

---

ANNO ACCADEMICO 2025/2026

---

# Tecnologie e Architetture Avanzate di Sviluppo Software

---

## Teoria

Altair's Notes



**UNIVERSITÀ**  
**DI TORINO**



---

DIPARTIMENTO DI INFORMATICA

---



## CAPITOLO 1 **INTRODUZIONE** **PAGINA 5**

1.1	Intro al Corso	5
	Esempio e Requisiti Non Funzionali — 6	
1.2	Panoramica Storica	6
	Dagli Anni '70 al 2000 — 6 • Dal 2000 ai Giorni Nostri — 9 • JavaEE e Cloud — 12	
1.3	Analisi dei Requisiti	13
	Architetture Monolitiche — 13 • Microservizi e DevOps — 14 • Domain-Driven Design — 16 • Modeling Practice — 17	
1.4	Strategic e Tactical DDD	19
	Strategic DDD — 19 • Tactical DDD — 22	
1.5	Ripasso su Spring Boot e React	23
	Maven — 23 • Gradle — 24 • SpringBoot — 25 • React — 28	

## CAPITOLO 2 **REST E DOCKER** **PAGINA 30**

2.1	REST	30
	Introduzione — 30 • Verbi e Status Responses — 31	
2.2	Docker	34
	Introduzione — 34 • Concetti Importanti — 35 • Architettura di Docker — 36 • Networking in Docker — 37 • Composing Containers — 38	

## CAPITOLO 3 **MICROSERVIZI** **PAGINA 40**

3.1	Comunicazione Sincrona	40
	RPC-Style — 40 • gRPC — 40 • Temi e Patterns — 42 • Richieste Duplicate e Idempotenza — 43	
3.2	Comunicazione Asincrona	44
	Comunicazione Orientata ai Messaggi — 44 • RabbitMQ — 46 • Temi — 48 • Il Pattern SAGA — 49	
3.3	Redis e Kafka	50
	Redis — 50 • Kafka — 51	



# Premessa

## Licenza

Questi appunti sono rilasciati sotto licenza Creative Commons Attribuzione 4.0 Internazionale (per maggiori informazioni consultare il link: <https://creativecommons.org/version4/>).



## Formato utilizzato

Box di "Concetto sbagliato":

### Concetto sbagliato 0.1: Testo del concetto sbagliato

Testo contenente il concetto giusto.

Box di "Corollario":

### Corollario 0.0.1 Nome del corollario

Testo del corollario. Per corollario si intende una definizione minore, legata a un'altra definizione.

Box di "Definizione":

### Definizione 0.0.1: Nome delle definizioni

Testo della definizione.

Box di "Domanda":

### Domanda 0.1

Testo della domanda. Le domande sono spesso utilizzate per far riflettere sulle definizioni o sui concetti.

Box di "Esempio":

### Esempio 0.0.1 (Nome dell'esempio)

Testo dell'esempio. Gli esempi sono tratti dalle slides del corso.

**Box di "Note":**

**Note:-**

Testo della nota. Le note sono spesso utilizzate per chiarire concetti o per dare informazioni aggiuntive.

**Box di "Osservazioni":**

**Osservazioni 0.0.1**

Testo delle osservazioni. Le osservazioni sono spesso utilizzate per chiarire concetti o per dare informazioni aggiuntive. A differenza delle note le osservazioni sono più specifiche.



# 1

## Introduzione

### 1.1 Intro al Corso

#### Parole chiave:

- Web Apps.
- Mission Critical.
- DevOps.
- Cloud Native.

#### Definizione 1.1.1: Mission Critical Applications

Un'applicazione o sistema le cui operazioni sono fondamentali per una compagnia o un'istituzione.

#### Osservazioni 1.1.1

- Enfasi sui requisiti non funzionali: i requisiti funzionali sono la baseline, ma ci si aspetta di più per rimanere competitivi.
- Da non confondere con life critical: non muore nessuno.

#### Definizione 1.1.2: Enterprise Application Integration (EAI)

Tutto l'insieme di pratiche architetturali, tecnologie, patterns, frameworks e strumenti che consentono la comunicazione e la condivisione tra diverse applicazioni nella stessa organizzazione.

#### Si ha enfasi sull'infrastruttura:

- *Data Integration*: combinare dati da più moduli diversi (coinvolge database).
- *Process Integration*: le interazioni tra più moduli.
- *Functional Integration*: si vuole fornire una nuova funzionalità sfruttando funzionalità già esistenti.



### 1.1.1 Esempio e Requisiti Non Funzionali

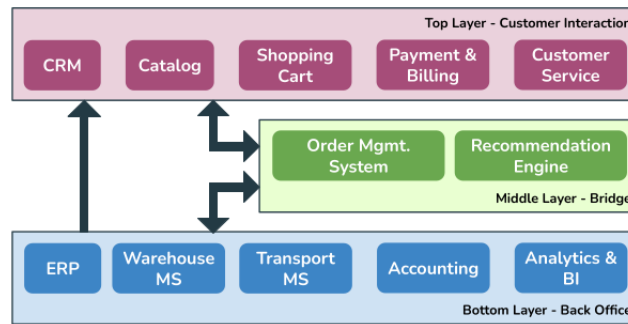


Figure 1.1: Esempio di e-commerce.

#### Commento dell'esempio:

- Ci sono tre livelli:
  - Top Layer: moduli che si rivolgono al cliente.
  - Middle Layer: gestione della comunicazione tra cliente e azienda.
  - Bottom Layer: moduli interni aziendali.

#### Requisiti non funzionali:

- High availability/zero downtime: l'applicativo deve essere sempre o quasi sempre disponibile.
- Affidabilità: in caso di interruzione di workflow si deve far sì che non ci siano stati danni (e.g. un'interruzione durante una transazione).
- Consistenza dei dati.
- Integrità dei dati.
- Low latency: per avere una buona performance, tutto deve essere fluido.
- Scalabilità.
- Sicurezza.
- Resilienza: capacità di reagire agli errori.
- Manutenibilità: quanto un pezzo di software sia mantenibile o riutilizzabile.
- Osservabilità: per comprendere eventuali problemi in un sistema distribuito.
- Auditability: le verifiche di qualità fatte su software<sup>1</sup>.

## 1.2 Panoramica Storica

### 1.2.1 Dagli Anni '70 al 2000

#### Definizione 1.2.1: Waterfall

Le metodologie a cascata<sup>a</sup> sono metodologie in cui ci sono fasi ben distinte e separate tra loro.

<sup>a</sup>Viste a "Sviluppo delle Applicazioni Software".

<sup>1</sup>Meglio visto in "Etica, Società e Privacy".

**Note:-**

È un modello prevedibile, ma lento a gestire i cambiamenti.

**Osservazioni 1.2.1**

- Software on the shelf: una volta acquistato è proprio.
- Software custom: prodotto su richiesta, ha bisogno di tutto un servizio di manutenzione.

**Definizione 1.2.2: Lean**

Metodologie nate negli anni '50 alla Toyota, verranno applicate al software dagli anni '90. Si basa su tre principi:

- Muda<sup>a</sup> (waste): si deve stare sui requisiti, non mettere troppe funzioni non necessarie.
- Mura (unevenness): è necessaria consistenza per aumentare la prevedibilità.
- Muri (overburden): non sovraccaricare le persone o le macchine. Non progettare software utilizzando strumenti greedy di risorse.

<sup>a</sup>JOJO'S Reference

**Note:-**

Lo strumento fondamentale è il *kanban*: la lavagna, per organizzare il lavoro.

**Definizione 1.2.3: Siloed**

Organizzazione aziendale a silos: si comunica poco e male. Ci sono 4 gruppi:

- BA Team: relazioni con gli stakeholders, requisiti, specifiche, documentazione.
- Dev Team: programma e fa un minimo di unit testing.
- Test Team: testa e decide se il sistema è pronto.
- Ops Team: si occupa del deployment.

**Note:-**

I vari team si parlano in maniera molto limitata.

**Definizione 1.2.4: Transaction Processing Monitor**

I TP monitor erano il primo esempio di soluzione middleware. Usata nei sistemi di mainframe erano: centralizzati, monolitici, mission critical, con accesso da vari terminali.

**Corollario 1.2.1 Middleware**

Software nel mezzo tra applicazioni e infrastrutture. Permette alle applicazioni di utilizzare le infrastrutture per farle comunicare tra di loro.

**Obiettivi:**

- Performance: si occupa di transazioni rispettando le proprietà ACID.
- Scalabilità: se un programma crasha ne avvia un'altra istanza.
- Affidabilità.
- Consistenza dei dati.

### Limiti:

- Proprietario.
- Tight coupling.
- Costosi.
- Complessi.

#### Domanda 1.1

Cosa rimane dei TP monitors?

- *Gestione delle transazioni e coordinazione:*
  - Soluzioni basate su 2PC (2 Phase Commit).
  - Le proprietà ACID, attualmente supportate internamente da molti database.
  - Proprietà BASE:
    - \* Basically: risposte basiche.
    - \* Available: si accetta che si possa non avere il dato più aggiornato.
    - \* State: la consistenza potrebbe non essere rispettata.
    - \* Eventually: prima o poi si riceverà il dato corretto.
- *Pool di connessioni.*
- *Distribuzione del carico:*
  - Le richieste vengono distribuite su varie istanze.
  - In caso di fallimento l'applicazione riparte.

#### Definizione 1.2.5: Remote Procedure Call

Si chiama una funzione da una macchina remota come se fosse locale. È indipendente dal linguaggio e a una struttura silos. Richiede aggiunte sia nello sviluppo che a runtime.

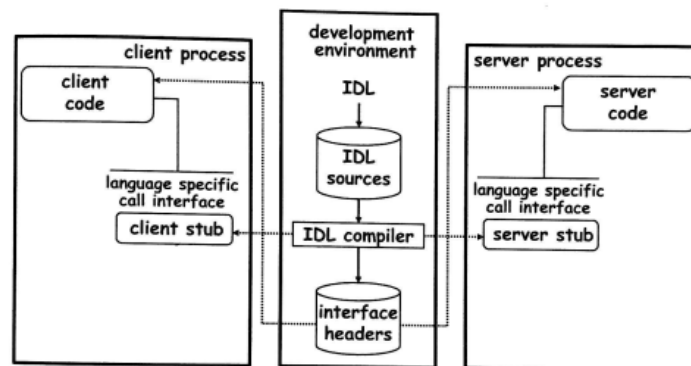


Figure 1.2: Remote Procedure Call - Development.

- Serializzazione: trasformare i dati in qualcosa che può essere comunicato.
- Marshalling: usa la serializzazione e inserisce meta-dati per permettere la ricostruzione della struttura dati.

#### Definizione 1.2.6: Common Object Request Broker Architecture (CORBA)

Evoluzione di rpc pensata per gli oggetti. Si possono creare oggetti in un server che possono rispondere a chiamate remote.

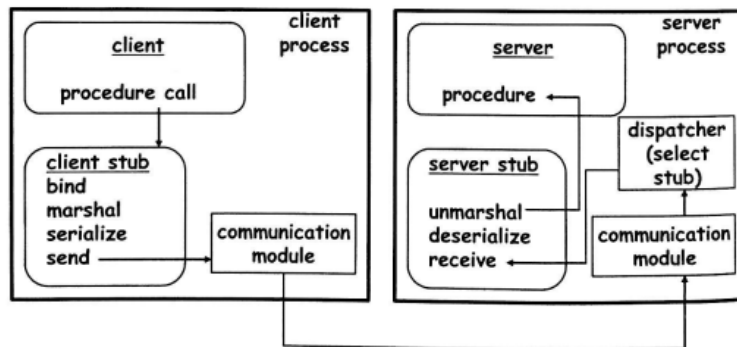


Figure 1.3: Remote Procedure Call - Runtime.

**Note:-**

Più successo lo ha avuto RMI (Remote Method Invocation) che è CORBA, ma solo con Java.

