APPLICAZIONI INTERNET

(Riassunti)

Studente: Jeanpierre Francois s243920

a.a. 2018/19

Indice

MALNATI

01-HTTP

Protocollo applicativo, semplice, stateless ed indipendente dal contenuto.
Modello richiesta/risposta, intestazione+corpo (corpo è tipo MIME).
Azioni: POST, GET, PUT/PATCH, DELETE.
Codici di stato:

- 1xx Informational,
- 2xx Successful,
- 3xx Redirection,
- 4xx Client error,
- 5xx Server error.

Oss: Http tratta tutte le richieste come tra loro indipendenti.

Versioni protocollo:

- 1.0 Ogni richiesta una connessione, plain text
- 1.1 Una connessione più richieste (cmq indipendenti per il server)
- 2.0 Singola connessione TCP più transazioni, **protocollo binario**.

Lo standard http prevede quattro possibili ruoli:

- User-agent, app che origina le richieste HTtp
- (Origin) Server, app che ospiti le risorse trasferibili via HTTP, accetta le richieste e genera le corrispondenti risposte
- Proxy, intermediario scelto esplicitamente dallo user-agent per inoltrare le richieste ad un server, può filtrare le richieste, tradurre gli indirizzi, servirsi di una memoria tampone (cache) per aumentare le prestazioni
- Gateway (reverse proxy), intermediario trasparente allo user-agent (che lo crede l'origin server), inoltra le richieste all'origin server effettivo.

Autenticazione:

- Basic authentication (dalla 1.0), psw in base64
- Digest access authentication (dalla 1.1), digest derivato dalla psw

Sessione:
Livello applicativo, un insieme di richieste provenienti dallo stesso browser verso lo stesso server in un lasso di tempo limitato volte ad interagire con risorse correlate.
Anonima o nominali. Ha un ID.

02- ANNOTAZIONI JAVA

public @interface AnnotationName{}

Permettono di inserire metadati all'interno del codice che descrivono caratteristiche di una classe.
Info al compilatore (eg *@suppressWarnings*), durante la compilazione o in esecuzione.
Poste prima delle dichiarazioni degli elementi, se prevedono un singolo elemento deve chiamarsi "value".

Possono essere annotati: Packages, classi, interfacce, enum, metodi, campi, (in compilazione) variabili locali e parametri. Predefinite: *@Deprecated, @Override, @SuppressWarnings.*

Le **meta annotazioni** servono ad annotare le dichiarazioni delle annotazioni:

- *@Target*, a che elementi è applicabile
- *@Retention*, quando (source, Class, Runtime)
- *@Documented*

  indica che l'annotazione deve essere inserita nella documentazione generata da Javadoc
- *@Inherited*, ereditata dalle sottoclassi

Famose le annotazioni JPA(*@Entity,@Table*..) o JUnit (*@Test, @Before, @After, @Ignore*..)
E' possibile creare classi che gestiscono annotazioni in fase di compilazione le quali implementano l'interfaccia *javax.annotation.processing.Processor* e ricevono una callback relativa alle classi che dichiarano una specifica o generica annotazione.

E' possibile estendere *javax.annotation.processing.AbstractProcessor* per avere alcuni comportamenti base già implementati, deve poi però fornire l'implementazione del metodo boolean process(...). L'elaborazione delle annotazioni avviene in cicli successivi (round) (a turnate diciamo).

LOMBOK è un'estensione del compilatore che converte le annotazioni presenti nel sorgente java (Eg *@Data* crea getter/setter/toString.. campi mutabili, *@Value*-> immutabili e privati, *@Builder*...).

Le annotazioni Java rappresentano una standardizzazione dell'approccio di annotazione del codice e non riguardano direttamente la semantica del programma ma possono semplificare notevolmente la quantità di codice che è necessario scrivere a mano, senza compromettere le funzionalità.

03-MAVEN

E' uno strumento di supporto per creare e mantenere il sw, è un'evoluzione di *make* con supporto all'importazione delle dipendenze, creazione di moduli differenti ed utilizzo di plugin per la compilazione e deploy.

Supporta le fasi del **ciclo vita di un prodotto sw**:

1. Compilazione del sorgente
2. Collegamento con le risorse
3. Compilazione ed esecuzione dei test
4. Preparazione pacchetto finale
5. Messa in campo
6. Rimozione dei file intermedi

Le singole operazioni sono eseguite da plugin.

POM è un file xml che descrive la struttura di un progetto (nome,versione,dipendenze,plugin..), è un packaging. All'interno del POM il progetto e gli artefatti utilizzati vengono identificati in modo univoco tramite la notazione **GAV** (*groupId:artifactId:version*) :

- groupId, identificatore del progetto (simile a package)
- artifactId, nome arbitrario del progetto (id java)
- version, *Major.Minor.Maintenance*

Un progetto Maven può essere composto da un insieme di moduli, i quali sono in sotto cartelle e hanno un proprio file pom.

Creazione artefatto: *mvn install* (clean o test clean)
Dipendenze => Locale o Maven repository.

Per le dipendenze oltre al gav si aggiunge lo scope(fase ciclo vita in cui servono).

Tutte le operazioni svolte da Maven avvengono usando appositi plugin.

04- Applicazioni JAVAEE:

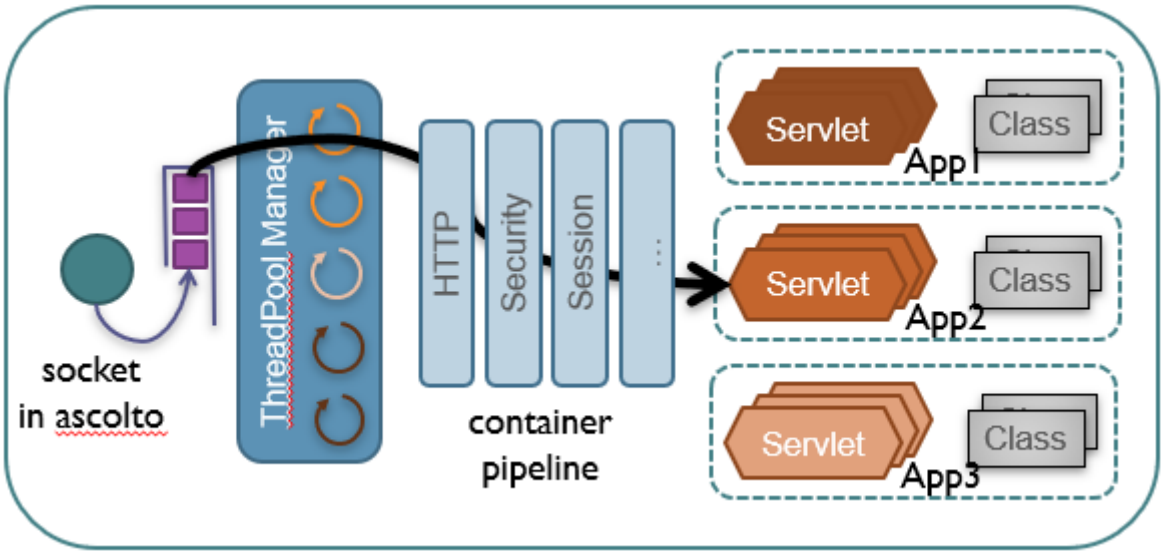
Le applicazioni web sono eseguite in modo discontinuo (reattivo), ogni volta che ricevono una richiesta producono una risposta specifica per la sessione in corso. Sebbene http sia stateless, introducendo il riferimento all'id della sessione si può ottenere un interazione stateful.

La navigazione può essere effettuata pagina per pagina, utile per farsi indicizzare dai motori di ricerca, o single page utilizzando ajax (Angular aiuta).
La single page application (SPA) oltre ad avere problemi di indicizzazione dei motori di ricerca (via crawler) necessita la gestione dello storico di navigazione (tasto indietro) e del ciclo vita della pagina.

Un'applicazione JavaEE è un insieme modulare di classi java dentro un programma contenitore che ne gestisce il ciclo vita. Le app JavaEE sono indipendenti dallo specifico contenitore e dall'OS del server.

Un programma contenitore monitora, gestisce, applica sicurezza in entrata e uscita, inoltra le risposte, instrada le richieste alla app/classe giusta. Utilizza un template standard in xml (Es tomcat).

Container JavaEE:



Le app JavaEE sono solitamente distribuite in **web archive (.war)** il cui nome solitamente definisce la url di base. Oltre ai file di configurazione e contenuti statici un war comprende i componenti elementari di una app JavaEE: Servlet, filtri, pagine JSP, Listener.

Il **Servlet** è la classe java per gestire le richieste ricevute dal contenitore, il quale ne gestisce il ciclo vita e la politica di concorrenza da adottare.
Il ciclo vita del servlet:

1. Contenitore invoca *init()* (e bom)
2. Ad ogni richiesta il contenitore invoca *service()* utilizzando un thread differente.
3. Per rimuovere il servlet il contenitore invoca il metodo *destroy()* (lo fa terminare istant)

HttpServlet è specializzato al trasporto di http, il metodo service() reindirizza ai metodi *doGet, doPost, doPut, doDelete*. Conf del servlet => annotazione *@WebServlet(...)* ove definire anche l'url del servlet.

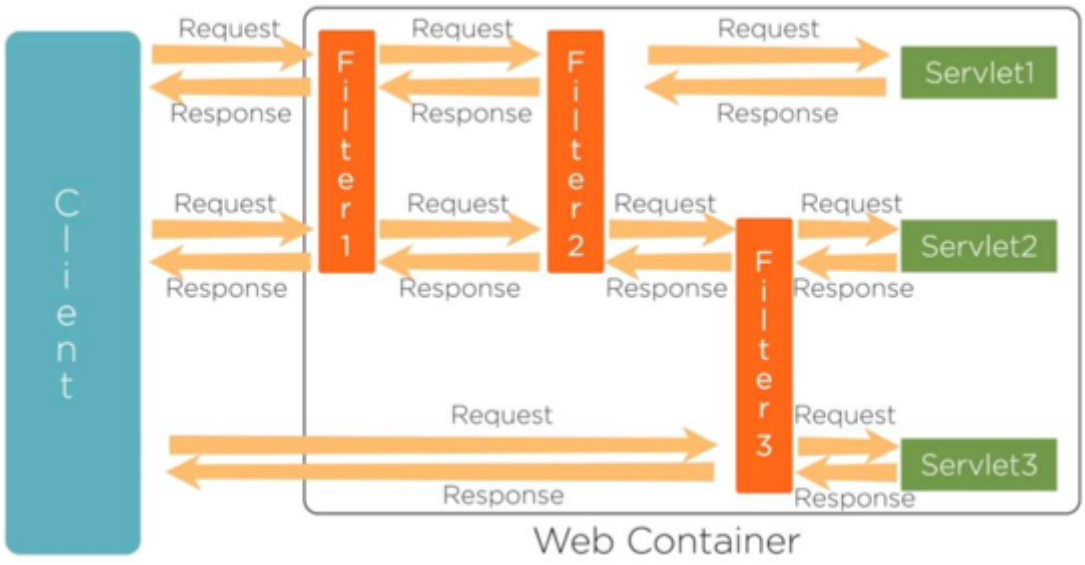
ServletContext è un'interfaccia che definisce le interazioni tra servlet e contenitore.

HttpSession è un'interfaccia per le info riguardanti la sessione di lavoro corrente.

Servlet vantaggi:
veloci, efficienti, scalabili, persistenti, flessibili, il contenitore garantisce la separazione fisica tra applicazioni (vengono usati threadGroup differenti).

Svantaggi:
Complicato modificare risposta html, presentazione e logica fortemente accoppiate, l'architettura può diventare complessa.

Filtri sono componenti java che trasformano richieste e risposte da/a un servlet. Più filtri possono essere collegati in cascata, seguendo il file web.xml che descrive anche la corrispondenza tra url e filtri. Sono utili per authN, compressione, crittografia...
(javax.servlet.Filter ---- @WebFilter).

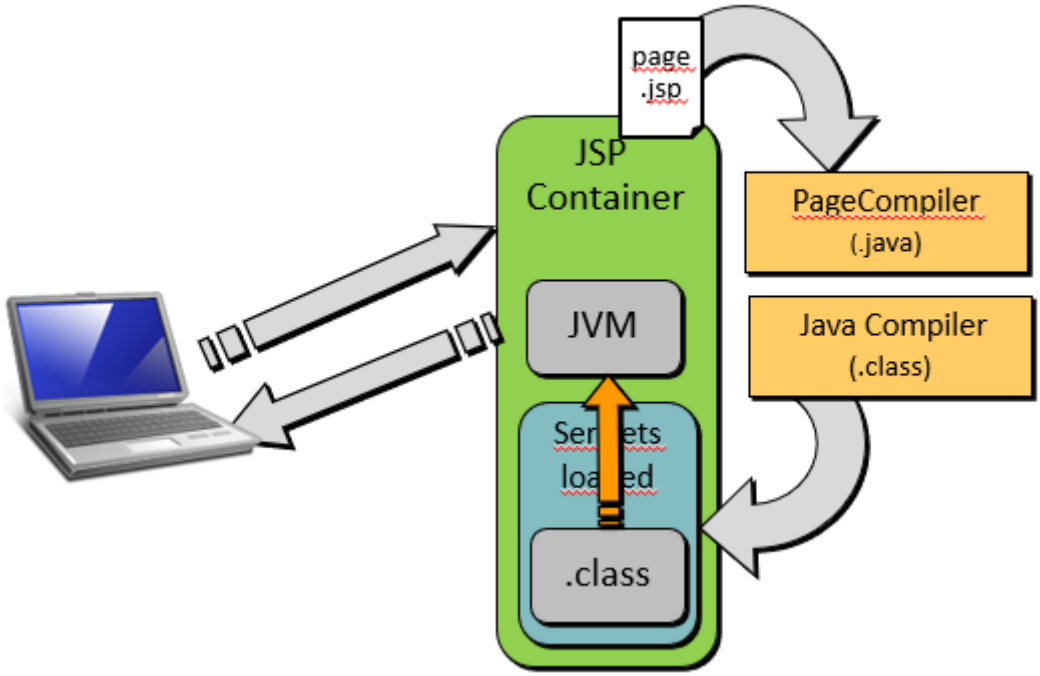


Java Server Pages (JSP) contengono testo (html,xml...), frammenti di codice java e direttive per l'ambiete di esecuzione. Vivono nel contenitore che ne gestisce il ciclovita, esecuzione, http e supporto alle sessioni. Le jsp generano contenuto dinamico per i browser (simil php).

Sono composte da tag html e tag jsp: Commenti, direttive, Scriptlets, JavaBeans.

Il **modello operativo:**

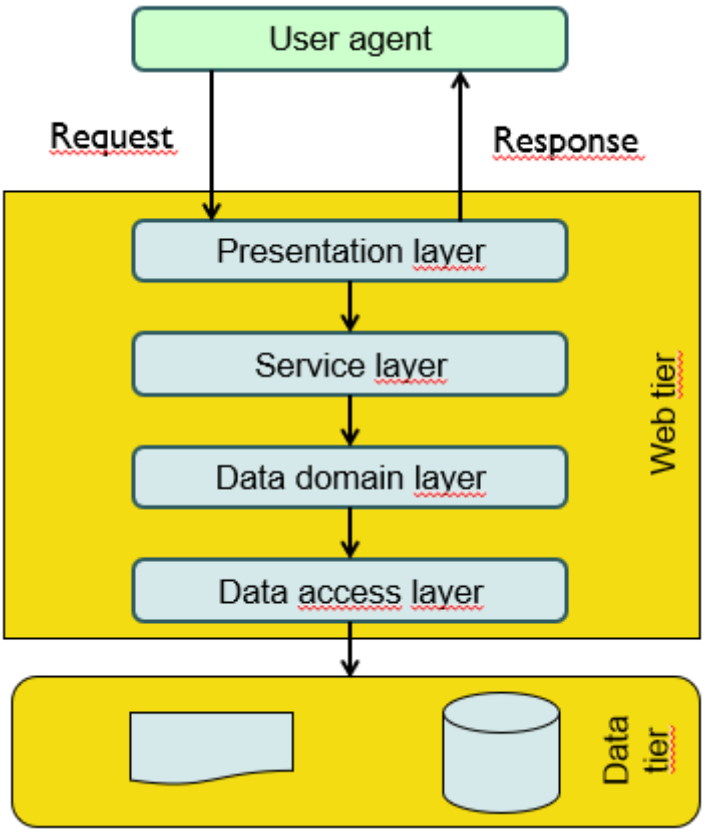
1. Il contenitore riceve la richiesta della pagina JSP direttamente o tramite un server HTTP
2. Il contenitore identifica il file richiesto e lo trasforma in classi sorgenti Java attraverso un programma specifico (*pageCompiler*)
3. Il compilatore Java compila la classe prodotta, ne crea un'istanza e la caricata nel contenitore
4. La richiesta ricevuta e i relativi parametri vengono inviati all'oggetto generato (servlet) che la elabora
5. Il risultato viene inviato al client



Listener gestisce l'insieme dei componenti ospitati all'interno di un contenitore JavaEE, governandone il ciclo vita, politiche di attivazione e controllandone risorse e concorrenza. Cambiamento di stato => listener.

05- Model View Controller (MVC):

Le app web, lato server, hanno più strati:
data access, data domain, service e presentation layer.



- **Data access layer** si occupa del trasferimento delle singole entità da e verso la sorgente dei dati, spesso costituito da un insieme di data access object (DAO) (Eg. classe UserDao --> persistenza di User).

- **Data domain layer** è il modello logico dei dati cui deve essere applicata la logica di business dell'app, si appoggia al data access per reperire/rendere persistenti le informazioni che lo alimentano. (eg. modello classe User)
- **Service layer** definisce il contratto tra utente e app (business logic), coordina e delega i vari compiti agli oggetti del dominio applicativo. (Eg. funzione register)
- **Presentation layer** trasforma le richieste in operazioni di servizio ed incapsula i risultati all'interno delle risposte che dovranno essere visualizzate. Può interagire con le sessioni.

