**Domain Driven Design [DDD]**

Contents

[1 Document Control 3](#_Toc94610834)

[1.1 Change Record 3](#_Toc94610835)

[1.2 Reviewer 3](#_Toc94610836)

[1.3 Approver 3](#_Toc94610837)

[2 Document Purpose 3](#_Toc94610838)

[3 Domain Driven Design [DDD] 3](#_Toc94610839)

[4 Layers in DDD 4](#_Toc94610840)

[4.1 Application Layer 4](#_Toc94610841)

[4.1.1 Components Application Layer 4](#_Toc94610842)

[4.2 Domain Model Layer 4](#_Toc94610843)

[4.2.1 Components of Domain Model Layer 4](#_Toc94610844)

[4.3 Infrastructure Layer 5](#_Toc94610845)

[4.3.1 Components of Infrastructure Layer 5](#_Toc94610846)

[4.4 Layers Overview 6](#_Toc94610847)

[4.5 Dependencies between Layers 6](#_Toc94610848)

[5 Events 7](#_Toc94610849)

[5.1 Domain Events 7](#_Toc94610850)

[5.1.1 Raising Domain Events 8](#_Toc94610851)

[5.2 Integration Events 8](#_Toc94610852)

[6 Characteristics of Strong Domain Model 8](#_Toc94610853)

[7 Advantages of Domain Driven Design 9](#_Toc94610854)

[8 Disadvantages of Domain Driven Design 10](#_Toc94610855)

# Document Control

## Change Record

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Author** | **Version** | **Change reference** |
| 01/12/2021 | Prajeesh T S | 1.1 | Initial version |

## Reviewer

|  |  |  |
| --- | --- | --- |
| **Name** | **Role** | **Approval/Review Date** |
| **Anoop Jose** | Staff Software Architect |  |

## Approver

|  |  |  |
| --- | --- | --- |
| **Name** | **Role** | **Approval/Review Date** |
| **Anoop Jose** | Staff Software Architect |  |

# Document Purpose

Purpose of the document is to define the standards and general guidelines to define & implement Domain Driven Design.

# Domain Driven Design [DDD]

Domain-Driven Design (DDD) is a collection of principles and patterns that help developers craft elegant object systems. Properly applied it can lead to software abstractions called domain models. These models encapsulate complex business logic, closing the gap between business reality and code.

Domain-driven design (DDD) advocates modeling based on the reality of business as relevant to your use cases. In the context of building applications, DDD talks about problems as domains. It describes independent problem areas as Bounded Contexts (each Bounded Context correlates to a microservice), and emphasizes a common language to talk about these problems.

DDD patterns help you understand the complexity in the domain. For the domain model for each Bounded Context, you identify and define the entities, value objects, and aggregates that model your domain. You build and refine a domain model that is contained within a boundary that defines your context. And that is explicit in the form of a microservice.

# Layers in DDD

## Application Layer

DDD patterns help you understand the complexity in the domain. For the domain model for each Bounded Context, you identify and define the entities, value objects, and aggregates that model your domain. You build and refine a domain model that is contained within a boundary that defines your context. And that is explicit in the form of a microservice.

### Components Application Layer

* Commands and Handlers [ CQRS]
* Query and query handler [ CQRS]
* API Contracts and implementations
* Data transfer objects [DTO]
* Domain Event Handlers
* Integration Event
* Integration Event Handlers
* Validators

A microservice's application layer in .NET is commonly coded as an ASP.NET Core Web API project. The project implements the microservice's interaction, remote network access, and the external Web APIs used from the UI or client apps. It includes queries if using a CQRS approach, commands accepted by the microservice, and even the event-driven communication between microservices (integration events). The ASP.NET Core Web API that represents the application layer must not contain business rules or domain knowledge.

## Domain Model Layer

Responsible for representing concepts of the business, information about the business situation, and business rules. State that reflects the business situation is controlled and used here, even though the technical details of storing it are delegated to the infrastructure. This layer is the heart of business software.

This layer must completely ignore data persistence details. These persistence tasks should be performed by the infrastructure layer. Therefore, this layer should not take direct dependencies on the infrastructure, which means that an important rule is that your domain model entity classes should be POCO’s.

