# Overview architetture software

Si è partiti negli anni 60’ con la nascita dei mainframe. Si tratta di macchine di grosse dimensioni dotate di, come unica interfaccia utente, **schede perforate**. Ciò bastava in quanto venivano utilizzati soltanto da scienziati.

Con la necessità di avere calcolatori all’interno delle aziende, si è sviluppato il personal computer che forniva agli utenti una interfaccia utente più accessibile. I personal computer inizialmente erano offline, per poi passare all’uso delle intranet.

Importante in un’architettura client-server è la suddivisione delle componenti di **dati**, **logica** e **presentazione**, in base a come vengono collocate queste componenti si può parlare di thin client o fat client.

## Classificazione delle soluzioni client-server

Per quanto riguarda il thin client abbiamo due soluzioni

* **Timesharing**

Dati, logica e presentazione erano tutti sul server e come client comprendeva il **char terminal**, ovvero un semplice terminale che inviava dati al mainframe e visualizzava l’output prodotto dal server. Non presenta parallelismo (poteva essere usato solo un terminale alla volta)

* **Client presentation**

Dati e logica sempre sul server ma il livello di presentazione veniva spostato sul client. Veniva messo a disposizione come cliente l’**x-terminal**, che comprendeva le prime interfacce grafiche così da comprendere il livello di presentazione

Queste 2 soluzioni ritornano in auge con l’avvento dei cloud

Per quanto riguarda il fat client abbiamo

* **Central database**

Dati presenti sul server ma logica e presentazione sul client; ciò richiedeva client con grande capacità di calcolo e server invece meno potenti.

* **Distributed database**

Il database era distribuito su server e sui client, ciò portava il problema della replicazione dei dati e dell’integrità referenziale

In mezzo si colloca

* **Distributed application**

La logica veniva divisa tra client e server, mentre il server si faceva completamente carico dei dati e il client della presentazione, ciò è stato possibile grazie all’aumento della potenza dei client. Questa tecnica era problematica in quanto alla modifica del server poteva essere richiesta anche la modifica del codice sul client, non tenendo quindi separati client e server

### Central database

Si tratta di contenitori di dati unico e in genere grossissimi. Tra le caratteristiche di cui dispone troviamo

* Stored procedures
* Two phase commit
* Replicazioni asincrone online
* Data warehouse

I vantaggi che presenta sono:

* Alta produttività dei linguaggi di quarta generazione (quindi SQL)
* Prodotti maturi ed efficienti

A suo discapito possiede dei limiti quali

* Non adatto per tutti i programmi (ad esempio, non si possono usare database NoSQL)
* Limitata scalabilità

Utilizza il protocollo TCP/IP per far comunicare client e server.

I problemi grossi nell’avere il fat client sono

* Forte sollecitazione della rete 🡪 una soluzione sono le stored procedures
* Forte uso delle risorse client
* Difficile scalabilità
* Manutenzione costosa
* Accesso ai dati poco protetto

### Distributed database

Avendo i database distribuiti tra client e server si ha il problema delle **replicazioni asincrone**. Database locali e pubblici possono avere dati replicati, il problema è quello della sincronizzazione dei dati. I database distribuiti devono garantire

* Configurabilità dei dati replicati
* Sincronizzazione automatica e “al più presto”
* Protezione delle transazioni
* Integrità referenziale dei dati replicati

Una volta veniva scelto dalle banche un orario nella notte per sincronizzare i dati dei database. L’integrità referenziale è fondamentale, ad esempio, nel caso d’uso delle transazioni.

## Middleware

Un middleware è un software che permette la comunicazione in un sistema distribuito. Si colloca al di sopra dei sistemi operativi. Si tratta di una **glue technology** perché fa da collante tra le applicazioni. Offre astrazione a livello di programmazione che nasconde parte della complessità per costruire un’applicazione distribuita. Le funzionalità non sono state inserite direttamente nel sistema operativo perché dipendono molto dal dominio di applicazione; quindi, sarebbe tante e andrebbero ad appesantire l’OS.

Al programmatore vengono date delle funzioni per fare comunicare la sua applicazione con un’altra (es. per far comunicare un’applicazione php con il database). È una soluzione per fare comunicare i sistemi legacy con i sistemi attuali andando a costruire dei wrapper (ad esempio, per cercare di comunicare con il vecchio mainframe di una banca)

Un esempio di middleware è

* JDBC, middleware che fa comunicare java con un database
* RPC, middleware che permette di chiamare una funzione remota come se fosse in locale
* Web services, funzionalità richiamabile all’interno di applicazioni

Tipologie di middleware

* TP Monitor
* MOM
* Publish/Subscribe
* ORB

MOM e Publish/Subscribe sono implementate nelle applicazioni enterprise.

### TP monitor

È il middleware che si occupa di gestire la **transazionalità** delle operazioni tra due applicazioni. Tra le sue caratteristiche ha

* Mantiene le **proprietà** **ACID**, cioè la transazione deve essere
  + Atomica, la sua esecuzione deve essere totale o nulla
  + Consistente, i dati nel db devono essere consistenti
  + Isolata, ogni transazione è eseguita in maniera indipendente dalle altre
  + Durabile (o persistente), i cambiamenti apportati devono persistere
* Sincronizzazione in **Two** **phase** **commit**
* Esegue il **load** **balancing**, creando nuove istanze dei servizi
* Fornisce la **high** **availability**
* Esegue il **funneling**, contingenta le richieste al server fornendo le risorse ai client che ne hanno più necessita facendo attendere in coda quelle che non hanno un bisogno immediato
* **Libera le risorse** appena il client smette di usarle assegnandole ad un altro

Come vantaggi fornisce

* Scalabilità per sistemi di grandi dimensioni
* Adatti per applicazioni **mission critical**

Il problema è che

* Ci sono pochi standard
* Ha un uso limitato dei linguaggi di quarta generazione

### ORB

Si tratta di un middleware che permette la comunicazione tra applicazioni distribuite scritte in linguaggi OO diversi (RMI, remote method invocation, fa la stessa cosa per applicazioni che si trovano su JVM diverse, ma solo per applicazioni scritte in Java). CORBA è lo standard open source che implementa il middleware ORB.

CORBA fornisce il linguaggio IDL (interface definition language) per la traduzione dei programmi in un linguaggio comune. IDL andrà a fornire un’interfaccia su cui il client potrà andare ad invocare i metodi del server.

Il funzionamento è come segue

* L’IDL stubs permette di simulare la presenza di metodi sul client che è possibile invocare, che in realtà si trovano in remoto sul server
* L’IDL compiler prende i metodi pubblici del server e li converte nel linguaggio IDL andando a costruire l’IDL skeleton che si occupa di spacchettare la richiesta e associarla al metodo corretto del server
* La comunicazione avviene attraverso l’ORB interface

L’**ORB core** è il middleware che permette di effettuare lo scambio dei messaggi. Lo scambio di messaggi avviene convertendoli in sequenze di byte per poi trasmetterle sulla rete e successivamente riconvertirli (processo chiamato **marshall**/**unmarshall** o **serializzazione**/**deserializzazione**). La cosa interessante è che il marshall converte il messaggio da un linguaggio A ad una sequenza di byte e successivamente l’unmarshall lo convertirà da una sequenza di byte ad un linguaggio B.

CORBA utilizza il **naming service**, cioè un servizio che fa il binding tra il nome invocato dal client di un metodo e il nome del metodo vero e proprio presente sul server, così da rendere indipendenti client e server (concetto che rivedremo negli api gateway).

La comunicazione viene fatta attraverso il **pattern broker**.

CORBA veniva usato all’interno delle reti locali, ai tempi non esisteva ancora internet. I servizi SOAP e REST usano un approccio simile.

### MOM

È un middleware orientato ai messaggi e in particolare alla comunicazione asincrona. Invece di usare RMI usa i messaggi asincroni. JMS e RabbitMQ sono due esempi di MOM.

#### Pattern fire & forget

Si tratta di una comunicazione one-to-one che procede come segue

1. Un programma A invia un messaggio su una coda e prosegue la sua esecuzione. Se il messaggio prevede una riposta un metodo di A catturerà la risposta successivamente.
2. Il messaggio inviato da A sarà inizialmente su una coda privata e verrà inviato alla coda pubblica appena sarà disponibile.
3. Una volta sulla coda pubblica il broker si occuperà di inviare il messaggio al programma B. Se il messaggio prevede una risposta si procederà con il processo inverso.

