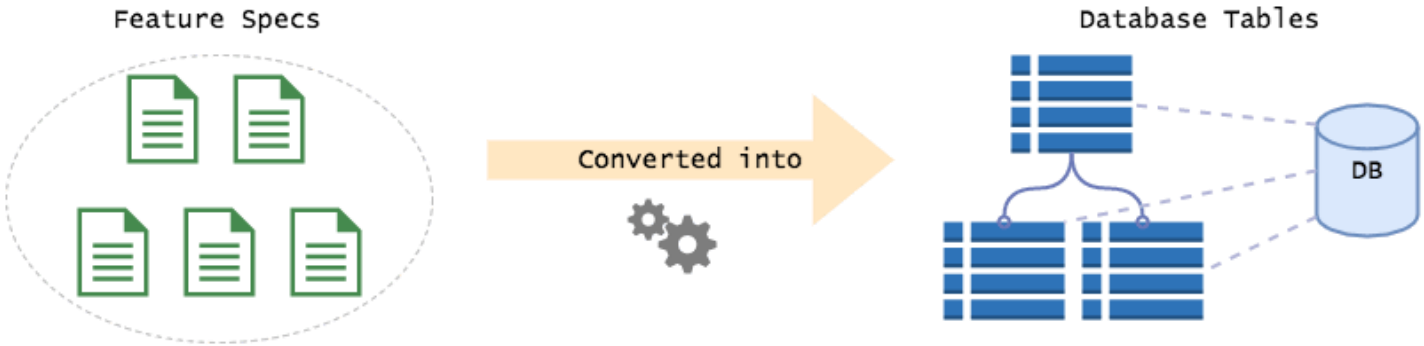
Event Sourcing, DDD & CQRS

Plan

1. Event Sourcing
2. Domain-driven design (DDD)
3. DDD vs Event Sourcing
4. Command Query Responsibility Segregation (CQRS)
5. Event Sourcing vs CQRS
6. Conclusion

Event Sourcing

Problematic



When we design web apps, we immediately translate the specs into concepts from our storage mechanism. If it's MySQL we design the tables, if it's MongoDB, we design the documents. This forces us to think of everything in terms of current state.

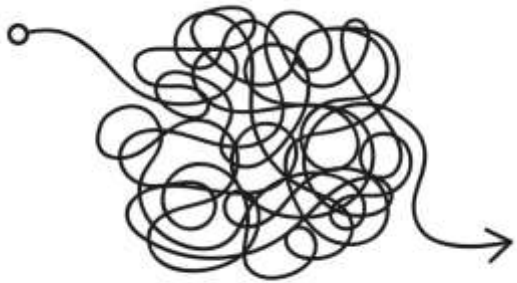
"How do I store this thing so I can retrieve (and potentially change) it later?".

Problems

1. It's not how we think

2. Single data model

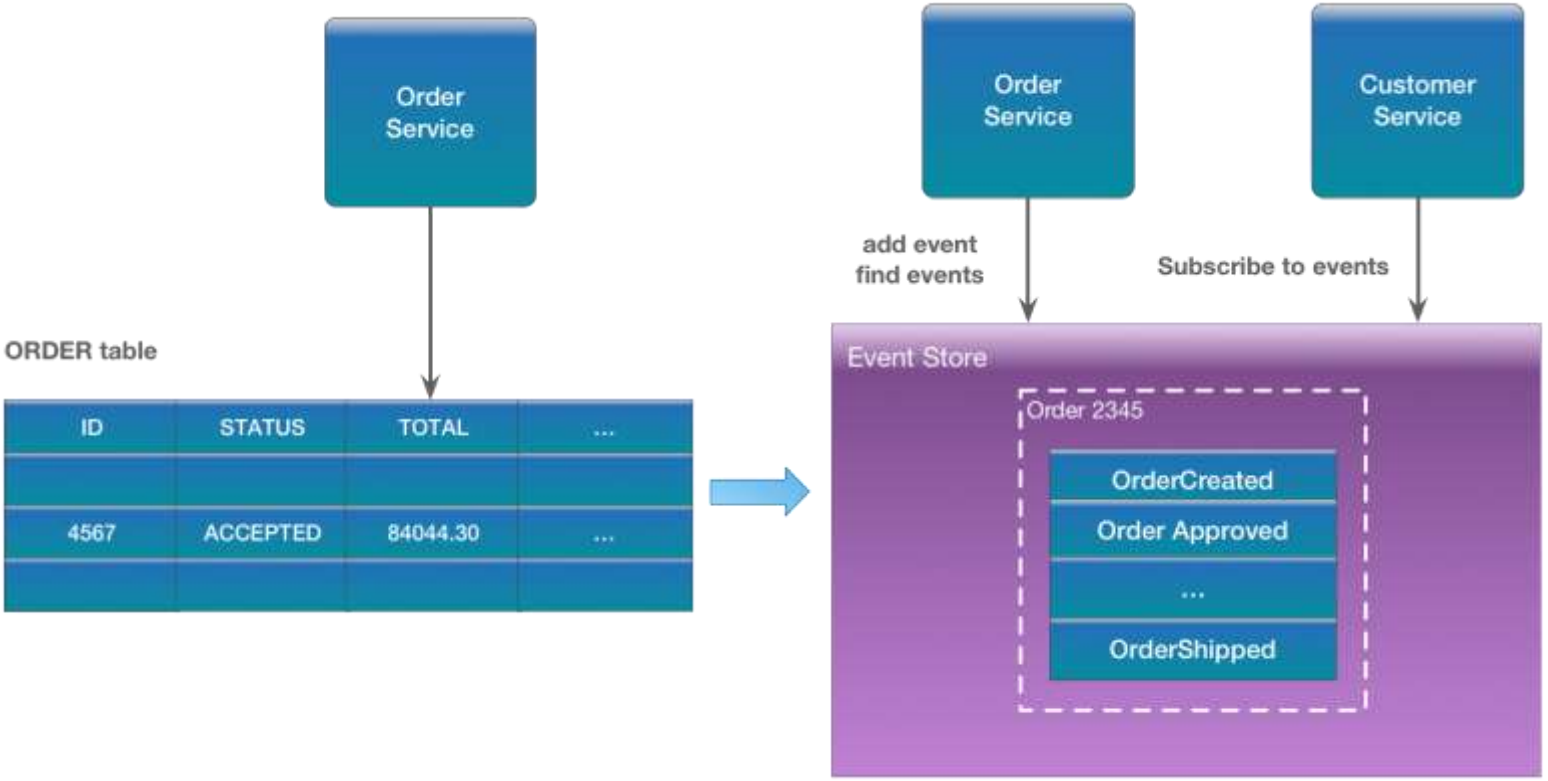
3. We lose business critical information



