# What is DDD

* 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)
* It use Domain entities with rich models (no anemic-domain model), value objects, aggregates, and aggregate root (or root entity) rules to support the **internal implementation**
* DDD is an approach to business-focused software development
* DDD helps to solve the problem of building complex systems. This pattern requires architects, developers, and domain experts to understand precisely the requirements first. Then, they define behaviors, understand rules, apply principles and business logic into the set of clauses (Abstractions, Interfaces, and so on). Nowadays, DDD is set as a standard to develop different popular architectures, such as Onion Architecture, Clean Architecture, Hexagonal Architecture
* DDD is a great pattern for systems with complex business logic, systems that require future maintenance and enhancement.
* Code sample - <https://github.com/developer20sujeet/HRManagement>
* <https://github.com/developer20sujeet/ddd-guestbook>

## Advantages and disadvantages of Domain-Driven Design

### Advantages of DDD

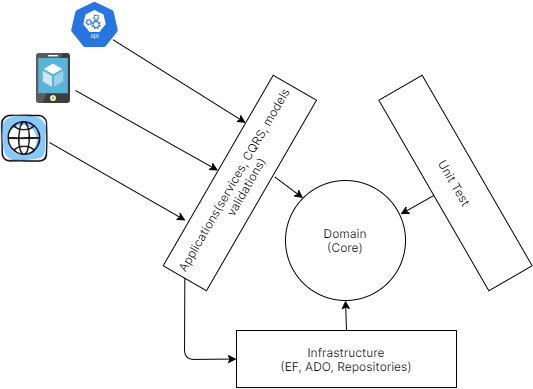
* **Loose coupling:** The parts of the system will interact with each other through the definitions and principles laid down in the Core layer **(interfaces, abstract classes, base classes, etc.**)
  + Implementations will be completed in the remaining layers. Setting up the implementation will be through DI
* **Flexibility** - The loose links and high-level definitions allow the team to enhance and adapt to new functional requirements more flexibly without considerable impact on the overall system.
* **Testability**: As mentioned above, separating the implementation from the interfaces defined in the Core layer, testing with mock data in a separate environment is allowed.
* **Maintenance**: DDD clearly divides functions among layers/tiers.
  + Specifically, the Domain implements business logic,
  + Infrastructure is in charge of data persistence,
  + and the Application handles API and integration logic. Following this approach ultimately gives you chances to write cleaner and more reliable codes.
  + Plus, your team can easily find code, limit its duplication and reduce maintenance time
* **Reduce Code duplication**

### Disadvantages of DDD

* **Domain expertise**: DDD requires extensive domain expertise. It means that your team needs to have at least one domain expert. They will help you define all of the processes, procedures, and terminology of that domain
* **Development costs**: Domain experts and the team have to implement a great deal of isolation and encapsulation within the domain model. This often results in a more extended development and duration that can come at a relatively high cost. Therefore, it is not well-suited for short-term projects or projects without a high domain complexity.

## Rich models (no anemic-domain model) in DDD

# Layers in DDD



* The architecture of DDD projects usually includes three main parts: **Domain, Infrastructure, Application**

## Domain

* **Domain**: A place to define logic concepts, principles, patterns, and **behaviors of data**, including **domain validation**, **calculations, and expressions** for system operations
  + **Entities**: POCO classes, construction, and model validation.
  + 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, ValueObjects have no identity,
    - and once been initialized, will not be modified.
    - They can be understood as immutable classes.
  + Interfaces: They help define business behaviors, etc.
    - Other layers will be responsible for implementing these definitions.
  + Repository Interfaces/ServiceBase:
    - The Interfaces of generic repositories, domain repositories, and services. Other layers will inherit and develop them.
  + ILogger/DTOs/Exceptions: Notifications and information are transferred to other services

## Infrastructure

* **Repositories**: Repositories will be implemented here, including GenericRepository and <Entity> Repository.
* Data access: Contexts and the API connections link to databases.
  1. SQL: ADO.NET, EntityFramework, Dapper, and ORM, etc.
  2. In-Memory stores.
  3. Caching, NoSQL, and so on.
  4. Data seeding
* Others:
* Logging.
* Cryptography.
* Etc