Questo procedimento permette

* **Disaccoppiamento tra mittente e destinatario**
* **Consegna garantita**

#### Pattern publish/subscribe

Si tratta di una comunicazione one-to-many che procede come segue

* Un publisher invia i messaggi al broker specificando il topic senza sapere chi sono i destinatari
* Il broker si occuperà di inviare a tutti i subscriber i messaggi del publisher, dove per subscriber si intendono tutti coloro che sono iscritti alla ricezione dei messaggi di quel particolare topic

# Java enterprise

Java enterprise è un framework per lo sviluppo di applicazioni server-side complesse. Si tratta di una **piattaforma java che serve a costruire le applicazioni di grandi dimensioni per le aziende**. Permette di creare architetture multilivello, di fare load balancing e di gestire la scalabilità dell’applicazione.

Serve per applicazioni di grandi dimensioni specifiche per il web. Realizza una **architettura** **a** **tre livelli**:

* **Web server** (Servlet, JSP, …)
* **EJB server** (Enterprise java bean)
* **Data**

Da usare quando si vuole

* Scalabilità
* Integrità dei dati dopo le transazioni
* Più client (mobile, web, …)

Solitamente applicazioni enterprise sono costruite per attività di impresa. Molto utile per far convivere vecchi sistemi legacy dell’azienda con i nuovi sistemi.

Due delle componenti più importanti di JEE sono

* EJB (enterprise java bean)
* JMS (java message service)

## Enterprise java bean

Ci sono diversi tipi di EJB

* Session bean
* Entity bean

### Session bean

Fanno da intermediari tra le richieste provenienti dal server web (effettuate tramite RMI) e le richieste al livello dei dati (effettuate, ad esempio, con JDBC o usando JPA). Si occupano di implementare la **logica di business** della nostra applicazione.

Un session bean può essere

* **Stateful**
  + manterrà lo stato per tutta la durata della sessione
  + ogni client crea una sua nuova istanza sarà condivisa con nessun altro
  + terminata la sessione l’istanza del session bean sarà cancellata
* **Stateless**
  + lo stato non viene gestito, per salvare i dati si dovrà fare riferimento al livello dei dati
  + il container EJB mantiene il pool dei session bean che potranno essere usati dai client
  + può essere convertito in un servizio in quanto stateless;
* **Singleton**
  + è stateful e mantiene lo stato per l’intero ciclo di vita dell’applicazione
  + può essere condiviso tra i client
  + l’accesso al bean è concorrenziale e può essere gestita
    - dal container
    - dal bean stesso

### Entity bean

Viene usato quando vogliamo avere degli oggetti persistenti, ad esempio

* CreditCardBean è un entity bean
* CreditCardVerifiedBean (chi verifica la carta di credito) è un session bean

Lo stato dell’entity bean è persistente. Se l’istanza del bean termina o se l’application server termina, lo stato del bean esiste in un datatabase.

Una classe rappresenta una tabella, mentre un’oggetto della classe rappresenta una riga della tabella. Gli entity bean sono accessibili dai session bean mediante i **facade** (implementa appunto il **pattern facade**).

### Message driven bean

Il **message driven bean** serve ad aiutare ad implementare il codice che riceve i messaggi asincroni (ha vita breve, solo per l’elaborazione del singolo messaggio) Quando ci si vuole mandare dei messaggi asincroni il client invia i messaggi a una coda e il message driven bean riceverà in modo asincrono i messaggi, andando a prenderli dalla coda e processandoli. L’invio di un messaggio invece è fatto da un semplice session bean.

## EJB server e container

EJB server e EJB container sono contenuti nel JEE server

### EJB server

Si occupa di fornire la JVM e le classi di supporto agli EJB. Garantisce le funzioni di base di ORB e TP Monitor, permette l’accesso ai db e si occupa di gestire il load balancing e l’high availability

### EJB container

Fornisce l’ambiente in cui vengono eseguiti gli EJB. Mantiene il pool di istanze dei bean pronti per gestire le richieste, gestendo il passaggio da attivi a inattivi e gestendo il threading.

Insieme a EJB server realizza i servizi sopra descritti e, oltre a ciò, si occupa di sincronizzare le variabili di istanza degli entity bean con il server.

### EJB object remote

È possibile definire un EJB con interfaccia remota o locale. Se è remota vuol dire che senza che il programmatore se ne accorga, questa classe può girare sulla JVM di un’altra macchina e senza preoccuparcene possiamo mandare messaggi alla classe (come se ci fosse un RMI nascosto sotto che ci permette di comunicare)

### Dependency injection

Dentro tutti gli EJB c’è la dependency injection (fatta attraverso le annotation), così da poter definire

* Data source
* Interfaccia dell’entity manager (cioè il gestore delle entity)
* Rendere l’EJB un web service
* …

## JMS

Viene utilizzato per implementare la comunicazione asincrona tra diverse applicazioni. Ciò ci permette di rendere l’applicazione

* Distribuita
* Loosley coupled

Mittente e ricevente dovranno solo conoscere il formato del messaggio per poter comunicare, JMS si occupa di tutto il resto.

Il messaggio viene inviato da un session bean e viene ricevuto da un message driven bean e sarà catturato tramite il metodo **onMessage**. **JMS provider** si occupa di fare il gestore delle code (il broker).

Il messaggio inviato sarà di tipo **Message**. Per stabilire le connessioni è possibile usare le **connection factories**.

Sarà possibile fare una comunicazione

* One-to-one (fire & forget)
* One-to-many (publish/subscribe)

## Persistenza

Una **entità** (entity bean) è un oggetto persistente nel dominio. Rappresenta una tabella nel db relazionale

* Ogni istanza corrisponde a un record della tabella
* Lo stato persistente dell’entità è rappresentato o dai campi o dalle proprietà
* Campi/proprietà usano le **annotazioni** per fare il mapping tra entità e relazioni ai dati nel db sottostante

Le tabelle sono le classi persistenti del nostro sistema. L’ORM (object relational mapping) verrà quindi fatto tramite le annotazioni.

### Entity manager

È la parte del container che gestisce la persistenza, l’istanza del contesto opererà in base al context specificato. Ci sono due tipi di persistenza

* Gestita dal container (più facile da usare ma con meno funzionalità)
* Gestista dal programmatore (più complicata da usare ma con più funzionalità)

### Persistence unit

Definisce l’insieme di tutte le classi entity gestite da un’istanza di Entity manager. Quindi, dentro un’applicazione è possibile avere più persistent unit.

Quindi, la persistent unit è l’insieme delle classi entity che saranno mappate dentro una particolare istanza di database.

Quindi il link tra db e entity bean è fatto attraverso la persistent unit (specificata in un file xml)

# SOA e Web services

## SOA

Il web nel corso degli anni si è spostato da una visione **human centric** ad una visione **application centric** (esempio dei servizi di prenotazioni di voli online). Le Service Oriented Architecture sono nate dal bisogno di dover integrare software proveniente da diverse parti. Ciò permette di legare vecchi sistemi legacy con sistemi più moderni.

Per implementare i SOA si è ricorso all’uso dei web services, cioè componenti software accessibile attraverso la rete mediante protocolli. Si è passato da uno sviluppo dei Web Service di granularità elevata (chiamati silos) a uno sviluppo di Web service di granularità più piccola.

Le caratteristiche dei SOA sono

* Modularità
* Facilità di sostituzione del codice
* Integrazione di servizi esterni nel proprio sistema

## Web services

I web services, hanno la stessa idea degli ORB, ovvero far comunicare applicazioni scritte in linguaggi differenti. I web services lo fanno mediante l’utilizzo di messaggi XML.

