawal_id, String
10          attribute :account_id, String
11          attribute :amount, Numeric
12          attribute :time, String
13          attribute :processed_time, String
14        end
```

I haven't yet figured out what Replies are used for. It seems interesting.

Model

I wonder where is the logic for changing account balance or checking if the funds are sufficient for withdrawal. Let's find out.

```ruby
#!ruby
# lib/account_component/account.rb
module AccountComponent
  class Account
    include Schema::DataStructure

    attribute :id, String
    attribute :customer_id, String
    attribute :balance, Numeric, default: 0
    attribute :opened_time, Time
    attribute :closed_time, Time
    attribute :transaction_position, Integer

    def open?
      !opened_time.nil?
    end

    def closed?
      !closed_time.nil?
    end

    def deposit(amount)
      self.balance += amount
    end

    def withdraw(amount)
      self.balance -= amount
    end

    def current?(position)
      return false if transaction_position.nil?

      transaction_position >= position
    end

    def sufficient_funds?(amount)
      balance >= amount
    end
  end
end
```

Projection

It's interesting that even though the model is event sourced you don't see it when looking at it. Let's find the place responsible for rebuilding model state based on domain events.

```ruby
#!ruby
# lib/account_component/projection.rb
module AccountComponent
  class Projection
    include EntityProjection
    include Messages::Events

    entity_name :account

    apply Opened do |opened|
      account.id = opened.account_id
      account.customer_id = opened.customer_id

      opened_time = Time.parse(opened.time)

      account.opened_time = opened_time
    end

    apply Deposited do |deposited|
      account.id = deposited.account_id

      amount = deposited.amount

      account.deposit(amount)

      account.transaction_position = deposited.transaction_position
    end

    apply Withdrawn do |withdrawn|
      account.id = withdrawn.account_id

      amount = withdrawn.amount

      account.withdraw(amount)

      account.transaction_position = withdrawn.transaction_position
    end
```

```
38
39      # ...
```

File structure

So far we haven't looked at these files in `controls` directory. I wonder what's there.

```
1   |   ├── controls
2   |   |   ├── account.rb
3   |   |   ├── commands
4   |   |   |   ├── close.rb
5   |   |   |   ├── deposit.rb
6   |   |   |   ├── open.rb
7   |   |   |   └── withdraw.rb
8   |   |   ├── customer.rb
9   |   |   ├── events
10  |   |   |   ├── closed.rb
11  |   |   |   ├── deposited.rb
12  |   |   |   ├── opened.rb
13  |   |   |   ├── withdrawal_rejected.rb
14  |   |   |   └── withdrawn.rb
15  |   |   ├── id.rb
16  |   |   ├── message.rb
17  |   |   ├── money.rb
18  |   |   ├── position.rb
19  |   |   ├── replies
20  |   |   |   └── record_withdrawal.rb
21  |   |   ├── stream_name.rb
22  |   |   ├── time.rb
23  |   |   └── version.rb
24  |   ├── controls.rb
```

Controls

```ruby
#!ruby
lib/account_component/controls/commands/close.rb
module AccountComponent
  module Controls
    module Commands
      module Close
        def self.example
          close = AccountComponent::Messages::Commands::Close.build

          close.account_id = Account.id
          close.time = Controls::Time::Effective.example

          close
        end
```

```ruby
#!ruby
# lib/account_component/controls/events/withdrawn.rb
module AccountComponent
  module Controls
    module Events
      module Withdrawn
        def self.example
          withdrawn = AccountComponent::Messages::Events::Withdrawn.build

          withdrawn.withdrawal_id = ID.example
          withdrawn.account_id = Account.id
          withdrawn.amount = Money.example
          withdrawn.time = Controls::Time::Effective.example
          withdrawn.processed_time = Controls::Time::Processed.example

          withdrawn.transaction_position = Position.example

          withdrawn
        end
```

```ruby
#!ruby
# lib/account_component/controls/account.rb
module AccountComponent
  module Controls
    module Account
      def self.example(balance: nil, transaction_position: nil)
        balance ||= self.balance

        account = AccountComponent::Account.build

        account.id = id
        account.balance = balance
        account.opened_time = Time::Effective::Raw.example

        unless transaction_position.nil?
          account.transaction_position = transaction_position
        end

        account
      end

      module Closed
        def self.example
          account = Account.example
          account.closed_time = Time::Effective::Raw.example
          account
        end
      end
```

It looks to me like these are helpers that help you build exemplary data, maybe test data. Maybe some kind of builders.