


TS E ES6  

1 Dichiarazione e gestione delle classi in ES6, Arrow function e binding del this.
I metodi di javascript funzionale find e findIndex

DICHIARAZIONE E GESTIONE DELLE CLASSI:

All'interno della classi si possono definire sia proprietà sia metodi

Davanti alle Property occorre omettere sia var sia let

Davanti ai Method occorre omettere function

Sintassi:

```
class User {  
    name=""  
    constructor(){  
        console.log("Ciao")  
    }  
}
```

ARROW FUNCTION:

Le Arrow Function, basate sull'operatore lamda, sono nate in ES6 per

- semplificare la scrittura delle funzioni di callback all'interno di una classe,
- migliorare la gestione del this

Omette la parola chiave function, sostituita dall'operatore lamda

Sintassi:

```
var somma = (x, y) => {return (x + y)}
```

La peculiarità delle arrow function è che non ridefiniscono il this che mantiene
pertanto il significato

che aveva all'interno dello scope genitore a monte della arrow function medesima.

FIND INDEX E FIND:

findIndex restituisce la posizione della prima occorrenza che soddisfa la
condizione

(-1 in caso di record non trovato).

find restituisce l'intero record della prima occorrenza che soddisfa la condizione
(undefined in caso di record non trovato).

Esempio:

```
let pos = students.findIndex(function(student) {  
    return student.matricola == "AL636GZ";  
});  
student[pos].assente = true;
```


2 Introduzione a Type Script: Tipizzazione delle variabili, la keyword as, gli operatori Punto Interrogativo e Punto Esclamativo

TIPIZZAZIONE e INTRODUZIONE:

TypeScript è un linguaggio evoluzione di javascript, sviluppato principalmente da Microsoft che, rispetto al semplice javascript, presenta la possibilità di tipizzare le variabili con ':' seguito dal tipo di tipizzazione, e sono: any, string, number, [], enum, {}, boolean, bigint, binary

LA KEYWORD AS:

Consente di assegnare un nuovo tipo ad una variabile, in modo da eseguire un cast al volo su variabili aventi un tipo differente da quanto atteso. Ad esempio, si supponga di avere una function elabora() che si aspetta come parametro un number :

```
let code : string = '123'; // proveniente ad es da un textBox
elabora(code as number);
```

OPERATORE PUNTO INTERROGATIVO E PUNTO ESCLAMATIVO:

In TypeScript una variabile tipizzata NON può assumere il valore undefined, a meno che non implementi a sua volta anche il tipo undefined:

```
let myVar : string | undefined
```

In alternativa al pipe undefined, si può usare il punto interrogativo:

```
myVar?:string
```

? in coda ad un parametro significa che quel parametro è facoltativo (javascript tradizionale)

```
function getName(std?: Student){ }
```

Il punto esclamativo sostanzialmente "toglie" il tipo null o undefined dalla variabile e garantisce al compilatore typescript che abbiamo la certezza che la variabile non può essere null o undefined:

```
let s: string | null;
console.log(s.toUpperCase())
```

diventa:

```
console.log(s!.toUpperCase())
```


Node 

1 Spiegare che cosa è la piattaforma mean

Nodejs è la base di una piattaforma denominata MEAN (Mongo, Express, Angular, Node) mirata alla realizzazione di applicazioni web di tipo client server

interamente basate sul linguaggio JavaScript, arricchito con diverse funzionalità server side.

Mongo = database

Express = web server

Angular = client side

Node = server side engine

Caratteristiche:

- Completamente basato sulla programmazione asincrona
- Ottime prestazioni (sia su server di piccole dimensioni sia su server di grandi dimensioni)
- Possibilità di interfacciamento con i socket a livello 4 della pila ISO OSI

In node.js tutte le principali attività vengono gestite in modo asincrono. Ad esempio

- L'accesso alle risorse del sistema operativo (come ad esempio l'accesso ad un dbms o l'accesso ad un file), in modo molto diverso rispetto ai web server classici in cui ogni operazione viene trattata in modo sincrono eventualmente sospendendo il thread durante l'attesa
- La gestione delle richieste di rete in cui ogni richiesta viene gestita mediante una istanza di una apposita funzione di callback associata alla richiesta stessa.

Anche questo è profondamente diverso rispetto ai web server tradizionali in cui il server esegue un ciclo di attesa su una specifica porta http e, per ogni richiesta, viene recuperato un thread da un pool di thread in attesa, che si occuperà di gestire la comunicazione con il client. Nella programmazione multithread tradizionale i vari thread vengono messi "in wait" tra una richiesta all'altra, e di fatto rimangono nell'elenco dei processi "idle" da gestire da parte del sistema operativo.

2 Spiegare sintassi e funzionamento del metodo readFile di nodejs sia nella versione sincrona che in quella asincrona.

Il metodo readFile legge e restituisce l'intero file ricevuto come parametro.

```
var fs = require('fs');  
fs.readFile(path, 'utf8', function (err, data){});
```

I metodi del modulo fs accettano sia path relativi sia path assoluti.

È analogo al precedente ma sincrono, cioè bloccante fino al termine della lettura.

```
var data = fs.readFileSync("./myFile", "utf8");
```

Se il file non esiste, readFileSync produce un errore sul server, errore gestibile solo ed esclusivamente mediante un TRY / CATCH.

```
fs.existsSync("./myFile"); Restituisce true se il file esiste, altrimenti false
```


3 Spiegare che cosa è e come opera npm. Spiegare il significato del file package.json e dei comandi npm init e npm install. L'opzione -g. Le devDependencies

COS'È E COME OPERA NPM:

Node Package Manager è un Gestore di Pacchetti java script (installato automaticamente insieme a nodejs) che gestisce il repository npmjs.com (simile a github) dove sono disponibili migliaia di moduli aggiuntivi javascript, fra i quali anche jquery e bootstrap. Chiunque può pubblicare sul repository npm un modulo di sua produzione che ritiene possa essere riutilizzato da altre persone.

Il modulo viene installato all'interno della cartella corrente in una sottocartella node_modules. Se il modulo è già presente all'interno di node_modules, viene sovrascritto dall'ultima versione.

SIGNIFICATO DEL FILE package.json

A monte della cartella node_modules dove npm installa tutti i moduli relativi al progetto, occorre creare un file package.json contenente principalmente l'elenco dei moduli utilizzati dal progetto, ciascuno con il proprio numero di versione. Elenco che rappresenta le dipendenze del progetto, cioè l'elenco dei moduli aggiuntivi necessari per il funzionamento del progetto. Nel momento in cui si pubblica un progetto su un repository le librerie NON vengono pubblicate. Quando qualcuno scaricherà il progetto potrà poi installare (comando npm install) tutte le librerie necessarie indicate all'interno del file package.json.

Per impostare il file bisogna eseguire sul cmd il comando 'npm init'

NPM INIT E NPM INSTALL

Il comando npm init (sostanzialmente analogo a git init) provvede ad inizializzare la cartella di lavoro come cartella NPM creando il file package.json. Quando si lancia npm init non viene aggiunta nessuna dipendenza.

Una volta creato il file package.json tutte le volte che si installa una nuova libreria il file viene automaticamente aggiornato. Viceversa i comandi npm install (o npm upgrade) utilizzati senza parametri provvedono ad installare in locale (nella sottocartella node_modules del progetto corrente) tutte le dipendenze indicate all'interno del file package.json non ancora presenti nel progetto. La differenza consiste nel fatto che npm install, in caso di libreria già esistente, non fa nulla mentre npm upgrade controlla anche le versioni, e se la libreria richiesta è presente sulla macchina ma in una versione più vecchia rispetto a quanto indicato nel file package.json, la aggiorna automaticamente.

All'interno di package.json è anche possibile aggiungere un campo devDependencies(development dependencies), cioè dipendenze che NON andranno a fare parte della build finale ma che sono necessarie in fase di sviluppo. Per aggiornare l'elenco delle devDependencies occorre utilizzare l'opzione npm install --dev. Ad esempio cordova.

4 Utilizzo di Type Scrypt in una applicazione node Caratteristiche e Funzionalità delle librerie typescript, ts-node e nodemon Creazione ed esposizione di un nuovo modulo ES6 in TypeScript

Per poter utilizzare typeScript in una app node occorre installare 2 librerie che devono essere installate:

- sia in globale in modo che siano visibili ed utilizzabili dal prompt dei comandi.

- sia in locale in ottica di una successive pubblicazione su Heroku

```
npm install -g typescript ts-node
```

```
npm install typescript ts-node
```

typescript è una libreria contenente un compilatore denominato tsc per la traduzione del codice TypeScript in javascript.

ts-node è una libreria che consente di avviare in esecuzione direttamente un file TypeScript.

In realtà esegue una compilazione in RAM e poi lancia node sul file compilato, esattamente come nel caso precedente. Occorre quindi avere sempre il compilatore installato.

```
ts-node server.ts
```

NODEMON:

La libreria nodemon, basata su ts-node consente anch'essa di avviare un server 'node' su un file scritto in TypeScript (o eventualmente anche js) però in modalità live reload cioè che riavvia automaticamente il server dopo ogni modifica sul codice. Installazione da fare sia in locale che globale.

```
nodemon server.ts
```

- Aprire il menù Run / Add Configuration scegliendo come ambiente di lavoro Nodejs. Viene creato un file launch.json contenente la configurazione di un progetto base nodejs

- Riaprire una seconda volta il menù Run / Add Configuration scegliendo questa volta come ambiente di lavoro Node.js: Nodemon Setup (che il primo giro non era disponibile).

All'interno del file launch.json viene aggiunta (in testa) una seconda configurazione per l'utilizzo di nodemon.

- Cancellare la configurazione relativa a nodejs (presente come seconda all'interno del file).

CREAZIONE ED ESPOSIZIONE DI UN NUOVO MODULO ES6:

Per la gestione dei moduli in nodejs esistono due standard differenti:

- CommonJS caratterizzato dall'utilizzo dei metodi module.exports e require()
- ES-modules (standard ES6) caratterizzato dall'utilizzo dei metodi import ed export

Node nasce utilizzando lo standard CommonJS (CJS), per cui tutte le librerie più vecchie scritte per Nodejs utilizzavano lo standard CommonJS.

TypeScript invece utilizza come default lo standard ES-modules (ES6). Le 'vecchie' librerie CommonJS sono utilizzabili anche con il nuovo standard ES6 ma occorre installare appositi adapter aggiuntivi.

Per esportare una variabile, una funzione o una classe, è sufficiente aggiungere l'istruzione `export` davanti al nome dell'oggetto da esportare

```
export let months = ['Jan', 'Feb', 'Mar', 'Apr', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];
```

```
import { months } from "../modulo" // user library senza .ts
```

Un modulo può esporre un unico 'export default' che non può essere associato ad una costante o ad una variabile, ma soltanto ad una funzione oppure, meglio, ad una classe.