Caratteristiche dei web services:

* Oggetti software incapsulati
* Loosley coupled
* Formati da contratti (da API durature)
* Disponibili sulla rete
* Utilizzano messaggi standard XML
* Indipendenti dal linguaggio
* Descritti (tramite WSDL)

Standard dei web services:

* SOAP, protocollo per il richiamo dei web services
* WSDL, linguaggio per la descrizione dei web services
* UDDI, protocollo per la ricerca dei web services (sorta di elenco telefonico)

Esempio di utilizzo:

* Client cerca il servizio nel registro UDDI
* Il registro ha il puntatore al servizio (URI) e alla sua descrizione nel file WSDL
* Viene richiamato il servizio dal client attraverso il protocollo SOAP

Tipi di invocazione dei web services

* RPC-style, invocazione sincrona
* Document-style, invocazione asincrona

I tipi di richiesta possono essere

* One-way
* Request-response
* Solicit response
* Notification

I messaggi SOAP sono formati da

* Header, contenente dati opzionali sulla chiamata
* Body, contenente dati delle chiamate o valori di ritorno

### WSDL

Linguaggio che permette a un web service di descrivere sé stesso mediante XML. La descrizione può contenere

* Funzioni disponibili del servizio
* Data types delle richieste e delle risposte
* Informazioni sul protocollo di trasporto
* Informazioni sull’indirizzo dov’è localizzato il servizio

Gli svantaggi della definizione WSDL sono che non specifica informazioni sul servizio (es. se è a pagamento, qualità del servizio, …).

### UDDI

Permette attraverso l’uso di un registry di trovare i web services, identificati in modo univoco. I dati vengono salvati anche in questo caso in un file XML. UDDI può essere usato sia per cercare che per pubblicare i web services.

UDDI prevede quattro tipologie base di informazioni

* Business information, informazioni sul proprietario
* Business service information, descrizione tecnica del web service
* Binding information, informazioni sull’invocazione del servizio
* Service specification information, altre info sul web service

I registry UDDI sono accessibili mediante API e si possono trovare all’interno di Cloud. Il registry ci dirà come fare l’invocazione al servizio e che parametri passare.

Come detto, è possibile esporre gli EJB al pubblico mediane un’interfaccia WSDL.

## Servizi rest

Le invocazioni ai servizi verranno fatte mediante le primitive di http e non più usando soap. Solitamente il tipo del valore di ritorno di un rest è in formato JSON (ma potrebbe essere anche XML o altro).

A svantaggio dei servizi rest c’è la non adattabilità con i servizi legacy (cosa che permettono invece i web service SOAP).

Immagine che contiene testo

Descrizione generata automaticamente

# Architettura a microservizi

## Caratteristiche

Si è deciso di lasciar perdere le vecchie architetture monolitiche perché

* Ostacolavano i frequenti aggiornamenti
* Per cambiare un componente bisognava rilanciare l’intera applicazione e quindi interrompere l’intera sua esecuzione
* Grossi rischi di failure
* Vincolate ad un solo linguaggio

L’architettura a microservizi

* Utilizza i servizi rest
* Fornisce dei pattern per l’uso dei servizi Rest
* Permette la continuos integration
* L’architettura può essere integrata al cloud computing

Questa architettura fornisce il **load balancing**.

Ogni servizio può avere il proprio database e può comunicare con altri database.

Vantaggi dei microservizi

* Servizi piccoli
* Indipendenti dal linguaggio
* Minimizza i rischi in quanto unità a sé stante dell’architettura
* Scalabile
* Agile (facile da adattare e da riutilizzare)
* Benefici organizzativi per adattare servizi diversi

Il punto importante è che possiamo lavorare sul singolo servizio senza dover intaccare il sistema nel suo complesso.

Difficoltà

* Interazione complicata
* Rischio di dati replicati da sincronizzare (ci sono dei pattern)
* Sistema di monitoring, che ci garantisce che controlla gli stati dei servizi

I rest sono stateless e gestiscono la persistenza salvando le informazioni sul db.

Ci permette una separazione netta tra backend e frontend, così da poter avere interfacce diversi che utilizzano i servizi.

Per effettuare la comunicazione asincrona, quindi come MOM, usa RabbitMQ (utilizzabile, ad esempio, per sincronizzare i dati nel db).

Nell’architettura standard dei microservizi possiamo trovare un **api gateway**, ovvero, un servizio rest, che fa da tramite tra client e microservizi. Le richieste ai microservizi da parte del client vengono inviate all’api gateway (che avrà un certo IP e una certa porta) ed esso si occuperà di instradare la richiesta al servizio vero e proprio (che avrà IP e porta diversa). L’api gateway per trovare la location del servizio può salvarsi in un file la location oppure può avvalersi del **registration services** (ad esempio, Eureka), ovvero un registro che mantiene aggiornati i riferimenti ai microservizi.

### Partizionamento

Una scelta difficile è quella dei confini del nostro servizio, fin dove spingersi? Bisogna fare delle valutazioni. Rischio **cross database join problem**, cioè due servizi comunicano troppo tra di loro, converrebbe unirli in un servizio solo.

È possibile usare l’api gateway come servizio rest che fa da tramite tra client e l’architettura.

L’architettura può fare uso solo di messaggi sincroni (usa solo RPC), asincroni oppure può essere **ibrida**. Esiste anche l’**architettura monolitica ibrida**, in cui molti servizi fanno riferimento ad un unico db.

Una architettura a microservizi fornisce buone prestazioni se

* Minimizza l’**impact of latency**, se c’è troppa comunicazione tra i servizi si ha troppo dispendio di rete
* Considera più **coarse-grained designs**, quindi considerare servizi che comprendono un po' più di operazioni in modo da non dover comunicare troppo con l’esterno
* Rimuove **comportamenti chatty**, cioè evitare troppi scambi di messaggi tra microservizi (fare refactoring quindi)

Vediamo quindi che il focus centrale è sempre sui bordi del servizio; scegliere buoni bordi ci permette di avere un servizio performante. Per capire come partizionare i servizi, una delle linee guida da seguire è quella del **single responsability principle**, quindi dare una o poche più responsabilità al microservizio (a livello di CRC cards).

Un buon approccio è anche quello **business domain centric**, cioè si individuano i vari business ed ogni business sarà un microservizio. L’idea è usare una tecnica per individuare e definire i boundaries dei servizi per mantenere l’architettura il più pulita possibile (andando a fare refactoring)

Attenzione: fare attenzione alle **Tangled dependencies**, cioè le dipendenze attorcigliate, cioè che ci sono microservizi che eseguono tante comunicazioni tra di loro, quindi, **i confini non sono stati disegnati bene**.

### Differenze tra SOA e architettura a microservizi

* In SOA alcuni servizi possono essere stateful, mentre nei microservizi sono completamente stateless
* SOA usa il protocollo SOAP per comunicare
* SOA mette più enfasi sulla riusabilità, mentre Microservizi sul decoupling
* Cambiamento in SOA rischia di porre un cambiamento in tutta la struttura monolitica
* SOA ha l’enfasi sui db relazionali, i microservizi su qualsiasi tipo

### Hosting dei microservizi

I microservizi possono essere ospitati su

* Macchine virtuali
* Piattaforme cloud
* Container

## Pattern

### Principi di design

I nostri servizi devono avere le seguenti caratteristiche

* High cohesion (fare operazioni riguardanti lo stesso dominio, ma non fare solo operazioni crud)
* Autonomi
* Business domain centric
* Resilienti (sopperire ai guasti, ai molti accessi, …)
* Osservabili (osservare il loro funzionamento)
* Automatici (testing automatico)
* Loosely coupled

### Design dell’architettura a microservizi

Dobbiamo distinguere

* **Greenfield software development**: equivale al partire da zero quando si sviluppa un software, bisogna fare analisi, scelte delle architetture, no restrizioni, … Molto raro da trovare
* **Brownfield software development**: bisogna scrivere nuovo software in presenza di vecchio software, quindi bisogna legarlo; la nuova architettura deve coesistere con ciò che già esiste; l’obiettivo è quello di riuscire ad estrapolare da un brownfield software, magari inizialmente monolitico, dei microservizi

#### Brownfield microservice

Si parte col partizionare man mano il db, andando a creare un db per ogni db estratto, fino ad ottenere l’architettura a microservizi.

Si cerca di estrapolare prima le parti che mettono meno a rischio il funzionamento del servizio; quindi, si cercano le parti meno legate con le altre e le si estrapolano.

Si attua un approccio incrementale (estrapolo e vedo che succede).

Chiaramente si otterranno db con dati duplicati, bisognerà utilizzare dei pattern per gestire la problematica.