Per gestire la complessità dei programmi è utile usare approcci come il Model-View-Controller (MVC).
Il modello contiene i dati dell'app e impone delle regole sulla loro evoluzione (data, domain, service).
La vista è una rappresentazione visuale del modello (html o xml riempiti dinamicamente, solitamente >1).
Il controllore intercetta le richieste, invoca lo strato di servizio e sceglie la vista da presentare la quale verrà popolata coi dati ottenuti dal modello.
Per ogni url occorre definire un controllore con i relativi metodi.

OSS: il modello dovrebbe essere il punto di partenza nello sviluppo di un progetto.

06- Spring Boot:

Maven fonde tutti i jar in un unico jar del progetto, cerca il main il quale diventa l'entrypoint del programma e ha un meccanismo di risoluzione delle dipendenze che imposta la versione di default per ciascuna libreria.

Il framework Spring mette a disposizione vari starter pack sotto forma di POM e ogni programma boot ha un singolo punto di ingresso. Il framework di Spring è basato su:

- **Inversion of Control (IoC)**
- **Dependency Injection (DI)**
- **Programmazione orientata agli aspetti (AOP)**

Definizione Cross Cutting Concern: Una qualunque funzionalità che riguarda più punti dell'applicazione.

Spring fa largo uso dei Plain Old Java Object (POJO) ovvero non obbliga ad avere una gerarchia di eredità (eg. android) o ad implementare specifiche interfacce.

- IoC: si elimina la dipendenza diretta nel codice usando le interfacce per astrarne il comportamento separando così il "cosa" (se ne occupano le classi concrete che la implementano) deve esser fatto dal "quando" (se ne occupano le classi che usano l'interfaccia).
- DI: è un pattern che **realizza l'accoppiamento tra gli oggetti in fase di esecuzione anziché in compilazione**, affidandosi ad una parte del framework appositamente delegata la quale usa elementi di natura dichiarativa (nome, tipo, annotazioni). Tutte le dipendenze vengono rappresentate da un grafo di dipendenze tra oggetti, salvate in un contenitore e successivamente iniettate nel programma principale sotto forma di proprietà degli oggetti. DI può esser configurata in via esplicita, con file xml o classi java, oppure attraverso la scoperta implicita dei beans e autowiring automatico.
- AOP: paradigma di programmazione in grado di descrivere comportamenti trasversali all'app, separandoli dal dominio applicativo (cross cuttin concern). Permette di definire funzionalità comuni a più oggetti in un unico posto (senza modificare manualmente le classi). I cross cutting concern possono essere modularizzati in speciali classi chiamate

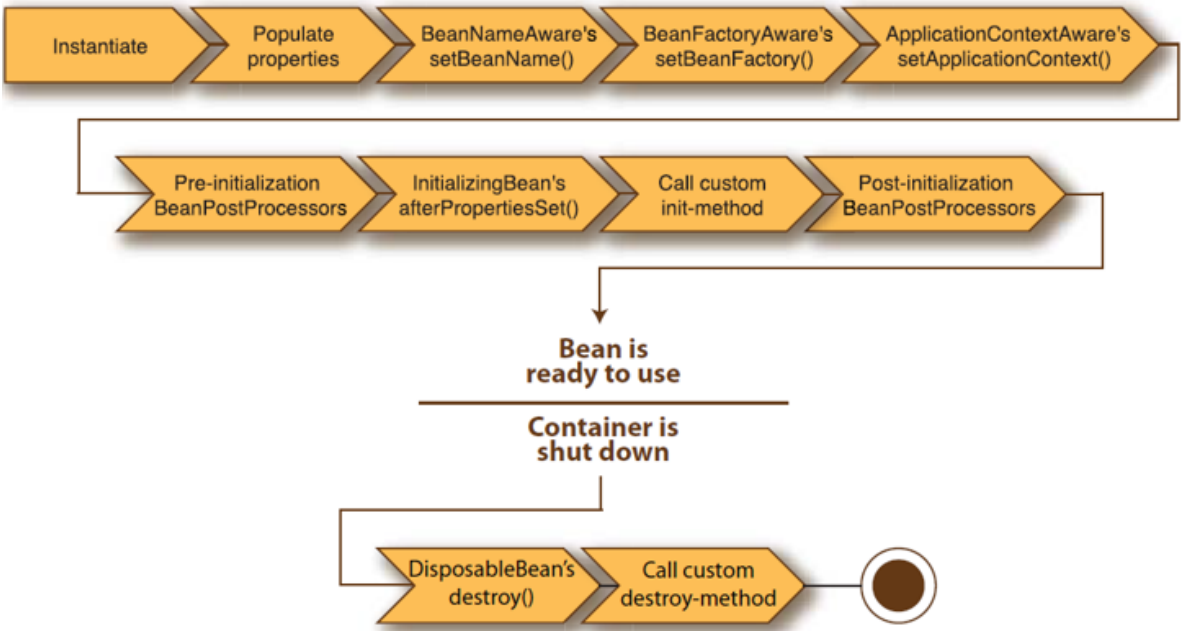
aspect evitando la ripetizione di logica per i comportamenti trasversali e rendendo più leggibili le classi che avranno solo le loro relative funzionalità primarie. Il lavoro compiuto da un aspect è chiamato **advice** (definendo what&when). Le classi scritte dal programmatore sono modificate in fase di caricamento.

In spring gli oggetti sono creati e legati tra loro da un container che si occupa della DI e dell'inserimento degli aspetti. Esistono due implementazioni di container:

- BeanFactory (adatti a scenari semplici)
- ApplicationContext (scenari complessi)

In Spring il ciclo vita di un Bean è più complesso di quello di un semplice oggetto, se una classe ha annotazioni o implementa opportune interfacce, le sue istanze potranno ricevere molteplici notifiche inerenti al procedere del loro ciclo di vita.

> (non così rilevante per l'esame a detta mia)



Autowiring è un modo per far risolvere a Spring automaticamente le dipendenze tra i bean, cercando nell'*ApplicationContext* i bean necessari.
Ogni processo Spring Boot ha associato un file di configurazione *application.properties* che viene caricato in modo implicito.

Una classe annotata con *@Controller* viene automaticamente instanziata in quanto *@Component*, Spring cerca al sui interno i metodi annotati con *@RequestMapping (@Get/PostMapping)* per identificare di quali url sono responsabili.

Se il controllore restituisce una stringa questa può indicare il nome della vista da cercare attraverso il bean *viewResolver.*

07- Spring Services Data

Lo strato di servizio è l'insieme delle funzionalità che un'applicazione offre, chiamate comunemente API. I metodi di tale interfaccia costituiscono la logica del servizio mentre i dati le astrazioni *(@Service)*.

ViewModel è un oggetto, volendo anche json per api REST, che veicola i dati da mandare alla vista mentre il **Data Trasfer Object (DTO)** modella le info che offre/consuma il servizio.
Controllore->DTO->Servizio, Servizio->DTO->Controllore->ViewModel->Vista.

Entity modella i dati così come vengono tenuti nel DB, disaccoppia lo strato di servizio da quello di persistenza.
E' solito aggiungere una funzione di utilità in grado di mappare i DTO da/verso un'entità.

Per accedere ai dati si può usare o meno il framework di Spring.

Spring Data è un progetto di alto livello per semplificare l'accesso ai meccanismi di persistenza relazionali e non. **Ci sono almeno tre metodi per usare Spring Data: JDBC o JPA per i relazionali oppure accedere direttamente ai non relazionali** con metodi differenti a seconda dell'architettura scelta.

- **JDBC** offre un accesso a basso livello, + flessibile, - automatismi.
- **JPA** si basa sul concetto di ORM (Object Relational Mapping), ogni tabella del db corrisponde ad una classe *@Entity*, ogni record ad un'instanza di tale classe. Lo strato di persistenza si occupa di gestire le operazioni astratte (save, load, find...).

DBMS non relazionali permettono di gestire dati non normalizzati, altamente scalabili ma spesso non garantiscono transazioni ACID ma solo BASE.

Repository è una terminologia di Spring per definire una classe **DAO (data access object)**, offre metodi per accedere alla tabella/collezione della base dati. **Se ne usufruisce definendo un'interfaccia** che estende *org.springsource.data.repository.Repository<T,Id>* o una sua sottointerfaccia. Spring provvede a **generare automaticamente** un'implementazione di un'interfaccia che estende Repository.

Un repository identifica la base dati soggiacente tramite un bean di tipo *javax.sql.DataSource*.

Per accedere direttamente ai dati occorre configurare un'istanza di *SessionFactory* (*EntityManagerFactory* per JPA) da iniettare nei componenti che la necessitano.
EntityManagerFactory è un oggetto che media il processo di ottenere connessioni con una specifica base dati (più basi dati più *EntityManagerFactory*).

Transaction Manager ha il compito di iniziare e chiudere una transazione attorno ai metodi degli oggetti etichettati con *@Transactional*, eseguendo un roll back in caso di eccezione.

08- JDBC

Java Data Base Connectivity ha un basso livello di astrazione avendo accesso a tutte le funzionalità del DBMS, di conseguenza il programmatore si deve occupare dei dettagli.

E' un'architettura aperta, indipendente dal db e dalla piattaforma.

Java App->JDBC->Driver->DBMS.

I Driver JDBC sono forniti dal produttore di db sotto forma di jar e sono il punto di contatto col dbms.

Il DriverManager si occupa di caricare il driver e inizializzare le connessioni, il db è rappresentato da una url e viene connesso usanto i metodi statici getConnection(DbUrl). Le url sono solitamente *jdbc:subprotocols:source*.

Ciascun driver ha il proprio sottoprotocollo il quale ha la propri sintassi. **I driver vengono registrati una sola volta.**

Per eseguire un'interrogazione: Registrazione driver (una sola volta)->connessione->interrogazione->lettura risultati->rilascio risorse.
L'oggetto che contiene i risultati è il ResultSet, i dati sono organizzati in forma tabulare, al suo interno vi è un cursore che punta al record corrente. Per dati grossi Blob e Clob (binary/Char Large Object).
E' possibile associare una classe ad ogni tabella del db definendola come DAO (Data access Object) i cui metodi offrono le funzionalità CRUD.

Transazioni: autocommit a false e poi commit.

09- Hibernate (e JPA)

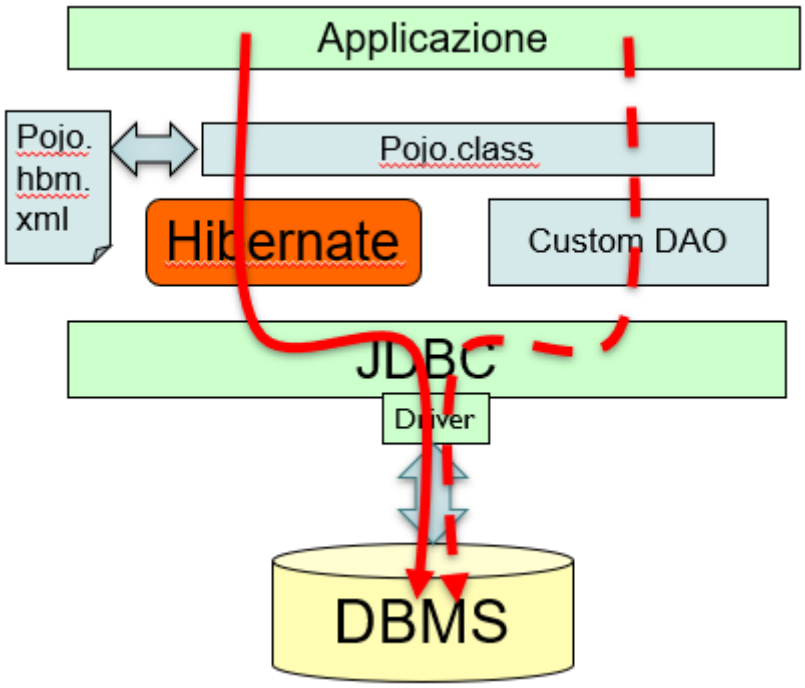
Obiettivo: associare in modo automatizzato e generalizzato classi a tabelle, oggetti a tuple, utilizzando un **ORM (Object-Relational Mapping)**.

E' l'ORM che genera internamente il codice boilerplate per accedere alle varie basi di dati, il loro utilizzo può però portare alla creazione di db mal progettati.

Per Java c'è JPA o Java Persistent API la cui implementazione più nota è il framework Hibernate.

Hibernate non richiede l'uso di un contenitore ed è adatto sia per app stand-alone che per app web, leggero e open-source ad elevate prestazioni dbms-indipendent.

App->Pojo->Hibernate (o Custom DAO)->JDBC->DBMS.



In Hibernate ogni classe entità viene preceduta da *@Entity*, *@id* per la chiave, *@ManyToMany*...

Le relazioni sono modellate tramite puntatori, per gli aggiornamenti in casscata aggiungere *@Cascade*.

Hibernate ha un suo sistema di **sessione** attraverso il quale compie le query e tiene traccia degli oggetti java, questi ultimi in una data sessione possono assumere **tre stati**:

- Transient (appena creati)
- Persistent (associato ad una sessione, le modifiche verranno propagate al db)
- Detached (dopo la chiusura della transazione perde il legame col contesto di persistenza, lo si può rilegare a necessità).

Hibernate utilzza un suo linguaggio per le query chiamato **HQL (Hibernate Query Language)**, simile a sql ma opera su classi java e non su tabelle.

JPA è l'interfaccia standard per rendere persistenti oggetti tramite ORM, richiede un'implementazione concreta (eg Hibernate). Per generalità JPA definisce l'interaccia *EntityManager*, costruita a partire da un *EntityManagerFactory*, che equivale ad una session di Hibernate.

10- MongoDB

I **db NoSQL** consentono l'inserimento e la manipolazione dei dati senza uno schema predefinito, supportano nativamente lo sharding (suddividere dati tra server in modo trasparente). Offrono elevate prestazioni e abbandonano il linguaggio sql in favore di più semplici api.

Esistono vari tipi di db nosql tra cui: Document Store (come mongo), key-value Store, Column-Oriented, Graph Store.

Per il **teorema CAP** è impossibile garantire contemporaneamente Consistency, Availability e Partition Tolerance di conseguenza si giunge ad un compromesso.

Molti DB NoSQL seguono le proprietà BASE anziché garantire CAP fornendo così la possibilità, a chi li utilizza, di avere un'architettura capace di **scalare orizzontalmente** in contrapposizione alle già esistenti ACID:

- Basic Available: Indica la capacità di un sistema di garantire l'availabilty nei termini del teorema CAP.
- Soft State: Indica che lo stato del sistema può cambiare nel tempo anche senza input, questo a causa del modello di eventually consistency.
- **Eventually Consistent**: Il sistema diventerà consistente secondo il teorema CAP, cioè tutti i nodi replica avranno la stessa copia dei dati, in un certo intervallo di tempo durante il quale il sistema non riceverà input dall'esterno.

MongoDB è un scalabile, ad alte prestazioni, open-source, privo di schema, document-oriented database.

I dati sono memorizzati Json-like, collection->document.
Il modello dei dati può essere normalizzato o referenziato (Eg. id dentro un doc che si riferisce ad un altro doc, sql like) oppure denormalizzati o embedded document (dati annidati).

La transazionalità viene supportata solo a livello di casual consistency.

Esistono vari modelli di consistenza per le transazioni:

- Eventually consistency
- Casual consistency
- Sequential consistency
- Strict consistency

More info @ Wikipedia: modelli consistenza

11- Spatial

DB per info di spazio. (Eg. geoPoint). I dati spaziali possono memorizzare dati geometrici come punti, linee, poligoni... e si riferiscono ad un piano cartesiano in due o tre dimensioni.

Vari db si sono adattati per permettere l'uso di query su poligoni e coordinate geografiche 2d/2dSphere (geometrici/geografici) come PostGis (postgres) e mongodb, nel secondo caso si possono fare query specifiche (anche di intersezioni) utilizzando i radianti come unità di misura per gli operatori di distanza.

I dati geo-spaziali in mongo vengono salvati come oggetti GeoJson o come coppie di coordinate legate tra esse.

12- RestSpring

Representational state transfer.
L'evoluzione di internet impone la realizzazione di sistemi distribuiti e scalabili.

Le proprietà fondamentali da rispettare sarebbero:

- Eterogeneità, capacità di far interoperare dispositivi diversi e os diversi
- Scalabilità
- Possibilità di evolvere, non introdurre vincoli strutturali che impediscano la manutenzione evolutiva del servizio
- Affidabilità/resilienza, mantenere il servizio anche in caso di malfunzionamenti locali
- Efficienza, misura risorse su valore risultato
- Prestazioni, tempo su risorse necessarie

Il modello REST è uno stile con il quale realizzare architetture distribuite di tipo client-server stateless.

Ogni server ospita uno o più servizi i quali gestiscono un'insieme di informazioni ciascuna delle quali ha una propria uri, rappresentate da testo/json/xml.. e supportanti operazioni di tipo CRUD.

Poiché ogni entità ha un nome univoco è possibile legare entità differenti specificando il nome della relazione. **Http** è un ambiente naturale per le richieste di tipo CRUD di questo genere e si specifica usando Content-type il formato dei dati scambiati.

Per misurare il grado di maturità di un progetto rispetto ai principi generali dell'architettura REST viene introdotta **la scala Richardson**:

- Livello 0, http come protocollo trasporto, tutte le richieste ad una singola url
- Lvl 1 Risorse, ogni entità dispone di una propria url
- Lvl 2 Uso dei metodi/stati http, i comandi da eseguire vengono mappati sui metodi http, esiti tramite codici di stato (200,404,..)
- Lvl 3 Collegamenti ipermediali, per ogni oggetto trasferito il server include dei collegamenti uri che indicano le azioni che è possibile compiere sull'oggetto stesso.