### Components of Domain Model Layer

* Domain Entities: POCO classes, construction
* Aggregate: The rules, computation, logic of domains, and related objects when updating the domain. According to Martin Fowler, an aggregate is a cluster of domain objects that can be treated as a single unit.
* Value objects: The value of an object related to Domain entities. In principle, Value Objects have no identity, and once been initialized, will not be modified. They can be understood as immutable classes.
* Interfaces: They help define business behaviours, etc. Other layers will be responsible for implementing these definitions.
* Domain Events
* Domain validations

## Infrastructure Layer

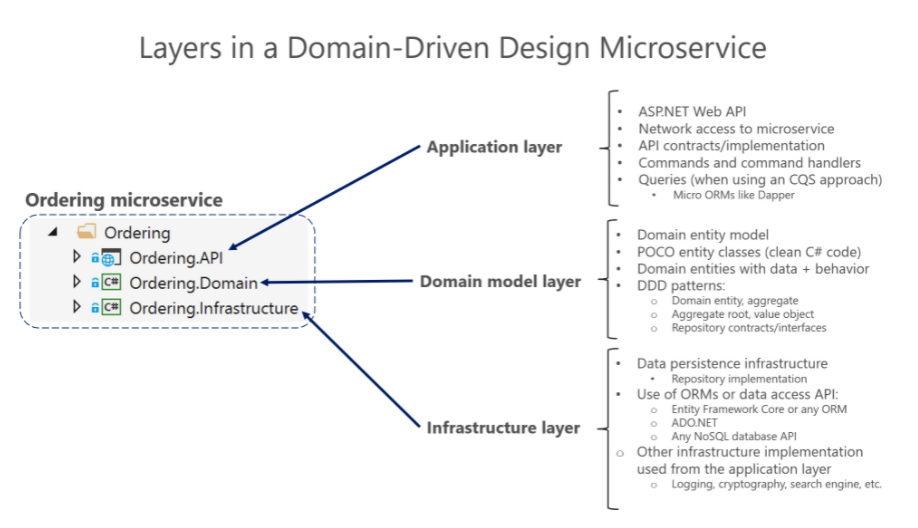
The infrastructure layer is how the data that is initially held in domain entities (in memory) is persisted in databases or another persistent store. An example is using Entity Framework Core code to implement the Repository pattern classes that use a DB Context to persist data in a relational database.

The domain model entity classes agnostic from the infrastructure that you use to persist data (EF or any other framework) by not taking hard dependencies on frameworks. The domain model layer class library should have only your domain code, just POCO entity classes.

### Components of Infrastructure Layer

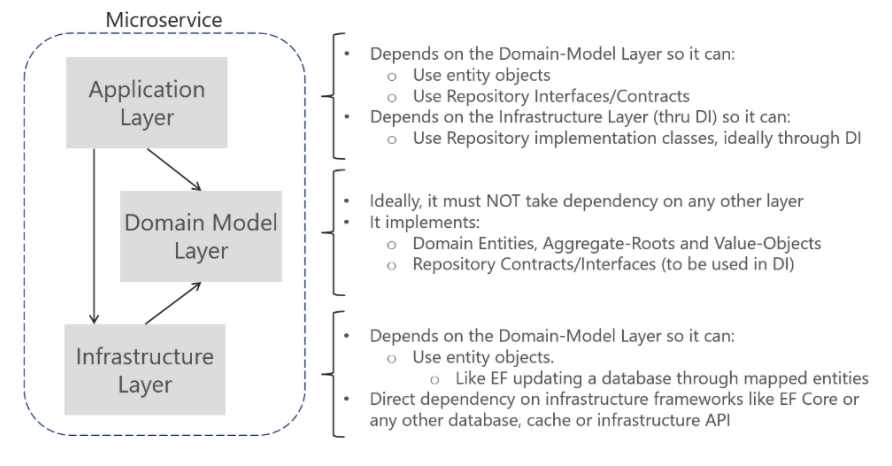
* Repositories: Repositories will be implemented here, including Generic Repository and <Entity> Repository.
* Data access: Contexts and the API connections link to databases.
  + SQL: ADO.NET, Entity Framework, Dapper, and ORM, etc.
  + In-Memory stores.
  + Caching, NoSQL, and so on.
  + Data seeding

## Layers Overview



## Dependencies between Layers

Dependencies in a DDD Service, the Application layer depends on Domain and Infrastructure, and Infrastructure depends on Domain, but Domain doesn't depend on any layer. This layer design should be independent for each microservice.



