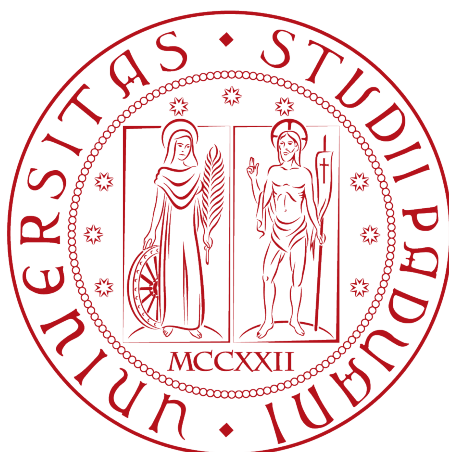


Università degli Studi di Padova

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



De omnibus dubitandum.

— Cartesio



# Sommario

Le moderne Web Application adottano un disaccoppiamento stretto tra client e backend.

Soluzioni architetturali come REST garantiscono di poter realizzare API di utilizzo generale fruibili da svariati client, dai browser sino alle più moderne applicazioni mobili.

Pur essendo REST lo standard de-facto per la scrittura di Web API, esso presenta alcune debolezze che nuovi strumenti sorti negli ultimi anni cercano di superare.

GraphQL è certamente una delle più recenti e popolari tecnologie che il mercato dell'Information Technology ci mette a disposizione per la realizzazione di Web API.

Obbiettivo di questa tesi è quello di realizzare una comparazione tra le tecnologie di data fetching REST e GraphQL individuando i vantaggi nell'adozione di una tecnologia rispetto all'altra; le caratteristiche dei due approcci emergono chiaramente durante lo sviluppo e migrazione di una applicazione da un approccio all'altro.



# Ringraziamenti

*Sentiti ringraziamenti al Prof. Paolo Baldan che mi ha seguito durante lo stage e la redazione della tesi.*

*Ringrazio anche il tutor aziendale Fabio Pallaro per la disponibilità che mi ha dimostrato durante tutto lo svolgimento dello stage.*

*Padova, Dicembre 2022*

Federico Marchi





# Indice

<b>1</b>	<b>Introduzione</b>	<b>1</b>
1.1	L'azienda . . . . .	1
1.2	L'idea . . . . .	1
1.3	Organizzazione del testo . . . . .	1
<b>2</b>	<b>Descrizione dello stage</b>	<b>3</b>
2.1	Introduzione al progetto . . . . .	3
2.2	Analisi preventiva dei rischi . . . . .	3
2.3	Requisiti e obiettivi . . . . .	3
2.4	Pianificazione . . . . .	3
<b>3</b>	<b>Protocolli di modellazione e trasferimento dati</b>	<b>5</b>
3.1	Introduzione ai protocolli . . . . .	5
3.1.1	Modello architetturale client-server . . . . .	5
3.1.2	Application Programming Interface . . . . .	5
3.1.3	Protocolli di trasferimento dati . . . . .	6
3.1.4	I primi protocolli . . . . .	6
3.2	Approfondimento sullo stile architetturale REST . . . . .	7
3.2.1	Principi dell'architettura REST . . . . .	7
3.3	Approfondimento sul linguaggio di query GraphQL . . . . .	8
3.3.1	Introduzione . . . . .	8
3.3.2	GraphQL schema . . . . .	9
3.3.3	Definizione dei tipi . . . . .	10
<b>4</b>	<b>Introduzione ai casi d'uso per l'analisi comparativa</b>	<b>15</b>
4.1	Prototipo . . . . .	15
4.1.1	Introduzione . . . . .	15
4.1.2	Architettura ed entità . . . . .	15
4.1.3	Progettazione e realizzazione backend . . . . .	17
4.1.4	Progettazione e realizzazione frontend . . . . .	25
4.1.5	Migrazione da REST a GraphQL . . . . .	31
4.2	SushiLab . . . . .	36
4.2.1	Confronto con stakeholder . . . . .	36
4.2.2	Comprensione dell'applicativo e obiettivi . . . . .	36
4.2.3	Panoramica del backend . . . . .	37
4.2.4	Migrazione del BE da REST a GraphQL . . . . .	39
4.2.5	Migrazione del FE da REST a GraphQL . . . . .	39
<b>5</b>	<b>Analisi comparativa dei protocolli REST e GraphQL</b>	<b>43</b>

