Clean Architecture

Introduction

Name: Christian Wrobel

Job Role: Lead Architect & Team Lead at EnBW

Skills / Tech Stack:

- **Software Architecture:** Applying architecture methods such as Quality Storming, Risk Analysis, and Architecture Evaluation to design scalable and maintainable systems
- **Domain Driven Design:** Creating business-focused and adaptable software solutions
- **Software Engineering:** Developing distributed, cloud-native systems using .NET and Java

Disclaimer

When it comes to building software architecture, there are no definitive ‘right’ or ‘wrong’ solutions. Likewise, there is no silver bullet - a solution with only advantages and no downsides. Every software architecture decision involves trade-offs and should be based on functional and non-functional requirements, such as quality scenarios

The architecture I will present is just one of many ways to implement a clean architecture in a large, distributed enterprise environment. While some aspects might also work in other environments, others may need to be adapted to better suit specific requirements.

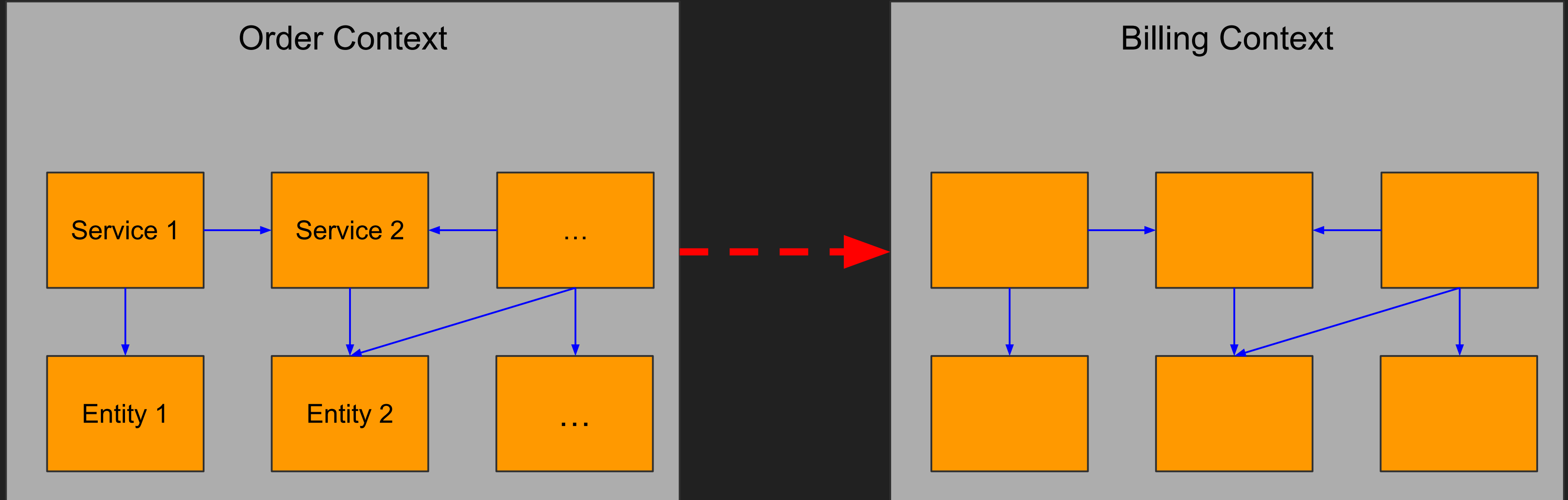
Business-Driven Architecture in Bounded Contexts

Many developers understand the importance of having a strategic architecture.

Domain-Driven Design and identifying the right Bounded Context are widely recognized and applied.

However, within a bounded context, establishing a well-modularized, business-driven architecture is equally essential.

