agap2IT

TRAINING CENTER

Domain Driven Design
Module 3
Architecture and Event-Driven Modeling

Domain Driven Design

Module Overview

Modular Architecture

Architectures that supports DDD

Domain Driven Design

Modular Architecture

Eric Evans
DDD Author

“Most enterprise applications with significant business and technical complexity are defined by multiple layers. The layers are a logical artifact and are not related to the deployment of the service. They exist to help developers manage the complexity in the code. Different layers (like the domain model layer versus the presentation layer, etc.) might have different types, which mandate translations between those types”

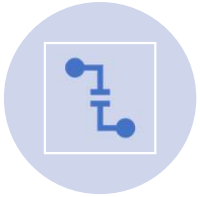
Modular Architecture Benefits

DDD PROMOTES MODULARITY BY DIVIDING THE SYSTEM INTO WELL-DEFINED, COHESIVE DOMAINS, EACH DEDICATED TO A SPECIFIC AREA OF THE BUSINESS. THIS MODULAR STRUCTURE ENHANCES CODE ORGANIZATION AND MAINTAINABILITY.

DDD PLACES A STRONG EMPHASIS ON CAPTURING AND MODELING THE BUSINESS DOMAIN WITHIN THE SOFTWARE, RESULTING IN A CLEARER UNDERSTANDING OF THE BUSINESS ITSELF. THIS ALIGNMENT BETWEEN THE SOFTWARE DESIGN AND THE BUSINESS CONCEPTS FOSTERS EFFECTIVE COMMUNICATION AND COLLABORATION.

DDD PROMOTES THE USE OF A SHARED, DOMAIN-SPECIFIC LANGUAGE (DSL) THAT IS COMPREHENSIBLE TO BOTH DOMAIN EXPERTS AND THE DEVELOPMENT TEAM. BY USING A COMMON LANGUAGE, DDD FACILITATES BETTER COMMUNICATION, REDUCES MISUNDERSTANDINGS, AND IMPROVES THE EXPRESSIVENESS AND READABILITY OF THE CODEBASE.

DDD ENCOURAGES THE CREATION OF BOUNDED CONTEXTS, WHICH ESTABLISH LOGICAL BOUNDARIES AROUND DOMAINS OR SUBDOMAINS. THESE BOUNDARIES ENABLE DEVELOPERS TO MAKE CHANGES WITHIN A SPECIFIC CONTEXT WITHOUT UNINTENDED REPERCUSSIONS ON OTHER PARTS OF THE SYSTEM. BOUNDED CONTEXTS AID IN REASONING ABOUT SYSTEM BEHAVIOR, LEADING TO MORE MAINTAINABLE AND EVOLVABLE ARCHITECTURES.



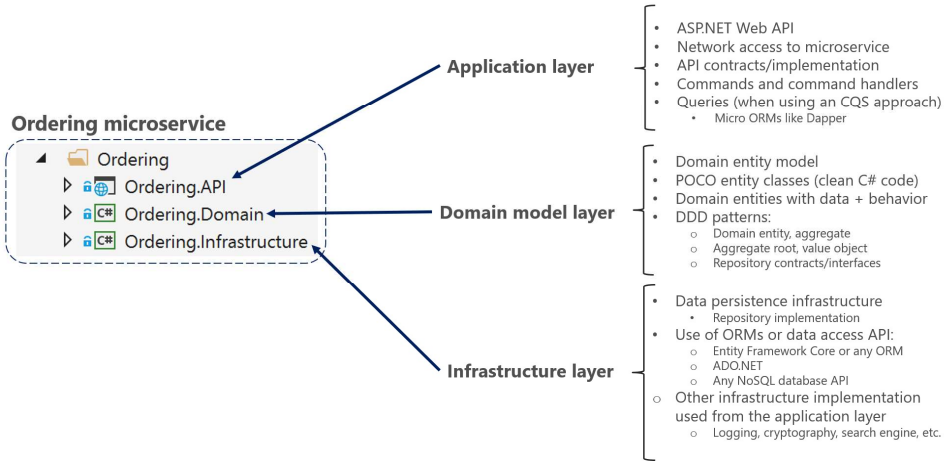
DDD ADVOCATES FOR THE USE OF DOMAIN-SPECIFIC TESTS THAT SPECIFICALLY TARGET THE BUSINESS BEHAVIOR OF THE SYSTEM. BY ISOLATING DOMAIN LOGIC FROM INFRASTRUCTURE CONCERNS, SUCH AS DATABASES OR EXTERNAL SERVICES, DDD PROMOTES THE CREATION OF FOCUSED AND EFFECTIVE TESTS. THESE TESTS ENSURE THAT THE SOFTWARE BEHAVES AS INTENDED AND FACILITATE CONFIDENCE IN SYSTEM BEHAVIOR, EVEN AS IT EVOLVES OVER TIME.