5.1	Introduzione . . . . .	43
5.2	Analisi comparativa . . . . .	44
5.2.1	Endpoints . . . . .	44
5.2.2	Overfetching e Underfetching . . . . .	45
5.3	Conclusioni . . . . .	52
5.3.1	Linee guida prima di intraprendere uno sviluppo o la migrazione di un applicativo da/in REST VS GraphQL . . . . .	52
5.3.2	Tabella di comparazione riassuntiva . . . . .	55
<b>6</b>	<b>Tecnologie utilizzate</b>	<b>57</b>
<b>7</b>	<b>Conclusioni</b>	<b>59</b>
7.1	Consuntivo finale . . . . .	59
7.2	Raggiungimento degli obiettivi . . . . .	59
7.3	Conoscenze acquisite . . . . .	59
7.4	Valutazione personale . . . . .	59
<b>A</b>	<b>Appendice A</b>	<b>61</b>
	<b>Bibliografia</b>	<b>65</b>

# Elenco delle figure

3.1	Connessione one-to-one. . . . .	11
3.2	Connessione one-to-many. . . . .	12
3.3	Connessione many-to-many. . . . .	12
4.1	Architettura del prototipo di Web Application. . . . .	16
4.2	Schema ER delle entità del prototipo. . . . .	17
4.3	Architettura interna backend. . . . .	17
4.4	Esempio di implementazione dell'entità Employee in Spring Boot. . . . .	18
4.5	Esempio di implementazione della repository di Employee in Spring Boot. . . . .	20
4.6	Esempio di implementazione dell'interfaccia EmployeeService. . . . .	21
4.7	Classe EmployeeServiceImpl. . . . .	21
4.8	Classe EmployeeController. . . . .	22
4.9	Classe <i>SmokeTest</i> sulla creazione dei controller. . . . .	24
4.10	Classe <i>EmployeeControllerTest</i> . . . . .	24
4.11	Metodo di test <i>createEmployee</i> della classe <i>EmployeeControllerTest</i> . . . . .	25
4.12	Architettura della Web Application del prototipo. . . . .	26
4.13	Interfaccia di Employee. . . . .	27
4.14	Prima pagina della Web Application. . . . .	27
4.15	Opzioni disponibili con operazione GET selezionata. . . . .	28
4.16	Classe <i>EmployeeService</i> . . . . .	28
4.17	Opzioni disponibili con operazione INSERT/UPDATE selezionata. . . . .	29
4.18	Opzioni disponibili con operazione DELETE selezionata. . . . .	29
4.19	Test per i metodi del servizio <i>EmployeeService</i> . . . . .	30
4.20	Test sul metodo <i>allEmployee()</i> di <i>EmployeeService</i> . . . . .	30
4.21	Tipi <i>Employee</i> e <i>EmployeeInput</i> nel GraphQL Schema. . . . .	31
4.22	Implementazione delle query, mutation e subscription nel GraphQL Schema. . . . .	32
4.23	Classe <i>EmployeeController</i> . . . . .	33
4.24	Subscription <i>newEmployeeAdded</i> . . . . .	34
4.25	Classe <i>GraphQLExceptionHandler</i> con metodo <i>resolveToSingleError</i> ridefinito per la risoluzione dell'eccezione <i>EmployeeNotFoundError</i> . . . . .	35
4.26	Metodo <i>allEmployees()</i> della classe <i>EmployeeService</i> . . . . .	36
4.27	Diagramma ER di <i>SushiLab</i> . . . . .	37
5.1	Grafico sull'utilizzo di GraphQL negli anni. . . . .	43
5.2	Gli endpoints multipli in REST. . . . .	44
5.3	Il singolo endpoint GraphQL. . . . .	45
5.4	Soluzione ibrida mantenendo sia REST API che GraphQL API. . . . .	54

## Elenco delle tabelle

4.1	Metodi del controller <i>LoginController</i> . . . . .	39
4.2	Metodi del controller <i>UtenteController</i> . . . . .	40

# 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<sup>1</sup>

### 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 ...

---

<sup>1</sup>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*<sup>[g]</sup>;
- \* 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

*Il capitolo tratta i protocolli di modellazione e trasferimento dati dal punto di vista teorico, in particolare viene approfondito lo stile architetturale REST e il linguaggio di query GraphQL.*

### 3.1 Introduzione ai protocolli

#### 3.1.1 Modello architetturale client-server

Prima di procedere nella spiegazione dei protocolli di trasferimento e modellazione dati, è necessario fare una breve introduzione sull'architettura delle Web Application moderne. Quest'ultime seguono il 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 server - client; di seguito viene introdotto il concetto di Application Programming Interface necessario per comprendere il funzionamento dei protocolli di data fetching.