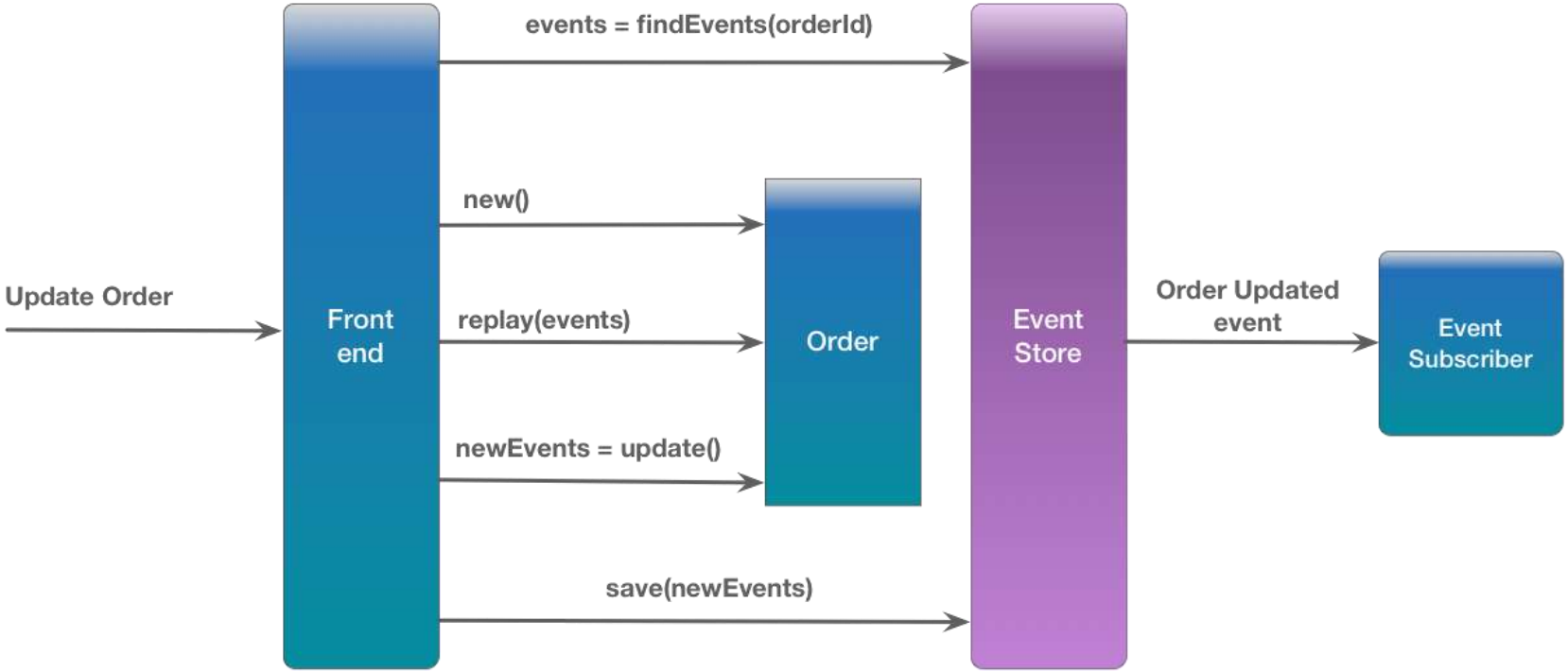
Event Sourcing (ES) is opposite of this.

Event
1. Shopping Cart Created
2. Item Added to Cart
3. Item Added to Cart
4. Item Removed from Cart
5. Shopping Cart Checked-Out

Example: Order Entity



How the Order Service handles a request to update an Order



- Event sourcing persists the state of a business entity such an Order or a Customer as a sequence of state-changing events
- Whenever the state of a business entity changes, a new event is appended to the list of events.
- The application reconstructs an entity's current state by replaying the events.
- Applications persist events in an event store, which is a database of events.
- .The store has an API for adding and retrieving an entity's events.
- The event store also behaves like a message broker.
- To optimize loading, an application can periodically save a snapshot of an entity's current state

How do you enforce business rules?

Isn't this expensive and time consuming?

What about showing data to the user?

If every piece of state is derived from events, how do you fetch data that needs to be presented to the user? Do you fetch all the events and build the dataset each time?

The benefits

1. Ephemeral data-structures.
2. Easier to communicate with domain experts
3. Expressive models
4. Reports become painless
5. Composing services becomes trivial
6. Lightning fast on standard databases
7. Easy to change database implementations

The issues

1. Eventual Consistency
2. Event Upgrading
3. Developers need deprogramming

Wrapping it up

Event Sourcing solves all the big problems our team has faced when building large scale distributed business software. It allows us to talk to the business in their language, and it gives us the freedom to change and adapt the system with ease. Throw in built in business analytics and you've got a winning combination. There is a learning curve, but once you get into Event Sourcing, you'll never want to look back, I know I don't.

Domain-driven design (DDD)

Problematic

When we face complex problems, we usually try to understand the individual pieces of complexity. By decomposing the problem, we turn it into more understandable and manageable pieces.