#### Greenfield Microservices

Dovremo definire sistema, dominio, cercare di capire quali sono le entità di business che diventano poi dei servizi. Si farà una revisione di tutto il codice per capire se diviene una libreria o un microservizio (si guarda se è a sé stante oppure se serve che sia incluso in tutti i microservizi)

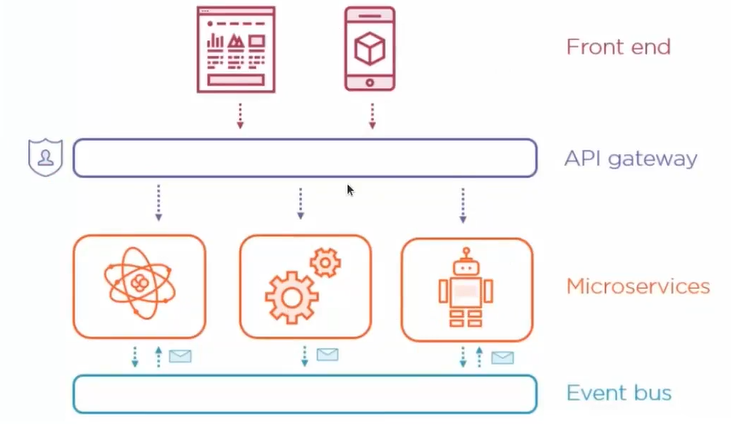
Si farà una **domain driven design**, cioè un design guidato dal dominio. Si individuano le classi, facendo presente che stiamo lavorando su un sistema più grosso. Diventa importante il **contesto** e i **confini** dei nostri microservizi.

Bisogna individuare i concetti e capire dove si collocano nei microservizi. Ciò viene fatto attraverso la valutazione del contesto.

### Pattern di comunicazione

Come sappiamo i microservizi possono comunicare tra di loro in maniera sincrona o asincrona

#### Microservice communication pattern



#### Competing worker

Io mando un messaggio e ogni consumatore del messaggio, lo processa; quindi, ci sono più istanze di servizio che fanno la stessa cosa (questo serve quando ci sono tantissime richieste come nel caso di Netflix), quindi c’è ridondanza.

#### Fanout pattern

Viene inviato il messaggio al broker che lo invia in broadcast a tutti i servizi registrati (l’invio avviene in parallelo a tutti i destinatari)

### Design pattern dei microservizi

#### Facade design pattern

Pattern visto per quanto riguarda l’accesso agli entity bean in JEE. Le API avranno una interfaccia facade che sta tra la business e gli API veri e propri

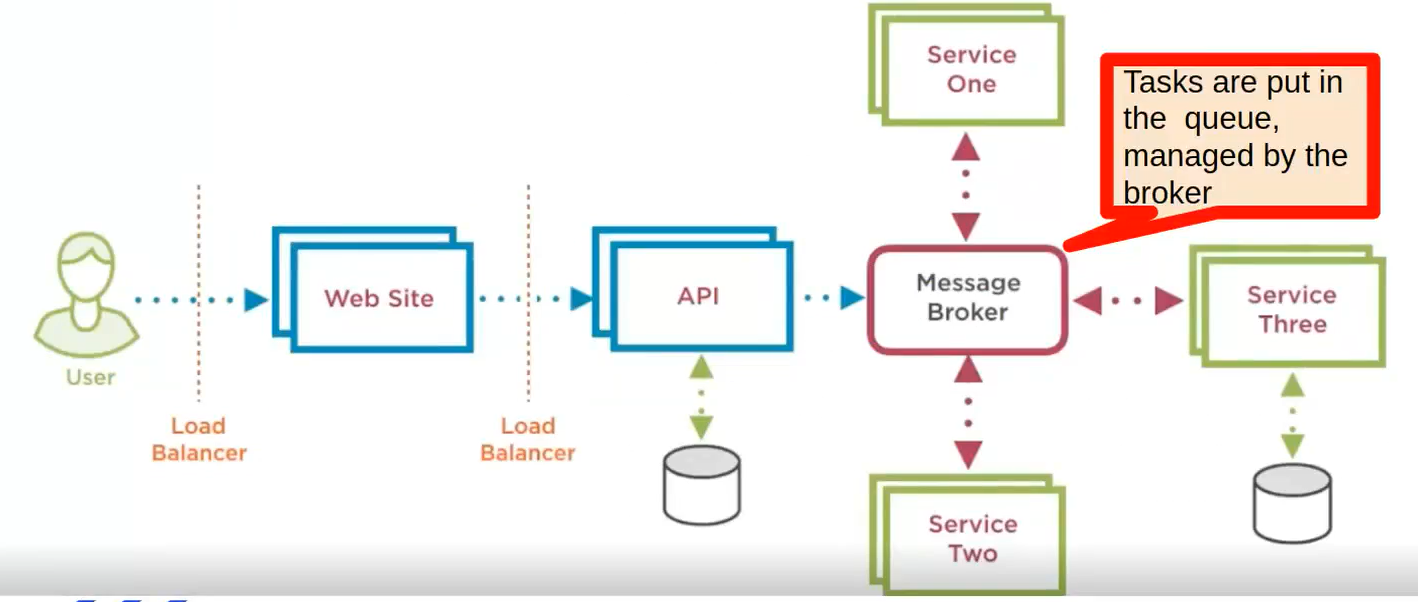
#### Proxy design pattern

Abbiamo il proxy, che è un altro strato di software che fa da intercapedine tra la business logic e le API (Raramente usato)

#### Stateless service pattern

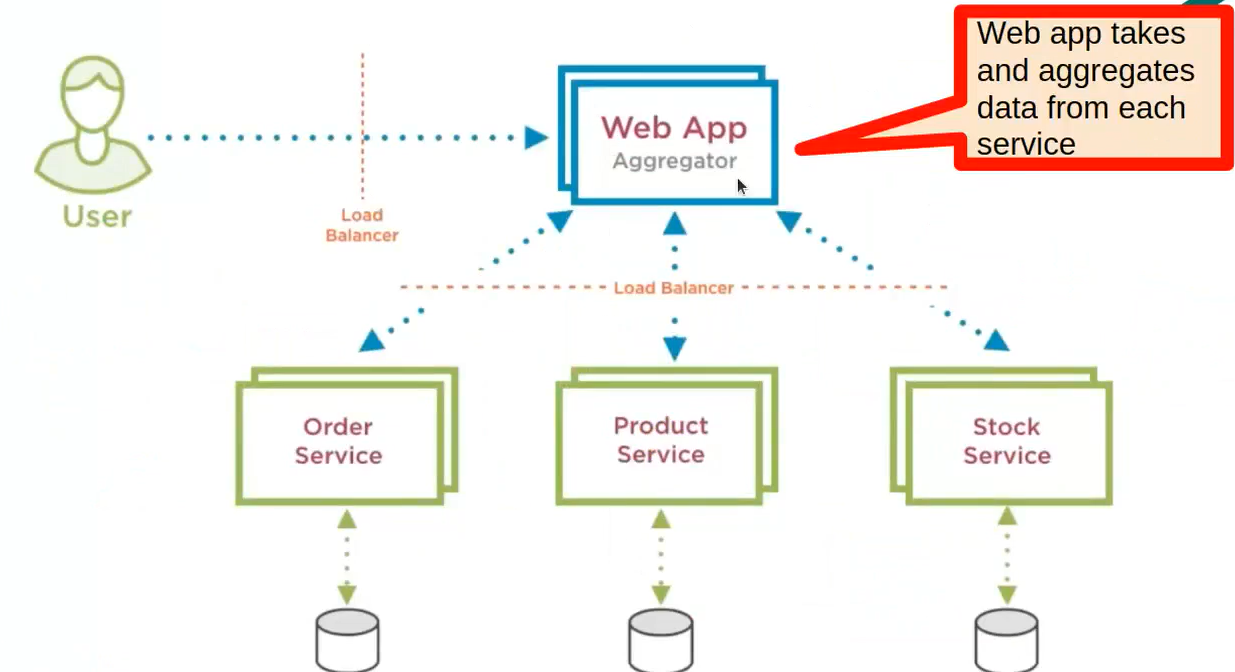
Sappiamo che i rest non hanno stato, per accedere ai dati di un utente, ad esempio, bisogna passare nella richiesta al microservizio l’id, ad esempio. Un’altra soluzione potrebbe essere passare lo stato nella richiesta, ma vorrebbe dire passare una grande quantità di dati.