#### 3.1.2 Application Programming Interface

Le API sono interfacce comunemente realizzate per permettere la comunicazione tra server e client. Ciascun applicativo/dispositivo è sviluppato con strutture di dati differenti che evolvono nel tempo, risulta perciò complessa la comunicazione tra di essi. Le API giocano un ruolo fondamentale nello scambio di dati poiché definiscono una interfaccia per la comunicazione; quest'ultima è indipendente dall'implementazione del dispositivo o dell'applicativo e permette di comunicare secondo rispettando regole ben documentate. Risultano dunque fondamentali nella comunicazione, collaborazione e integrazione di nuovi componenti applicativi.

Al giorno d'oggi le API sono utilizzate dalla maggior parte delle Web Application, dispositivi IoT e applicativi di vario genere.

Il documento fa riferimento alle Web API, cioè a quelle interfacce che sfruttano principalmente il protocollo HTTP per la comunicazione server - client. Tra i vari tipi di Web API troviamo:

- \* **API pubbliche:** API accessibili da tutti;
- \* **API private:** API create con lo scopo di essere utilizzate solo ed esclusivamente all'interno dell'azienda;
- \* **API partner:** API utilizzate tra aziende in collaborazione;
- \* **API composte:** 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 di data fetching.

### 3.1.3 Protocolli di trasferimento dati

Per protocolli di modellazione e trasferimento dati s'intende un insieme di regole, strutture e vincoli che regolano il corretto funzionamento delle API. Permettono dunque di definire una sorta di standard al quale gli sviluppatori possono far riferimento per implementare e interagire con esse. 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.

### 3.1.4 I primi protocolli

Di seguito i più importanti protocolli in uso prima dell'avvento di REST e GraphQL. Si tratta di tecnologie aventi tutte lo stesso scopo, ma di natura completamente differente.

#### Remote Procedure Call

Il protocollo RPC prevede l'invocazione di una procedura o subroutine del server da parte del client. Il server esegue dunque la logica di business corrispondente senza che il client ne conosca i dettagli. Dal punto di vista del client l'esecuzione avviene localmente, ma in realtà è il server remoto a garantirla.

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

SOAP è un vero e proprio protocollo che definisce la struttura dei dati che devono essere trasferiti ed il modo in cui 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 tre principali componenti:

- \* **Envelope:** necessario al fine di identificare il documento come messaggio SOAP;
- \* **Header:** opzionale. Il suo scopo è quello di fornire informazioni che vengono interpretate dai moduli di rete durante il trasporto del messaggio;
- \* **Body:** il body contiene il vero e proprio messaggio che si vuole trasferire.

## 3.2 Approfondimento sullo stile architetturale REST

REST è uno stile architetturale introdotto da Roy Fielding nel 2000 e viene considerato al giorno d'oggi come uno standard per la realizzazione di Web API. Questo stile architetturale prevede l'astrazione degli elementi che compongono il sistema da modellare; di questi elementi REST ignora i dettagli implementativi ed espone esclusivamente gli aspetti necessari per l'interazione con essi; introduce inoltre convenzioni da rispettare nella comunicazione server - client.

### 3.2.1 Principi dell'architettura REST

Una interfaccia API è definibile RESTful se rispetta i sei principi indicati da Roy Fielding.

#### Server - client

Il primo principio sposa uno dei paradigmi cardine dell'informatica: il principio di *Separation of concerns* secondo il quale conviene sempre separare un sistema complesso in moduli distinti, ognuno avente un ruolo ben definito.

Ciò viene ripreso nell'architettura REST separando il client dal server; così facendo server e client possono essere implementati in maniera indipendente usando qualsiasi lingua o tecnologia, è sufficiente che siano conformi al prossimo principio detto *Uniform Interface*.

#### Uniform Interface

Si tratta di un principio fondamentale che caratterizza lo stile architetturale REST. Secondo questo principio l'interazione tra componenti Web come client, server e gli intermediari del network, dipendono dalla uniformità delle loro interfacce.

I componenti Web dunque sono in grado di comunicare coerentemente seguendo i seguenti vincoli di interfaccia delineati da Fielding:

- \* **Identification of resources:** le risorse coinvolte devono essere identificate nella richiesta stessa, dunque specificandole nell'url;
- \* **Manipulation of resources through representations:** il client non si deve curare di come sono strutturate le risorse all'interno del server, ma deve conoscere la loro rappresentazioni in termini di dati ed utilizzare correttamente i metodi HTTP per manipolarle;
- \* **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. È dunque

necessario includere i link per ciascuna risposta, così che il client possa navigare tra le risorse correlate.