Layers in DDD Microservices

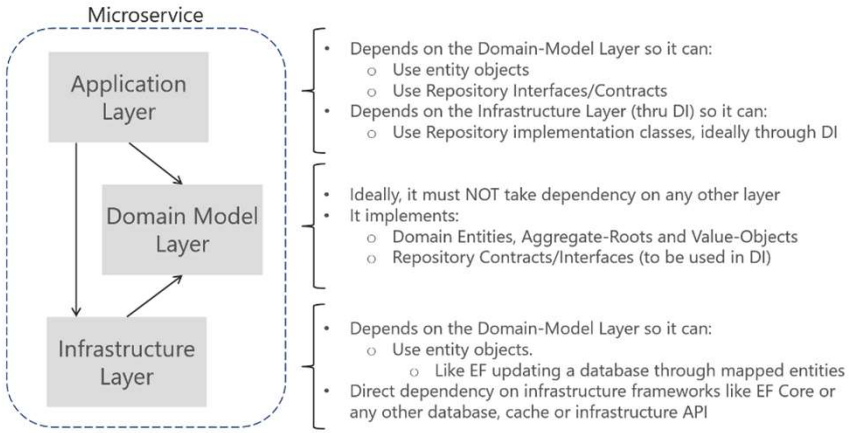
“Domain-driven design (DDD) advocates modeling based on the reality of business as relevant to your use cases.

Keep the microservice context boundaries relatively small”

Layers in a Domain-Driven Design Microservice



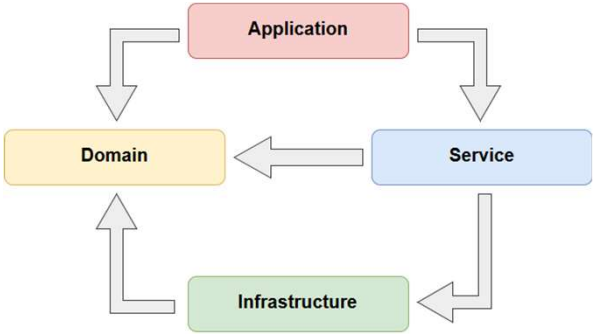
Dependencies between Layers in a Domain-Driven Design service