I microservizi sono un approccio architetturale per la realizzazione di applicazioni distribuite basate su un insieme di componenti lato server indipendenti tra loro, ogni componente viene eseguito nel proprio processo/container/macchina e coopera con gli altri utilizzando protocolli di rete. Quello che distingue l'architettura basata su microservizi dagli approcci monolitici tradizionali è la suddivisione dell'app nelle sue funzioni di base. **Ciascuna funzione, denominata servizio, può esser messo in campo indipendentemente l'una dall'altra** e facilita la scalabilità e manutenibilità del sistema, contenendo al suo interno tutti gli

strati sw che necessita (db, UI, logica server..).
Essendo REST stateless ogni oggetto trasferito porta con sé il proprio stato, il che facilita la scalabilità orizzontale non essendo legati ad una sessione. L'identità del client viene esplicitamente passata quindi ad ogni richiesta via basic AutheN, HMAC o OAUTH2 (token del server).

In spring tutti i metodi definiti in una classe annotata con *@RestController* possono diventare endpoint di un servizio.

13- Spring Security

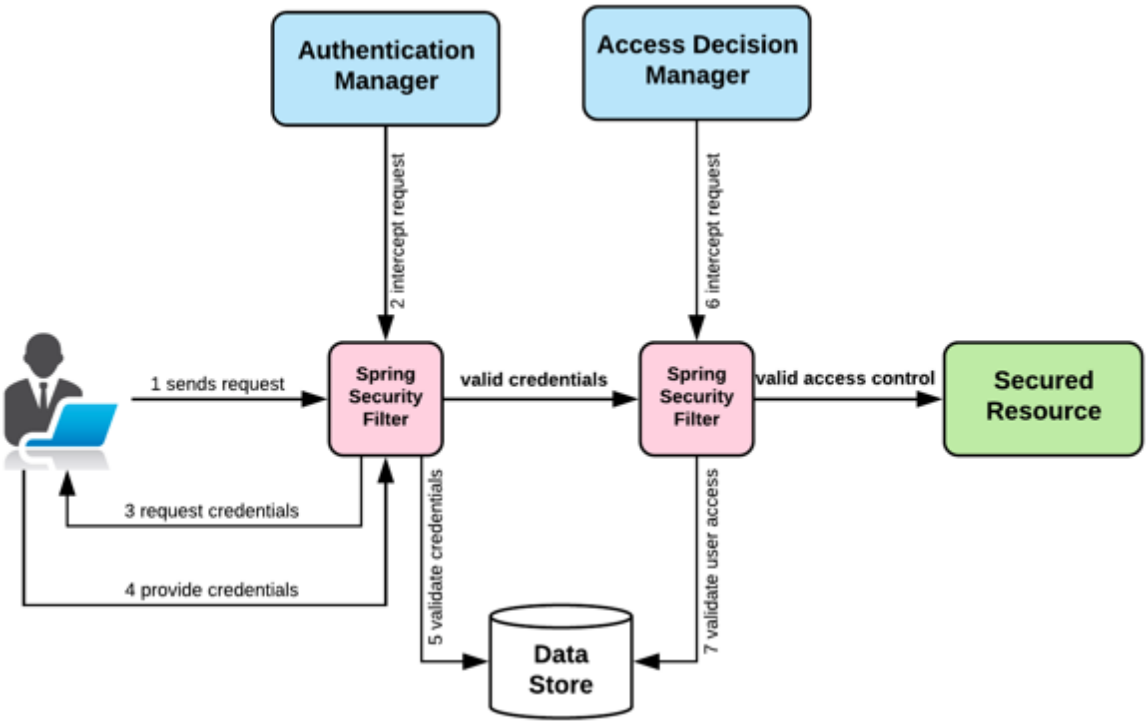
Sebbene sia possibile inserire a livello applicativo vincoli di sicurezza non è una buona idea, la sicurezza deve essere implementata in modo dichiarativo.

Spring Security implementa la sicurezza dichiarativa a livello web tramite filtri e a livello di invocazione dei metodi tramite "aspetti" dinamici che specificano chi ha diritto di accedere.

La gestione di chi può fare cosa implica la gestione di un db degli utenti e la presenza di un meccanismo di authZ.
Avere a disposizione una lista di utenti con i relativi ruoli è il primo passo per abilitare il controllo degli accessi.

Con Spring è possibile esprimere restrizioni in due modi, a livello url o a livello dei singoli metodi di Bean (tipicamente servizi).

Durante il tentativo di accesso **il client prima deve authN** (gestita dall'AuthN manager) **e poi authZ** (gestita dall'access decision manager) **attraverso i filtri di Spring security.** Per il controllo via url si usa *HttpSecurity* per proteggere le url e definire le regole da applicare.



Le funzionalità di Spring security si prestano particolarmente ad essere inserite a livello di servizio (è possibile aggiungere annotazioni a supporto dei singoli metodi).

Json Web Token (JWT) è un meccanismo di autorizzazione per l'accesso ad API web, prevede che il client invii nella propria richiesta una intestazione formata da tre parti separate da un '.': header, payload e signature.
JWT è tipicamente codificato in Base64 al fine di semplificarne la manipolazione.

14- Architetture (principi)

Un'architettura sw è l'insieme delle strutture di alto livello che formano un sistema sw e delle relazioni implicite ed esplicite che intercorrono tra loro. E' la conseguenza di scelte guidate dai casi d'uso e dai requisiti di business legati alla valorizzazione dell'artefatto prodotto.

Una data architettura è il risultato dell'interazione di molteplici persone e sensibilità che concorrono a definirne la struttura e le relazioni, la conoscenza condivisa tra tutti i membri di un gruppo permette di ridurre il gap tecnico dei singoli ma affiché tale evoluzione possa avvenire occorre che il sistema realizzato sia suddivisibile in componenti e che sia questi che i gruppi godano di due proprietà fondamentali: **basso accoppiamento** (di funzionalità/specialità) ed **alta coesione**.

L'accoppiamento è il grado di dipendenza di un componente dai dettagli implementativi di un altro, il processo di refactoring è normalmente finalizzato a ridurre l'accoppiamento di un componente ridefinendone la struttura interna mantenendone però l'interfaccia.

La coesione è il grado di focalizzazione di un componente sw su un dato compito, tutte le parti che lo costituiscono dovrebbero supportare un unico scopo fondamentale. Più alta è la coesione più le funzionalità che un componente contiene saranno altamente correlate tra loro e non ulteriormente separabili.

Qualità logiche di un'architettura:

- Estensibilità, facilità di modificare le specifiche
- Testabilità, facilità di mettere alla prova i componenti
- Comprensibilità, sviluppatore standard comprende
- Resilienza, capacità di restare in funzione nonostante la presenza di malfunzionamenti
- Sicurezza, grado di difficoltà a violare i meccanismi di protezione

Qualità operative:

- Scalabilità
- Manutenibilità
- Installabilità, facilità di deploy
- Usabilità, grado di efficacia/soddisfazione/efficenza
- Disponibilità, %tempo sistema è operativo

Le 4 dimensioni di scelta nel progetto delle architetture sw sono:

- Funzionalità, dominio operativo e requisiti di business
- Dati, quali/quanti, da dove e come conservarli
- Tecnologia, quali linguaggi, framework, os
- Operatività, come mappare moduli sul server, come comunicano, come si accede

La disponibilità di piattaforme cloud come hosting per app web ha portato alla definizione di **12 fattori** (principi architetturali):

1. Codebase, un'unica base di codice
2. **Dipendenze, devono esser dichiarate esplicitamente ed isolate dal resto del codice**

3. Configurazione, la configurazione di un'app deve esser presente in ambiente di esecuzione e non dentro dei sorgenti
4. Servizi di supporto, i servizi devono esser trattati come risorse dell'applicazione (eg dbms).
5. **Costruzione/rilascio/esecuzione, le tre fasi del ciclo vita del sw devono esser ben separate.**
6. Processi, l'app viene eseguita come uno o più processi privi di stato.
7. Collegamento delle porte, i servizi offerti vengono esposti su apposite porte di rete.
8. **Processi, si gestisce la scalabilità di un'app instanziando più processi della stessa.**
9. Terminabilità, i processi devono poter esser avviati velocemente e fermati senza problemi.
10. Parità tra sviluppo e produzione, gli ambienti di sviluppo prova e produzione dovrebbero essere il più simili possibile tra loro.
11. Log, vanno trattati come flussi di eventi e non gestiti direttamente dal codice sorgente.
12. Processi di amministrazione, i compiti amministrativi vanno eseguiti come processi una tantum (eg. popolare il db).

Un sistema di integrazione continua (CI) provvede ad estrarre dal repository una copia del codice sorgente ogni qualvolta viene inserito un nuovo contributo (eg. Jenkins), compilandolo e testandolo in modo automatizzato.

15- Applicazioni web in tempo reale

Nel caso ci sia bisogno di aggiornamenti in tempo reale da parte del server storicamente la comunicazione viene iniziata lato client utilizzando vari meccanismi quali:

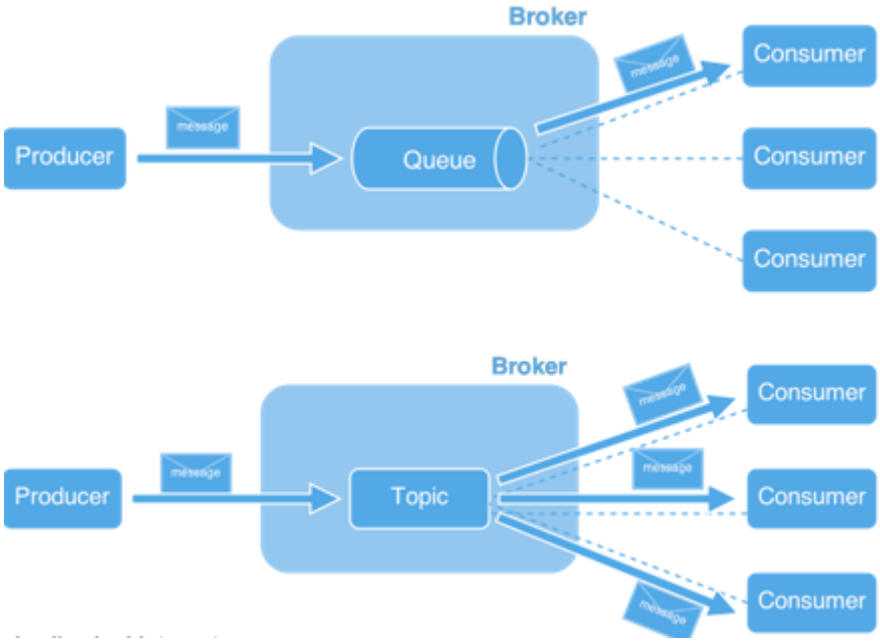
- Polling, non scala al crescere degli utenti
- Long polling (comet), risposta server ritardata
- Streaming di contenuti, simile a long polling ma basato su codifica di trasferimento chunked
- WebSocket, protocollo bidirezionale asincrono

Long polling e streaming obbligano il thread server a rimanere aperto finché non viene conclusa la richiesta ma i server web utilizzano per lo più un meccanismo di concorrenza per gestire molte richieste ma brevi.

Utilizzando i **servlet asincroni** all'arrivo di una richiesta non soddisfabile atm può esser incapsulata in un *asyncContext* e soddisfatta in futuro senza bloccare il thread. Per gestire richieste asincrone il server deve implementare un meccanismo per organizzare e distribuire i messaggi. **Il problema** è che essendo le richieste asincrone possibilmente prese in carico da thread differenti devono contenere al loro interno tutte le informazioni necessarie al loro svolgimento riducendo l'efficienza.

Il webSocket utilizza una connessione permanente, volendo anche crittografata con TLS, ottimizzando così le risorse di rete. La sessione di lavoro di un webSocket modella il canale di comunicazione e tiene traccia degli eventi legati al ciclovita. Gli oggetti di classe *Session* offrono anche accesso a tutte le sessioni attualmente in corso e quindi comunicare con tutti i client connessi.

E' possibile comunicare messaggi semplici o complessi (Json) ma se richiesto si può utilizzare un **broker** per i messaggi a più client. Un broker utilizza il protocollo *STOMP* (Simple Text oriented message protocol).

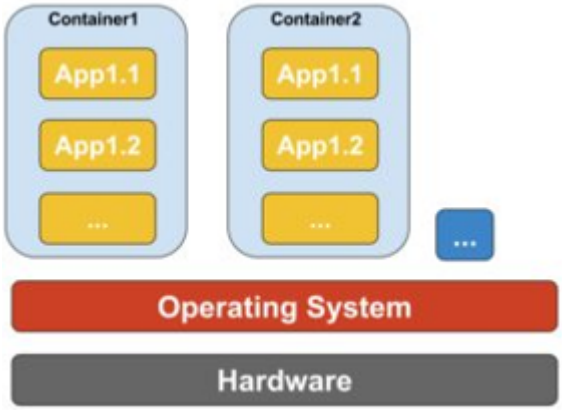


16- Docker

La creazione di servizi distribuiti richiede la progettazione e messa in campo (deploy) di vari componenti sw i quali necessitano di specifiche configurazioni di rete e possono essere ospitati su differenti macchine fisiche.
Al fine di affrontare queste sfide si procede in due direzioni: virtualizzazione oppure l'utilizzo di container (come docker).

La **virtualizzazione** consente l'ottimizzazione delle risorse hw tra diverse configurazioni sw indipendenti tra loro (Eg. Hypervisor).

I container si basano sulla virtualizzazione sw condividendo quindi il kernel con la macchina host garantendo però lo stesso livello di isolamento dei processi e delle risorse di hypervisor. Alcontempo richiedono meno risorse di una virtual machine non dovendo emulare l'hw ed eseguire un intero OS.

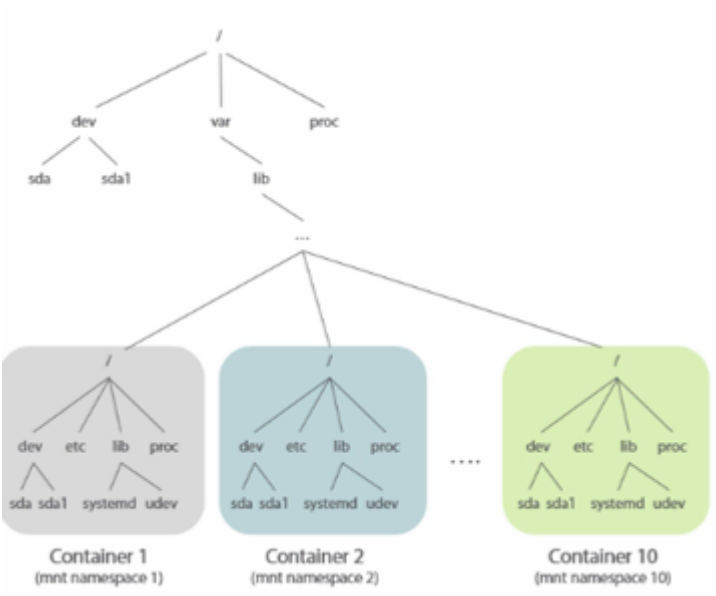


Dal punto di vista dell'os i container sono costituiti da namespace distinti ed isolati, un namespace è un insieme di processi che accedono a risorse kernel virtualizzate. Il filesystem di ciascun contenitore può essere indipendente da quello degli altri, risolvendo così il problema dello stesso nome logico in contenuti diversi (dramma lib condivise con versioni diverse).

Docker è una tecnologia open source per la costruzione, spostamento e rilascio di app basate su container. I suoi **4 componenti principali** sono:

- Engine, è il core: un processo demone in background sulla macchina ospitante.
- Client, interfaccia per le API esposte all'engine, mette a disposizione una serie di comandi (CLI), ogni comando inizia con la parola chiave docker.
- Image, un'immagine è un insieme di file e parametri che definisce e configura un'app da usare a runtime. **Essa non ha uno stato ed è IMMUTABILE.**
- Container, è un'istanza in esecuzione di un'immagine in cui il file system è costruito da **uno strado R/W sovrapposto agli strati immutabili dell'immagine.**

Union mounts è il meccanismo attraverso il quale l'os riesce a montare diversi filesystem uno sull'altro combinando gli strati in un unico fs, se un dato file è presente in diversi layer quello più in alto nello stack nasconderà gli altri.



Le immagini per creare i container vengono scaricate/salvate all'interno di repository che a loro volta vivono all'interno dei **registry**. Ogni docker host ha un proprio registry in locale, in caso si cerchi di runnare un'immagine non presente in locale docker proverà a cercarla in un registry remoto. Il registry ufficiale di docker è chiamato docker hub.

Il modo più veloce per condividere una parte di fs con l'host è utilizzare il concetto di **volume**: costituito da una cartella presente nel fs dell'host che viene mappata nel fs del contenitore, le operazioni sui volumi vengono interamente controllate dal driver dell'host. **Le info contenute in un volume sono persistenti.**

E' possibile costruire una nuova immagine descrivendola in un **dockerfile**, la cui posizione deve essere nella cartella root del suo contesto (esplora solo le sottocartelle e la sua cwd).

Sebbene l'intero stack di rete sia accessibile ad ogni contenitore, l'esistenza del corrispettivo namespace rende tale stack inaccessibile dall'esterno.

Di default ogni container è abilitato ad aprire connessioni verso l'esterno, per aprirle verso un'altro container si può configurare una connessione diretta oppure configurarli come appartenenti a sottoreti virtuali.

All'installazione di docker verrano create tre reti:

- bridge, (default) interfaccia docker0 presente in tutti i docker host. I container di questa interfaccia potranno comunicare tra loro utilizzando l'ip assegnato loro dalla rete (opzione "--link" per utilizzare il nome del container al posto dell'ip).
- none, i container di questa rete non avranno altra interfaccia fuorché la loopback.
- host, i container di questa rete vedranno l'intero stack di rete visto dal docker host.

E' possibile creare le proprie configurazioni di rete per un miglior isolamento dei docker, utilizzando vari driver di rete. Un container può appartenere a più reti.

Docker Swarm è la clusterizzazione di una rete docker su host diversi creando una rete overlay tra gli host.

17- Docker compose

E' un tool per definire ed eseguire applicazioni complesse, formate da più eseguibili, con docker. E' possibile definire applicazioni multicontainer con un singolo file **yaml** (yet another markup language) o yml.