### Layered System

Secondo questo principio l'architettura di un applicativo deve essere composta da più strati. Ciascuno strato deve ingorare gli altri strati tranne quelli ad esso adiacenti. Questi layer possono essere composti da intermediari basati sul network i quali intercettano la comunicazione server - client 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 architettura Web, secondo il quale un web server deve etichettare i dati di ogni risposta come adatti o meno alla permanenza in cache. Così facendo gli intermediari tra server e client e il client stesso sanno come comportarsi riguardo alla memorizzazione in cache dei dati ritornati.

### 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 tale comunicazione viene definita priva di stato.

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 il server può inviare al client della logica da eseguire localmente.

## 3.3 Approfondimento sul linguaggio di query GraphQL

### 3.3.1 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 alcune debolezze.

Più nello specifico GraphQL è un linguaggio di query. 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 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. Uno 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 design" da tenere in considerazione durante lo sviluppo:

- \* **Hierarchical:** i tipi ricercati in una query GraphQL seguono una struttura gerarchica; 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 del client, per questo bisogna realizzarle in maniera flessibile cercando di tener conto delle possibili richieste client;
- \* **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 maggior parte 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 client possono consultare a fondo il GraphQL Schema e possono dunque vedere tutte le query disponibili, i vari tipi e i loro campi.

### 3.3.2 GraphQL schema

GraphQL ha cambiato il modo di pensare alle API: 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 e relazioni distribuiti su grafo. Questa collezione di tipi e loro relazioni deve essere riportata in maniera chiara in una pagina chiamata **GraphQL Schema**. In questo schema devono essere definiti tutti quei tipi che si vuole esporre alle richieste del client per esser modificati, aggiunti, eliminati o semplicemente visualizzati. Il GraphQL schema viene realizzato con un linguaggio denominato SDL (Schema Definition Language) il quale permette di specificare le operazioni disponibili, i tipi e le relazioni tra di essi.

Durante la progettazione e la realizzazione di un server di backend possono essere seguite due differenti tecniche per implementare schema e codice:

- \* **Schema first:** questa tecnica si discosta dal classico ordine di realizzazione di un server di backend infatti, come deducibile dal nome, prevede la realizzazione del GraphQL Schema prima di realizzare i resolver delle API;

- \* **Code first:** questa tecnica è opposta alla precedente in quanto prevede la realizzazione prima dei resolver e solo in seguito dello schema.

Non è chiaro quale delle due tecniche sia migliore, sono aperti ancora diversi dibattiti a riguardo, in questo progetto è stato seguito il modello Schema first durante la migrazione.

Grazie alla presenza del GraphQL Schema gli sviluppatori frontend sono in grado di conoscere nel dettaglio la struttura di ciascun tipo e delle varie operazioni disponibili. Le API sviluppate con GraphQL si autodocumentano: è sufficiente la consultazione dello schema per comprendere la natura delle entità che si desidera interrogare o manipolare.

### 3.3.3 Definizione dei tipi

Come già evidenziato GraphQL è un linguaggio di query fortemente tipizzato. I tipi sono l'unità principale di un GraphQL Schema.

Per tipo si intende un oggetto costruito campo per campo e dovrà poi corrispondere ad una entità nel backend dell'applicativo. All'interno dello schema compaiono tutti i tipi che andranno a rappresentare la struttura dati dell'applicativo.

Un tipo può contenere come campi dati altri tipi definiti nel medesimo schema. Di seguito l'esempio della dichiarazione del tipo *Employee*:

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

In questo caso il tipo *Employee* è composto dai 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* devono 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 keyword "*scalar*", oppure delle enumerazioni attraverso l'utilizzo della keyword "*enum*". Infine è prevista anche l'unione tra più tipi tramite la keyword "*union*" come segue:

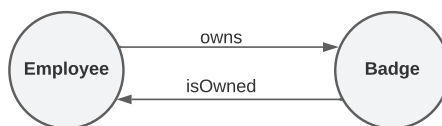
```
union worker = Employee | Manager | Chief
```

In questo caso sono stati uniti in un unico tipo *worker* i tipi *Employee*, *Manager* e infine *Chief*. 3exConnessioni tra tipi GraphQL è così 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 un grafo composto da tipi interconnessioni, da questo deriva il prefisso *Graph*. È fondamentale durante la definizione del GraphQL Schema riportare le relazioni tra tipi in modo consistente.