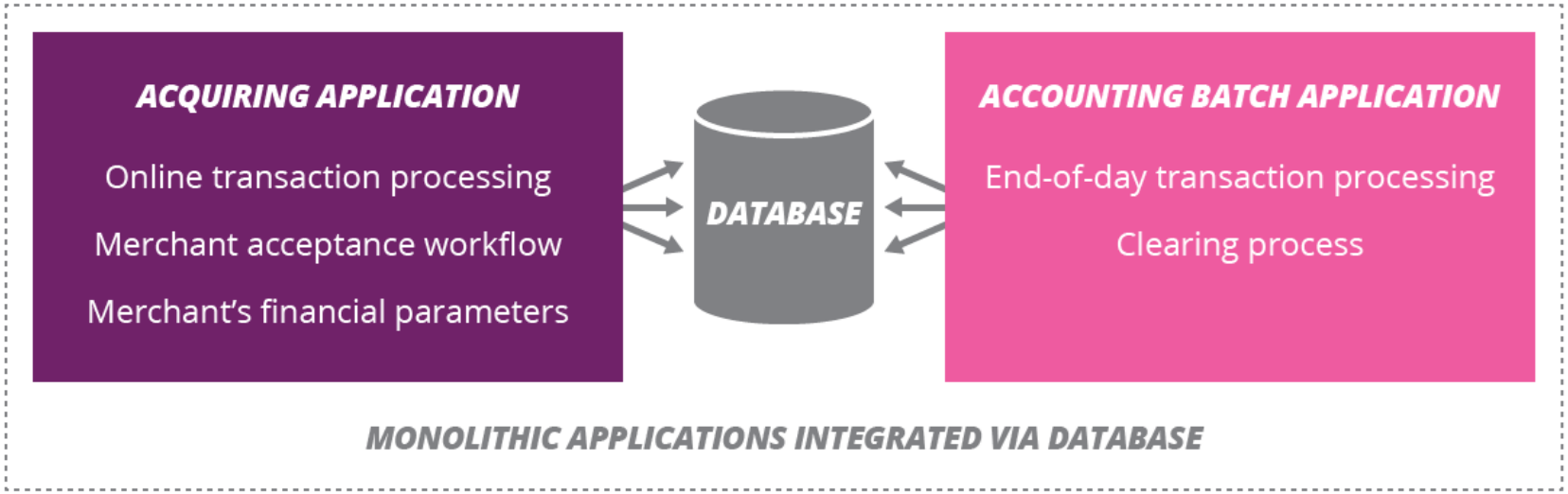
What is Domain-Driven Design?

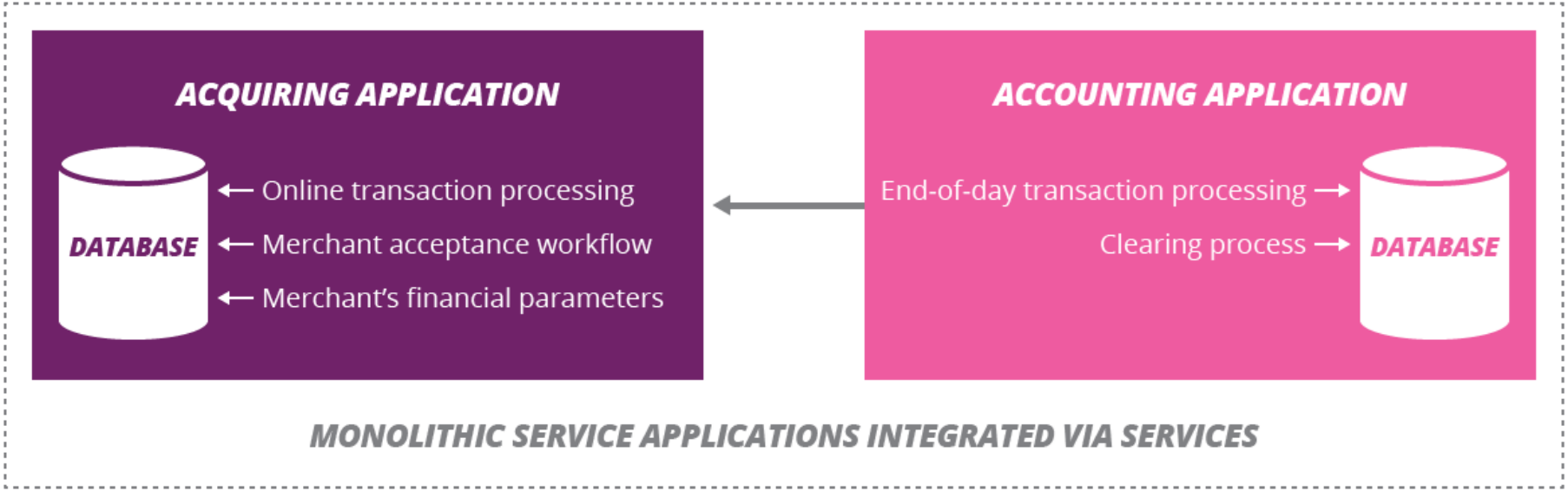
Domain-driven design (DDD) is an approach to developing software for complex needs by deeply connecting the implementation to an evolving model of the core business concepts.