**Limiti:**

- Nascondere le cose al programmatore: si ha un falso senso di disaccoppiamento e i programmatori tendono a non vedere la rete.
- La programmazione sembra semplice perché i problemi vengono sottovalutati.

**Definizione 1.2.7: Message Oriented Middleware**

Invece di chiamarsi a vicenda le applicazioni si inviano messaggi a vicenda:

- Sincronizzazione tra operazioni in applicazioni diverse.
- Notifiche di eventi.
- Non c'è necessità di conoscere il ricevente.

**Due modelli di comunicazione:**

- Point-to-Point: il mittente manda un messaggio nella coda del middleware, il ricevente lo consuma.
- Publish and Subscribe: c'è una bacheca su cui chiunque può pubblicare un evento.

**Definizione 1.2.8: Enterprise Service Bus (ESB)**

Un middleware cosciente della logica di business. Si occupa di tradurre protocolli e dati.

**Note:-**

Caduto totalmente in disuso.

**1.2.2 Dal 2000 ai Giorni Nostri****Definizione 1.2.9: AGILE**

Metodologie fondate su iteratività e incrementalità.

**Corollario 1.2.2 XP - Xtreme Programming**

Si concentra sul codice, lo sviluppo di software si fa in team. Si dà importanza ai feedback sia dai clienti che dagli sviluppatori (small release, test-driven development, on-site customer).

### Principi di XP:

- Comunicazione.
- Semplicità.
- Feedback.
- Coraggio.
- Rispetto.

#### Definizione 1.2.10: Scrum

Prassi di organizzazione dell'attività lavorativa degli sviluppatori, si concentra sulla comunicazione:

- Organizzazione: esiste una lista del lavoro che deve essere svolto (Product Backlog e PDI), un'iterazione di lavoro di massimo 4 settimane (sprint) e deve esserci un incremento (valore percepibile dal cliente).
- Ruoli: ci si organizza in piccoli teams per ogni modulo.

### Ruoli in Scrum:

- Product owner: persona che gestisce il Backlog, in contatto con i clienti (non è il capo).
- Scrum master: organizza le riunioni, fa da mediatore.
- Development team.

### Eventi per ogni sprint:

- Sprint planning: riunione in cui si decide cosa fare.
- Daily scrum: meeting in piedi, deve durare poco.
- Sprint review: alla fine dello sprint, si mostra l'incremento agli stakeholders.
- Sprint retrospective: dopo la review, è una riunione interna al team.

#### Definizione 1.2.11: Kanban

Si vuole mantenere il flusso di lavoro. Non si mette più lavoro di quello che si riesce a fare.

### Principi di Kanban:

- Visualizzazione: si vede il proprio lavoro attraverso delle lavagne su cui vengono appiccicati post-it.
- WIP limit: si fanno un certo numero di cose contemporaneamente (non più di 3-4).
- Pull system<sup>2</sup>: le cose vengono spostate dal to do al doing quando si libera un posto.
- Continuous delivery: si integra la feature implementata e la si consegna.

### Board:

- Backlog.
- To do: roba da fare.
- Doing (WIP limit): roba che si sta facendo.
- Done: roba fatta.

#### Definizione 1.2.12: Scrumban

Scrum: ha i ruoli, il product Backlog e PBI, daily meeting, sprints.

Kanban: il flusso è pull-based e usa i WIP limit, le lavagne e i Continuous delivery.

<sup>2</sup>Gacha moment.

**Siloed evoluta:**

- Biz team: relazioni con gli stakeholders, marketing e vendite.
- Dev team: requisiti, sviluppo, testing e comunicazione con il biz team.
- Ops team: deploy, setting, validazioni.

**Note:-**

La divisione c'è ancora, ma c'è più comunicazione tra i vari team.

**Definizione 1.2.13: Service-Oriented Architecture**

Si inizia a ragionare sul fatto che l'integrazione debba avvenire mediante moduli che forniscono servizi l'uno all'altro.

**Domanda 1.2**

Cos'è un servizio?

**Corollario 1.2.3 Servizio**

Un servizio è una capacità di business autocontenuta che viene esposta secondo un contratto standard (un'interfaccia).

**I servizi:**

- *Coarse-grained*: ogni servizio implementa tutto (più pesanti dei microservizi).
- Condivide dati e funzioni attraverso interfacce (o API).
- *Scopribili*: i servizi si scoprono attraverso nomi e non IP.

**SOA:**

- Comunicazione attraverso applicazioni apposta o protocolli basati su HTTP.
- Le infrastrutture hanno un ruolo importante nel comporre i servizi in funzioni.
- Deployment centralizzato.

**Definizione 1.2.14: Web Services**

Istanza di Service-Oriented Architecture che stabilisce:

- Protocollo di comunicazione (SOAP):
  - XML su HTTP.
  - Consente sia comunicazione sincrona che asincrona.
- Service registry: UDDI
  - Elenco di servizi registrati secondo le loro features generali.
  - Consente ai servizi di essere scopribili.
  - Comunicazione mediante SOAP.
- Contratto (WSDL, Web Service Description Language):
  - Fornisce informazioni per contattare effettivamente un servizio.
  - La struttura dei messaggi.
  - Le strutture dati.
  - Protocollo e indirizzo.

### Osservazioni 1.2.2 Sui Web Services

Idealmente:

- Il client cerca il servizio su UDDI.
- Ottiene il link dal WSDL del servizio.
- Utilizzando WSDL collega dinamicamente il servizio alle operazioni.

In Pratica:

- La scoperta di servizi "in tempo reale" era impraticabile.
- I WSDL erano in maggioranza statici.
- Le informazioni venivano salvate in file di configurazione.

### 1.2.3 JavaEE e Cloud

#### Definizione 1.2.15: Enterprise Java Beans (EJB)

Gli EJB sono oggetti resi disponibili dinamicamente. Offrivano:

- Gestione del lifecycle.
- RMI.
- Sicurezza basata sui ruoli.
- Persistenza tramite Object-relational mapping (ORM).
- Gestione delle transazioni ACID.

I Java Beans erano pesanti:

- Oggetti collegati alla JVM (al container).
- Molto accoppiati all'ambiente di esecuzione.
- Necessitavano un java application server.
- Molto codice boiler-plate.
- Annotazioni XML.
- Non portabili.

#### Note:-

Tutto questo fino al 2006 in cui la terza edizione di EJB li fa diventare più leggeri:

- Annotazioni Java al posto di XML.
- POJOs (Plain Old Java Objects).
- JPA (Java Persistence).
- Si integrano con web service.
- Introduzione della *dependency injection*: design pattern per collegare due o più moduli tramite l'ambiente di sviluppo stesso.

**Definizione 1.2.16: Cloud**

Insieme di risorse sia computazionali, sia di storage, sia di networking. Queste risorse sono rese disponibili come servizi mediante API.

**Osservazioni 1.2.3**

- Il cloud è un'astrazione che nasconde la struttura fisica delle macchine.
- C'è un livello simile a un OS.
- I servizi sono offerti su base dichiarativa: diventa possibile avere un servizio che si conformi alle proprie necessità.

**Modelli:**

- *Infrastructure as a Service (IaaS)*: l'azienda mette a disposizione macchine virtuali, di storage o sottoreti visibili a chi compra il servizio.
- *Platform as a Service (PaaS)*: si acquista una piattaforma che nasconde cose e ottimizza.
- *Function as a Service (FaaS)*: si carica su una piattaforma una serie di funzioni e si sviluppa solo il front end.

**Applicazioni native sul cloud:**

- Moduli molto leggeri e loosely-coupled.
- Deployment containerizzato (impacchettato e Platform independent), orchestrazione (ignorante rispetto all'architettura ma che può operare su essa) e elasting scaling (cambiare il livello dei servizi).
- Dev Cycle features: integrazione continua, continuous delivery, deploy, infrastrutture dichiarative, consistenza tra dev/test/prod, anticipare i test sulla sicurezza, l'applicazione deve essere osservabile.
- NFRs: scalabilità, portabilità, sicurezza, evoluzione, mantenibilità, affidabilità.

**Definizione 1.2.17: DevOps**

Gestione del sistema mediante l'utilizzo di tools e pratiche basate sui principi Lean.

## 1.3 Analisi dei Requisiti

### 1.3.1 Architetture Monolitiche

**Domanda 1.3**

Ma ci serve un'architettura a microservizi?

**Definizione 1.3.1: Architettura Monolitica**

Un'architettura monolitica è semplice da sviluppare, non è distribuita, non ha integrazioni complesse, è facile da deployare e scalare, non ha coupling ed è facile da scalare.

**Note:-**

Può essere utile se piccola (non è vero, ma facciamo finta che lo sia). Il threshold è da 5 a 10 persone.



**Corollario 1.3.1 Monolithic Hell**

Il monolithic hell è un termine utilizzato per descrivere i rischi di un'applicazione monolitica.

**Monolithic Hell:**

- Teams che crescono e la coordinazione diventa ingestibile.
- Frammentazione della conoscenza: nessuno comprende tutto il sistema.
- I cambiamenti diventano rischiosi e costosi.
- All-or-Nothing update: o si cambia tutto o non si cambia nulla.
- I deployment possono essere lenti e causare downtime.
- Un fail su una funzione può buttare giù tutto il sistema.

	<b>SOA</b>	<b>Microservices</b>
<b>Scope</b>	business high level function (coarse)	subdomain / single purpose (fine grained)
<b>Communication</b>	Smart pipes (e.g. Enterprise Service Bus), heavyweight protocols	Dumb pipes (e.g. message brokers or service-to-service with lightweight protocols - REST, gRPC)
<b>Storage</b>	Shared databases	Database-per-service
<b>Deploy</b>	Tightly coupled services sharing same stack	Loosely coupled "polyglot" services
<b>Scalability</b>	Difficult, due to shared dependencies	Highly scalable, each service scales independently

Figure 1.4: SOA vs. Microservizi.

**Microservizi e Miniservizi:**

- I microservizi propriamente detto dovrebbe implementare una sola funzione.
- I miniservizi svolgono un'unità funzionale coesa e coerente, ma non necessariamente una sola funzione.

**Note:-**

Per chi fa questa distinzione quelli che vedremo nel corso sono considerati miniservizi.

**1.3.2 Microservizi e DevOps**

Nella figura 1.5:

- L'architettura a microservizi permette un'organizzazione autonoma e teams polifunzionali.
- L'organizzazione in questi teams permette a sua volta il continuous delivery e il continuous deployment.
- I teams devono essere piccoli in modo che possano essere eventualmente riorganizzati.
- In sostanza: dividere un'app in microservizi (invece che monolitica) consente un deployment costante. Per gestire i microservizi si fa affidamento su piccoli teams autonomi, che vanno a rinforzare il DevOps.

**Definizione 1.3.2: API**

Le API sono un insieme di regole e protocolli che permettono a software diversi di "parlare" tra di loro.

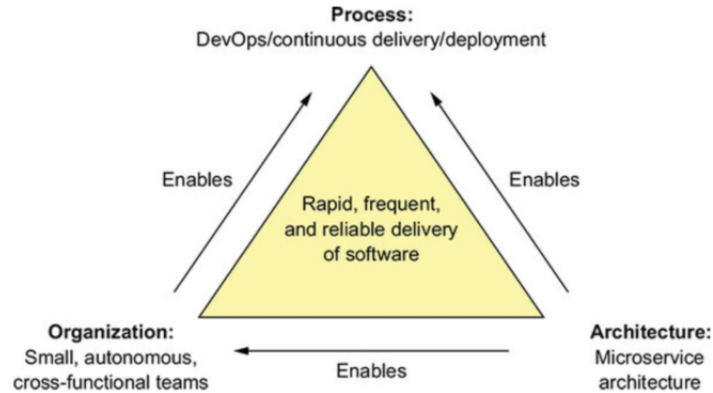


Figure 1.5: Microservizi e DevOps.

### Architettura a microservizi:

- Normalmente viene eseguita su un server che espone al cliente delle cose (per esempio user interface).
- C'è un modulo dedicato che funge da gateway. Il front-end si collega a un indirizzo web tramite esso.
- Ogni servizio espone un API REST.