5 Spiegare il funzionamento del personal dispatcher realizzato manualmente con node.js

Il dispatcher rappresenta l'interfaccia di front-end di un web server, cioè il modulo che si prende in carico le Http Request e ritorna al chiamante le corrispondenti Http Response.

In italiano significa letteralmente centralino o smistatore, cioè un componente che permette di differenziare le chiamate effettuate dall'utente, effettuando la selezione sulla base della risorsa richiesta

In node.js il dispatcher può essere richiamato all'interno della funzione di callback del metodo `createServer()` dell'oggetto `http` passandogli i parametri `request` e `response`.

Per ogni richiesta ricevuta, il Dispatcher dovrà prendere in considerazione le solite tre informazioni :

- Metodo usato per la richiesta
- Risorsa richiesta
- Eventuali Parametri aggiuntivi

La proprietà `listeners` è un JSON di JSON preposto a contenere un elenco di `listeners` che dovranno servire le varie richieste provenienti dal client.

La proprietà `listeners` contiene al primo livello 5 chiavi relative ai 5 principali metodi http (GET, POST, DELETE, PUT PATCH).

Ad ogni metodo http è associato un JSON interno contenente un elenco di `listeners` in cui:

- la chiave del listener è rappresentata dalla risorsa da servire,
- il valore è rappresentato dalla funzione di callback da eseguire in corrispondenza della richiesta di quella risorsa

La procedura `init()` è una procedura che, in fase avvio, carica in memoria la pagina di errore.

Il metodo invocato (GET o POST) viene di solito restituito dal browser in MAIUSCOLO.

Il metodo `addListener` dovrà essere invocato più volte dal main in fase di avvio in modo da registrare i vari `listeners` che dovranno rispondere alle diverse chiamate

relative ai servizi

Il metodo `dispatch` provvede a ricercare ed eseguire il listener corrispondente alla richiesta ricevuta.

Riceve i soliti due parametri `Request` e `Response` e li passa al listener corrispondente il quale provvederà ad inviare la risposta al client.

Per le richieste relative a pagine statiche, è sufficiente cercare la risorsa all'interno del File System, leggerla mediante `readFile` e restituirla al chiamante (function `staticListener`).

6 Spiegare in quale momento e in che modo in nodejs si può eseguire la lettura dei parametri GET e POST. Spiegare inoltre che cosa cambia nel caso dei parametri DELETE PUT PATCH

I parametri GET vengono accodati alla URL in formato `url-encoded`, i parametri POST vengono inseriti nel body della http request in formato json serializzato.

Problematiche relative ai parametri `url-encoded`:

- Impostando `contentType=url-encoded` è possibile passare a `$.ajax()` anche oggetti json che verranno automaticamente convertiti da `$.ajax()` in `url-encoded` (sia in GET che in POST).

Però questi json non possono contenere al loro interno altri oggetti annidati.

Impostando invece `contentType=json`, diventa possibile utilizzare oggetti annidati.

In realtà volendo trasmettere oggetti json annidati in `url-encoded`, è comunque possibile utilizzare lato client il metodo `$.param(obj)` che esegue la conversione da json a `url-encoded` applicando la codifica esadecimale `urlencode`

- Nel caso di parametri vettoriali, se più variabili hanno lo stesso nome (es `chkHobby`) l'ultimo valore sovrascrivebbe i precedenti. In tal caso occorre assegnare al nome del controllo (o in genere al parametro `url-encoded`) un nome vettoriale (es `chkHobby[]`). In tal caso lato server il parsing del parametro `url-encoded` restituisce un unico parametro (avente sempre nome vettoriale `chkHobby[]`) con all'interno tutti i valori presenti separati da virgola.

Non solo, ma se si assegna ad un parametro `url-encoded` un valore vettoriale, automaticamente il protocollo http, in fase di richiesta, aggiunge le parentesi quadre in coda al nome del parametro.

Passando invece i parametri in formato JSON (in modalità POST), questo problema non si pone !!

Cioè si può normalmente assegnare un vettore come valore di una qualsiasi chiave senza doverla necessariamente terminare con le `[]`

LETTURA PARAMETRI GET

I parametri GET possono essere letti direttamente all'interno dell'oggetto `request.url`.

A tale fine occorre parsificare l'oggetto `request.url` e poi leggere i parametri all'interno del campo `.query` che restituisce, in format json, l'elenco completo di tutti i parametri GET.

POST DELETE PUT PATCH

La lettura dei parametri POST è più complessa rispetto alla lettura dei parametri GET.

I parametri POST, PUT, PATCH e DELETE vengono infatti inviati al server all'interno del body della HTTP request e vengono ricevuti dal server DOPO che è arrivata la richiesta ed è stato lanciato il dispatching, per cui il listener li vedrebbe "undefined".

Occorre quindi ritardare l'esecuzione del dispatching soltanto DOPO che i parametri del body sono stati completamente ricevuti e memorizzati.

Al precedente metodo dispatch() si assegna il nome innerDispatch() e viene creato un nuovo metodo dispatch() indicato di seguito. Il metodo innerDispatch deve però continuare a far parte della classe perché accede alle property listeners e prompt.

Mongo ♀ ♂

1 Enunciare il CAP Theorem e spiegare la sua correlazione con l'avvento dei database no sql

CAP THEOREM:

Nel 2000 Eric Brewer ha dimostrato un teorema fondamentale relativamente ai vincoli di utilizzo dei database in ambito distribuito.

Ogni lettera di CAP sta a rappresentare una caratteristica che un sistema distribuito (SQL o noSQL) deve avere :

- Consistency (Consistenza): dopo qualsiasi modifica, tutti i nodi del sistema distribuito devono riflettere le modifiche allo stesso modo. Ogni server deve restituire lo stesso dato più recente, ovvero proveniente dall'ultima scrittura dello stesso, oppure un messaggio di errore.

Ogni client che interroga un qualsiasi cluster del sistema distribuito ha la stessa "vista" dei dati.

- Availability (Disponibilità): Un server deve sempre dare una risposta ad ogni richiesta ricevuta da un client. In altre parole il server deve essere sempre disponibile e non andare in stand-by perché sta facendo altro. Una risposta troppo ritardata è negativa quanto una risposta non data.

- Partition Tolerance (Tolleranza al Partizionamento): I dati devono essere sufficientemente replicati tra i nodi in modo tale da garantire una risposta anche in caso di interruzione di una connessione tra i server. Un sistema tollerante alle partizioni deve poter sopportare qualsiasi problema di rete tra i vari server. Nei sistemi distribuiti moderni, il Partition Tolerance non è un'opzione, ma una necessità.

Il CAP theorem afferma che questi tre vincoli, in un database distribuito, NON possono essere realizzati contemporaneamente.

Secondo il teorema, se ne possono realizzare contemporaneamente soltanto 2, potendo scegliere quali prediligere in base alle necessità.

AVVENTO DEI DB NO SQL:

L'idea di base dei DB noSQL è quella di rinunciare a qualche vincolo e ridurre il

costo di gestione delle join in modo da ottenere prestazioni migliori e maggiore scalabilità in ambito distribuito.

La dimostrazione del CAP Theorem ha fornito una spinta fondamentale ai database noSQL che lo hanno subito abbracciato puntando sulla velocità di funzionamento e scegliendo di prediligere i fattori AP a scapito della Consistenza.

Il DB noSql diventa dunque preferibile quando si hanno grandi quantità di dati e/o si richiede al sistema elevata reattività. La prima grande società ad utilizzare un DB noSQL in ambito distribuito è stata Amazon nel 2002.

Riducendo il numero di vincoli (si pensi alle imposizioni delle forme normali) potrebbero crearsi problemi di inconsistenza inammissibili nei DBSQL.

Tutto l'eventuale controllo della consistenza è demandato al livello applicativo trasformando l'NRDMBS in uno storage altamente performante ma "poco intelligente".

2 Caratteristiche e Struttura di un DB NO-SQL

CARATTERISTICHE:

Il modello noSQL basato sugli oggetti, è molto più vicino alla programmazione OOP di quanto non lo sia il modello relazionale. I dati sono salvati direttamente come oggetto e quindi le risposte sono molto più rapide. Viceversa nel caso relazionale occorre eseguire continue ed onerose conversioni tra recordset e oggetti e viceversa.

- Nei DB NoSQL cade il concetto di schema. I database relazionali nascono negli anni 70 con l'obiettivo primario di ridurre al massimo l'occupazione spaziale dei dati e quindi la potenzialità dell'hardware sottostante, da cui la strutturazione molto rigida delle tabelle. Nei DB NoSQL non ci sono strutture. Una collezione può contenere documenti strutturalmente anche molto diversi fra loro.

- Nei DB NoSQL non esiste il concetto di relazione con i relativi vincoli di integrità referenziale. e, di conseguenza, decade anche il concetto di join, operazione onerosa che consiste in un incrocio di tabelle in modo da restituire all'interno di un unico recordset tutte le informazioni necessarie. Nei db noSql, in caso di informazioni memorizzate su più collezioni differenti, occorrerà eseguire query consequenziali dove ogni query utilizzerà le chiavi restituite dalla query precedente.

- La mancanza dell'integrità dei dati rappresenta il limite principale dei database noSQL. Poco adatto per quelle applicazioni in cui l'integrità dei dati è basilare, anche a discapito della Availability e della Partition Tollerance. Ad esempio nelle applicazioni di tipo bancario in cui è preferibile una risposta consistente anche a costo di dover aspettare un tempo più lungo

- I db noSQL rendono più difficoltosa l'analisi statistica ed i processi di migrazione dei dati da un database all'altro.

STRUTTURA:

Rispetto agli SQL cambia anche la terminologia, Ogni database è organizzato in più collezioni.

Una collezione è una raccolta di documenti che sono semplicemente dei JSON.

Ogni documento, come tutti i JSON, è strutturato su più chiavi.

Ogni chiave a sua volta può contenere un altro JSON Object o un vettore enurativo.

Riassumendo:

tabella -> collezione
record -> documento di tip
campo -> chiave del JSON

3 Spiegare la sintassi delle Query di selezione (metodo find()) di mongodb e fornire un esempio relativamente al DB degli unicorni. Concetto di projection. findOne() e distinct()

db.name.find()

utilizzato senza parametri restituisce un recordset contenente tutti i documenti presenti nella collezione.

Questo recordset può essere visualizzato cos'è come è, oppure si può applicare il metodo .toArray() che restituisce i record sotto forma di vettore enumerativo di object.

La condizione where viene impostata tramite un UNICO json del tipo {campo:valore} in cui ogni campo di filtro rappresenta una chiave del json.