Al fine di utilizzare il compose sono necessari tre passi:

1. Definire i dockerfile dei servizi che fanno parte dell'app
2. Definire i servizi che devono essere avviati in un file chiamato docker-compose.yml
3. Eseguire il comando **docker-compose up**

Tramite compose è possibile gestire l'intero ciclo vita del complesso di applicazioni. E' possibile esprimere le dipendenze tra i servizi che quindi saranno eseguiti in ordine.

18- Spring e Docker

Per dockerizzare le applicazioni spring esistono due modalità:

- manuale, generazione jar->creazione dockerfile-> build dell'immagine -> run del container.
- automatica, utilizzo di plugin per maven o gradle.

SERVETTI:

01- Javascript advanced

Javascript è un linguaggio di scripting ad oggetti utilizzato principalmente nei browser web per le pagine dinamiche e l'interazione con gli utenti.

02- OOP with JS and Typescript

In **java** le classi sono template per gli oggetti, un oggetto è un'istanza di una classe con le proprietà ed i metodi della classe definiti e fissi. Dopo l'istanziazione dell'oggetto non ha assolutamente nessuna relazione attiva con la classe parente.

In **Javascript** (che è un linguaggio basato su prototipo) instanziare una classe crea un nuovo oggetto linkato al prototipo, i cambiamenti al prototipo vengono propagati all'oggetto anche dopo l'instanziazione.

Typescript è un superset di Javascript (o ECMAScript) con **optional static typing**. Il suo scopo principale è rendere più rigido js per applicazioni complesse utilizzando l'"optional" type system, ovvero controlli durante la compilazione anziché solo a runtime. Il codice typescript viene transpilato (Transpiles) a Javascript usando il **tsc compiler**.

03- Intro to Single Page Web Applications

Web applications are client-server-db app where the client sw is downloaded to the client machine (browser) when visiting the relevant web page.

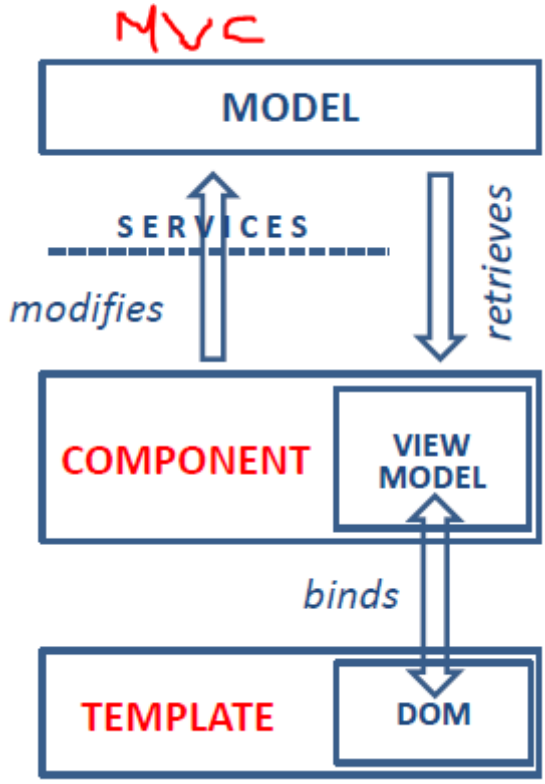
It comes with the **benefits** of cross-platform compatibility, easy to deploy and update, well known user interface and a rich interactive enviroments (html5/ajax).
It comes with some **drawbacks** as are browser based and not native (limited access to OS functionalities), network and server dependent.

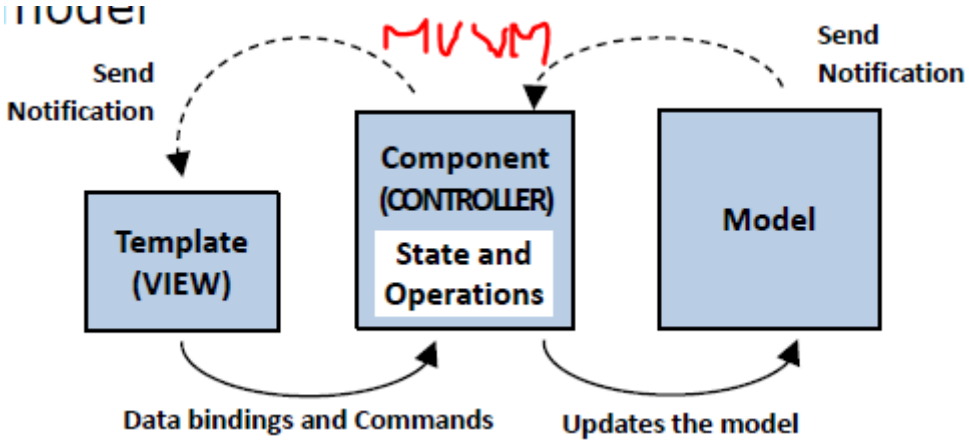
Single page application (SPA) may have an architecture that focus rendering on server or client side. Resources are dynamically loaded and added to the page when needed. **SPAs instead of changing the page, change a view inside the same page**. When a view is needed it's not loaded from the server but generated in the client with additional data/content (JSON) requested from the server-side rest api. Typical SPA consists of small pieces of interface representing logical entities, components, all of which have their own UI, business logig and data.

Model-View-Controller in server-side implementation every URL has its own controller that manipulates the model according to the request through services. The controllore selects the view that reads the data from the model and formats it accordingly.

Model-View-Controller in client-side has a domain model (persistence) retrieved from the server that exposes a RESTful API, the UI (view) requires a tighter access to the domain model for user interaction and the controller exposes to the view (binds) a subset of the domain model called view-model. **ViewModel** is a binder that binds data between the view and the model.

In Angular il controllore del MVC diventa il component, la vista il template. Angular permette di strutturare il codice secondo le preferenze del programmatore sia in MVC (maggior separazione tra vista e logica) sia MVVM (il modello legge i cambiamenti della vista in modo autonomo e si aggiorna), il termine utilizzato per indicare questa possibilità in un framework è Model-View-Whatever (MVW). From MVC to MVVM (ModelView-ViewModel): il *ViewModel* non è un controllore ma lega i dati tra vista e modello al fine di farli comunicare l'un con l'altro.





Components are UI fundamental building blocks that should handle data transformations and controls passing data between template and model, and viceversa, without modifies to the domain model.
A component is a sw element that encapsulate a set of related functions (and/or data). All of the data and functions inside a component are semantically related (as with classes). Components communicate via interfaces. **A component delegates everything nonTrivial to services.**

An SPA is nothing more than a tree of components.

Web components: A collection of specifications that enable developers to create their web applications as a set of reusable components. Each component lives in its self-defined encapsulated unit with corresponding style and behavior logic. These components can not only be shared across a single web application but can also be distributed on the web for use by others.

To obtain self-contained style and html into a component a new spec was needed to modify the DOM so w3c introduced **shadow DOM** that is a scoped DOM for the web platform.

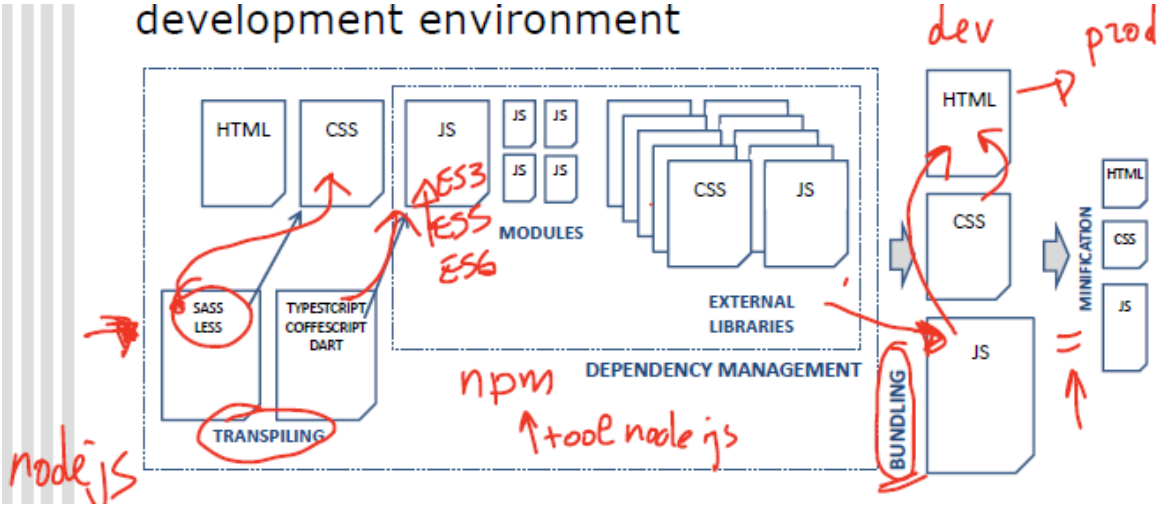
04- JavaScript development enviroment

Una web app ha una struttura di developing divisa in varie fasi, essendo la loro complessità incrementata esponenzialmente.

L'html contiene anche il template della pagina, il css oltre allo stile possiede anche un linguaggio più espressivo (SASS) mentre il js è transpilato da altri linguaggi (typescript). I browser non capiscono sass, typescript o ES2015 quindi questi file devono esser compilati a css e js (source to source).

Angular ha una sua **CLI**, spesso si basa sull'enviroment offerto da nodeJs il quale ha un suo packet manager (npm) attraverso cui si può scaricare svariati moduli, il registry di npm è pubblico -> gigante.

Sloppy coding can lead to hard-to-find bugs --> Linting, usato per controllare le convenzioni di programmazione.

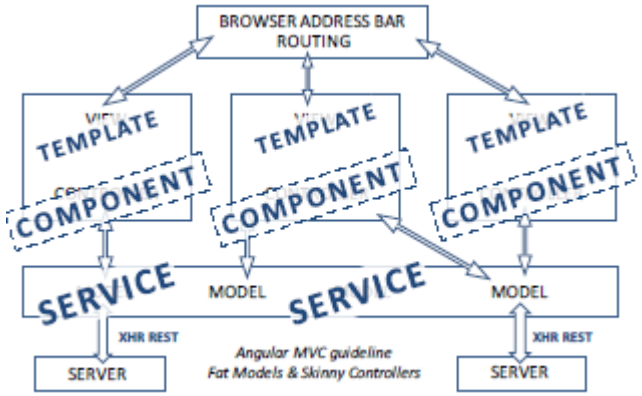


Il **bundling** può essere dinamico o statico, se dinamico è il browser che esegue le varie import al volo, se statico inveve viene fatta precedentemente la combinazione di tutto il codice e varie dipendenze annesse in un singolo file (webpack).
La **minification**, una volta fatto il bundling, rimuove tutti i caratteri non necessari riducendo le dimensioni dei file da inviare ed al contempo offusca il codice e lo rende indebuggabile.
Al fine di debuggare il codice in produzione esistono le source maps che mappano il sorgente originale a quello minimifato.

05- Angular framework (single page app architecture)

Web app frontend complesse necessitano di solidi framework di sviluppo, con Angular si può fare routing, REST, data binding, dependency injection ed estendere gli html per combinare presentazione e logica.

Angular ha una sua CLI attraverso cui si possono scaricare moduli e pacchetti.
Angular Schema:



In Angular l'entry point dell'app è il componente **app-root**. Al bootstrapping il file di configurazione Angular.json specifica il main.ts da cui lanciare l'app attraverso un modulo chiamato appModule. **L'AppModule** specifica quale/i componente/i usare come top-level.

I Decorators (metadata) dicono ad Angular come processare una classe (annotazioni).

Modules sono contenitori ove risiedono codice, componenti, direttive... in un blocco coesivo di funzionalità.

Tra i decoratori (annotazioni) è importante *@NgModule* che permette di definire:

- Declarations, quali componenti sono definiti nel modulo
- Imports, quali dipendenze ha il modulo
- Providers, i servizi di cui necessita attraverso la dependency injection
- Bootstrap, il componente root usato per il bootstrapping dell'app .

I component hanno la logica di controller e data dentro la classe typescript *{componentName}.component.ts* mentre la vista nel template html *{name}.component.html* ed il css *{name}.component.css*(o scss).

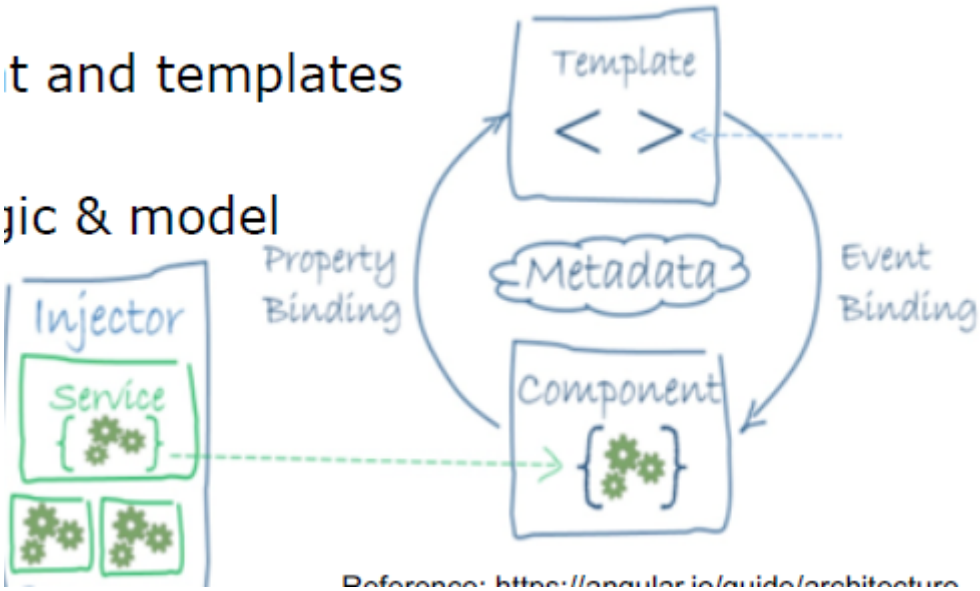
Il ruolo del template è di presentare i dati all'utente, può contenere della logica ma dovrebbe rimanere semplice e ridotta al più a chiamate a funzione.

La classe component controller (.ts) interagisce col template attraverso delle api delle proprietà e dei metodi. Le proprietà rappresentano i dati che il template presenterà mentre i metodi le azioni che attiverà (ma che saranno eseguite dal component).

Il component di default è chiamato AppComponent.

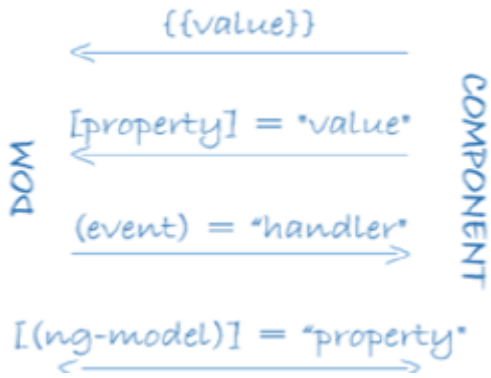
I component sono creati a distrutti man mano che l'utente si muove attraverso le viste dell'app, l'app può fare azioni ad ogni momento del **lifecycle** attraverso l'utilizzo di chiamate specifiche (eg OnInit).

Nel css tradizionale le proprietà hanno scope "system wide" ma in Angular è ristretto a quel component, **i cambiamenti di stile da altre parti non affliggono lo stile del singolo componente.**



Il **Data Binding** permette di legare i dati tra componente e template. Ha 4 forme e due direzioni:

- Interpolation `{{ < expression > }}` DOM<---Component
- Property binding `[ < property > ]` DOM<---Component
- Eventi binding `( < event > )` DOM--->Component
- Two-way binding `[( < > )]` DOM<--->Component



Flusso unidirezionale: property (R),event (W)
Le espressioni vengono prima valutate nel contesto del componente quindi convertite a stringhe.
Interpolation ritorna sempre una stringa mentre il property binding qualsiasi tipo vogliamo.

Oss: Angular processa tutti i data binding una volta per js event cycle, dalla root dell'albero dell'app component fino a tutti i component figli.

La **change detection** è il meccanismo per tener traccia dei cambiamenti dello stato dell'app e può esser fatta in **single pass dirty checking** (check per l'albero dei componenti tenendo traccia del valore vecchio e poi confronto) o **smart checking** (utilizzando immutables e observables).

Nel *property binding* essendo unidirezionale non ci sono side effects, si potrebbe usare un getter ma **i metodi non dovrebbero cambiare i valori che sono bindati alla view.**

Nell'*event binding* i side effects sono voluti, si usano gli eventi per aggiornare lo stato dell'app dalle azioni dell'utente, solitamente non si passa tutto l'evento ma solo i valori necessari.

Anche il template può avere le sue variabili non condivise col componente di due tipi, globali se precedute da hashtag oppure locali al contesto se precedute da "let".

Esistono varie direttive built-in che possono semplificare la sintassi (eg. *ngClass* o **ngFor*), la

direttiva *NgModel* è un bridge che abilita il two-way binding per gli elementi form.

OSS: Angular rifà il rendering degli elementi solo se modificati ma se è una collection, ed è stata rigenerata, tutti gli oggetti verranno resi di nuovo. Per evitarlo si usa la direttiva trackBy (funzione).

Servizi:

> *Services are objects that provide common functionability to support other building blocks in an application.*

I servizi implementano la business logic dell'app, modificandone lo stato (model) e sincronizzando i dati col backend. I components sono i grandi consumatori di servizi.

In Angular gli oggetti vengono creati quando necessario e distrutti quando non più usati. Gli agganci al **lifecycle** sono varie funzioni tra cui OnInit, OnDestroy, OnChanges, DoCheck.

Angular non crea i servizi ogni qualvolta un componente lo richiede ma essi sono creati dai providers. Il componente cerca di risolvere la dipendenza usando la sua gerarchia di providers. Se un provider viene definito nel modulo root tutte le dipendenze per un token all'interno dell'app saranno risolte utilizzando lo stesso oggetto (singleton).

