Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica

Analisi comparativa di protocolli di modellazione e trasferimento dati: REST API vs GraphQL

Tesi di laurea

Relatore

Prof.Paolo Baldan

Laureando

Federico Marchi

Anno Accademico 2021-2022

Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

— Oscar Wilde

Dedicato a ...

Sommario

Il presente documento descrive il lavoro svolto durante il periodo di stage, della durata di circa trecento ore, dal laureando Pinco Pallino presso l'azienda Azienda S.p.A. Gli obbiettivi da raggiungere erano molteplici.
In primo luogo era richiesto lo sviluppo di ... In secondo luogo era richiesta l'implementazione di un ... Tale framework permette di registrare gli eventi di un controllore programmabile, quali segnali applicati Terzo ed ultimo obbiettivo era l'integrazione ...

"Life is really simple, but we insist on making it complicated"

— Confucius

Ringraziamenti

Innanzitutto, vorrei esprimere la mia gratitudine al Prof. NomeDelProfessore, relatore della mia tesi, per l'aiuto e il sostegno fornitomi durante la stesura del lavoro.

Desidero ringraziare con affetto i miei genitori per il sostegno, il grande aiuto e per essermi stati vicini in ogni momento durante gli anni di studio.

Ho desiderio di ringraziare poi i miei amici per tutti i bellissimi anni passati insieme e le mille avventure vissute.

Padova, Dicembre 2022 Federico Marchi

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Introduzione al contesto applicativo.

Esempio di utilizzo di un termine nel glossario
Application Program Interface (API).

Esempio di citazione in linea
Manifesto Agile. URL: http://agilemanifesto.org/iso/it/.

Esempio di citazione nel pie' di pagina
citazione[1]

1.1 L'azienda

Descrizione dell'azienda.

1.2 L'idea

Introduzione all'idea dello stage.

1.3 Organizzazione del testo

Il secondo capitolo descrive ...

Il terzo capitolo approfondisce ...

Il quarto capitolo approfondisce ...

Il quinto capitolo approfondisce ...

Il sesto capitolo approfondisce ...

Nel settimo capitolo descrive ...

[1] Daniel T. Jones James P. Womack. *Lean Thinking, Second Editon*. Simon & Schuster, Inc., 2010.

Riguardo la stesura del testo, relativamente al documento sono state adottate le seguenti convenzioni tipografiche:

* gli acronimi, le abbreviazioni e i termini ambigui o di uso non comune menzionati vengono definiti nel glossario, situato alla fine del presente documento;
* per la prima occorrenza dei termini riportati nel glossario viene utilizzata la seguente nomenclatura: *parola*[g];
* i termini in lingua straniera o facenti parti del gergo tecnico sono evidenziati con il carattere *corsivo*.

Capitolo 2

Descrizione dello stage

Breve introduzione al capitolo

2.1 Introduzione al progetto

2.2 Analisi preventiva dei rischi

Durante la fase di analisi iniziale sono stati individuati alcuni possibili rischi a cui si potrà andare incontro. Si è quindi proceduto a elaborare delle possibili soluzioni per far fronte a tali rischi.

1. Performance del simulatore hardware
Descrizione: le performance del simulatore hardware e la comunicazione con questo potrebbero risultare lenti o non abbastanza buoni da causare il fallimento dei test.
Soluzione: coinvolgimento del responsabile a capo del progetto relativo il simulatore hardware.

2.3 Requisiti e obiettivi

2.4 Pianificazione

Capitolo 3

Protocolli di modellazione e trasferimento dati

Nel seguento capitolo vengono trattati dal punto di vista teorico i protocolli di modellazione e trasferimento dati, in particolare viene approfondito lo stile architetturale REST e il linguaggio di query GraphQL.

3.1 Introduzione ai protocolli

Modello architetturale client-server

Prima di procedere nella spiegazione sui protocolli di trasferimento e modellazione dati, è necessario fare una breve introduzione sull'architettura delle Web Application moderne. Queste seguono ormai tutte un modello server - client, ovvero un modello architetturale che divide in due processi l'applicazione: un client che richiede servizi al server, il quale li esegue ritornando una risposta contenente l'esito dell'operazione. I protocolli di trasferimento e modellazione dati trovano il loro maggior utilizzo proprio nella comunicazione client - server, tuttavia prima di procedere con la loro spiegazione è necessario introdurre il concetto di Application Programming Interfaces.

Application Programming Interfaces

Comunemente dette API, ovvero l'acronimo di Application Programming Interfaces, sono interfacce comunemente realizzate per aggevolare la comunicazione tra server e client. Ciascun applicativo/dispositivo è sviluppato con strutture di dati differenti che evolvono nel tempo, dunque risulta complessa la comunicazione tra queste entità. Le API giocano un ruolo fondamentale nello scambio di dati: infatti definiscono una interfaccia per la comunicazione, la quale è indipendente dall'implementazione specifica del dispositivo o dell'applicativo e permette di comunicare secondo delle regole specifiche riportate nella propria documentazione. Risultano dunque fondamentali nella comunicazione, collaborazione e integrazione di nuovi componenti applicativi.

Al giorno d'oggi le API vengono utilizzate dalla maggior parte delle web applications, dispositivi IoT, applicativi di vario genere e molto altro ancora. Nello specifico nella tesi si fa riferimento alle Web API, ovvero a quelle interfacce che sfruttano il

protocollo HTTP per la comunicazione con altri applicativi/dispositivi. Ci sono diversi tipi di Web API, tra queste:

* **API pubbliche**: si tratta di API accessibili da tutti (possono essere anche a pagamento);
* **API private**: si tratta di API create con lo scopo di essere utilizzate solo ed esclusivamente all'interno dell'azienda;
* **API partner**: si tratta di API utilizzate tra aziende in collaborazione;
* **API composte**: si tratta di API differenti combinate tra loro per creare una sequenza di operazioni.

La necessità di standardizzare il modo in cui vengono sviluppate le interfacce API ha portato dunque alla nascita dei protocolli sul trasporto di dati.

Protocolli di trasferimento dati

Per protocolli di modellazione e trasferimento dati s'intende un insieme di regole, strutture e vincoli che regolano il funzionamento delle API. Permettono dunque di definire una sorta di standard al quale gli sviluppatori possono far riferimento per implementare e interagire con le API. Il termine protocollo non si addice perfettamente a tutte le varie tecnologie di data fetching, tuttavia a grandi linee può racchiuderle e dunque verrà utilizzato per questione di comodità.

I primi protocolli e loro evoluzione

Al giorno d'oggi il protocollo di modellazione e trasferimento dati più utilizzato è sicuramente REST, tuttavia sono presenti anche altre tipologie di protocolli in utilizzo o che comunque sono state utilizzate in passato. Si tratta di tecnologie con lo stesso scopo, ma di natura completamente differente, di seguito le principali.

Remote Procedure Call

Viene spesso indicato con l'acronimo RPC, si tratta di protocollo secondo il quale una procedura o subroutine viene invocata da un client esterno al server che deve eseguire la procedura, senza che il client conosca i dettagli del network. Viene utilizzato per chiamare processi in sistemi remoti, ma come fossero locali.
Di seguito riportata la definizione attribuita all'RPC dagli informatici Andrew Birrell e Bruce Nelson nel 1984:

> "Meccanismo sincrono che trasferisce il flusso di controllo e i dati attraverso una chiamata di procedura tra due spazi di indirizzo su una rete a banda stretta."

Come nelle chiamate a procedure locali, un RPC è una operazione sincrona che tiene in pausa il client fino al momento in cui ritorna il risultato della procedura invocata.

Simple Object Access Protocol

Indicato spesso con l'acronimo SOAP, si tratta di un vero e proprio protocollo che definisce la struttura dei dati che devono essere trasferiti e come questi devono essere elaborati. Richiede esclusivamente il formato XML per trasferire dati e tipicamente

viene utilizzato il protocollo HTTP per il trasferimento di file, tuttavia possono essere utilizzati anche protocolli differenti come ad esempio il protocollo SMTP. La struttura di un messaggio SOAP è composta da 3 principali componenti:

* **Envelope**: necessario al fine di identificare il documento come messaggio SOAP;
* **Header**: è opzionale. Lo scopo dell'header nei messaggi SOAP è quello di trasportare indicazioni estranee al messaggio che si vuole trasportare, ma che vengono interpretate da i diversi nodi durante il cammino del messaggio;
* **Body**: il body contiene il vero e proprio messaggio che si vuole trasferire.

Ad hoc

3.2 Approfondimento sullo stile architetturale REST

REST è l'acronimo di "Representational State Transfer" e si tratta di un tipo di stile architetturale introdotto da Roy Fielding nel 2000 e viene considerato al giorno d'oggi come uno standard per la realizzazione di web API. Si tratta di una astrazione degli elementi di un architettura di un sistema, del quale REST ne ignora i dettagli del'implementazione delle componenti e della sintassi del protocollo imponendo dei vincoli sul loro ruolo e sulla loro interazione.

Principi di un architettura REST

Un architettura REST dunque deve rispettare alcuni principi, di seguito verranno elencati i sei gruppi definiti da Fielding.

Client-server

Il primo principio sposa uno dei paradigmi cardine dell'informatica, ovvero il principio di *Separation of concerns*, secondo il quale conviene sempre separare un sistema complesso in moduli distinti in modo che ognuno possa avere un proprio compito.
Ciò viene ripreso nell'architettura REST separando il client dal server, dunque dividendo due logiche diverse in due moduli distinti. Così facendo server e client possono essere implementati in maniera indipendente, usando qualsiasi lingua o tecnologia, basta che siano conformi al prossimo principio detto *uniform interface*.

Uniform interface

Si tratta di un principio fondamentale che differenzia le REST API da qualsiasi API non REST. Secondo questo prinpicio l'interazione tra componenti Web, dunque client, server e tutti gli intermediari del network, dipendono dalla uniformità delle loro interfacce.
I componenti Web dunque sono in grado di comunicare coerentemente seguendo quattro vincoli sull'interfaccia delineati da Fielding; questi sono:

* **Identification of resources**: le risorse che vengono richieste devono essere identificate nella richiesta stessa, dunque specificandole nell'url;
* **Manipulation of resources through representations**: il client deve avere la rappresentazione delle risorse e deve poter sapere come modellarle sul server. L'idea alla base è che la rappresentazione (attraverso un qualsiasi formato, ad es.

JSON, XML, ecc...) è una modo per interagire con le risorse, ma non è la risorsa stessa;

* **Self-descriptive messages**: in ciascun messaggio devono esser presenti le informazioni necessarie a descrivere come deve essere processata la richiesta;

* **Hypermedia as the Engine of Application State**: la rappresentazione dello stato di una risorsa deve includere i riferimenti alle risorse correlate. É dunque necessario includere i link per ciascuna risposta, così che il client possa navigare tra le altre risorse facilmente.

Layered System

Secondo questo principio l'architettura di un applicativo deve essere composta da più strati. Ciascuno strato inoltre è cieco rispetto agli altri strati, tranne per quanto riguarda gli strati adiacenti. Questi layer possono essere composti da intermediari basati sul network i quali intercettano la comunicazione client-server con uno scopo specifico (ad esempio per questioni di sicurezza, caching, controllo del flusso dati, ecc...), possono essere ad esempio proxy e gateways. Per il principio di Layered System questi intermediari devono aderire alle interfacce al fine di mantenerne l'uniformità.

Cache

Si tratta di uno dei vincoli fondamentali in un archiettura Web, secondo il quale un web server deve dichiarare la *cacheability* di ciascuna risposta ritornata. Più specificatamente qualsiasi risposta di un server deve etichettare come cacheabili o meno i dati presenti all'interno di esso. Così facendo gli intermediari tra server e client e il client stesso sanno come comportarsi riguardo alla memorizzazione dei dati.

Stateless

Il vincolo di stateless fa riferimento al fatto che un server non deve memorizzare lo stato dell'applicazione client. Questo implica però che ogni richiesta che il server riceve dal client deve essere sufficientemente dettagliata sullo stato del client affinché il server sia in grado di eseguirla. Dunque le richieste non sono correlate tra loro e per questo viene definito "stateless".
Questo vincolo porta un vantaggio fondamentale secondo il quale un server così facendo può gestire richieste da molti client. Può inoltre esser scalato molto più facilmente con l'aiuto ad esempio di un load balancer.

Code on demand

Per ultimo troviamo il vincolo di code on demand, si tratta di un vincolo facoltativo secondo il quale la logica del client può essere aggiornata indipendentemente da quella lato server. Un esempio pratico lo troviamo nella signel web application le quali rispettano totalmente queste vincolo.