Chiavi e valori sono ENTRAMBI case sensitive.

\$lt, \$lte, \$gt, \$gte, \$ne esprimono le condizioni dei filtri.

'projection' consente di specificare l'elenco dei campi da visualizzare. Non è possibile impostare alcuni campi a 1 ed altri a 0.

In alternativa a find() si può utilizzare il metodo findOne() che ottimizza l'esecuzione di query che restituiscono un solo record. Il metodo findOne():

- non ammette il metodo finale .toArray().
- non ammette il metodo .project(). Al suo posto si utilizza un secondo parametro projection
- La funzione di callback deve essere passata come terzo parametro dopo projection

Esempio di findOne:

```
db.unicorns.findOne({"name": "Aurora" }, {"projection":{"hair":1, "weight":1}},  
function(err, data) {  
});
```

Il metodo distinct() accetta due parametri:

1. Il campo da visualizzare in modo "distinct" (da passare come semplice stringa)
2. Un eventuale filtro di ricerca (json)

Es: Ricercare l'elenco di tutti i frutti amati dagli unicorni F, con ogni frutto riportato una sola volta.

```
db.unicorns.distinct("loves", {gender: 'f'});
```

Restituisce un vettore di stringhe (senza l'aggiunta del metodo .toArray() che non è supportato)

E' l'unico metodo che restituisce come risultato un vettore di stringhe e non un vettore di json

4 Spiegare la sintassi base del metodo `update()` di mongodb e spiegare il significato dei suoi tre parametri, in particolare `upsert`.

Utilizza due parametri:

- Il primo parametro rappresenta il filtro di selezione con la stessa sintassi del `find`
- Il secondo parametro rappresenta i campi da aggiornare
- È disponibile un terzo parametro di opzioni (come `es multi / upsert`)

Aggiungendo come terzo parametro l'opzione `{upsert:true}`, se il record da aggiornare non viene trovato viene automaticamente creato un nuovo documento con un suo `_id`:

```
db.unicorns.updateOne({name:'pippo'}, {$inc:{vampires:1}}, {upsert:true})
```

Se il documento con nome „pippo” non esiste, viene automaticamente creato con `vampires=1`

Se un'operazione di aggiornamento con `{upsert: true}` termina con l'inserimento di un nuovo documento, l'opzione `$setOnInsert`, consente di assegnare tutte le chiavi indicate come parametro.

5 Spiegare la differenza tra i metodi `find()` e `aggregate()` di mongodb e fornire un breve approfondimento su `aggregate` (metodi `match`, `group`, `project`)

Il metodo `aggregate()` consente di eseguire query di selezione più complesse (ad esempio con il `group by`), mentre `find()` esegue un singolo comando (costituito eventualmente da una AND di tante condizioni), il metodo `aggregate()` si aspetta come parametro un vettore enumerativo contenente un elenco ordinato di operatori di aggregazione (pipeline) che:

- vengono eseguiti sequenzialmente uno dopo l'altro nell'ordine in cui sono scritti
- ognuno opera a catena sul recordset restituito dall'operatore precedente

Gli operatori principali sono:

- `$match` applica un filtro sul recordset ricevuto ed utilizza la stessa sintassi di `find()`
- `$group` raggruppa i record sulla base di uno o più campi indicati.
- `$project` consente di realizzare una proiezione sui campi che dovranno essere utilizzati dall'operatore successivo. Accetta al suo interno l'utilizzo delle funzioni di aggregazione (come ad esempio `$avg()`) in modo da poter eseguire calcoli che coinvolgono più colonne oppure su campi singoli costituiti da vettori enumerativi
- `$addField` consente di aggiungere nuovi campi al recordset su cui agisce, campi che di solito contengono valori provenienti dall'elaborazione di altri campi già esistenti. Nell'esempio si suppone che `homework` e `quiz` siano vettori enumerativi numerici. `$addField: { totalHomework: { $sum: "$homework" }, totalQuiz: { $sum: "$quiz" } }` La sintassi è la stessa di `project`; però tutti i campi precedentemente esistenti rimangono presenti all'interno del dataset restituito;
- `$unwind` consente di „srotolare” un campo costituito da un vettore enumerativo

producendo un elenco di N documenti ciascuno contenente una sola delle voci presenti nel vettore.

- \$sort esegue un ordinamento sul recordset finale. Accetta come parametro un JSON avente come chiavi una o più delle chiavi restituite dall'operatore \$group
- \$limit e \$skip A differenza del find(), in cui viene eseguito SEMPRE prima lo SKIP, questa volta i due operatori vengono eseguiti nell'ordine in cui sono scritti:

{\$limit:100}, {\$skip:10} Ne prende 100 e poi elimina i primi 10 [10-100]

{\$skip:10}, {\$limit:100} Ne esclude 10 poi prende i primi 100 [10-110]

L' opzione \$match, se utilizzata DOPO \$group, agisce sui gruppi restituiti da \$group, realizzando di fatto il tipico costrutto having L'opzione \$group a sua volta può essere richiamata una 2° volta per raggruppare ulteriormente i gruppi creati dal primo \$group ed eventualmente filtrati da \$match e \$project.

6 Indicare le principali Linee Guida per la progettazione di un DB NO-SQL:
embedding and referencing

Nei DBMS NoSQL scompare il concetto di relazione e la modellizzazione dei dati può seguire due direttive principali: Embedding (integrazione) or Referencing (riferimento)

- Embedding: significa annidare un oggetto all'interno di un altro, cioè in pratica inserire tutte le informazioni necessarie all'interno di uno stesso documento: se un cliente ha più numeri di telefono, integro tutti i numeri di telefono all'interno dello stesso documento.
- Referencing (o linking): più simile alle relazioni dei RDBMS, e consiste nel creare collezioni separate e fare in modo che un documento contenga, tra i suoi dati, gli ID di tutti i documenti collegati.

L'indicazione generale è quella di integrare tutto dove possibile.

- L'integrazione (EMBEDDING) rappresenta il vero punto di forza dei DB noSQL. Il sistema diventa molto più performante. Quando tutti i dati sono nello stesso documento le query sono molto più veloci perché non occorre andare a cercare i dati in posti diversi..

- In un modello SQL le relazioni più frequenti sono sicuramente le 1:N. EMBEDDING significa sostanzialmente integrare la tabella N all'interno della tabella principale 1 (il contrario non avrebbe nemmeno senso perché i dati della tabella 1 dovrebbero essere replicati all'interno di ogni record della tabella N)

Il referencing diventa preferibile quando:

1. il documento annidato (elemento N) cresce di dimensione in maniera sproporzionata rispetto a quello che lo contiene (elemento 1).
2. la frequenza di accesso ad una delle due tabelle è molto superiore rispetto all'altra (ad esempio se le fatture sono utilizzate molto più spesso rispetto all'anagrafica dei clienti a cui la fattura è destinata)
3. le due visualizzazioni sono indipendenti, nel senso che quando visualizzo le fatture non intervengono le informazioni anagrafiche del cliente e viceversa
4. per realizzare strutture complesse non rientranti tra quelle consigliate per l'embedding

Espress ☹☹

1 Spiegare come express gestisce la scansione delle routes ed il significato del parametro next

Nel linguaggio express i listener vengono indicati come routes.

Per ogni route occorre definire :

- route method (get, post, put, delete, etc)
- route path (la risorsa a cui rispondere)
- route handler (le funzioni di callback da eseguire come risposta alla richiesta)

In corrispondenza dell'arrivo di una richiesta, express scorre sequenzialmente l'elenco delle routes fermandosi automaticamente alla prima route che va a buon fine.

`app.get(resource, callback(req, res, next))`

Crea un listener per la gestione della risorsa GET indicata.

`app.post(resource, callback(req, res, next))`

Crea un listener per la gestione della risorsa POST indicata.

`app.put(resource, callback(req, res, next))`

Crea un listener per la gestione della risorsa PUT indicata.

`app.use(resource, callback(req, res, next))`

Crea un listener per la gestione della risorsa indicata, listener che verrà eseguito sempre, indipendentemente dal metodo utilizzato per la richiesta (get, post, put, delete, etc)..

In tutti i metodi il parametro resource può assumere il valore "*" che ha il significato di "qualsiasi risorsa". In pratica il listener viene eseguito in corrispondenza di qualsiasi richiesta.

Nel momento in cui una route va a buon fine, express la esegue ed interrompe la ricerca a meno che, al termine della procedura, non venga richiamato il metodo `next()`, quello passato come terzo parametro (opzionale) che è un puntatore ad una funzione interna di express che fa ripartire la scansione delle routes a partire dalla route successiva.

Le routes che utilizzano un `next()` finale, nel linguaggio di express sono indicate come middleware (codice intermedio). Le funzioni middleware sono normalmente associate a routes aventi come risorsa "*", che dunque saranno eseguite in corrispondenza di ogni richiesta, ad esempio per stampare dei log oppure per eseguire altre elaborazioni.

Se la scansione arriva al termine del file senza eseguire nessuna `end()/send()`, express invierebbe automaticamente un messaggio di errore del tipo : `Cannot GET/POST resource`.

2 Spiegare le differenze tra i metodi `app.use()` e `app.get()` (tre differenze)

Mentre lo scopo di `app.get()`, `app.post()`, etc è quello di rispondere alle richieste del client, lo scopo di `app.use()` è quello di eseguire codice middleware, che solitamente deve essere eseguito indipendentemente sia dal metodo di richiesta, sia dalla risorsa richiesta, per cui:

- `app.use()` risponde a su tutti i metodi di chiamata (`get`, `post`, `put`, `delete`, `patch`) mentre gli altri rispondono soltanto sul metodo indicato
- `app.use()` a differenza di `app.get()` e `app.post()` che eseguono un match perfetto sulla risorsa richiesta, `app.use` non esegue un match esatto sulla risorsa richiesta, ma va a buon fine anche nel caso in cui la richiesta riguardi una qualsiasi sottorisorsa. Ad esempio :

`app.use('/apple', ..)` will match `„/apple'`, `„/apple/img'`, `„/apple/img/news'`

In quest'ottica la richiesta `app.use('/')` va sempre a buon fine e, nel caso di `app.use()`, diventa quasi equivalente a `"*"`. L'unica piccola differenza (come spiegato dopo nel Mount Point) è che usando `"*"` la chiave url risulta vuota, per cui è preferibile utilizzare `"/"`

- `app.use()` consente di omettere la risorsa. Il valore di default infatti è `"/"` che significa tutte le risorse. In `app.get()` e `app.post()` la risorsa NON PUO' ESSERE OMESSA. Al limite si usa un `"*"`.