In GraphQL risulta essere una buona pratica dare bidirezionalità alle connessioni ove possibile, questo con lo scopo di lasciare flessibilità allo sviluppatore client il quale da 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.

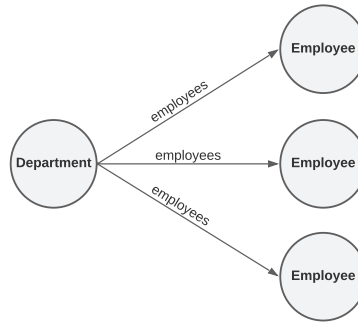
Come mostrato in figura 3.1 il collegamento tra i due tipi è definito come "owns" se si legge nel verso che parte da *Employee* per raggiungere *Badge* e fa riferimento all'omonimo campo di *Employee*. Se altrimenti la connessione si percorre nel verso opposto viene definita "isOwned", come l'omonimo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

Come mostrato in figura 3.2 il collegamento tra il dipartimento e i vari impiegati



**Figura 3.2:** Connessione one-to-many.

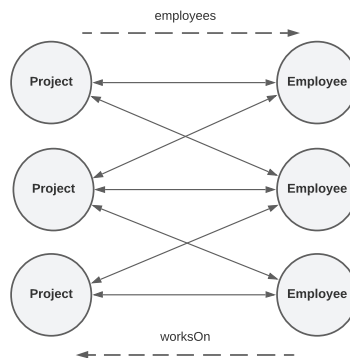
viene chiamato "employees" se si considera il verso che parte da Department per raggiungere Employee e corrisponde all'omonimo campo di Department. Volendo è possibile attribuire doppia direzionalità alla relazione. Di seguito il tipo Department con il campo "employees":

```

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 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



**Figura 3.3:** Connessione many-to-many.

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

Ci sono tre tipi di operazioni che possono essere eseguite sui dati: *Query*, *Mutation* e *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 l'id del dipartimento riceveremo come risposta un JSON contenente un campo "data" che presenterà al suo interno la lista di impiegati. In caso di insuccesso verrà ritornato un JSON con un campo "error" contenente la descrizione dell'errore.

Se il client desiderasse invocare la query definita, dovrebbe invocarla nel seguente modo:

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

In caso di successo verrà ritornata una lista di impiegati per ognuno dei quali compariranno i campi specificati nella query: *id*, *name*, *worksOn*. *worksOn* contiene la lista di progetti per ognuno dei quali verranno indicati esclusivamente i campi: *id*, *name*.

La tecnica *data paging* consente di realizzare meccanismi personalizzati di paginazione specificando la numerosità delle ricorrenze desiderate tramite appositi parametri previsti nella query.

### Mutation

L'operazione di mutation viene utilizzata per eseguire modifiche sui dati. Equivale all'unione delle operazioni POST, DELETE e PUT utilizzate in REST. Come per le query è necessario dichiarare le mutation che si vogliono 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 nell'archivio 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* di cui è stato specificato esclusivamente il nome. La definizione della mutation prevede il ritorno di un oggetto di tipo *Employee*. In questo caso il client dichiara di essere interessato esclusivamente all'id dell'impiegato appena creato.

### Subscription

L'ultimo tipo si chiama Subscription e si tratta di una funzione particolare resa disponibile in GraphQL. Grazie a questa funzione i client possono dichiarare il loro interesse ad una subscription; il client non è costretto ad eseguire polling poiché, al verificarsi della condizione, sarà il server ad avvertire il client.

Un esempio di definizione di una Subscription può essere:

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

Il client che desidera sottoscrivere 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, in questo caso *id* e *name* dell'impiegato.

Essendo il server a dover inviare i dati al client e non viceversa, non sarà utilizzato il protocollo HTTP per la comunicazione server - client bensì il protocollo WebSocket che aprirà un canale di comunicazione a doppia via sopra un socket TCP.

## Capitolo 4

# Introduzione ai casi d'uso per l'analisi comparativa

*Il capitolo descrive in dettaglio il prototipo realizzato e la sua migrazione da REST a GraphQL. A seguire si illustrerà una ulteriore migrazione realizzata sull'applicativo aziendale SushiLab.*

### 4.1 Prototipo

#### 4.1.1 Introduzione

Il prototipo è stato realizzato avendo presente i seguenti obiettivi:

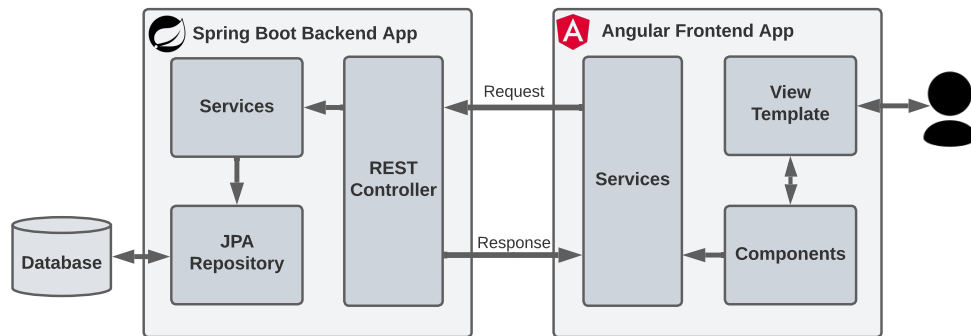
- \* familiarizzare con le tecnologie Spring e Angular per la realizzazione rispettivamente di backend e frontend;
- \* familiarizzare con la realizzazione delle API sia con lo stile architetturale REST che con il linguaggio di query GraphQL;
- \* avere un caso d'uso ulteriore a conferma delle analisi che verranno poi ricavate dalla migrazione dell'applicativo SushiLab;