#### Broker composition pattern

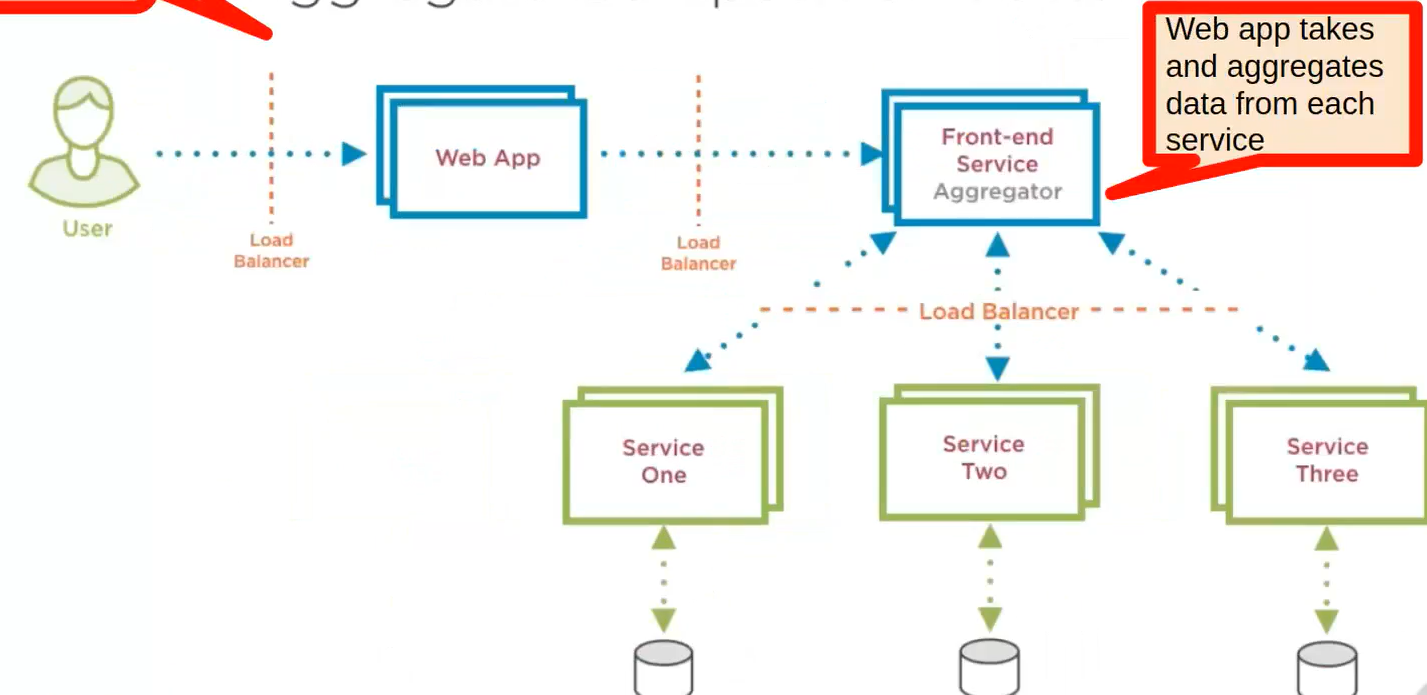


I servizi vengono invocati tramite dei messaggi fatti dall’api gateway all’interno del message broker

#### Aggregate composition pattern

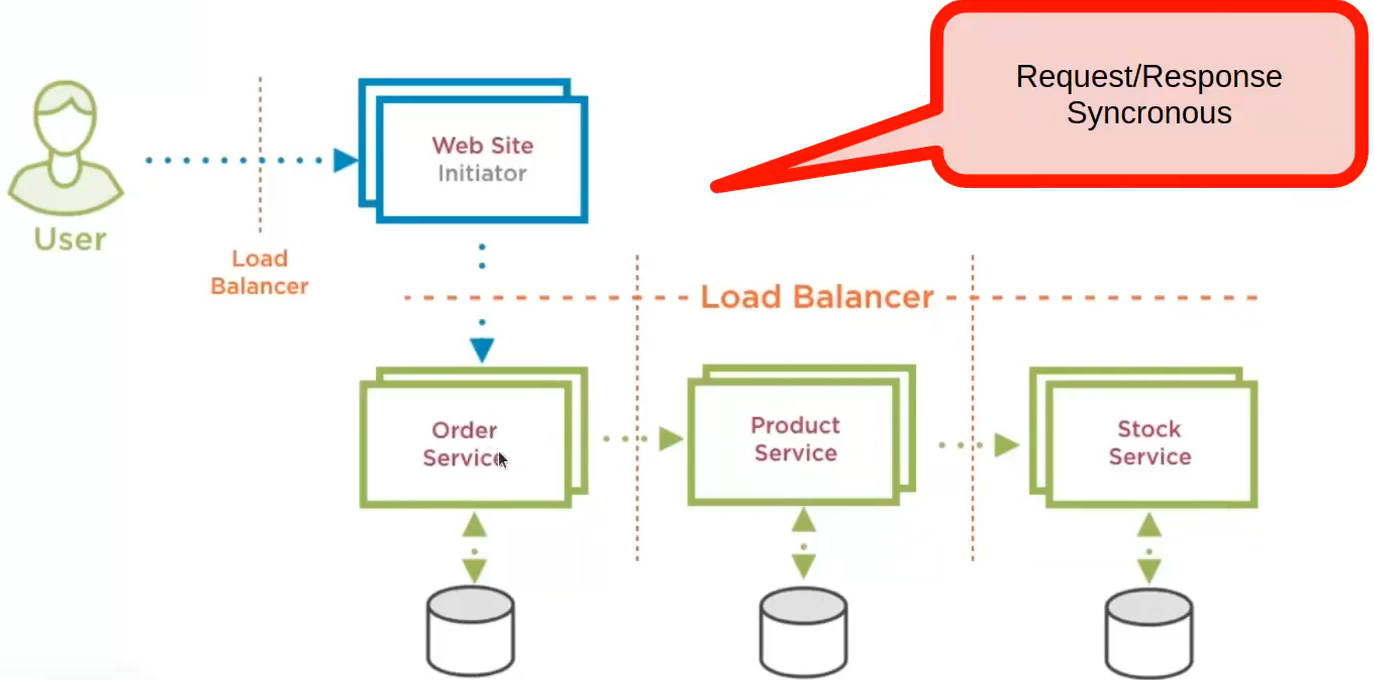


La web app in questo caso fa da aggregatore, cioè gestisce la composizione dei servizi rest, aggrega i dati di ogni servizio. Una soluzione migliore è la seguente



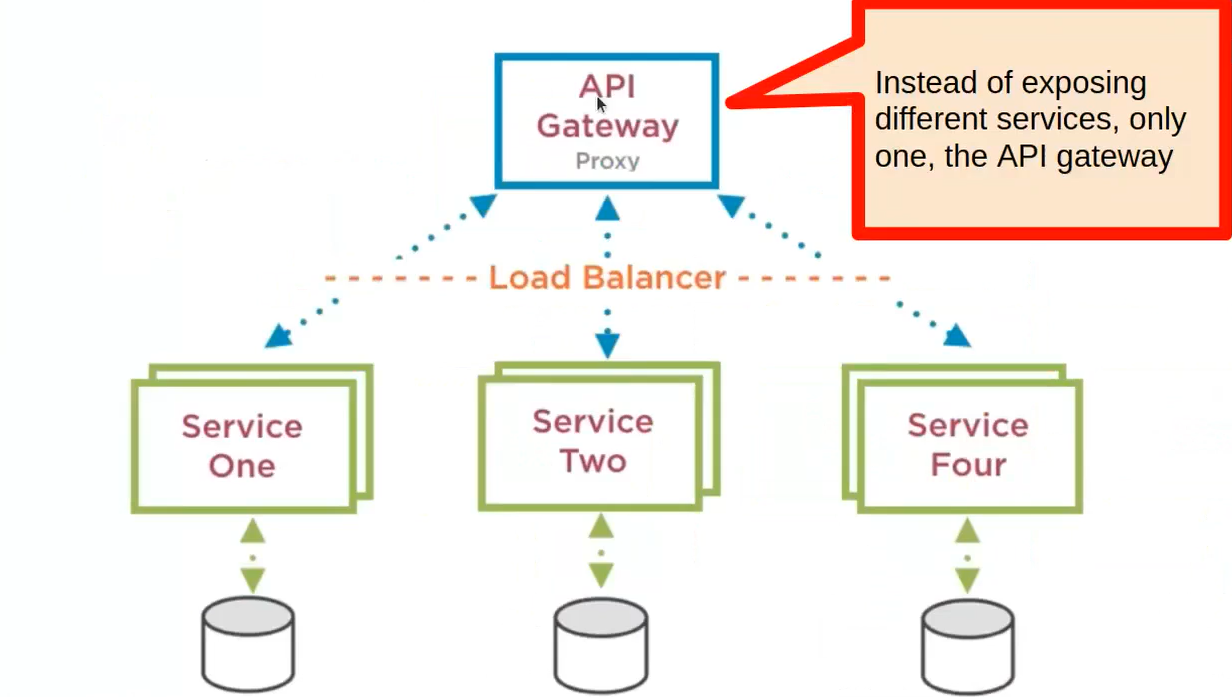
Il front end service è un servizio che fa l’aggregazione dei dati dei 3 servizi

