
#PITCHONLINE PRESENTS
Microservices By
Domain Driven Design
#Coding

Scifoni Ivano

Fabio Mannis

Francesco Del Re

Matteo Riccardi

Valerio Benedetti

Francesco Del Re

Engineer in Computer Science

Principal Solutions Architect & Project Leader

Microsoft CERTIFIED AZURE DEVOPS ENGINEER EXPERT

Microsoft CERTIFIED AZURE DEVELOPER ASSOCIATE

Microsoft CERTIFIED Professional

.NET foundation

GitHub

Arctic Code Vault Contributor

francesco.delre.87@gmail.com

https://www.linkedin.com/in/francesco-delre

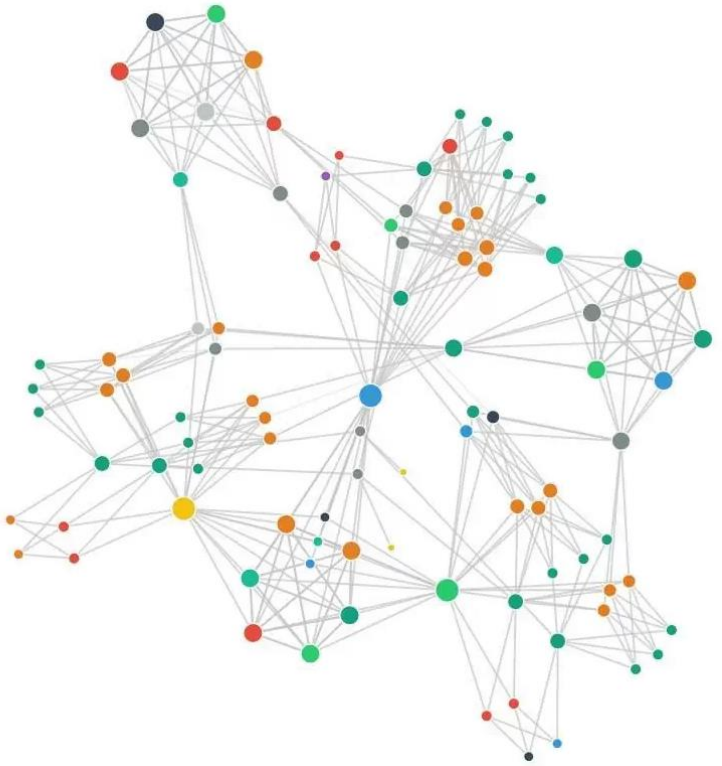
https://github.com/engineering87

Agenda

- ❑ Introduzione ai microservizi
- ❑ Comunicazione nei microservizi
- ❑ Accesso ed esposizione dati nei microservizi
- ❑ Gestione della consistenza del dato
- ❑ Complessità nei microservizi
- ❑ Introduzione al Domain-Driven Design
- ❑ Building Blocks del Domain-Driven Design
- ❑ Progettare microservizi con il Domain-Driven Design

"Modello architetturale che decompone un'applicazione in un insieme di servizi ognuno dei quali ha un insieme focalizzato e coeso di responsabilità con un accoppiamento debole con gli altri servizi"

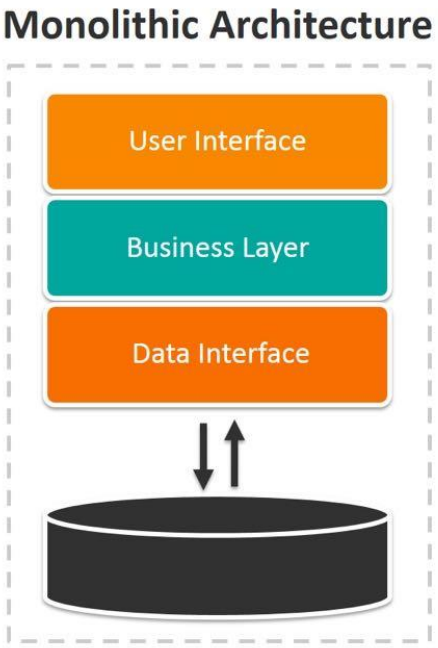
- Insieme di servizi eterogenei che **comunicano** tra di loro attraverso la rete
- Introduzione dei **Bounded context**, ovvero confini ben definiti delle funzionalità e del relativo dominio dei dati
- Ogni servizio è **indipendente** a fronte di un basso accoppiamento

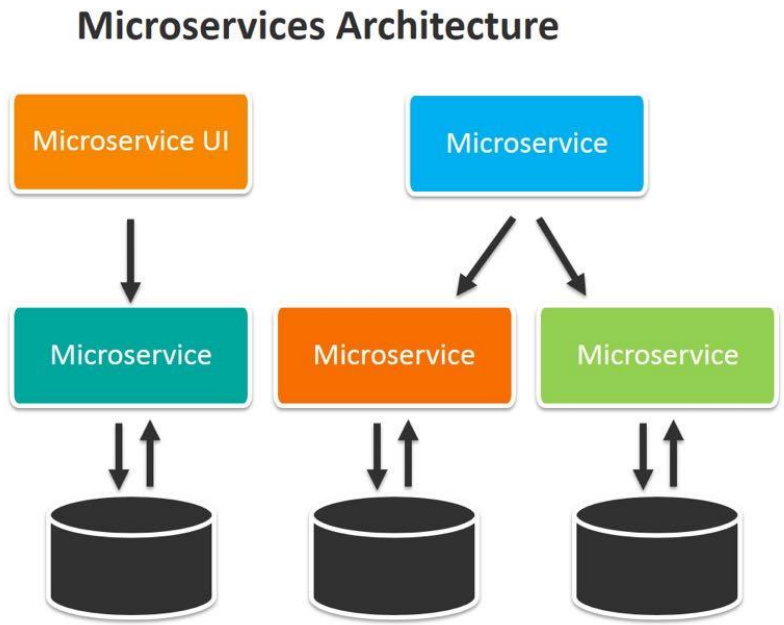


Architettura monolitica VS microservizi

Focal points

- ❑ Facilità di sviluppo
- ❑ Maggiore resilienza
- ❑ Deploy agile
- ❑ Scalabilità
- ❑ Riutilizzo di moduli
- ❑ Velocità di ripristino
- ❑ Manutenibilità

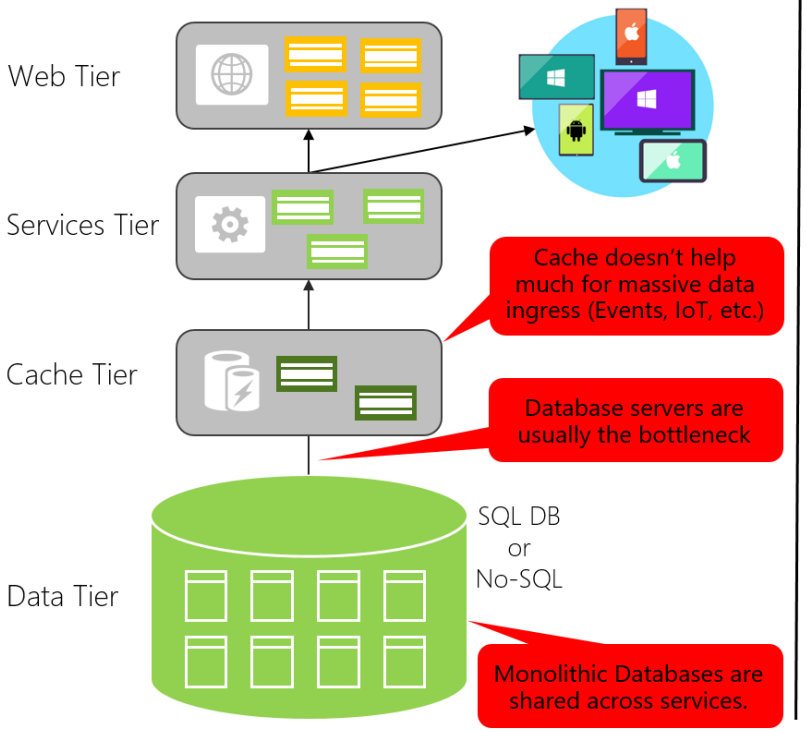




Accesso ai dati: approccio tradizionale VS microservizi

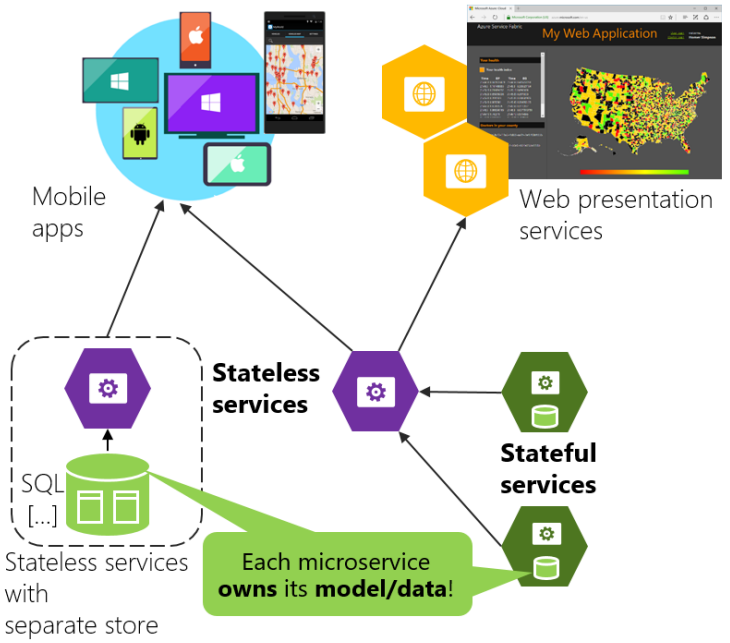
Data in Traditional approach

- Single monolithic database
- Tiers of specific technologies



Data in Microservices approach

- Graph of interconnected microservices
- State typically scoped to the microservice
- Remote Storage for cold data



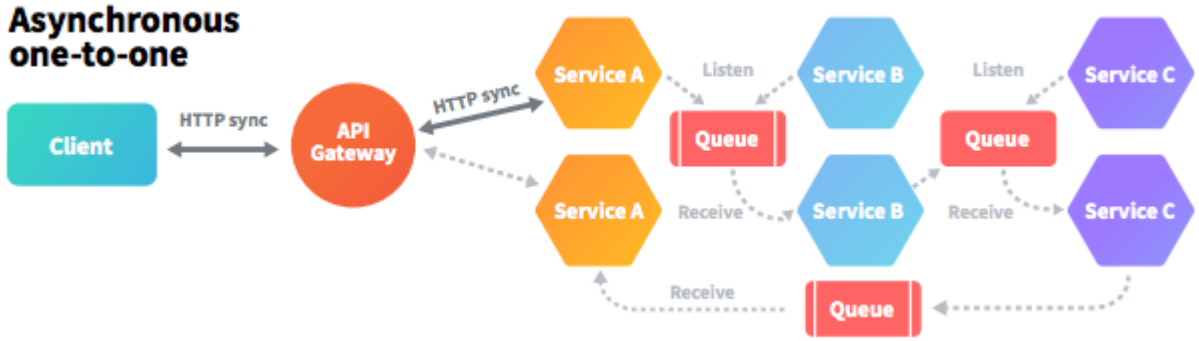
Comunicazione nei microservizi

Comunicazione Sincrona

HTTP è un protocollo sincrono. Il client invia una richiesta e attende una risposta dal servizio