Quel che differenzia un servizio da un normale oggetto è che sono gestiti dalla **dependency injection**, sono ***provided/injected** to building block** da un provider esterno.

Un servizio viene reso disponibile ad un component utilizzando la provision e la injection. La provision dichiara il servizio come dipendenza utilizzando il costruttore che poi verrà risolto utilizzando i provider dichiarati nell'AppModule. L'injection utilizza il costruttore per la dependency injection e l'ngOnInit per lanciarla.

06- Angular Material

Angular Material is a UI component library for Angular JS developers. Angular Material components help in constructing attractive, consistent, and functional web pages and web applications while adhering to modern web design principles like browser portability, device independence, and graceful degradation. It helps in creating faster, beautiful, and responsive websites. It is inspired by the Google Material Design.

07- Angular HttpService

Le chiamate http get ad un server remoto sono asincrone e non se ne può aspettare il risultato in modo sincrono (freeze). La get deve avere una signature asincrona di qualche tipo: promise/callback/observable.

HTTPClient è un service che offre alle app Angular le api http, è basato sulle api observable di RxJS.

La *httpclient.get* ritorna il body della risposta come un untyped json object di default.

Gli **observable** sono degli oggetti che non contengono il risultato immediatamente ma lo conterranno nel futuro, per definirli si aggiunge il *$* al nome della variabile del component e *variabile$* | *async* quando la si usa nel template.

La **Async pipe** si preoccupa di fare la subscribe, dell'aprire i dati e dell'unsubscribe quando il componente viene distrutto (ogni volta che viene chiamata | *async* viene creata una subscribe).

Quando non si può usare la pipe si usa la subscribe all'observable nel component per prendere l'hard value quando sarà disponibile, è possibile definire una callback col dato quando è pronto.

Finché non viene compiuta la unsubscribe viene mantenuto un reference all'observable e se si perde si crea un memory leak. HttpClient unsubscribes asap it gets the response or an error occurs, è importante farlo per observable streams che emettono più valori (eg. eventi). La unsubscribe può esser fatta nell'OnDestroy.

E' più sicuro usare httpclient poiché definendo il tipo della risposta puoi permettere al compilatore di controllarla. La gestione degli errori della subscribe può esser fatta nella callback ma andrebbe gestita nel servizio.

La libreria RxJS mette a disposizione vari operatori per gestire gli observable, *pipe()* ad esempio permette di combinare varie funzioni che poi vengono eseguite in sequenza. La gestione degli errori non può esser fatta coi try/catch --> *pipe(catchError(this.handleError))*, l'operatore catchError passa l'errore all'handler il quale ritorna un errorObservable. L'operatore ***retry*** fa il re-subscribing al risultato di un httpClient rieseguendola.

Http Interceptors sono usati per modificare le http request e responses ai fini di autenticazione, caching, logging... La catena di interceptors segue l'ordine stabilito dal programmatore (req A>B>C, resp C<B<A), si usa il comando *next()* per passare all'interceptor successivo. Solitamente gli interceptors sono provided nella root injector (*AppModule*) e devono implementare l'interfaccia *httpInterceptor*. Trasformano la richiesta in ***httpEvent*** così da poter emettere più di un observable (eg. upload download progress tracking). **Le httpRequest e response sono quasi interamente read only, non devono essere cambiate poiché il retry e la gestione degli errori devono essere lanciate sulla stessa richiesta. Per alterare la request si clona e si altera la copia.** Si utilizza un interceptor per impostare il token di autorizzazione nell'header.

08- JavaScript Asynchronous programming

> Callbacks -> annidate
>
> Promise -> chaining
>
> Observable -> promise ma estese a sequenza di eventi

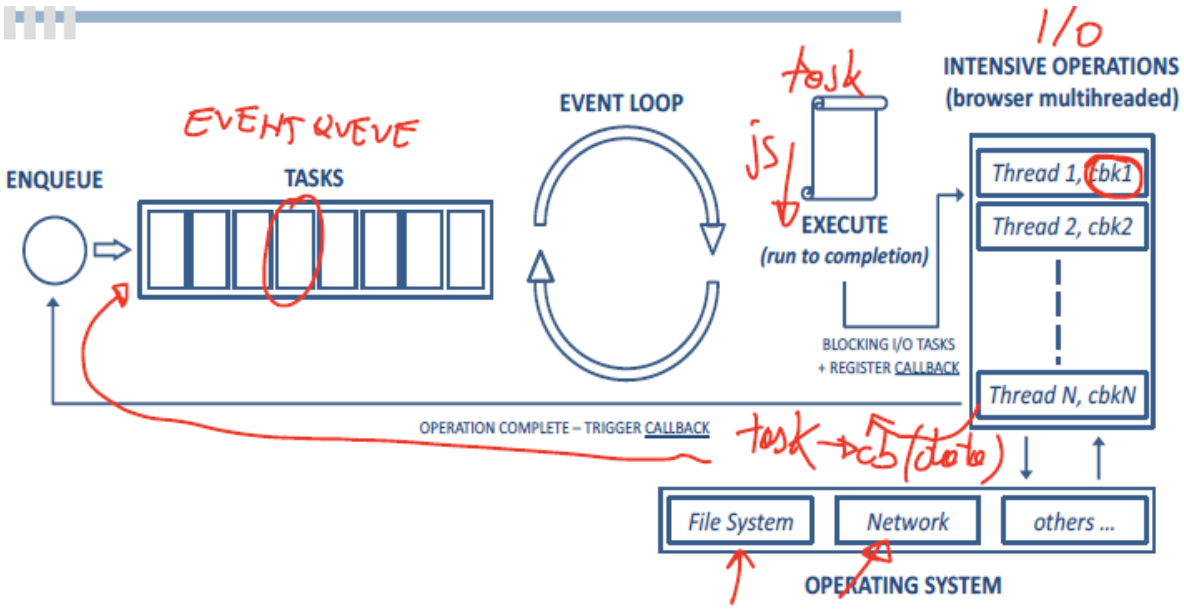
La callback è una funzione passata come parametro ad un'altra funzione ed eseguita al suo interno, può essere sincrona (eseguita prima che la funzione esterna ritorni) o asincrona (eseguita dopo che la funzione ritorna simil exec ma in un altro contesto). Js permette di registrare funzioni come handler per eventi specifici. **Le callback sono eseguite in un altro contesto** (dynamic) che è diverso da quello dove è stata definita (lexical/static).

Nei vecchi browser c'era un singolo thread per tutte le tab, utilizzando il paradigma **run-to-completion**, mentre oggi s'implementano più processi per tab/sito anche se con un overhead di memoria. Questo permette di sfruttare la programmazione asincrona in parallelo (background) sincronizzando eventualmente i thread con messaggi (**web workers**).

Run-to-completion avendo solo un thread le task dovevano essere veloci e ritornare altrimenti si finiva ad avere una ui non reattiva. La task corrente veniva sempre finita prima di iniziarne un'altra (good practice: be fast or split), avendo il controllo di tutto lo stato corrente non c'era bisogno di preoccuparsi delle modifiche in concorrenza.

Nella **programmazione ad eventi** ogni task agisce in risposta ad un evento (eg. Dom cambiato,

user click, ajax call returned). Per elaborare queste task si utilizza lo scheduler asincrono del browser il quale esegue un event loop, ogni task da lanciare viene aggiunta ad una coda FIFO Task queue ed una volta completata la prima task della coda viene eseguita. La callback della task completata viene a sua volta aggiunta alla coda.

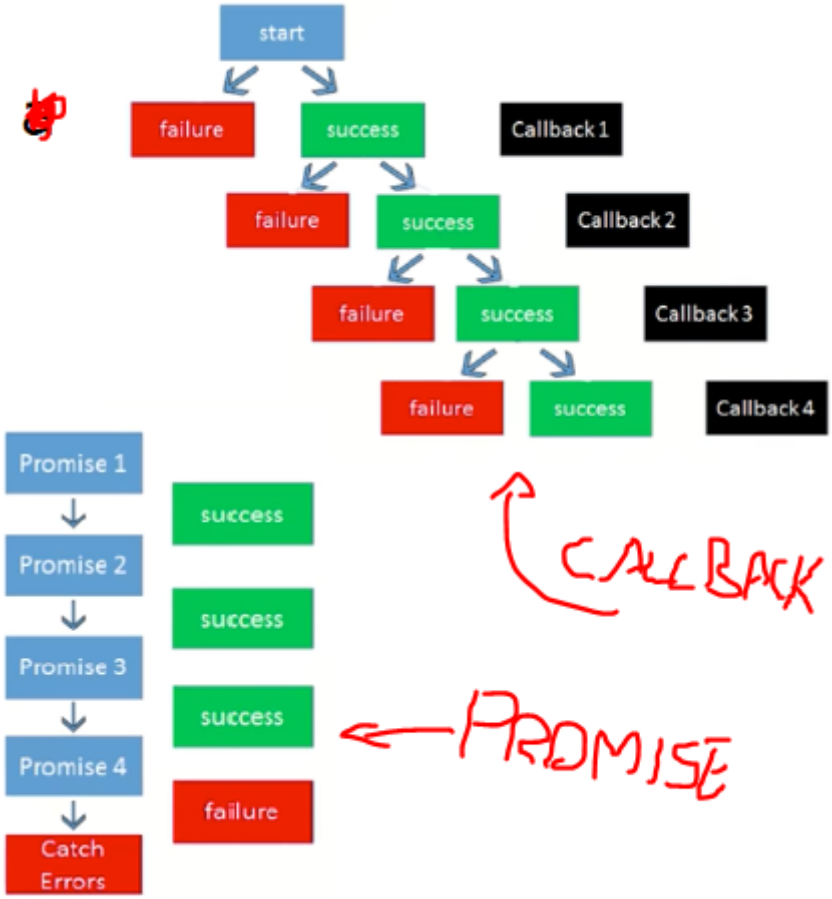


OSS: Js lavora su un thread singolo ergo nessun nuovo evento può essere processato finché la call stack attuale non è di nuovo vuota.

Pull vs Push:

- pull, data consumer decide quando prendere i dati dal producer, una funzione produce i dati e una chiamata a questa ritorna un singolo valore.
- push (usata negli observable), il producer di dati decide quando il consumer riceve i dati (eg. newsletter).

Callback Hell accade quando c'è la necessità di concatenare una sequenza di operazioni asincrone. Annidando una callback dentro l'altra si arriva ad avere una piramide di callback, ramificata in successo e fallimento, troppo profonda da leggere e gestire. Utilizzando invece le promise si pretende che le chiamate asincrone siano sincrone, ritornando una promessa per il dato, in questo modo la gestione degli errori e la concatenazione delle callback diventa più semplice.



La promise è un'interfaccia che rappresenta un proxy per un valore non necessariamente noto quando questa viene creata. Permette di associare handlers, per successo o fallimento ad un'azione asincrona. Queste permettono ai metodi asincroni di ritornare valori, anziché il valore finale, una promessa di avere quel valore in un certo momento nel futuro. Non "possiedono" i dati ricevuti, bisogna definire una callback per manipolarli una volta disponibili.

Le promise possiedono tre stati:

- Pending, non c'è ancora il dato, stato temporaneo -> può diventare fulfilled o rejected.
- Fulfilled, stato final (settled)
- Rejected, stato final (settled)

Le promises sono pensate per essere concatenate, ogni call alla funzione *.then()* permette di seguire una operazione async dopo l'altra e **ritorna (come ogni manipolazione) una nuova promise, è importante manipolare (e ritornare) sempre la fine della coda di promises**.

Async/await (ES2017) create per semplificare l'utilizzo di catene asincrone di promises (ma funge anche senza). La await mette in pausa la funzione async finché la promise non diventa settled. Await può essere usato solo all'interno di una funzione async. Try/catch per gestire gli errori.

09- JavaScript functional reactive programming

Gli observables sono un nuovo metodo per pushare dati in js. Un observable è un producer di valori multipli, ad es. uno stream di dati, a cui fare query e manipolare.

Risolve il problema delle callbacks trattando gli streams di eventi asincroni con la stessa semplicità con cui si usano le collections di dati stile array.

Risolve il problema delle promises nei casi in cui si aggiungano livelli di asincronismo d'esecuzione per i quali queste diventano non semplici da gestire e comporre in un flow di

esecuzione condizionato.

Gli observables sono ideati per comporre flussi e sequenze di dati asincroni.

La **Reactive functional programming** consiste nel programmare con streams di dati asincroni. Uno stream è una sequenza di eventi in corso ordinati **nel tempo** (anziché in memoria), gode di immutabilità aka gli operatori non lo modificano ma ritornano un nuovo stream.

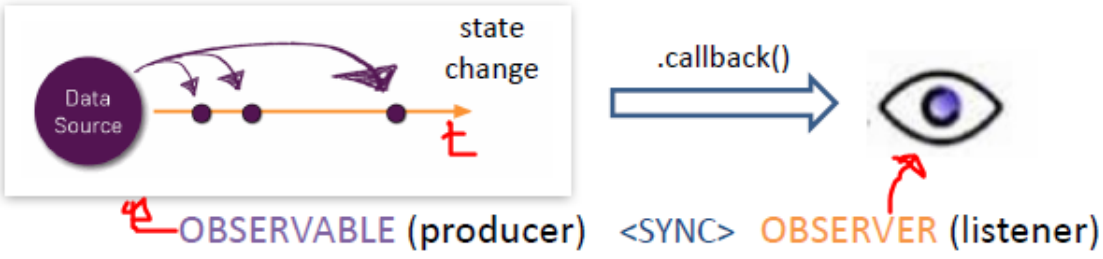
Gli eventi non erano abbastanza poiché devono forzare i side effects per avere un impatto (push degli observable) sul mondo e soprattutto negli eventi una serie di click non può essere passata come parametro e manipolata come sequenza che è.

Il pattern observable combina concetti dell'observer e dell'iterator.

- Il pattern observer (push):
  Abbiamo un oggetto chiamato producer che mantiene una lista interna di listeners che gli han fatto la subscribe. I listeners sono notificati chiamando il loro metodo update ogniqualvolta lo stato del producer cambia.
- Il pattern iterator (pull):
  Un iteratore è un oggetto che provvede al consumer un modo semplice per attraversare il suo contenuto. L'interfaccia è semplice (*next()* e *hasNext()*).

Un observable emette il suo valore in un ordine simile ad un iterator ma, anziché avere il consumer che lo richiede, pusha il valore al consumer quando disponibile. L'observable ha un ruolo simile a quello del producer, l'observer è l'equivalente del listener (riceve il valore nella sequenza in cui si rendono disponibili senza chiederli direttamente).
RxJS observable ha tre differenze essenziali dal pattern observer tradizionale:

- Cold:
  un observable non inizia a streamare finché almeno un observer non gli ha fatto la subscribe. (eg. Angular httpclient)
- Complete:
  come gli iterator, un observable può segnalare quando una sequenza è completata.
- Not Shared:
  un observable non condivide le chiamate subscribe a più observers (funzioni), ovvero **le chiamate *subscribe* sono *not shared among multiple observers of the same observable*.**
  (eg. return this.http.get().subscribe(cb) -> component -> template -> data$ | async).



RxJS è una libreria per comporre codice asincrono e basato su callback in uno stile functional "reactive".

Un observable può emettere tre cose differenti:

- un valore (di qualche tipo) più volte
- un segnale di errore
- un segnale di completamento

Un observer è un consumatore di valori delivered da un observable.

Gli observable sono created, subscribed to, executed or disposed. L'**unsubscribe** rilascia le risorse e cancella l'esecuzione dell'observable. Fare la subscribe ad un observable è come chiamare una funzione dandole una callback ove i dati saranno manipolati, ogni chiamata alla subscribe trigghera il suo setup indipendente per il dato observer.

Un observable non mantiene una lista degli observer a lui attaccati. La subscribe è semplicemente un metodo per iniziare l'esecuzione dell'observable e consegnare i valori o gli eventi all'observer di quella esecuzione.

Con la subscribe il dato observer non è registrato come listener nell'observable.

Un **hot observable** (eg. mouse clicks) produce valori indipendentemente dalla subscription, anche prima che sia attivo. I cold no.

Subject (producer/consumer):

il motivo per usarli è per il **multicast**, un observable di default è unicast ovvero ognuno dei suoi subscribed observer possiede un'esecuzione indipendente dell'observable. I subjects sono simili agli *eventEmitter*, **mantengono un registro di vari listeners**: quando subscribe su un subject non invoca una nuova execution che consegni i dati ma semplicemente registra il dato observer in una lista di observers. Mentre gli observables producono solo dati, i subjects possono esser usati sia per produrre che per consumare i dati. Usandoli come consumer si può convertire gli observables da unicast a multicast.

Il Subject è allo stesso tempo observer e observable.

Se ho bisogno di fare più subscribe uso il subject e faccio la subscribe sul suo observable usando *asObservable()*.

I subject si dividono in tre categorie:

- **BehaviorSubject**
  richiede un valore iniziale ed emette il suo valore corrente ai nuovi subscribers, di conseguenza si può prendere sempre l'ultimo valore emesso. Una volta che giunge a completion non emette più il valore corrente.
- ReplaySubject
  fa il "replay" cioè emette valori vecchi ai nuovi subscribers, ha un buffer di valori ed emetterà quei valori immediatamente ai nuovi subscribers insieme ai nuovi valori ai subscribers già esistenti. (eg. messaggi di una chat)
- AsyncSubject
  emette solo l'ultimo valore alla completion. Può esser visto come una promise, da usare quando i risultati intermedi non sono necessari.

Handle streams with operators

Vi sono vari operatori per lavorare con RxJS. Ad esempio per fare un observable sui click si usa *fromEvent*, una *map* che ritorna un nuovo stream e la *scan* prima del subscribe.

La *scan* a differenza della *reduce*, la quale emette solo dopo che l'observable ha completato, fa la *reduce* su un determinato lasso di tempo. Con *debounceTime* si può accumulare una lista di eventi, ogni tot di tempi senza eventi, prima di emetterli in uno stream di liste.

L'operatore *pipe()* riceve come argomento le funzioni che si vogliono combinare e ritorna una nuova funzione che una volta eseguita lancia le funzioni composte nella sequenza prescelta.