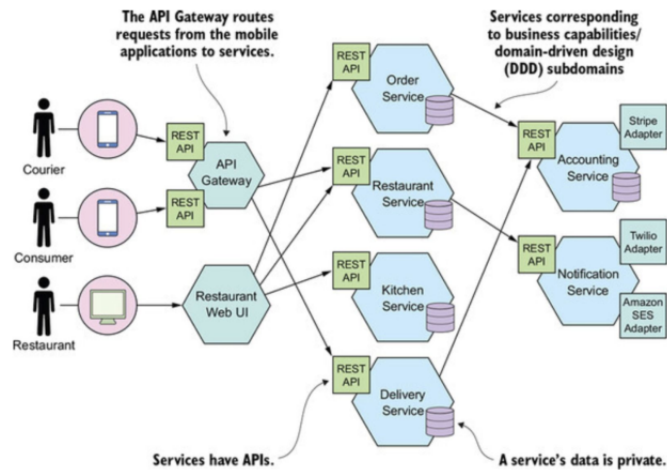


Figure 1.6: API

#### Definizione 1.3.3: Conway's Law

Le organizzazioni che producono softwares sono obbligate a produrre softwares che sono copie delle strutture comunicative dell'azienda.

#### Corollario 1.3.2 Inverse Conway Maneuver

L'idea è quella di strutturare la propria organizzazione in modo tale che la struttura rispecchi la propria architettura a microservizi. Così facendo i dev teams sono debolmente collegati ai servizi.

#### Note:-

"Spesso i softwares delle organizzazioni pubbliche come l'università non è granché", *citazione necessaria*.

### Sfide dei microservizi:

- Complessità: un'architettura a microservizi è un sistema distribuito.
- Richiede ristrutturazione dell'organizzazione aziendale.
- Deve tenere conto della performance della rete:
  - Evitare *chatty service*: servizi che si scambiano tanti piccoli messaggi.
  - Evitare messaggi enormi.
  - Minimizzare la latenza.
- Misure di sicurezza per ogni servizio.

### 1.3.3 Domain-Driven Design

#### Definizione 1.3.4: Domain-Driven Design (DDD)

Il Domain-Driven Design è un approccio al design software che nasce intorno alla nozione di "modello di dominio":

- Il modello di dominio cattura in una maniera formale, ma concettuale, i concetti rilevanti, le entità, le relazioni e le regole di uno specifico business.
- Il modello di dominio è l'output dell'analisi dei requisiti ed è l'input della fase di design.
- Si utilizza un approccio AGILE.

#### Note:-

L'idea di questo approccio: tutto il DevOps ha un'idea chiara del dominio e delle sue regole.

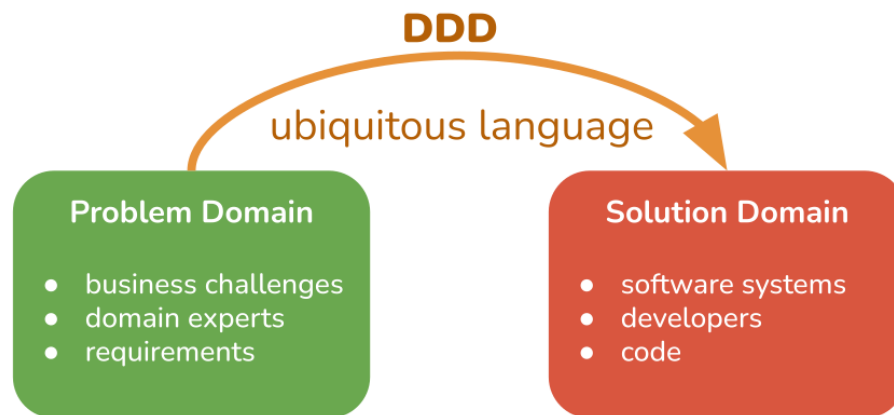


Figure 1.7: Dal problema alla soluzione.

#### Definizione 1.3.5: Ubiquitous Language

Un linguaggio comune costruito insieme e condiviso dagli esperti del dominio e dal development team. Deve essere usato:

- Nel glossario.
- In tutta la documentazione.
- Nel codice.

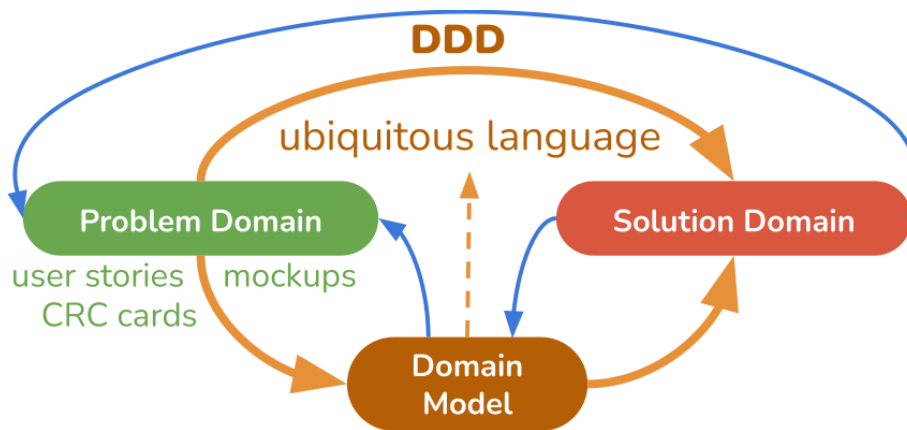


Figure 1.8: Connessione tra problema e soluzione.

**Osservazioni 1.3.1**

- Non ci si può aspettare che un modello di dominio rifletta completamente il mondo reale.
- È una rappresentazione selettiva della prospettiva del problema che si cerca di risolvere.
- Astrazioni, confini e conoscenza condivisa.
- "Tutti i modelli sono sbagliati, ma alcuni sono utili".



Figure 1.9: Quando si utilizza DDD.

**1.3.4 Modeling Practice****Definizione 1.3.6: Event Storming**

L'event storming è una tecnica di modellazione collaborativa in cui si esplora un dominio, si scrivono gli eventi rilevanti su dei post-it e li si attacca alla parete. Dopo di che si creano delle sequenze cronologiche per far capire quali sono i flussi.

**Note:-**

Ciò permette di chiarire eventuali ambiguità o fraintendimenti.

#### Definizione 1.3.7: Value Streams

Un value stream è una sequenza end-to-end di attività che un'organizzazione effettua per portare un valore a un cliente o a uno stakeholder.

#### Note:-

Deve mostrare chiaramente il valore, essere comprensibile alle altre entità coinvolte e deve essere in terza persona.

#### Corollario 1.3.3 Support Streams

Come i value streams ma interni all'azienda, non riguardano direttamente il cliente finale.

#### Definizione 1.3.8: User Stories

Una user story è una breve narrativa che descrive un processo o un goal dal punto di vista di un solo attore. Cattura le motivazioni, le azioni e il risultato desiderabile.

#### Note:-

Una variante sono le AGILE user stories che sono composte da una sola frase (As a [actor], I want [goal], so that [reason]).

#### Definizione 1.3.9: CRC Cards

Le Class/Responsability/Collaborator (CRC) cards sono carte che:

- Catturano le entità rilevanti e le loro relazioni.
- Possono essere introdotte in diversi stadi del processo.
- Inizialmente sono usate per fare brainstorming.
- L'obiettivo è quello di far emergere termini chiavi usati negli ubiquitous languages.
- Un tipo di entità → Una carta.

#### CRC:

- *Classe*: il tipo di entità, classe concettuale.
- *Responsability*: quale entità di questo tipo *conosce* o *fa*.
- *Collaborators*: altre entità con cui si interagisce per soddisfare le responsibility.

#### Il processo:

- *Identificare* le entità rilevanti in una storia. Gli NPC contano come entità mentre MC no. Per ognuno si fa una carta con il nome e le responsibilities.
- *Role-Play* la storia con gli altri membri del team per:
  - Scoprire le relazioni.
  - Trovare responsibilities mancanti.
  - Trovare entità mancanti.
- Se un'entità diventa troppo grande:
  - Si splitta in due entità.
  - Si stanno confondendo più punti di vista di un'entità.

## 1.4 Strategic e Tactical DDD

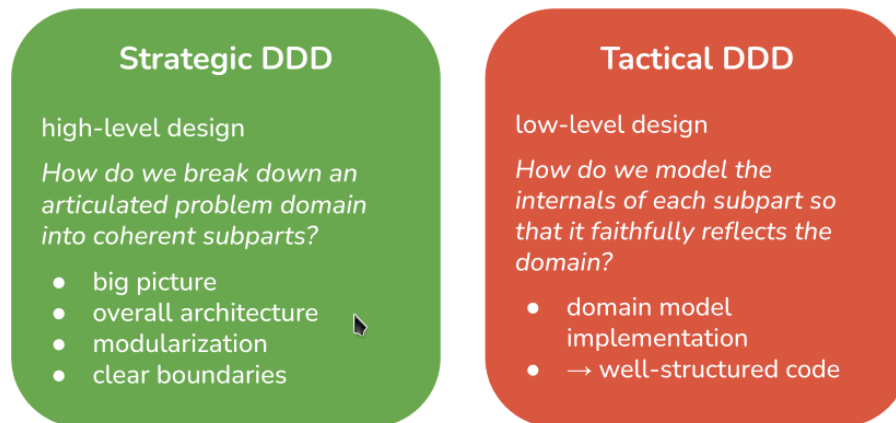


Figure 1.10: Strategic DDD vs. Tactical DDD.

### 1.4.1 Strategic DDD

#### Definizione 1.4.1: Subdomains

Dipartimenti diversi possono avere modelli di dominio o linguaggi diversi. Differenti subdomains possono:

- Utilizzare lo stesso nome per entità concettuali diverse.
- Utilizzare nomi diversi per la stessa entità concettuale.
- Considerare la stessa entità concettuale da punti di vista diversi.
- Considerare relazioni diverse tra le stesse entità.

**I subdomains possono essere:**

- Core: il "cuore" delle attività delle aziende, attività che non offrono altre aziende.
- Generic: molto comuni nelle aziende (e.g. parte finanziaria, gestione clienti). Possono essere "acquistate" da pacchetti già esistenti.
- Supporting: nel mezzo, abilitano il core business e hanno complessità variabile.

#### Definizione 1.4.2: Bounded Context

Un bounded context rappresenta un'area all'interno di un modello in cui il linguaggio è univoco e consistente, la logica è coerente ed è possibile immagazzinare al suo interno una parte della business logic esponendo degli endpoints.

#### Osservazioni 1.4.1

I subdomains sono il punto di partenza:

- one-to-one: un subdomain è legato a un bounded context.
- many-to-one: un bounded context è legato a più subdomains:
  - La maggior parte delle operazioni richiede una stretta interazione tra funzioni in subdomains diversi.

- C'è un overlapping tra data models.
- Scelte tecnologiche.
- one-to-many:
  - Dall'analisi emerge la possibilità di fare decoupling.
  - La tecnologia permette uno split: front-end vs. back-end.

**Note:-**

I bounded context sono legati da *pattern di dinamiche di potere*.

**Definizione 1.4.3: Separate Ways**

Non ci sono interazioni tra due bounded context.

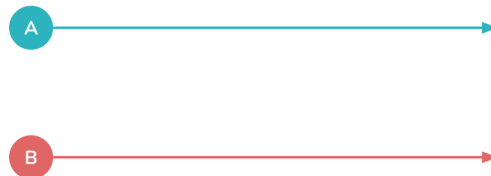


Figure 1.11: Separate Ways.

**Definizione 1.4.4: Partnership**

Due bounded context che lavorano in stretto contatto.

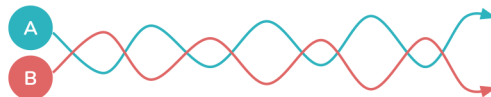


Figure 1.12: Partnership.

**Definizione 1.4.5: Customer Supplier**

Un bounded context fornisce dei servizi a un altro.