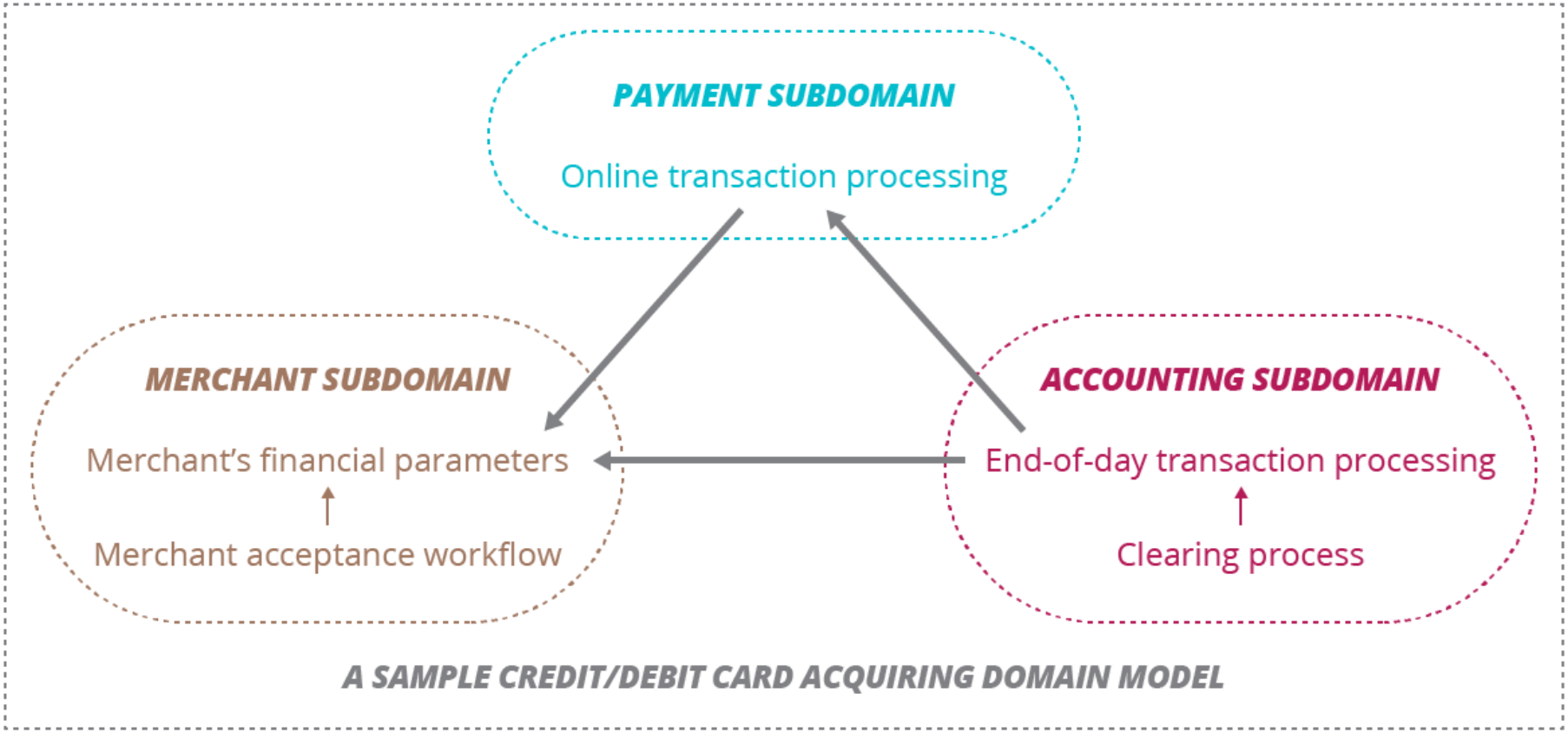
Its premise is:

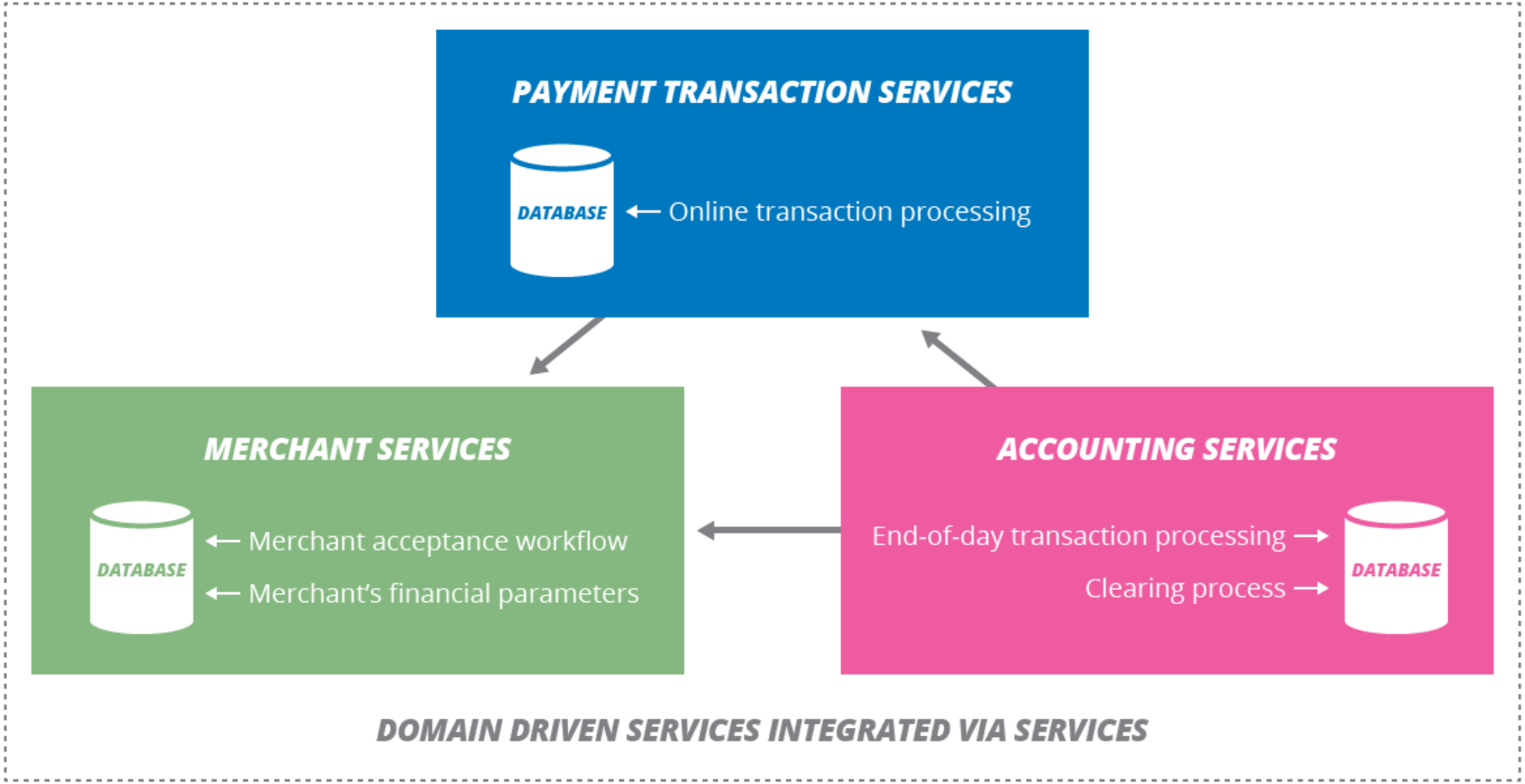
- Place the project's primary focus on the core domain and domain logic
- Base complex designs on a model
- Initiate a creative collaboration between technical and domain experts to iteratively cut ever closer to the conceptual heart of the problem.