3 Spiegare come express gestisce l'accesso alle risorse statiche

I file statici (immagini, css, js, `favicon.ico`) sono in genere gestiti all'interno di una cartella il cui nome può essere impostato a scelta dal programmatore (es `public` o `static`) ed indicato ad express tramite il middleware `express.static` che accetta indifferentemente sia path assoluti che relativi.

Le seguenti righe sono del tutto equivalenti :

```
app.use('/', express.static('./static') );
app.use('/', express.static('static') );
app.use('/', express.static(".")); // file client nella stessa cartella del server
app.use('/', express.static(__dirname + '/static') );
```

Il metodo `express.static()` provvede a verificare se, all'interno delle cartella indicata, esiste la risorsa richiesta.

- Se esiste, la restituisce al client senza eseguire il metodo `next`.
- Se invece non esiste esegue il metodo `next()` e prosegue nella scansione delle routes dinamiche

4 Spiegare come Express restituisce i parametri GET e POST all'applicazione server. Spiegare anche il passaggio dei parametri come parte della risorsa

PARAMETRI GET:

I parametri GET possono essere letti mediante `req.query`.

Attenzione però che `request.query` parsifica soltanto i parametri ricevuti in `url-encoded`.

Si assume infatti che i parametri GET siano sempre passati in modalità `url-encoded`.

PARAMETRI POST:

Per leggere i parametri POST occorre preventivamente installare il modulo `body-parser`.

La lettura dei parametri può poi essere eseguita attraverso le seguenti routes middleware che entrambe restituiscono i parametri in formato json all'interno di `req.body`

```
app.use(bodyParser.urlencoded({limit:'50mb', extended:true}));
app.use(bodyParser.json({limit:'50mb'}));
```

L'opzione `limit` serve per l'upload di immagini base64 all'interno dei parametri post

L'opzione `extended:false` consente di passare come parametri `url-encoded` stringhe, vettori e anche oggetti json semplici ma non annidati

L'opzione `extended:true` consente anche l'utilizzo di oggetti estesi (json annidati)

PASSAGGIO DEI PARAMETRI COME PARTE DELLA RISORSA

La tendenza attuale è quella di passare i parametri GET non accodandoli alla risorsa nel formato tradizionale della `queryString` `?nome=valore`, ma di inserirli direttamente all'interno della risorsa richiesta separandoli semplicemente tramite SLASH. Dovrà essere il codice server a gestire questi parametri in modo diverso, andando a leggerli direttamente all'interno della risorsa richiesta.

Supponendo di voler inviare al server la seguente richiesta GET:

```
http://localhost:1337/api/risorsa?nome=pippo&eta=16
```

i due parametri `nome` e `eta` possono essere direttamente accodati alla risorsa in forma anonima:

```
http://localhost:1337/api/risorsa/pippo/16
```

Il parametro passato in questo modo non sarà intercettabile all'interno della `queryString` tradizionale, ma occorrerà una apposita route scritta nel modo seguente:

```
app.get('/api/risorsa/:nome/:eta', function (req, res, next) {
  var nome = req.params.nome;
  var eta = req.params.eta;
  res.send('good morning ' + nome );
})
```

Questa tecnica vale non solo per chiamate GET, ma anche per tutte le altre chiamate.

5 Spiegare la gestione degli errori in express

Nel caso in cui durante l'esecuzione si verifichi un qualsiasi tipo di errore, il router automaticamente richiama uno specifico middleware caratterizzato da una firma avente non tre parametri (req, res, next), ma quattro (err, req, res, next). Notare che il gestore degli errori deve necessariamente avere tutti e quattro parametri; il parametro next NON può essere omissivo.

Il listener di errore dovrà essere posizionato alla fine dello script dopo tutti i listener di risposta.

Se il listener di errore non è presente, il server si blocca terminando la sua esecuzione.

Nel caso di metodi che restituiscono una promise (es `await client.connect()`), in caso di errore la promise genera l'evento `catch`. Se questo non è gestito ma è gestito solo l'evento `then`, questo evento non si genererà mai (e di conseguenza nemmeno l'`await`) per cui l'applicazione dopo un po' va in `timeout` (e quindi in errore) purtroppo però senza richiamare la route di errore.

Per cui, per fare le cose bene, il `catch` delle promise dovrebbe sempre essere gestito.

È anche possibile generare nuovi errori ed interrompere l'esecuzione passando un apposito oggetto `err` al metodo `next`. Da usare ad esempio in caso di parametri mancanti o di valori non validi.

```
var err = new Error("Bad Request. Manca il parametro NOME");
```

6 Spiegare che cosa si intende con il termine crud server

Le operazioni base che si possono eseguire sui record di un database relazionale sono tipicamente quattro: Create/Insert, Read (Select), Update e Delete operazioni solitamente indicate con l'acronimo CRUD.

I principali metodi con cui un client http può accedere ad una risorsa sono essenzialmente anche quattro:

GET, POST, PUT e DELETE.

Un server CRUD è un http server che, facendo riferimento sempre alla medesima risorsa, associa i quattro metodi HTTP con le quattro operazioni CRUD. Cioè il medesimo servizio si comporta in modo diverso a seconda del metodo con cui viene effettuata la chiamata:

GET -> Lettura di un record

POST -> Inserimento di un nuovo record passato come parametro

DELETE -> Cancellazione di un record il cui ID di solito viene passato come risorsa `/people/13`

PUT/PATCH -> Aggiornamento di un record il cui ID di solito viene passato come risorsa `/people/13`

Notare che questo principio è in contrasto con quello che è l'utilizzo comune dei metodi HTTP.

Molto spesso infatti il metodo GET viene utilizzato per eseguire qualsiasi tipo di

interazione con il server.

Ad esempio, in una applicazione Web, per l'inserimento di un nuovo record si può comunemente eseguire una richiesta di tipo GET su un URI

7 Spiegare come può essere eseguito l'upload di un file dal client al server, sia in formato binario che in formato base64.

Per intercettare il file lato server è disponibile la libreria `express-fileupload` che intercetta :

- sia il vettore di oggetti `FILE` che restituisce all'interno di `req.files`
- sia gli eventuali ulteriori parametri scalari che restituisce all'interno di `req.body`

Il parametro `myFile` restituito all'interno dell'oggetto `req.files` presenta il seguente formato:

- vettore enumerativo di `json` in caso di selezione di file multipli
- singolo `json` in caso di file singolo. (indipendentemente dalla presenza o meno dell'attributo `multiple`)

Salvataggio su disco di un file trasmesso in base64

Trasmettere un file in formato base64 è abbastanza pesante.

Prima di trasmettere bisognerebbe per lo meno eseguire un `Resize` proporzionato. La trasmissione di files base64 all'interno dei parametri `POST` va bene per trasmettere piccoli files che devono essere salvati come stringa all'intero del database.

In pratica prima di salvare su disco si elimina l'intestazione base64 dal file, lo si trasforma in un buffer binario ed infine lo si scrive su disco tramite il metodo `fs.writeFile`

8 Spiegare il meccanismo di gestione delle richieste extra domain (CORS Cross-Origin Resource Sharing)

Cross-Origin Resource Sharing

Per ragioni di sicurezza, storicamente i browser non potevano inviare cross-origin request, (dette anche extra-domain), cioè richieste verso origini diverse rispetto all'origine da cui è stata scaricata la pagina, dove per origine si intende l'insieme delle seguenti informazioni:

- protocollo utilizzato per l'accesso alla pagina (`http`, `https`)
- dominio da cui la pagina è stata scaricata (es `localhost`)
- porta di ascolto del server da cui la pagina è stata scaricata.

CORS (Cross-Origin Resource Sharing) è il meccanismo utilizzato dai server per decidere se accettare o meno le richieste extra-domain provenienti da un browser. In caso di richiesta ritenuta non valida, il server risponderà con un errore CORS ed il browser visualizzerà un messaggio del tipo "Blocked for CORS policy error".

Nel momento in cui il browser deve eseguire una `SAFE REQUEST`, automaticamente

aggiunge nella richiesta l'intestazione origin.

Il server può ispezionare il campo Origin e decidere se accettare o meno la richiesta sulla base ad esempio di una white list.

Se la accetta deve aggiungere all'interno della risposta una intestazione speciale Access-Control-Allow-Origin che può essere impostata in due modi, elencando più origini separate da virgola oppure * (tutte le origini).

Il browser funge da mediatore: ricevuta la risposta dal server controlla la presenza dell'intestazione Access-Control-Allow-Origin e verifica se l'origine corrente è presente fra le Allow-Origin.

In caso di match il browser restituisce i dati alla callback javascript, altrimenti genera un CORS Policy Error.

In caso di richiesta accettata, il browser può accedere soltanto alle seguenti HTTP RESPONSE SAFE HEADERS, considerate innocue dal punto di vista della sicurezza

Il tentativo di accesso a qualunque altra intestazione genera errore.

Per consentire a javascript di accedere anche ad altre intestazioni, il server lo deve scrivere esplicitamente settando l'intestazione

Access-Control-Expose-Headers. Ad es per consentire l'accesso all'header authorization il server dovrà impostare la seguente intestazione:

Access-Control-Expose-Headers = "Authorization"

UNSAFE Request

Nel caso delle Unsafe Request il browser non esegue direttamente la request ma prima invia una richiesta, chiamata "preflight request", per richiedere il permesso al server.

La metodologia per l'invio della main request è la stessa utilizzata per le safe request il browser può accedere soltanto alle HTTP RESPONSE SAFE HEADERS, considerate (le stesse di prima)

9 Spiegare i passi necessari per la creazione di un server HTTPS

Il protocollo HTTPS (basato su TLS/SSL) crea un canale cifrato tra client e server. Prevede un tipo di autenticazione detta unilaterale in cui solo il server è autenticato (il client conosce l'identità del server), ma non viceversa (il client rimane anonimo e non autenticato). Il client valida il certificato del server controllando

la firma digitale e verificando che questa sia valida e riconosciuta da una Certificate Authority conosciuta.

Il passo necessario per poter creare un server https è quello di creare una coppia di chiavi RSA:

- una chiave privata da tenere memorizzata sul server
- una chiave pubblica da distribuire tramite Certificato Digitale

controfirmato digitalmente da una Certification Authority.

Per il test delle applicazioni si possono creare semplici certificati self-signed utilizzando:

- OpenSSL da installare in locale

- RSA Key Generator 1024/2048 bit utilizzabile online

Per i server reali è possibile creare Certificati controfirmati da una Certification Authority tramite, ad esempio, letsencrypt.org (without shell access) ed utilizzando il cosiddetto manual mode

OpenSSL è una applicazione che consente di creare certificati self-signed ma anche controfirmati.

Il sito ufficiale openssl.org fornisce e mantiene soltanto la versione Linux.