Typical Architecture Landscape



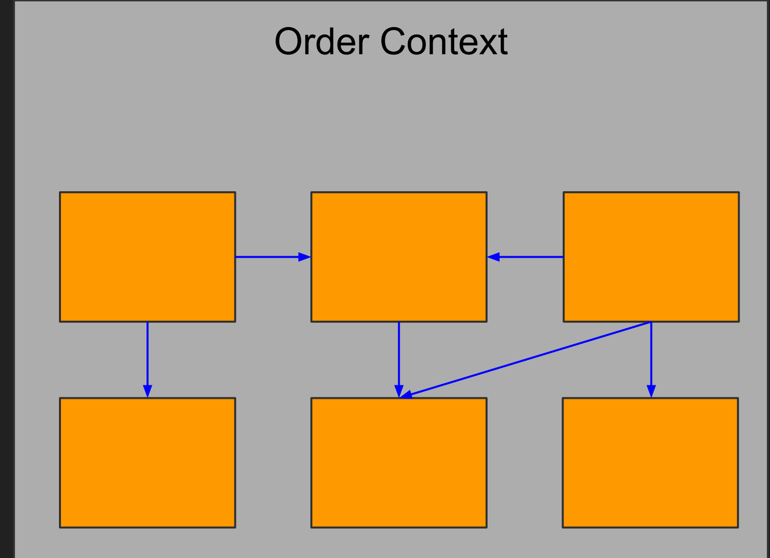
Unstructured Bounded Context

In an unstructured Bounded Context, there is no clear organization for business-driven modules.

This design tend to become a big ball of mud.

Sometimes, the architectural boundaries of a Bounded Context may need to be redefined. Without an encapsulated architecture, it becomes challenging to move modules to a different Bounded Context.

A Bounded Context may have poor testability if its design lacks proper structure and modularity.



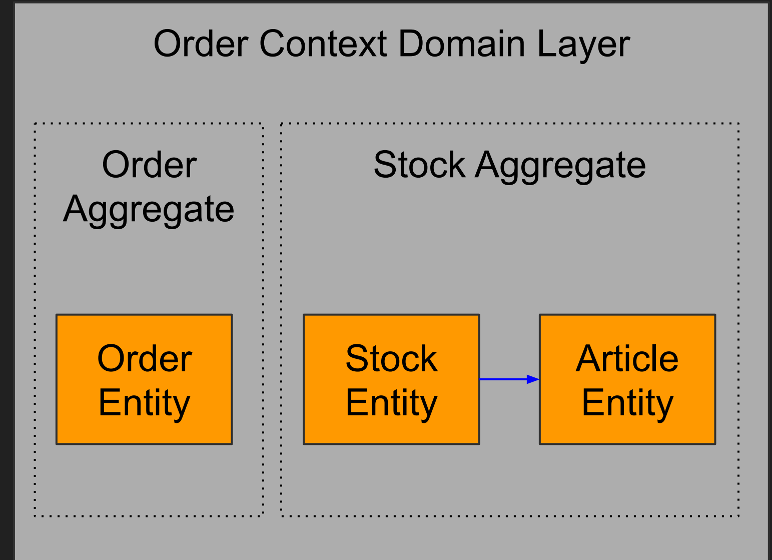
Domain Driven Bounded Context

It's essential to implement a business-driven design within a Bounded Context.

Domain-Driven Design helps structure a Bounded Context into well-defined business modules.

The domain layer of a Bounded Context is organized into Aggregates, where each Aggregate serves as a Unit of Work and can contain multiple Domain Entities and Value Objects.

The Entities represent the current state of an Aggregate and implement their own business logic.

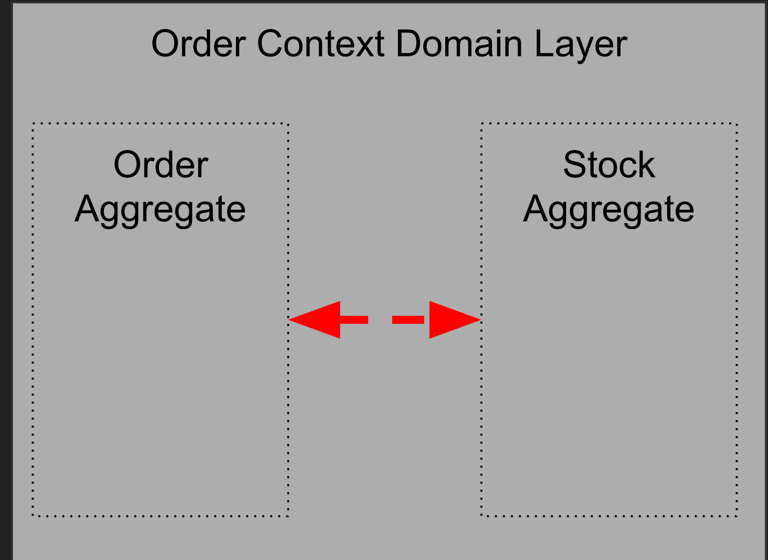


Communication between Aggregates

Aggregates are loosely coupled and communicate with each other through Domain Events.