3.3 Approfondimento sul linguaggio di query GraphQL

Introduzione

GraphQL è stato ideato da Facebook nel 2012 e condiviso e reso pubblico nel 2014. Al giorno d'oggi molte importanti applicazioni utilizzano GraphQL, come ad esempio GitHub, Twitter, PayPal e Pinterest. Viene considerato come il principale competitor e possibile successore di REST nell'ambito del data fetching, tuttavia come verrà spiegato in seguito, oltre a svariati punti di forza e di innovazione ha anche alcuni problemi.
Più nello specifico GraphQL è un linguaggio di query per le APIs. Viene definito agnostico rispetto al mezzo di trasporto perché non dipende dal modo in cui client e server comunicano, ma solitamente viene utilizzato sul protocollo HTTP. Il principale punto di forza di GraphQL è la possibilità di specificare nella query esattamente i dati che si è interessati a ricevere, questo permette dunque di non occupare la rete per dati non richiesti. Altro importante punto di forza, ma che talvolta può risultare un problema, è che è fortemente tipizzato.
Affermare che GraphQL abbia lo scopo di servire esclusivamente come linguaggio di query può risultare riduttivo. Una dei principali motivi d'utilizzo di GraphQL è quello di riuscire a raggruppare tutti i dati e servizi di un'applicazione insieme in uno stesso posto, e fornire così un'interfaccia unica che risulti consistente, sicura e infine semplice da utilizzare.
GraphQL non specifica come deve essere costruita un'API, tuttavia ci sono cinque linee guida dette "Principi di desing" da tenere in considerazione durante lo sviluppo di un API:

* **Hierarchical**: i tipi ricercati in una query GraphQL seguono una struttura gerarchica, infatti i tipi possono avere come campi altri tipi e così via. Inoltre i dati che vengono ritornati dalla query, vengono ritornati esattamente con la medesima struttura con cui sono stati richiesti;

* **Product centric**: le API sono inevitabilmente guidate dalle richieste dal client, per questo bisogna realizzarle in maniera flessibile cercando di tener conto delle richieste client per permettere quanto richiesto;

* **Strong typing**: un server GraphQL è supportato da un type system specifico a seconda dell'applicazione. Data una query, il server assicura che questa sia sintatticamente corretta, valida e che i tipi in gioco rispettino esattamente la struttura dei tipi definiti nel GraphQL schema;

* **Client-specified queries**: in GraphQL, la codifica della query avviene nel client e non nel server e si tratta di query che vanno a specificare campo per campo. Nella maggiorparte dei sistemi che non utilizzano GraphQL, il server determina quali dati ritornare. In GraphQL ciò non accade, vengono infatti ritornati solo i dati specificati dal client;

* **Introspective**: GraphQL è introspettivo, infatti i clients possono consultare a fondo il GraphQL schema e possono dunque vedere tutte le query disponibili, i vari tipi e i loro campi.

GraphQL schema

GraphQL ha cambiato il modo di pensare alle APIs: queste non vengono più considerate come un insieme di endpoints dai quali ottenere dati ed eseguire servizi, ma vengono piuttosto considerate come una collezione di tipi.
La progettazione delle APIs GraphQL risulta essere differente da come avviene negli altri protocolli, infatti prima di procedere con l'implementazione delle APIs, è necessario definire i tipi di dati che verranno esposti e richiesti dalle APIs. Questo approcio viene denominato "**Schema first**" e si tratta di una tecnica che prevede appunto come prima fase della progettazione del sistema di APIs, la creazione di una sorta di pagina nella quale radunare tutti i tipi necessari, questo posto viene definito **GraphQL Schema**. È importante definire nel dettaglio all'interno dello schema tutti i tipi che possono essere richiesti e inviati dai clients. Così facendo poi gli sviluppatori frontend saranno in grado di conoscere nel dettaglio la struttura di ciascun tipo e delle varie query. Le APIs sviluppate con GraphQL si autodocumentano proprio perché è sufficiente la consultazione dello schema per comprendere la natura delle entità che si desidera interrogare o modellare.

Definizione dei tipi

Come detto in precedenza la caratteristica principale di GraphQL è che si tratta di un linguaggio di query fortemente tipizzato. I tipi sono l'unità principale di un GraphQL schema.
Per tipo s'intende un oggetto costruito dettagliatamente campo per campo che deve poi corrispondere ad una entità nel backend dell'applicativo. Dunque all'interno dello schema dovranno essere definiti tutti i tipi che andranno a rappresentare la struttura dati dell'applicativo.
Un tipo può contenere come campi dati altri tipi che sono definiti nel medesimo schema. Segue un esempio di tipo dichiarato in uno schema GraphQL, in questo caso si tratta della dichiarazione di un Employee:

```
type Employee {
  id: ID
  name: String!
  owns: Badge
  worksIn: Department
  worksOn: [Project]
  ...
}
```

In questo caso il tipo Employee avrà come campi:

* **id**: un codice identificativo di tipo *ID*;
* **name**: un nome di tipo *String*, il punto esclamativo indica che si tratta di un campo che non può essere nullo;
* **owns**: un badge di tipo *Badge* per l'accesso al dipartimento;
* **worksIn**: un dipartimento di tipo *Department* nel quale lavora l'impiegato,
* **worksOn**: una lista di progetti di tipo *Project* al quale l'impiegato sta lavorando.

I tipi *Department*, *Badge* e *Project* dovranno essere necessariamente definiti all'interno dello stesso GraphQL schema di Employee; I built-in type che GraphQL mette a disposizione vengono detti **scalar type** e sono: *Int*, *String*, *Boolean*, *ID*, *Float*. È possibile inoltre dichiarare anche degli scalar type personalizzati con la keywork "*scalar*", oppure delle enumerazioni attraverso l'utilizzo della keyword "*enum*". È possibile infine unire diversi tipi, molto utile nel caso in cui si volesse ritornare uno tipo tra un insieme di tipi, questo è possibile farlo con la keyword "*union*" come segue:

```
union worker = Employee | Manager | Chief
```

In questo caso sono stati uniti in un unico tipo *worker* i tipi *Employee*, *Manager* e infine *Chief*. Verrà molto utilizzato successivamente il tipo unione nella gestione degli errori di ritorno al client dopo l'esecuzione delle query.

Connessioni tra tipi

GraphQL è cosi denominato perché oltre ad essere un Query Language come suggeriscono le ultime due lettere del nome, permette di definire connessioni di vario genere tra i tipi definiti nello schema, queste connessioni vanno di fatto a creare uno grafo composto da tipi interconnessioni, da questo deriva il prefisso *Graph*. È fondamentale durante la definizione del GraphQL schema riportare le relazioni nella maniera corretta delle entità corrispondenti nel database dell'applicativo. Un'ultima premessa prima di visualizzare i vari tipi di connessioni riguarda la direzionalità delle connessioni: in GraphQL risulta essere una buona pratica dare bidirezionalità alle connessioni ove possibile, questo con lo scopo di lasciare più flessibilità possibile allo sviluppatore client il quale dalla una query specifica può raggiungere diversi tipi e spostarsi nel grafo come più desidera.
Di seguito vengono elencate le varie relazioni con relativi esempi.

Connessione one-to-one

Nelle relazioni one-to-one ad un tipo viene associata una sola istanza di un altro tipo e viceversa. Riprendendo il caso del tipo Employee riportato sopra, possiamo trovare una relazione del tipo one-to-one tra i tipi Employee e Badge. Di seguito la rappresentazione del grafo:

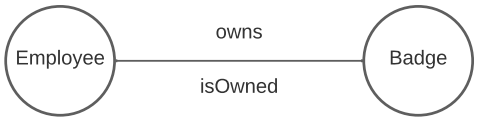

Figura 3.1: Connessione one-to-one.

Come mostrato in figura 3.1 il collegamento tra i due tipi è definito come "owns" se si legge nel verso che parte da Employee per raggiungere Badge e fa riferimento all'omonimo campo di Employee. Se altrimenti la connessione si percorre nel verso opposto viene definita "isOwned", come l'omonimo campo di Badge, riportato in seguito:

```
type Badge {
  id: ID
```

```
  isOwned: Employee
  ...
}
```

Connessione one-to-many

In questo caso bisogna focalizzarsi sul campo *worksIn* di Employee. Questo campo definisce la connessione con un elemento di tipo *Department*, dunque a ciascun impiegato corrisponde un dipartimento nel quale lavora. Tuttavia pensando alla connessione in senso opposto a ciascun dipartimento possono corrispondere più impiegati. Segue dunque la rappresentazione della connessione nel grafo:

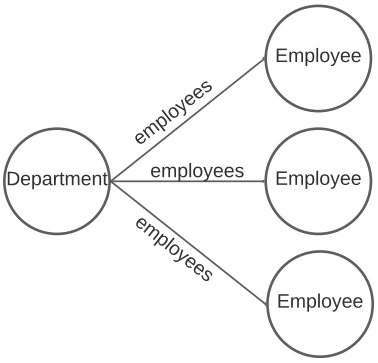

Figura 3.2: Connessione one-to-many.

Come mostrato in figura 3.2 il collegamento tra il dipartimento e i vari impiegati viene chiamato "employees" se si considera il verso che parte da Department per raggiungere Employee e corrisponde all'omonimo campo di Department. Tuttavia il collegamento è definito "worksIn" se la connessione viene percorsa nel verso opposto. Segue la rappresentazione del tipo Department:

```
type Department {
  id: ID
  name: String!
  address: String!
  employees: [Employee]
}
```

Connessione many-to-many

Consideriamo ora il campo *worksOn* di Employee che collega ciascun impiegato con una lista di progetti ai quali sta lavorando. In questo caso però considerando il collegamento nel verso opposto anche ciascun progetto può avere più impiegati che ci lavorano. In questo caso si tratta di una relazione many-to-many e segue la rappresentazione nel grafo:

Le relazioni many-to-many non sono altro che l'unione di due relazioni one-to-many.

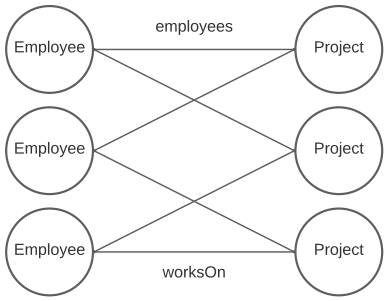

Figura 3.3: Connessione many-to-many.

La connessione in questo caso, come nei casi precedenti, a seconda del verso in cui viene percorsa può esser definita come "worksOn" o "employees" come mostrato in figura 3.3. Di seguito la rappresentazione del tipo Project:

```
type Project {
  id: ID
  name: String
  employees: [Employee]
}
```

Operazioni sui dati

Come detto in precedenza GraphQL è un linguaggi di query e come tale permette di interrogare i dati o eseguire operazioni su di essi. Ci sono tre tipi di operazioni che possono essere fatte sui dati e queste sono: *Query*, *Mutation* e infine *Subscription*.

Query

L'operazione di query viene utilizzata per richiedere dati da una determinata API ed equivale alla GET nel protocollo REST. È necessario dichiarare nel GraphQL schema la query che il programmatore backend desidera rendere disponibile, così facendo vengono dichiarati anche i tipi che possono eventualmente essere passati come argomenti e quelli che verranno ritornati dalla query.
Un esempio di dichiarazione di query che ritorna una lista di tutti gli oggetti di tipo Employee presenti in un determinato Department può essere:

```
type query {
  employeesInDepartment(departmentId: ID!): [Employee]
}
```

In questo caso invocando la query *employeesInDepartment* e passando come argomento alla query l'id (non può essere un valore nullo) del dipartimento, riceveremo come risposta un JSON contenente un campo "*data*" contenente a sua volta la lista di impiegati, questo se la query ha avuto successo. In caso di insuccesso della query per qualsiasi motivo, ad esempio per avere passato un id errato, allora verrà ritornato un JSON con un campo "*error*", contenente la descrizione dell'errore.
Dal punto di vista del client, qualora si volesse invocare questa query bisognerebbe strutturare la richiesta come segue:

```
query {
  employeesInDepartment(id: "2BR4S") {
    id
    name
    worksOn {
      id
      name
    }
  }
}
```

Se la query non dovesse fallire, verrà ritornata una lista di impiegati e per ciascun impiegato verranno ritornati i campi specificati nella query dunque: l'*id*, il *name*, una lista di oggetti di tipo Project nel campo *worksOn* per i quali bisognerà a loro volta specificare i campi ai quali si è interessati, in questo caso all' *id* e al *name* di ciascun progetto.
È inoltre possibile utilizzare gli argomenti delle query per controllare la quantità di dati che possono esser ritornati con un processo chiamato *data paging*, oppure usarli per decidere in che ordine vogliamo che vengano ritornati i dati.

Mutation

L'operazione di mutation viene utilizzata per eseguire modifiche sui dati. Equivale all'unione delle operazioni POST, DELETE, PUT, PATCH nel protocollo REST. Come per le query è necessario dichiarare le mutation che si voglion rendere disponibili al client.
Un esempio di mutation può essere:

```
type mutation {
  addNewEmployee(employee: Employee!): Employee
}
```

In questo caso la mutation *addNewEmployee* andrà ad aggiungere un nuovo impiegato nella struttura dati dell'applicativo. Il client per invocare questa mutation dovrà strutturare la richiesta come segue:

```
mutation {
  addNewEmployee(employee: {
    name: "Mario"
  }){
    id
  }
}
```

In questa mutation è stato passato come argomento un oggetto di tipo Employee, del quale è stato specificato esclusivamente il nome (va obbligatoriamente specificato essendo un campo dichiarato non nullo). Come da definizione la mutation ritorna un oggetto di tipo Employee, del quale però in questo caso si vuole ricevere solo l'id generato. Come nel caso della query, se l'aggiunta dell'impiegato avrà successo, nel JSON di ritorno ci sarà un campo *data* contenente l'id generato in seguito all' aggiunta dell'impiegato, in caso contrario sarà ritornato un JSON con un campo *error* che descrive l'origine dell'errore.