Comunicazione Asincrona

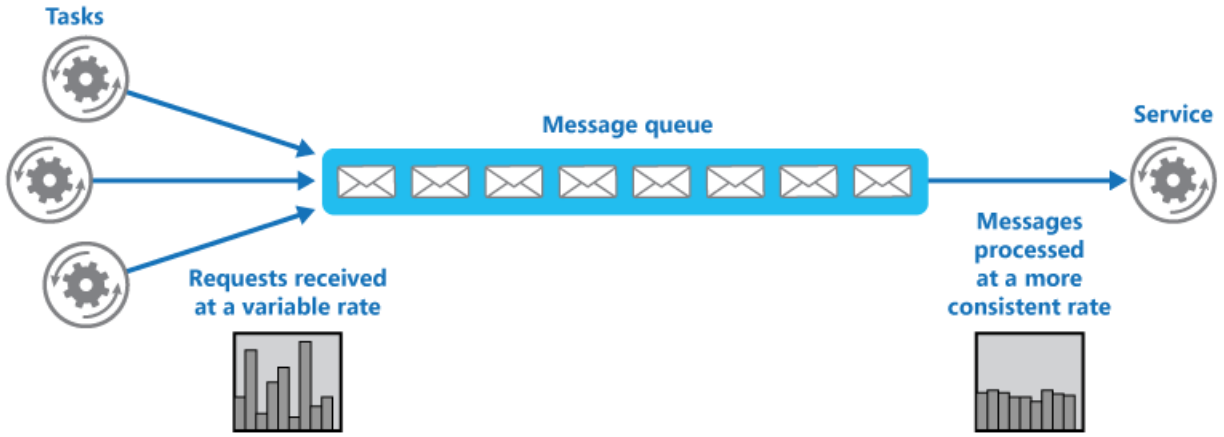
Il mittente del messaggio non attende una risposta. Protocolli come AMQP (*Advanced Message Queuing Protocol*) utilizzano messaggi asincroni

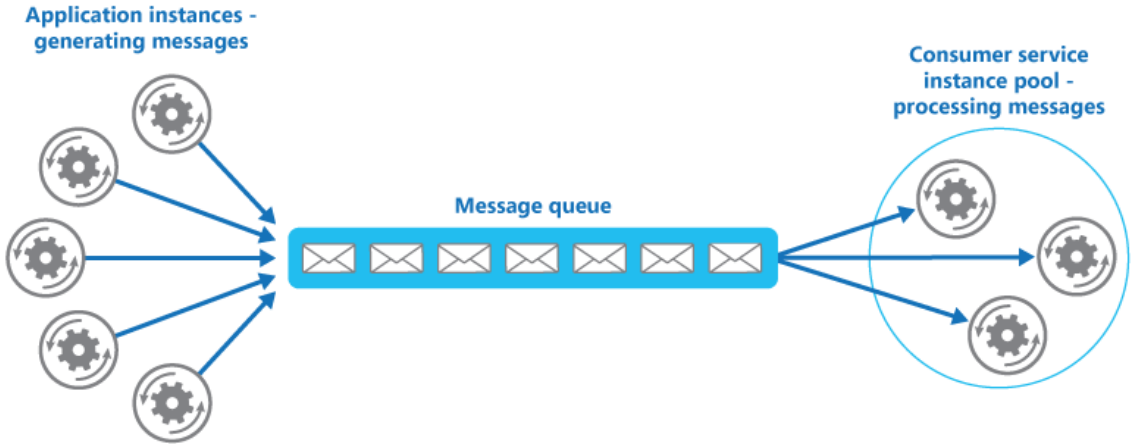


Comunicazione tramite code

Principi

- **Disaccoppiamento** tra produttore e consumatore
- Comunicazione **asincrona**
- **Scalabilità** indipendente
- **FIFO** processing
- Persistenza e maggiore resilienza
- Integrazione **Hybrid Cloud**





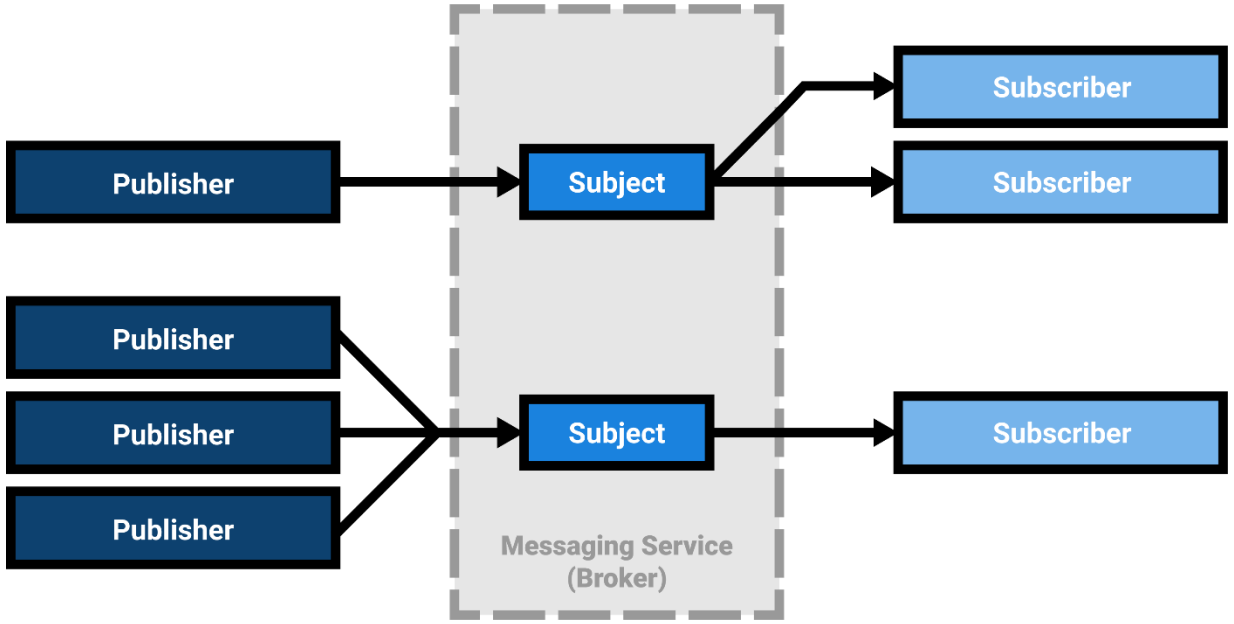
Comunicazione Publish\Subscribe

Principi

- Comunicazione **topic-based**
- Comunicazione orientata agli **eventi**
- Disaccoppiamento spaziale, temporale e di sincronizzazione

Elementi

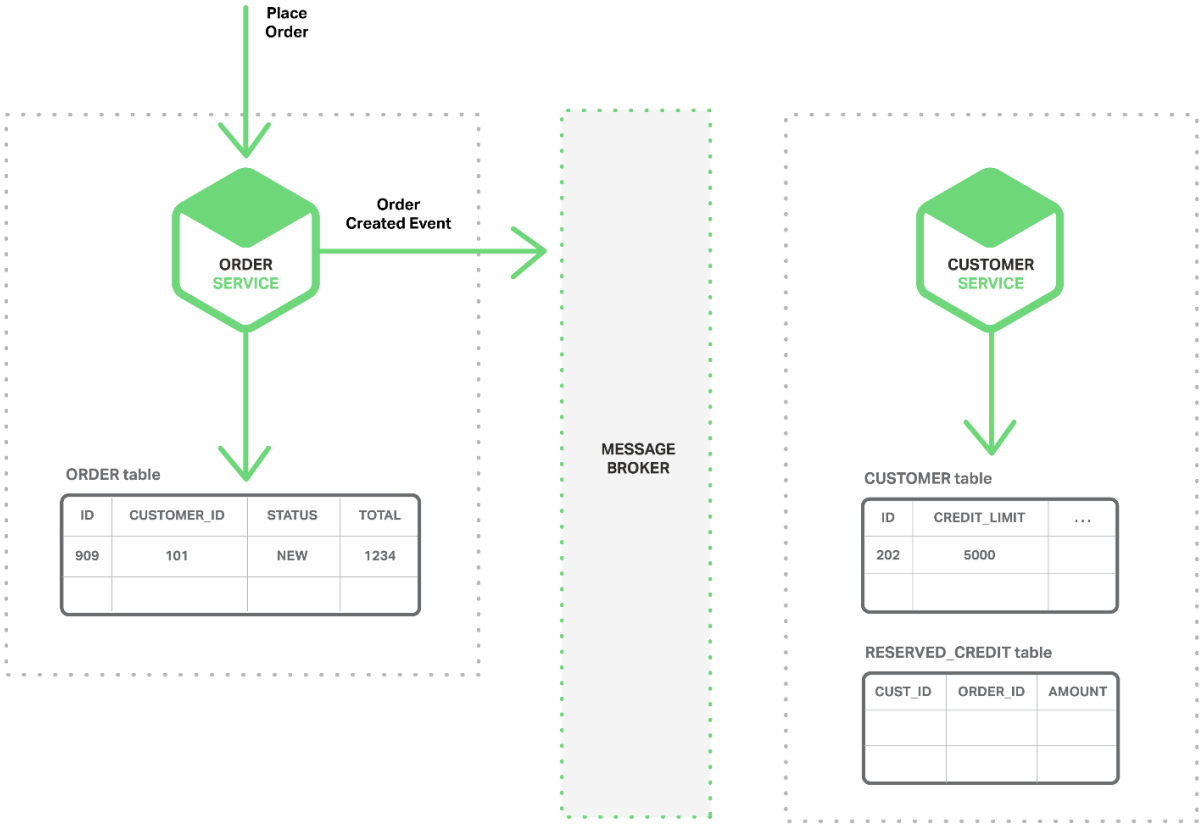
- Publisher
- Subscriber
- Communication Infrastructure (Message Broker)



Pattern Database per Service

Principi

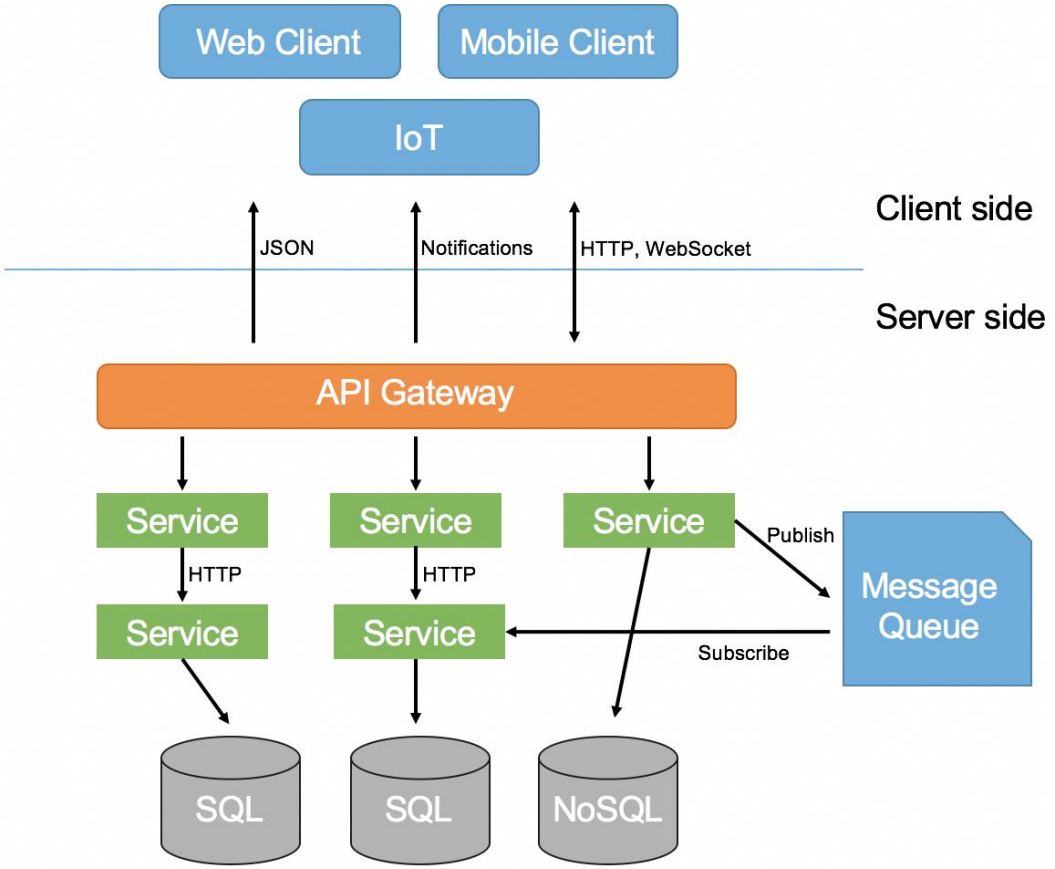
- Ogni servizio deve interagire direttamente **solo con i propri dati**
- Ogni servizio si occupa dell'intero ciclo di vita dei propri dati
- La comunicazione e lo scambio dati possono avvenire esclusivamente attraverso **API** ben definite
- Richiede una definizione precisa del **dominio dati** per ogni servizio



