# SOLID Principles

S(ingle Responsibility Principle) -> ogni parte dovrebbe fare una sola cosa , essere incapsulata ed indipendente. Ogni parte dovrebbe essere dipendente da un solo attore (business). Tutte le parti di questa dovrebbero essere altamente coese.

O(pen-close principle) -> un modulo dovrebbe essere modificato soltanto aggiungendo codice, non riescrivendo quello già scritto. Questo vuol dire che il codice del modulo deve essere estendibile.  
  
L(iskov substitution principle) -> i subtypes dovrebbero essere sostituibili tra loro poiché gli utilizzatori utilizzano astrazioni (es classi astratte o interfacce) e quello da cui dipendono è il comportamento (e non i dati). Puo’ essere anche enunciato come “bisogna lavorare per contratti” (interface abstraction)

I(nterface segregation) -> un modulo non dovrebbe dipendere da cio’ che non usa

D(ependency inversion) -> gli attori (componenti) piu’ astratti non dovrebbero dipendere da quelli piu’ specifici, ma viceversa sì.

# DDD (Domain Driven Design) for microservices

Quando si vuole modellare un sistema , ad esempio a microservizi, partendo dal dominio di business, si creano moduli (attori, es microservizi) ciascuno per ogni macrocomponente di business.  
Si accoppia quanto piu’ possibile le componenti altamente coese (Single Responsibility) nello stesso modulo (es microservizio) e si tengono disaccoppiati i moduli (es microservizi) poco coesi e disaccoppiabili.  
A quel punto ogni modulo (es microservizio) avrà un suo DB.  
A quel punto ad esempio si puo’ creare un facade o saga pattern (orchestratore come facade, o come saga pattern di coordinazione) nel caso dei microservizi.

# TDD (Test Driven Design)

Si parte dai requisiti funzionali e non funzionali.  
Si scompongono tutti i requisiti funzionali e non funzionali nei micro-requisiti.  
Si creano le storie.  
Si parte implementando i microrequisiti sulla base dei test che devono rispettare, e si creano gli unit testing prima del codice.

### Proxy VS Reverse Proxy

Proxy -> nasconde il client  
Reverse proxy -> nasconde il server (destinazione).  
Un’api gateway è un reverse proxy.  
Una vpn è un proxy.

# Scaling vs Sharding vs Normalizzazione DB

**Scaling** -> aumento delle risorse. Puo’ essere verticale (aumento le risorse ad una macchina) o orizzontale, duplico le macchine.  
**Sharding/Partitioning in un DB** -> le righe di una stessa tabella le divido su macchine differenti del db, o divido le colonne della tabella su macchine differenti (NB: DIVISIONE FISICA, LA TABELLA NEL DB E’ COMUNQUE LA STESSA ENTITA’ NELL’EER se relazionale)  
**Normalizzazione tabelle DB** -> una tabella la splitto in piu’ tabelle dal punto di vista LOGICO (indipendentemente da come salvo fisicamente i record, questa è una divisione in due differenti entità)

# Https (e TSL)

Https è la versione criptata di http.  
Https è ottenuta usando a livello transport un protocollo tcp/ip criptato (con chiave asimmetrica), questo è chiamato TLS (transport secure layer).

# OAuth2

Meccanismo che permette di utilizzare, per dare parte delle proprie info ad un sito (evitando di iscriversi) una fonte di terze parti presso la quale siamo già registrati (es google).  
Il sito che vuole le info ci chiede di avviare la sessione con l’auth manager di terze parti (es google).  
Noi accettando veniamo rediretti sulla pagina di terze parti. Accettando, e loggando, l’auth manager di terze parti manda un token autorizzativo al sito, e questo verrà usato dal sito per contattare direttamente il resource manager di terze parti per avere le nostre informazioni.