Openssl in realtà viene già installato insieme a git, per cui se si ha git installato sul pc non c'è bisogno di ulteriori installazioni. In tutti i casi occorre però creare le seguenti variabili di ambiente (dove OPENSSL_CONF esiste solo nella versione full) :

PATH -> C:\programmi\git\usr\bin

OPENSSL_CONF -> C:\programmi\git\usr\ssl\openssl.cnf

A questo punto da qualsiasi cartella di lavoro si può lanciare il comando sotto indicato.

Se non si impostano le variabili di ambiente si può lanciare il comando direttamente dalla cartella BIN di openssl. E' anche possibile digitando solo openssl aprire un terminale all'interno del quale si possono inviare gli stessi comandi omettendo la voce iniziale openssl.

openssl req -x509 -days 365 -newkey rsa:2048 -nodes -keyout privateKey.pem -out certificate.crt

Per i soli certificati, esistono diversi siti di tipo SSL Certificate Decoder che consentono di visualizzare il contenuto del certificato (CRT, DER) con tutte le informazioni in esso contenute. In windows i files CRT possono essere aperti con "Estensioni della shell di crittografia" oppure con Internet Explorer.

CREAZIONE DI UN HTTPS SERVER TRAMITE EXPRESS:

```
const privateKey = fs.readFileSync("keys/privateKey.pem", "utf8");
const certificate = fs.readFileSync("keys/certificate.crt", "utf8");
const credentials = { "key": privateKey, "cert": certificate };
```

```
var express = require('express');
var app = express();
var httpServer = http.createServer(app);
```

```
httpServer.listen(8080);
var httpsServer = https.createServer(credentials, app);
httpsServer.listen(HTTPS_PORT, function(){
    console.log("Server in ascolto sulle porte HTTP:" + PORT + ", HTTPS:" +
    HTTPS_PORT)
});
```

I due server http e https possono essere in esecuzione contemporanea consentendo accesso http e https

Quando si usa https ricordarsi di aprire le CORS Policy ad https e aggiungere secure=true al cookie.

10 Spiegare che cosa sono i web socket ed il loro meccanismo di funzionamento. I metodi statici `io.sockets.emit()` e `io.to("room").emit()`

I web socket consentono la creazione di una connessione TCP tra due endpoint HTTP in modo da consentire al server di poter inviare al client messaggi asincroni come ad es le notifiche.

Due importanti caratteristiche:

- Il server può inviare notifiche al client senza che sia il client a richiederle
- La connessione TCP, lato server, “passa” sempre attraverso la porta 80 del web server, evitando così di dover aprire altre porte specifiche sul firewall di accesso al server.

Per prima cosa occorre creare un server HTTP standard da avviare al solito modo :

```
const http = require ("http");  
const httpServer = http.createServer(app);  
httpServer.listen(1337, function () { });
```

Lato server occorre quindi creare una istanza del Socket Server passando al costruttore l'oggetto `httpServer` su cui appoggiarsi.

```
import { Server, Socket } from 'socket.io'  
const io = new Server(httpServer);
```

La medesima libreria `socket.io` è disponibile lato client sul sito ufficiale `socket.io`

La richiesta di connessione deve partire dal client che farà uso del seguente metodo:

```
let serverSocket = io.connect();
```

che invia una richiesta di connessione al server `http` da cui è stata scaricata la pagina corrente.

Questo metodo è sincrono e restituisce il socket del server. A seguito di una richiesta di connessione, sia sul client che sul server viene comunque generato un evento `on("connect")` :

```
serverSocket.on('connect', function() {console.log("connessione ok")} )
```

L'accettazione della connessione da parte del server è automatica.

In corrispondenza dell'accettazione della connessione viene generato un evento `on("connection")`.

A questo punto, stabilita la connessione, sia il client che il server possono inviare dei dati sulla connessione mediante il metodo `emit()` che presenta due parametri entrambi di tipo stringa.

```
socket.emit(key, data);
```

Il primo parametro `key` è un identificativo che identifica univocamente il messaggio.

Il secondo parametro `data` rappresenta il dato vero e proprio che deve essere di tipo string.

Sull'host remoto, in corrispondenza di ogni emit(), viene generato un apposito evento identificato dalla stessa key. Quindi gli eventi saranno in ascolto sulla key definita dal mittente ed avranno come callback una funzione alla quale vengono iniettati i dati corrispondenti.

```
socket.on(key, function (data) { }
```

socket.broadcast.emit(key, data); Questo metodo consente al server di inviare dei dati in broadcast a tutti gli host attualmente connessi, con esclusione di quello identificato dal socket corrente.

io.sockets.emit(key, data); Metodo statico che consente al server di inviare un messaggio a tutti gli host attualmente connessi, mittente compreso.

La richiesta di chiusura della connessione può essere eseguita indifferentemente in qualsiasi momento sia dal client che dal server tramite il richiamo del metodo socket.disconnect();

Il server può gestire stanze differenti ed inoltrare i messaggi di un utente soltanto all'interno della stanza indicata. A tal fine:

- Il client, in fase di accesso alla chat, deve comunicare al server (con un normale messaggio di tipo STRING) la stanza all'interno della quale desidera entrare.

```
serverSocket.emit("room", "room1")
```

- Il server, ricevuta la richiesta di accesso, dovrà aggiungere il client all'interno della stanza utilizzando il metodo join:

```
clientSocket.join("room1")
```

A questo punto tutti i messaggi provenienti da quel client dovranno essere inoltrati dal server soltanto sulla stanza scelta. Il messaggio verrà inoltrato a tutti COMPRESO il mittente

```
io.to("room1").emit('key', message)
```

La stessa cosa può essere fatta a partire da un generico socket. In questo caso il messaggio verrà inviato a tutti i client presenti nella stanza, ESCLUSO il socket mittente

```
clientSocket.to('room1').emit('key', message);
```


JWT

1 Spiegare le ragioni che hanno portato dall'autenticazione "Server Based" basata sulle variabili Session all'autenticazione tramite token. Vantaggi dell'autenticazione tramite token.

Problemi legati alla Server Based Authentication

- Overhead: Ogni volta che un utente si è autenticato, il server deve creare un record relativo alla sessione. Questo di solito viene fatto in memoria e quando ci sono molti utenti che autenticano, aumenta l'overhead sul server.

- Scalability: Il fatto che le sessioni siano memorizzate in memoria, crea problemi di scalabilità. Se la session viene creata sul server di login, l'utente

non potrà inviare richieste ad altri server. I cloud provider di solito replicano i server per gestire il carico applicativo; il fatto di avere informazioni vitali nella memoria di sessione limitano la capacità di scalare. Se abbiamo creato la sessione quando eravamo sul server A e successivamente il bilanciatore di risorse ci porta sul server B, la sessione sul server A che aveva i nostri dati di autenticazione la perdiamo (in realtà per avere sessioni condivise su più server le sessioni non si memorizzano più localmente ma possono essere memorizzate all'interno di uno spazio condiviso su un database no-sql).

Vantaggi dell'autenticazione tramite Token

- Il meccanismo del token è completamente stateless, cioè le informazioni di autenticazione stanno direttamente nel token, evitando di dover passare dal database o di usare le sessioni per memorizzare le informazioni sull'autenticazione. Quindi non richiede la memorizzazione di alcuna informazione sul server. Inoltre se i server sono replicati è sufficiente che tutti utilizzino lo stesso tipo di token ed il problema è completamente risolto.
- Il token viene trasmesso nell'intestazione della richiesta. Se il token non è valido al dispatching della richiesta non si arriva nemmeno, per cui l'impegno del server è minimo.
- Lo stesso token può essere utilizzato anche per accedere a sistemi diversi rispetto a quello che lo ha generato. Meccanismo detto SSO Single Sign On utilizzato da google per uniformare gli accessi ai diversi servizi ma ormai utilizzato anche in grandi aziende che espongono molteplici portali relativamente a servizi differenti (spedizioni, logistica, etc).

2 Spiegare la struttura di un token JWT. con cenno ai metodi `jwt.sign(payload, key)` e `jwt.verify(token, key, callback(err, data))` e alle possibili modalità di trasmissione del token.

L'utente si autentica mediante username e password

- In caso di credenziali valide il server restituisce un token 'firmato' digitalmente
- Il client (mediante java script) memorizza il token e lo re-invia in ogni richiesta successiva
- In corrispondenza di ogni richiesta il server controlla il token e, se valido, restituisce i dati richiesti

Un token JWT è costituito da tre part:

1) header

E' un json contenente il tipo di token utilizzato (es JWT) ed il nome dell'algoritmo da utilizzare per la firma digitale (signature). Esempio:

```
{ "alg": "HS256", // SHA256  
  "typ": "JWT" }
```

Questo oggetto viene serializzato mediante una codifica encodebase64 che divide la sequenza binaria in gruppi di 6 bit creando in pratica quattro cifre ogni 3 caratteri. I caratteri utilizzati sono in tutto 64:

26 lettere minuscole, 26 lettere maiuscole, 10 caratteri numerici, + e / . In realtà JWT utilizza una variante che sostituisce + e / con - e _ Questa variante

viene utilizzata quando la stringa codificata deve essere usata in una URL o in un filename.

2) payload

Nel payload sono contenute le informazioni relative all'utente che si è appena autenticato (claims = dichiarazioni dell'utente). Esempio:

```
{ "id": "1234567890",  
  "username": "pippo@gmail.com",  
  "admin": false }
```

Il payload viene anch'esso codificato in encodebase64 ottenendo la seconda parte del token.

3) signature

In coda al token viene apposta una firma digitale (signature) realizzata utilizzando l'algoritmo indicato nell'header, ad esempio l'algoritmo HMAC SHA 256 che crea impronte di lunghezza 256 bit, cioè 32 caratteri, che diventano 43 in encodebase64.

La firma digitale viene creata nel modo seguente:

Si costruisce una stringa s costituita da:

```
string s = base64UrlEncode(header) + "." + base64UrlEncode(payload)
```

Si applica a questa stringa l'algoritmo SHA 256 utilizzando come chiave una chiave privata del server

```
string signature = HMACSHA256(s, key)
```

La signature così ottenuta viene codificata in base64 producendo il seguente risultato

Il token finale viene ottenuto con il concatenamento delle 3 stringhe precedenti suddivise da un puntino

Il server, in corrispondenza di ogni richiesta, dovrà verificare la presenza del token e la sua validità.

I passi da eseguire sono i seguenti:

- Sulla base dell'header e del payload ricalcola la signature utilizzando la propria chiave privata. Se la signature differisce da quella ricevuta significa che il token è corrotto (o modificato) e la richiesta viene rifiutata.
- Legge il payload verificando che la data di scadenza sia ancora valida
- Decide se eventualmente aggiornare il token

Il metodo `jwt.sign()` crea un token completo (header, payload, signature) memorizzando nel payload i campi dichiarati nel primo parametro. Va bene qualsiasi campo informativo (esclusa ovviamente la password) con l'aggiunta di eventuali „options” fra quelle indicate nella tabella precedente.