Domain Events exist within the transaction scope of a Bounded Context.

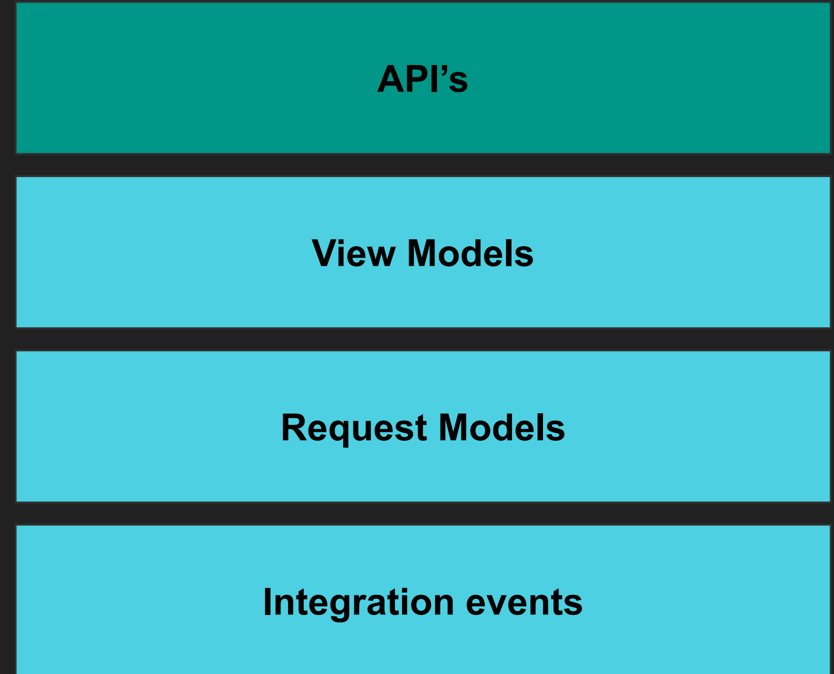
Several libraries, such as MediatR in .NET and Spring Boot in Java, provide support for Domain Events.



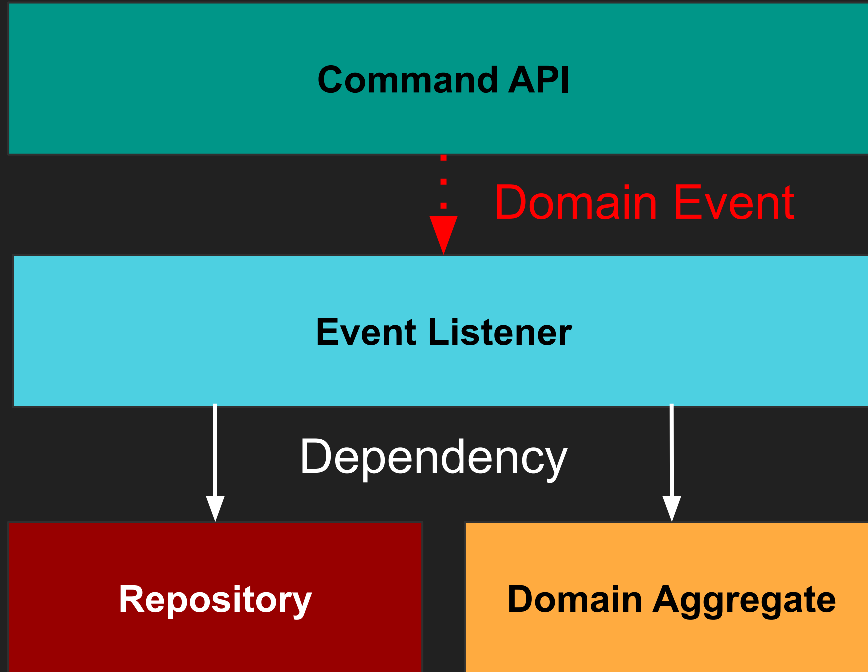
Application Layer

The Application layer is responsible for exposing APIs to the outside world, such as RESTful APIs or handlers for Integration Events.

To protect incoming and outgoing communication from breaking changes, the Application Layer manages tasks such as API contract versioning and mapping them to the internal Domain Layer.