Data as a Service (DaaS)

Principi

- Un servizio **non può accedere direttamente** al data store di un altro servizio
- Il dato viene esposto attraverso un insieme di **interfacce** standard e ben definite
- Architetture a microservizi permettono di mixare diverse tipologie di persistenza, questa implementazione è chiamata persistenza **poliglotta**

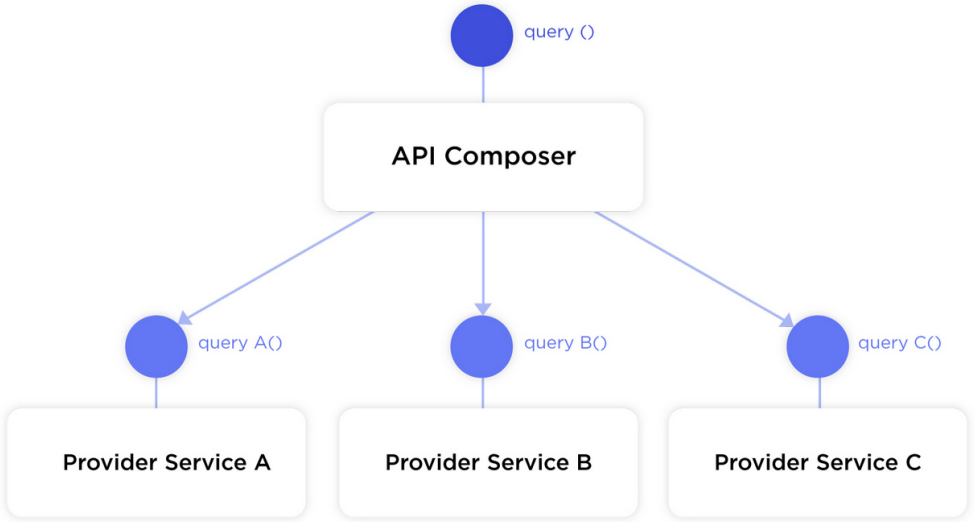


Pattern API Composition

CONTESTO
Data as a Service

PROBLEMA
Come implementare query che impattano base dati distribuite?

SOLUZIONE
Utilizzo di un API Composer

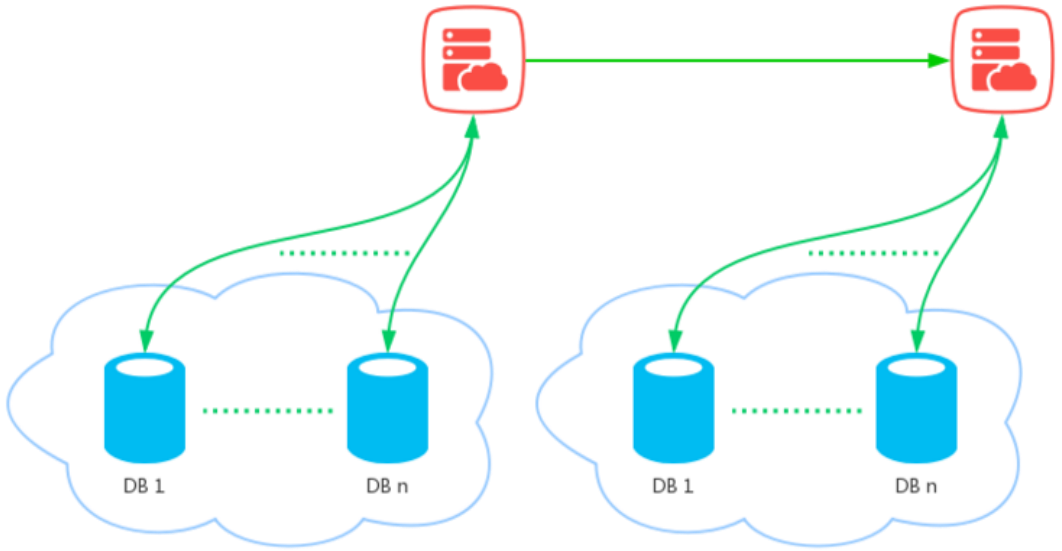


Realizza la logica delle query distribuite invocando i singoli servizi impattati e **aggregando** successivamente le risposte in un unico risultato

Data Consistency nei sistemi distribuiti

Focal points

- Il dato è **replicato** per assicurare la disponibilità, la scalabilità e le performance
- Le repliche forniscono implicitamente **ridondanza** del dato
- In un contesto distribuito le proprietà **ACID** non sono garantite
- Necessità di garantire la **consistenza** del dato tra tutte le repliche

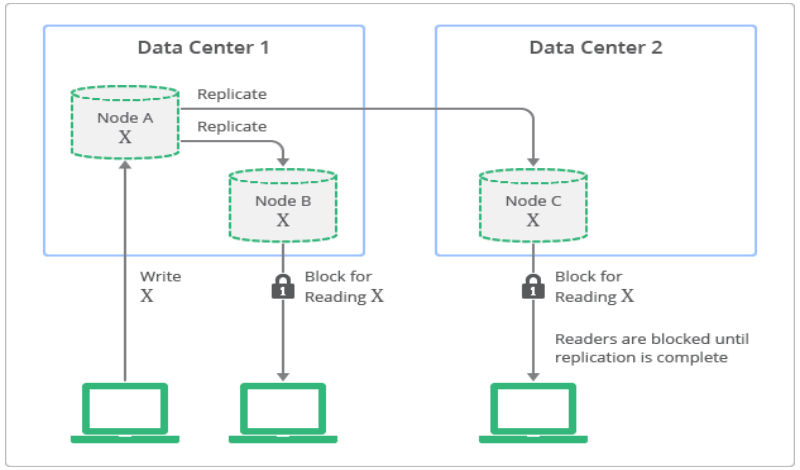


Il **CAP Theorem** afferma che è impossibile per un sistema distribuito fornire simultaneamente tutte e tre le proprietà di **Concistency**, **Availability**, **Partitioning**

Strong Consistency VS Eventual Consistency

Strong Consistency

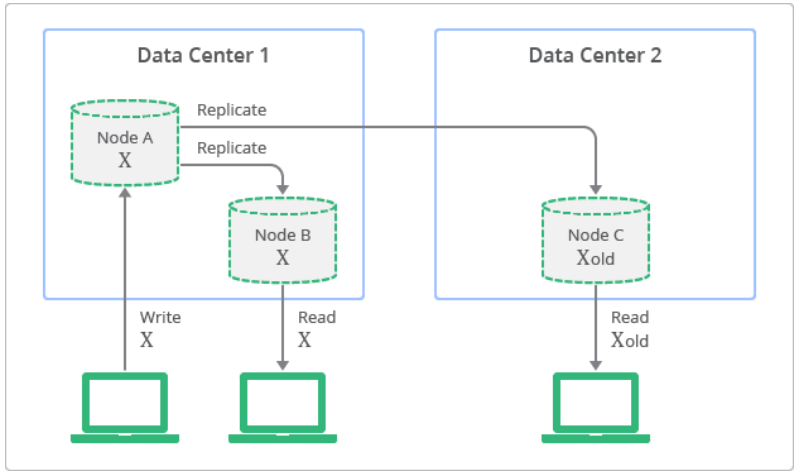
- In presenza di aggiornamenti, tutte le successive operazioni di lettura restituiranno il dato più aggiornato
- Necessità di un processo di **linearizzazione** delle transazioni



VS

Eventual Consistency

- In assenza di aggiornamenti, **prima o poi** le operazioni di lettura restituiranno il dato più aggiornato
- Non necessità di logiche di linearizzazione

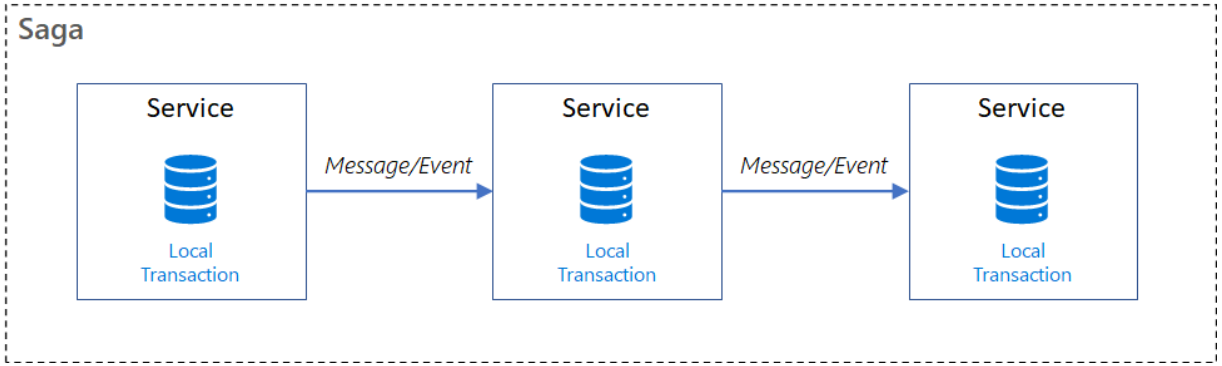


Transazioni distribuite e SAGA Pattern

CONTESTO
Transazioni che impattano più servizi

PROBLEMA
Come gestire transazioni distribuite?