Example: debit/credit card




ACQUIRING APPLICATION
DATABASE
Online transaction processing
Merchant acceptance workflow
Merchant's financial parameters
ACCOUNTING APPLICATION
End-of-day transaction processing
Clearing process
DATABASE
MONOLITHIC SERVICE APPLICATIONS INTEGRATED VIA SERVICES


PAYMENT SUBDOMAIN
Online transaction processing
MERCHANT SUBDOMAIN
Merchant's financial parameters
Merchant acceptance workflow
ACCOUNTING SUBDOMAIN
End-of-day transaction processing
Clearing process
A SAMPLE CREDIT/DEBIT CARD ACQUIRING DOMAIN MODEL


PAYMENT TRANSACTION SERVICES
DATABASE
Online transaction processing
MERCHANT SERVICES
DATABASE
Merchant acceptance workflow
Merchant's financial parameters
ACCOUNTING SERVICES
End-of-day transaction processing
Clearing process
DATABASE
DOMAIN DRIVEN SERVICES INTEGRATED VIA SERVICES

DDD common terms

- **Context:** The setting in which a word or statement appears that determines its meaning. Statements about a model can only be understood in a context.
- **Model:** A system of abstractions that describes selected aspects of a domain and can be used to solve problems related to that domain.
- **Ubiquitous Language:** A language structured around the domain model and used by all team members to connect all the activities of the team with the software.
- **Bounded Context:** A description of a boundary (typically a subsystem, or the work of a specific team) within which a particular model is defined and applicable.

Approach to architecture. Defining responsibilities and splitting project on logical layers

1. Application layer
2. Domain layer
3. Infrastructure layer

Beside from layers we have concepts of entities, value object, services and modules.

1. Entities
2. Value objects
3. Domain Event
4. Aggregate
5. Services
6. Repositories
7. Modules
8. Factories

Advantages of Domain-Driven Design

1. Eases Communication
2. Improves Flexibility
3. Emphasizes Domain Over Interface

Disadvantages of Domain-Driven Design

1. Requires Robust Domain Expertise
2. Encourages Iterative Practices
3. Suited for Highly Technical Projects

Wrapping it up

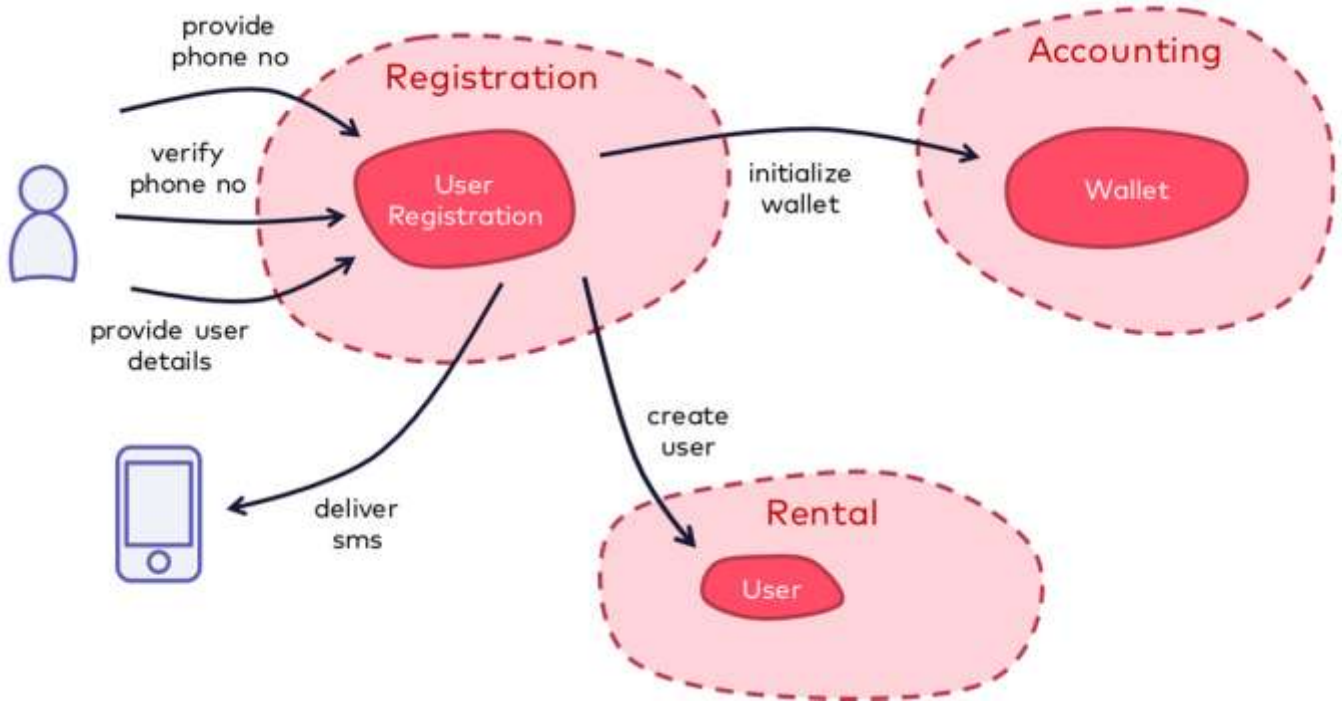
Main thing is decoupling business logic from rest of the system and creating healthy model. Creating healthy model consider a lot of talking with domain experts and understanding domain on level needed for creating successful software. We don't have to understand low level details about domain but we need to recognize purpose of the software and to shape model around domain knowledge.