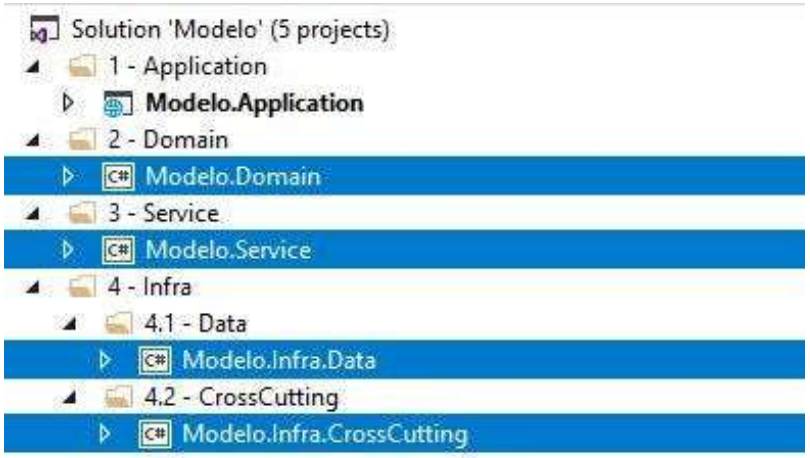


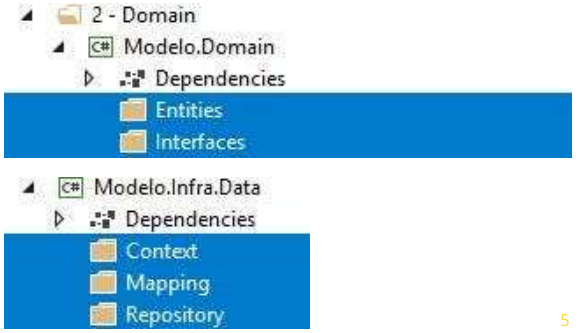
[Designing a DDD-oriented microservice | Microsoft Learn]

Layers in DDD Monoliths

“While monolithic code projects may not fully leverage the scalability and deployment advantages of microservices architectures, applying DDD principles can still provide significant benefits. It helps tackle the challenges of complex codebases, improves maintainability, facilitates collaboration, and leads to a better alignment of the software with the business goals.”