Il secondo parametro rappresenta la chiave privata da utilizzare per apporre la firma digitale.

Il metodo `jwt.verify()` estrae le prime due componenti del token, ricalcola la firma digitale utilizzando la chiave privata ricevuta come parametro, e confronta la firma ricalcolata con quella ricevuta restituendo `err = null` in caso di esito positivo del confronto.

In realtà `jwt.verify()` :

- Verifica anche la data di scadenza dl token, generando un errore in caso di

token scaduto.

- Restituisce all'interno del campo data il payload del token, rendendo di fatto inutile l'utilizzo del metodo `jwt.decode()`

Cordova

1 Spiegare architettura e funzionamento di cordova. Le varie fasi di creazione e build di un progetto.

ARCHITETTURA

L'architettura di Apache Cordova è strutturata mediante un web server locale creato da cordova il quale rende disponibili al browser locale (webView) le risorse memorizzate all'interno della cartella `www` della app. In corrispondenza del lancio della app viene sostanzialmente aperto il browser incorporato nel sistema operativo (detto webView) il quale invia una richiesta al server incorporato all'interno dell'app stessa. La richiesta inviata in fase di avvio è sempre la seguente:

`https://localhost/index.html` (porta standard 443)

dove in realtà sia il nome della pagina di apertura (`index.html`) sia il nome del server (`localhost`) sono entrambi configurabili all'interno del file di configurazione `config.xml`. `https` invece è fisso.

All'interno del progetto è possibile aggiungere dei plugin che consentono di wrappare tutte le funzionalità native del dispositivo, quali ad esempio fotocamera, accelerometro, bussola, GPS.

Sono cioè delle interfacce che wrappano al loro interno le API del SO. Queste API

- verso il basso parlano il linguaggio nativo

- verso l'alto parlano JavaScript, introducendo un livello di astrazione tale da consentire

l'accesso alle funzionalità native tramite semplici funzioni JavaScript. Il file `cordova.js` introduce un'ulteriore interfaccia astratta tra la app ed i plugin javascript.

Definisce ad esempio l'oggetto `navigator` di accesso ai plugin e la gestione dell'evento `deviceReady`. Il file `cordova.js` viene automaticamente creato al momento del build, però deve essere linkato manualmente in tutte le pagine che fanno uso di un plugin.

FASI DI CREAZIONE E DI BUILD DI UN PROGETTO

1) Creazione di un nuovo progetto

Comando `cordova create`:

`cordova create <cartella_nuovo_progetto> <nome_package> <nome_app>`

Il package serve per identificare univocamente la app all'interno del dispositivo. App differenti devono assolutamente avere nomi di packages differenti.

Come package si utilizza di solito il reverse-domain, cioè il nome di dominio

scritto a partire destra.

2) Creazione delle platform

Il comando `cordova platform add` (da eseguire dalla root del progetto corrente) specifica la piattaforma di destinazione per la quale si desidera creare l' APK. Modifica di conseguenza il file `config.xml` ed aggiunge diverse informazioni all'alberatura corrente.

In particolare nella cartella `platforms` viene aggiunta una nuova cartella preposta a contenere il progetto nativo per la piattaforma indicata.

Il comando può essere ripetuto più volte specificando platform differenti.

Al momento del build verranno create le PAK relative a tutte le piattaforme aggiunte.

```
cordova platform add android
```

3) Installazione dei plug-in

```
cordova plugin add *nome*
```

Quando si scarica un plug-in, questo viene automaticamente scaricato per tutte le platform per cui quel plug-in risulta disponibile. I plug-in vengono scaricati dal repository di GitHub.

La cartella plug-in funge in pratica da repository per l'applicazione in corso di sviluppo, dal quale attingeranno poi le varie platform. Non tutti i plug-in sono disponibili per tutte le platform.

Nella sottocartella `cordova-plugin-dialogs/src` viene creata una sottocartella per ogni platform per cui il plug-in risulta disponibile.

L'intera cartella plug-in può essere copiata da una macchina all'altra senza problemi. (occorre copiare anche il file `fetch.json`).

4) Build

Necessari:

- java JDK
- android SDK (software development kit)
- gradle compattatore che fisicamente andrà a creare l'APK (pacchetto compresso "simile" agli zip)
- Un eventuale emulatore per poter eseguire rapidamente il test della app (es Pixel 2)

Dopo aver fatto tutte le installazioni, occorre definire le variabili d'ambiente necessarie per indicare a cordova i path da utilizzare. A tal fine si può utilizzare un file `setEnv.bat` strutturato all'incirca come indicato di seguito. Questo file dovrà essere eseguito ogni volta che si apre un nuovo terminale. In alternativa queste stesse variabili possono essere impostate all'interno delle Variabili di Ambiente nel Pannello di Controllo di Windows

```
cordova build android // versione di debug
```

```
cordova build android --release // versione di relese
```

Se si omette `android` viene eseguito il build per tutte le piattaforme installate.

Specificando `android` viene eseguito soltanto il build per android. In corrispondenza del build parte il download di tutte le API necessarie alla realizzazione del progetto. Il download viene fatto dal repository di `maven.org`.

Le opzioni `--stacktrace` e `--verbose` aggiungono delle informazioni utili in caso di

errori.

Angular

1 principali caratteristiche di Angular2

- Angular2 è completamente centrato sul concetto di componente e presenta una spiccata modularità, basata sulla possibilità di poter creare nuovi componenti o utilizzare facilmente componenti di terze parti. Ogni componente contiene sia la grafica sia il corrispondente codice di elaborazione, per cui la pagina html principale risulta estremamente concisa e si limita a richiamare i vari componenti scritti separatamente.
 - mentre Angular.js era basato su javascript, Angular utilizza un nuovo linguaggio che si chiama TypeScript che è una astrazione molto più strutturata rispetto a javascript, cioè maggiormente orientato agli oggetti e che, tramite i tipi, consente un maggiore controllo sul codice.
 - Angular crea al suo interno una vera e propria applicazione che andrà a caricare via via le pagine html quando necessarie. Il comando `ng build` “compila” l'applicazione trasformando i files angular in files html / css / js tradizionali in modo che il browser li possa correttamente interpretare.
 - Il comando `ng serve` esegue un build temporaneo in memoria e, in più, avvia l'esecuzione di un server locale nodejs in esecuzione sulla porta 4200 denominato “Angular live development server” che invia al browser le pagine compilate che verranno via via richieste.
- Inoltre avvia un monitoraggio costante su tutti i file che compongono il progetto. Nel momento in cui si applica una qualsiasi modifica ad un qualsiasi file l'applicazione viene automaticamente ribuildata e ricaricata dal browser senza necessità di eseguire un refresh e soprattutto mantenendo lo stato corrente (modalità indicata come live-reload).
- Notare che Il refresh della pagina è di solito una operazione abbastanza onerosa perché comport il reinserimento di tutti i dati necessari per arrivare alla pagina di interesse.

Altre caratteristiche

- E' fortemente orientato al pattern MVC
- E' interamente scritto sulla base delle direttive di ECMAScript 2015 (ES6)
- E' di tipo mobile first, cioè garantisce elevate prestazioni per assicurare una interazione fluida sui dispositivi mobile. Supporta i principali eventi tipici del mobile (es touch e gesture)

2 Spiegare il concetto di componente e di decoratore. Cos'è appComponent

Un componente è una classe (con iniziale maiuscola) con associato un decoratore (messo a disposizione dal core) che la „trasforma" in componente. `app.component` è il componente principale visualizzato da `main.ts`.

Associati ad ogni componente ci sono sempre 4 files:

1)app.component.ts è il file più importante fra i quattro. Rappresenta il file Type Script all'interno del quale è definita la classe che realizza il componente

All'inizio viene importato il decoratore Component che è un oggetto definito all'interno della libreria @angular/core

- Il decoratore atComponent viene applicato alla classe AppComponent aggiungendo nuove caratteristiche alla classe stessa e trasformandola in un Componente Angular.

- Il decoratore atComponent assegna le tre caratteristiche fondamentali di un componente Angular:

- definisce un selettore che consentirà di utilizzare il componente all'interno di una qualsiasi pagina html.

- assegna al componente un file html associato in cui viene definita la struttura html del componente

- assegna al componente un file css associato inizialmente vuoto.

In realtà il link al file css è costituito da un vettore enumerativo in quanto è possibile associare ad una stessa classe più files css / scss

- L'oggetto AppComponent viene „esportato' tramite una istruzione export (ES6-modules) AppComponent è l'oggetto che viene mandato in esecuzione da AppModule.ts

- All'interno della classe AppComponent è possibile definire liberamente nuove Proprietà e nuovi Metodi personalizzati.

2)app.component.html memorizza il contenuto html relativo al componente. Si può provare ad esempio a cambiare la scritta “Welcome” in alto a sinistra. L'icona di twitter a destra è disegnata manualmente tramite un tag SVG. All'interno del file html si possono inserire normali tag html oppure altri componenti angular creati appositamente. Al fine di migliorare la portabilità è raccomandato l'utilizzo di soli componenti angular.

3)app.component.css E' la pagina in cui è possibile definire i formati di stile in formato css/scss

4)app.component.spec.ts Facoltativo, consente di definire dei test sull'applicazione

Un decoratore, introdotto dal simbolo @ (at), è un particolare metodo di classe che consente di aggiungere nuove caratteristiche all'oggetto a cui viene applicato. Si aspetta come parametro sempre un json che conterrà informazioni diverse a seconda del tipo di decoratore. (Es. @Input, @Output).

3. Spiegare il funzionamento delle direttive strutturali *ngIf e *ngFor ed i Template Reference

Le direttive strutturali consentono di inserire delle istruzioni Angular all'interno dei tag HTML, istruzioni che permettono di modificare il DOM a runtime aggiungendo o togliendo elementi.

In fase di creazione di un tag è possibile salvarsi un riferimento tramite l'utilizzo di un segnaposto (Template Reference), riferimento che potrà poi essere passato alla procedura di evento.

Per accedere in modo diretto agli elementi della pagina HTML esistono due appositi decoratori che si aspettano come parametro un segnaposto (Template Reference) tipo quello visto poc'anzi.

- @ViewChild restituisce il primo matching element all'interno della pagina
- @ViewChildren restituisce tutti gli elementi individuati come una QueryList of items

Esempio

Per accedere ad un textbox avente come segnaposto

```
<input type="text" #txtName >
```

occorre definire nel file di classe una variabile di tipo ElementRef (puntatore a oggetto che andrà a 'wrappare' l'oggetto html) preceduta dal decoratore @ViewChild

```
import { ViewChild, ElementRef } from '@angular/core';
```

```
@ViewChild("txtName") _txtName! : ElementRef;
```

cioè definiamo una variabile denominata _txtName di tipo ElementRef che punta all'elemento HTML avente come segnaposto "txtName".