#### Chained composition pattern

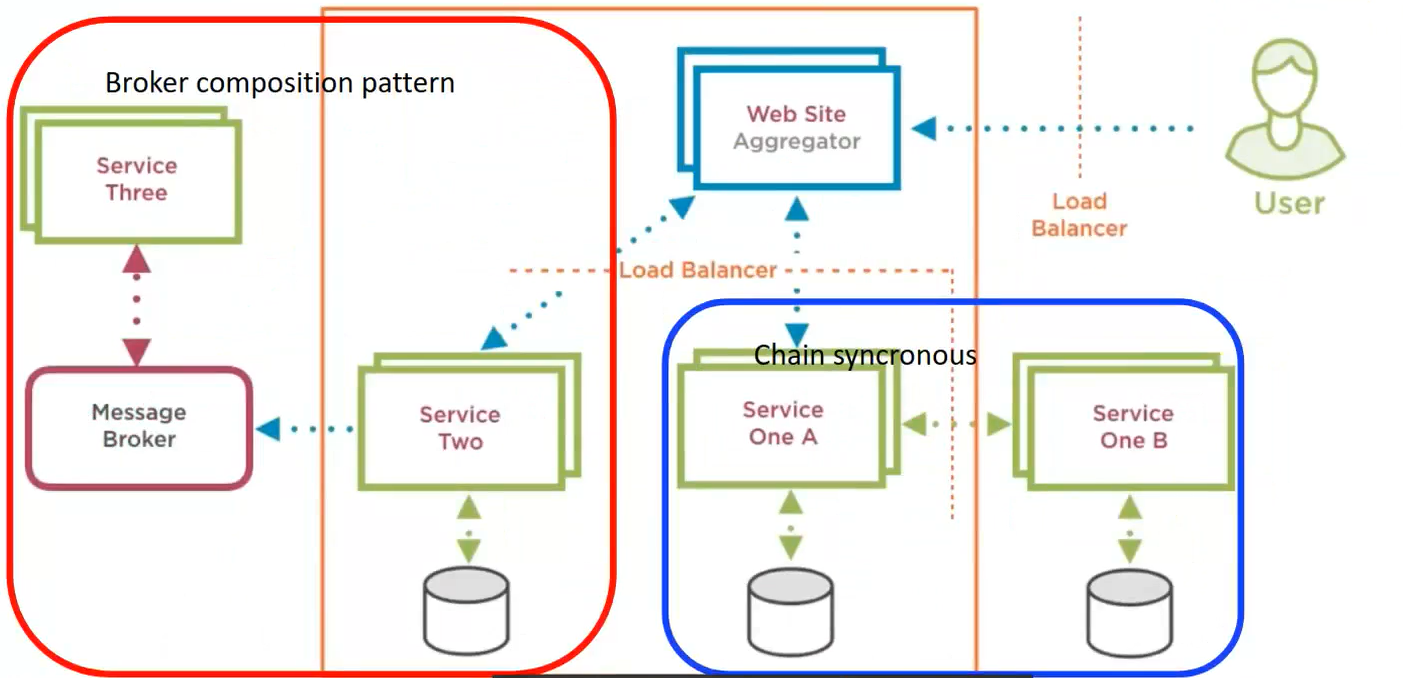


Il web si interfaccia a un microservizio e gli altri vengono richiamati a catena (tramite invocazioni sincrone)

#### Proxy composition pattern



#### Branch composition pattern



Architettura di comunicazione ibrida (alcuni servizi comunicano in maniera sincrona, altri in maniera asincrona)

### Pattern per la data consistency

I microservizi presuppongono che ci siano dei dati duplicati nei db dei singoli servizi, come occuparcene? Dopo che era stato superato il fat client si pensava che fosse superato questo problema, ma ora sta tornando di attualità

L’**ideale è fare delle transazioni** quando si eseguono operazioni sui vari db, ma il problema è che abbiamo una architettura distribuita, non è così facile fare ciò. Quindi si andranno ad usare dei pattern

* Pattern Saga
* Eventual consitency

#### Pattern saga

Si basa sulla presenza del **log**, esso colleziona i dati di tutte le operazioni fatte di scambio messaggi tra i vari servizi. L’idea è che devo tener traccia delle operazioni fatte. Si va a dividere la transazione in tante richieste che vengono tracciate. Se ci sono dei fallimenti (che si vedranno nel log), verranno fatte delle richieste di **compensazione**, quindi ci sarà un servizio, il **SEC** (saga execution coordinator), che si occupa di controllare il log. Il SEC invia le compensazioni per le Operzioni che hanno avuto fallimento. Il SEC quindi ha un suo database che contiene i log, si tratta di un rest service.

#### Eventual consistency

Si da prevalenza al fatto che il servizio sia disponibile piuttosto che dare consistenza, quindi ogni tanto potrebbe non esserci, ma in quel caso non è un problema.

### Database design

#### Greenfield database

Importante è fare l’analisi del dominio. Evitare di fare **data first**, cioè partire dal db per fare design, ma fare **function first**, cioè partire dalle funzionalità e poi in base ai dati che vogliamo salvare creiamo il db. Come faccio a disegnare prima le funzioni? Si pensa al contesto, a quello che deve fare il microservizio.

#### Brownfield database

Come nel caso dell’architettura brownfield, si va per **approssimazioni successive**.

# CMS

Per CMS si intende **configuration management system**. Al suo interno implementa il controllo di versione (git è un esempio).

Scenari che portano all’uso di un CMS

* Nella versione attuale c’è un bug, vogliamo tornare a una versione in cui non ci sono bug
* Ci sono più sviluppatori che lavorano ad un unico progetto (è possibile lavorare su versione locale e poi fare commit)

Perché gestire la configurazione del software?

* Più persone possono lavorare sullo stesso software
* Poter gestire la compilazione delle librerie nei vari sistemi operativi (serve coordinazione)
* Specificare i ruoli
* Permette di costruire software di qualità

Quali sono i ruoli?

* Configuration manager (responsabile di gestire le persone e definire release, promotion, …)
* Change control board member (approvano le modifiche fatte da un developer)
* Developer
* Auditor (valuta la consistenza, completezza, … valuta il programma, anche leggendo il codice)

Ci sono anche standard nelle aziende per dare i nomi alle cose (form, bug, …)

In generale, ci sono quindi le **change request** che possono riguardare

* Nuove funzionalità
* Bug

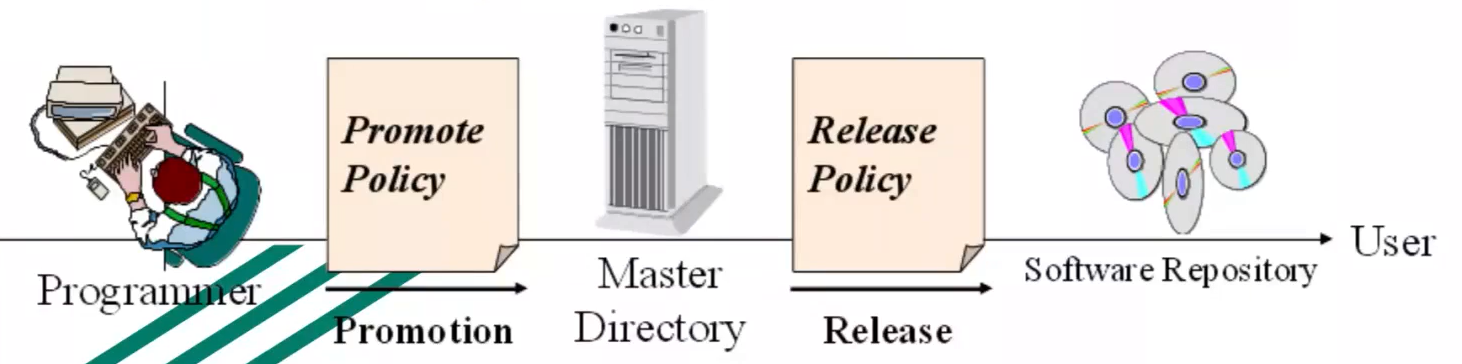
Un insieme di change request può portare a una nuova release.