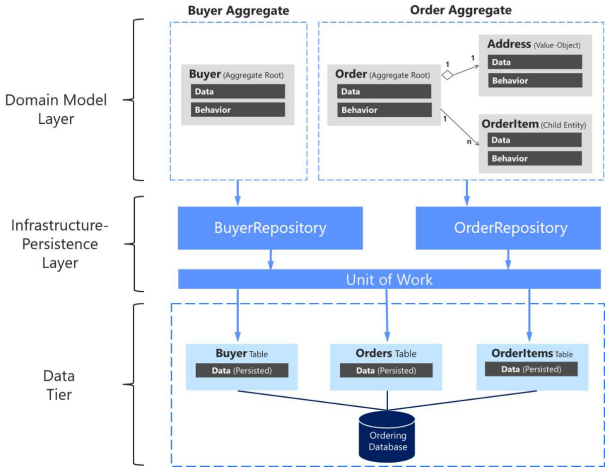




Repository Pattern

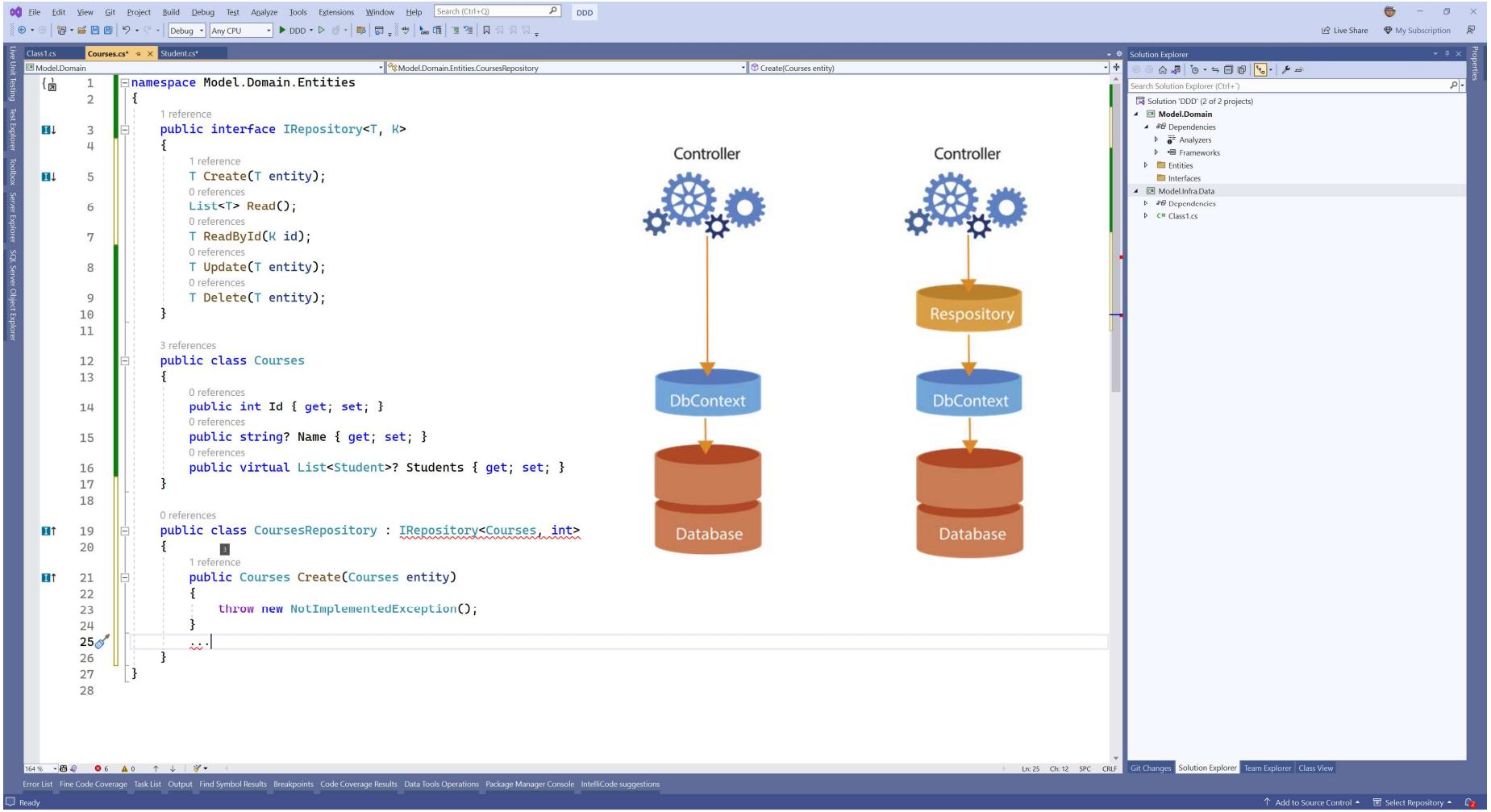
“Domain-Driven Design (DDD) and the Repository Pattern are two complementary concepts that are often used together in software development. DDD focuses on modeling the core business domains and their behavior within a software system.

On the other hand, the Repository Pattern is a design pattern that provides an abstraction layer between the domain model and the persistence layer of an application. It enables the domain model to work with objects without having to directly interact with the underlying data storage or database. ”




```
public interface Repository<T, K> {
    List<T> read();
    T readById(K id);
    T create(T entity);
    T update(T entity);
    T delete(T entity);
}
```

Repository Pattern with EFCore




```
namespace Model.Domain.Entities
{
    1 reference
    public interface IRepository<T, K>
    {
        1 reference
        T Create(T entity);
        0 references
        List<T> Read();
        0 references
        T ReadById(K id);
        0 references
        T Update(T entity);
        0 references
        T Delete(T entity);
    }

    3 references
    public class Courses
    {
        0 references
        public int Id { get; set; }
        0 references
        public string? Name { get; set; }
        0 references
        public virtual List<Student>? Students { get; set; }
    }

    0 references
    public class CoursesRepository : IRepository<Courses, int>
    {
        1 reference
        public Courses Create(Courses entity)
        {
            throw new NotImplementedException();
        }
    }
}
```