A questo punto in qualunque evento si può utilizzare la variabile _txtName seguita dalla property nativeElement che consente di accedere all'oggetto html interno e quindi a tutte le sue proprietà.

```
this._txtName.nativeElement.disabled=true;
```

4. Binding unidirezionale di un attributo HTML tramite le parentesi quadre [].

Binding bidirezionale tramite la direttiva [ngModel]

Per collegare un attributo html al valore di una variabile typescript si può utilizzare il cosiddetto Property Binding che consiste nello scrivere il nome dell'attributo HTML tra parentesi quadre e poi assegnare come valore il nome della variabile typescript.

Le parentesi quadre stanno ad indicare che nella stringa di destra non è contenuto un valore diretto, ma una variabile della quale occorre leggere il contenuto.

Le seguenti due sintassi sono praticamente equivalenti:

```

```

```
<img [src] ="recipe.imagePath">
```

```
<input type="text" [value]="recipeName">
```

- Il Property Binding rispetto alle {{ }} è più generale (può essere applicato anche alle classi) e gestisce meglio le eccezioni (es img non trovata) per cui, a parità di condizioni, risulta preferibile.

- Il Property Binding può essere SOLO unidirezionale dal codice verso

l'interfaccia. NON può diventare bidirezionale aggiungendo delle semplici parentesi tonde all'interno delle parentesi

quadre. Le parentesi tonde possono anche essere inserite, ma il flusso rimane unidirezionale.

La direttiva ngModel consente di eseguire un binding bidirezionale tra una variabile typescript e la proprietà predefinita del controllo a cui viene applicata (value nel caso di un Text Box, checked nel caso di un checkbox o radiobutton).

```
<input type="text" [(ngModel)]="serverName">
```

Omettendo le tonde intorno ad ngModel , il binding diventa unidirezionale, dal

codice verso l'interfaccia e diventa equivalente al Property Binding eseguito tramite parentesi quadre [value]

Se si utilizza ngModel all'interno di un tag <form>, è necessario impostare all'interno dei tag che usano [(ngModel)] anche l'attributo name che viene utilizzato da Angular per gestire il collegamento tra l'attributo gestito da ngModel e la variabile collegata (vedasi Direttiva ngForm).

5. Binding delle proprietà di stile tramite le direttive di attributo [ngStyle] e [ngClass]

Il valore delle proprietà di stile può essere gestito tramite le seguenti

Direttive di attributo:

[ngStyle] (Direttiva di Stile)

[ngClass] (Direttiva di Classe)

Le direttive devono sempre essere scritte fra parentesi quadre perchè come valore a destra si aspettano sempre una variabile typescript oppure una funzione. Come sempre davanti al nome della variabile type script si può indifferentemente usare / non usare il this

[ngStyle]

Utilizzato all'interno di un tag html, consente di definire le sue proprietà di stile. All'interno di un tag ci può essere un unico [ngStyle] che si aspetta come parametro un json di stili in cui ad ogni property può essere assegnato il valore di una variabile oppure il valore restituito da una espressione booleana o da una funzione typescript.

```
<p [ngStyle]='{'backgroundColor': student.gender=='F' ? 'pink' : 'cyan',  
  'fontWeight' : nVulnerabilities > 10 ? 'bold' : 'normal'}" >
```

Nell'esempio alle due proprietà di stile viene assegnato un valore differente a seconda che la successiva espressione booleana di valorizzazione risulti vera oppure falsa

[ngClass]

Consente di assegnare una più classi al tag HTML nel quale viene inserita.

All'interno di un tag ci può essere un unico [ngClass] che si aspetta come parametro un json di classi in cui ad ogni classe viene assegnata una espressione booleana. Se l'espressione booleana risulta vera la classe viene applicata, altrimenti no.

```
<p [ngClass]='{'female': student.gender=='F',  
  'male': student.gender=='M',  
  'underlined': student.city=='Fossano'}" >  
<p [ngClass]='{'green':serverStatus=='online',  
  'red':serverStatus=='offline',  
  'blinking':nErrors >10 }" >
```

6. Descrivere la gestione degli eventi e l'accesso all'oggetto che ha scatenato l'evento

Per quanto riguarda gli eventi, è sufficiente scrivere il nome dell'evento all'interno di parentesi tonde:

```
<button (click)="assegnaNome()"> cliccami </button>
```

In corrispondenza di ogni click verrà richiamata la procedura "assegnaNome()" scritta nel file .ts.

Notare che la procedura deve essere scritta come stringa con le tonde finali, esattamente come si fa per richiamare una procedura javascript dall'interno di una pagina html: onClick = "assegnaNome()"

Si tratta infatti di un semplice richiamo diretto a procedura.

Il this non è utilizzabile in quanto gli eventi non sono associati tramite addEventListener(), ma sono semplicemente „richiamati" dall'html. Per accedere dalla procedura di evento al puntatore dell'elemento che ha scatenato l'evento ci sono due possibilità:

1) utilizzo di event.target

A tutti gli eventi javascript viene automaticamente iniettato un parametro event che al suo interno contiene una proprietà event.target che rappresenta il puntatore all'elemento che ha scatenato l'evento.

Il parametro event non ha altre proprietà particolarmente interessanti per cui, se si ha a disposizione il this, il parametro event perde di significato.

In Angular il passaggio del parametro event alla procedura di evento non è automatico ma, in fase di chiamata, occorre utilizzare la parola chiave \$event che indica ad Angular di propagare i parametri alla procedura di evento. In questo modo la procedura di evento riceverà il parametro event che potrà rinominare a suo piacimento (ed esempio e) ed utilizzare per accedere al parametro e.target

2) utilizzo di un segnaposto (Template Reference)

In fase di creazione di un tag è possibile salvarsi un riferimento tramite l'utilizzo di un segnaposto (Template Reference), riferimento che potrà poi essere passato alla procedura di evento.

7. Spiegare il Property Binding tramite il decoratore @Input() per passare un parametro da un Top Component ad un Bottom Component

Un Top Component (che sta sopra), nel momento in cui utilizza all'interno del proprio file html il selettore di un Bottom Component (Bottom Component che quindi vivrà all'interno del Top

Component), può passare al Bottom Component uno o più parametri definiti al momento stesso dell'utilizzo del selettore.

A tal fine il Top Component deve definire tra parentesi quadre una nuova variabile (in questo caso denominata genericamente [nuovoStudiante]) alla quale, mediante un Property Binding andrà ad assegnare il contenuto di una variabile di ciclo oppure di una variabile definita nella classe.

Il Bottom Component dovrà dichiarare la Property nuovoStudiante tramite il decoratore @Input() che lo avvisa che questa variabile verrà passata dalla classe dal Top Component in fase di dichiarazione del componente. La variabile nuovoStudiante potrà poi essere normalmente utilizzata all'interno della classe e

della pagina html associata

```
@Input()nuovoStudiante:string
```

Si parla di Property Binding in quanto la comunicazione avviene mediante il passaggio di una Property dal Top Component al Bottom Component. Tale proprietà viene identificata anche col nome di target property.

Attenzione che le variabili ricevute tramite @Input() NON sono visibili all'interno del costruttore, ma sono invece visibili all'interno di ngOnInit() che viene richiamato DOPO rispetto alla valorizzazione dei parametri di tipo @Input().

8. Spiegare l'Event Binding tramite il decoratore @Output() per passare un parametro da un Bottom Component ad un Top Component

Si parla questa volta di Event Binding in quanto la comunicazione avviene attraverso la generazione di un evento custom emesso dal Bottom Component verso il Top Component.

Il Bottom Component (nell'esempio la barra di navigazione <app-header>) deve generare un evento custom al quale passa come parametro i valori da inviare al top component (nell'esempio appcomponent) .

EventEmitter è un Typed Object, cioè occorre specificare quale o quali tipi di parametri passeremo a questo evento. Nell'esempio si passa una semplice stringa identificativa, ma si potrebbero passare anche più parametri o eventualmente nessuno (<void>)

L'oggetto featureSelected è preceduto dal decoratore @Output che fa sì che questo evento possa essere propagato al top component, cioè a tutti i Top Component che stanno utilizzando il componente corrente.

Il metodo emit() dell'oggetto featureSelected è quello che genera fisicamente l'evento passandogli come parametro il valore feature ricevuto da onSelect()

Il top component (nell'esempio app-component), all'interno dell'html, al momento della dichiarazione del bottom component (<app-header>) deve mettersi in ascolto dell'evento custom generato dal bottom component e poi gestirlo

```
<app-header (featureSelected)="onNavigate($event)"> </app-header>
loadedFeature:string ='recipes'
onNavigate(feature:string){
  this.loadedFeature = feature
}
```

In corrispondenza dell'evento featureSelected viene richiamato il metodo onNavigate() definito all'interno del file di classe

Al metodo onNavigate() viene passato come parametro \$event che è quello visto nella sezione di gestione degli eventi, cioè una parola chiave che indica ad angular di propagare i parametri alla procedura di gestione dell'evento (che per default non vengono propagati).

9 Spiegare caratteristiche e funzionamento di un service

Un servizio è una semplice classe che implementa delle funzionalità e/o gestisce dei dati che possono essere condivisi fra tutti i vari componenti che costituiscono l'applicazione.

Un servizio, oltre che essere utilizzato dai componenti, può, a sua volta, sfruttare le funzionalità fornite da altri servizi. Nelle prime versioni il servizio era indicato con il termine provider

In generale è buona regola sollevare i componenti dagli incarichi di logica “business” concentrandola invece all'interno dei Servizi. Un servizio è solitamente rappresentato da una classe indipendente dalle View che viene definita per svolgere un compito ben preciso nel rispetto del principio di singola responsabilità: ogni servizio si occupa di svolgere un unico e ben preciso compito.

Un tipico esempio di servizio è quello per l'invio centralizzato di richieste Ajax ad un REST server.

Ogni servizio è costituito da un unico file .ts come per le Directive.

Il service può essere creato manualmente come semplice classe oppure può essere autogenerato tramite CLI fermando il server ed utilizzando il seguente comando:
`ng generate service services/recipe --skip-tests`

In fase di creazione Angular aggiunge automaticamente al servizio una desinenza Service e soprattutto un decoratore `@Injectable()` che rende il servizio injectable, cioè rende l'intera classe iniettabile al costruttore di un qualsiasi altro componente o servizio. Questo meccanismo per cui i servizi possono iniettarsi uno dentro l'altro è detto Dependency Injection. Molto comodo per utilizzare i servizi senza dover registrare ogni singolo servizio all'interno del file `app.module.ts`.