DDD vs Event Sourcing

Why domain events and event sourcing should not be mixed up

Let's consider the following example: in a domain for bike sharing, a user wants to register in order to rent a bike. Of course one also has to pay for it, which is done through a pre-paid approach using a wallet.

Events from Event Sourcing ≠ Domain Events



The registration process works as follows:

- The user enters his/her mobile number through the mobile app.
- The user receives an SMS code to confirm the phone number.
- The user enters the confirmation code.
- The user enters the additional details such as full name or address and completes the registration.

If event sourcing is used

The following events (containing the corresponding relevant state) are created and persisted over time:

- MobileNumberProvided (MobileNumber)
- VerificationCodeGenerated (VerificationCode)
- MobileNumberValidated (no additional state)
- UserDetailsProvided (FullName, Address, ...)

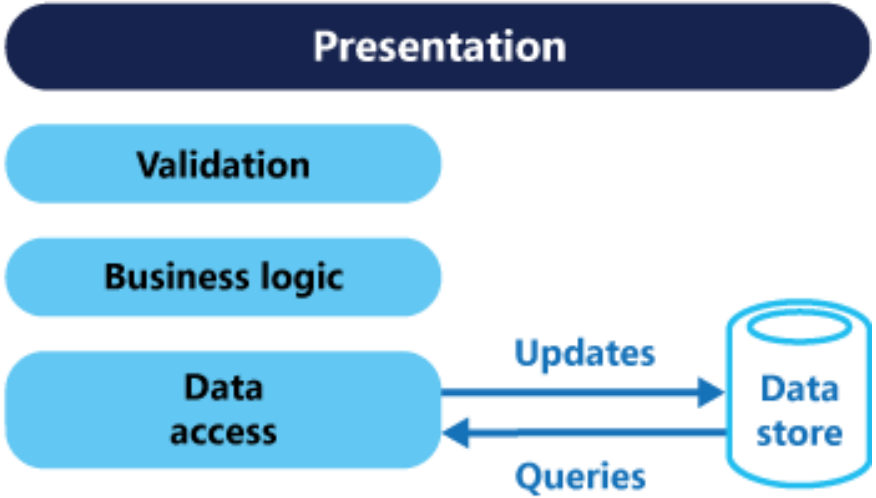
If this is to be done using the events from event sourcing, each consuming bounded context must

- process these fine-granular events and know at least parts of the domain logic from the UserRegistration aggregate (e.g. after which event a user is considered fully registered).
- combine several events to get the whole state required about the user (e.g. the phone number from MobileNumberProvided and additional details from UserDetailsProvided)
- ignore events which are not of any interest in the respective bounded context (e.g. VerificationCodeGenerated or MobileNumberValidated to confirm the phone number)

This approach breaks the intended encapsulation between different parts of the system, leads to chatty communication between bounded contexts and thus increases the coupling between bounded contexts. The main reason is that the semantics of the fine-grained events from event sourcing is too low-level, both in terms of the event itself and the associated information (the “payload”).

Command Query Responsibility Segregation (CQRS)

Problematic

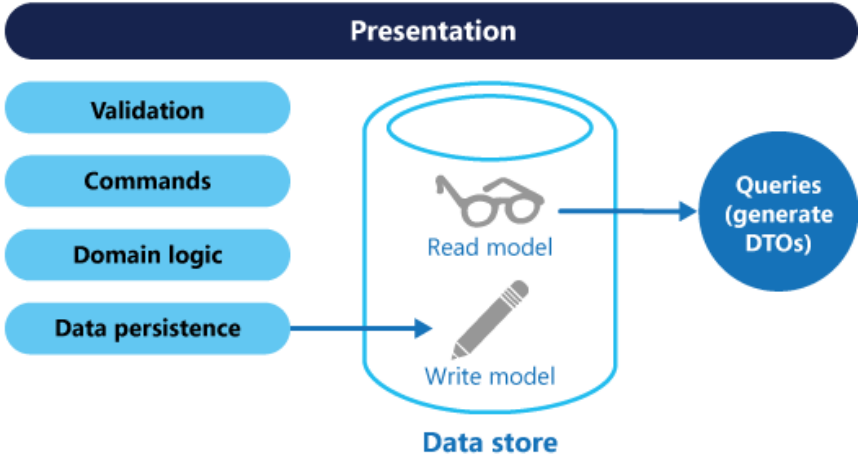


- There is often a mismatch between the read and write representations of the data, such as additional columns or properties that must be updated correctly even though they aren't required as part of an operation.

- Data contention can occur when operations are performed in parallel on the same set of data.

- The traditional approach can have a negative effect on performance due to load on the data store and data access layer, and the complexity of queries required to retrieve information.

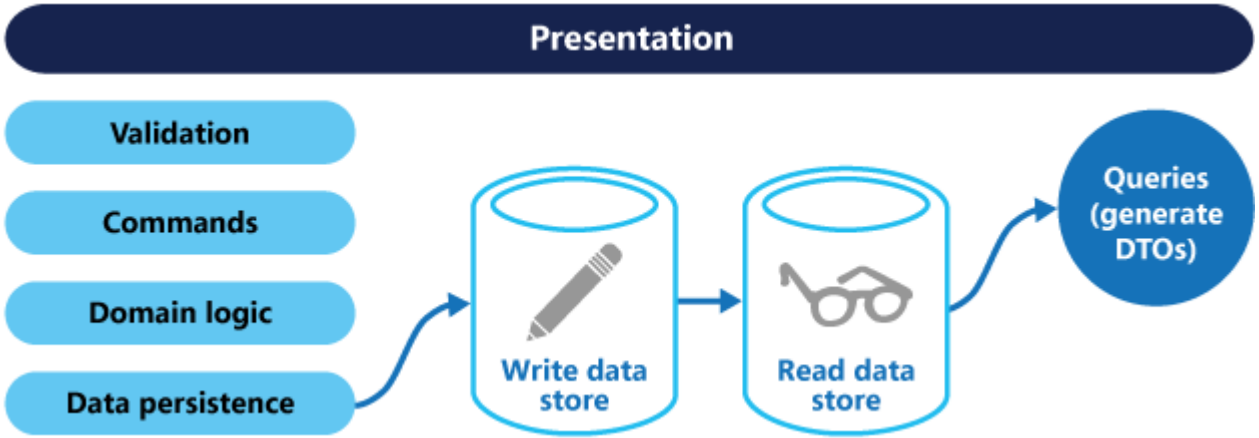
- Managing security and permissions can become complex, because each entity is subject to both read and write operations, which might expose data in the wrong context.