Subscription

L'ultimo tipo si chiama Subscription e si tratta di una funzione particolare resa disponibile in GraphQL, infatti grazie a questa funzione i client possono sottoscriversi ad una subscription e così facendo sarà il server ad inviare al client i dati richiesti non appena questi sono disponibili, dunque non è più necessario che sia il client a richiedere periodicamente i dati aggiornati.
Un esempio di definizione di una Subscription può essere:

```
type subscription {
  newEmployeeAdded: Employee!
}
```

Il client che desidera sottoscriversi alla subscription *newEmployeeAdded* dovrà mandare una richiesta strutturata come segue:

```
subscription {
  newEmployeeAdded {
    id
    name
  }
}
```

Così facendo il server, appena viene aggiunto un nuovo impiegato, invierà direttamente al client i dati che il client ha specificato nella sottoscrizione, ovvero in questo caso l'*id* e il *name* dell'impiegato.
Essendo il server a dover inviare i dati al client e non il client che richiede i dati dal server, non è utilizzabile il protocollo HTTP per la comunicazione server - client, bisogna quindi utilizzare il protocollo WebSocket per aprire un canale di comunicazione a doppia via sopra un socket TCP.

Capitolo 4

Introduzione ai casi d'uso per l'analisi comparativa

Illustrazione del prototipo realizzato, prime considerazioni su di esso. Successivamente caso d'uso di SushiLab, migrazione e considerazioni.

4.0.1 Confronto con stakeholder

Valuto se parlare del confronto con stakeholder...

4.1 Prototipo

FORSE QUESTA PARTE VA SU UNO DEI PRIMI DUE CAPITOLI.... INTANTO LASCIO QUA.

Prima di procedere con la spiegazione nella progettazione, realizzazione e migrazione del prototipo vanno fatte delle premesse.
Si tratta di un prototipo realizzato al fine di:

* familiarizzare con le tecnologie Spring e Angular per la realizzazione rispettivamente di backend e frontend, il tutto in preparazione alla migrazione dell'applicativo aziendale riportato al punto 4.2;
* familiarizzare con la realizzazione delle API sia con lo stile architetturale REST, che con il linguaggio di query GraphQL;
* avere un caso d'uso ulteriore a conferma delle analisi che verranno poi ricavate dalla migrazione dell'applicativo SushiLab;

Per questi motivi si tratta di un prototipo specifico che mira alla realizzazione delle API e al loro massimo utilizzo.

Il prototipo che è stato scelto di realizzare è un applicazione client-server con funzione di gestionale. Deve permettere di gestire gli impiegati e i progetti ai quali stanno lavorando, il tutto in diverse sedi con diversi dipartimenti.

4.1.1 Progettazione del prototipo

Architettura generale dell'applicativo

L'applicativo seguie l' architettura raffigurata nell'immagine 4.1. Il backend del prototipo è stato realizzato con framework Spring Boot, mentre il frontend con framework Angular.

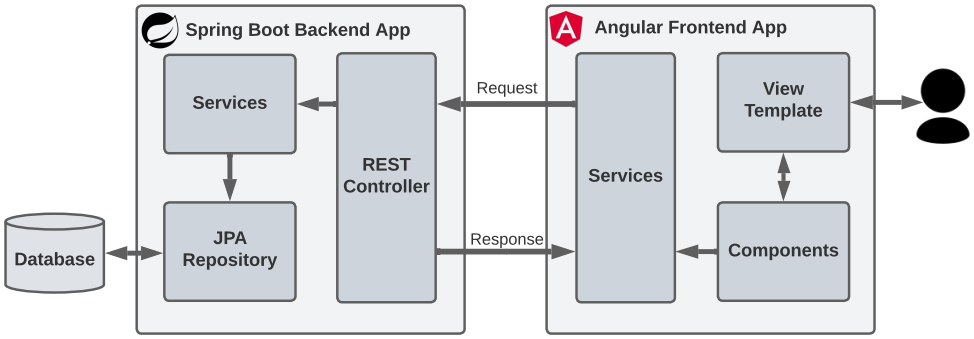

Figura 4.1: Architettura del prototipo di Web Application.

L'architettura segue il modello server-client introdotto precedentemente. Le richieste effettuate dai servizi del client venggono mappate sul REST controller del server il quale, dopo la rielaborazione interna delle richiesta, ritorna la risposta al client; quest'ultimo può dunque aggiornare la vista visualizzata dall'utente. In seguito vengono affrontati più nel dettaglio i moduli interni sia per quanto riguarda il server che la Web Application.

Architettura del server

Controller - Service - Repository È stato deciso di seguire il pattern controller - service - repository per la realizzazione del server del prototipo. Questa scelta è stata presa in quanto è consigliato nello sviluppo del backend con framework Spring Boot. Inoltre il pattern rispetta perfettamente il principio di "Separation Of Concerns".

PARLARE E INSERIRE QUI DIAGRAMMA DI SEQUENZA

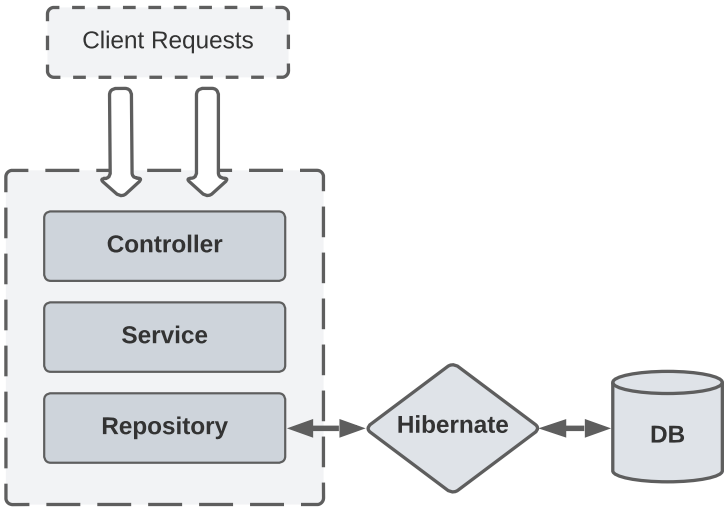

Figura 4.2: Architettura interna backend.

Il pattern prevedere la gestione delle entità e delle chiamate alle API attraverso tre strati:

* **Controller layer**: si trova in cima all'immagine 4.2, è l'unico responsabile della interazione con entità esterne, inoltre gestisce le interfacce REST e invoca lo strato di servizio;
* **Service layer**: è lo strato tra controller e repository, si occupa della business logic e qualora sia necessario visualizzare, salvare, modificare o eliminare dati allora comunica con lo strato di persistenza;
* **Repository layer**: si tratta dello strato inferiore dell'architettura, si occupa della gestione dei dati e delle loro modifiche. Lo strato di repository inoltre si occupa della comunicazione e gestione del database.

Entità e relazioni Trattandosi di un gestionale aziendale semplificato, sono previste solo quattro entità principali, queste sono:

* **Employee**: l'impiegato che può lavorare ad uno o più progetti e in un dipartimento;
* **Project**: un progetto aziendale a cui partecipano più impiegati;
* **Site**: si tratta di una sede aziendale, può avere più dipartimenti;
* **Department**: un dipartimento che appartiene ad una sede.

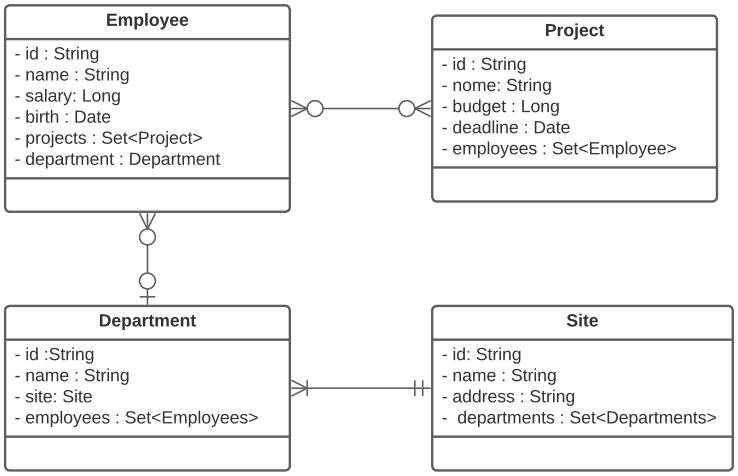

Figura 4.3: Diagramma ER del prototipo.

Ciascuna entità è caratterizzata da i campi presenti in figura 4.3 dove è stato utilizzato il linguaggio UML. Le relazioni presenti tra le varie entità sono:

* **many to many**: è presente tra Employee e Project, infatti ciascun impiegato può lavorare a più progetti e ciascun progetto può avere più impiegati al quale ci lavorano;
* **one to many**: è presente tra due coppie di entità:
  - tra Employee e Department, infatti ciascun impiegato può lavorare in uno o nessun dipartimento, mentre ciascun dipartimento può ospitare più impiegati;
  - tra Department e Site, ciascun dipartimento può appartenere esclusivamente ad una sede, mentre ciascuna sede può esser composta da più dipartimenti.

Realizzazione e testing server

Durante la realizzazione viene seguito il percorso inverso rispetto a quanto visto nell'immagine 4.2, infatti la realizzazione avviene partendo dallo strato di persistenza, dunque dalla creazione delle entità e delle repositoryes.

Entità e repository A ciascun entità nel database viene fatta corrispondere una classe in Java. Per questo devono essere realizzate 4 classi rappresentanti le entità Employee, Project, Department e Site.
Ciascuna classe entità implementa la classe *Serializable*, così facendo è possibile serializzare i dati in flussi di byte. La serializzazione viene utilizzata poiché si tratta di dati che dovranno essere memorizzati nel database, dunque è necessario serializzarli poiché abbandonano la Java Virtual Machine. Viene inoltre utilizzato un *serialVersionUID* per attribuire una versione a ciascuna classe di entità serializzabile, necessario per riconoscere quando nella comunicazione con il database o con altri moduli esterni la versione dell'entità risulta differente, si tratta dunque di entità che non corrispondono

totalmente, in quel caso viene ritornato un erorre *InvalidClassException*.
Le quattro entità descritte quindi nel capitolo 4.1.1 dovranno essere implementate come classi, segue l'esempio dell'implementazione della classe Employee:

```
@Entity
@Table(name = "EMPLOYEE")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Employee implements Serializable {
    private static final long serialVersionUID = 8452515756703751450L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private String id;
    private String name;
    private String surname;
    private Long salary;
    private Date birth;
    @ManyToMany
    @JoinTable(name = "EMPLOYEE_PROJECTS", joinColumns = { @JoinColumn(name = "EMPLOYEE_ID")},
            inverseJoinColumns = { @JoinColumn(name = "PROJECT_ID")})
    @JsonIgnore
    private Set<Project> projects = new HashSet<>();
    @ManyToOne
    @JsonBackReference(value = "employee_department")
    private Department department;
}
```

Figura 4.4: Esempio di implementazione dell'entità Employee in Spring Boot.

Vengono attribuite alla classe Employee diverse annotazioni Spring, tra queste:

* **@Entity**: si tratta dell'annotazione che permette di mappare la classe Employee come una corrispondente tabella nel database. Questa annotazione è resa disponibile dal modulo Spring Data JPA il quale implementa la specifica delle Java Persistence API attraverso Hibernate ORM e permette dunque il mapping della classi con corrispettive entità nel database;

* **@Table**: questa annotazione permette di specificare il nome della tabella generata o presente nel database durante la sua creazione o aggiornamento;

* **@Data**: grazie alla libreria *lombok* è possibile, attribuendo alla classe questa annotazione, generare automaticamente tutti i metodi get e set per tutti i campi della classe;

* **@NoArgConstructor e @AllArgConstructor**: permettono di generare tutti le combinazioni di costruttori con parametri e quello senza parametri.

Spostando invece il focus sui campi della classe Employee in figura 4.4, possiamo notare che il campo *id* ha due annotazioni associate. La annotazione **@Id** permette di specificare nel mapping che si tratta della chiave primaria, mentre l'annotazione **@GeneratedValue** permette di specificare che quando una nuova istanza di una entità viene creata, deve essere generato un nuovo id randomico.

Proseguendo sono presenti tutte le dichiarazioni dei vari campi dati dell'entità Employee e infine, troviamo le relazioni che Employee ha con le enetità Project e Department.

Anche in questo caso le annotazioni fornite dalla specifica JPA permettono di specificare nel dettaglio le varie relazioni. Sono dunque presenti le annotazioni:

* **@ManyToMany** e **@ManyToOne**: queste specificano il tipo di relazione che è presente con le altre entità, sono rispettivamente associate ai campi *projects*, con il quale Employee ha una relazione molti a molti e infine al campo *department*, con il quale Employee ha una relazione molti a uno;
* **@JoinTable**: è associata al campo *projects*, e poiché le relazioni molti a molti necessitano di una ulteriore tabella per la memorizzazione di tutte le associazioni, questa annotazione permette di specificarne il nome, ovvero *EMPLOYEE-PROJECTS* e i nomi delle due colonne, ovvero *EMPLOYEE-ID* e *PROJECT-ID*;
* **@JsonIgnore**: associato al campo *projects*, permette di escluderlo dalla serlizzazione;
* **@JsonBackReference**: associato al campo *department*, permette di dare una direzionalità alla relazione molti a uno con Department, fondamentale per evitare il problema della ricorsione infinita (QUI NON SO SE SPIEGARE).