Processo del change generale:

* La modifica è richiesta (questo può essere fatto da chiunque compreso utenti e sviluppatori)
* La richiesta di modifica viene valutata in base agli obiettivi del progetto
* A seguito della valutazione, la modifica viene accettata o rifiutata
* Se viene accettato, la modifica è assegnata a uno sviluppatore e implementato
* La modifica implementata viene controllata

È possibile commitare anche codice che non compila, basta che sia all’interno della policy del progetto. Basta definire la **promote policy**, cioè quando lo sviluppatore può fare la commit nel repository.

Per quanto riguarda la **release** invece bisogna decidere attraverso **release policy**, quando farla (di solito si ragiona sulla profondità dei test fatta)



Le policy possono essere informali o formali.

In un’azienda grossa ogni cosa che viene fatta deve essere registrata, quindi, bisogna indicare nei commit cosa si è fatto e il perché e rispetto a quale change request, quale urgenza, … deve essere tutto documentato.

La risposta a questi cambiamenti fatti sarà una valutazione dei cambiamenti, una valutazione dei rischi e le analisi di impatto (cioè, se è una modifica fatta in un’area localizzata oppure se riguarda molti pezzi del codice e quindi può essere difficile da valutare il livello di rischio)

Il **control board** non si occupa solo di valutare il codice, ma anche di valutare analisi e design. C’è quindi un controllo delle versioni anche per quanto riguarda i documenti di analisi e design

In queste situazioni rigide, prima di iniziare il progetto, bisogna decidere che tipi di linee guida seguire, quali policy attuare, quando una promotion diviene release

Noi saremo più flessibili e ci servirà per fare controllo versione, vedere chi ha fatto le modifiche, fare il merge del codice, …

Terminologia

* **Working copy**, copia locale del progetto
* **Revision**, un cambiamento committato che avrà un certo nome
* **Repository** (come repository remoto si può usare GitHub, BitBucket, …)
* **Check out**, porto un certo insieme di sorgenti dal repository pubblico al mio locale (nella mia working copy)
* **Commit**
* **Log message**, messaggio che descrive la modifica nel commit
* **Update**, la classica pull
* **Conflict**, quando ci sono conflitti nel codice, si andrà a fare merge manualmente

# Docker

Per alleggerire l’hosting dei microservizi ci sono stati diversi passaggi

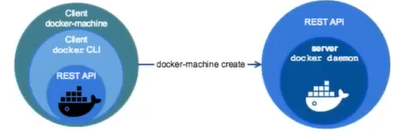
1. su server 🡪 costoso, in termini di manutenzione, hardware
2. su macchine virtuali 🡪 costoso, in termini di manutenzione, hardware (meno della prima soluzione, ma ogni VM doveva contenere un SO)
3. su **container** 🡪permette di usare un solo SO (quello che ospita i container) andando a creare componenti più piccole

in un container è possibile ospitare uno o più microservizi

## Container

Ci permette di impacchettare la nostra applicazione direttamente dentro il container e di eseguirla tranquillamente. Un’applicazione funzionante messa su un container può essere eseguita tranquillamente da altri utenti senza doversi preoccupare di settare un determinato ambiente o avere determinate librerie.

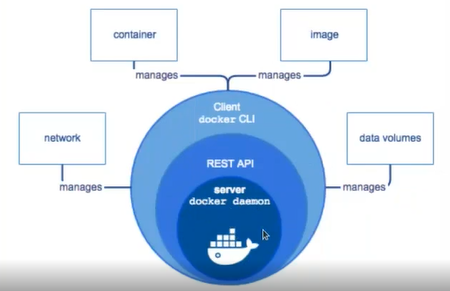
## Componenti docker



Comprende

* **docker** **machine**, entry point a docker per poter inserire i container (dalla command line vengono inviati i messaggi alla docker machine)
* **docker** **compose**, gestisce i container (li tira su)
* docker registry, è un servizio di storage per immagini docker (è possibile usare quello pubblico o privato); funziona un po' come il repository Maven pubblico, ma al posto delle librerie contiene immagini
* **docker** **image**, istantanea dell’applicazione, per essere eseguita bisognerà caricarla in un container e mandarla running
* **docker** **volume**, ci permette di salvare i dati alla cancellazione del container
* **docker** **engine**, software core che gestisce immagine, container, volumi, …
* **docker** **file**, documento che specifica l’immagine
* **docker-compose.yml**, specifica i container che lavorano insieme
* **docker** **network**, permette di far comunicare i container all’interno di docker

### Docker engine



Il **Server Docker deamon** è continuamente in ascolto e serve, come fa un container per Java enterprise, o come fa Tomcat per le Servlet, per far girare l’intera applicazione.

Da cli possiamo dare i comandi al server docker deamon

# Kubernetes

Kubernetes è un **orchestratore di container**, che si occupa di garantire

* High availability
* Scalabilità (definita in maniera dichiarativa)
* Disaster recovery

Potremmo fare girare K. su un cloud, così da non dover caricare direttamente i nostri servizi e avere la possibilità di poter migrare da un provider all’altro senza problemi, oppure on-premise, quindi sul proprio pc.

K. è formato da cluster che contengono nodi. I nodi dentro un cluster possono essere divisi in **master node** e **working node**.

## Master node

Il master node si occupa di gestire i working node. Il master, tra le sue componenti ha

* **Server api**, entry point per inviare il file yalm con le istruzioni da eseguire per caricare i container
* **Controller manager**, esegue comandi specificati nel file yalm e supervisiona i cluster per intervenire in caso un nodo dovesse cadere
* **Scheduler**, pianifica i container sui diversi nodi in base a carico di lavoro e risorse possedute dai nodi
* **Cluster store**, mantiene lo stato del cluster

Il cluster all’interno possiede una rete virtuale per far si che i nodi possano comunicare tra di loro

## Working node

Il working node è una componente che contiene uno scheduler (agente che gestisce i pod) e una serie di **pod**. L’agente si occupa di eseguire i comandi ricevuti dal master e di gestire i nodi.

I pod all’interno di un nodo possono comunicare tra di loro tramite un network. Ogni pod ha uno specifico IP che cambia nel caso in cui dovesse cadere ed essere tirato su. Container all’interno di uno stesso pod hanno l’IP del pod ma porte differenti.

Il punto è che i pod hanno IP dinamici, quindi c’è bisogno del **service**, che si occupa di fare da tramite tre le richieste ai pod e i pod veri e proprio (associa la richiesta a IP e porta corretti). Il **service**, oltre a ciò, fa da **load balancer** per le richieste.

PS. Su un singolo pod conviene inserire pod che comunicano tanto o che utilizzano lo stesso database; l’ideale comunque è avere container loosley coupled e quindi in quel caso dovranno stare su pod diversi

## Tips

* KubeCtl è la command line per inviare comandi al server API
* È possibile assegnare etichette a versioni diverse dei servizi, per indicare la versione e far capire al service su quale versione fare affidamento
* Gestisce in maniera ottima l’update dei servizi (perfetto per la continuous integration)

# Cloud computing

Si tratta di un paradigma di fornitura delle risorse da un fornitore ad un cliente attraverso rete internet. Queste risorse vengono fornite dal **cloud service provider**.