Combining observables (streams)

Essendo che le app hanno più sorgenti di input c'è bisogno di metodi per combinare le sequenze.

Concat(enation) fa la serializzazione, emette tutti gli elementi di un observable prima di iniziare l'emissione degli elementi del successivo.

Merge fa la parallelizzazione, combina gli output di più observables così da agire come un singolo observable interfogliando nel caso gli elementi degli observables mergiati. L'operatore *from* converte gli oggetti in observable, *toArray* crea una lista da una sequenza di observable.

Higher-order observables sono observable i cui valori sono essi stessi observables (eg. simile a lista di liste), outer observables i cui valori sono inner observables.

L'operatore ***mergeMap*** fa praticamente la *flatMap* su un observable high-order, ***concatMap*** invece li rende sequenziali.
Essendo che con la *mergeMap* gli observables sono eseguiti in parallelo ma i risultati ritornano in qualsiasi order (eg. http result are in response not request order), esiste la ***forkJoin*** la quale riceve una lista di observables, li esegue (subscribe and run) in parallelo e solo quando ogni observable nella lista emette il valore questo emetto un singolo array contenente tutti gli observables completati. (eg. tutte le risposte alle http requests). Se ne fallisce uno ritorna subito.

10- Angular forms

I form sono cruciali per ottenere i dati dall'utente in tanti applicativi. Sono complessi poiché modificano dati non solo sulla pagina ma anche sul server, dati che devono esser validati e gli errori visualizzati.

Molte app frontend si possono ridurre a dei giganteschi form.
Con Angular, ***FormControls*** ci ritorna un oggetto per gestire gli input dei form, ***Validators*** definisce e applica regole agli input, ***Observers*** ci permette di tracciare i cambiamenti del form.

C'è un approccio duale ai form:

- Template driven
  Controlli e regole di validazione definiti nel template con direttive (creato implicitamente e asincrono)
- Model driven (or reactive)
  Controlli e regole di validazione definiti nella classe componente o servizio (creato esplicitamente/programmaticamente e sincrono -> no template rendering)

I **Template driven** forms features sono esportati da *FormsModule*. *ngForm* accresce implicitamente ogni elemento form.
Essendo scritti in html non possono essere testati con UnitTest. Ogniqualvolta il valore di un form di controllo cambia Angular lancia una validazione ed aggiorna la lista delle proprietà di validazione. Esistono validatori prefatti (required, pattern, min/maxlength..) ed è possibile visualizzare messaggi di errore in base al loro esito. (onSubmit diventa in Angular *ngSubmit*).
Usato per controlli semplici.

Reactive forms (model driven) sono i form i cui controlli e validazioni sono definite nella classe component o servizio come funzioni. (*FormControl*, *FormGroup* per fare il binding). I form reattivi

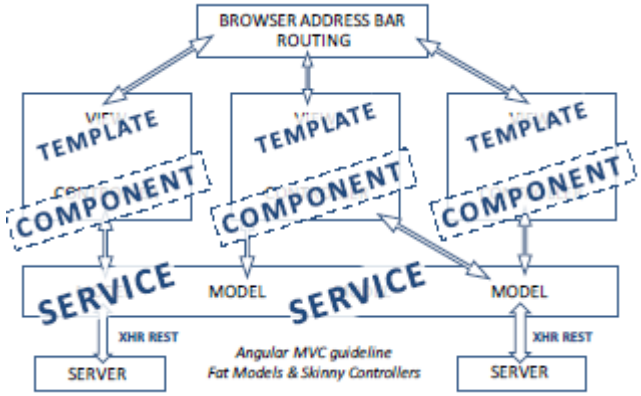
hanno metodi per cambiare un controllo o un valore programmaticamente (eg. *setValue()*, *patchValue()*).

Le istanze dei form reattivi hanno un metodo *valueChanges* che ritorna un observable il quale emette gli ultimi valori del form, si può fare la subscribe a *valueChanges* per aggiornare le variabili dell'istanza o perfomare azioni come aggiornare il modello (da aggiornare solo nel caso di un form valido ovviamente). Usato per controlli complessi.

11- Angular routing

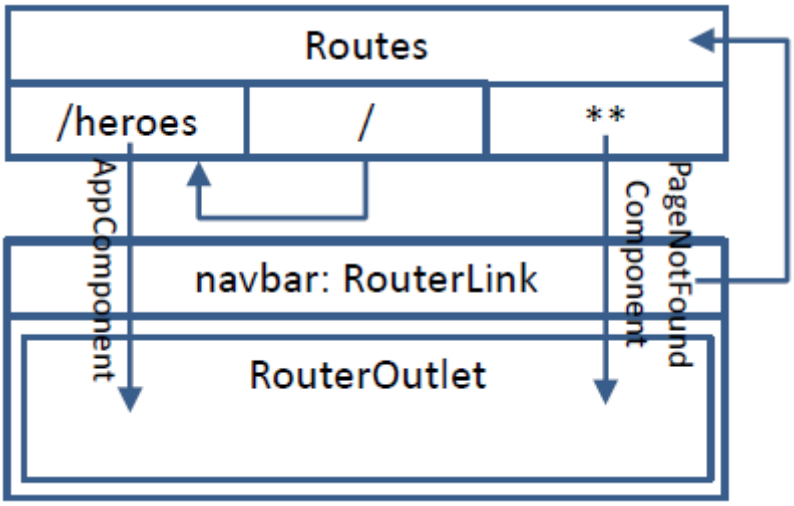
SPA complesse hanno stati multipli che possono corrispondere a view multiple le quali possono essere mappate con url diverse.

Il **routing** permette di definire ed instradare ad un unico stato dell'app basandosi sulla url corrente, così da caricare una view differente nella spa. Si ci riferisce sempre a SPAs perché il server ci ritorna sempre una singola pagina ma è il nostro js che la renderizza in pagine diverse.



I componenti del routing di Angular sono:

- Routes
  descrivono le routes che supporta l'app
- RouterOutlet
  un componente placeholder che mostra ad Angular dove mettere il contenuto di ogni route
- RouterLink
  una direttiva usata per linkare alle route.



Angular Router è un servizio opzionale che presenta una vista di un particolare componente per una data url, ogniqualvolta l'url cambia il router cerca la route che determina il componente da far vedere.

> Slides Servetti_11_AngularRouting-v2 2018/19 p10->p41 riguardano l'implementazione e non sono qui riportate in riassunti.

12- Angular architecture

Le interazioni tra componenti padre-figlio (o fratello) possono avvenire via:

- Input/output
  il padre passa al figlio dati bindati in input, *@Input()*, ed ascolta gli eventi (output) del figlio.
- Direct reference
  via local variable o *@ViewChild()*.
- Condividendo un servizio

La **vista** dell'app sarà composta da più componenti alcuni di questi stateful (eg. main) altri stateless.

Essenzialmente esistono due tipi di componenti specializzati:

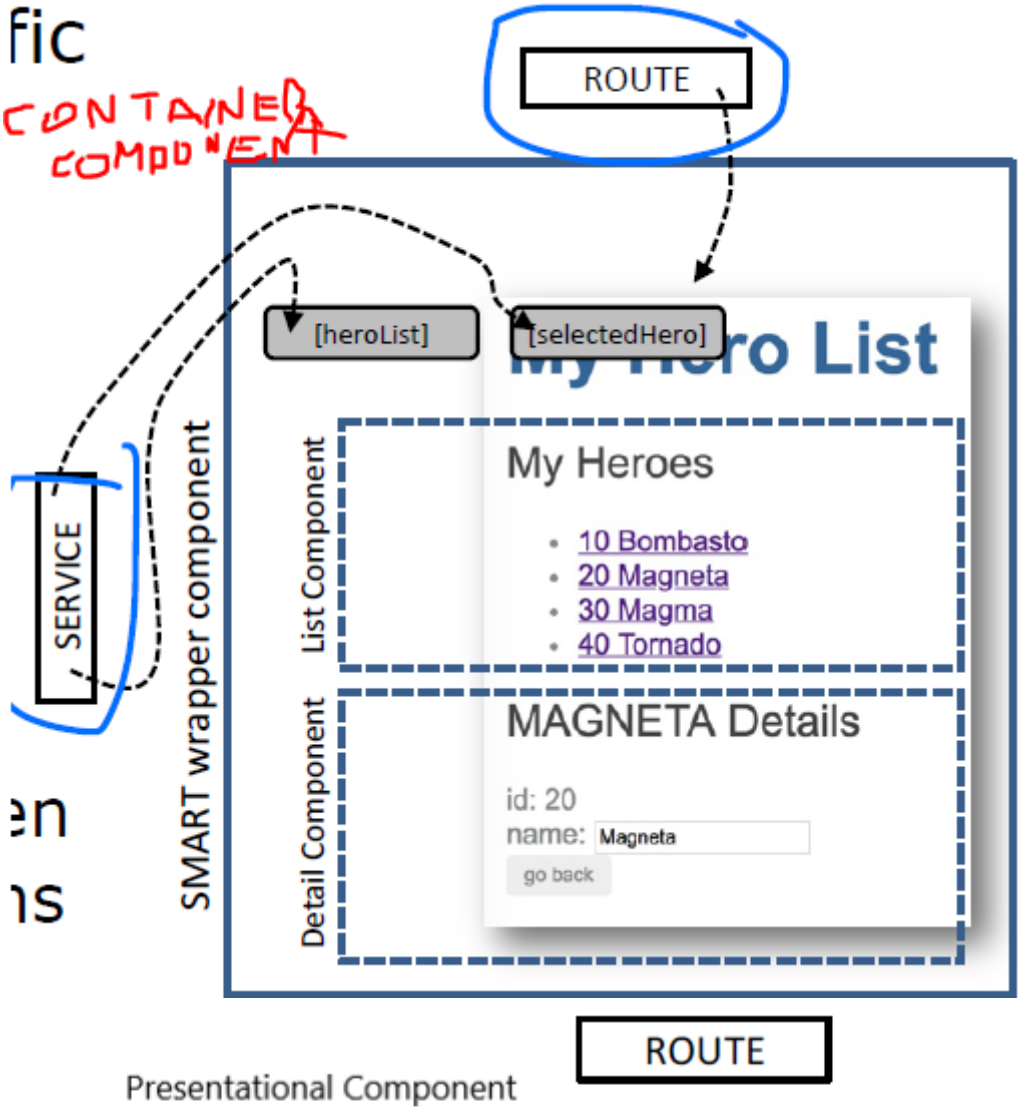
- Smart/Container components
  stateful e responsabili di fecciare i dati che potrebbe dover essere visualizzato e mantenere lo stato.
- Dummy/Presentational components
  riutilizzabili attraverso un'interfaccia esplicita, prendendo input (sapendo come mostrarlo) ed emettendo eventi custom come output.

I component SMART top-level o container ricevono nel costruttore le dipendenze specifiche per la loro applicazione, sanno come recuperare i dati da un servizio/route e di che tipo sono, sono stateful e possono gestire le operazioni asincrone. Gestiscono gli eventi dei componenti interni (sub components) eseguendo le azioni corrette.

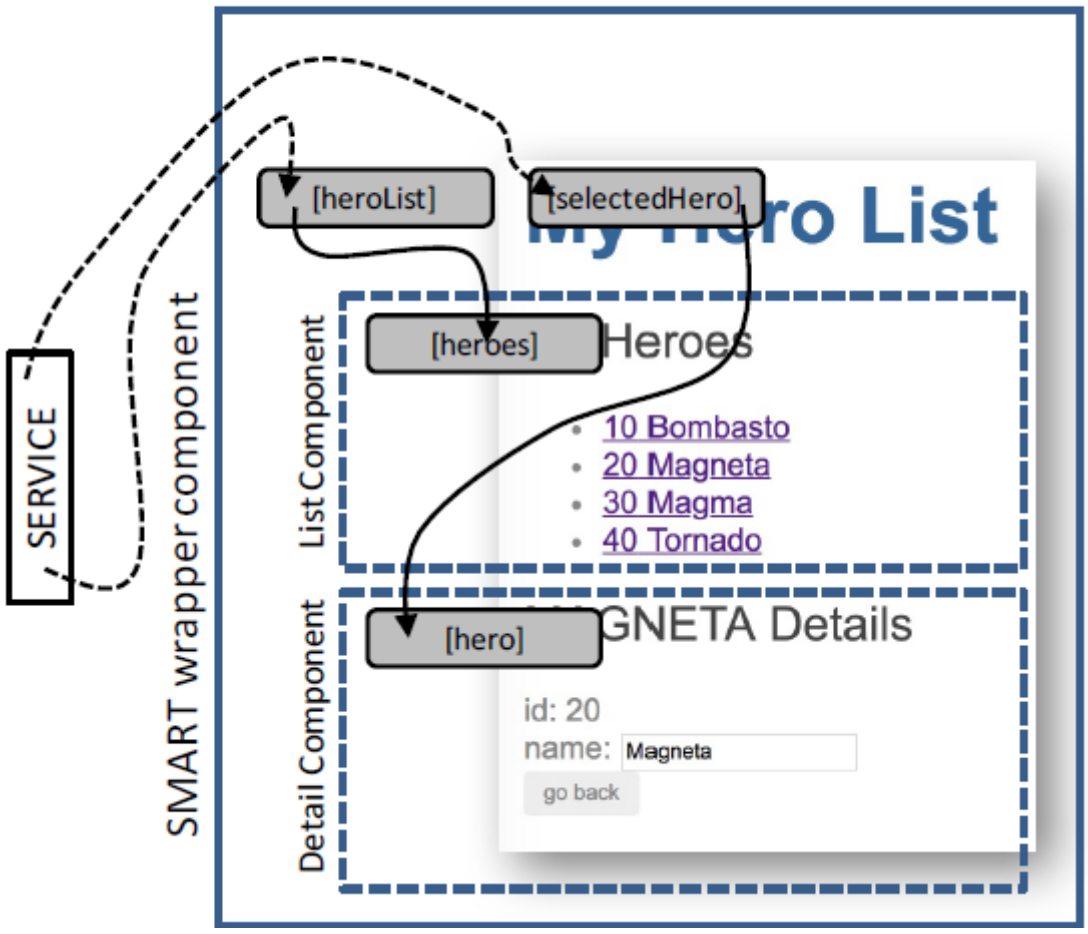
I presentational component devono essere riutilizzabili mantenendoli più semplici possibile e riducendo le dipendenze implicite, quando un componente child fa un'azione che deve essere notificata al parent viene lanciato un evento delegando così anche tutti i cambiamenti e gli update dei dati ai componenti stateful.

fic

CONTAINER COMPONENT

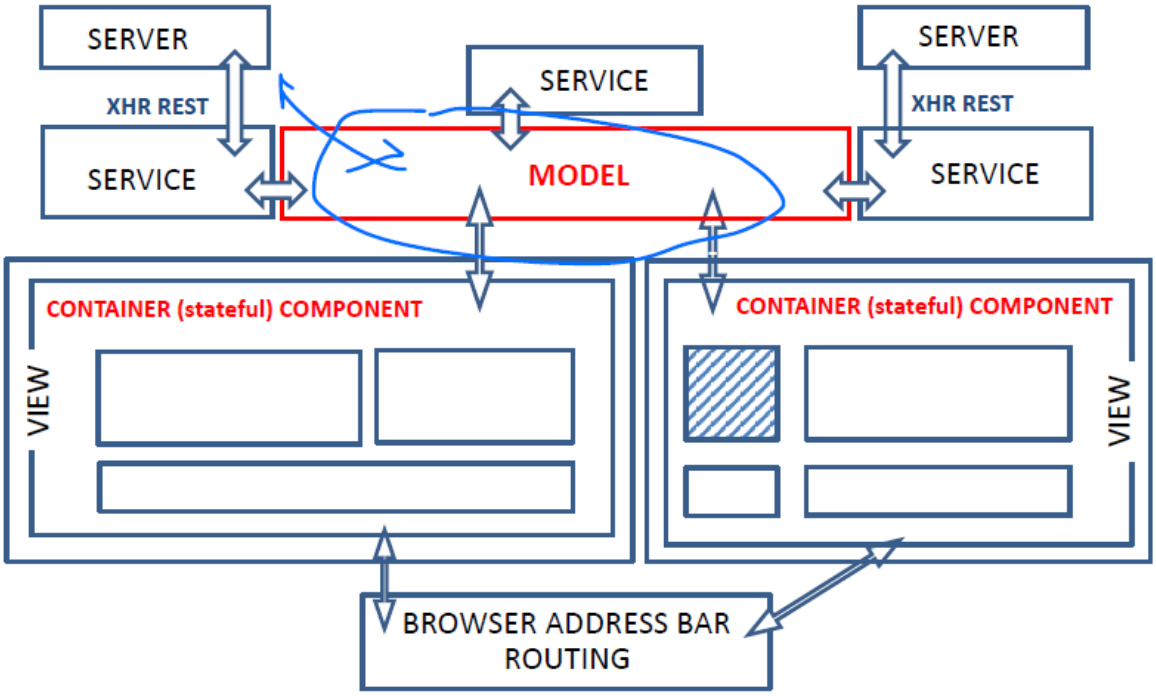


Presentational Component



Best practice:

- Includere tutti i file pertinenti ad un component nella stessa cartella.
- Mantenere i templates abbastanza piccoli da metterli direttamente nel componente main.
- Delegare la logica di business dal componente al servizio.
- Non aver paura di suddividere un componente se sta diventando troppo grosso.
- Considerare costantemente i risvolti di ogni cambiamento mentre si sviluppa l'app.



Client-side model
Se si vuole mantenere un modello lato client dei dati del server ad esempio per lavorare offline (eg google Docs) abbiamo bisogno di un servizio di store che si comporti al contempo come producer di dati per i components e come consumer di dati del server.

Un servizio che può esser chiamato "**store**" è basato su RxJS Subject ed è un Observable Service, uno speciale caso di Subject che viene utilizzato al fine di emettere (per una subscription) il valore più aggiornato i.e. BehaviorSubject.

Un observable service crea un BehaviorSubject privato che emetterà i dati ed un pubblico Observable a cui si registreranno i components.