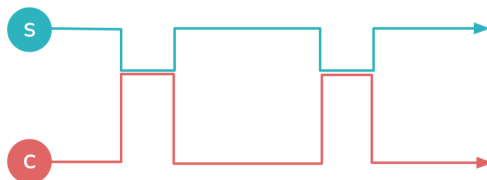


Figure 1.13: Customer Supplier.

**Definizione 1.4.6: Conformist**

Il customer si deve adeguare al supplier.

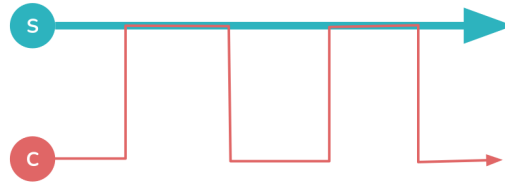


Figure 1.14: Conformist.

**Note:-**

Conformist è tipico dell'utilizzo di API di librerie di terze parti.

**Definizione 1.4.7: Shared Kernel**

Patterns che assumono che due bounded context abbiano un modello di dominio indipendente:

- Crea tight coupling tra i due bounded context.
- È necessario avere una comprensione dei concetti chiave tra i due teams.
- Ci deve essere disponibilità tra due teams a collaborare (partnership).
- La parte condivisa deve evolversi lentamente.



Figure 1.15: Shared Kernel.

**Definizione 1.4.8: Anti-Corruption Layer**

Un livello di traduzione tra il bounded context e altri modelli esterni:

- Evita contaminazioni da altri bounded context o fornitori esterni.
- Difesa contro Conformist: un team non deve più fare design sulle scelte dell'altro.
- È molto costosa come scelta.

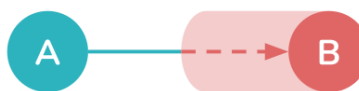


Figure 1.16: Shared Kernel.



### Tre tipi di Front-Ends:

- Front-End specifico di uno specifico bounded context:
  - L'utente interagisce solo con un singolo bounded context e ne usa l'ubiquitous languages.
  - Il front-end è solo questioni di scelte tecnologiche.
- Portale/Federazione, multi bounded context:
  - Ogni bounded context ha le sue capacità.
  - Il front-end deve fornire un container uniforme.
  - Per esempio il portale dell'ateneo (bleah!).
- Orchestratore:
  - Il front-end orchestra un processo che interagisce con multipli bounded context.
  - Il front-end è un bounded context di suo.

#### Definizione 1.4.9: Mockups

Disegno che mostra la UI e come gli utenti possono interagire con essa.

### I primi Mockups:

- Seguono user stories.
- Mostrano le interazioni.
- Testano i bounded context.

## 1.4.2 Tactical DDD

#### Definizione 1.4.10: Tactical DDD

Il Tactical DDD si occupa di implementare un subdomain model in un bounded context.

### Il Tactical DDD:

- È agnostico rispetto all'implementazione:
  - Non c'è un commitment verso uno specifico linguaggio.
  - Descriviamo il COSA, non il COME.
- Diagrammi UML con specifici stereotypes DDD:
  - Ruoli delle classi in un bounded context.
  - Possono essere rappresentati in UML come stereotypes.

### Parti del Tactical DDD:

- Entity: un concetto definito dalla sua identità piuttosto che dai suoi attributi. Rappresenta un qualcosa che persiste e cambia nel corso del tempo:
  - Implementazione anemica: ha solo ed esclusivamente i dati.
  - Implementazione ricca: include le responsabilità.
- Value Object: un concetto di dominio immutabile definito dai suoi attributi. Gli attributi sono impostati alla creazione del concetto e non possono cambiare.
- Aggregato: un insieme di oggetti che deve essere tenuto consistente e trattato come un'unità:
  - Un'entità viene scelta come root aggregate che permette l'accesso alle entità.
  - Il root aggregate è responsabile delle regole di business.

**Patterns in DDD:**

- **Factory:** si usa quando la creazione di un oggetto non può essere delegata all'oggetto stesso perché bisogna tenere conto di regole che riguardano più oggetti. Si crea un *factory object*.
- **Repository:** un livello di astrazione che permette di fare decoupling tra il dominio di business e la sua rappresentazione di persistenza. Permette di cambiare il modo in cui vengono gestiti i files (e.g. passare da files a database).
- **Service:** un orchestratore che sa come gestire le varie operazioni sulle entity.

## 1.5 Ripasso su Spring Boot e React

**Note:-**

DISCLAIMER: è il mio primo approccio alla programmazione web (dato che sono specializzata in robe teoriche e/o a basso livello) per cui potrei fare qualche imprecisione, sorry.

### 1.5.1 Maven

Dato che gli IDE moderni consumano un sacco di batteria e risorse includo anche una mini guida per setuppare un progetto java con Maven (in questo modo potete usare vim, gedit o nano se vi va). Se usate IntelliJ, Vs Code o altro potete saltare<sup>3</sup>.

**Definizione 1.5.1: Maven**

Maven è un tool per creare automaticamente delle build di progetti java. Permette di compilare codice, fare testing, packaging, etc.

**Note:-**

Maven utilizza il *Project Object Model (POM)* per descrivere la configurazione di un progetto e gestire le dipendenze.

**Domanda 1.4**

Come si crea un progetto con Maven?

Listing 1.1: Creazione di un progetto Maven

```
mvn archetype:generate \
  -DgroupId=com.example \
  -DartifactId=myapp \
  -DarchetypeArtifactId=maven-archetype-quickstart \
  -DinteractiveMode=false
```

**Nello specifico:**

- **DgroupId** indica il nome di una compagnia o di un'organizzazione.
- **DartifactId** indica il nome del progetto.
- **DarchetypeArtifactId** indica il template (in questo caso un semplice HelloWorld java).

<sup>3</sup>Questi IDE possono utilizzare anche Maven, ma lo gestiscono loro.

Listing 1.2: Esempio di pom.xml per Spring Boot

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.3.4</version>
    <relativePath/>
  </parent>

  <groupId>com.example</groupId>
  <artifactId>myapp</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>myapp</name>

  <properties>
    <java.version>24</java.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
</project>

```

**Spiegazione:**

- Il **parent** imposta la versione di Spring Boot e le configurazioni di default.
- Le **dependencies** includono il modulo web e quello per i test.
- Il plugin **spring-boot-maven-plugin** permette di eseguire l'app con `mvn spring-boot:run`.

**1.5.2 Gradle**

Per alcune persone può essere più facile utilizzare Gradle (inclusa me), quindi aggiungo qualcosa anche per questo.

**Definizione 1.5.2: Gradle**

Come Maven, Gradle è un tool per creare automaticamente progetti java, C/C++, kotlin, etc. A livello di base ha le stesse funzionalità di Maven, le differenze principali sono il linguaggio utilizzato (Maven è basato su xml, Gradle su Groovy), velocità (Gradle è più veloce per build incrementali), etc.

**Domanda 1.5**

Come si crea un progetto Spring Boot con Gradle?

Listing 1.3: Creazione di un progetto Spring Boot con Gradle

```
curl https://start.spring.io/starter.tgz \
  -d type=gradle-project \
  -d dependencies=web \
  -d groupId=com.example \
  -d artifactId=test \
  -d name=test \
  -d packageName=com.example.test \
  -o test-gradle.tgz
tar -xvf test-gradle.tgz
cd test
```

**Alcune osservazioni importanti:**

- Di default il progetto creato usa java 17, per cambiarlo basta andare nel file `build.gradle`.
- Inizialmente darà errore perché non si sono definiti endpoint.

Listing 1.4: Avvio del progetto Spring Boot con Gradle