Analogamente sono state realizzate le classi corrispondenti alle entità Project, Department e Site.

A questo punto si procede con la realizzazione delle classi repository: ciascuna entità ha una propria repository corrispondente. Dunque viene estesa l'interfaccia **JPARepository<T, ID>** con T il tipo della entità che si vuole gestire, mentre ID è il tipo della chiave primaria dell'entità T. La repository JPA deriva da diverse interfacce, tra le quali:

* **CrudRepository<T, ID>**: la quale contiene le API per gestire le classiche operazioni CRUD;
* **PagingAndSortingRepository<T, ID>**: la quale contiene le API per gestire la pagination e il sorting;

Dunque estendendo la JPARepository per ciascun tipo è possibile avere a disposizione diversi metodi per eseguire operazioni già implementate come: findAll, count, existById, SaveAndFlush, ecc...
Qualora invece si volesse rendere disponibili nuovi metodi è possibile dichiararli nell'estensione della repository, senza necessariamente implementarli, poiché è sufficiente attribuire il nome corretto al metodo. Più specificatamente il nome del metodo corrispondente alla query che si vuole render disponibile è composto da un introducer che può essere uno tra: *find*, *read*, *query*, *count* o *get*, e successivamente il criterio seguito dalla keywork *By*, quindi ad esempio se si volesse fare una ricerca per salario, è sufficiente dichiarare un metodo chiamato *findBySalary*.
Infine ritroviamo il caso in qui si vuole realizzare una query complessa o personalizzata, in questo caso ci viene in aiuto la annotazione **@Query** alla quale è possibile passare come attributo la query che desideriamo in linguaggio JPQL. Di seguito è possibile visualizzare quanto spiegato nell'implementazione della **EmployeeRepository** in figura 4.5:

```java
@Repository
public interface EmployeeRepository extends JpaRepository<Employee, String> {

    1 usage
    List<Employee> findByName(String name);

    1 usage
    List<Employee> findBySurname(String surname);

    1 usage
    List<Employee> findByDepartment_id(Long id);

    1 usage
    @Query("SELECT DISTINCT e FROM Employee e WHERE e.birth BETWEEN ?1 AND ?2")
    List<Employee> findByBornDateRange(@Param("from") Date from, @Param("to") Date to);

}
```

Figura 4.5: Esempio di implementazione della repository di Employee in Spring Boot.

Dunque oltre ai classici metodi di ricerca disponibili già dopo l'estensione dell'interfaccia JPARepository<T, ID>, sono stati realizzati alcuni metodi per la ricerca di impiegati per nome, per cognome e per id di dipartimento in cui lavorano. Infine è stata realizzata una query personalizzata per la ricerca di impiegati nati in un range di date.
Infine è possibile notare in figura 4.5 l'annotazione **@Repository** attribuita alla'interfaccia, fondamentale al fine di indicare che la classe fornisce meccanismi per modellare i dati dell'applicativo.

Service Lo strato di servizio è lo strato che si trova tra lo strato di controller e quello di repository, il suo compito è facilitare la comunicazione tra controller e repository e inoltre contiene la business logic dell'applicativo. Per ciascun repository, dunque per ciascuna entità, è stato realizzato un servizio specifico per gestirne le logiche.
Al fine di rispettare i principi SOLID della programmazione, per questioni di loose coupling e semplicità nel testing, è stato scelto di implementare il pattern secondo il quale per ogni entità viene realizzata una interfaccia del servizio ed la sua implementazione, come mostrato in figura 4.6 nel caso del servizio per l'entità Employee.

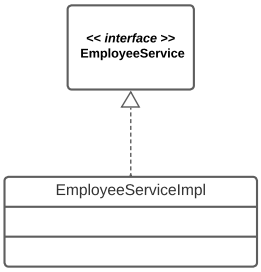

Figura 4.6: Esempio di implementazione dell'interfaccia EmployeeService.

Di seguito, in figura 4.7, viene riportato un esempio di servizio implementato, in questo caso si tratta dell'implementazione del servizio per l'Employee, ovvero della classe **EmployeeServiceImpl**.

```
@Service
public class EmployeeServiceImpl implements EmployeeService{
    @Autowired
    private EmployeeRepository employeeRepository;
    @Override
    public List<Employee> SelAll() {...}
    @Override
    public Optional<Employee> SelById(String id){...}
    @Override
    public List<Employee> SelByName(String name){...}
    @Override
    public List<Employee> SelBySurname(String surname){...}
    @Override
    public List<Employee> SelByDepartment(Long id){...}
    @Override
    public List<Employee> SelByBornInRange(Date from, Date to){...}
    @Override
    public void InsEmployee(Employee employee){...}
    @Override
    public void DelEmployee(Employee employee){...}

}
```

Figura 4.7: Classe EmployeeServiceImpl.

In figura 4.7 è possibile notare come la classe sia caratterizzata dall'annotazione **@Service** la quale viene utilizzata per indicare classi che contengono la business logic e viene utilizzata dunque per marcare la classe come service provider.
Continuando ed andando ad analizzare i campi dati è possibile visualizzare la dipendenza che la classe *EmployeeServicImpl* ha con la repository *EmployeeRepository*. In Spring questa dipendenza viene risolta con l'annotazione **@Autowired**, la quale permette di eseguire la dependency injection del bean *employeeRepository*.
Infine sono presenti tutti i metodi, ciascuno con annotazione **@Override** poiché sono stati dichiarati anche nell'interfaccia implementata da *EmployeeServiceImpl*. Sono stati resi disponibili metodi semplici che vanno ad invocare, grazie alla dipendenza con la repository, le query già disponibili con la *JPARepository<T, ID>* e la query vista precedenemente *SelByBornInRage*.
Per concludere nello strato di servizio vengono sollevate eventuali eccezioni. La gestione delle eccezzioni è stata effettuata implementanto un gestore di eccezioni specifico attraverso l'estensione del gestore *HandlerExceptionResolver*. Questo è stato fatto al

fine di eliminare il problema presente con il gestore di eccezioni di default, ovvero il *DefaultHandlerExceptionResolver*, il quale non permette di impostare nulla nel body della risposta al client e quindi per alcune tipologie di errori potrebbe risultare complesso comprenderne la natura. Non verrà mostrata l'implementazione della ridefinizione del gestore di eccezioni per evitare di approfondire eccessivamente i dettagli prettamente tecnici.

Controller Infine troviamo i controller, ovvero l'ultimo strato che si occupa di gestire le richieste che il server riceve attraverso il protocollo HTTP, e di mapparle inoltre ai relativi metodi. In figura 4.8 il controller di Employee, ovvero la classe *EmployeeController*:

```
@RestController
@RequestMapping(value = "api/employee")
public class EmployeeController {

    @Autowired
    private EmployeeService employeeService;

    @CrossOrigin
    @GetMapping(value = "/all", produces = "application/json")
    public ResponseEntity<List<Employee>> listAllEmployee(){...}
    @GetMapping(value = "name/{name}", produces = "application/json")
    public ResponseEntity<List<Employee>> listEmployeeByName(@PathVariable("name") String name){...}
    @GetMapping(value = "surname/{surname}", produces = "application/json")
    public ResponseEntity<List<Employee>> listEmployeeBySurname(@PathVariable("surname") String surname){...}
    @GetMapping(value = "department/{department_id}", produces = "application/json")
    public ResponseEntity<List<Employee>> listEmployeeByDepartment(@PathVariable("department_id") Long id){...}
    @GetMapping(value = "born/{from}/{to}", produces = "application/json")
    public ResponseEntity<List<Employee>> listEmployeeBornInDateRange(@PathVariable("from")
            @DateTimeFormat(pattern="yyyy-MM-dd") Date from, @PathVariable("to") @DateTimeFormat(pattern="yyyy-MM-dd") Date to){...}
    @PostMapping(value = "/insert", consumes = "application/json")
    public ResponseEntity<Employee> createEmployee(@RequestBody Employee employee){...}
    @DeleteMapping(value = "/delete/{id}")
    public ResponseEntity<?> deleteEmployee(@PathVariable("id") String id){...}
    @PutMapping(value = "/update/{id}", consumes = "application/json")
    public ResponseEntity<?> updateEmployee(@PathVariable("id") String id, @RequestBody Employee employee) {...}
}
```

Figura 4.8: Classe EmployeeController.

Questa classe permette di gestire e mappare le richieste a seconda dell'url dal quale proviene la richiesta. In figura 4.8 è possibile visualizzare due annotazioni associate alla classe, l'annotazione **@RestController**, utilizzata per definire la classe come un controller di tipo REST, mentre l'annotazione **@RequestMapping** indica l'url al quale il client dovrà mandare le richieste per quello specifico controller.
Proseguendo è possibile indivudare una dipendenza della classe con lo strato di servizio, infatti con l'annotazione **@Autowired** e dunque con la dependency injection viene risolta la dipendenza. La classe *EmployeeController* necessita una dipendenza con la classe *EmployeeServiceImpl* poiché dovrà andare ad invocarne i metodi.
Dunque proseguendo è possibile visualizzare i vari metodi, tutti hanno una annotazione che può essere :

* **@GetMapping**: indica che si tratta di un metodo per la risoluzione di una richiesta GET; specifica l'url al quale ricevere la richiesta e ciò che viene ritornato, ovvero un file JSON;
* **@PostMapping**: indica che si tratta di un metodo per la risoluzione di una richiesta POST, specifica l'url al quale ricevere la richiesta e ciò che richiede in input, ovvero un file JSON passato attraverso il body della richiesta HTTP;

* **@DeleteMapping**: indica che si tratta di un metodo per la risoluzione di una richiesta DELETE, specifica l'url al quale ricevere la richiesta;
* **@PutMapping**: indica che si tratta di un metodo per la risoluzione di una richieste PUT, specifica l'url al quale ricevere la richiesta e ciò che richiede in input, ovvero un file JSON passato attraverso il body della richiesta HTTP;

Oltre alle annotazioni sopra riportate, sono presenti tra gli argomenti le annotazione **@PathVariable** la quale vuole indicare che si tratta di una variabile che verrà fornita nell'url nel posto definito dal nome specificato, **@RequestBody** ovvero un argomento che verrà fornito nel body della chiamata HTTP e infine **@DateTimeFormat** per specificare il formato del tipo di dato *Date* che verrà passato dal client nell'url della richiesta. Sono state rese dunque disponibili le seguenti query:

* **listAllEmployee**: ritorna tutti gli impiegati presenti;
* **listEmployeeByName**: ritorna tutti gli impiegati con una determinato nome;
* **listEmployeeBySurname**: ritorna tutti gli impiegati con un determinato cognome;
* **listEmployeeByDepartment**: ritorna tutti gli impiegati di un dipartimento;
* **listEmployeeByBornInDataRange**: ritorna tutti gli impiegati nati in un determinato range di date;
* **updateEmployee**: vengono aggiornati i campi dati di un impiegato;
* **createEmployee**: aggiunta di un nuovo impiegato;
* **deleteEmployee**: rimozione di un impiegato.

Testing API Essendo un prototipo incentrato sulla realizzazione delle API, sono stati svolti in maniera semplice e veloce i test per gli strati di servizio e repository, dunque non verranno riportati. Per quanto riguarda i testi sui controller stati svolti dei test più approfonditi.
Per eseguire i test sui metodi del controller è stato scelto di utilizzare il framework JUnit5 (REINDIRIZZARE AL CAPITOLO TECNOLOGIE).
Il primo test realizzato è uno *SmokeTest* e da qui il nome della classe realizzata per testare che il contesto Spring abbia effettivamente creato i controller dell'applicazione. In figura 4.9 la classe *SmokeTest* con una dipendenza per ciascun controller risolta con la dependency injection grazie all'annotazione **@Autowired**. Sono poi presenti quattro metodi, uno per controller, per verificarne se sono stati creati nel contesto. Infine ciascun metodo deve essere annotato con **@Test** per indicare al framework JUnit5 che si tratta di un metodo di test.
Come possiamo notare sempre nell'immagine 4.9 è presente l'annotazione **@SpringBootTest**, necessaria per indicare a Spring Boot dove si trova la principale classe di configurazione, e dunque avviare il contesto Spring. Da notare inoltre che ciascun metodo è dichiarato in maniera da poter sollevare eccezzioni se necessario.

```java
@SpringBootTest(classes = PrototypeManagerApplication.class)
public class SmokeTest {
    @Autowired
    private EmployeeController employeeController;
    @Autowired
    private ProjectController projectController;
    @Autowired
    private DepartmentController departmentController;
    @Autowired
    private SiteController siteController;
    @Test
    public void contextLoadsEmployee() throws Exception {...}
    @Test
    public void contextLoadsDepartment() throws Exception {...}
    @Test
    public void contextLoadsProject() throws Exception {...}
    @Test
    public void contextLoadsSite() throws Exception {...}
}
```

Figura 4.9: Classe *SmokeTest* sulla creazione dei controller.