Code Example - Commands



The Command API is loosely coupled through Domain Events

The event listener orchestrates Domain Aggregates with their Repositories

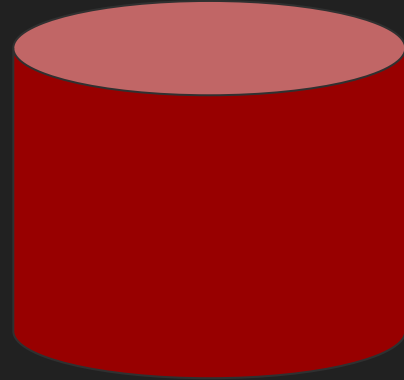
This design is flexible: the Event Listener, Repository, and Domain Aggregate can be moved to other bounded contexts and used with different protocols, like AMQP integration events instead of a HTTP command API

The business logic is well encapsulated within the Aggregates

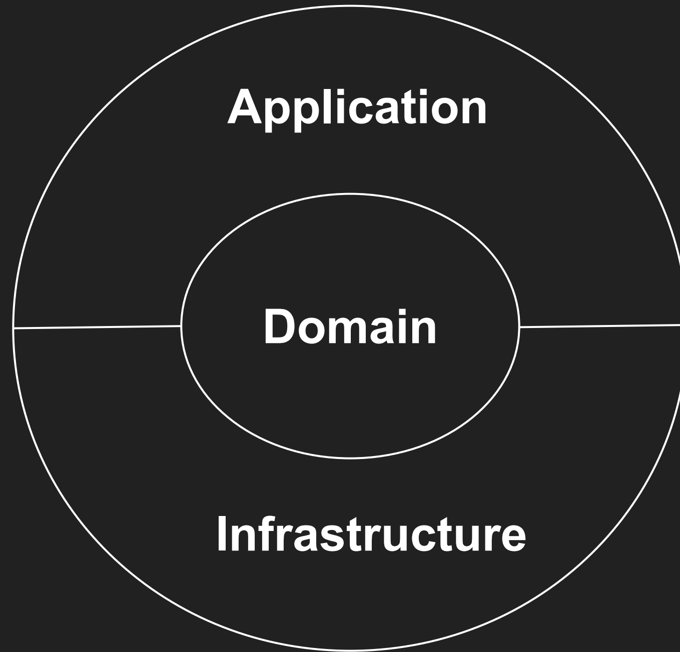
An Aggregate can contain multiple Domain Entities and Value Objects.

Infrastructure Layer

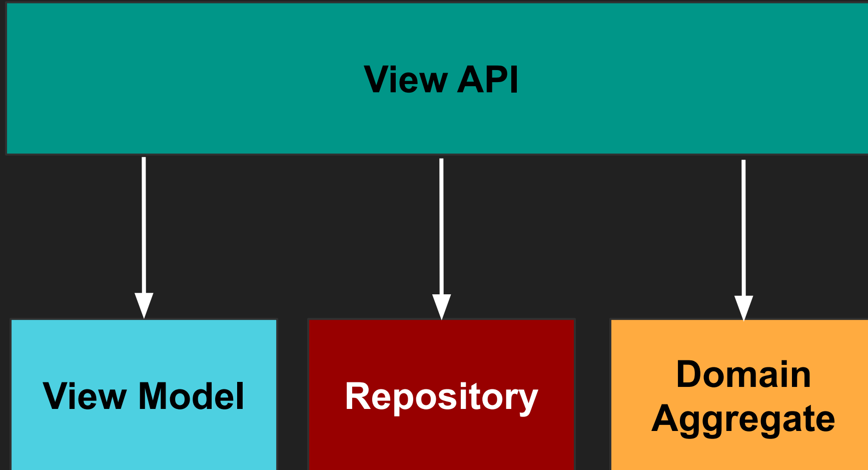
The Infrastructure Layer is responsible for infrastructure tasks such as loading and persisting data, publishing integration events, and handling logging and monitoring.



