Bounded Contexts With Axon

(/users/337575/idugalic.html) by Ivan Dugalic (/users/337575/idugalic.html) · Apr. 10, 19 · Microservices Zone (/microservices-news-tutorials-tools) · Tutorial

"A Context is the setting in which a word or statement appears, and that determines its meaning" - Airbrake.io (https://airbrake.io/blog/software-design/domain-driven-design)

A Bounded Context is an explicit boundary within which a domain model exists. The domain model expresses a Ubiquitous Language as a software model.

When starting with software modeling, Bounded Contexts are conceptual and are part of the 'problem space.' In this phase, the task it to try to find actual boundaries of specific contexts, and then to visualize the relationships between these contexts.

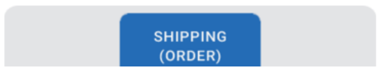
As the model starts to take on a deeper meaning and clarity, Bounded Contexts will transition to the 'solution space,' with the software model being reflected in the project source code.

From a run-time perspective, Bounded Contexts represent logical boundaries, defined by contracts within software artifacts where the model is implemented. In Axon applications, the contract (API) is represented as a set of messages (commands, events and queries) which the application publishes and consumes.

Bounded Contexts are a strategic concept in Domain-Driven Design (https://axoniq.io/resources/domain-driven-design), and it is important to know how it is reflected in the architecture and organizational/team structure.

Architecture

Let's introduce a sample subdomain of Shipping management which is responsible for managing courier information and also contains a courier view of an order (shipping) for managing the delivery of orders.



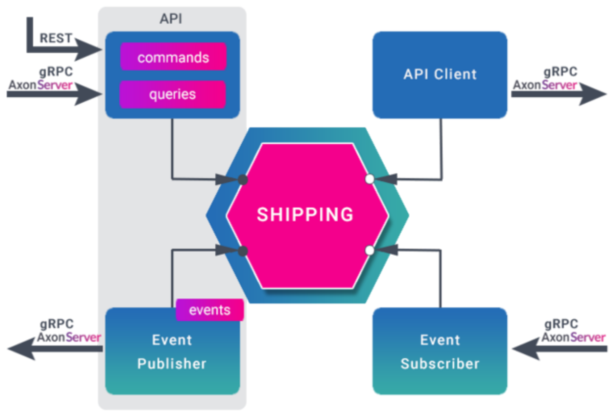
In Axon applications, the CQRS architectural pattern (https://axoniq.io/resources/cqrs) is used to decouple command components from the query side components. Additionally, these components are communicating via messages (commands, events, queries) in a location transparent manner. These components aren't interested in the actual destination of a message. It's simply a matter of configuration whether the system runs on a single node or is distributed on several nodes.

This design enables different deployment strategies with different scalability options, for example:

- Deploy all components within one service:
  - Shipping service
- Deploy command and query components separately (CQRS), as two independent services:
  - Shipping-Command service
  - Shipping-Query service
- Deploy command and query components as two services (CQRS), and extract their HTTP/REST adapters as services as well:
  - Shipping-Command service
  - Shipping-Command-REST service-adapter
  - Shipping-Query service
  - Shipping-Query-REST service-adapter

Shipping Service

The core of the service is it's business logic, which is surrounded by adapters that communicate with other services and applications. It has a hexagonal architecture (http://wiki.c2.com/?HexagonalArchitecture):

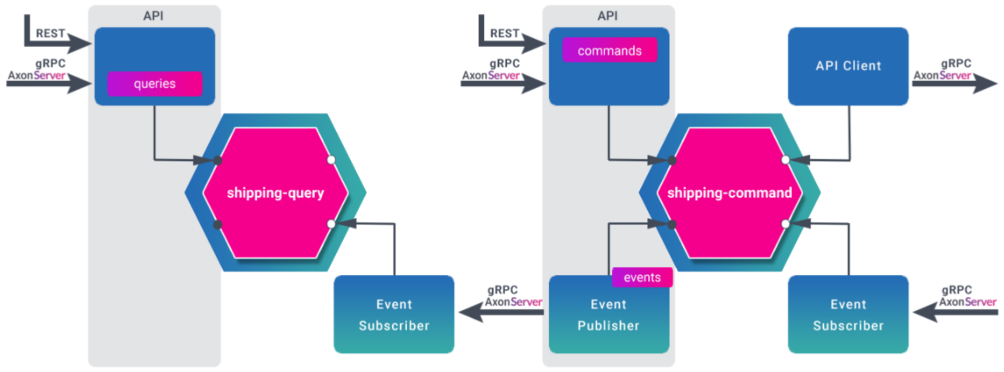


The API of this service is represented as a set of commands and queries it consumes and events it publishes. A layer of adapters (REST, WebSockets, gRPC, ...) maps the 'user friendly' API to the core messaging API of the service.