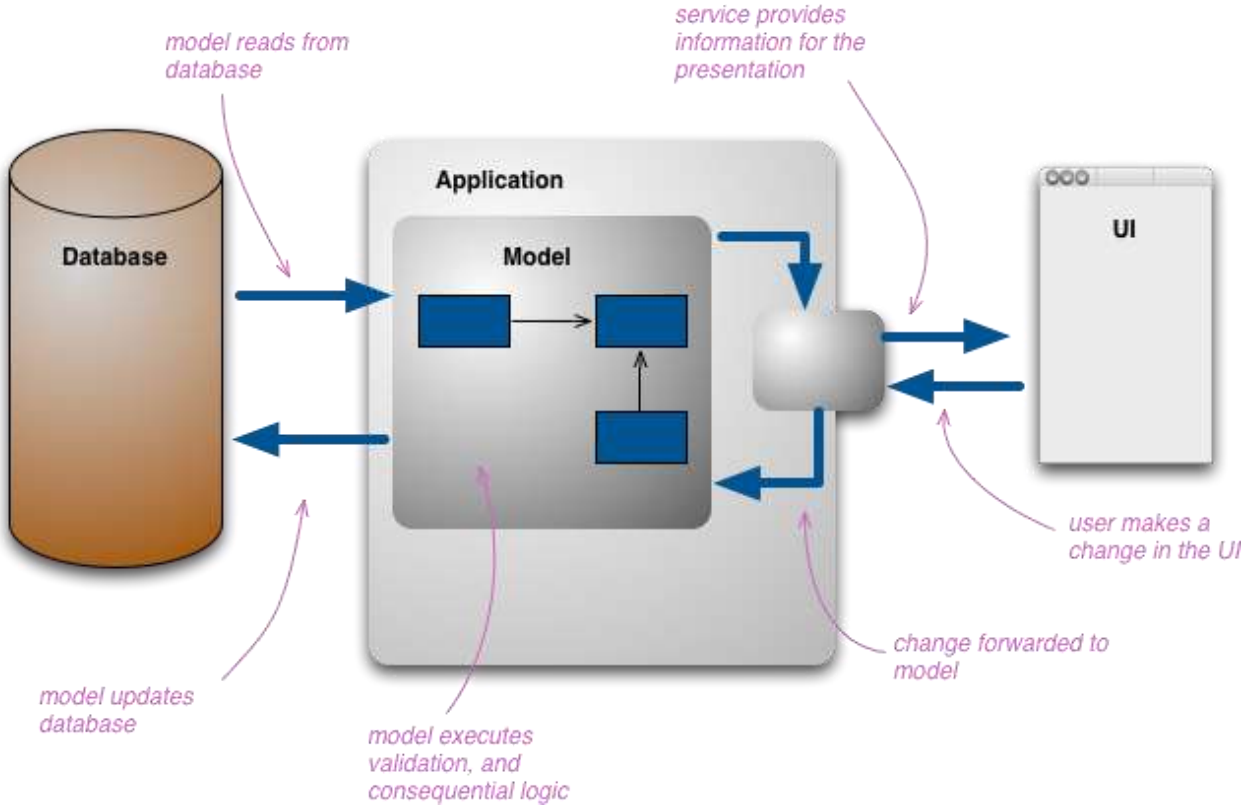
CQRS separates reads and writes into different models, using commands to update data, and queries to read data.

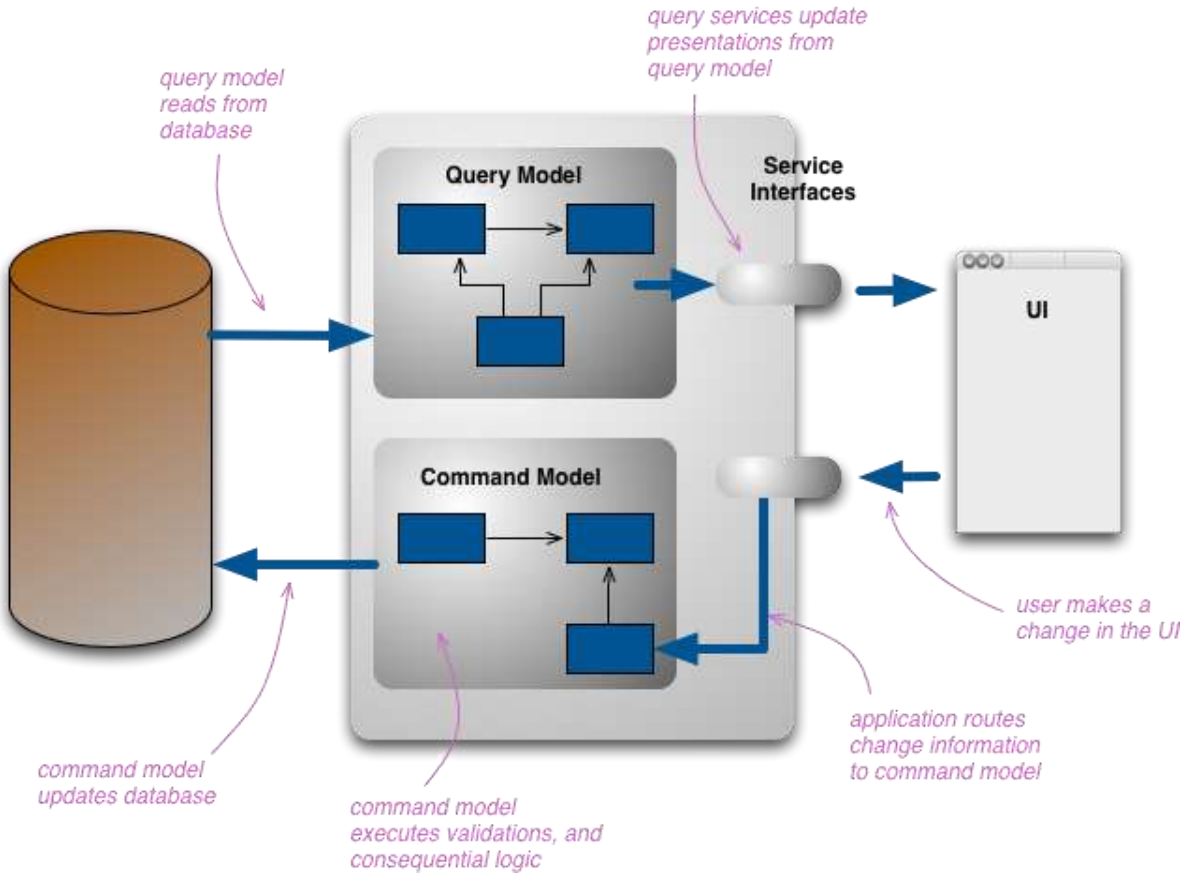
- Commands should be task based, rather than data centric. ("Book hotel room", not "set ReservationStatus to Reserved").
- Commands may be placed on a queue for asynchronous processing, rather than being processed synchronously.
- Queries never modify the database. A query returns a DTO that does not encapsulate any domain knowledge.