```
# Su Linux/macOS
./gradlew bootRun

# Su Windows
gradlew.bat bootRun
```

**Note:-**

Al primo avvio Gradle scaricherà tutte le dipendenze necessarie. Una volta completato, l'app sarà disponibile su `http://localhost:8080/`. Se non hai ancora definito controller o endpoint, vedrai la *Whitelabel Error Page*.

**1.5.3 SpringBoot****Note:-**

Non descriverò come fare un progetto su IntelliJ, se non ci riuscite è skill issue.

**Definizione 1.5.3: SpringBoot**

SpringBoot è un frameworks open-source per la programmazione di webapp.

Spring usa il pattern MVC:

- **Controller:** punto di ingresso delle richieste esterne.
- **Model:** consultato dal controller quando arriva una richiesta.
- **View:** prodotta dal controller.

#### Domanda 1.6

Come si fa a fare un controller in spring?

#### Definizione 1.5.4: Annotazioni java

Modo per aggiungere metadati nel codice java. Forniscono informazioni extra al compilatore.

Annotazioni in spring:

- **@RestController:** fa capire a spring che è un controller e quindi deve stare in attesa di richieste HTTP.
- **@RequestMapping(...):** specifica dove devono arrivare le URL. Per esempio in una classe `TavoliController` può esserci l'annotazione `@RequestMapping("/tavoli")`.
- **@GetMapping(...):** handler per le varie richieste. Il suo contenuto viene aggiunto dopo la root string specificata da `@RequestMapping(...)`. Per esempio `@GetMapping({id})` indica che si vuole un tavolo con un determinato menu.
- **@PathVariable:** si mette nella signature dei metodi per passare i parametri presi dal `@GetMapping(...)`. Per esempio `ResponseEntity<Tavolo> getTavoli(@PathVariable int id)` va a utilizzare l'id specificato in precedenza. Questo porta a due casi:
  - Se l'id c'è si restituisce una `ResponseEntity.ok(...)`.
  - Se non c'è viene restituito `ResponseEntity.notFound().build()`<sup>4</sup>. Il `notFound()` non dà una risposta definitiva, ma permette di aggiungere altre caratteristiche (è il `build()` che formula la risposta HTTP).



Figure 1.17: Not found.

- **@PostMapping(...):** il controller si aspetta che la richiesta abbia un body. Per esempio `@PostMapping("crea")` si occupa di creare un ordine.
- **@RequestBody:** si aspetta un body (quindi si usa nelle `@PostMapping(...)`), per cui un oggetto JSON che può essere deserializzato. Questo può portare a:

<sup>4</sup>Il famigerato 404.

- `ResponseEntity.badRequest().build()`: significa che l'utente ha sbagliato qualcosa nella richiesta (e.g. chiedere di entità che non esistono).
- `ResponseEntity.InternalServerError().build()`: errore nell'esecuzione per cui il server non riesce a soddisfare la richiesta.
- `@Service`: l'istanza dell'oggetto viene creata automaticamente da spring. Viene creata a partire da un costruttore.
- `@PostConstruct`: metodi che vanno invocati subito dopo la costruzione del `@Service`.
- `@RestControllerAdvice`: per il `GlobalExceptionHandler`, la gestione di errori di alto livello.
- `@ExceptionHandler(...)`: quando arriva un'eccezione la gestiscono.

#### Jakarta (persistence), per gestire i database:

- `@Entity`: la classe corrisponde a una tabella del database.
- `@Table(...)`: per specificare il nome della tabella (di default è il nome della classe).
- `@Id/@EmbeddedId`: campo chiave.
- `@GeneratedValue(...)`: per scegliere la strategia di generazione e altri parametri.
- `@Column(...)`: una colonna della tabella, si possono specificare varie opzioni come lunghezza o nullable.
- `@JoinColumn(...)`: per effettuare join, va inoltre specificato il tipo di relazione con un'altra annotazione (`@ManyToOne(...)`, `@OneToOne`, etc.).
- `@MapsId(...)`: quando ci sono più id che indicano la stessa cosa.
- `@Enumerate(...)`: di default gli enum vengono tradotti come numeri, con questa annotazione si possono specificare altri tipi come String.

#### Note:-

È importante che sia presente un costruttore vuoto per l'entità.

#### Per avere corrispondenza tra gli oggetti e le entità del database si devono definire dei repositories:

- Sono interfacce che estendono `JpaRepository<>`.
- Si specificano il tipo di oggetto e il tipo di id.
- Si può utilizzare l'annotazione `@Repository`, ma non ha sostanzialmente effetto.
- Questi repositories sono connessi ai services.
- `@Transactional`: importante per tenere traccia delle modifiche. Si usa nei metodi dei services.

#### Definizione 1.5.5: Applicazione Headless

Un'applicazione che non restituisce delle pagine, ma dei JSON. I files JSON possono essere intesi come view nel pattern MVC.

### Spring Security:

- Appena inserito lo starter viene creato un utente fittizio.
- Da questo momento tutte le operazioni necessitano un'autenticazione, se non la si ha viene restituito 401 - Unauthorized.
- I metodi POST sono soggetti a Cross-Site Request Forgery (CSRF), per prevenire richieste false da parte di malintenzionati. Per risolvere bisogna configurare la Security.
- Si aggiunge al progetto la classe SecurityConfig con l'annotazione @Configuration.
- Il metodo filterChain è annotato come Bean, ossia un *singleton*.
- @EnableMethodSecurity: i controlli verranno fatti nel controller.
- @PreAuthorize(...): va a specificare condizioni tipo se l'user ha un ruolo specifico. Se non è presente i metodi sono accessibili a chiunque sia autenticato.

### 1.5.4 React

#### Definizione 1.5.6: React

React è un framework javascript per fare user interface. Segue il pattern MVVM.

#### Note:-

PS. usate typescript e non javascript.

#### Corollario 1.5.1 MVVM

Il pattern Model-View ViewModel in cui è presente il ViewModel che è strettamente collegato alla view.

#### Definizione 1.5.7: npm

npm è un package manager per javascript. Gestisce le dipendenze dei progetti. Con il comando `npm install` vengono inserite le dipendenze in una cartella "node\_modules".

### Files:

- `index.html`: il punto di ingresso dell'applicazione.
- `main`: il file deciso nell'index.
- `app`: l'effettiva app che contiene tutti i componenti di react.

#### Note:-

I componenti di react sono in `tsx` che permette di unire typescript e HTML.

### I componenti React:

- Sono definiti andando a definire la loro funzione di rendering.
- Si tratta di HTML sottoposto a interpolazione con espressioni javascript/typescript.
- Si possono mettere dei tag con dei componenti, andando a creare gli alberi di componenti.
- Lo stato è trattenuto internamente da react e viene chiamato nelle funzioni di rendering.
- Le funzioni devono essere pure (no side-effect, haskell-like).
- Le props sono gli input delle funzioni.





# 2

## REST e Docker

### 2.1 REST

#### 2.1.1 Introduzione

##### Definizione 2.1.1: REST

REST è uno stile architetturale per creare APIs in rete: specifica la forma e la semantica della comunicazione tra un provider (server) e un consumer (client).

##### Note:-

REST sta per REpresentational State Transfer.

##### Osservazioni 2.1.1

- Il provider fornisce *risorse concettuali*.
- Il consumer può manipolare una risorsa scambiando sue rappresentazioni.
- Le interazioni avvengono mediante richieste *stateless* con un'interfaccia uniforme (HTTP, URI, etc.).
- Solitamente si implementa con HTTP (mappa naturalmente con i principi REST).
- Sia il provider che il consumer devono essere *compliant* con la specifica API concordata.

##### Corollario 2.1.1 REST Resources

Gli Endpoint URIs (pathname) dovrebbero esprimere risorse (nomi), non azioni. Mappano sui DDD:

- Part di path rappresentano entità (tipi + id).
- Path annidate rappresentano oggetti aggregati o contenuti.



Figure 2.1: Esempio di mappatura REST resources su DDD.

### 2.1.2 Verbi e Status Responses

#### Corollario 2.1.2 REST Verb

I verbi esprimono le azioni effettuate su una risorsa. I verbi di REST mappano su metodi HTTP.

I verbi:

- GET ottiene una rappresentazione di una risorsa specifica (legge dei dati):
  - Non ha side effects ed è idempotente.
  - Non ha un body.
  - *Query String*: usata per aggiungere filtri sulla richiesta.

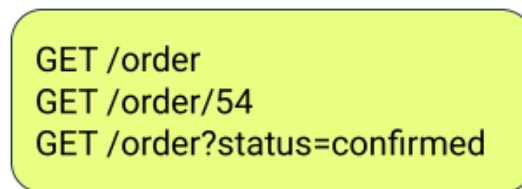


Figure 2.2: REST GET.

- POST viene usata per creare una nuova risorsa:
  - Ha side effects e non è idempotente.
  - Il body contiene la rappresentazione della nuova risorsa.
  - *Query String*: usata per esprimere modificatori per la richiesta.



Figure 2.3: REST POST.

- PUT rimpiazza una risorsa esistente con una nuova rappresentazione:
  - Ha side effects, ma è idempotente.
  - Il body contiene una rappresentazione aggiornata della risorsa.
  - *Query String*: usata per aggiungere filtri sulla richiesta (come in POST).



Figure 2.4: REST PUT.

- DELETE cancella una risorsa esistente:
  - Ha side effects, ma è idempotente.
  - Non ha un body (come in GET).
  - *Query String*: usata per aggiungere filtri sulla richiesta (come in POST).

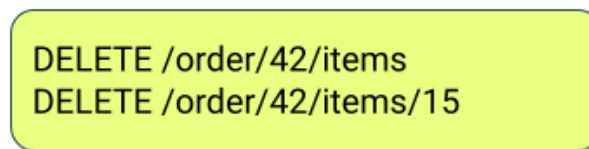


Figure 2.5: REST DELETE.

- PATCH aggiorna solo una parte della risorsa:
  - Ha side effects, può essere idempotente o meno (dipende dal tipo di aggiornamento).
  - Il body contiene o una rappresentazione parziale delle risorse o i cambiamenti applicati.
  - *Query String*: usata per aggiungere filtri sulla richiesta (come in POST).

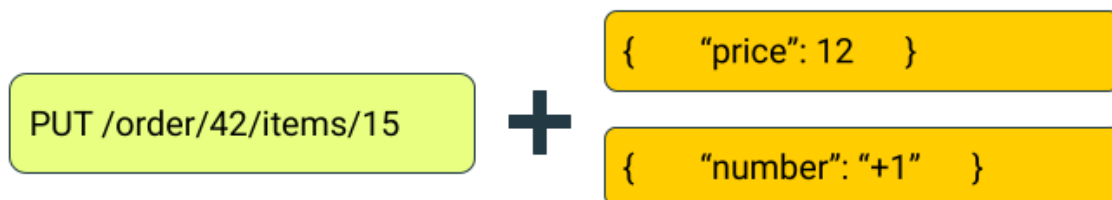


Figure 2.6: REST DELETE.

### Definizione 2.1.2: REST Status Responses

REST associa specifici significati agli status code che devono essere rispettati per la compliance.

#### Risposte:

- 2XX - Successo:
  - 200 - OK: successo generico.
  - 201 - Created: una POST che ha creato con successo una nuova risorsa.
  - 204 - No Content: operazione che ha avuto successo ma non ha restituito un body (DELETE o PUT).
- 3XX - Redirezione:
  - 303 - See Other: dopo che una POST redirige a una GET della risorsa creata.

- 4XX - Errore Client:
  - 400 - Bad Request: la richiesta è malformata.
  - 401 - Unauthorized: credenziali mancanti o invalide.
  - 403 - Forbidden: l'utente è autenticato, ma non è autorizzato ad accedere a tali risorse.
  - 404 - Not Found: la risorsa desiderata non esiste.
  - 409 - Conflict: lo stato attuale del sistema è in conflitto con l'operazione (per business rules, etc.)
  - 422 - Unprocessable Entity: la richiesta viola delle regole semantiche.
  - 429 - Too Many Requests - supererà il limite di richieste possibili<sup>1</sup>.
- 5XX - Errore Server:
  - 500 - Internal Server Error: eccezione generica.
  - 502 - Bad Gateway: il servizio fallisce per via di un fallimento upstream (proxies).
  - 503 - Service Unavailable: servizio temporaneamente down.
  - 504 - Gateway Timeout: timeout upstream (proxies).

### REST vs. RPC:

- REST: gli endpoint identificano risorse, i verbi dicono cosa fare con essere e il body rappresenta la rappresentazione delle risorse.
- RPC: gli endpoint esprimono le operazioni da fare e il body rappresenta i parametri (nella richiesta) o il valore restituito (nella risposta).

### Pragmatic REST:

- Anche usando i verbi a volte è necessario che gli endpoint esprimano operazioni:
  - Quando la semantica operativa è a grana troppo fine per essere limitata ai verbi.
  - Quando le operazioni non corrispondono a risorse esistenti nel server.
- Si può provare a rimanere REST:
  - Piazzando il verbo dopo l'identificazione della risorsa.
  - Rispettando la caratterizzazione del verbo.

#### Domanda 2.1

Dove usiamo REST in un'architettura a microservizi?

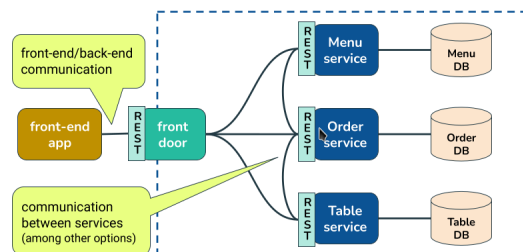


Figure 2.7: Utilizzo di REST in MSOAs.

<sup>1</sup>Indovinate chi è stata bannata dall'AUR per aver spamato richieste LOL.

### Definizione 2.1.3: External Exposed API

API che sono esposte al mondo, Solitamente con un gateway che blocca l'accesso.

### Definizione 2.1.4: Internal Service API

API interne all'applicazione con un ingresso.

### Note:-

L'ingresso pubblica quello che esiste, il gateway si occupa di cosa esporre.

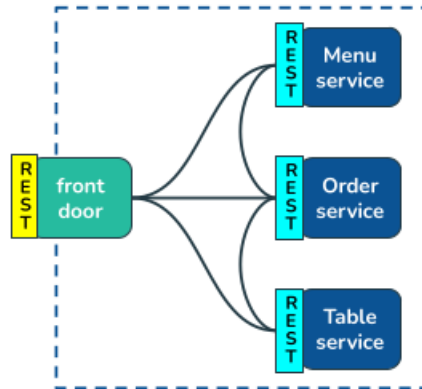


Figure 2.8: Relazione tra API interne ed esterne.

## 2.2 Docker

### 2.2.1 Introduzione

Docker nasce per la necessità di avere basso accoppiamento tra servizi nel momento del deploy e in tutta la fase successiva:

- Problemi di *middleware*: legati al "collante" tra i vari servizi.
- Le architetture modulari sono interessanti se il loose coupling avviene sia a livello logico che a livello fisico.

### L'idea delle Virtual Machine:

- Tante macchine virtuali che eseguono applicazioni.
- Si faceva per portabilità tra macchine fisiche.
- Ma ciò aveva un enorme overhead.
- Da questo nasce l'idea dei container e Docker: aree chiuse, mini sistemi operativi "finti" che danno l'illusione alle applicazioni di girare ognuna sul proprio SO. In questo modo si riduce di molto l'overhead.

### Definizione 2.2.1: Docker

Docker è un software open-source che utilizza la virtualizzazione a livello di sistema operativo per eseguire applicazioni in ambienti isolati.

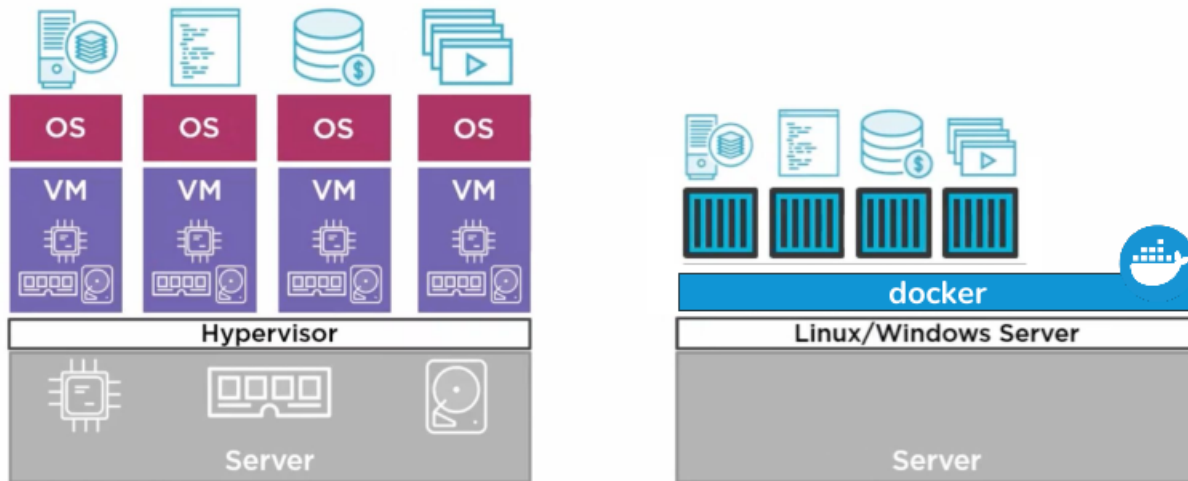


Figure 2.9: Virtual Machine vs. Docker.

### 2.2.2 Concetti Importanti

#### Definizione 2.2.2: Immagine

Un'immagine è un pacchetto standalone e leggero che include tutto ciò che è necessario per eseguire un'applicazione.

#### Osservazioni 2.2.1 Immagine

- Sono come dei mini sistemi operativi con le loro specifiche configurazioni e dipendenze.
- Le applicazioni "viaggiano" con tutto ciò che serve per poterle eseguire.
- Sono parzialmente-portatili: hanno comunque una dipendenza dall'architettura (eccezione per applicazioni java).
- Sono fatte da strati (read-only) separati e da un manifest che dice a docker come gli strati dovrebbero essere impilati.
- Solo l'ultimo strato è scrivibile.

#### Definizione 2.2.3: Container

Un'istanza in esecuzione di un'immagine Docker.

#### Features di un container:

- *Leggeri*: non sono macchine virtuali, condividono il kernel del loro OS.
- *Isolati*: ogni container esegue in un environment isolato che non interviene con gli altri container.
- *Portabili*: i container non dipendono dal loro host.
- *Effimeri*: possono partire ed essere fermati facilmente.

#### Definizione 2.2.4: Docker Inc.

Docker Inc è la società che fornisce:

- Il Docker engine distribuito sotto Apache 2.0.
- Il Docker desktop distribuito per piccoli utenti e aziende sotto licenza GNU.
- Docker hub: servizio pubblico di immagini Docker.

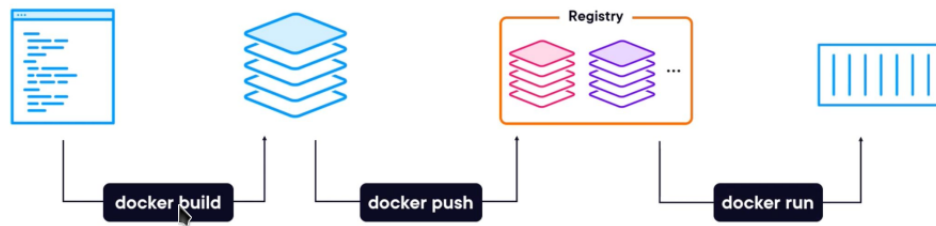


Figure 2.10: Docker workflow.

#### Docker workflow:

- build: il progetto viene costruito.
- push: su un repository condiviso (tipo docker-hub).
- run: creazione del container.

### 2.2.3 Architettura di Docker

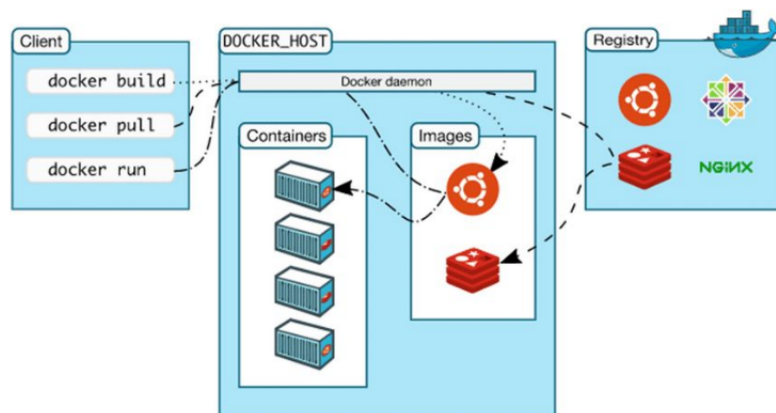


Figure 2.11: Architettura di Docker.

#### Architettura:

- Client: per poter eseguire i comandi.
- Docker host.
- Registry: locale o remoto.

**Domanda 2.2**

Come vengono fatti i containers di docker?

**Sfruttando due caratteristiche di linux:**

- Control groups: limitano l'accesso alle risorse date a dei gruppi di processi (CPU, RAM, etc.).
- Namespaces: permettono di partizionare le risorse della macchina (id dei processi, filesystem, segmenti di memoria, etc).

**Volumi persistenti:**

- Il filesystem è scrivibile a patto di avere i giusti permessi.
- Di base non è persistente.
- Per ottenere una completa persistenza si può:
  - Definire un volume in Docker e mapparlo sul OS.

**2.2.4 Networking in Docker****Un container è assimilabile a un computer attaccato alla rete:**

- Docker crea una rete virtuale dove i containers sono i nodi.
- Ogni container ha un IP.
- Al loro interno localhost è 127.0.0.1.

**Quando un container espone una porta:**

- La porta esposta è mappata all'IP internal port del container.
- Le richieste vengono ridirette al proprio container.

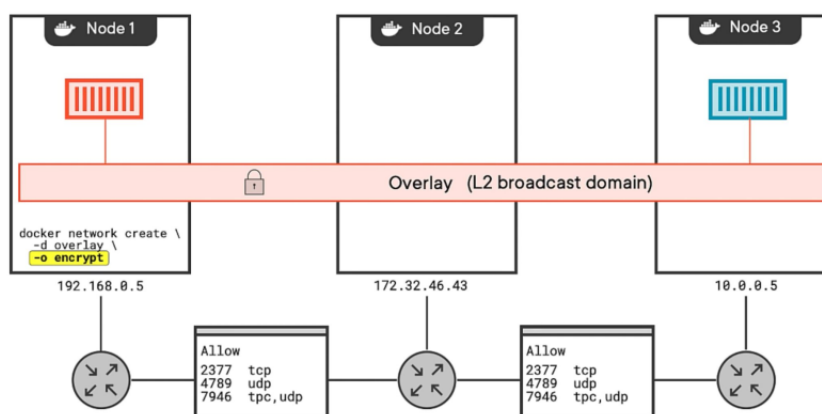


Figure 2.12: Docker overlay network.

**Esistono vari tipi di rete:**

- Bridge.
- IPvlan.
- Macvlan.
- Overlay.



### 2.2.5 Composing Containers

#### Definizione 2.2.5: Docker Compose

Sistema per gestire il ciclo di vita di un'applicazione Docker composta da più container.



Figure 2.13: Docker Compose.

#### Note:-

È un unico file yml che specifica tutti i dati necessari per avviare/fermare i servizi.

#### Docker workflow:

- build: costruisce le immagini, i volumi e le reti.
- up: avvia i servizi.
- down: ferma i servizi.

#### Compose:

- È usato per lo sviluppo e i test, non in produzione.
- Viene usato per far ripartire i servizi in caso di fallimento.



# 3

## Microservizi

### 3.1 Comunicazione Sincrona

#### 3.1.1 RPC-Style

##### Definizione 3.1.1: Comunicazione Sincrona

Un servizio client invoca un'operazione al provider di un servizio. Ci sono 2 tipi di operazioni:

- Queries: il client chiede al provider per informazioni.
- Commands: il client chiede al provider di fare qualcosa.

**Questo tipo di comunicazione è anche detto RPC-Style perché:**

- L'esecuzione delle operazioni richieste deve iniziare immediatamente.
- Il client è bloccato ad aspettare il risultato dell'operazione.

**In questo contesto REST è un protocollo sincrono:**

- REST è distante dalla nozione di procedura e si concentra sulla nozione di risorsa.
- Tuttavia, *a basso livello* la manipolazione di risorse è comunque un insieme di operazioni.
- HTTP è un protocollo sincrono.
- Ergo REST viene usato per fare comunicazione sincrona sui microservizi.