## Application

* Mobile application
* Web MVC/API application
* Desktop application
* IoT
* Others services

# What is Bounded Context DDD

# What is Value Object in DDD?

# What is ***aggregates in DDD***

# What is aggregate root (or root entity)

# What is Clean Architecture

# What is Microservice in DDD

# What is CQRS Pattern

* Command Query Responsibility Segregation (CQRS) is an architectural pattern for **separating** reading data (a 'query') from writing to data (a 'command').

# What is Mediator

# Why to use Auto mapper

# What is Cloud Native

## 12 Factor

* + A widely accepted methodology for constructing cloud-based applications is the Twelve-Factor Application
* The following table highlights the Twelve-Factor methodology:

| **THE TWELVE-FACTOR APPLICATION** | |
| --- | --- |
| **Factor** | **Explanation** |
| 1 - Code Base | A single code base for each microservice, stored in its own repository. Tracked with version control, it can deploy to multiple environments (QA, Staging, Production). |
| 2 - Dependencies | Each microservice isolates and packages its own dependencies, embracing changes without impacting the entire system. |
| 3 - Configurations | Configuration information is moved out of the microservice and externalized through a configuration management tool outside of the code. The same deployment can propagate across environments with the correct configuration applied. |
| 4 - Backing Services | Ancillary resources (data stores, caches, message brokers) should be exposed via an addressable URL. Doing so decouples the resource from the application, enabling it to be interchangeable. |
| 5 - Build, Release, Run | Each release must enforce a strict separation across the build, release, and run stages. Each should be tagged with a unique ID and support the ability to roll back. Modern CI/CD systems help fulfill this principle. |
| 6 - Processes | Each microservice should execute in its own process, isolated from other running services. Externalize required state to a backing service such as a distributed cache or data store. |
| 7 - Port Binding | Each microservice should be self-contained with its interfaces and functionality exposed on its own port. Doing so provides isolation from other microservices. |
| 8 - Concurrency | When capacity needs to increase, scale out services horizontally across multiple identical processes (copies) as opposed to scaling-up a single large instance on the most powerful machine available. Develop the application to be concurrent making scaling out in cloud environments seamless. |
| 9 - Disposability | Service instances should be disposable. Favor fast startup to increase scalability opportunities and graceful shutdowns to leave the system in a correct state. Docker containers along with an orchestrator inherently satisfy this requirement. |
| 10 - Dev/Prod Parity | Keep environments across the application lifecycle as similar as possible, avoiding costly shortcuts. Here, the adoption of containers can greatly contribute by promoting the same execution environment. |
| 11 - Logging | Treat logs generated by microservices as event streams. Process them with an event aggregator. Propagate log data to data-mining/log management tools like Azure Monitor or Splunk and eventually to long-term archival. |
| 12 - Admin Processes | Run administrative/management tasks, such as data cleanup or computing analytics, as one-off processes. Use independent tools to invoke these tasks from the production environment, but separately from the application. |

* In the book, [Beyond the Twelve-Factor App](https://content.pivotal.io/blog/beyond-the-twelve-factor-app), author Kevin Hoffman details each of the original 12 factors (written in 2011). Additionally, he discusses three extra factors that reflect today's modern cloud application design.

| **TABLE 3** | |
| --- | --- |
| **New Factor** | **Explanation** |
| 13 - API First | Make everything a service. Assume your code will be consumed by a front-end client, gateway, or another service. |
| 14 - Telemetry | On a workstation, you have deep visibility into your application and its behavior. In the cloud, you don't. Make sure your design includes the collection of monitoring, domain-specific, and health/system data. |
| 15 - Authentication/ Authorization | Implement identity from the start. Consider [RBAC (role-based access control)](https://docs.microsoft.com/en-us/azure/role-based-access-control/overview) features available in public clouds. |

* We'll refer to many of the 12+ factors in this chapter and throughout the book.

# 