Passiamo ora ai test metodi del controller *EmployeeController*. Nell'immagine 4.10 è raffigurata la classe *EmployeeControllerTest* con metodi i vari test da effettuare sul controller.

```java
@WebMvcTest(EmployeeController.class)
@ExtendWith(SpringExtension.class)
public class EmployeeControllerTest {
    @Autowired
    private MockMvc mockMvc;
    @Autowired
    private ObjectMapper objectMapper;
    @MockBean
    private EmployeeService employeeService;
    //Testing listAllEmployee()
    @Test
    void listAllEmployeeWihtNotEmptyList() throws Exception{...}
    @Test
    void listAllEmployeeWihtEmptyList() throws Exception{...}
    //Testing listEmployeeByName()
    @Test
    void listEmployeeByName() throws Exception{...}
    @Test
    void listEmployeeByNameEmpty() throws Exception{...}
    //Testing listEmployeeBySurname()
    @Test
    void listEmployeeBySurname() throws Exception{...}
    @Test
    void listEmployeeBySurnameEmpty() throws Exception{...}
    //Testing listEmployeeByDepartment()
    @Test
    void listEmployeeByDepartmentWithExistingDepartmentAndNotEmptyList() throws Exception{...}
    @Test
    void listEmployeeByDepartmentWithNotExistingDepartment() throws Exception{...}
    @Test
    void listEmployeeByDepartmentEmptyList() throws Exception{...}
    //Testing listEmployeeBornInDateRange()
    @Test
    void listEmployeeBornInDateRangeWithRightInputAndNotEmptyList() throws Exception{...}
    @Test
    void listEmployeeBornInDateRangeWithWrongInputDateRange() throws Exception{...}
    @Test
    void listEmployeeBornInDateRangeEmptyList() throws Exception{...}
    //Testing createEmployee()
    @Test
    void createEmployee() throws Exception{...}
    //Testing deleteEmployee()
    @Test
    void deleteNotExistingEmployee() throws Exception{...}
    @Test
    void deleteExistingEmployee() throws Exception{...}
    //Testing updateEmployee()
    @Test
    void updateNotExistingEmployee() throws Exception{...}
    @Test
    void updateExistingEmployee() throws Exception{...}
}
```

Figura 4.10: Classe *EmployeeControllerTest*.

Nell'immagine 4.10 la classe *EmployeeControllerTest* ha due annotazioni: la prima è **@WebMvcTest**, utile poiché permette di caricare nel contesto di test esclusivamente il controller di Employee e inoltre permette di tralsciare tutte le configurazioni classiche per lasciar spazio alle configurazioni di test, mentre **@ExtendWhit** permette di integrare nel contesto di test la *SpringExtension* utile poiché fornisce supporto per il testing (Non necessaria qui forse...).

Nella classe di test *EmployeeControllerTest* sono dichiarate tre dipendenze fondamentali:

* **MockMvc**: dipendenza risolta cona la dependency injection, viene utilizzata sucessivamente per simulare chiamate HTTP e verificarne la risposta;
* **ObjectMapper**: dipendenza risolta con la dependency injection, oggetto utilizzato nei test per la serializzazione e deserializzazione degli oggetti JSON in oggetti java e viceversa;
* **EmployeeService**: servizio che viene utilizzato nei test, per questo motivo ha l'annotazione **@MockBean**, la quale permette di aggiungere un mock dell'oggetto *EmployeeService*.

Per finire la sezione sulle REST API viene di seguito riportato in figura 4.11 l'implementazione nel dettaglio del metodo di test *createEmployee* seguendo il pattern *Arrange - Act - Assert*.

```
@Test
void createEmployee() throws Exception{
    //Arrange
    Employee employee = new Employee();
    employee.setName("Mario");
    employee.setSurname("Rossi");
    LocalDate date = LocalDate.of( year: 1992,  month: 05,  dayOfMonth: 03);
    employee.setBirth(Date.from(date.atStartOfDay(ZoneId.systemDefault()).toInstant()));
    //Act & Assert
    mockMvc.perform(post( urlTemplate: "/api/employee/insert").contentType(MediaType.APPLICATION_JSON)
            .contentType(objectMapper.writeValueAsString(employee))).andExpect(status().isCreated())
            .andDo(print());
}
```

Figura 4.11: Metodo di test *createEmployee* della classe *EmployeeControllerTest*.

Nella prima parte (arrange) viene creato un nuovo Employee e gli viene assegnato un nome, un cognome e una data di nascita, successivamente nella fase successiva (act), viene eseguita la chiamata GET utilizzando l'oggetto *MockMvc* e passando nel body della richiesta HTTP l'oggetto Java Employee trasformato in JSON grazie all'*ObjectMapper*. Infine nella fase finale (Assert), si verifica che lo stato di ritorno sia *created*, in caso contrario il test fallisce.

Frontend

Per quanto riguarda il frontend dell'applicativo è stato utilizzato il framework Angular per realizzare una single-page application. Si tratta di un frontend minimale, realizzato con il solo scopo di comprendere e sviluppare la parte di comunicazione con il server utilizzando al massimo le REST API disponibili. Per questo motivo non è stata effettuata alcuna analisi sull' utenza e sono stati tralascati completamente aspetti quali estetica grafica, accessibilità e responsive design.

Architettura L'architettura della web application realizzata in Angular prevede l'utilizzo del pattern Model-View-Controller:

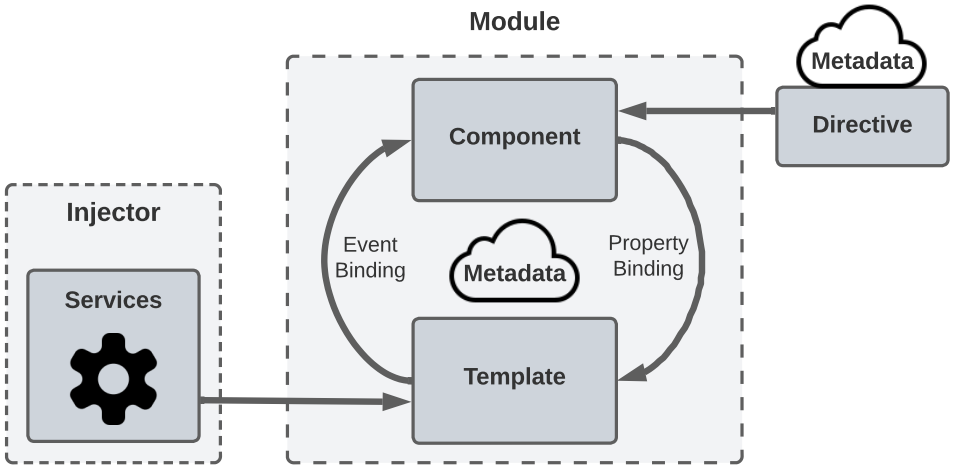

Figura 4.12: Architettura della Web Application del prototipo.

Come visibile in figura 4.12 l'utente interagisce con la user interface (View) e ne modifica lo stato. Le resources (Model) vengono modificate dai services (Controller) anche su invocazione dei servizi da parte della user interface. I services sono coloro che si occupano della comunicazione con il server di backend attraverso le richiesta e risposte HTTP.

Funzionalità Per le ragioni spiegate precedentemente saranno rese disponibili nell'interfaccia grafica quelle funzionalita che andranno a permettere il massimo utilizzo delle richieste API al backend.
Come fatto in precedenza, verrà mostrata e spiegata esclusivamente la parte riguardante l'entità Employee per questioni di praticità, questo perché le altre entità sono state trattate in maniera analoga.
Dunque, in linea con le query rese disponibili dal backend riportate sulla sezione riguardante i controller, devono essere presenti le seguenti funzionalità:

* **Visualizzazione degli Employee**: devono poter essere visualizzati gli impiegati in lista, o ricercati attraverso i campi: nome, cognome, data di nascita o dipartimento di appartenenza;

* **Aggiunta di un Employee**: deve essere possibile aggiungere un nuovo impiegato potendone specificare: nome, cognome, salario, data di nascita, dipartimento di appartenenza e infine i progetti che segue;

* **Aggiornamento di un Employee**: deve essere possibile aggiornare i campi dati di un impiegato specificandone l'id e il/gli campo/i da modificare;

* **Eliminazione di un Employee**: deve essere possibile eliminare un impiegato specificandone l'id.

Nel paragrafo sucessivo verrà mostrato come sono state implementate le funzionalità appena introdotte.

Implementazione FE Inizialmente sono state create le interfacce per poter creare gli oggetti corrispondenti alle quattro entità del prototipo. Di seguito in figura 4.13 l'esempio della dichiarazione della interfaccia dell'entità Employee:

```
export interface Employee {
  id: string;
  name: string;
  surname: string;
  salary: number;
  birth: Date;
  department: Department;
  project: [Project];
}
```

Figura 4.13: Interfaccia di Employee.

Analogamente sono state create le interfacce delle altre entità.
Proseguendo, è stata realizzata la realizzazione dell'interaccia grafica inserendo nella pagina iniziale la possibilità di effettuare due scelte: la prima scelta riguarda l'entità, mentre la seconda riguarda l'operazione che si vuole eseguire sull'entità selezionata. In figura 4.14 è possibile visualizzare la prima pagina:

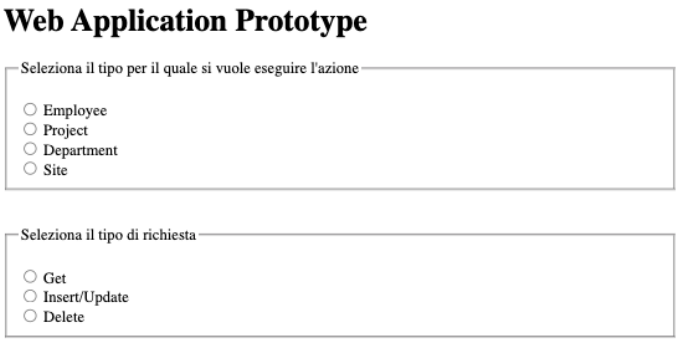

Figura 4.14: Prima pagina della Web Application.

Una volta selezionate le due opzioni viene aggiunta dinamicamente la possibilità di specificare i dettagli della richiesta. Considerando solo l'entità Employee per le ragioni spiegate precedentemente, vengono di seguito mostrate le diverse interfacce che variano al variare della scelta selezionata e il servizio che permette di eseguire la chiamata alle API del backend.
In figura 4.15abbiamo il caso in cui è stato selezionata l'entità Employee con operazione GET:

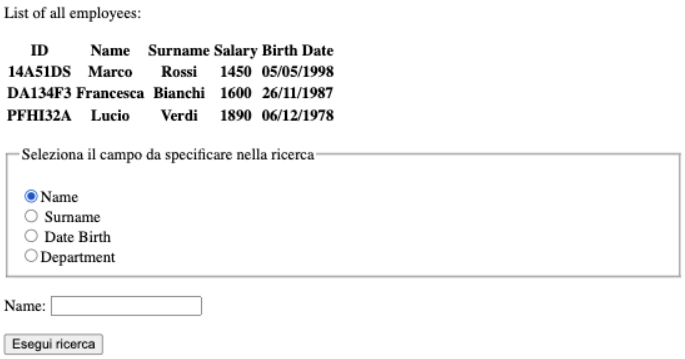

Figura 4.15: Opzioni disponibili con operazione GET selezionata.

Viene dunque visualizzata immediatamente una lista contenente tutti gli impiegati, per ciascun impiegato vengono mostrati id, nome e cognome, salario e data di nascita. Successivamnete viene visualizzata la scelta riguardante il campo per il quale eseguire la ricerca, in questo caso è stato scelto il nome, dunque inserendo nell'apposito campo di input il nome e selezionendo il bottone è possibile effettuare la ricerca per nome. La scelta dell'opzione get permette alla vista di invocare immediatamente il servizio che esegue una chiamata alle REST API del backend per ricevere la lista di tutti gli employee, stessa cosa accade anche quando premiamo il bottone "Esegui ricerca " dopo aver inserito il nome. Il servizio in questione si chiama *EmployeeService* e di seguito in figura 4.16 ne è riportata l'implementazione:

```
export class EmployeeService {

  url : String = 'http://localhost:8080/api/employee/';

  constructor(private http: HttpClient) { }

  allEmployees(){
    return this.http.get(this.url + "all")
  }
  employeeByName(name: string){…
  }
  employeeBySurname(surname: string){…
  }
  employeeByBornInDateRange(from: Date, to: Date){…
  }
  employeeByDepartment(id: string){…
  }
  createOrUpdateEmployee(employee: Employee){…
  }
  deleteEmployee(id: string){…
  }
}
```

Figura 4.16: Classe *EmployeeService*.

Il servizio realizzato per le chiamate alle REST API del backend, permette di essere invocato direttamente dalla vista alla selezione di un elemento HTML. È inoltre presente una dipendenza con l'oggetto *HttpClient*, il quale viene passato come parametro

al costruttore ed è necessario per eseguire le chiamate alle API.