#### 3.1.2 gRPC

##### Definizione 3.1.2: gRPC

Si tratta di un'implementazione FOSS di RPC. Strutturata specificatamente pper la comunicazione inter-servizio nelle architetture a microservizi. Utilizza HTTP/2 o nuovi.

**gRPC non può essere applicata alla comunicazione back-end/front-end:**

- gRPC-web permette di effettuare alcune chiamate mediante HTTP/1.1 (che viene tradotto in HTTP/2 da un *proxy*).
- Si può usare un *sidecar*: un servizio addizionale che si attacca a un servizio per tradurre le chiamate.

**Osservazioni 3.1.1**

- HTTP/2 offre *streams*: connessioni con continuo passaggio di dati.
- gRPC permette la comunicazione *unaria* (semplice richiesta).
- Server streaming: il server deve mandare informazioni continuative al server.
- Client streaming: come il server, ma al contrario.
- Bidirezionale: tipicamente usato per le chat.

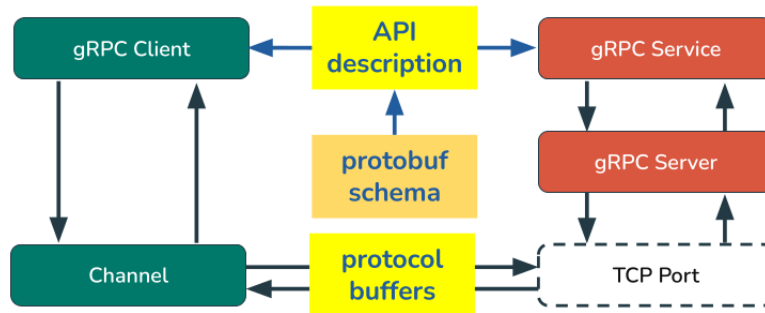


Figure 3.1: Architettura di gRPC.

- Si parte da una descrizione delle API.
- Viene generato dal compilatore di gRPC uno stub per il client e uno stub per il service:
  - Lato client: si chiamano funzioni e metodi come fossero locali.
  - Lato server: si occupa di serializzare/deserializzare i dati che arrivano dal server.
  - Client e server comunicano mediante *protocol buffer*.

**Definizione 3.1.3: Protocol Buffer (Protobuf)**

Formato serializzato per lo scambio di messaggi strutturati tra servizi in formato binario. Il programmatore deve definire un protobuf schema.

**Note:-**

L'idea è quella di ottenere serializzazione/deserializzazione in modo efficiente ed essere sia backward che forward compatible.

**Il protoc compiler genera codice per:**

- Dichiarare tipi di dato secondo uno schema.
- Creare strutture dati rappresentanti un messaggio.
- Serializzare messaggi come data streams binari.
- Deserializzare un binario secondo un determinato schema per ottenere la struttura dati originale.

**Sviluppare services e clients gRPC richiede:**

- Compilatore gRPC per il linguaggio scelto.
- Librerie gRPC per il linguaggio scelto.

## 3.1.3 Temi e Patterns

**Definizione 3.1.4: Service Discovery**

Un servizio, per poterne interrogare un altro, deve sapere come raggiungerlo (indirizzo). Questo tema fa un mapping tra il nome logico di un servizio e il suo IP (un DNS).

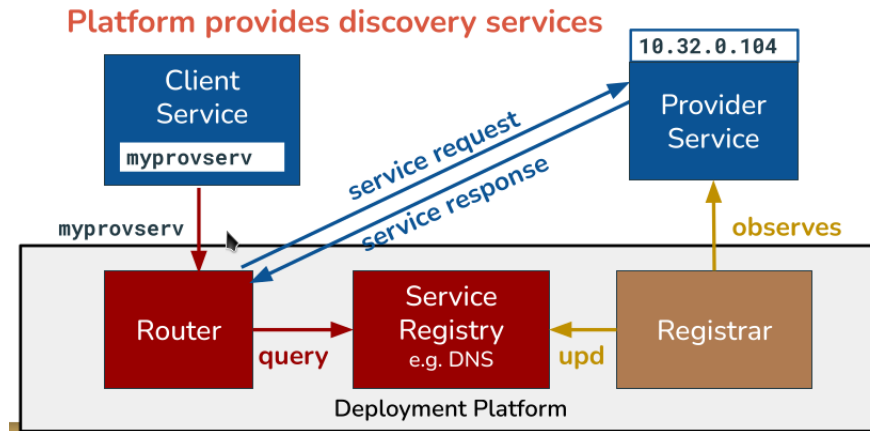


Figure 3.2: Service Discovery.

**Definizione 3.1.5: Stateless - Stateful**

- Un microservizio stateless non trattiene informazioni tra richieste:
  - Funzione pura: la computazione dipende solo dall'input.
  - Condotte: questi servizi agiscono da intermediari tra altri servizi.
- Un microservizio stateful mantiene uno stato del processo di interazione:
  - Scalabilità complessa: lo stato deve essere sincronizzato tra istanze di servizi.
  - Poco tollerante ai fallimenti: lo stato può essere perso o diventare inconsistente ai fallimenti.

**Note:-**

La maggior parte delle attività "interessanti" hanno un concetto di stato.

**Definizione 3.1.6: Network Failure**

Problemi di rete:

- Può capitare che un utente se non riceve risposta ritenta una richiesta che però è avvenuta con successo (richieste duplicate).
- Se è il sistema di pagamento a fallire si rischia di intasare il sistema (e.g. si rompe VISA e anche MASTERCARD non funziona).

**Prevenire il fallimento a cascata:**

- *Meccanismi di fallback:*
  - Vanno studiati caso per caso.
  - Restituiscono un messaggio di errore, un valore di default o un valore pre-cached.
- Intra service (*service meshes*:-)

- Strato infrastrutturale che assegna a ogni servizio un proxy.
- Stabilisce timeout di rete.
- Limita il numero di richieste aperte.
- Implementa il *circuit breaker pattern*.

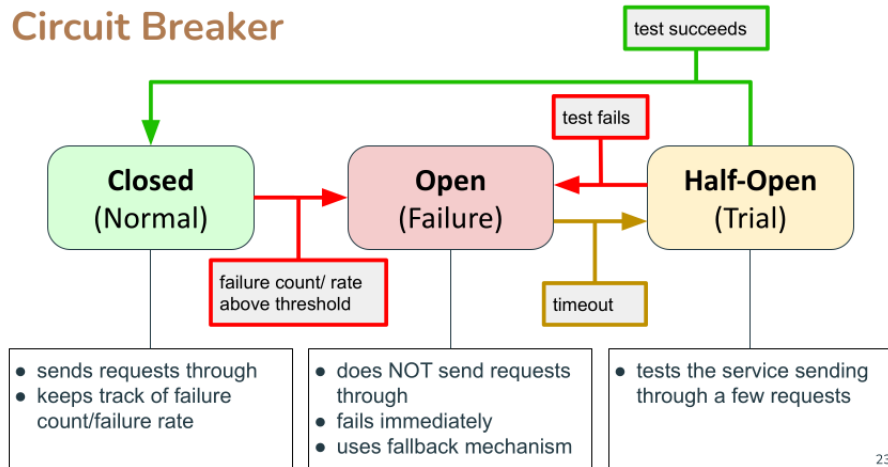


Figure 3.3: Circuit Breaker.

### 3.1.4 Richieste Duplicare e Idempotenza

#### Domanda 3.1

Un servizio o una procedura invocata è idempotente?

- Le operazioni read-only sono sempre idempotenti.
- Le operazioni di scrittura possono o meno essere idempotenti.

#### Definizione 3.1.7: Deduplicazione

La strategia utilizzata è la deduplicazione:

- Logica basata su DB constrains:
  - Il servizio ha una conoscenza logica sulle strutture dati e possono usare la chiave primaria in un DB per accorgersi di duplicati.
  - E.g. se due pagamenti hanno lo stesso ordine id.
- Payload hashing:
  - Se la duplicazione implica payload identici.
  - Non funziona con timestamp o IDs random.
  - Non funziona se due richieste hanno lo stesso payload.
- Chiavi di idempotenza:
  - Una chiave univoca è generata per ogni richiesta del client.
  - Il service provider distingue richieste duplicate da richieste diverse con lo stesso payload.

**Domanda 3.2**

Chi genera la chiave di idempotenza?

- Il client non può generarla:
  - Crea accoppiamento non necessario.
  - Difficoltà di garanzia dell'unicità.
- È preferibile farla generare al server:
  - Simile a token di sessione.
  - Il server genera la chiave, la invia al client e il client usa la stessa chiave se deve ripetere una richiesta.
  - Si usa un two-phase commit (agreement o handshake).

## 3.2 Comunicazione Asincrona

### 3.2.1 Comunicazione Orientata ai Messaggi

#### Definizione 3.2.1: Message Broker

Servizio infrastrutturale che funziona da intermediario tra il mittente e i riceventi di un messaggio.

#### Osservazioni 3.2.1

- *Fire & Forget*: il mittente dà il messaggio al broken e poi torna a farsi gli affari suoi.
- *Servizi disaccoppiati*: il messaggio non ha bisogno di essere gestito nell'immediato e il mittente non ha bisogno di conoscere l'organizzazione fisica o logica del ricevente.

#### Features del messaggio:

- Payload (il contenuto di business del messaggio) + headers (intestazione, le metainformazioni).
- Spesso negli headers è presente il correlation ID, si usa per gestire messaggi di diversi servizi.
- Solitamente i messaggi sono piccoli, se bisogna inviare grandi payloads si può usare un blob storage e inviare ai riceventi un link per scaricare il contenuto.

#### Corollario 3.2.1 Point-to-Point

Il message broker mette in atto una coda tra un mittente e un destinatario. La coda tiene i messaggi e il destinatario lo leggerà e invierà un acknowledge al mittente.

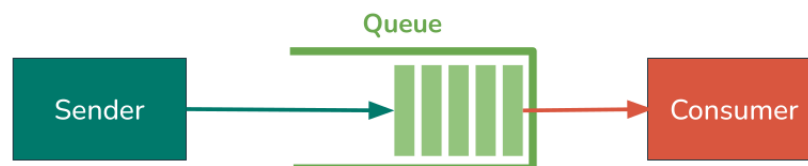


Figure 3.4: Point-to-Point.

**Corollario 3.2.2 Fan-In**

Come il Point-to-Point, ma con multipli mittenti.

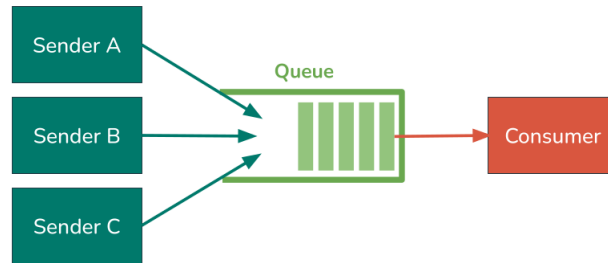


Figure 3.5: Fan-In.

**Corollario 3.2.3 Competing Consumers**

Caso simmetrico del Fan-In. I vari consumers competono per il messaggio: solo uno leggerà effettivamente il messaggio.

**Note:-**

Questo modello si usa quando si vuole suddividere il carico di lavoro.

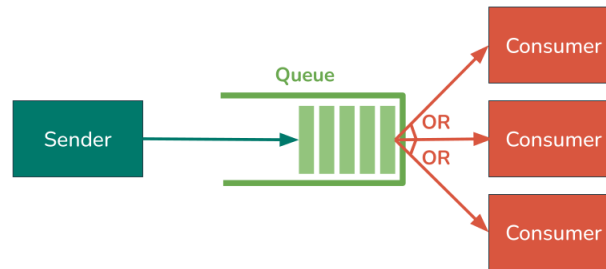


Figure 3.6: Competing Consumers.

**Corollario 3.2.4 Publish & Subscribe**

Ci sono  $n$  publisher e  $m$  subscriber. Tra di loro è presente un canale/topic/bacheca che contiene i messaggi pubblici. I subscriber ricevono in una propria coda una copia dei messaggi pubblicati.

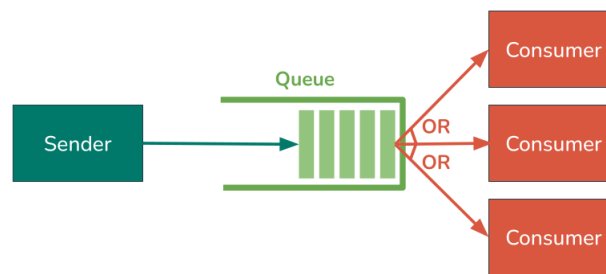


Figure 3.7: Publish &amp; Subscribe.



### Error handling:

- *Scadenza*: si aspetta un acknowledge ma si va in timeout. In questo caso si ritenta  $n$  volte.
- *Rejected*: il consumer invia indietro un messaggio negativo (NACK).
- *Overflow*: la coda è piena.

#### Definizione 3.2.2: Dead Letter Queue (DLQ)

In caso di errori il message broker muove i messaggi in una DLQ. I consumers possono controllare la DLQ e decidere di rispondere ai messaggi.

### 3.2.2 RabbitMQ

#### Definizione 3.2.3: RabbitMQ

RabbitMQ è un message broker:

- Producers si connettono a una RMQ per inviare messaggi.
- Consumers si connettono a una RMQ per ricevere messaggi.

#### Note:-

RabbitMQ è basato su AMQP 0-9-1 (Advance Message Queuing Protocol).

### RabbitMQ è un servizio middleware:

- Nelle architetture a microservizi è solitamente hostato in un container di Docker.
- Espone diverse porte, AMQP va sulla 5672.
- Può essere associata a un utility di management, accessibile via HTTP sulla porta 15672.

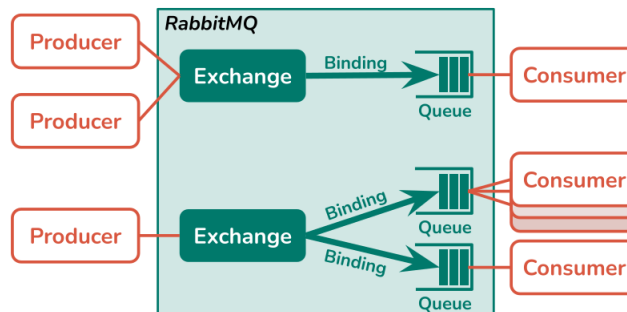


Figure 3.8: RabbitMQ.

#### Note:-

RabbitMQ è scritto in Erlang.

**Definizione 3.2.4: Erlang**

Erlang è un linguaggio concorrente funzionale:

- È multiprocesso e usa processi concorrenti leggeri.
- Ha un meccanismo di fault tolerance (let it crash) per cui quando un processo cade viene recuperato insieme al suo stato.
- Ha un clustering interno che lo rende scalabile.
- Aggiorna a nuove versioni in zero-downtime.
- È resiliente.

**Concetti chiave di RabbitMQ:**

- Exchange: i producers inviano qua i loro messaggi. Sono il punto di smistamento dei messaggi.
- Binding: collega queue a exchange.