Architectures that supports DDD

Eric Evans
DDD Author

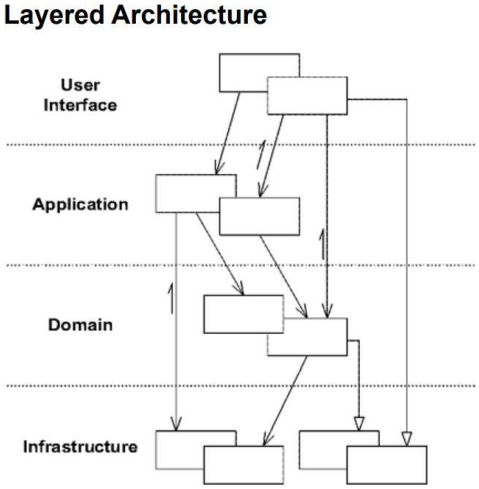
”There are several architectural patterns and styles that support the principles and practices of Domain-Driven Design (DDD). These architectures provide a framework for implementing DDD concepts effectively. Here are a few notable ones:”

1. Layered Architecture (already presented)
2. Hexagonal Architecture (Ports and Adapters)
3. Clean Architecture
4. CQRS (Command Query Responsibility Segregation)
5. Event-Driven Architecture
6. Microservices Architecture

Layered Architecture

Eric Evans
DDD Author

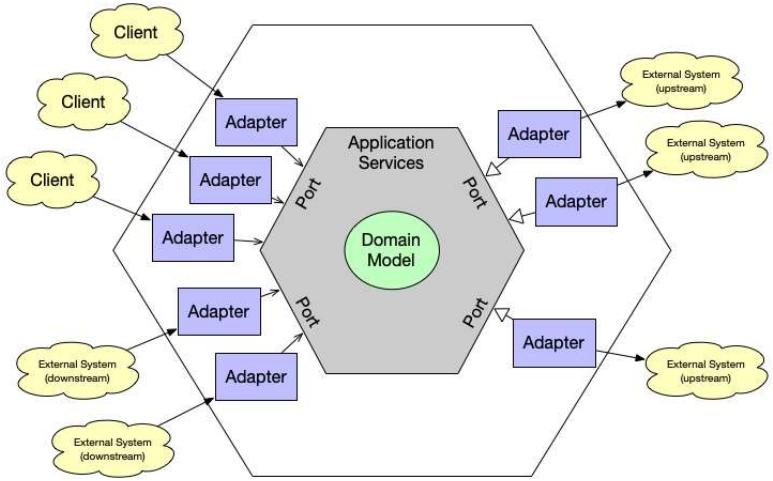
”Layered architecture is a common choice for DDD implementations. It organizes the application into horizontal layers such as presentation, application, domain, and infrastructure. Each layer has specific responsibilities and dependencies, allowing for separation of concerns and modularity. The domain layer, in particular, contains the domain model, entities, aggregates, and business rules”



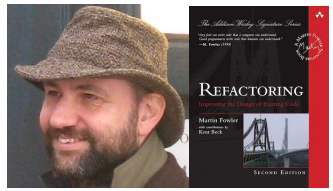
Hexagonal Architecture (Ports and Adapters)

Eric Evans
DDD Author

” Hexagonal architecture, also known as Ports and Adapters, focuses on decoupling the core business logic from external dependencies. The core domain is at the center, surrounded by ports (interfaces) that define interactions with the external world. Adapters implement these ports, bridging the gap between the domain and infrastructure. This architecture promotes flexibility, maintainability, and testability by isolating the domain from implementation details”