Proseguendo vengono visualizzati ora i casi in cui si sceglie come operazione quella di modificare o aggiungere un nuovo impiegato. In figura 4.17 è possibile visualizzare la porzione di interfaccia grafica per l'inserimento o l'aggiornamento di un impiegato:

Figura 4.17: Opzioni disponibili con operazione Insert/Update selezionata.

Dunque una volta inseriti tutti i valori per ciascun campo, tranne il campo ID che viene generato automaticamente, è possibile aggiungere un nuovo impiegato. Qualora invece si specificasse anche l'id, allora si tratterebbe di una modifica di un employee già esistente, questo solo se l'id inserito corrisponde veramente all'id di un employee. Anche in questo caso selezionando il bottone "Insert or update Employee!" viene invocato il metodo del servizio riportato in figura 4.16.
Per finire in figura 4.18 il caso in cui venga selezionata l'opzione delete:

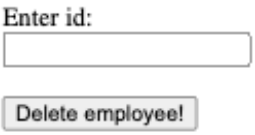

Figura 4.18: Opzioni disponibili con operazione Delete selezionata.

Testing FE I test di unità sono stati implementati seguendo il pattern *Arrange - Act - Assert* sui metodi del servizio riportato in figura 4.16. Di seguito nell'immagine 4.19 è possibile visualizzare la funzione *describe* la quale viene utilizzata per raggruppare un insieme di test:

```
describe('EmployeeService', () => {
  let service: EmployeeService;
  let httpMock: HttpTestingController;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpClientTestingModule],
      providers: [EmployeeService],
    });
    service = TestBed.inject(EmployeeService);
    httpMock = TestBed.inject(HttpTestingController);
  });
  afterEach(() => {
    httpMock.verify();
  });
  it('should retrieve all employees'), () => {…
  }
  it('should retrive all employees by name'), () => {…
  }
  it('should retrive all employees by surname'), () => {…
  }
  it('should retrive all employees by department'), () => {…
  }
  it('should create one employee'), () => {…
  }
  it('should update one employee'), () => {…
  }
  it('should delete one employee'), () => {…
  }
});
```

Figura 4.19: Test per i metodi del servizio *EmployeeService*.

Viene utilizzato il modulo Angular *HttpClientTestModule*, necessario al fine di importare il servizio inettabile *HttpTestingController*, utilizzato per il mocking e il flushing per eliminare le microtasks in sospeso. Sono inoltre presenti le funzioni *BeforeEach()* utilizzata per configurare l'ambiente di test, dunque importare i moduli necessari e risolvere le dipendenze, e la funzione *afterEach()* utilizzata per controllare, dopo l'esecuzion di ciascun test, che non siano rimaste richieste in sospeso. Infine, con la funzione *it()* viene definito un titolo per ciascun test e viene implementato il test. Più nello specifico in figura 4.20 si può visualizzare l'implementazione di un test per il metodo *allEmployee* del servizio *EmployeeService*:

```
it('should retrieve all employees'), () => {
  //Arrange
  const employeesList: Employee[] = [
    {id: "F3BASD", name: "Mario", surname: "Rossi", salary: 1500, birth: new Date("05/04/1999")},
    {id: "AIF07S", name: "Giulio", surname: "Gialli", salary: 1500, birth: new Date("02/01/1988")},
    {id: "LOSF2D", name: "Serafino", surname: "Bianchi", salary: 1500, birth: new Date("22/11/1979")},
    {id: "PIFA2A", name: "Marta", surname: "Verdi", salary: 1500, birth: new Date("30/01/1995")},
  ];
  //Act & Assert
  service.allEmployees().subscribe(employees => {
    expect(employees).toBe(employeesList);
  })

  const request = httpMock.expectOne(`${service.url}/all`);

  expect(request.request.method).toBe('GET');
  request.flush(employeesList);
}
```

Figura 4.20: Test sul metodo *allEmployee()* di *EmployeeService*.

Viene inizialmente creato un array di Employee che ci si aspetterà di ricevere dalla

chiamata, e sucessivamente viene controllato un test per verificare che ciò accada effettivamente con la funzione *expect()*. Infine viene controllato che l'url sia corretto, che venga effettuata una sola chiamata GET e che si tratti effettivamente di una richiesta GET.

4.1.2 Migrazione da REST a GraphQL

Nella seguente sottosezione verrà mostrato come è stato migrato sia il backend che il frontend del prototipo realizzato da REST API a GraphQL API.

Migrazione Backend

Il backend realizzato in Spring Boot con l'aiuto del modulo Spring Data REST per la realizzazione dei controller di REST API, deve essere riscritto in parte utilizzando il modulo Spring GraphQL per la realizzazione dei controller in GraphQL.
Seguendo la metodologia *Schema First* la prima cosa che è stata creata è il GraphQL Schema.

GraphQL Schema Il GraphQL Schema è necessario al fine di definire i tipi, le query, le mutation e le subscription che il server GraphQL renderà disponibili al client. Inizialmente è necessario fare un'analisi sulle enetità presenti e comprendere quali entità riportare nel GraphQL Schema. Infatti non tutte le entità devono essere necessariamente riportate nel GraphQL Schema, ma andranno riportate solo le entità che verranno utilizzate API. Nel caso del prototipo le quattro entità presenti, ovvero Employee, Project, Department e Site devono essere riportate tutte nel GraphQL schema poiché sono tutte accessibili dal client.
Partendo dunque dalle quattro entità, sarà necessario riportarle nel GraphQL Schema, sia come entità di input che di output nelle query, questo è necessario poiché GraphQL distingue i due tipi. In figura 4.21 è possibile visualizzare l'implementazione dell'entità *Employee* e *EmployeeInput*.

```
type Employee {
    id: ID!
    name: String
    surname: String
    salary: Int
    birth: Date
    projects: [Project]
    department: Department
}
```

```
input EmployeeInput{
    id: ID
    name: String
    surname: String
    salary: Int
    birth: Date
    departmentId: ID
    projectsId: [ID]
}
```

Figura 4.21: Tipi *Employee* e *EmployeeInput* nel GraphQL Schema.

Come si può notare i due tipi *Employee* e *EmployeeInput* non corrispondono completamente. Infatti i campi *projects* e *department* di *Employee* non sono presenti in *EmployeeInput*, o meglio sono presenti ma in diversa forma. Questo è dovuto al fatto che il tipo *Employee* è un tipo che verrà ritornato al client su richiesta, dunque

dovrà contenere le varie istanze dei progetti a cui un impiegato sta partecipando e anche l'oggetto dipartimento. Mentre per quanto riguarda il tipo *EmployeeInput* che corrisponde al tipo di input per l'impiegato, qualora ad esempio il client volesse creare attraverso una mutation un nuovo impiegato, sarà sufficiente specificare l'id del dipartimento in cui lavora o gli id dei progetti ai quali sta partecipando, non serve passare l'interno oggetto Project o Department.
A questo punto è possibile definire le query, le mutation ed eventualmente le subscription che si vogliono rendere disponibili al client. In figura 4.22 è possibile visualizzare la dichiarazione delle query, mutation e della subscription riguardanti l'entità Employee.

```
type Query {
    allEmployees: [Employee]
    employeeByName(name: String!): Employee
    employeeBySurname(surname: String!): Employee
    employeByDepartment(id: ID!): Employee
    employeeBornInDateRange(from: Date, to: Date): [Employee]
}
type Mutation {
    addEmployee(employee: EmployeeInput!): Employee
    updateEmployee(employee: EmployeeInput!): Employee
    delEmployee(id: ID!): Employee
}
type Subscription{
    newEmployeeAdded: [Employee]
}
```

Figura 4.22: Implementazione delle query, mutation e subscription nel GraphQL Schema.

Come è possibile notare sono state rese disponibili le medesime query e mutation rese disponibili nelle REST API. Il tipo di ritorno è il tipo *Employee* per tutte le query, mutation e subscription.
È stata aggiunta inoltre una subscription per sfruttare a pieno le funzionalità di GraphQL. Questa subscription permette dunque, come spiegato nel capitolo **??**, che il client riceva i nuovi impiegati aggiunti senza per forza richiederli.

Controller refactoring A questo punto resta solo la riscrittura dei controller, mentre lo strato di servizio e quello di repository rimangono invariati durante la migrazione, infatti le logiche del server e l'accesso e la gestione del database rimangono invariati. Tuttavia prima di analizzare l'implementazione del controller GraphQL, è necessario dichiarare le classi Java corrispondenti ad i vari errori ed ai tipi di input; questo è necessario poiché qualora dovessere essere ritornato un errore o dovesse essere ricevuto in input ad esempio il tipo dichiarato precedentemente *EmployeeInput*, il quale non corrisponde al tipo Employee dichiarato in Java, si creerebbero degli errori durante il mapping tra i tipi Java e i tipi dichiarati nel GraphQL Schema. Per questo motivo è stato dichiarato un Java record per ciascun errore/successo, più le classi relative ai vari tipi di input.
L'implementazione del controller risulta differente rispetto a quella vista con le REST API. Di seguito in figura 4.23 è possibile visualizzarne la struttura ed i metodi:

```java
@Controller
public class EmployeeController {

    @Autowired
    private final EmployeeService employeeService;
    @QueryMapping
    Object allEmployees(){...}
    @QueryMapping
    Object employeeByName(@Argument String name) {...}
    @QueryMapping
    Object employeeBySurname(@Argument String surname) {...}
    @QueryMapping
    Object employeeByDepartment(@Argument String id) {...}
    @QueryMapping
    Object employeeBornInDateRange(@Argument Date from, @Argument Date to) {...}
    @MutationMapping
    Object addEmployee (@Argument EmployeeInput employee){...}
    @MutationMapping
    Object updateEmployee (@Argument EmployeeInput employee){...}
    @MutationMapping
    Object delEmployee(@Argument String id){...}
    @SubscriptionMapping
    Object newEmployeeAdded(){...}
}
```

Figura 4.23: Classe *EmployeeController*.

Come si può notare in figura 4.23 si tratta di un controller molto simile al medesimo controller in REST, ma con annotazioni differenti. Innanzitutto ha l'annotazione **@Controller** e non @RestController, poiché quest'ultima serve per indicare esclusivamente Spring Controller per REST API, mentre la prima è più generica ed indica esclusivamente che si tratta di un controller. Poi è possibile notare come le annotazioni sui metodi siano differenti, per le query si usa l'annotazione **@QueryMapping**, per le mutation l'annotazione **@MutationMapping**, mentre per le subscription l'annotazione **@SubscriptionMapping**, infine per gli argomenti dei metodi l'annotazione **@Argument**. Queste annotazioni sono fondamentali poiché permettono di effettuare il mapping tra i tipi, query, mutation o subscription definiti nel GraphQL Schema e quelli definiti nel controller. Per questo motivo il nome di ciascun metodo o tipo negli argomenti deve corrispondere esattamente al nome nel GraphQL Schema, altrimenti il mapping non andrà a buon fine e verranno generati degli errori.
L'implementazione dei metodi del controller risulta identica a quella dei corrispettivi metodi nel REST controller. Tuttavia, trattandosi di una particolare funzionalità disponibile esclusivamente in GraphQL, in figura 4.24 viene riportata l'implementazione della subscription *newEmployeeAdded*:

```java
@SubscriptionMapping
Publisher<List<Employee>> newEmployeeAdded(){
        return subscriber -> Executors.newScheduledThreadPool( corePoolSize: 1).scheduleAtFixedRate(() -> {
            List<Employee> employees= employeeService.SelAll();
            subscriber.onNext(employees);
        }, initialDelay: 0, period: 2, TimeUnit.SECONDS);
}
```

Figura 4.24: Subscription *newEmployeeAdded*.

L'implementazione della subscription permette al subscriber, in questo caso il client, di ricevere la lista aggiornata appena viene aggiunto un nuovo impiegato. L'oggetto *Publisher<>* permette di inviare la lista aggiornata tutti i subscribers in maniera dinamica, così facendo il client non dovrà richiedere periodicamente la lista con la query *allEmployee*, bensì riceverà automaticamente i nuovi impiegati. Mancano test in graphQL

Strutture delle risposte HTTP in GraphQL e gestione degli errori Come spiegato in precedenza Le risposte che il server GraphQL appena realizzato ritorna al client in seguito ad una query, mutation o subscription sono dei file JSON. Questi JSON possono contenere tre tipi di campi:

* **Data**: il campo data viene ritornato nella risposta solo qualora la query abbia iniziato l'esecuzione, in caso si verifichino errori durante l'esecuzione possono essere ritornati dati parziali, oppure il campo *Data* può essere ritornato con valore *null*;
* **Errors**: il campo errors viene ritornato nel JSON solo ed esclusivamente quando si sono verificati degli errori prima o durante l'esecuzione della query, questo ritorna i vari errori presenti. Ogni errore ritornato nel campo *errors* può contenere a sua volta altri campi che indicano la posizione dell'errore e/o un messaggio;
* **Extensions**: si tratta di un campo aggiuntivo a disposizione degli sviluppatori, può essere utilizzato ad esempio per ritornare il timestamp dell'esecuzione della query.