Separation of the read and write stores



More general




query services update
presentations from
query model
query model
reads from
database
Query Model
Service
Interfaces
UI
Command Model
user makes a
change in the UI
application routes
change information
to command model
command model
updates database
command model
executes validations, and
consequential logic

Benefits

1. Independent scaling
2. Optimized data schemas
3. Security
4. Separation of concerns.
5. Simpler queries

Issues and considerations

1. Complexity
2. Messaging
3. Eventual consistency

When to use this pattern

- Collaborative domains where many users access the same data in parallel.
- Task-based user interfaces where users are guided through a complex process as a series of steps or with complex domain models.
- Scenarios where performance of data reads must be fine tuned separately from performance of data writes, especially when the number of reads is much greater than the number of writes.
- Scenarios where one team of developers can focus on the complex domain model that is part of the write model, and another team can focus on the read model and the user interfaces.
- Scenarios where the system is expected to evolve over time and might contain multiple versions of the model, or where business rules change regularly.
- Integration with other systems, especially in combination with event sourcing, where the temporal failure of one subsystem shouldn't affect the availability of the others.

This pattern isn't recommended when:

- The domain or the business rules are simple.
- A simple CRUD-style user interface and data access operations are sufficient.

CQRS naturally fits with some other architectural patterns.

- As we move away from a single representation that we interact with via CRUD, we can easily move to a task-based UI.
- CQRS fits well with event-based programming models. It's common to see CQRS system split into separate services communicating with Event Collaboration. This allows these services to easily take advantage of Event Sourcing.
- Having separate models raises questions about how hard to keep those models consistent, which raises the likelihood of using eventual consistency.
- For many domains, much of the logic is needed when you're updating, so it may make sense to use EagerReadDerivation to simplify your query-side models.
- If the write model generates events for all updates, you can structure read models as EventPosters, allowing them to be MemoryImages and thus avoiding a lot of database interactions.
- CQRS is suited to complex domains, the kind that also benefit from Domain-Driven Design.

Wrapping it up

CQRS is an exciting architectural approach that demands an unusual handling of data. The separation of writing and reading might be familiar to the fewest developers, but makes sense in terms of scalability, modern asynchronous user interfaces, and the proximity to event sourcing and DDD.

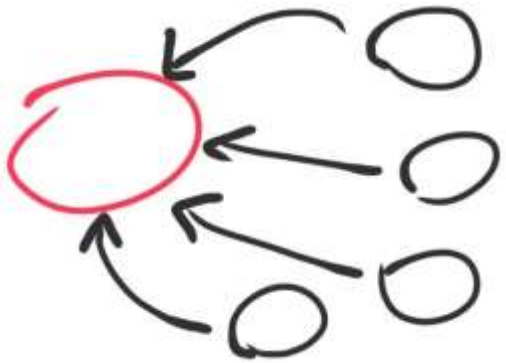
Nevertheless, CQRS is not the magic silver bullet that solves all problems. CQRS is particularly not suitable for small applications that do not require a high degree of scalability and that do not have complex domain logic, and for applications that have a direct impact on life or health, CQRS is not or only to a very limited extent suitable. Other approaches may be preferable here.

However, CQRS is ideal for most web and cloud applications: here, scalability is often an essential requirement for the software. In addition, much more is usually read than written, which speaks for the individual scalability of both sides. If you add event sourcing and DDD to CQRS, you have an excellent basis for the development of modern web and cloud applications.

Event Sourcing vs CQRS

So should events from event sourcing only be used within the corresponding aggregates?

Basically yes. A possible and meaningful exception, however, can be the use of these events in connection with read models in CQRS. Of course, this also has impacts encapsulation, but my experience shows that read models from CQRS are often anyway rather closely linked to an aggregate, since they provide a specific view on the data of that aggregate. Thus one may argue that the coupling resulting from the processing of the fine-granular events in the read model is acceptable.



Conclusion

References

- https://dev.to/barryosull/event-sourcing-what-it-is-and-why-its-awesome
- https://eventuate.io/whyeventsourcing.html
- https://medium.com/@hugo.oliveira.rocha/what-they-dont-tell-you-about-event-sourcing-6afc23c69e9a
- https://dddcommunity.org/learning-ddd/what_is_ddd/
- https://www.thoughtworks.com/insights/blog/domain-driven-design-services-architecture
- https://dev.to/stefanpavlovic94/domain-driven-design-first-thoughts-part-one-232f
- https://airbrake.io/blog/software-design/domain-driven-design
- https://www.innoq.com/en/blog/domain-events-versus-event-sourcing/#eventsourcingandcqrs
- https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs
- https://martinfowler.com/bliki/CQRS.html

Thank you!

Contact.OussamaZEBDA@gmail.com