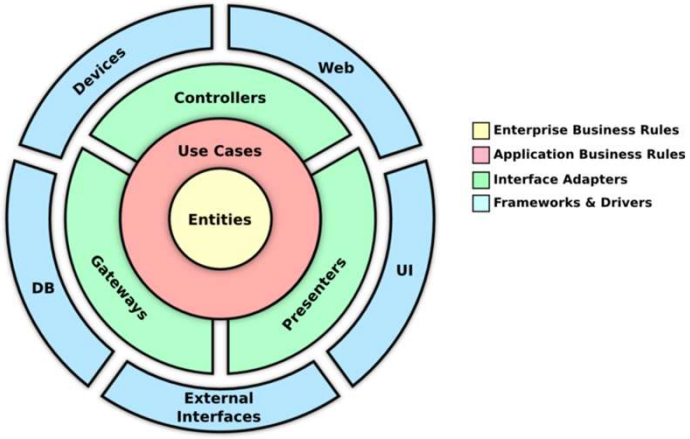


Clean Architecture

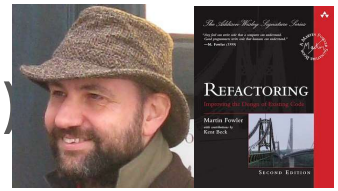


Martin Fowler
Refactoring Author

” Clean Architecture, introduced by Robert C. Martin, emphasizes separation of concerns and independence of frameworks and libraries. It employs concentric circles with the domain at the center, surrounded by application, interface, and infrastructure layers. The architecture defines dependencies flowing inward, with higher-level layers being independent of lower-level layers. This architecture supports the DDD principles of encapsulating business logic and isolating it from external dependencies”

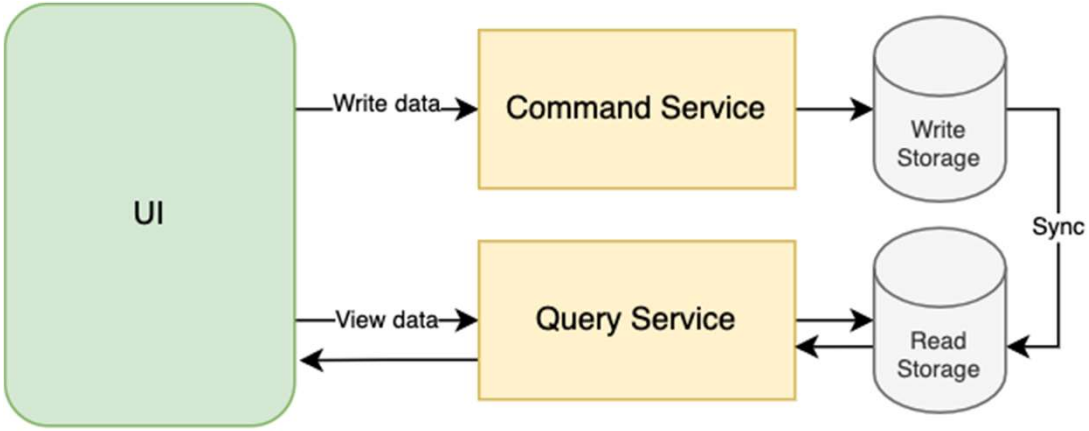


CQRS (Command Query Responsibility Segregation)



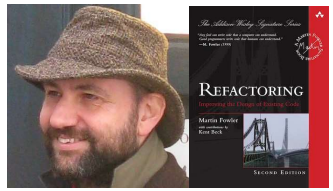
Martin Fowler
Refactoring Author

” CQRS separates the read and write operations of an application into distinct models. It allows for different models to optimize for different use cases. The Command model handles write operations and enforces business rules, while the Query model handles read operations for efficient querying. CQRS aligns well with DDD's emphasis on modeling behavior and supports scalability, performance, and maintainability.”