- Queue: viste solo dai consumers. Se ci sono più consumatori il messaggio verrà dato in modo round-robin (equo).

**Default exchange:**

- È un direct exchange.
- Sempre attivo.
- Quando una coda è dichiarata c'è un binding tra il suo nome e la chiave di binding per accedere a essa.

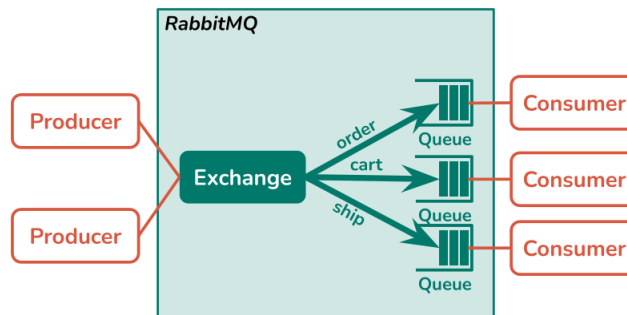


Figure 3.9: Direct Exchange.

**Competing Consumers:**

- Si vuole scalare:
  - Molteplici repliche/istanze dello stesso servizio.
  - Utile per il riutilizzo del codice.
- Si bilancia il carico:
  - Applica round-robin.
  - Salta un consumer se non risponde.
- QoL:
  - Prefetch count: threshold dei messaggi prima di assumere che un consumer non risponda.
  - Il default è 1, ma può essere configurato, specialmente per servizi che richiedono operazioni lunghe.

**Exchange types:**

- Direct exchange.
- Fanout exchange: broadcasting puro.
- Topic exchange: permette Publish & Subscribe sofisticato.

**Gestione degli errori:**

- ACK automatici (non ci sono acknowledge) o manuali (di default sono manuali).
- Con gli ACK manuali i consumers possono inviare ack o nack.
- I consumers possono configurare il prefetch time.
- I producers possono decidere se un messaggio è persistente o meno.
- Le queue possono essere durabili o meno.

**3.2.3 Temi****Definizione 3.2.5: Message Ordering & Sharing**

La code di messaggio sono FIFO, ma in caso di più consumers in competizione con differenti velocità i messaggi possono essere consumate out-of-order.

**Possibile soluzione (*sharding*):**

- Scegliere una chiave nel payload del messaggio da usare come *sharding key*.
- La queue è divisa in *shards*.
- I messaggi vengono assegnati a ciascuna shard in maniera dipendente dalla chiave.
- Ogni shard è gestita da un solo consumer.
- In caso di scaling se un'istanza di un servizio va giù la sua shard viene riassegnata.

**Note:-**

Nei casi in cui lo sharding non è supportato dal broker è possibile implementare uno sharding manuale (logico) tramite *message routing*.

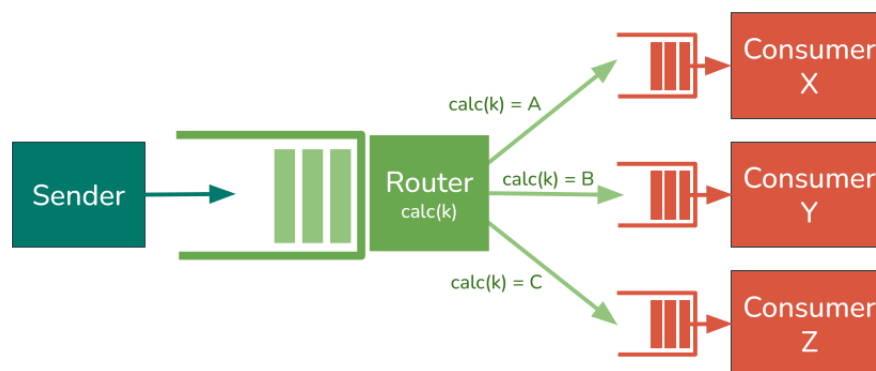


Figure 3.10: Sharding.

**Definizione 3.2.6: Message Duplication**

I message broker solitamente garantiscono at-least-once delivery:

- Garantire esattamente una consegna è troppo costoso.
- La duplicazione è spesso causata da una mancanza di acknowledgement alla ricezione.

Le possibili soluzioni sono:

- Se il message handler è idempotente la duplicazione non ha alcun effetto.
- Altrimenti i messaggi possono essere deduplicati usando chiavi di idempotenza.

**Definizione 3.2.7: Transactional Message**

Aggiornamento atomico del DB e pubblicazione del messaggio.

Le transazioni distribuite non sono una soluzione:

- Non supportate dai DB NoSQL.
- Non supportate dai moderni message broker (RabbitMQ, Kafka, etc.).
- Sono una forma di comunicazione sincrona tra processi.

Pattern per gestire l'upgrade atomico del DB:

- *Transactional outbox*: quando si aggiorna il DB si tiene una tabella con la coda dei messaggi che si vogliono inviare.
- Questo sfrutta l'atomicità del DB.
- Con un servizio a parte si vanno a leggere questi messaggi e li si invia.
- Si scorpora l'intenzione dall'azione.

### 3.2.4 Il Pattern SAGA

**Teorema 3.2.1 CAP**

In un sistema distribuito sono garantite solo 2 delle seguenti proprietà:

- Partition-tolerance: il sistema davanti a un fallimento deve comunque rimanere in piedi.
- Availability: ogni richiesta riceve una risposta.
- Consistency: ogni richiesta riceve la risposta basata sui dati più recenti.

**Note:-**

La Partition-tolerance non è negoziabile.

**Osservazioni 3.2.2**

- Assicurare la consistenza vuol dire effettuare transazioni distribuite, riducendo di molto availability.
- Le architetture microservizi prediligono l'availability sopra la consistenza.

**Definizione 3.2.8: SAGA**

Il pattern SAGA è un modo di preservare l'availability garantendo un'eventuale consistenza.

### Idea di base di SAGA:

- Si organizza il workflow in modo che sia composto da tre blocchi consecutivi di transazioni locali (LTs):
  - Un blocco di transazioni compensabili: in cui è possibile fare rollback.
  - Un blocco pivot: l'ultima transazione che può fallire.
  - Un blocco di transazioni ripetibili.
- Per ogni transazione compensabile:
  - Se c'è un fallimento si invocano tutte le transazioni compensabili in ordine inverso (rollback esplicito).

**Note:-**

Le saghe possono essere implementate sia come coreografia che come orchestrazione.

### SAGA:

- Non è ACID.
- È solo ACI.
- Non esiste una soluzione reale, ma solo contromisure.

## 3.3 Redis e Kafka

### 3.3.1 Redis

**Definizione 3.3.1: Redis**

Redis è un data store tenuto in memoria basato su coppie (chiave, valore).

#### Redis è usato per:

- Salvare dati transienti.
- Accesso rapido con caching.
- Gestire lock distribuiti.
- Messaging publisher-subscriber.

**Note:-**

Non viene usato per salvare dati persistenti di business logic.

#### Persistenza di Redis:

- RDB (Redis DataBase): un dump che avviene ogni  $N$  secondi.
  - Veloce, files piccoli.
  - Buono per: caching, quick restarts.
- AOF (Append Only File): operazione di logging.
  - Sicuro, files grandi, lento rispetto a RDB.
  - Buono per: perdita di dati minimale.

**Note:-**

Spesso sono usati entrambi.

**Rappresentazione dei dati:**

- **Chiavi:** stringhe.
  - Create quando le si assegna un valore.
  - Cancellate quando non servono più.
- **Valori:** stringhe, liste, set, hash, set ordinati, stream.
  - Per ogni tipo sono definite delle possibili operazioni (mutuamente esclusive tra tipi).
  - Quando un valore è acceduto in scrittura si crea una chiave.
  - Tutte le operazioni sono atomiche e sicure rispetto alla concorrenza.

**Quando usare Redis:**

- Quando bisogna salvare nella memoria per gestire workflow o sessioni:
  - Salvare stati di un servizio per renderlo stateful.
  - Il servizio non può essere facilmente replicato.
  - Il servizio diventa stateless: Redis gestisce gli accessi concorrenti.
- Redis può essere utilizzato se lo stato è:
  - Ricalcolabile: può essere costruito a partire dal database.
  - Usato da repliche del servizio.
  - Effimero: non ha bisogno di vera persistenza.

**3.3.2 Kafka****Definizione 3.3.2: Apache Kafka**

Apache Kafka è un message broker distribuito:

- Adotta solo un approccio Publish/Subscribe.
- Si basa sulla nozione di Event Streaming.
- Permette uno storage persistente di Event Streams.

**Note:-**

Garantisce esattamente una consegna mediante deduplicazione a tempo di pubblicazione.

**Osservazioni 3.3.1 Use Cases**

- Tracking di attività su un website.
- Monitoraggio e statistiche aggregate.
- Logs.
- Streaming di dati.
- Sincronizzare istanze di database distribuiti.

- *Event Sourcing:*

- Pattern architetturale in cui tutti i cambiamenti di stato sono salvati come eventi.

Un *evento* è caratterizzato da:

- Una chiave (qualifica l'evento).
- Valore (descrive l'evento).
- Timestamp.
- Meta-Headers.

Gli eventi sono pubblicati in *topics*:

- I topics sono logs append-only.
- I consumatori li leggono all'infinito.
- Non è possibile cancellare eventi da un topic:
  - Si può cancellare l'intero topic.
  - È possibile compattare il topic e tenere solo gli ultimi eventi per ciascuna chiave.
  - Cancellazione automatica in base a un *retention time*.

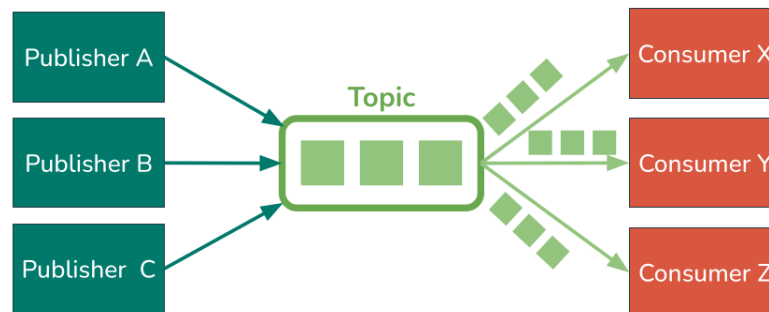


Figure 3.11: Event Streaming.

**Note:-**

Kafka può essere eseguita come cluster di brokers per migliorare la disponibilità e la scalabilità. I topic possono essere partizionati.

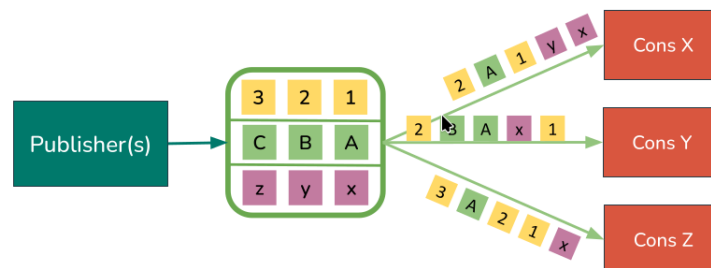


Figure 3.12: Partizionamento.

**I consumers:**

- Sono organizzati in gruppi.
- È il gruppo a iscriversi al topic.
- *Fan Out*: tutti ricevono tutto.
- Ogni partizione è assegnata a un solo consumer in ogni gruppo.

**Corollario 3.3.1 Offset**

Ogni evento in una partizione è associato a uno specifico offset:

- Ogni evento in un topic viene indicizzato.
- Univoco all'interno delle partizioni.

**Kafka:**

- Tiene traccia dell'offset di ogni consumer group in una data partizione:
  - Quando un consumer legge da un topic Kafka fornisce l'offset successivo.
  - Dopo aver processato un evento il consumer conferma la lettura.
- Usa gli offset per:
  - Sapere la posizione di ogni consumer group.
  - Dove un consumer deve riprendere dopo un crash.
  - Quali consumers sono rimasti indietro.

**Domanda 3.3**

Come comunica Kafka?

- Kafka esegue come un cluster di nodi:
  - *Broker*:
  - *Controller*
- Comunica con il proprio protocollo TCP:
  - PLAINTEXT.
  - Le porte di default sono 9092 per publishers e subscribers e 9093 per controllo interno.
-