SOLUZIONE
Utilizzo del pattern SAGA



Sequenza di transazioni che aggiorna ogni servizio e pubblica un messaggio o un evento per **attivare il passaggio successivo** della transazione

SAGA Pattern: Orchestration VS Choreography

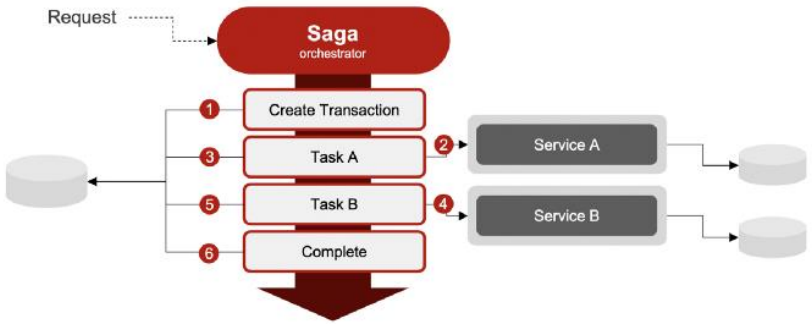
Orchestration

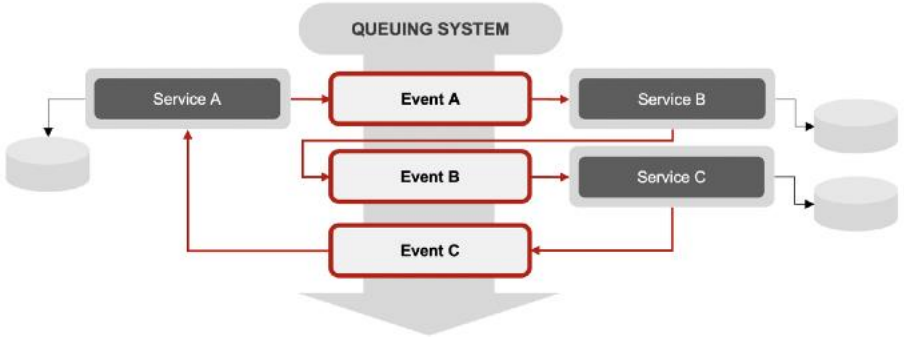
- ❑ **Controller centrale** per il coordinamento dei flussi
- ❑ Facilità di gestione dell'intero flusso
- ❑ Sviluppo semplificato dei singoli servizi
- ❑ I servizi presentano un alto accoppiamento con l'orchestratore

VS

Choreography

- ❑ Coordinamento dei flussi distribuito attraverso l'utilizzo di **eventi**
- ❑ Facilità di implementazione
- ❑ I servizi presentano un basso accoppiamento
- ❑ Possibile limitazione nel numero di transazioni locali
- ❑ Gestione degli errori più difficile





Dati nei microservizi: funzionalità e complessità

- ❑ Necessità di una gestione delle transazioni distribuite
- ❑ Necessità di una gestione consistente del dato anche a fronte di fault
- ❑ Necessità di pattern di aggregazione del dato
- ❑ Base dati dedicata e specializzata al singolo microservizio
- ❑ Disaccoppiamento completo delle funzionalità
- ❑ Possibilità di avere una persistenza poliglotta

E' fondamentale una fase dedicata all'analisi del dominio dati per ottenere il giusto **trade-off** tra **funzionalità** dell'architettura e **complessità** di implementazione

Cos'è il Domain-Driven Design (DDD)?

"It is a way of thinking and a set of priorities, aimed at accelerating software projects that have to deal with complicated domains"

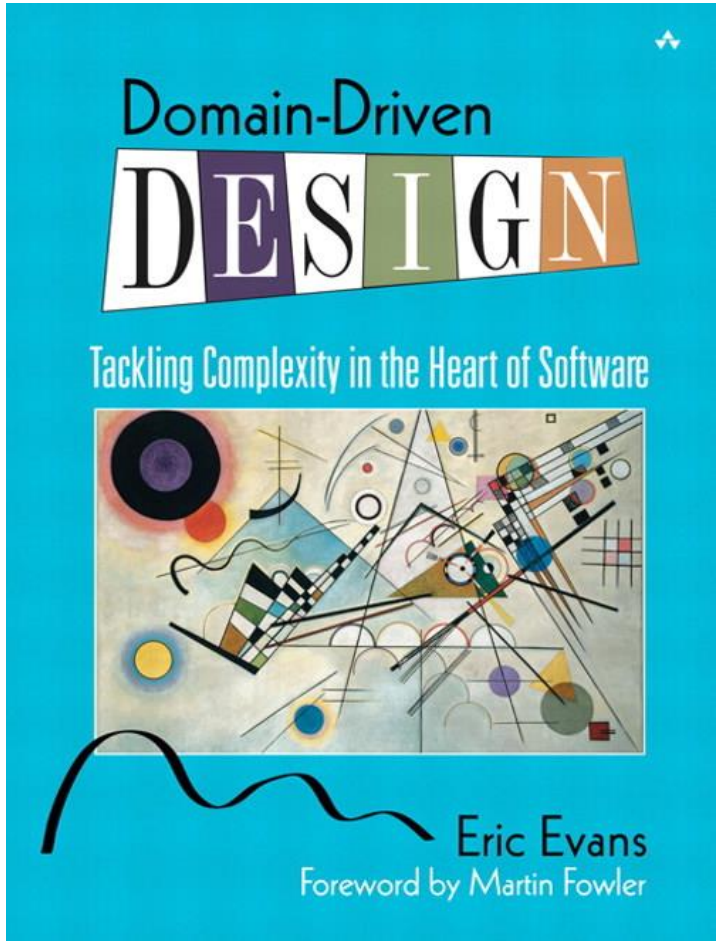
- DDD non è formalmente una metodologia ma **l'insieme di alcune pratiche** di supporto allo sviluppo software, integrate con metodologie collaudate come i processi agili.
- DDD definisce **un insieme di pattern** che normano e rendono metodiche le operazioni di realizzazione di un modello di dominio.
- DDD è orientato agli **oggetti**, assume che la nostra applicazione sia realizzata sfruttando il **Domain Model Pattern**.

Un **approccio** alla realizzazione del software **pragmatico**, **consistente** e **scalabile** anche in presenza di complessità crescente del dominio applicativo

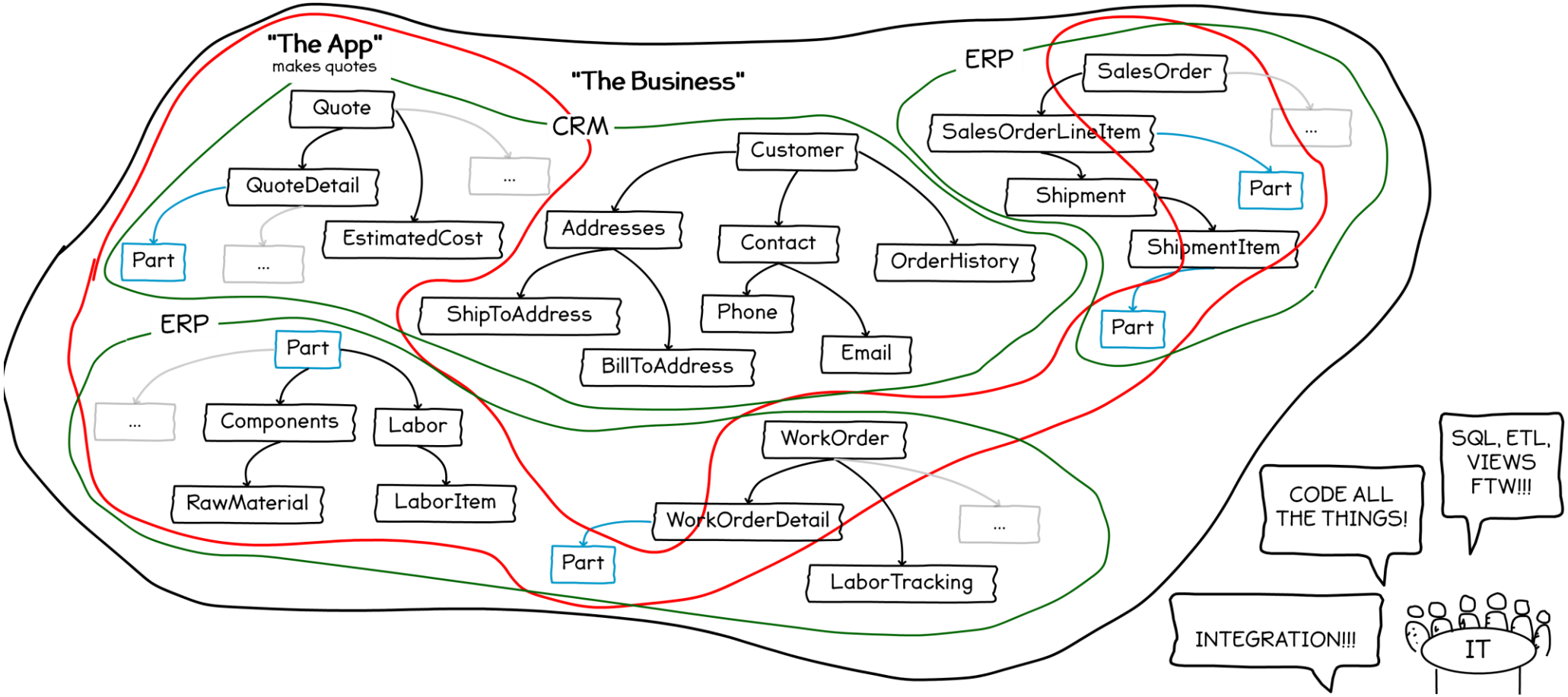
Introduzione al Domain-Driven Design

"The critical complexity of most software projects is in understanding the business domain itself " ***(Eric Evans)***

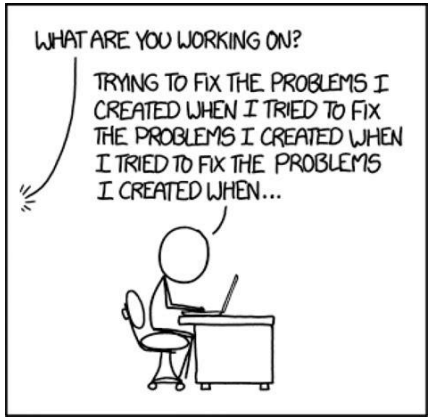
- Insieme di principi e pattern per la realizzazione di architetture efficienti in **contesti complessi**
- Introduce un approccio **Domain First** al fine di astrarre la logica di business e semplificare gli sviluppi
- Basato su una stretta **collaborazione** tra esperti funzionali e tecnici
- Riduzione del gap tra analisi funzionale e successiva implementazione



Big Ball of Mud …



… Spaghetti Code



Partizionare un dominio dati complesso con il DDD

Focal Points

- **Domain model:** contiene conoscenze, dati, metriche e obiettivi che ruotano attorno al business; definizione delle **entità**
- **Bounded context:** definizione del confine di un particolare modello al fine di **isolare** una conoscenza specialistica
- **Services**: rappresentano i canali di comunicazione tra differenti *Bounded context*
- **Ubiquitous Language (UL)**: con un linguaggio comune, analisi e design diventano processi di *knowledge crunching*