Analizzando gli errori che si possono verificare durante l'esecuzione di una query, mutation o subscription nel server GraphQL, si possono raggruppare in tre categorie:

* **Errori di sintassi**: si tratta di errori nella sintassi della query. Quando viene inviata una query sintatticamente scorretta viene ritornato un JSON contenente il campo "errors", il quale indica che si tratta di un "Syntax Error". In questo caso l'eseguzione della richiesta non parte nemmeno;
* **Errori di validazione**: si verificano quando vengono specificati nella richiesta tipi o campi non presenti o non corretti. In questo caso l'errore si verifica durante l'esecuzione, dunque viene ritornato sia il campo *data*, con valore null o con i dati parziali che il server è riuscito a recuperare, che il campo *errors* che indica la presenza di uno o più errrori di data fetching;
* **Errori nei resolver:** si verificano durante la risoluzione della query e non sono legate al fatto che la query sia formulata male o con tipi sbagliati. Questi errori devono essere gestiti lato server;

Mentre quando si verificano gli errori di sintassi o di validazione la risposta generata con l'errore è chiara riguardo la natura dell'errore, nel terzo caso il messaggio dell'errore ritornato è quasi sempre incomprensibile. Questo accade perché la gestione delle eccezzioni durante l'esecuzione di una richiesta è presa in carico dall'interfaccia predefinita *DataFetcherExceptionHandler*, la quale permette la dichiarazione di più risolutori di eccezzioni detti *DataFetcherExceptionResolver* che vengono invocati sequenzialmente qualora si presentasse un errore, fino a quando uno di questi non riesce a risolvere l'eccezzione, altrimenti viene ritornato un errore con messaggio incomprensibile e categoria *"INTERNAL_ERROR"*. Questo per uno sviluppatore può essere un problema

poiché non è facile individuare la provenienza dell'errore. Per questo motivo sono stati creati differenti resolvers per le varie eccezioni che possono verificarsi. In figura 4.25 l'esempio di implementazione di un resolver che riscrive il metodo di risoluzione delle eccezioni *resolveToSingleError* al fine di risolvere l'eccezione *EmployeeNotFound*.

```
@Component
public class GraphQLExceptionResolver extends DataFetcherExceptionResolverAdapter {
    @Override
    protected GraphQLError resolveToSingleError(Throwable ex, DataFetchingEnvironment env) {
        if (ex instanceof EmployeeNotFoundError) {
            return GraphqlErrorBuilder.newError() GraphqlErrorBuilder<capture of ?>
                    .errorType(ErrorType.NOT_FOUND) capture of ?
                    .message(ex.getMessage())
                    .path(env.getExecutionStepInfo().getPath())
                    .location(env.getField().getSourceLocation())
                    .build();
        } else {
            return null;
        }
    }
}
```

Figura 4.25: Classe *GraphQLExceptionResolver* con metodo *resolveToSingleError* ridefinito per la risoluzione dell'eccezione *EmployeeNotFoundError*.

Com'è possibile notare viene ritornato un oggetto di tipo *GraphQLError* con i campi impostati correttamente. Dunque quando viene sollevata l'eccezione, è ora possibile specificare il messaggio che il resolver andrà a ritornare nell'errore, oltre ad impostarne il tipo corretto dell'errore.

Migrazione Frontend

La migrazione del frontend richiede uno sforzo minore rispetto al backend, infatti è sufficiente effettuare la migrazione esclusivamente dei servizi che si occupano dell'invocazione delle API.
Il servizio visualizzato precedentemente in figura 4.16 *EmployeeService*, non cambia la sua struttura, i metodi rimangono gli stessi, tuttavia l'implementazione del metodo deve cambiare. Oltre all'implementazione dei metodi cambia anche l'oggetto che permette il fetching dei dati dal server, infatti trattandosi ora di un GraphQL Server, sarà necessaria la dipendenza dall'oggetto *Apollo* il quale permette il fetching dei dati da un server GraphQL.
In figura 4.26 è possibile visualizzare l'implementazione del metodo *allEmployees* che si differenzia dal medesimo metodo ma per l'invocazione di REST API.

```
allEmployees(){
  return this.apollo.watchQuery<any>({
    query: gql`
    {
      allEmployees {
        id
        name
        surname
        birth
        salary
      }
    }`,
    }).valueChanges;
  }
```

Figura 4.26: Metodo *allEmployees()* della classe *EmployeeService*.

Attraverso il metodo *watchQuery<>* dell'oggetto *apollo* è possibile eseguire la query all'endopoint del server GraphQL. Il metodo ritorna un oggetto di tipo *Observable* ed il subscriber dovrà occuparsi di gestire quanto ritornato dal servizio, ovvero: nel caso in cui la query sia andata a buon fine, dovrà occuparsi di estrapolare i dati dal campo *"data"* del file JSON ricevuto in risposta, altrimenti dovrà occuparsi di mostrare il messaggio ed il tipo dell'errore presente nel campo *"errors"*.

4.2 SushiLab

4.2.1 Comprensione dell'applicativo

Breve panoramica su SushiLab, ambito d'uso, funzionalità, ecc...

Panoramica del backend

Panoramica su architettura del backend (sviluppato in Spring), entità e relazioni, business logic, strato di persistenza, test, **API** (Parte preponderante della panoramica sul backend).

Panoramica del frontend

Panoramica su architettura del frontend (sviluppato in Angular), principali components, **strato di servizio** (parte preponderante perché gestisce le chiamate alle API del backend).

4.2.2 Migrazione del BE da REST a GraphQL

Molto simile a quanto scritto per il prototipo nella parte di migrazione adattato alle API specifiche di SushiLab.

4.2.3 Migrazione del FE da REST a GraphQL

Capitolo 5

Analisi comparativa dei protocolli REST e GraphQL

In questo capitolo verrà svolta l'analisi comparativa tra i due protocolli REST e GraphQL, sia dal punto di vista teorico che da quello pratico .

5.1 Introduzione

REST è stato ed è tutt'oggi lo standard più seguito per la realizzazione delle Web API, tuttavia dopo l'uscita di GraphQL gli sviluppatori hanno iniziato ad utilizzare sempre di più la nuova tecnologia. Infatti GraphQL ha portato con se delle interessanti soluzioni per molti dei problemi e dei vincoli dello stile architetturale REST.
L'innovazione portata da GraphQL è stata apprezzata in larga scala tra gli sviluppatori, a conferma di ciò è possibile visualizzare nell'immagine 5.1 l'aumento nell'utilizzo di questa tecnologia con il passare degli anni.

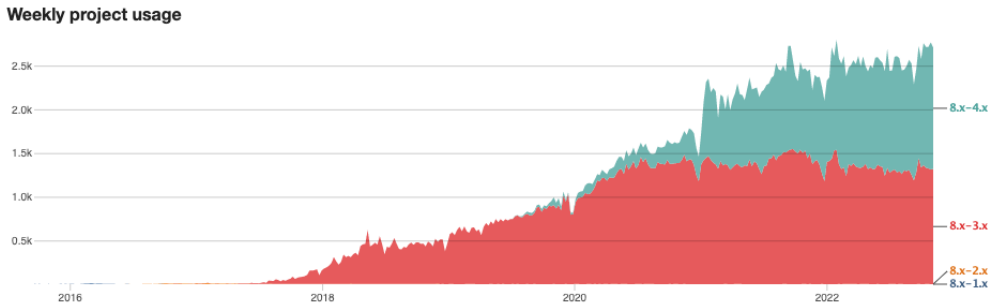

Figura 5.1: Grafico sull'utilizzo di GraphQL negli anni.

Nel seguente capitolo verranno analizzati nel dettaglio e paragonati i due protocolli, sotto tutti i punti di vista, mettendo in risalto vantaggi e svantaggi di ciascuno; infine verranno riportate le deduzioni elaborate durante lo stage sul protocollo che è meglio seguire in base all'applicativo che si vuole sviluppare.

5.2 Analisi comparativa

Come sottolineato in precedenza, GraphQL e REST hanno diversi aspetti che li differenziano. La più grande differenza tra questi due protocolli è legata alla loro natura: infatti quando si fa riferimento a REST, si sta parlando di uno stile architetturale, dunque di un modo di costruire le proprie API le quali, se rispettano i vincoli REST illustrati al punto 3.2, vengono definite RESTful. Quando si fa riferimento a GraphQL invece, si sta parlando di un linguaggio di query fortemente tipizzato.

Di seguito è presente un'analisi comparativa dettagliata per ciascun aspetto che differenzia i due protocolli di data fetching.

5.2.1 Endpoints

La prima grossa differenza tra i due protocolli riguarda gli endpoint.
Lo stile REST prevede l'utilizzo di più endpoint, sfrutta infatti la molteplicità degli endpoint per differenziare le richieste possibili. Quando un client implementa una richiesta a delle REST API deve sapere esattamente a quale endpoint inviare la richiesta per ricevere i dati necessari. In figura 5.2 viene rappresentato la struttura degli endpoint multipli di un REST server con il client che invia diverse richieste ai diversi endpoint.

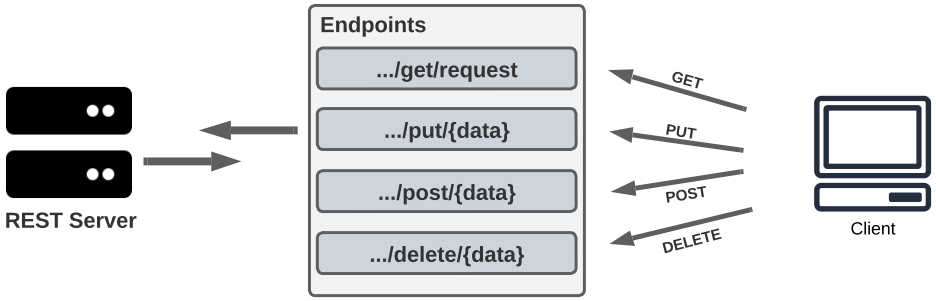

Figura 5.2: Gli endpoints multipli in REST.

Con GraphQL questo non avviene, infatti GraphQL prevede l'esposizione di un unico endpoint. A questo singolo endpoint possono essere inviate tutte le richieste inserendo nel body della richiesta la query, la mutation o la subscription. In figura 5.3 è possibile visualizzare come il GraphQL server fornisca un unico endpoint e come il client invii tutti i tipi di richieste allo stesso medesimo endpoint.

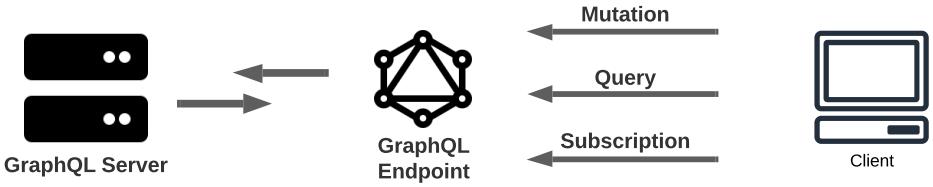

Figura 5.3: Il singolo endpoint GraphQL.

A proposito di ciò viene riportata di seguito una citazione di Lee Byron, il co-creatore di GraphQL:

> *"Think in graphs, not endpoints."*

5.2.2 Overfetching e Underfetching

La questione dell'overfetching e underfetching è uno degli aspetti che viene maggiormente considerato nella decisione architetturale riguardo quale protocollo di data fetching utilizzare tra REST e GraphQL.
Lo stile architetturale REST non prevede di definire lato client esattamente quali dati ricevere. Un client che necessita un certo insieme da un server con REST API, è costretto ad eseguire una o più richieste e dunque a prendersi in carico della rielaborazione dei dati. Per uno sviluppatore backend è molto complesso riuscire a creare delle REST API che siano in grado di soddisfare esattamente le tutte le richieste dei client. L'introduzione di GraphQL come nuovo protocollo di data fetching ha posto una soluzione a questo problema, permettendo al client di specificare esattamente la forma e il quantitativo di dati necessari. Quando si decide di implementare un applicativo e si valuta quale protocollo di data fetching utilizzare, questo è sicuramente un punto da considerare.