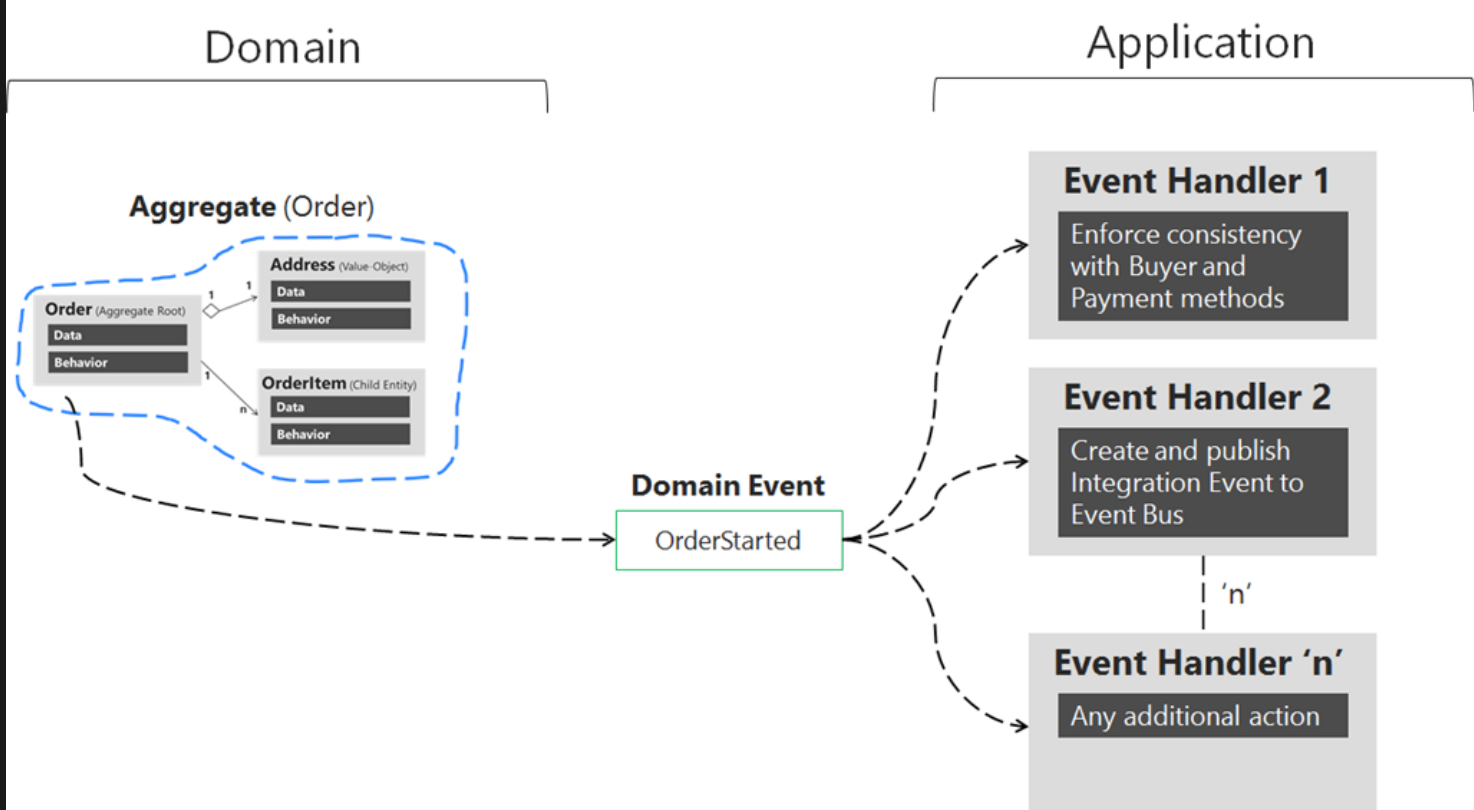
# Events

## Domain Events

A domain event is, logically, something that happened in a particular domain and you wish other parts of the same domain could be aware and react based of that. These events are used for in boundary communications.