Si è scelto di realizzare un gestionale server - client in grado di gestire un insieme di impiegati. Ogni impiegato può lavorare in un dipartimento e può dividere il suo tempo su uno o più progetti. Una sede aziendale può contenere uno o più dipartimenti.

#### 4.1.2 Architettura ed entità

##### Architettura generale dell'applicativo

L'applicativo segue 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.

Le richieste effettuate dai servizi del client vengono mappate sul REST controller del server il quale, dopo la rielaborazione interna delle richieste, 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 il client.

### Entità e relazioni

Trattandosi di un gestionale aziendale semplificato si prevedono solo quattro entità principali:

- \* **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:** la sede aziendale, può avere più dipartimenti;
- \* **Department:** un dipartimento che appartiene ad una sede.

Ciascuna entità è caratterizzata dai campi presenti in figura 4.2. I campi in grassetto rappresentano la chiave primaria, quelli sottolineati le chiavi esterne. 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; viene scomposta in due relazioni *one-to-many* con la tabella *Employees\_Projects*;
- \* **one to many:** è presente in due situazioni:
  - tra Employee e Department, infatti ciascun impiegato deve lavorare almeno in un 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.

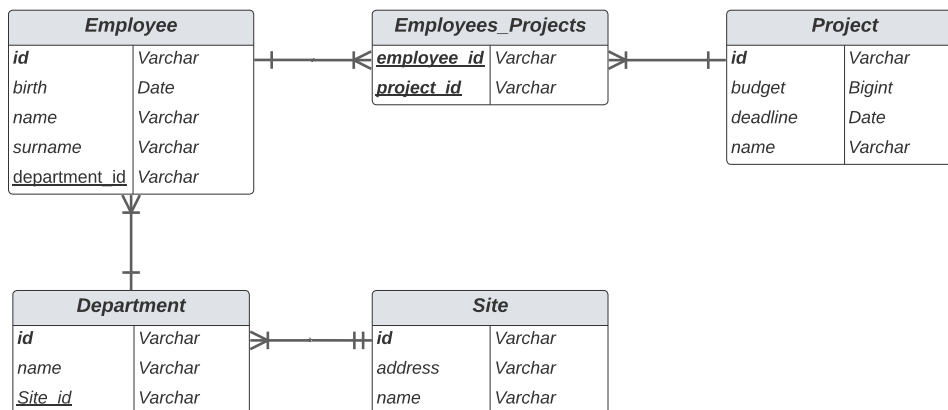


Figura 4.2: Schema ER delle entità del prototipo.

### 4.1.3 Progettazione e realizzazione backend

#### Architettura backend

È stato deciso di seguire il pattern **controller - service - repository** per la realizzazione del server del prototipo. Il pattern rispetta perfettamente il principio di "Separation Of Concerns". Di seguito in figura 4.3 è possibile visualizzare l'architettura del backend.

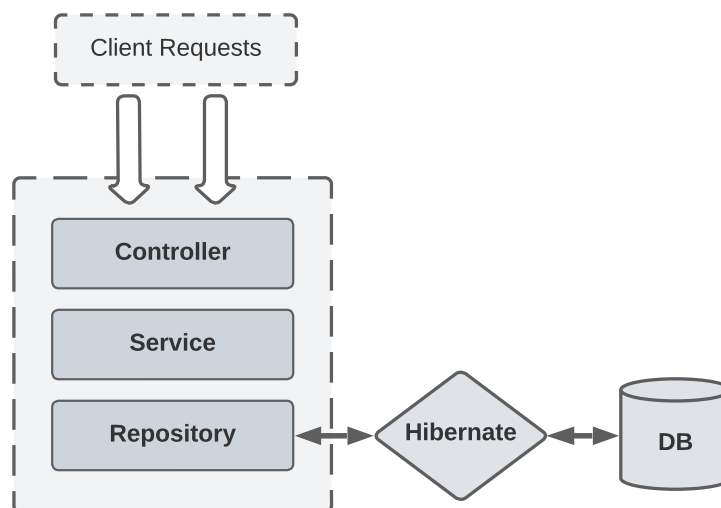


Figura 4.3: Architettura interna backend.

#### PARLARE E INSERIRE QUI DIAGRAMMA DI SEQUENZA

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

- \* **Controller layer:** è l'unico responsabile della comunicazione con i client, 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 nonché della gestione del database.

Durante la realizzazione viene seguito il percorso inverso rispetto a quanto visto nell'immagine 4.3: la realizzazione avviene partendo dallo strato di persistenza, dunque dalla creazione delle entità e del livello di repository.

### Entità e repository

A ciascuna entità del database viene fatta corrispondere una classe in Java quindi sono state realizzate quattro classi rappresentanti le entità Employee, Project, Department e Site.

Ciascuna classe 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, è necessario serializzarli poiché abbandonano la Java Virtual Machine. Viene inoltre utilizzato un *serialVersionUID* per attribuire una versione a ciascuna classe serializzabile in modo da garantire la consistenza delle classi memorizzate in banca dati.

Di seguito 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 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 corrispondenti 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 ciascun campo della classe;
- \* **@NoArgConstructor** e **@AllArgConstructor**: permettono di generare tutte le combinazioni di costruttori con e senza parametri.

Le annotazioni applicate agli attributi della classe sono:

- \* **@Id**: associato al campo *id*, specifica durante il mapping che si tratta della chiave primaria;
- \* **@GeneratedValue**: associato al campo *id*, specifica che quando una nuova istanza dell'entità viene creata, deve essere generato un nuovo id randomico;
- \* **@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 serializzazione;
- \* **@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.

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. 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 estende le seguenti interfacce:

- \* **CrudRepository<T, ID>**: contiene i metodi per gestire le classiche operazioni CRUD;
- \* **PagingAndSortingRepository<T, ID>**: contiene i metodi per gestire la paginazione e l'ordinamento;

Estendendo la `JpaRepository` per ciascun tipo è possibile avere a disposizione diversi metodi per eseguire operazioni già implementate come: `findAll`, `count`, `existById`, `SaveAndFlush`, ecc...

Per realizzare ulteriori metodi è sufficiente rispettare le corretta sintassi del nome del metodo affinché Spring si occupi in automatico della sua implementazione.

Nel caso in cui si voglia realizzare una query personalizzata si utilizza l'annotazione **@Query** alla quale è possibile passare come attributo la query che desideriamo in linguaggio JPQL. In figura 4.5 un esempio di quanto appena detto:

```
@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.

Oltre ai classici metodi di ricerca disponibili 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.

Da evidenziare l'uso dell'annotazione **@Repository** attribuita all'interfaccia, fondamentale al fine di indicare che la classe fornisce meccanismi per modellare i dati del database.

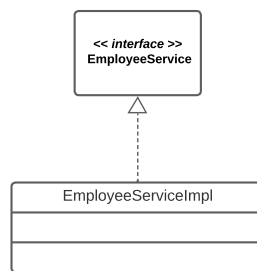
## Service

Il compito dello strato di servizio è facilitare la comunicazione tra controller e repository oltre a realizzare 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, IoC e semplicità nel testing, è stato scelto di implementare il pattern secondo il quale per ogni entità viene realizzata una interfaccia del corrispondente servizio e sua relativa implementazione. In figura 4.6 relativo all'entità Employee.





**Figura 4.6:** Esempio di implementazione dell'interfaccia EmployeeService.

Di seguito (fig. 4.7) viene riportato l'esempio dell'implementazione del servizio 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 per marcare la classe come service provider.

Le annotazioni utilizzate sono:

- \* **@Autowired**: attribuita al campo *employeeRepository* produce la dependency injection della repository *EmployeeRepository* nel servizio;
- \* **@Override**: utilizzata da tutti i metodi, avverte che i metodi dichiarati nell'interfaccia vengono implementati;

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 precedentemente *SelByBornInRage*.