Tutti i componenti a cui verrà iniettato il servizio avranno accesso condiviso a Proprietà e Metodi del servizio medesimo.

10 Con riferimento al servizio di gestione delle richieste Ajax, spiegare la caratteristiche dell'oggetto Observable

Tutti i metodi dell'oggetto `HttpClient` restituiscono al chiamante un oggetto `Observable` che è un oggetto analogo all'oggetto `Promise` di java script e che consente al chiamante di mettersi in attesa asincrona della risposta. L'oggetto `Observable` rende disponibile un unico metodo di evento denominato `.subscribe()` il quale si aspetta come parametri due funzioni di callback, una success ed un error.

In pratica il componente asincrono `Observable` si mette in ascolto della ricezione della risposta e, quando arriva, fa scattare il metodo di evento „`subscribe()`” iniettandogli il json ricevuto dal server.

11 Spiegare il meccanismo di routing di Angular con l'utilizzo dell'`appRoutingModule` e del componente `<router-outlet>`. L'attributo `routerLink` ed

il metodo `router.navigate()`

Per la gestione del routing, Angular fornisce un apposito modulo chiamato Router che gestisce il routing tramite la definizione di apposite routes che devono essere definite manualmente dal programmatore all'interno di un modulo denominato `AppRoutingModule` definito all'interno del file `src/app/app-routing.module.ts`.

Alternativa più strutturata rispetto all'utilizzo di `ngIf`.

Quando si crea una nuova applicazione, se la si crea utilizzando il flag `--routing` automaticamente Angular crea il modulo `AppRoutingModule` all'interno del file `app-routing.module.ts`

Occorre quindi definire una costante di tipo `Routes[]`, denominata solitamente `appRoutes` che è un array in cui occorre definire tutte le varie routes necessarie all'applicazione.

- Per ogni route l'unico parametro obbligatorio è `path` che definisce il nome della route.

Le routes di primo livello devono essere scritte come routes relative (senza slash) che però il chiamante dovrà "passare" all'oggetto Router antepo-
nendo lo slash davanti. Ad es `/recipes`

- Le routes, in linea di massima, possono essere di due tipi:

- routes per il caricamento di un component,
- routes per l'esecuzione di un redirect.

Il path iniziale (vuoto) è un path di redirect ed indica che cosa bisogna visualizzare quando l'utente accede al sito senza specificare nessuna risorsa. Nel nostro caso vogliamo eseguire un redirect al path `/recipes`. L'opzione `pathMatch:'full'` è obbligatoria e sta ad indicare che il match deve essere eseguito per intero, cioè deve andare a buon fine SOLO quando l'utente inserisce veramente stringa vuota. Diversamente il match andrebbe a buon fine anche quando l'utente inserisce un qualunque altro path indicato nelle route successive.

Nella sezione principale della APP (tipicamente all'interno di `app-component`) si può inserire una tipica barra di navigazione che contiene i link alle varie "pagine" del sito (es ricette e shopping-list).

Sempre all'interno di `app-component` si può inserire un componente detto `<router-outlet>` che funge da visualizzatore per i vari componenti indicati all'interno della costante `appRoutes`.

Ogni volta che dalla barra di navigazione oppure da codice si imposta una nuova route, angular provvederà a visualizzare all'interno di `<router-outlet>` il componente associato a quella route

Per indicare al router quale route visualizzare all'interno del componente `<router-outlet>` ci sono almeno tre possibilità:

- 1) Scrivere la url direttamente nella barra di navigazione
- 2) Utilizzare all'interno dell'html un semplice tag `<a>` impostando all'interno dell'attributo `routerLink` la route da caricare. L'attributo `routerLink` è simile all'attributo `href` ed è disponibile SOLO per il tag `<a>`.
- 3) Quando non si ha a disposizione un tag `<a>`, la terza alternativa consiste nel gestire l'evento `click()` su un qualunque tag e poi caricare la pagina da Type Script utilizzando il metodo `router.navigate()`. A tal fine occorre iniettare al costruttore il componente `Injectable Router`.

12. Le routes parametriche e la lettura dei parametri tramite l'evento
activatedRoute.params per capire quale documento visualizzare

Si supponga di voler visualizzare sempre all'interno della pagina principale (cioè
all'interno del
medesimo <router-outlet> di app.component) i dettagli di una specifica ricetta, ad
esempio la
ricetta avente id=2.

A tale scopo si può aggiungere nel vettore delle routes la seguente route sempre
di primo livello: { path: ':id', component: DettagliRicettaComponent }
Notare i due punti che indicano appunto che si tratta di una route parametrica, a
cui potrà essere passata qualsiasi informazione, numerica o alfanumerica.

Questa route potrà essere richiamata nei seguenti modi:

```
<li><a routerLink = "2"> Dettagli ricetta 2</a></li>  
<li><a [routerLink]="[id]">Dettagli ricetta</a></li>  
caricaDettagli(){  
  this.router.navigate(['/2'])  
}
```

Il componente DettagliRicetta, richiamato in corrispondenza del click su un link,
avrà necessità di accedere all'id della ricetta richiesta, in modo da poter
richiedere le informazioni al server e procedere alla visualizzazione.

Per poter accedere ai parametri passati ad una route parametrica, occorre
innanzitutto iniettare al costruttore del componente DettagliRicetta l'oggetto
ActivatedRoute contenente la route attualmente selezionata: constructor(private
route:ActivatedRoute) {}

L'oggetto ActivatedRoute genera un evento params ogni volta che viene attivata una
qualsiasi route associata al componente corrente (parametrica o non parametrica).

In caso di route parametrica (che inizia con i due punti) alla funzione di
gestione dell'evento viene iniettato un json contenente il parametro di
attivazione della route, in questo caso il parametro id. Per cui, in questo caso,
il json iniettato alla funzione di gestione dell'evento, sarà:

```
{"id":2}
```

Il metodo .subscribe() definisce, all'interno della classe corrente, un listener
di evento che intercetterà tutti gli eventi params generati ogni volta che una
route porta al caricamento del componente attuale. All'interno dell'evento si può
accedere all'ID e richiedere al server i dettagli della ricetta selezionata
L'associazione tra l'evento subscribe() e la procedura di gestione dell'evento
dovrà essere eseguita in fase di avvio del componente, cioè all'interno del metodo
onInit()

```
ngOnInit(): void {  
  this.route.params.subscribe(  
    (params: Params) => {  
      if(params['id']){  
        console.log(params['id'])  
      }  
    }  
  )  
}
```

```

        this.recipeService.getRecipe(params['id']);
    }
}
)
}

```

 13. Le child routes (navigate {relativeTo}) ed il criterio di visualizzazione delle routes.

Riprendendo l'esempio precedente, il fatto di posizionare la route parametrica al primo livello non è la soluzione migliore. Sarebbe meglio impostare la route parametrica come sotto-route di recipes, in modo che l'utente possa richiamare i dettagli di una ricetta nel modo seguente:

<http://localhost:4200/recipes/2>

Una route di primo livello, tramite l'attributo children (che ha la forma di un vettore enumerativo), può esporre una o più child route, assegnando a ciascuna un corrispondente componente.

```

{
  path: 'recipes', component: RecipesComponent,
  children: [
    { path: '', component: RecipeStartComponent },
    { path: 'new', component: RecipeNewComponent },
    { path: ':id', component: RecipeDetailComponent },
    { path: ':id/edit', component: RecipeEditComponent }
  ]
}

```

- path: 'new', in corrispondenza della route recipe/new, carica un componente per l'inserimento di una nuova ricetta
- path: ':id', in corrispondenza della route recipe/2 carica il componente recipe-detail per la visualizzazione dei dettagli della ricetta indicata.
- path: '' interviene nel momento in cui l'utente richiede /recipes senza specificare nessun id, nel qual caso caricherà nell'area di destra un messaggio

<p>Please select a Recipe!</p>
- path: ':id/edit' caricherà un componente per l'editazione della ricetta corrente, precaricando tutti i valori correnti.

Le child routes relative ai vari sottocomponenti potranno essere richiamate da html o da TypeScript nel solito modo:

HTML

```
<a [routerLink]="['new']"> </a>
```

```
<a [routerLink]="[recipe.id]"> </a>
```

Notare che, a differenza delle routes di primo livello, le child routes passate a routerLink NON devono iniziare con lo slash, proprio perchè sono routes relative che verranno automaticamente concatenate alla route corrente. Il componente contenente questo link è un componente che sarà visualizzato soltanto in corrispondenza della route /recipes.

In corrispondenza del click, Angular provvederà a concatenare il valore di routerLink alla route corrente, producendo come risultato ad esempio /recipes/2

Type Script

Usando il metodo `navigate()` il concatenamento precedente non è automatico ma occorre specificare un secondo parametro che indichi a quale route deve essere concatenato il “path relativo” contenuto nel primo parametro. Il valore del secondo parametro sarà praticamente sempre `this.route`

```
constructor(private router:Router, private route:ActivatedRoute) { }  
onNewRecipe() {  
  this.router.navigate(['new'], { relativeTo: this.route }) // oppure  
  this.router.navigate(['/recipe/new'])  
}
```

Al solito, con `navigate()`, è obbligatorio l’uso del vettore.

`new`, essendo una child-route, DEVE essere scritta senza slash. Viceversa lo slash davanti a `recipes` può indifferentemente essere inserito oppure tralasciato.

In caso di click sui dettagli della ricetta corrente `/recipes/2` andiamo a concatenare “edit” per il caricamento del componente preposto all’editazione del record corrente

```
this.router.navigate(['edit'], { relativeTo: this.route })
```

Quando viene impostata una route mediante click sul un tag `<a>` oppure mediante il metodo `router.navigate()`, Angular in prima battuta verifica se si tratta di una route assoluta (`/recipes`)

oppure di una child-route (`/recipes/new`)

- Se si tratta di una route assoluta, il componente associato alla route viene caricato all’interno del `<router-outlet>` di `app.component`, il quale deve sempre avere al suo interno un `<router-outlet>`. Se non ci fosse, semplicemente la route non viene „servita“

- Se si tratta invece di una child-route, Angular guarda alla Parent-Route (`/recipes`). Se la Parent-Route (`/recipes`) ha un componente assegnato, questo componente dovrebbe esporre un `<router-outlet>`. Ogni componente che ha delle child-routes deve sempre esporre un `<router-outlet>` in cui verranno visualizzati i componenti ‘puntati’ dalle child-routes. Se il componente padre non espone un `<router-outlet>`, allora la child route non viene servita.

Per cui lo scenario precedente può essere completato assegnando a `RecipesComponent` un `<router-outlet>` in cui visualizzare TUTTI i sotto componenti associati alle varie child-routes.