Domande di esempio di Servetti

Descrivere

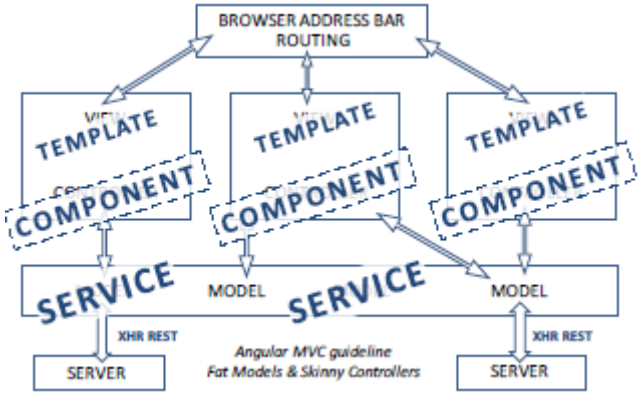
i) l'architettura di una applicazione realizzata con Angular

ii) il ruolo dei service, il loro ciclo di vita e il meccanismo della injection. Fornire un esempio minimale (omettendo imports e decorators) di come due componenti possono condividere informazioni attraverso un service.

Angular è un framework per gestire applicazioni web sul modello Single Page Application. Permette le seguenti funzionalità:

- Routing
- Rest
- Data binding
- Dependency injection https://angularfirebase.com/lessons/sharing-data-between-angular-components-four-methods/
- Estende il modello html per gestire parte della logica di business dal client

L'architettura può essere rappresentata nel modo seguente:



Componenti*:* è l'unità base dell'ui di un'applicazione Angular. Di fatto questa non è nient'altro che un albero di component. Un component è costituito da più classi:

- *View*: è un file html che si occupa della presentazione dei dati locali gestiti dal component.
- *Style*: lo stile può essere gestito con dei file css con uno scope specifico sul component per permettere maggiore riusabilità, oltre al tradizionale css system-wide.
- *Classe typescript*: si occupa di gestire la logica di controllo dei dati, idealmente recuperati tramite servizi.

Servizi: sono degli oggetti con le seguenti funzionalità:

- implementano la business logic dell'applicazione
- modificano il modello dei dati
- sincronizzano i dati con il server

In Angular gli oggetti vengono creati quando necessario e distrutti quando non più usati. Gli agganci al lifecycle sono varie funzioni tra cui OnInit, OnDestroy, OnChanges, DoCheck.

Angular non crea i servizi ogni qualvolta un componente lo richiede ma essi sono creati dai providers. Il componente cerca di risolvere la dipendenza usando la sua gerarchia di providers. Se un provider viene definito nel modulo root tutte le dipendenze per un token all'interno dell'app saranno risolte utilizzando lo stesso oggetto (singleton).

Quel che differenzia un servizio da un normale oggetto è che sono gestiti dalla dependency injection, sono *provided/injected to building block* da un provider esterno. Un servizio viene reso disponibile ad un component utilizzando la provision e la injection. La provision dichiara il servizio come dipendenza utilizzando il costruttore che poi verrà risolto utilizzando i provider dichiarati nell'AppModule. L'injection utilizza il costruttore per la dependency injection e l'ngOnInit per lanciarla.

Le interazioni tra componenti padre-figlio (o fratello) possono avvenire via:

- Input/output(padre-figlio)
- https://angularfirebase.com/lessons/sharing-data-between-angular-components-four-methods/

  il padre passa al figlio dati bindati in input, @Input(), ed ascolta gli eventi (output) del figlio.
- Direct reference (padre figlio)

  via local variable o @ViewChild().
- Condividendo un servizio (Sibling)

Un esempio del terzo caso:

data.service.ts

```
import { Injectable } from '@angular/core';
import { BehaviorSubject } from 'rxjs';

@Injectable()
export class DataService {

  private messageSource = new BehaviorSubject('default message');
  currentMessage = this.messageSource.asObservable();

  constructor() { }

  changeMessage(message: string) {
    this.messageSource.next(message)
  }

}
```

parent.component.ts

```
import { Component, OnInit } from '@angular/core';
import { DataService } from "../data.service";

@Component({
  selector: 'app-parent',
  template: `
    {{message}}
  `,
  styleUrls: ['./sibling.component.css']
})
export class ParentComponent implements OnInit {

  message:string;

  constructor(private data: DataService) { }

  ngOnInit() {
    this.data.currentMessage.subscribe(message => this.message = message)
  }

}
```

sibling.component.ts

```
import { Component, OnInit } from '@angular/core';
import { DataService } from "../data.service";

@Component({
  selector: 'app-sibling',
  template: `
    {{message}}
    <button (click)="newMessage()">New Message</button>
  `,
  styleUrls: ['./sibling.component.css']
})
export class SiblingComponent implements OnInit {

  message:string;

  constructor(private data: DataService) { }

  ngOnInit() {
    this.data.currentMessage.subscribe(message => this.message = message)
  }

  newMessage() {
    this.data.changeMessage("Hello from Sibling")
  }

}
```

Descrivere

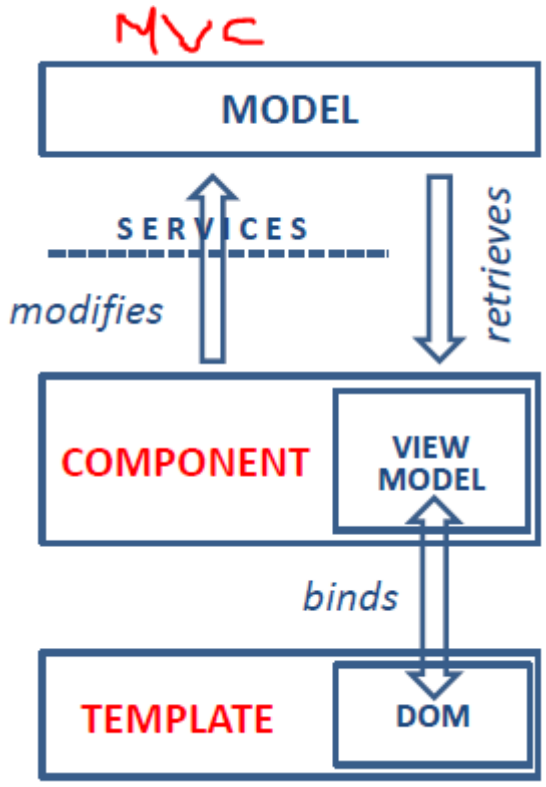
i) l'architettura di una applicazione realizzata con Angular

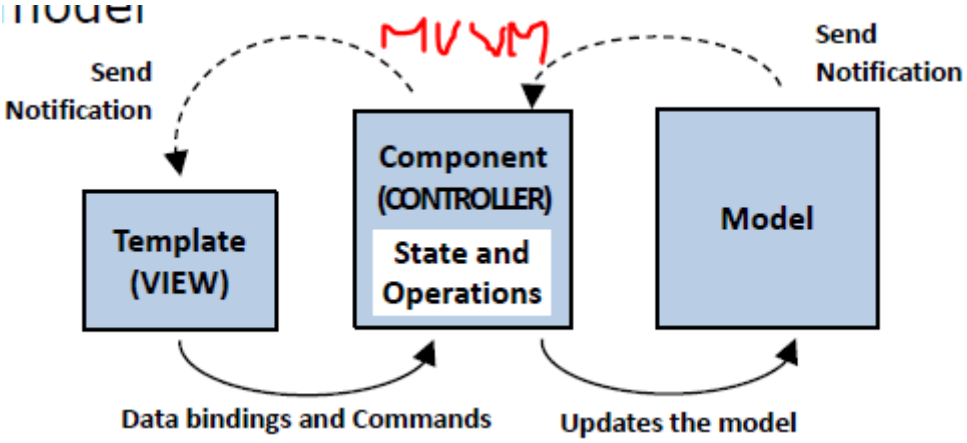
ii) quale ruolo svolgono i template, i componenti e i servizi in relazione al pattern MVC/MVVM

iii) il concetto di componenti container (smart) e presentational (dummy). Fornire un esempio minimale (omettendo imports e decorators) di utilizzo di un componente.

i) vedi domanda precedente

ii) MVC e MVVM:





In Angular il controllore del MVC diventa il component, la vista il template. Può essere tenuto un modello dati locale all'interno del servizio, ma la persistenza vera e propria viene mantenuta sul server e sincronizzata attraverso i servizi. Angular permette di strutturare il codice secondo le preferenze del programmatore sia in MVC (maggior separazione tra vista e logica) sia MVVM (il modello legge i cambiamenti della vista in modo autonomo e si aggiorna), il termine utilizzato per indicare questa possibilità in un framework è Model-View-Whatever (MVW). From MVC to MVVM (ModelView-ViewModel): il ViewModel non è un controllore ma lega i dati tra vista e modello al fine di farli comunicare l'un con l'altro.

iii) Essenzialmente esistono due tipi di componenti specializzati:

- Smart top level/Container components

  stateful e responsabili di fetching data che potrebbe dover essere visualizzato e mantenere lo stato.

- Dummy/Presentational components

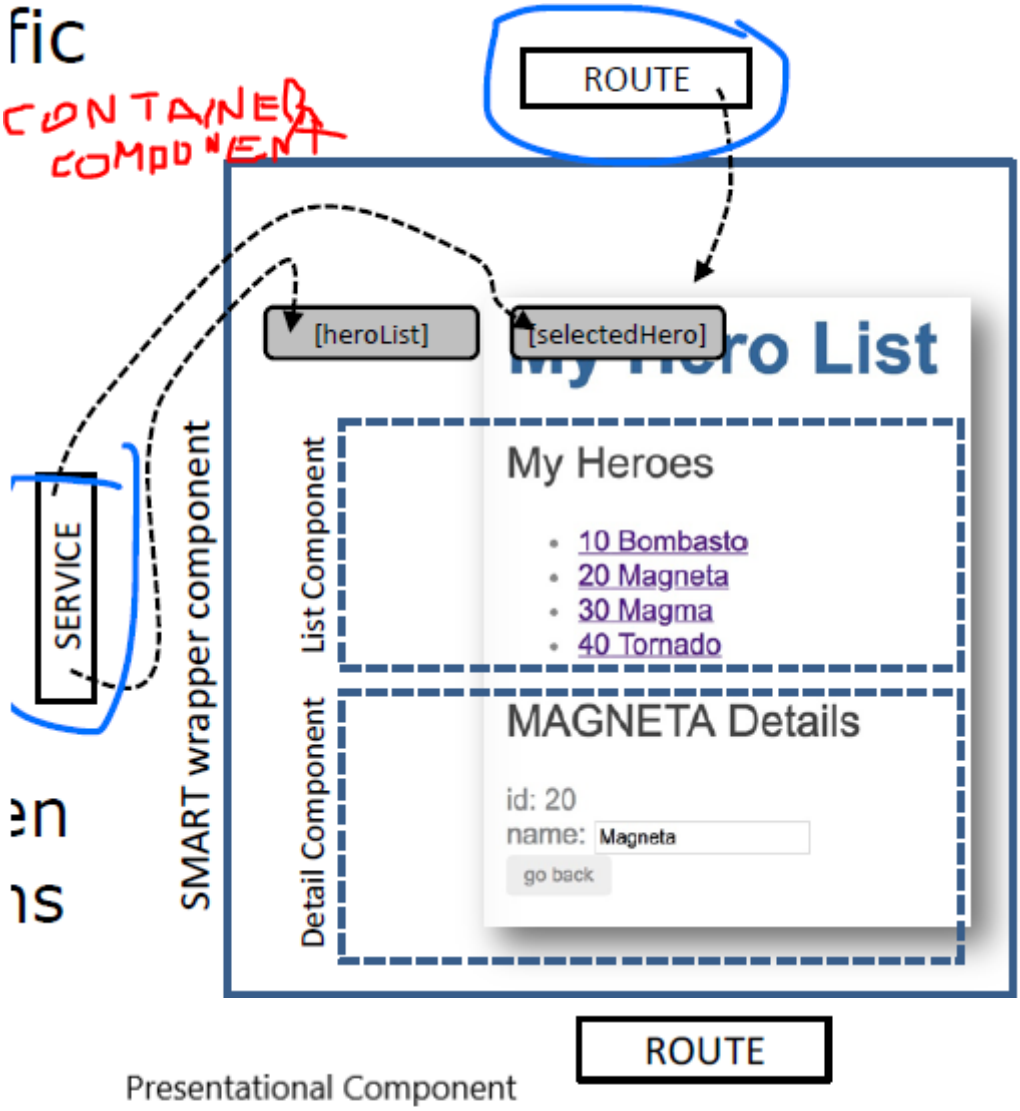
  riutilizzabili attraverso un'interfaccia esplicita, prendendo input (sapendo come mostrarlo) ed emettendo eventi custom come output.

I component SMART top-level o container ricevono nel costruttore le dipendenze specifiche per la loro applicazione, sanno come recuperare i dati da un servizio/route e di che tipo sono, sono stateful e possono gestire le operazioni asincrone. Gestiscono gli eventi dei componenti interni (sub components) eseguendo le azioni corrette.

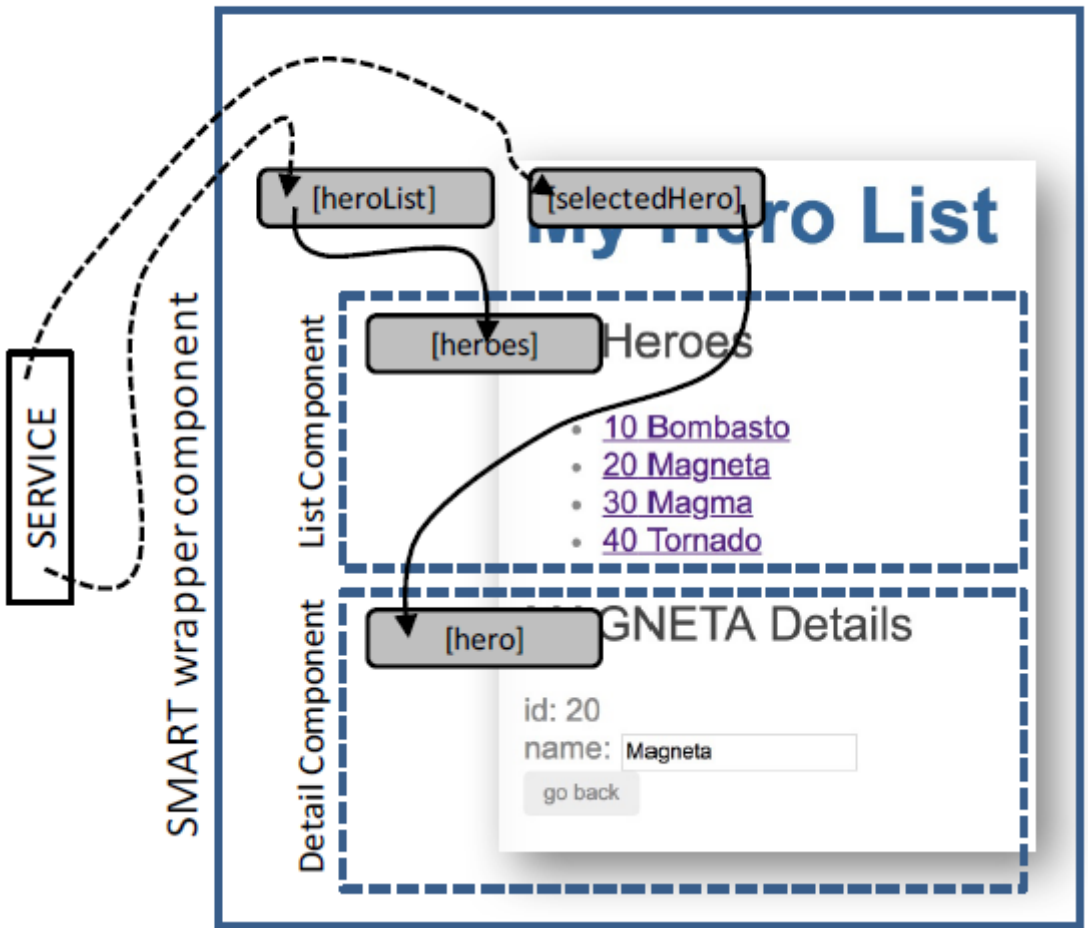
I presentational component devono essere riutilizzabili mantenendoli più semplici possibile e riducendo le dipendenze implicite, quando un componente child fa un'azione che deve essere notificata al parent viene lanciato un evento delegando così anche tutti i cambiamenti e gli update dei dati ai componenti stateful.

fic

CONTAINER COMPONENT



Presentational Component



mycomponent.html:

{{ title }}

mycomponent.ts:

```
 @Component({
  selector: 'app-mycomponent',
  templateUrl: './mycomponent.component.html',
  styleUrls: ['./mycomponent.component.scss']
})
export class mycomponent implements onInit {
  title: String = ' '
  // other attributes
 ngOnInit(){
    setMessage()
  }
  void setMessage(){
    this.title = "Hello world!"
  }
  //other functions
}
```

Descrivere brevemente i due approcci per la realizzazione di form in Angular. Chiarire cosa li distingue e in quali contesti è preferibile l'uno all'altro. Fornire un esempio di validatore personalizzato, ad esempio, per verificare che le due copie della password immessa non siano identiche.

C'è un approccio duale ai form:

- Template driven

  Controlli e regole di validazione definiti nel template con direttive (creato implicitamente e asincrono), no UnitTest

- Model driven (or reactive)

  Controlli e regole di validazione definiti nella classe componente o servizio (creato

  esplicitamente/programmaticamente e sincrono -> no template rendering)

I Template driven forms features sono esportati da FormsModule. ngForm accresce implicitamente ogni elemento form. Essendo scritti in html non possono essere testati utilizzando gli UnitTest. Ogniqualvolta il valore di un form di controllo cambia Angular lancia una validazione ed aggiorna la lista delle proprietà di validazione. Esistono validatori prefatti (required, pattern, min/maxlength..) ed è possibile visualizzare messaggi di errore in base al loro esito. (onSubmit diventa in Angular ngSubmit). Utile per controlli semplici.