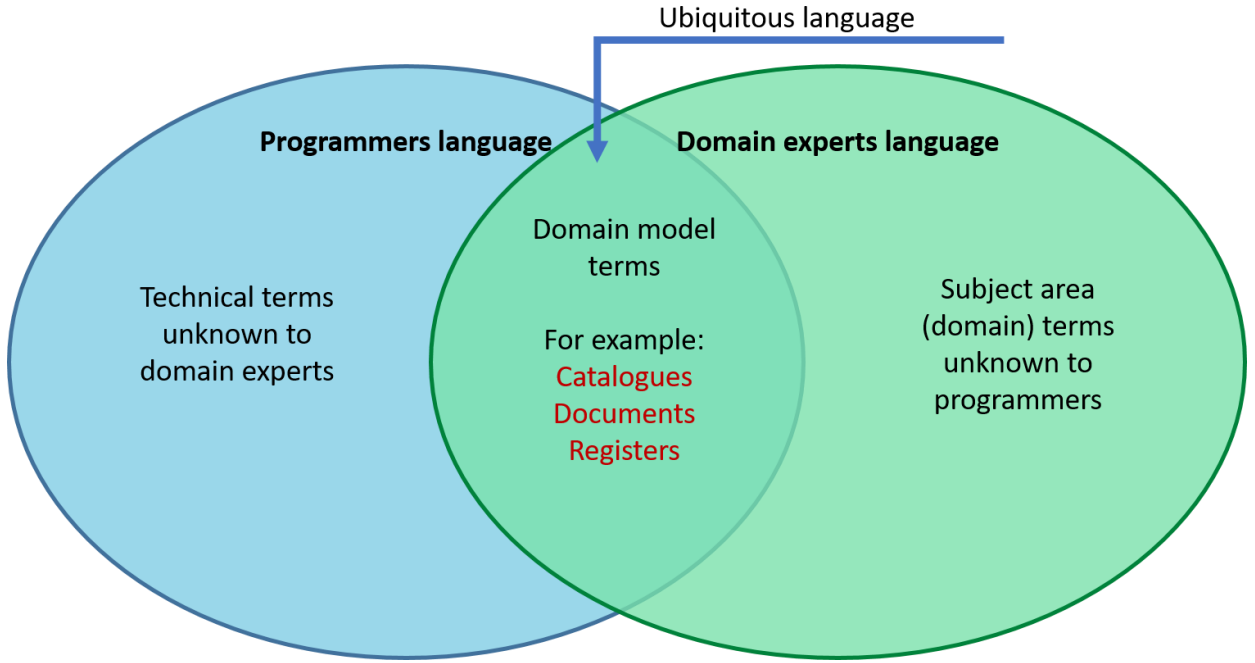
Vantaggi

- **Riduzione della complessità:** Focus chiaro sui dati che caratterizzano il business
- **Comunicazione semplificata**: grazie all'**Ubiquitous Language**, la comunicazione tra sviluppatori e team diventa molto più semplice
- **Maggiore flessibilità**: DDD è orientato agli oggetti ed è quindi modulare e facilmente modificabile
- **Focus sul dominio e non sulla UI/UX**: DDD è *domain first*, gli sviluppatori creeranno applicazioni adatte per il particolare dominio

Ubiquitous Language (UL)

Focal points

- Un **metalinguaggio** che integra termini universali al fine di migliorare la comunicazione tra **tecnici** ed **esperti funzionali**
- Insieme di **termini** il cui significato è **chiaro**, **condiviso** e **privo di ambiguità** all'interno del **dominio** specifico
- L'insieme **non è statico** ma deve essere continuamente aggiornato



DDD Building Blocks

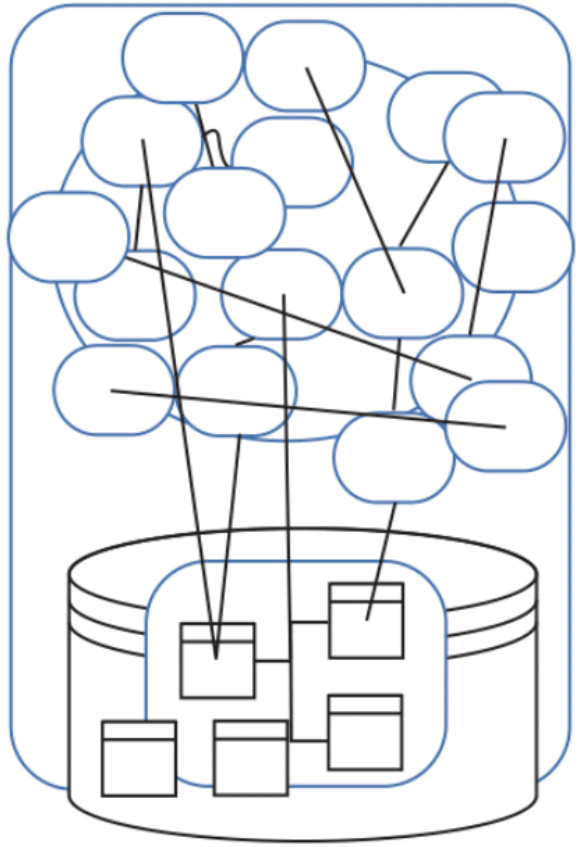
- **Entità**: un oggetto che non è definito solo dai suoi attributi, ma piuttosto dalla sua identità
- **Value Object**: un oggetto che ha una serie di attributi ma non ha identità concettuale
- **Aggregati**: una collezione di oggetti che sono legati insieme da un'entità padre, nota come la radice di aggregato
- **Servizio**: operazione che non appartiene logicamente a nessun oggetto
- **Factory**: si occupa della creazione degli oggetti di dominio come aggregati
- **Repository:** incapsula una collezione di oggetti persistiti sul database
- **Evento**: un evento rappresenta una qualche tipo di cambiamento nel dominio (side effects)

Microservizi attraverso il Domain-Driven Design

Steps

Domain-Driven Design

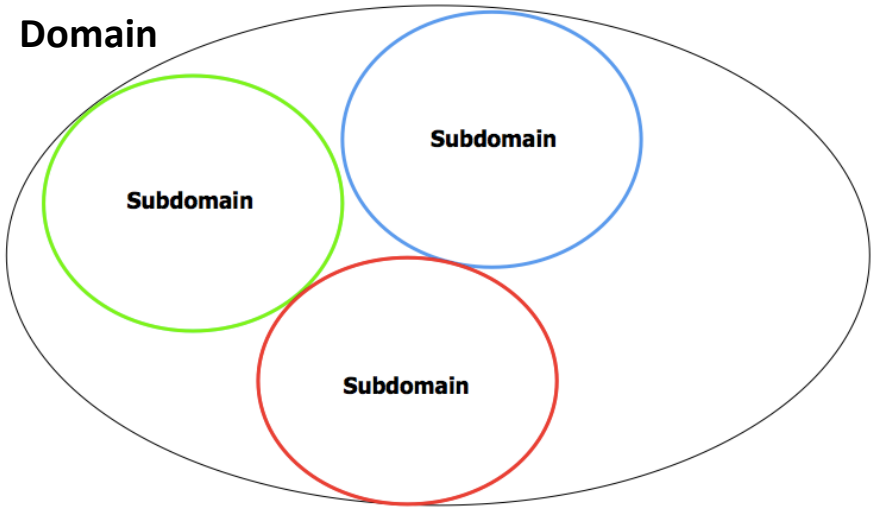
1. Analisi del **dominio dati**
2. Definizione dei **Bounded context**
3. Definizione delle **entità**, degli **aggregati** e dei **servizi**
4. **Identificazione** dei microservizi
5. **Sviluppo** dei microservizi

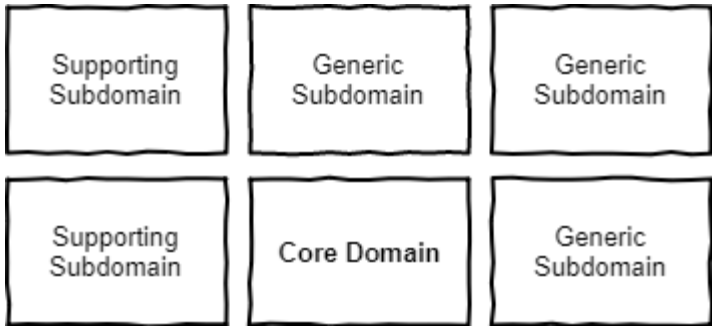


Analisi del dominio dati

Primo Step

- Un dominio è **un contesto di conoscenza** che rappresenta una precisa attività del business
- Il dominio può suggerire una forma di **segregazione** di alto livello per le diverse aree di business
- L'**Event Storming** è un metodo basato su workshop per scoprire le interazioni nel dominio, segue lo schema:
  1. WHEN
  2. WHAT
  3. WHO

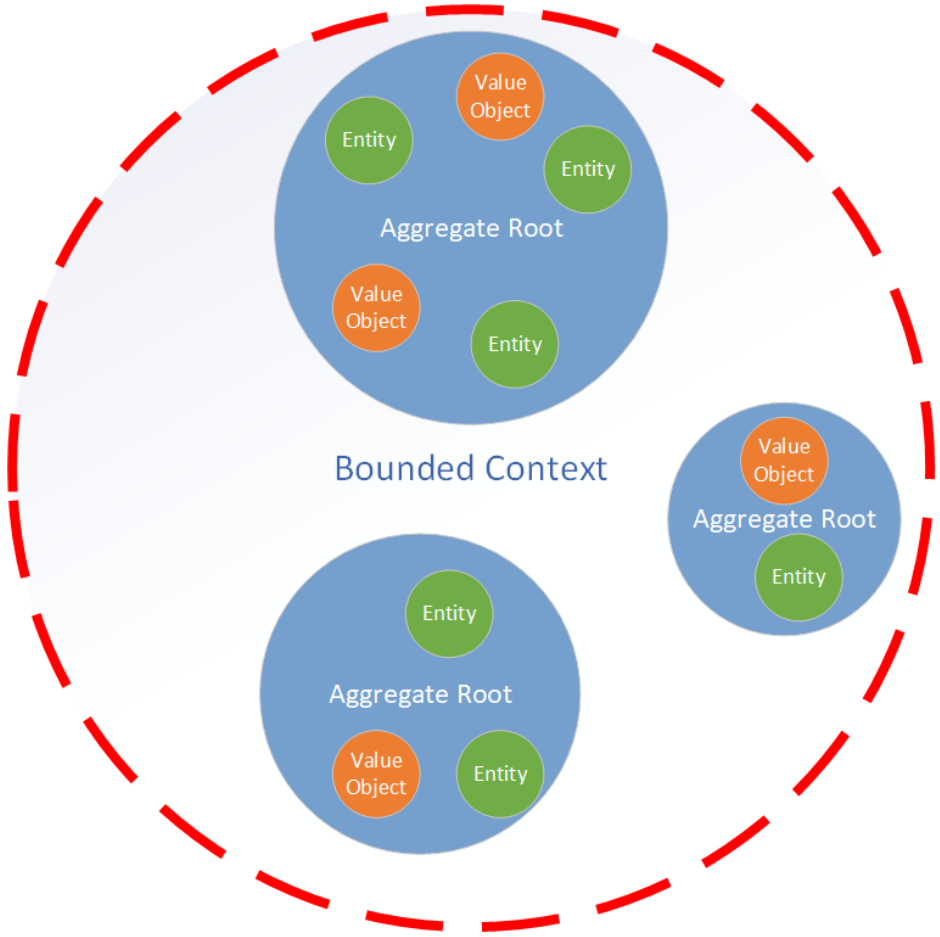




Definizione dei Bounded context

Secondo Step

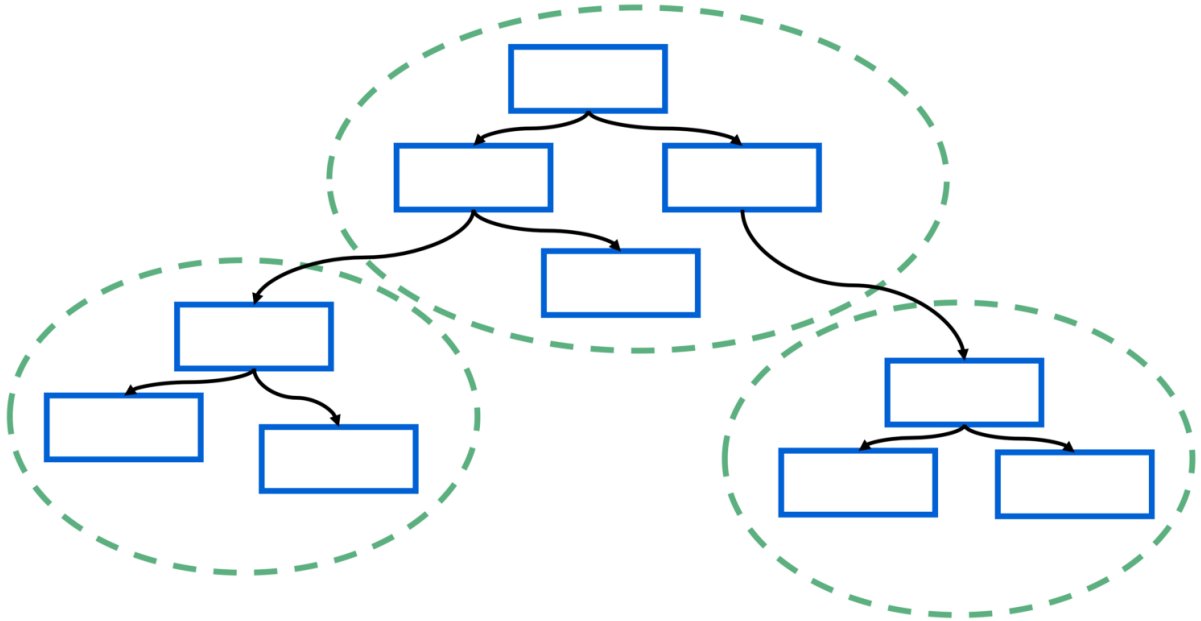
- Un **Bounded context** è la definizione del **confine, logico e funzionale,** di un particolare modello al fine di isolare una conoscenza specialistica
- Ogni Bounded context comunica con gli altri attraverso delle **interfacce**
- Un Bounded context è **potenzialmente** un buon candidato per essere mappato su un **microservizio** ma è necessario valutare attentamente **l'ampiezza** del dominio in esame