Domain events are partially similar to messaging-style events, with one important difference. With true messaging, queuing and a service bus, a message is fired and always handled asynchronously and communicated across processes and machines. This is useful for integrating multiple bounded-contexts, microservices or even different applications. However, with domain events, you want to raise an event from the domain operation you are currently running but you want any side effects of the domain event to occur within the same domain.

Independently of the chosen implementation, the domain events and their side effects (the actions triggered afterwards that are managed by event-handlers) should occur almost immediately, usually in-process, and within the same domain.



### Raising Domain Events

Instead of dispatching to a domain event handler immediately, a better approach is to store/add the domain events in a collection and *right after or before*committing the transaction (like with Save Changes () in EF), dispatch those domain events.

Better to add/store the events happening in your entities into a collection or list of events per entity. That list would be part of the entity object, then raise the events after committing the transaction

## Integration Events

Integration events are part of the **public API contracts** that clients can subscribe to.

These could be published to subscribers using an Event Bus like RabbitMQ, or some HTTP call back method like Webhooks or SignalR.

We need to send asynchronous events to communicate and propagate changes from one original domain model (the original microservice or original bounded-context, for instance) to multiple subscribed microservices or even external subscribed applications

# Characteristics of Strong Domain Model

* **Being Aligned** with the business’ model, strategies, and processes.
* **Being isolated** from other domains and layers in the business.
* **Be loosely designed** with no dependencies on the layers of the application on either side of the domain layer.
* **Being reusable** to avoid models that are duplicated.
* **Be an abstract and cleanly separated layer** to create easier maintenance, testing, and versioning.
* **Minimum dependencies on infrastructure frameworks** to avoid outliving those frameworks and tight coupling on external frameworks.

# Advantages of Domain Driven Design

* **More flexibility -** As DDD is object-oriented, everything about the domain is based on and object is modular and caged. Thanks to this, the entire system can be modified and improved regularly. Since DDD is so heavily based around the concepts of object-oriented analysis and design, nearly everything within the domain model will be based on an object and will, therefore, be quite modular and encapsulated. This allows for various components, or even the entire system as a whole, to be altered and improved on a regular, continuous basis.
* Business Necessities Are Oriented - Developers communicate better with the business team and the work is more efficient when it comes to establishing solutions for the models that reflect how the business operates, instead of how the software operates.
* Extensible- The domain model is often modular and flexible, making it easy to update and extend as conditions and requirements change.
* Keeping Track Is Made Easier- If everyone is using the same terminology, it becomes quite simple to keep track of requirement implementation.
* Better Code- With DDD we end up with more readable code and less duplication.
* Agility is a Standard - By following an Agile approach that is iterative and incremental, DDD clarifies the mental model of domain experts into a useful model for the business.
* Communication Counts- Generally speaking, DDD comes in handy when it comes to helping the team creating a common model. The teams from the business’ side and from the developer’s side can then use this model to communicate about the business requirements, the data entities, and process models.
* Separation of Concerns – DDD guides us to divide and conquer, to focus on a single subdomain at a time, gather as many concepts from the domain experts as possible, and hopefully realize that each domain has its own tasks, terminology, and challenges.
* Testable. The domain model objects are loosely coupled and cohesive, allowing them to be more easily tested.

# Disadvantages of Domain Driven Design

* Requires Robust Domain Expertise:

Even with the most technically proficient minds working on development, it’s all for naught if there isn’t at least one domain expert on the team that knows the exact ins and outs of the subject area on which the application is intended to apply. In some cases, domain-driven design may require the integration of one or more outside team members who can act as domain experts throughout the development life cycle.

* **Low interactions**: The loose connection among different parts requires the team to communicate and exchange regularly. So before applying the DDD approach, the team needs to discuss its principles in detail first.