Domain Driven Design

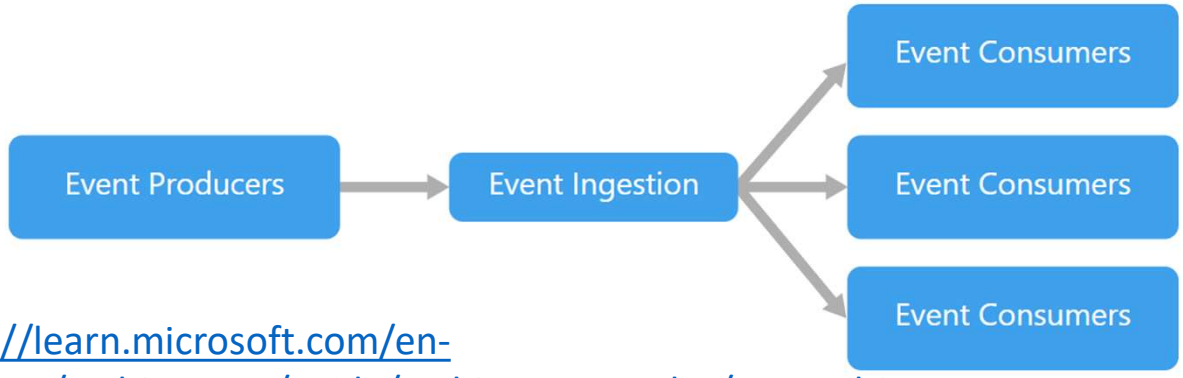
Event-Driven Architecture



Martin Fowler
Refactoring Author

” Event-Driven Architecture (EDA) focuses on the flow of events and messages within a system. It enables loose coupling, scalability, and responsiveness. DDD can leverage EDA by using events to communicate changes and domain events to represent significant occurrences within the business domain. Event sourcing, a technique closely related to EDA, can be employed to store and replay events, providing an auditable log of domain behavior.”

An event-driven architecture consists of **event producers** that generate a stream of events, and **event consumers** that listen for the events.

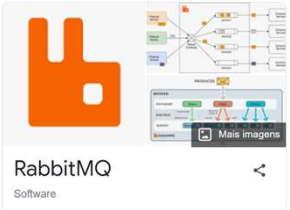


https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/event-driven

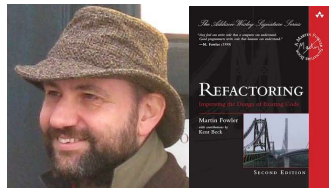
Azure Service Bus

Azure Event Grid

Azure Event Hub

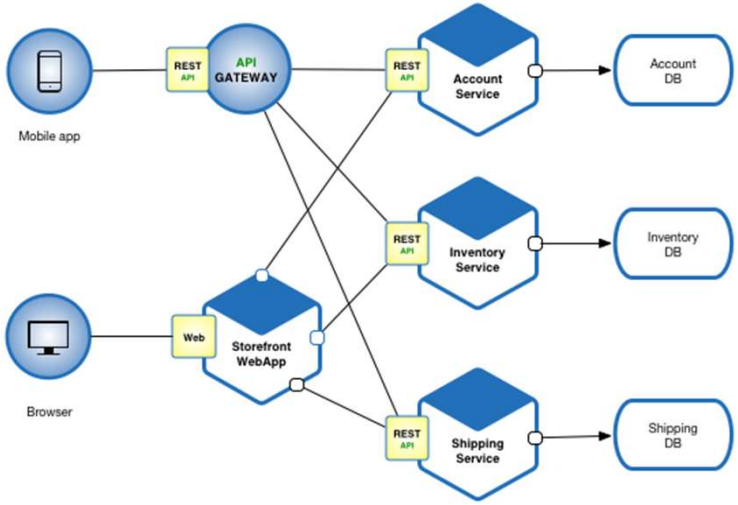


Microservices Architecture



Martin Fowler
Refactoring Author

” While not exclusive to DDD, Microservices Architecture aligns well with DDD principles. It decomposes a system into small, independent services, each owning a specific business capability. Each microservice encapsulates its own domain and can be developed and deployed independently. DDD fits naturally within microservices, as each microservice can have its own bounded context, enforcing clear boundaries and encapsulation of business logic.”



Conclusion

” It's important to note that the choice of architecture depends on various factors such as the complexity of the domain, scalability requirements, team expertise, and project constraints. Architects and development teams should carefully evaluate the characteristics and trade-offs of different architectures to determine the best fit for their DDD implementation”

Who else needs some coffee ?

Domain Driven Design

Technical Training

Let’s do it