Interazioni tra Bounded context: Context Map

Focal points

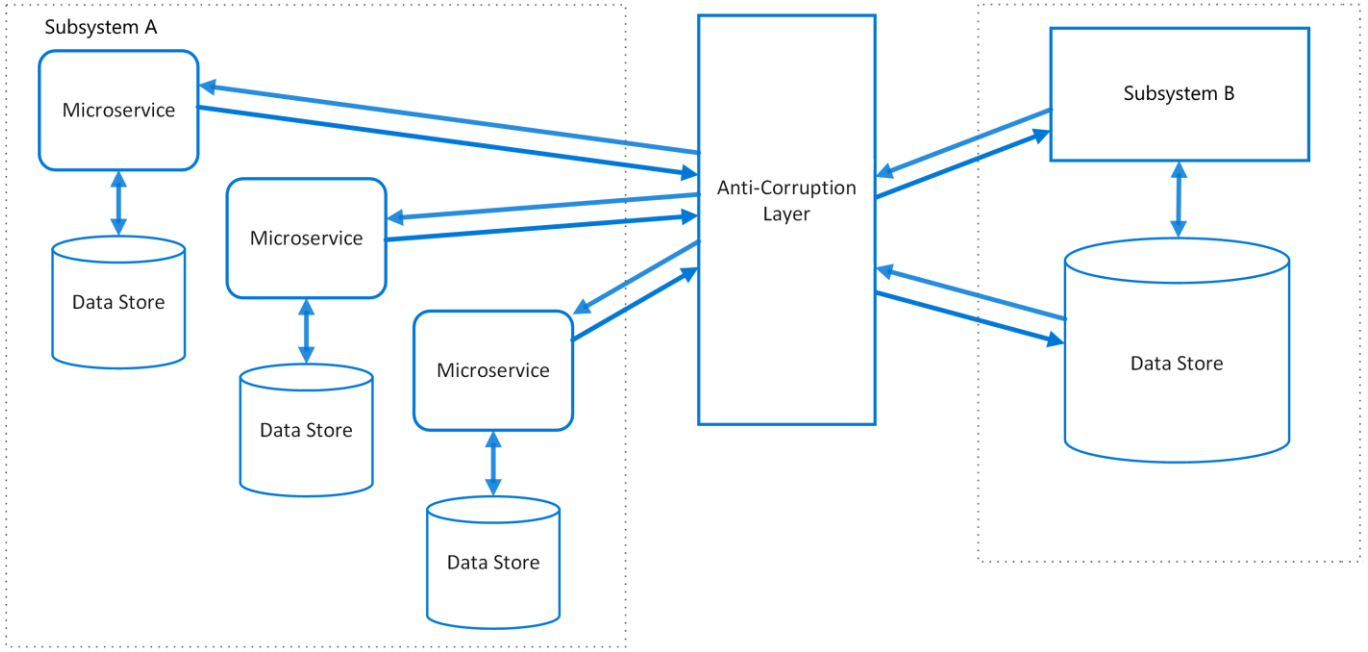
- Un modello è **valido** in uno **specifico contesto**
- Contesti simili potrebbero utilizzare versioni differenti del modello a causa di particolari scopi
- Abbiamo la necessità di definire un **perimetro concettuale** sui modelli
- La **Context Map** riflette le interazioni e la collaborazione tra differenti **Boundex context**



Anti-Corruption Layer (ACL)

Focal points

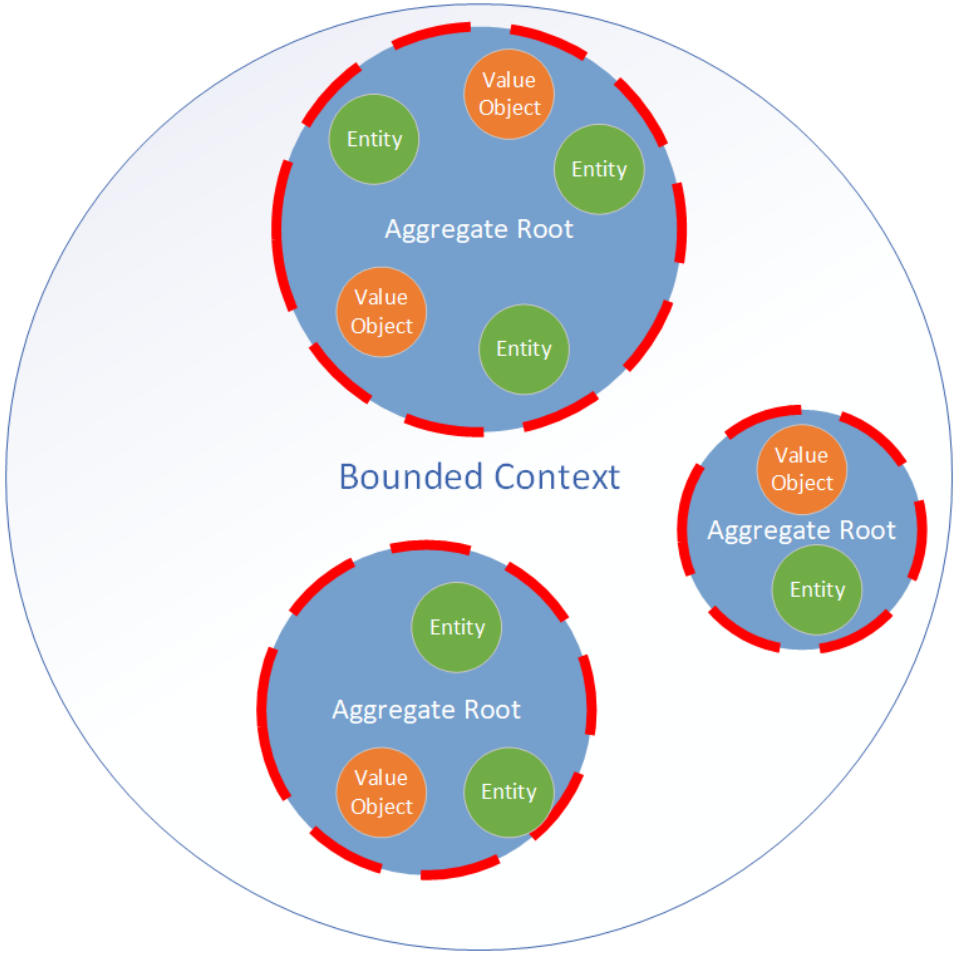
- ACL è uno strato dedicato **all'isolamento** di due o più sottosistemi
- Un ACL media la comunicazione tra diversi sottosistemi al fine di garantire la **consistenza** dei sistemi
- Pattern molto utilizzato nelle **migrazioni graduali** di sistemi legacy verso architetture più moderne
- Attenzione: un ACL aggiunge latenza nelle comunicazioni



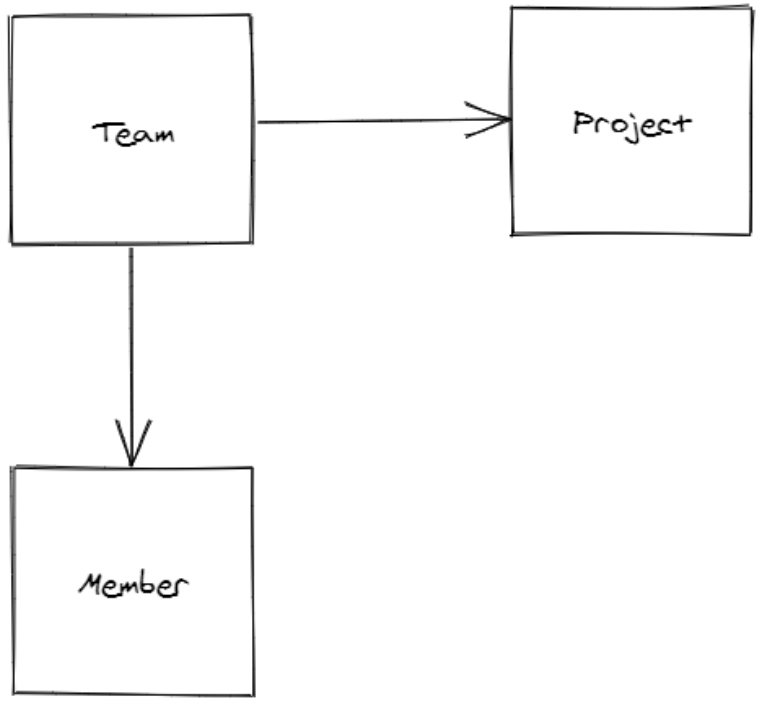
Definizione delle entità, degli aggregati e dei servizi

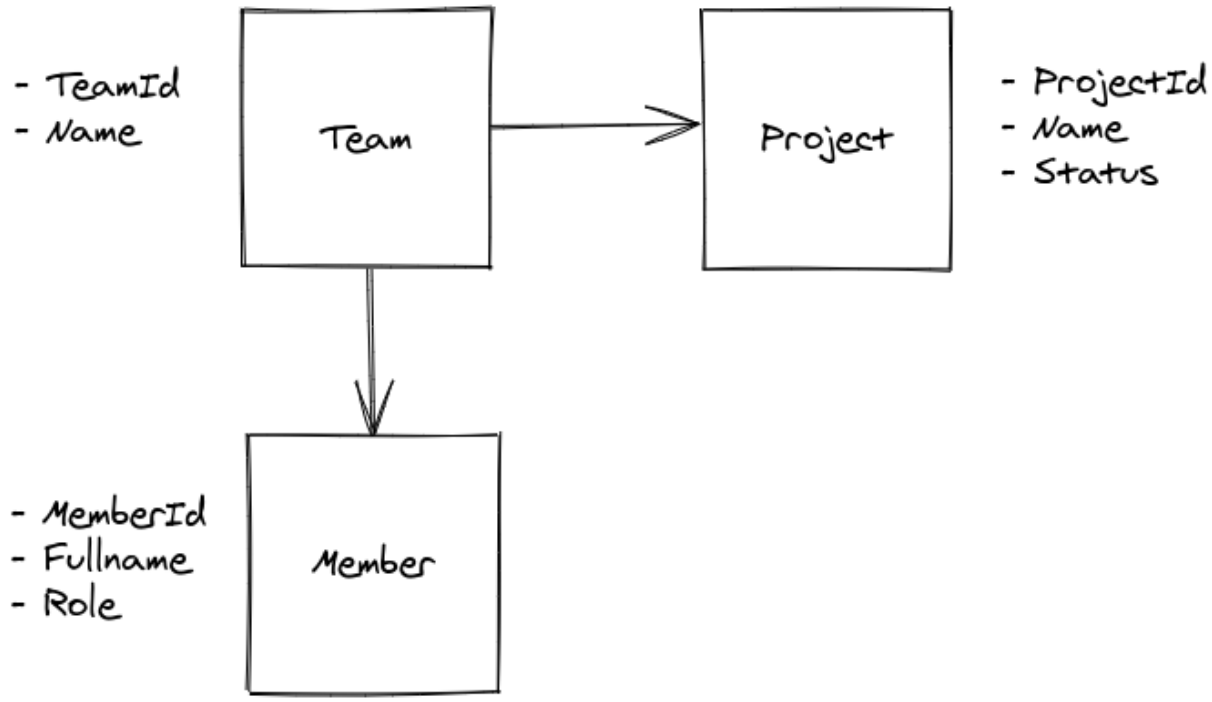
Terzo Step

- Cercare di **minimizzare il numero di entità e Value Objects** all'interno degli aggregati
- Valutare attentamente la consistenza cercando di prediligere **la eventual consistency**
- Un aggregato deve avere **un solo elemento radice** e deve essere aggiornato **un singolo aggregato per transazione**
- Definire i servizi come operazioni **stateless** sugli oggetti di dominio