# CSRF-Attack

Cross Site Request Forgery -> quando il sito dell’attaccante manda codice javascript che esegue chiamate verso siti terzi leciti (es banca) con i dati dell’utente. Se questi ultimi usano, per l’autenticazioni, info locali al client (es sessione, jwt) e la chiamata viene eseguita quando l’utente si era precedentemente autenticato, il sito lecito pensa che la chiamata sia lecita e quindi esegue quanto richiesto. -> Soluzione CSRF Token  
Tipicamente il CSRF Token serve per proteggere solo le interazioni con browser e solo i PUT/POST/DELETE (non le GET).  
Tipicamente il client manda richiesta GET al server. Il server risponde con XSRF-TOKEN, che viene salvato nei cookie dal client. Il client manda successive richieste POST/PUT/DELETE rimandando il token indietro nell’header e il server sa che le richieste arrivano dal client valido.  
Questo cookie tipicamente non è attaccabile da javascript , poiché settato come cookie HttpOnly (cioè non modificabile da js)  
Il motivo per cui tipicamente il CSRF-Token si usa sui PUT/POST/DELETE e’ perche’ le GET dovrebbero essere di sola lettura ovviamente

# XSS-Attack

Cross Site attack -> tramite un sito lecito ma con campi non protetti, viene fatto injection di codice javascript che esegue un attacco come il CSRF.

# CORS-Policy

Cross Origin Resource Sharing -> la politica che fa si che il codice js in una pagina possa chiamare solo risorse dello stesso dominio della pagina (non blocca il csrf poiché il codice malicious viene eseguito sul dominio malicious e quindi il cors sarà disabilitato, ma puo’ bloccare xss)

# Throttling

Quando una chiamata ad un’api / servizio (tipicamente fatto da un api gateway o da un load balancing) viene rifiutata perché il totale delle chiamate fatte da quel client è > certa soglia.  
C’e’ **hard throttling** (soglia definita) , **soft throttling** (% in piu’ del max) , **adaptative throttling** (la soglia dipende dall’uso totale del sistema).  
Inoltre c’e’ **fixed window throttling** (il numero di chiamate vengono contate in una finestra temporale il cui inizio è scandita dall’orario del sistema, indipendentemente da quando arriva la prima richiesta del client per quella finestra).  
**Rolling Window throttling** invece quando si inizia a contare il numero delle risorse, in una finestra di tempo, che inizia quando arriva la prima chiamata del client.

# Transazioni distribuite : 2PhaseCommit (2PC) vs Saga

Il saga è quello classico (coordinatore oppure coreografia, central bus o a stella etc).  
Nel saga pero’ ogni attore committa sul suo DB appena termina, e avvisa il coordinatore (o il successivo attore) quando ha terminato. Se uno fallisce, avvisa il coordinatore che avvia le politiche di rollback (vantaggi: elevato throughput ma la transazione globale non è ACID, ed è di difficile implementazione).  
Nel 2PC invece c’e’ un coordinatore della transazione (NB: differente dal coordinatore del SAGA perché nel SAGA il coordinatore coordina proprio le operazioni degli attori, nel 2PC il coordinatore della transazione coordina solo i commit/rollback, ma potrebbe essere un attore differente da quelli dei microservizi, non necessariamente il coordinatore).  
Quando un attore nel 2PC termina, non committa subito, avvisa solo il coordinatore della transazione che ha terminato. Quando tutti hanno terminato (e avvisato il coordinatore della transazione) questo invia a tutti gli attori in attesa di committare, un “PREPARE TO COMMIT”. Se tutti gli attori che lo ricevono rispondono con SUCCESS, allora il coordinatore della transazione invia a tutti gli attori (SECONDA FASE QUESTA) un COMMIT., e tutti gli attori che lo ricevono committano.  
Se invece almeno un attore invia un FAIL, allora il coordinatore della transazione invia a tutti gli attori un ROLLBACK.  
Questo piu’ garantisce le transazioni distribuite ACID, ma abbassa il throughput perché gli attori devono stare in attesa di ricevere il COMMIT, e quindi tenere lockata la risorsa (che non puo’ essere intanto letta da altri thread/processi per non avere dirty read).  
Inoltre questo approccio è un Single Point of Failure perché se cade il coordinatore della transazione non si potranno avere piu’ commit /rollback.   
  