Overfetching Quando si parla di overfetching si fa riferimento al fatto che vengano forniti più dati di quanti realmente necessari. Riprendendo come esempio il prototipo visto nel capitolo **??**, si suppone che il client necessiti della lista degli impiegati e che, per ciascun impiegato, necessiti esclusivamente l'id e il nome. Qualora si tratti della version REST del server, il client inviare la richiesta HTTP all'endpoint mappato sul metodo *allEmployee()*, il quale ritorna una lista di impiegati e, per ciascun impiegato, tutti i campi. Di seguito il JSON di risposta che il client riceve in seguito alla richiesta nel caso in cui fossero presenti solo due impiegati:

```
[
  {
      "id": "3AFASDF12F",
      "name": "Matteo",
      "surname": "Verdi",
      "salary": 1500,
      "birth": "1995-02-21"
  },
  {
      "id": "GA14PL3FAV",
      "name": "Marco",
      "surname": "Blu",
      "salary": 1500,
      "birth": "1993-12-20"
  }
]
```

Si può subito notare come i campi *surname*, *salary* e *birth* non siano necessari in quanto il client utilizza solo i campi *id* e *name*. Nel caso del prototipo si tratta di un problema irrisorio data la ridotta grandezza dei dati, tuttavia in applicativi a larga

scala che richiedono grossi quantitativi di dati complessi può risultare un problema in termini di occupazione di rete e di rallentamenti dell'applicativo. Questo problema può essere risolto in un server con REST API fornendo endpoint specifici per ciascun tipo di richiesta, tuttavia questa soluzione rischia di introdurre confusione e nel tempo la manutenzione potrebbe risultare sempre più complessa.
GraphQL risolve questo problema attribuendo al client la responsabilità di definire quali siano i campi di cui necessita. Questo è possibile specificando nella query i campi richiesti, di seguito l'esempio dell'invocazione contenente a sinistra la query *allEmployees*, mentre a destra il JSON ricevuto in risposta:

```
QUERY

query {
  allEmployees {
      id
      name
      }
    }
  }
```

```
JSON RETURNED

"data": {
   "allEmployees": [
     {
        "id": "3AFASDF12F",
        "name": "Matteo"
     },
     {
        "id": "GA14PL3FAV",
        "name": "Marco"
     }
   ]
 }
```

In GraphQL questo approcio permette di non sovraccaricare inutilmente la rete e di mantenere ordinato e di facile manutenzione la struttura API del server. Lato client è richiesto un maggior sforzo nella specifica della query, ma d'altra parte si evitano problemi di rallentamento o errori dovuti al fetching di grossi quantitativi di dati inutili.

Underfetching L'underfetching è il problema opposto all'overfetching, ovvero ciò accade quando il client dopo una richiesta alle REST API riceve solo una parte dei dati necessari. Questo implica che il client deve eseguire più chiamate per ottenere i dati completi.
Riprendendo l'esempio precedente nel paragrafo riguardante l'overfetching, qualora il client desiderasse visualizzare i progetti ai quali sta lavorando un impiegato, dovrà prima ricevere la lista degli impiegati e, solo successivamente, ricercare i progetti con l'id dell'impiegato.
La stessa medesima operazione utilizzando GraphQL è risolvibile in una sola richiesta:

```
QUERY

query {
  allEmployees {
    id
    projects {
        id
        name
      }
    }
```

```
JSON RETURNED

"data": {
   "allEmployees": [
     {
       "id": "3AFASDF12F",
       "projects": [
         {
            "id": "7NFAISH280",
            "name": "Progetto Beta"
```

```
    }
  }
```

```
        },
        {
            "id": "N2A8F234SD",
            "name": "Progetto Teta"
        }
      ]
    }
  ]
}
```

A sinistra è possibile visualizzare la query *allEmployees* nella quale viene specificato l'id di ciascun impiegato ed il campo *projects*, il quale, a sua volta, ha specificato i campi *id* e *name*. A destra invece è visualizzato il JSON ritornato con l'impiegato e i due progetti ai quali sta partecipando.

Utilizzo del protocollo HTTP

Nonostante entrambi i protocolli non richiedano per forza il protocollo HTTP per funzionare, entrambi nella maggior parte dei casi vengono utilizzati con esso. Per questo motivo viene eseguita un'analisi su come REST e GraphQL si comportano con il protocollo HTTP.
I due protocolli comparati utilizzano il protocollo HTTP in maniera molto differente. Lo stile REST alla sua creazione è stato fortemente basato sul protocollo HTTP, per questo motivo lo sfrutta ampiamente. GraphQL invece, per il modo in cui opera, sfrutta solo in parte ed in maniera "stupida" il protocollo HTTP e per questo motivo GraphQL viene spesso definito agnostico rispetto al protocollo di trasporto.

Metodi HTTP Per metodi HTTP s'intendono le possibili operazioni che il protocollo HTTP prevede nella comunicazione tra due moduli di rete. Le API REST supportano i metodi POST, GET, PUT e DELETE per la gestione delle risorse sul server e nel caso della POST e della PUT è possibile specificare i dati all'interno del body della richiesta HTTP. GraphQL invece sfrutta solamente l'operazione POST e specifica nel body la query, la mutation o la subscription desiderata.

Codici di stato I codici di stato vengono utilizzati per dare informazioni sull'esito di una richiesta HTTP e risultano fondamentali nella comprensione degli errori. Mentre REST utilizza ampiamente i codici di stato nelle risposte, GraphQL ritorna esclusivamente il codice di stato 200 e specifica il problema nel JSON ritornato; talvolta in alcuni GraphQL server, qualora si dovessero verificare degli errori, può essere ritornato anche il codice di stato 500 riferito all'*Internal Server Error*.

Caching Per caching s'intende il meccanismo attraverso il quale il browser, il client, i server proxy o altri moduli della rete riescono ad archiviare localmente i dati a cui si accede frequentemente senza dover ogni volta mandare la medesima richiesta al server. Si tratta di fattore importante poiché, se utilizzato correttamente, permette di ridurre il traffico dati tra i moduli della rete e i tempi di latenza, i dati sono recuperabili molto più velocemente e questo è fondamentale per fornire velocemente i dati nei siti web, vengono eseguiti meno accessi al database lato server e le performance di conseguenza migliorano.
Il caching è parte integrante del protocollo HTTP e viene ampiamente sfruttato in

REST, infatti per il metodo GET e in parte anche per i metodi PUT e DELETE il caching, a meno di direttive specifiche, viene utilizzato di default. Per quanto riguarda il metodo POST il caching non viene utilizzato di default, ma con apposite direttive nell'header della risposta HTTP è possibile permetterlo.
Per il modo in cui opera GraphQL, il caching previsto dal protocollo HTTP non viene sfruttato. Per questo motivo, anche nella documentazione GraphQL, viene specificato come sia un dovere del client quello di abilitare e gestire il caching. Alcune librerie permettono di risolvere questo problema, ad esempio il modulo *Apollo* utilizzato per la realizzazione del frontend del prototipo e nella migrazione dell'applicativo SushiLab, include una implementazione del caching di default chiamata *InMemoryCache*.

Altri aspetti comparativi

Sono presenti molti altri aspetti che differenziano i due protocolli analizzati. Di seguito vengono riporatati i più importanti.

Documentazione La documentazione delle API risulta fondamentale quando uno sviluppatore client necessita di utilizzare i servizi web resi disponibili da un server.
Uno degli aspetti più interessanti di GraphQL è proprio quello che si autodocumenta durante la definizione del GraphQL Schema. Infatti definendo il GraphQL schema vengono dichiarati i tipi e la loro struttura campo per campo, le loro relazioni, le query, le mutation, le subscription ed eventuali altri tipi particolari come tipi unione ed enumerazioni. Inoltre il linguaggio SLD, utilizzato nella realizzazione dello schema GraphQL, supporta anche il linguaggio di markup *Markdown* e dunque è possibile, direttamente dal GraphQL schema, specificare ulteriori informazioni su determinati elementi dello schema inserendo delle frasi.
Lo sviluppatore client che desidera la documentazione su un determinato schema GraphQL, può facilmente recuperarla con un processo chiamato Introspection per il quale è possibile effettuare una query sempre disponibile richiedendo il campo $\_\_schema$ e definendo ciò che vuole visualizzare dei tipi nello schema, così facendo può ottenere come risposta direttamente tutte le informazioni che più desidera. Ci sono inoltre dei tool che permettono di semplificare il processo di instrospezione, ad esempio usando il GraphQL playground.
In REST non è previsto nessun modo per documentare le API se non affidandosi a servizi esterni, come ad esempio il noto toolset opensource *Swagger*.

Sicurezza Si tratta di un fattore fondamentale per la trasmissione sicura di dati tra moduli sulla rete. Le REST API supportano i protocolli crittografici, ad esempio il Transfer Layer Security il quale assicura che i dati che vengono passati tra moduli della rete rimangano invariati e privati. Inoltre sono presenti moltpelici specifiche per garantire la sicurezza nello scambio di messaggi attraverso API REST, tra le quali: JWT, JWS, JWk, ecc...
Anche GraphQL ha dei modi per l'autenticazione e l'autorizzazione le richieste del client, tuttavia risultano sicuramente meno sviluppati e consolidate rispetto a quelli disponibili con le REST API.
Un altro punto critico di GraphQL legato alla sua flessibilità, ovvero al fatto che permetta di richiedere qualsiasi tipo di dato in qualsiasi forma, è quello che proprio per questa ragione sia possibile, se non vengono strutturate bene le risorse, essere vittime di attacchi DoS attraverso query annidate che vanno a sovraccaricare il database e il server. Questa parziale mancanza di GraphQL è dovuta probabilmente al fatto che

si tratti di una tecnologia giovane, ma sono presenti sempre più modi per gestire la sicurezza ed evitare questo tipo di situazioni.

Formato dati supportati Le REST API supportano diversi tipi di dati come: JSON, XML e YAML. GraphQL dall'altra parte supporta solo il formato JSON.

Versionamento ed evoluzione delle API Il versionamento e l'evoluzione delle API sono argomenti molto dibattuti, ci sono diversi filoni di pensiero e diversi approci classici, alcuni prediligono l'evoluzione delle API, altri ritengono necessario versionarle. GraphQL predilige un approcio di evoluzione delle API, infatti è possibile evolvere le proprie GraphQL API introducendo nuovi campi nei tipi e preservando i vecchi campi così da mantenere la retrocompatibilità. A conferma di quanto appena affermato, di seguito viene riportato quanto scritto nella pagina riguardante le migliori pratiche nella documentazione GraphQL:

> *"While there's nothing that prevents a GraphQL service from being versioned just like any other REST API, GraphQL takes a strong opinion on avoiding versioning by providing the tools for the continuous evolution of a GraphQL schema."*

Questo approcio inoltre può essere utilizzato anche con API REST.
Per quanto riguarda però l'eliminazione di alcune funzionalità o campi dati, in GraphQL è possibile definire dei campi deprecati con apposite direttive e, quando gli sviluppatori frontend interrogano lo schema con l'introspezione, vengono scoraggiati dall'utilizzare quella determinata funzionalità o campo dati che verrà presto rimossa. A differenza degli schemi di versionamento come il *Semantic Versioning*, in GraphQL non è possibile specificare quando verrà rimosso effettivamente il campo deprecato. Anche con le REST API può essere indicato un campo deprecato, ma questo è possibile specificarlo solo nella documentazione delle API.
Dunque in accordo con quanto riportato precedentemente dalla documentazione GraphQL, nulla impedisce il versionamento in GraphQL, per questo sono stati ideati dei modi per versionare anche le GraphQL API, come ad esempio includere nei vari tipi dei campi versionamento e dunque in base alla versione trattare i vari campi in maniere differenti. In REST invece il versionamento avviene in maniera differente, infatti vengono utilizzati principalmente due approci, uno che prevede di realizzare più versioni di API e richiede però che venga specificato la versione nell'URI della chiamata; l'altro che invece richiede di specificare la versione includendola nell'header della richiesta HTTP.

Trasmissione di dati in tempo reale Nelle Real Time Application è necessario ricevere i dati dal server in tempo reale. Questo non è possibile se il server fornisce delle REST API, a meno che non vengano utilizzati degli escamotage come ad esempio il *long pooling* secondo il quale il server, dopo aver ricevuto una richiesta dal client, mantiene aperta la connessione fino a che non arrivano dati in tempo reale e quel punto invia la risposta al client, il quale subito dopo invia una nuova richiesta al server per riaprire immediatamente la connessione.
In GraphQL non serve utilizzare degli escamotage per permettere questo tipo di scambio dati in tempo reale, infatti è stato reso disponibile un tipo detto *Subscription*, spiegato precedentemente nel capitolo **??**, che permette al client di "iscriversi" ad un certo tipo di dati e, al verificarsi di un certo evento, il server invia direttamente i dati al client senza che il client li richieda. Il tutto è implementato con i WebSocket che permettono

di mantenere la connessione server-client attraverso una connessione TCP.
È possibile utilizzare in WebSocket anche con API REST, ma risulta essere comunque un adattamento guidato da esigenze di poter utilizzare connessioni bidirezionali, non come nel caso di GraphQL in cui si tratta proprio di una funzionalità prevista dal protocollo stesso.

File uploading Spesso può essere necessario permettere al client di effettuare l'upload di file di vario genere sul server. In REST questa necessità viene soddisfatta pienamente, infatti è possibile passare il file inserendolo nel body della richiesta e impostanto alcune direttive nell'header. In GraphQL ciò non è previsto del protocollo, sono però presenti diverse soluzioni già implementate in molte librerie utilizzate.

Aspetti comparativi pratici

Durante la realiz

Analisi comparativa prestazionale

5.3 Conclusioni

Capitolo 6

Tecnologie utilizzate

Capitolo 7

Conclusioni

7.1 Consuntivo finale

7.2 Raggiungimento degli obiettivi

7.3 Conoscenze acquisite

7.4 Valutazione personale

Appendice A

Appendice A

Citazione

Autore della citazione

Bibliografia

Riferimenti bibliografici

James P. Womack, Daniel T. Jones. *Lean Thinking, Second Editon*. Simon & Schuster, Inc., 2010 (cit. a p. 1).

Siti web consultati

Manifesto Agile. URL: http://agilemanifesto.org/iso/it/ (cit. a p. 1).