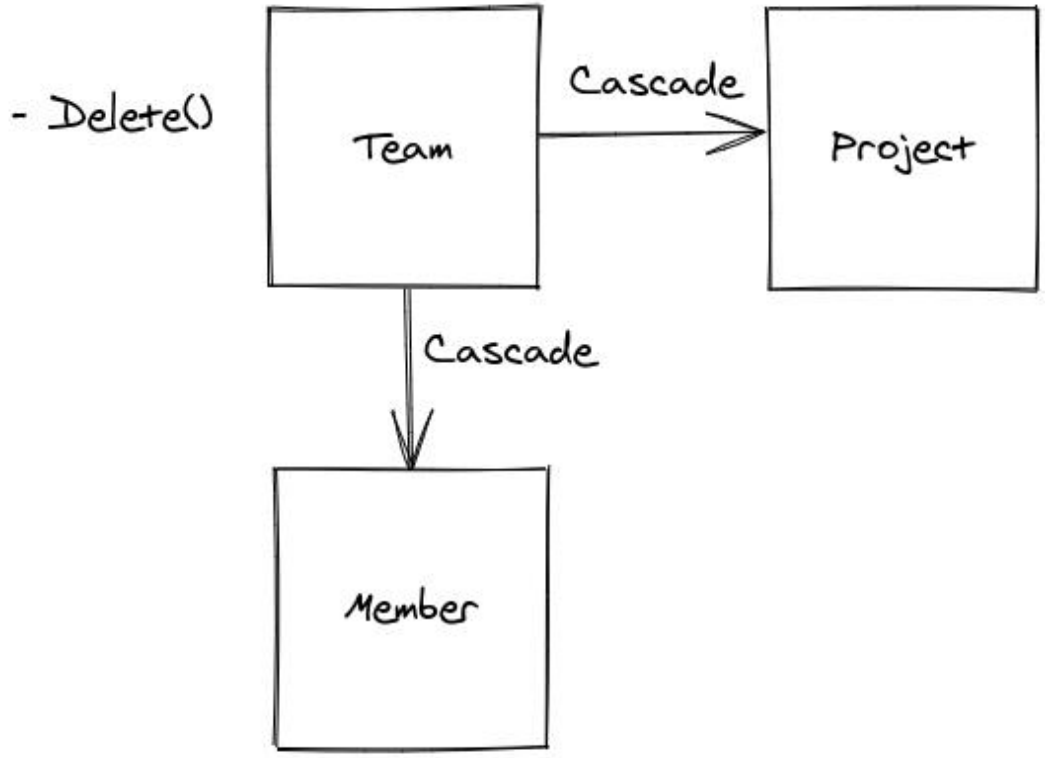
Aggregati: un esempio concreto





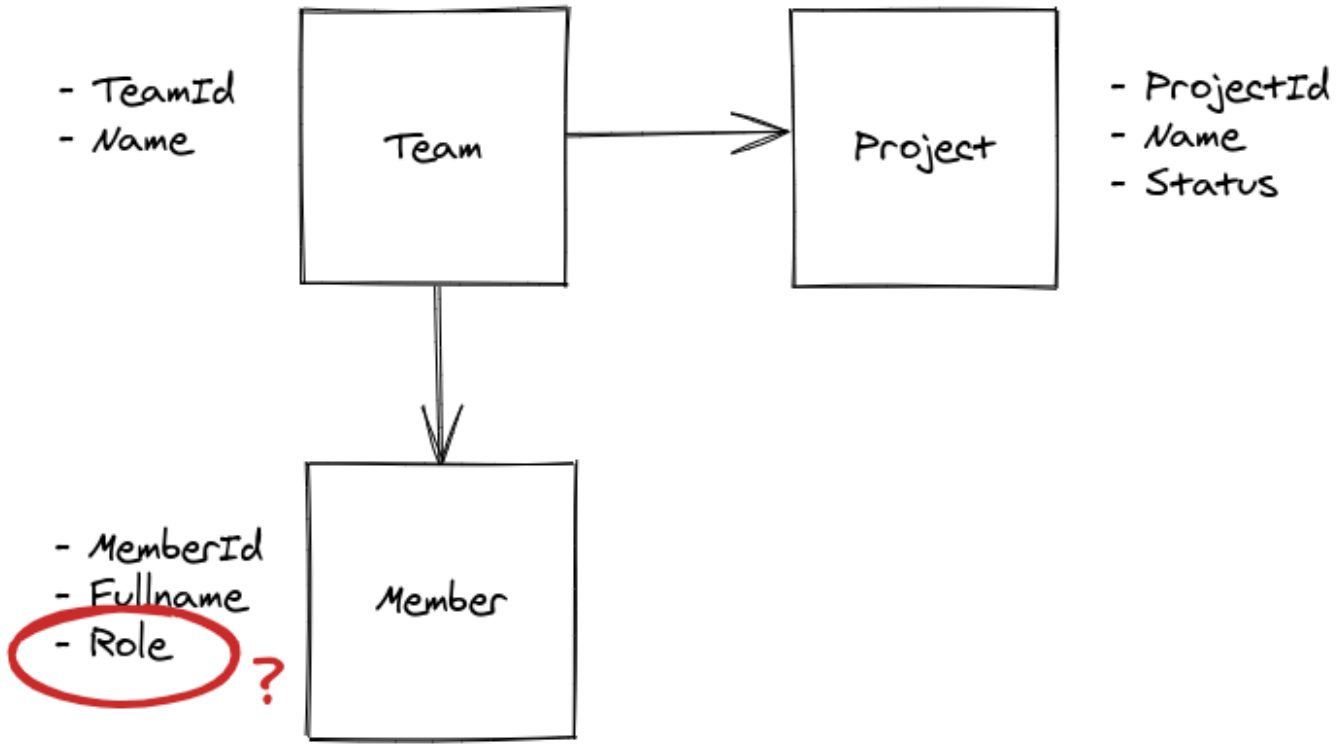
Aggregati: un esempio concreto

Non progettarlo seguendo il modello Entità-Relazione (ER) ma seguendo il comportamento dell'applicazione e quindi in base gli **eventi**

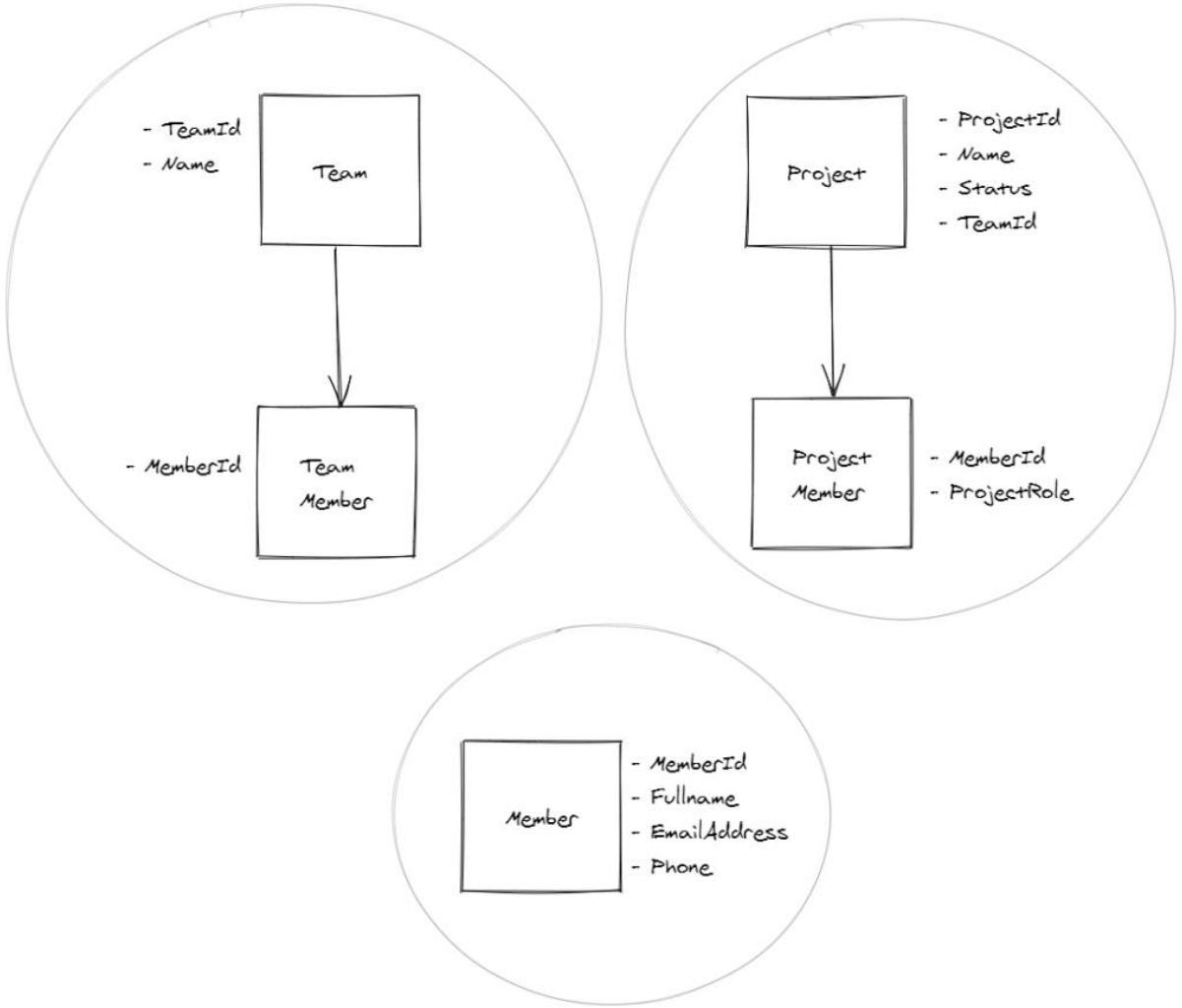


Aggregati: un esempio concreto

E se i requisiti cambiano, ad esempio il **ruolo** di un membro può variare in base al Team?