Il **cloud computing** consiste nell’esecuzione del carico di lavoro in un ambiente **cloud**. Ciò viene fatto perché tutti usano Internet, quindi viene visto come un’**estensione del computer**, quindi, non prendiamo solo più dati da internet, ma anche software. Ciò comporta che i dati non risiedono più sul nostro pc ma saranno online.

Possono essere distinti in maniera generale tre tipi di servizi cloud:

* Infrastructure as a service (host)
* Platform as a service (build)
* Software as a service (consume)

Il cloud è un **ambiente multitenant**, cioè un’architettura software, in cui si avranno più istanze (chiamate tenant) di una o più applicazioni che operano in maniera indipendente in un ambiente condiviso. Quindi queste istanze sono separate in maniera logica ma risiederanno fisicamente sullo stesso hardware. Si ha l’obiettivo di fornire robustezza e sicurezza, per fare ciò bisogna gestire bene gli utenti (operazioni che fanno, allocazione di risorse, …).

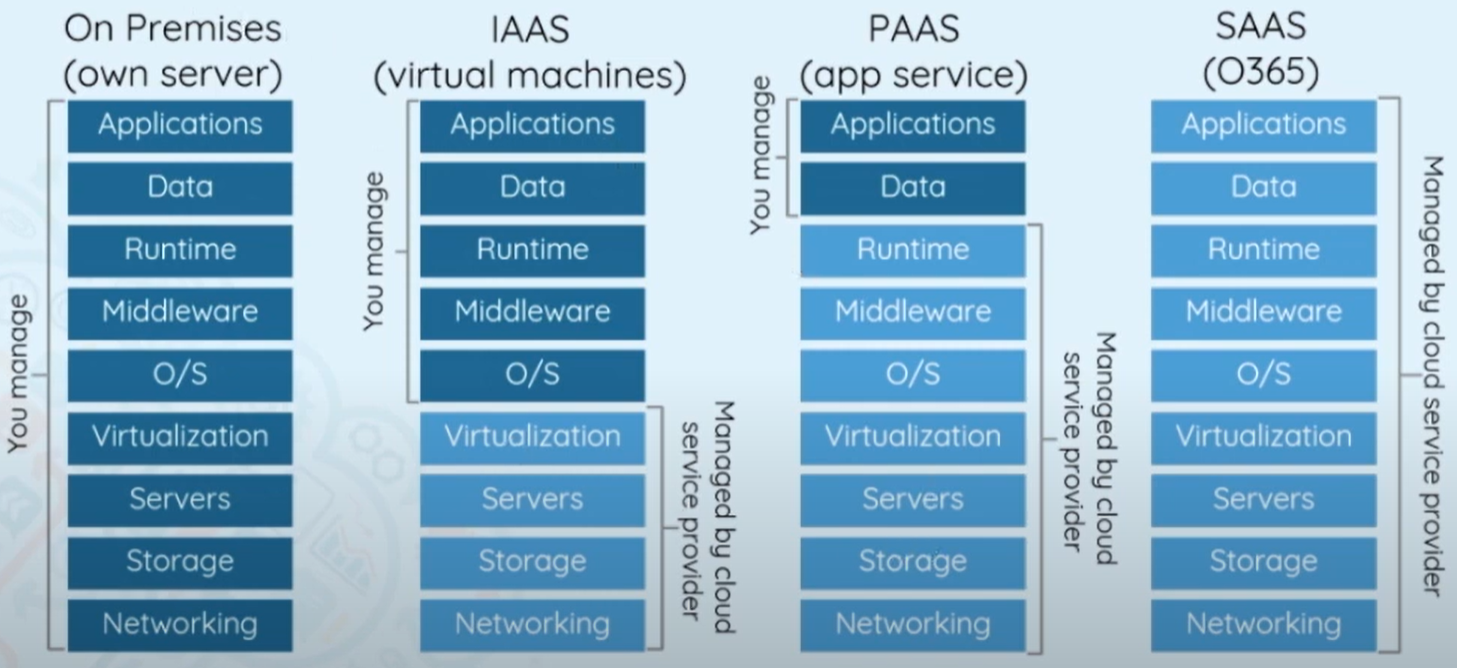
Quindi ambiente multitenant possiede più istanze (ognuna usata da un utente diverso) che risiedono sullo stesso hardware e condividono risorse. Un app **single tenant** possiederà una sola istanza (per un solo utente) e utilizzerà un set di risorse (hardware, network, …)

Vantaggi di un’architettura multitenant

* **Risparmio** **di** **denaro** (più economico eseguire quel servizio sul cloud rispetto ad eseguirlo sul proprio pc, soprattutto per le grosse aziende)
* **Flessibilità**, le risorse sono allocate per gli utenti quando ne hanno bisogno
* **Efficienza** (e risparmio di tempo), i singoli utenti non si dovranno preoccupare di gestire manutenzione, aggiornamenti e infrastruttura

I software “on-premise” (che risiedono sul pc) sono in contrapposizione con i software come servizio (SaaS). I vantaggi dei sistemi on-premise sono

* I dati rimangono sul nostro pc (sul cloud non si sa dove sono allocati fisicamente i nostri dati)



Lo spazio cloud può essere pubblico, privato, community o ibrido.

## Servizi

### IaaS

Si parla in questo caso di host, perché questo servizio fornisce all’azienda che lo utilizza server, reti, spazio di archiviazione, backup, in genere tramite **pagamento a consumo**.

Un utente IaaS solitamente è una software house. Essa non gestirà il sistema hardware ma avrà il controllo dell’OS, dei dati, delle applicazioni, …

Esempio di IaaS: Microsoft Azure, Amazon AWS, Google computer engine, …

### PaaS

Si usa il termine build, in quanto è una piattaforma rivolta agli sviluppatori. Permette loro di sviluppare applicazioni sopra questa infrastruttura. Non bisognerà preoccuparsi della configurazione e manutenzione del server.

Il provider fornirà la **solution stack**, che conterrà linguaggi di programmazione, librerie, servizi, … che gli consentiranno lo sviluppo delle applicazioni. L’utente si occuperà di gestire solo dati e applicazioni

Esempio di PaaS: Heroku

### SaaS

Si parla di consumo, gli utenti possono essere di ogni tipo. Prevede che il software (un’applicazione) sia erogato come servizio dal provider. Possono essere gratuiti o a pagamento.

Permettono di usare software sul browser senza doverli installare sul proprio computer. L’utente non potrà gestire nulla, potrà solo usare il software finale

Esempio di SaaS: Google gmail, Microsoft office, …

### DaaS

Data as a service, gli utenti hanno a disposizione dati

### Haas

Hardware as a service, affitto di server utile per poter incrementare la propria capacità elaborativa

### CaaS

Container as a service, offre spazio per ospitare container, cosi da rendere le componenti facile da migrare da un provider all’altro

## Funzionalità cross dei servizi

I servizi devono poter offrire le seguenti funzionalità

* Gestione di accesso alla rete
* Elasticità (aumentare o diminuire facilmente la capacità di calcolo, memorizzazione, …)
* Pooling delle risorse (risorse assegnate in modo dinamico dal servizio, senza che l’utente se ne preoccupi)
* Contabilizzazione dei corsi
* Controllo delle risorse (devono essere sempre efficienti)

## Problematiche

Le problematiche del cloud computing sono

* Sicurezza informatica e privacy
* Problemi internazionali di tipo economico politico
* Continuità del servizio offerto (se server cade l’utente dovrà aspettare che il provider lo faccio tornare attivo, oltre a ciò, il malfunzionamento colpire un numero molto elevato di persone in contemporanea)
* Difficoltà di migrazione dei dati (nel caso di cambio del servizio cloud)
  + Dannoso nel caso di cambio di tariffario

Questo perché non esistono standard

## Heroku

Si tratta di un Paas e supporta i seguenti linguaggi

* Php
* Node.js
* Java
* Python
* …

I Dynos sono dei servizi offerti da Heroku, alcuni sono gratis altri no. Il Dynos che permette di usare postgres è gratuito. Sono dei gettoni per l’uso di componenti nella piattaforma. È possibile sviluppare la propria applicazione e deploy su Heroku. La URL sarà quella di Heroku. Gratis si possono avere 5 applicazioni running in parallelo. Heroku e git possono lavorare insieme