Lightweight Onion Architecture



Code Example - Views



The View API loads domain aggregates and maps them to view models

Multiple repositories and aggregates can be combined into a single view model, depending on the requirements of the View API

From a modularization perspective, this approach makes sense when the read database is separated from the write database (see CAP theorem for advantages and disadvantages)

Even when the databases are not split, this design can serve as a good foundation and is flexible for future database separation if needed

The API and View Model are versioned. The API controller maps the current domain model to its corresponding version of View Models

Example Usecase

1. Customer make order
2. Stock get reduced with ordered articles

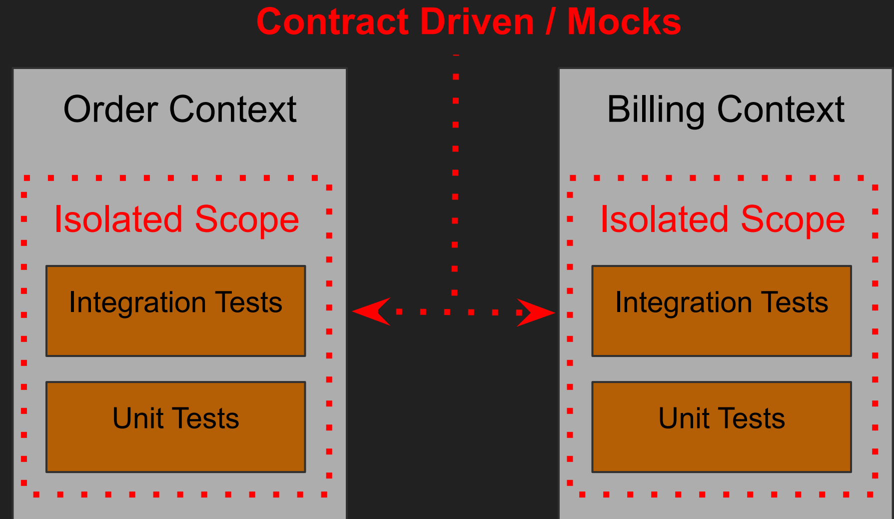
Code Example

Open

Testing Strategy

Within a Bounded Context, Unit and Integration tests are commonly effective.

Tests spanning multiple Bounded Contexts should be avoided, as they risk creating a shared monolith. Instead, contract-driven testing or the use of mocks are better alternatives.



Clean Testing Architecture

For better maintainability and readability, test data should be structured using personas.

The builder pattern can be used to create these personas, which can then be reused across multiple test cases.

Ideally, personas should be defined and utilized in the requirements, such as user stories.

Name	Mickey Mouse
Address	Main Street 1, 12345 Disneyland
Age	96
Role	Iconic Cartoon Character / Entertainer

Code Example

Open

Creating the Domain Model

A great starting point is **event storming**. These sessions, ideally conducted cross-functionally with business stakeholders, help identify key business events and shape aggregates, entities, and value objects, forming the foundation for software architecture. For larger changes, it’s beneficial to repeat these sessions iteratively, as new requirements lead to evolving business events and refined domain models.

Common Pitfalls and Bad Practices to Avoid

Reading Data from a Domain Model:

To read data, aggregations often span multiple aggregates. Instead of coupling aggregates, it's better to use view models. For scalability and performance, separating the write and read databases can be beneficial if eventual consistency is acceptable.

Aggregates can become too large:

A common pitfall in a bounded context is when an aggregate grows too large, leading to a tangled, unmanageable structure. Aggregates should be designed around meaningful business events, not just in a static manner.

Stop working on the domain model:

In the early phase of new projects, teams often collaborate cross-functionally to design the initial version of the domain model. However, as they move on to develop additional features, they frequently stop refining the model. This is often the point where the software design begins to weaken.

Trade-offs

Readability of Code:

In event-driven, modular software, a trade-off is the complexity of following logic paths, as they are distributed across multiple modules and connected only by domain-driven events.

Domain Driven Design:

Teams often face challenges designing event-driven domain models and identifying the correct domain boundaries.

Architecture Testing

To protect architectural principles, it's useful to test them with Fitness Functions.

Frameworks like ArchUnit and ArchUnitNET can assist in testing the software architecture.

Fitness Function Examples:

- ➔ Testing layers with allowed and not allowed access
- ➔ Verify that each aggregate has no dependencies on other aggregates
- ➔ Test naming conventions for classes should follow consistent patterns, such as ensuring all repository classes end with *Repository
- ➔ Test the size of an aggregate to prevent it from becoming too large
- ➔ An event listener references at most one repository to maintain high cohesion

Feedback

Mentimeter

Contact and Further Informations

Blog:

www.jknowledge.de

Github:

https://github.com/jknowledge