Tipicamente mentre per il SAGA deve essere lo sviluppatore ad implementarlo, per il 2PC esistono soluzioni software trasparenti.  
Queste si basano su **Java** **JTA** (java transaction api) che a sua volta si basa su **java** **XA** (eXstended architecture) che è un protocollo per gestire in java il 2PC.  
Un software che si basa su JTA/XA è ad esempio **Atomik.**Questi software che offrono un 2PC “inscatolato” necessitano tipicamente di appoggiarsi ad un bus di messaggi (kafka,jms …) per poter far girare i messaggi di prepare commit e commit/rollback, e gli ack dei client.   
Inoltre bisogna assicurarsi che il DB usato supporti il 2PC (spesso pero’ questi software permettono di tentare un best effort di 2PC anche su DB che non accettano le transazioni distribuite, questo è detto **2PC : Last Resource Gambit**)  
**NB: Atomik ha un limite, e cioè che la transazione è distribuita sì, su diversi DB, ma deve essere avviata da un unico attore (quindi non supporta microservizi !!!! )**Oppure Jboss ha un suo transaction manager che supporta JTA/XA nativamente

# Politiche per chiamate http di lunga durata (http push)

Tipicamente l’http classico è stateless e connectionless.  
Il modo in cui lavora è tipicamente detto http pull (il client prende informazioni dal server con una richiesta).  
Quando si vuole ottenere l’http push, cioè il server fa push al client, in maniera “async” , ci sono varie tecniche:

## http Ajax Polling

Visto che l’http è connection-less, vengono fatte continue chiamate ajax di brevissima durata, connection-less, fino a che il server non risponde con il risultato dell’elaborazione.

## http Long (ajax) polling

Viene fatta una chiamata http ajax, il server non risponde subito ma si mette in attesa.  
O va in timeout o risponde col risultato dopo tanto tempo.  
In entrambi i casi il client rilancia poi una nuova long call.

## WebSocket

WebSocket crea invece un meccanismo (a livello transport) con connessione. E client e server possono ricevere e inviare dati su quel canale (quindi con connection e full duplex)

## SSEs

Server-Sent Events-> come il websocket, con connessione, ma half duplex (dati solo dal server al client)

# System Design steps Requisiti :

Raccogliere i requisiti funzionali.  
Da questi , organizzare e raccogliere i requisiti non funzionali.

Flussi  
Capire i flussi delle operazioni, identificarli (es inserimento, salvataggio) e i vari giri.

### API :

Capire le API, e i contratti.  
Accoppiare e disaccoppiare le parti, per creare moduli altamente coesi, e disaccoppiare i poco coesi.

Requisiti di Sistema applicativi:  
Capire i volumi in gioco (quante richieste lettura al secondo, quante richieste scrittura al secondo, quanti dati salvare, quanti server quindi, quanto spazio per la cache, quanta % cachare).   
Necessità di scalare verticalmente o orizzontalmente ?

### Database

Capire gli attori in gioco e le componenti sul modello dati (es usando DDD per la base dati).  
Scegliere nosql o sql (nosql se dati con schema che cambia spesso, grossi volumi, e non c’e’ necessità di transazioni acid, sql se invece dati fortemente strutturati con schema che non cambia mai, e necessità di proprietà acid).  
Partitioning/sharding verticale o orizzontale per le righe del DB ?   
Scaling verticale o orizzontale per le macchine DB ?   
(NB: il partitioning verticale o orizzontale delle righe del db, sono sempre scaling orizzontale dal punto di vista delle macchine)  
Tipicamente i Nosql si adattano di piu’ a scaling orizzontale delle macchine, mentre i SQL a scaling verticale.  
La scelta del db dipende anche dal fatto che non è possibile ottenere tutti e 3 gli angoli del CAP theorem. A seconda dei 2 che si vogliono ci sono soluzioni db.

C(onsistency)  
A(vailability)  
P(artition tolerance).  
  
Availability e Resiliency  
Ci sono colli di bottiglia ? Come risolverli ?  
Ci sono SPOF (single point of failure) ?   
Caching ?  
Dove mettere i load balancer ? Sia davanti alle api applicative, che tra l’applicativo (application server) e i db ?

C’e’ necessità di fare throttling ?