Reactive forms (model driven) sono i form i cui controlli e validazioni sono definite nella classe component o servizio come funzioni. (FormControl, FormGroup per fare il binding). I form reattivi hanno metodi per cambiare un controllo o un valore programmaticamente (eg. setValue(), patchValue()).

Le istanze dei form reattivi hanno un metodo valueChanges che ritorna un observable il quale emette gli ultimi valori del form, si può fare la subscribe a valueChanges per aggiornare le variabili dell'istanza o perfomare azioni come aggiornare il modello (da aggiornare solo nel caso di un form valido ovviamente). Usato per controlli complessi.

Esempio:

// mycomponent.ts

```
 function checkPasswords(group: FormGroup){
  let pass = group.controls.password.value;
  let confirmPass = group.controls.confirmPassword.value;

   return pass === confirmPass ? null : { error: true };
}
  export class MyComponent implements OnInit {
  myForm: FormGroup;

   constructor(private formBuilder: FormBuilder) {
     this.myForm = this.formBuilder.group({
       email: ['', [Validators.required, Validators.email]],
       password: ['', Validators.required],
       confirmPassword: ['']
     }, { validators: [checkPasswords]});
   }
}
```

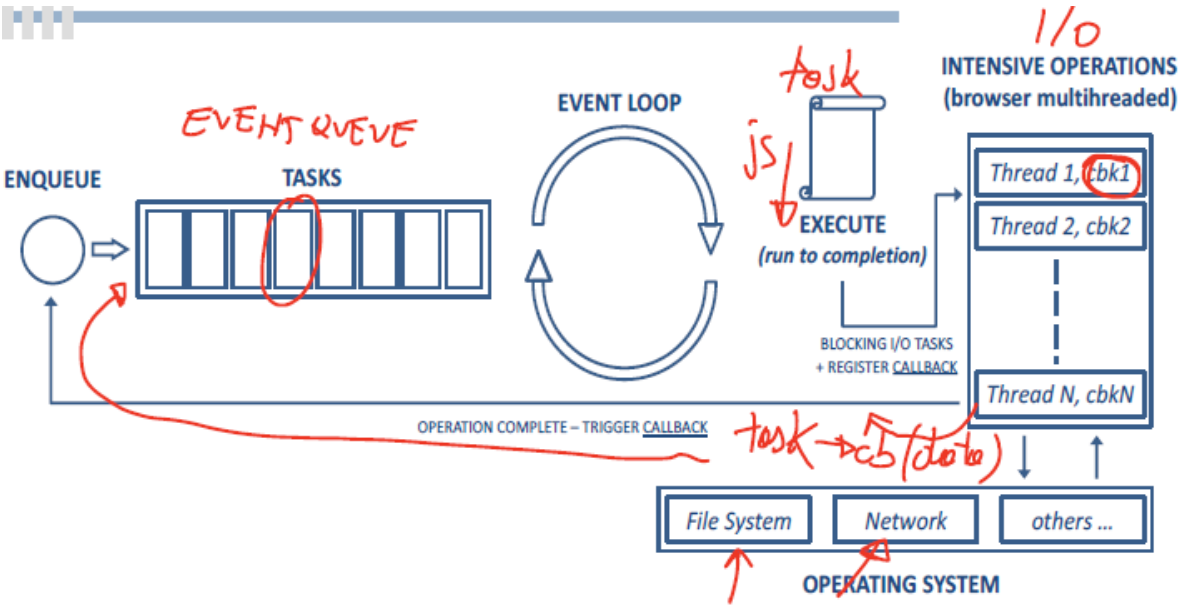
// mycomponent.html: <form [formGroup]="myForm">

Enter your email

Password

Repeat password

Descrivere i) il meccanismo di gestione degli eventi in Javascript tramite l'event loop, ii) come viene gestita in particolare una richiesta AJAX e relativa callback, iii) chiarire quali sono i vantaggi e svantaggi della gestione/programmazione asincrona



Nella programmazione ad eventi ogni task agisce in risposta ad un evento (eg. Dom cambiato, user click, ajax call returned). Per elaborare queste task si utilizza lo scheduler asincrono del browser il quale esegue un event loop, ogni task da lanciare viene aggiunta ad una coda FIFO-Task ed una volta completata la prima task della coda viene eseguita. La callback della task completata viene a sua volta aggiunta alla coda. In particolare una richiesta ajax asincrona quando completata viene inserita nella coda degli eventi, una volta estratta dalla coda la sua relativa callback viene reinserita all'interno della FIFO ed eseguita solo al loop successivo.

OSS: Js lavora su un thread singolo ergo nessun nuovo evento può essere processato finché la call stack attuale non è di nuovo vuota.

I vantaggi della programmazione asincrona sono:

- c'è un minor overhead per la creazione e la manutenzione dei thread
- non c'è overhead per il context-switching

Gli svantaggi sono:

- paradigma di programmazione diverso (callback)
- i task devono essere piccoli per non bloccare o ritardare troppo altri task

Descrivere i tre approcci callback, promise, observable per la gestione delle operazioni asincrone in javascript. Precisare in particolare quali sono i vantaggi delle promise rispetto alle callback e degli observable rispetto alle promise. Fornire un semplice esempio di utilizzo degli observables per la gestione di una richiesta AJAX (con gestione degli errori).

La callback è una funzione passata come parametro ad un'altra funzione ed eseguita al suo interno, può essere sincrona (eseguita prima che la funzione esterna ritorni) o asincrona (eseguita dopo che la funzione ritorna simil exec ma in un altro contesto). Js permette di registrare funzioni come handler per eventi specifici. Le callback sono eseguite in un altro contesto (dynamic) che è diverso da quello dove è stata definita (lexical/static).

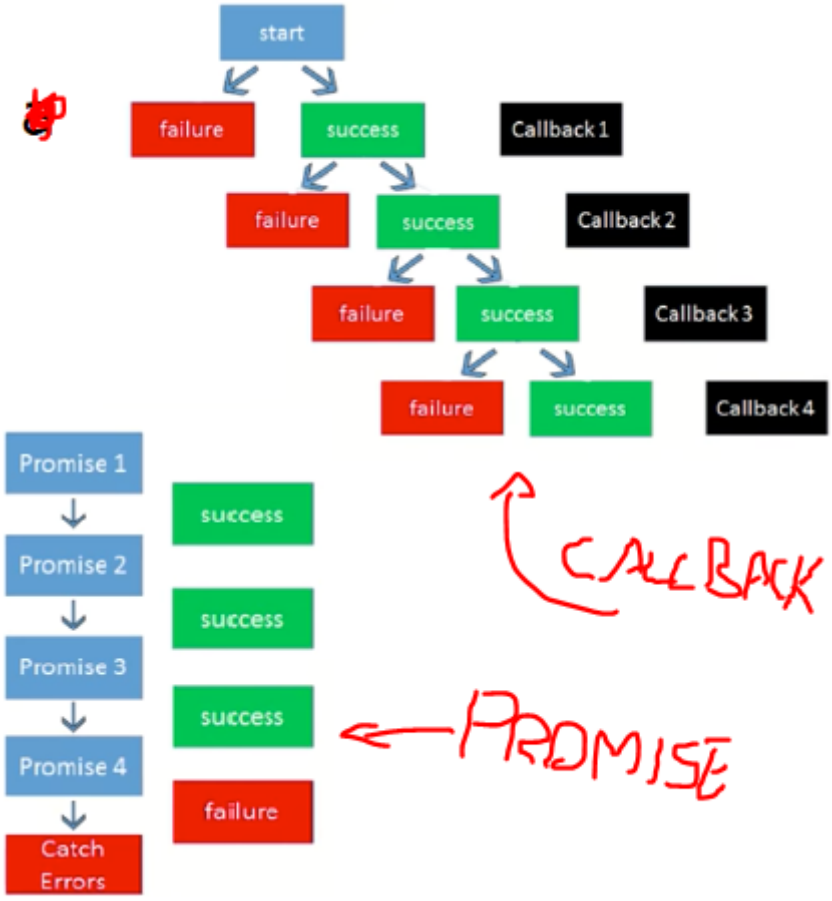
Callback Hell accade quando c'è la necessità di concatenare una sequenza di operazioni asincrone. Annidando una callback dentro l'altra si arriva ad avere una piramide di callback, ramificata in successo e fallimento, troppo profonda da leggere e gestire. Utilizzando invece le promise si pretende che le chiamate asincrone siano sincrone, ritornando una promessa per il dato, in questo modo la gestione degli errori e la concatenazione delle callback diventa più semplice.

La promise è un'interfaccia che rappresenta un proxy per un valore non necessariamente noto quando questa viene creata. Permette di associare handlers, per successo o fallimento ad un'azione asincrona. Queste permettono ai metodi asincroni di ritornare valori, anziché il valore finale, una promessa di avere quel valore in un certo momento nel futuro. Non "possiedono" i dati ricevuti, bisogna definire una callback per manipolarli una volta disponibili.

Le promise possiedono tre stati:

- Pending, non c'è ancora il dato, stato temporaneo -> può diventare fulfilled o rejected.
- Fulfilled, stato final (settled)
- Rejected, stato final (settled)

Le promises sono pensate per essere concatenate, ogni call alla funzione .then() permette di seguire una operazione async dopo l'altra e ritorna (come ogni manipolazione) una nuova promise, è importante manipolare (e ritornare) sempre la fine della coda di promises.



Gli observables sono un nuovo metodo per pushare dati in js. Un observable è un producer di valori multipli, ad es. uno stream di dati, a cui fare query e manipolare.

Risolve il problema delle callbacks trattando gli streams di eventi asincroni con la stessa semplicità con cui si usano le collections di dati stile array.

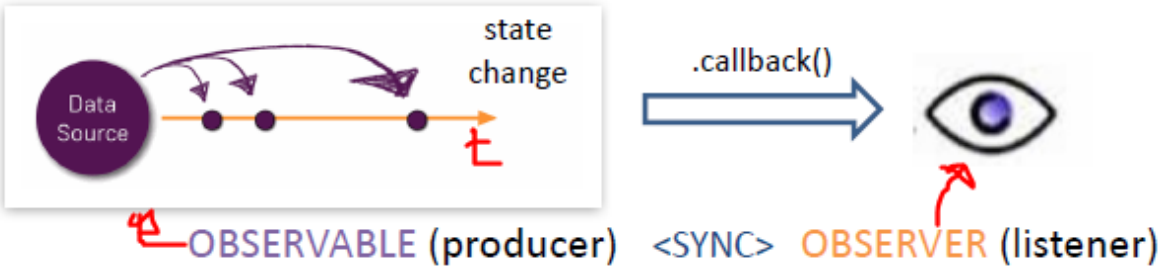
Risolve il problema delle promises nei casi in cui si aggiungano livelli di asincronismo d'esecuzione per i quali queste diventano non semplici da gestire e comporre in un flow di esecuzione condizionato.

Gli observables sono ideati per comporre flussi e sequenze di dati asincroni.

La Reactive functional programming consiste nel programmare con streams di dati asincroni.

Uno stream è una sequenza di eventi in corso ordinati nel tempo (anziché in memoria), gode di immutabilità aka gli operatori non lo modificano ma ritornano un nuovo stream.

Gli eventi non erano abbastanza poiché devono forzare i side effects per avere un impatto (push degli observable) sul mondo e soprattutto negli eventi una serie di click non può essere passata come parametro e manipolata come sequenza che è.




```
const obs$ = ajax(url/to/request).pipe(
  operation_on_returned_data,
  catchError(error => {
    console.log('error: ', error);
    return of(error);
  }).subscribe(
  data => (operation_on_returned_piped_data),
  error => (console.log('error: ', error);)
  )
);
```

Descrivere il pattern observable/observer e il concetto di functional reactive programming. Fornire un esempio di calcolo del numero di click del mouse sia come marble diagram, sia come codice. Spiegare brevemente il compito di ciascun operatore utilizzato.

Il pattern observable combina concetti dell'observer e dell'iterator.

Il pattern observer (push):

Abbiamo un oggetto chiamato producer che mantiene una lista interna di listeners che gli han fatto la subscribe. I listeners sono notificati chiamando il loro metodo update ogniqualvolta lo stato del producer cambia.

Il pattern iterator (pull):

Un iteratore è un oggetto che provvede al consumer un modo semplice per attraversare il suo contenuto. L'interfaccia è semplice (next() e hasNext()).

Un observable emette il suo valore in un ordine simile ad un iterator ma, anziché avere il consumer che lo richiede, pusha il valore al consumer quando disponibile. L'observable ha un ruolo simile a quello del producer, l'observer è l'equivalente del listener (riceve il valore nella sequenza in cui si rendono disponibili senza chiederli direttamente).

RxJS observable ha tre differenze essenziali dal pattern observer tradizionale:

- Cold

  un observable non inizia a streamare finché almeno un observer non gli ha fatto la subscribe. (eg. Angular httpclient)

- Complete

  come gli iterator, un observable può segnalare quando una sequenza è completata.

- Not Shared

  un observable non condivide le chiamate subscribe a più observers (funzioni), ovvero le chiamate subscribe sono not shared among multiple observers of the same observable. (eg. return this.http.get().subscribe(cb) -> component -> template -> data$ | async).

Un observer è un consumatore di valori forniti da un observable.

Gli observable sono created, subscribed to, executed or disposed. L'unsubscribe rilascia le risorse e cancella l'esecuzione dell'observable. Fare la subscribe ad un observable è come chiamare una funzione dandole una callback ove i dati saranno manipolati, ogni chiamata alla subscribe trigghera il suo setup indipendente per il dato observer.

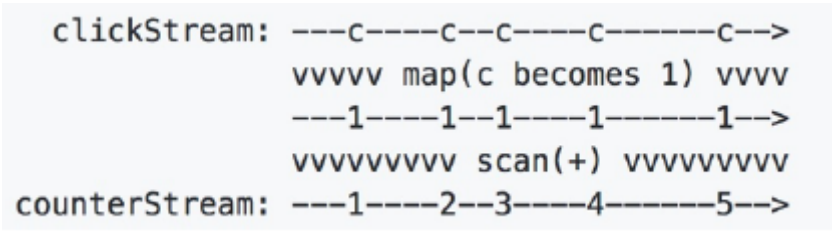
Un observable non mantiene una lista degli observer a lui attaccati. La subscribe è semplicemente un metodo per iniziare l'esecuzione dell'observable e consegnare i valori o gli eventi all'observer di quella esecuzione.

Con la subscribe il dato observer non è registrato come listener nell'observable.

Un hot observable (eg. mouse clicks) produce valori indipendentemente dalla subscription, anche prima che sia attivo. I cold no.

Esempio:

```
Rx.Observable.fromEvent(button, 'click')
.map(
  (val) => 1)
.scan(
  (acc, cur) => acc+cur, 0)
.subscribe(
  (count) => console.log(`Clicked ${count} times`)
);
```



- From event: genera l'observable.
- Map: scambia ogni elemento dello stream col risultato dell'operazione data
- Scan: a differenza della reduce, la quale emette solo dopo che l'observable ha completato, fa la reduce su un determinato lasso di tempo.
- Subscribe: questo operatore collega un observer ad un observable. Al fine di vedere gli elementi emessi da un observable o di ricevere errori o notifiche di completamento, un observer deve prima "registrarsi" a quel dato observable con questo operatore

Descrivere che cosa si intende per higher-order observables? In quali casi si ha la necessità di gestirli? Fornire almeno un esempio con relativo codice e spiegazione dello scenario e del funzionamento.

Higher-order observables sono observable i cui valori sono essi stessi observables (eg. simile a lista di liste), outer observables i cui valori sono inner observables.

L'operatore mergeMap fa praticamente la flatMap su un observable high-order, concatMap invece li rende sequenziali.

Essendo che con la mergeMap gli observables sono eseguiti in parallelo ma i risultati ritornano in qualsiasi order (eg. http result are in response not request order), esiste la forkJoin la quale riceve una lista di observables, li esegue (subscribe and run) in parallelo e solo quando ogni observable nella lista emette il valore questo emetto un singolo array contenente tutti gli observables completati. (eg. tutte le risposte alle http requests). Se ne fallisce uno ritorna subito.

```
getPeopleByIdsForkJoin(): Observable<Array<Character>>{
return forkJoin( [1,2,3,4].map(
    id => <Observable<Character>>
    this.http.get(`${this.apiurl}/people/${id}/`)
  ));
}
```

Illustrare le differenze tra Subject e Observer/Observable. In quale contesto sono utili le funzionalità offerte dal Subject? Fornire un esempio descrivendone anche il contesto di utilizzo all'interno di una applicazione Angular.

Subject (producer/consumer):

il motivo per usarli è per il multicast, un observable di default è unicast ovvero ognuno dei suoi subscribed observer possiede un'esecuzione indipendente dell'observable. I subjects sono simili agli eventEmitter, mantengono un registro di vari listeners: quando subscribe su un subject non invoca una nuova execution che consegni i dati ma semplicemente registra il dato observer in una lista di observers. Mentre gli observables producono solo dati, i subjects possono essere usati sia per produrre che per consumare i dati. Usandoli come consumer si può convertire gli observables da unicast a multicast.

Il Subject è allo stesso tempo observer e observable. Se ho bisogno di fare più subscribe uso il subject e faccio la subscribe sul suo observable usando asObservable().

I subject si dividono in tre categorie:

- BehaviorSubject

  richiede un valore iniziale ed emette il suo valore corrente ai nuovi subscribers, di conseguenza si può prendere sempre l'ultimo valore emesso. Una volta che giunge a completion non emette più il valore corrente.
- ReplaySubject

  fa il "replay" cioè emette valori vecchi ai nuovi subscribers, ha un buffer di valori ed emetterà quei valori immediatamente ai nuovi subscribers insieme ai nuovi valori ai subscribers già esistenti. (eg. messaggi di una chat)
- AsyncSubject

  emette solo l'ultimo valore alla completion. Può esser visto come una promise, da usare quando i risultati intermedi non sono necessari.

Esempio: guarda esempio della prima domanda.