Shipping Query and Shipping Commandservices

Shipping-Query and Shipping-Commandservices

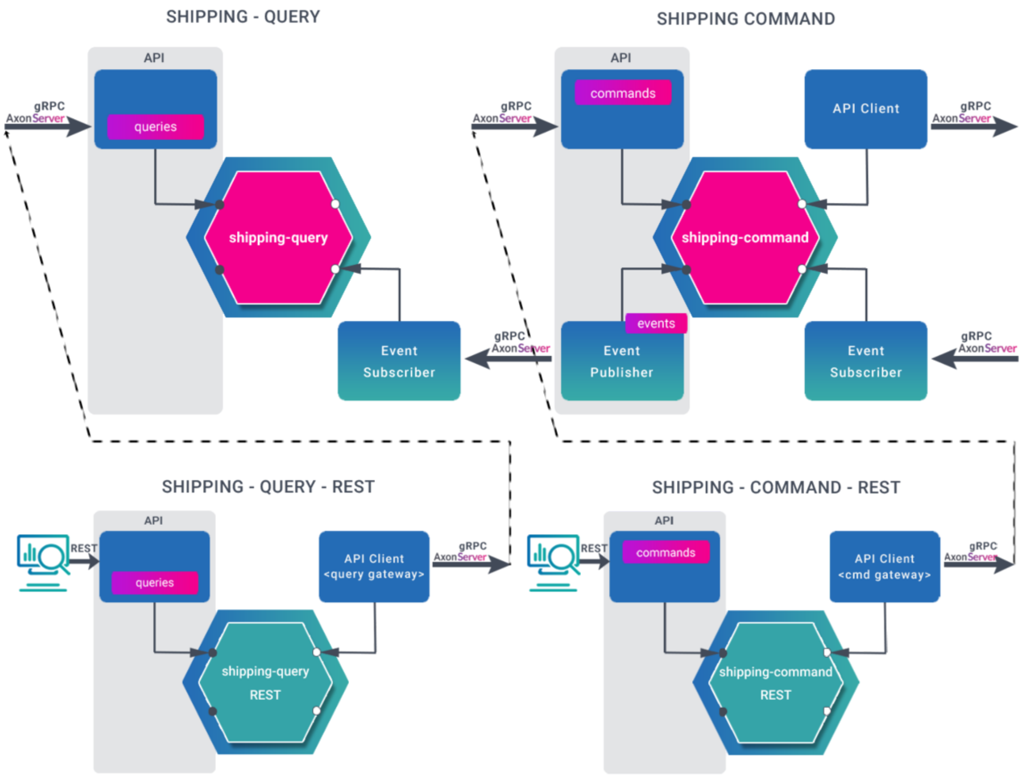
By applying CQRS architectural pattern the system can be further decouple by separating the command component from the query component of Shipping service and deploy them individually as two services:



The Shipping-Command service consumes commands only. The commands are handled by the business logic (aggregates), and domain events are published as a result. The Shipping-Query service is subscribed to these domain events, and it creates projections (materialized views) of the system's aggregates. These projections are optimized for querying, and Shipping-Query service is exposing 'queries' as an API.

Shipping-Query, Shipping-Commandand Adapters as Services

Location transparency allows for additional extraction of REST/HTTP adapters as services, and deploy them individually. This has enormous benefits for operational teams allowing them to release an update to one component without making other components unavailable.



These diagrams show the Hexagonal (Ports & Adapters) architecture (http://alistair.cockburn.us/Hexagonal+architecture) of three different options, with the service contracts (API) published as a schema. This generally means that if the events/commands/queries are published as JSON, or perhaps a more economical object format, the consumer can consume the messages by parsing them to obtain their data attributes.

The logical boundary of the Shipping context is represented as a set of messages we choose to consume (commands, queries) or publish (events). Inter-service communication is performed via gRPC through Axon Server (https://axoniq.io/product-overview/axon-server), which is an infrastructure component capable of routing these messages to interested services.

Whatever the choice of deployment, these services will speak the same language and belong to the same bounded context.

Organizational Structure

Inverting Conway's Law (http://www.melconway.com/Home/Conways_Law.html) allows for the organizational structure to align with the bounded contexts.

"Any organization that designs a system will produce a design whose structure is a copy of the organization's communication structure."

Therefore, there are a number of rules that should be followed:

- Explicitly set boundaries in terms of team organization.
- Keep the model strictly consistent within these bounds, and don't be distracted or confused by issues outside.
- Ideally, keep one subdomain model per one Bounded Context.

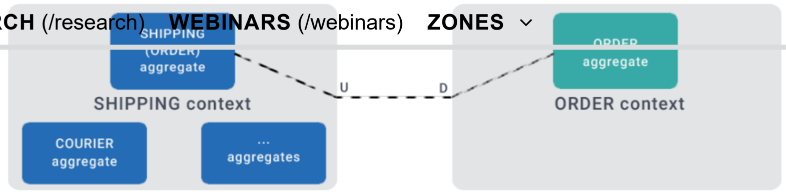
There should be a single team assigned to work on one Bounded Context. There should also be a separate source code repository for each Bounded Context. It is possible that one team could work on multiple Bounded Contexts, but multiple teams should not work on a single Bounded Context.