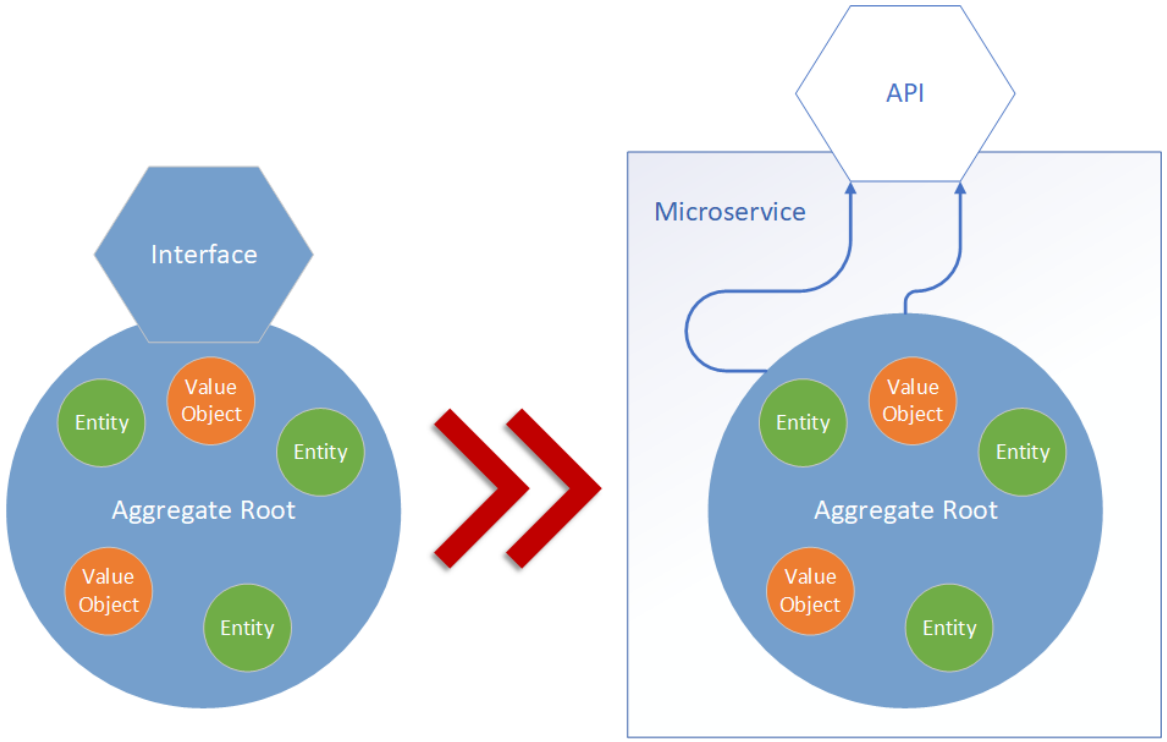
Aggregati: un esempio concreto



Identificazione dei microservizi

Quarto Step

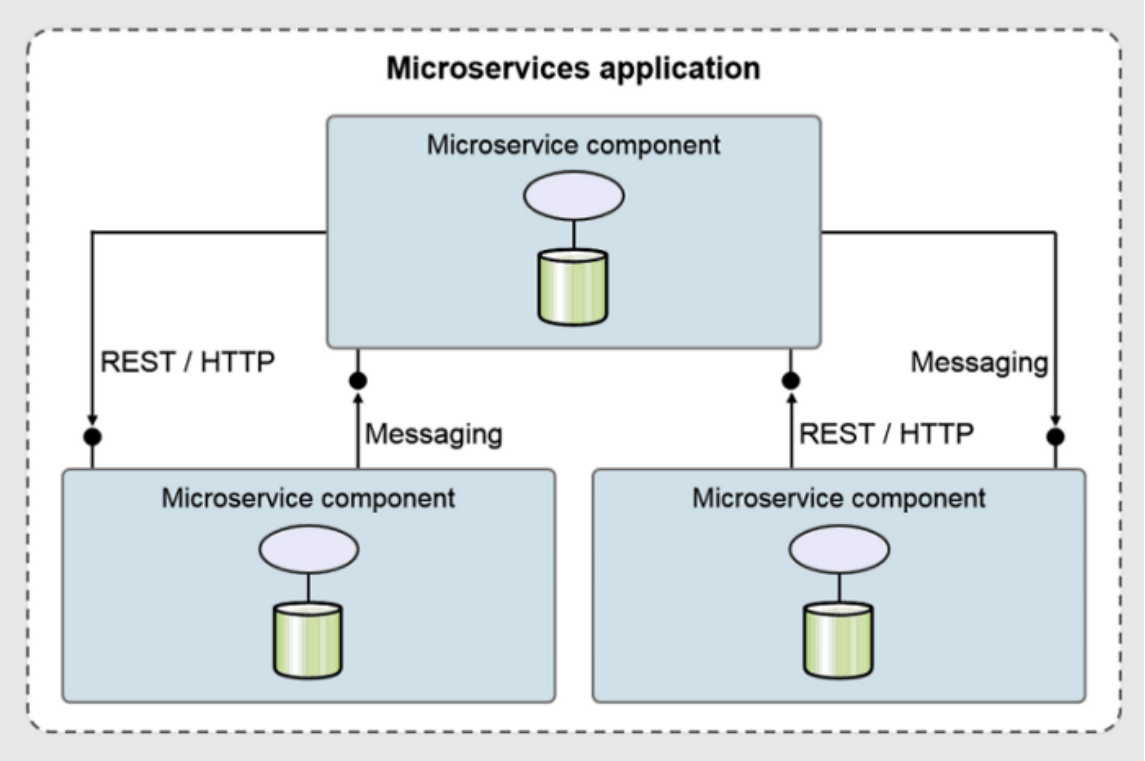
- Determinate le entità e gli aggregati è possibile individuare i microservizi dell'architettura
- Identificare i microservizi significa scegliere la giusta **granularità**
- Un aggregato incapsula la logica e le regole di business, tutte le complessità associate all'archiviazione e al recupero dei Value Objects
- Un aggregato si occupa di garantire **l'integrità** delle entità e dei Value Objects



Sviluppo dei microservizi

Quinto Step

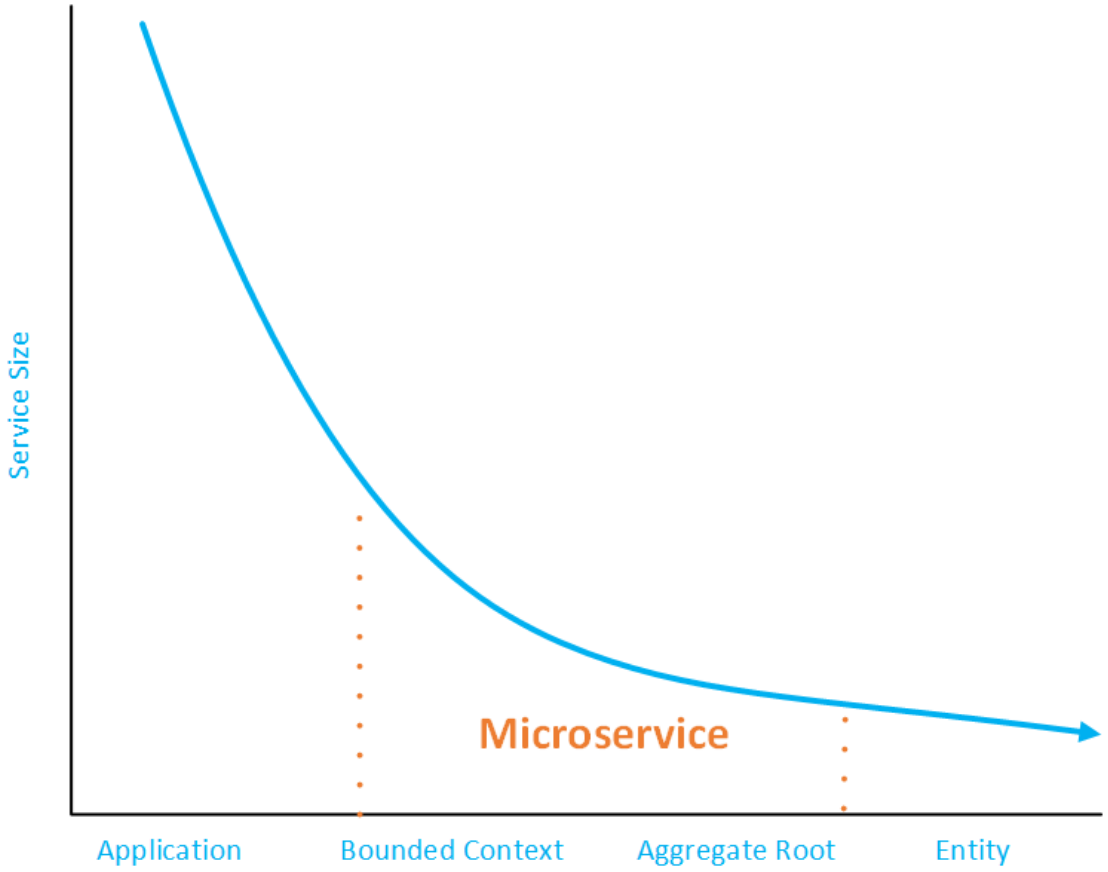
- Favorire l'utilizzo di sistemi di comunicazione **orientati agli eventi**
- Sviluppare l'architettura sfruttando la **eventual consistency**
- Utilizzare il pattern **Backend for Frontends (BFFs)** evitando di customizzare il backend per le troppe funzionalità
- Evitare orchestrazioni troppo complesse



Scegliere la granularità

Focal points

- La scelta del livello di granularità del singolo microservizio impatta **dimensione, complessità** e **prestazioni**
- Generalmente scegliere gli **aggregati** come base per i microservizi è la giusta soluzione
- In alcuni casi è possibile utilizzare direttamente i Bounded context a fronte di una maggiore dimensione
- Utilizzare direttamente le entità è invece sconsigliato



Conclusioni

- **Costoso**: la fase di analisi del dominio di business è costoso in termini di tempi e di risorse coinvolte
- **Necessita di esperti del dominio**: non è possibile applicare il DDD in assenza di figure esperte sul dominio di business da analizzare
- **Flessibilità**: il software viene modellato partendo dal dominio di business, è quindi flessibile a fronte di nuovi requisiti
- **Comunicazione**: grazie all'UL la comunicazione tra tecnici ed esperti del business è facilitata
- **Manutenibilità**: il dominio dati è modellato in modo modulare e incapsula le informazioni del business, risulta quindi facile da gestire

Reference

- https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/ddd-oriented-microservice
- https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns
- https://medium.com/walmartglobaltech/building-domain-driven-microservices-af688aa1b1b8
- https://medium.com/swlh/domain-driven-design-and-microservices-c62255790c3b
- https://programhappy.net/2020/10/18/implementing-domain-driven-design-with-microservices
- http://www.mokabyte.it/2008/11/domaindriven-1
- https://medium.com/walmartglobaltech/building-domain-driven-microservices-af688aa1b1b8
- https://www.jamesmichaelhickey.com/domain-driven-design-aggregates/

Questions & Answers

Progetti della community

SharpRedisMonitor

WebApplication for real-time monitoring of Redis, developed in ASP.NET Core with Razor Pages, fully compatible for both on-premise and Azure instances. The dashboard is developed with Bootstrap and AdminLTE.

redis adminlte dashboard csharp asp-net-core

JavaScript MIT 0 1 5 0 Updated 23 days ago

SharpHelpers

SharpHelpers is a collections of some handy code packages and tutorials to make developer's life easier

microsoft community library csharp dotnet helpers

1 package C# MIT 0 6 0 0 Updated on 5 Jan

Coding

Thank You!

Our Socials