Context Mapping

A bounded context never lives entirely on its own. Information from different contexts will eventually be synchronized. It is useful to model this interaction explicitly. Domain-Driven Design names a few relationships between contexts, which drive the way they interact:

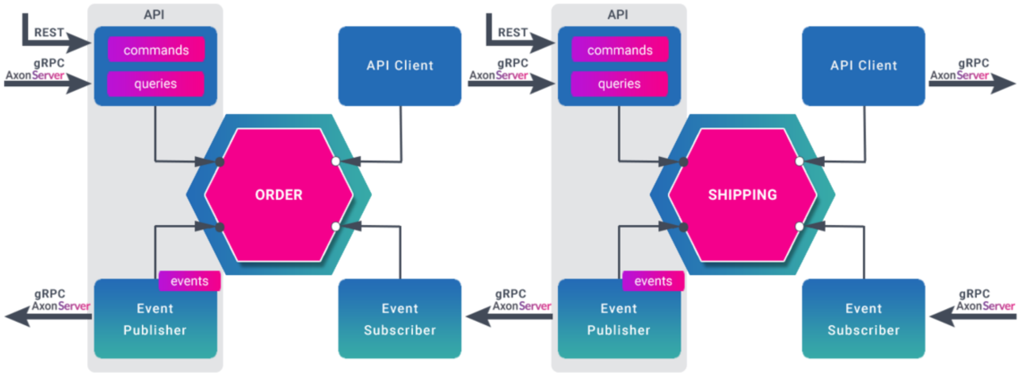
- *partnership* (two contexts/teams combine efforts to build interaction)
- customer-supplier (two contexts/teams in upstream/downstream relationship - upstream can succeed independently of downstream contexts)
- conformist (two contexts/teams in upstream/downstream relationship - upstream has no motivation to provide to downstream, and downstream context does not put effort in translation)
- shared kernel (explicitly, sharing a part of the model)
- separate ways (cut them loose)
- *anti-corruption layer* (the downstream context/team builds a layer to prevent upstream design to 'leak' into their own models, by transforming interactions)

Let's introduce the second sample subdomain: Order management (order taking and fulfillment process).



The Order and Shipping aggregate class in each subdomain model represent a different term of the same 'Order' business concept. Shipping's version of an Order simply consists of a status and address, which tell the courier how and where to deliver the order.

These two contexts are in the Upstream-Downstream relationship where the Order service (downstream) depends on the API of the Shipping service (upstream). For example, the Order service is responsible for the order fulfilment process and it will trigger a `command` to the Shipping service to create a Shipping 'Order.' Once the courier delivers the shipping/order, the Order service will receive an event from the Shipping service and will continue with the order fulfilment process. It is important to note that the Order service has a dependency on the Shipping service and not the other way around. If restricted to using events only (https://axoniq.io/blog-overview/blog-its-not-just-about-events-frans-van-buul) (no commands or queries) the services would become programmatically interdependent, which would introduce tight coupling.



More specifically, these two contexts are in the customer-supplier relationship.

To align with Convoy's Law we should organize two teams to produce these bounded contexts.

The teams define automated acceptance tests which validate the interface the upstream team provides. The upstream team can then make changes to their code without fear of breaking something downstream. This is where a Consumer Driven Contract (https://www.martinfowler.com/articles/consumerDrivenContracts.html) test comes into play. This test is part of our domain model and it reflects the consumer-supplier relationship in the source code.

When two contexts with an upstream/downstream relationship are not in a cooperative environment, a pattern such as customer-supplier is not going to work. In this case the downstream team builds an anti-corruption layer (independently deployable Axon application/component) to prevent the upstream design from 'leaking' into their own models, by transforming interactions.

Conclusion

Applying concepts from Domain-Driven Design will enable us to design our domain model effectively.

Axon deals with Bounded Contexts in a few different ways. From the Axon Framework (https://axoniq.io/product-overview/axon-framework) perspective, by separating business logic from the configuration. This allows logic to focus on the relevant aspect of the context itself by using configuration of serializers, upcasters, etc., to explicitly define how messages and interactions are shared beyond the boundaries of the context.

Additionally, the Axon Server (https://axoniq.io/product-overview/axon-server) (Enterprise) explicitly supports bounded contexts, by allowing different (groups of) applications, to connect to different contexts within the Axon Server. Unless specifically indicated otherwise, contexts are strictly separated and information/messages are not shared between them.

The Axon Framework is an open source Java framework for event-driven Microservices and Domain Driven Design. Axon Server is a zero-configuration message router and event store for Axon based application. You can download the quick start package here (https://axoniq.io/download).

Topics: DOMAIN DRIVEN DESIGN, BOUNDED CONTEXTS, MICROSERVICES, AXON, EVENT DRIVEN ARCHITECTURE

Microservices Partner Resources