Lo strato di servizio si incarica della gestione delle eccezioni estendendo il gestore *HandlerExceptionResolver*; in questo modo si migliora il comportamento di default che non sempre fornisce informazioni comprensibili sulla natura dell'errore.

## Controller

Lo strato di controller si occupa di gestire le richieste che il server riceve attraverso il protocollo HTTP e di mapparle inoltre sui metodi corrispondenti. 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 = "/", 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. Di seguito le annotazioni associate alla classe:

- \* **@RestController**: definisce la classe come controller di tipo REST;
- \* **@RequestMapping**: indica l'url al quale il client dovrà mandare le richieste per quello specifico controller.

Il controller ha una dipendenza con il servizio *EmployeeService*, necessaria per l'invocazione dei suoi metodi. Le annotazioni associate ai metodi del controller sono:

- \* **@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 richiesta 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 annotazioni **@PathVariable** per 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.

Si rendono disponibili i seguenti metodi:

- \* **listAllEmployee**: ritorna tutti gli impiegati presenti;
- \* **listEmployeeByName**: ritorna tutti gli impiegati con un 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**: consente l'aggiornamento dei campi dati di un impiegato;
- \* **createEmployee**: aggiunge un nuovo impiegato;
- \* **deleteEmployee**: rimuove 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, essi non verranno riportati. Per quanto riguarda i test sui controller sono stati svolti test più approfonditi. Per eseguire i test sui metodi del controller è stato scelto di utilizzare il framework JUnit5.

Il primo test realizzato è uno *Smoke Test* che si incarica di verificare 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**. Per ciascun controller è presente il relativo metodo di verifica, ciascuno annotato con **@Test** per indicare a 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 eccezioni se necessario.

```

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

```

@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 listAllEmployeeWithNotEmptyList() throws Exception {...}
    @Test
    void listAllEmployeeWithEmptyList() 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*.

La classe *EmployeeControllerTest* utilizza l'annotazione **@WebMvcTest** per caricare nel contesto di test esclusivamente il controller testato. Nella classe di test *EmployeeControllerTest* sono dichiarate tre dipendenze fondamentali risolte con la dependency injection:

- \* **MockMvc**: oggetto utilizzato per la simulazione di chiamate HTTP e la verifica della risposta;

- \* **ObjectMapper**: oggetto utilizzato nei test per la serializzazione e deserializzazione degli oggetti JSON in oggetti java e viceversa;
- \* **EmployeeService**: servizio che viene utilizzato nei test che usa l'annotazione **@MockBean** per aggiungere un mock dell'oggetto *EmployeeService*.

Infine si riporta in figura 4.11 l'implementazione dettagliata 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*.

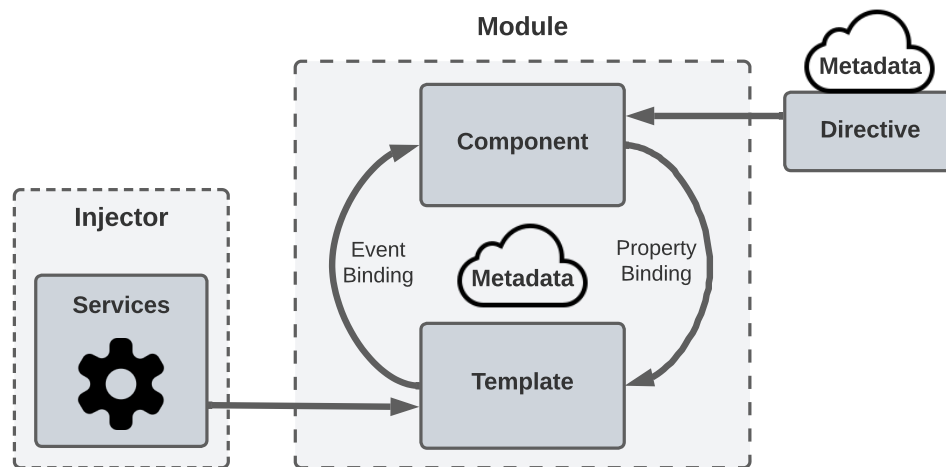
Nella prima parte (arrange) viene creato un nuovo *Employee* e gli viene assegnato un nome, un cognome e una data di nascita; successivamente (act) si esegue la chiamata GET utilizzando l'oggetto *MockMvc* e passando nel body della richiesta HTTP l'oggetto Java *Employee* trasformato in JSON grazie all'*ObjectMapper*.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

#### 4.1.4 Progettazione e realizzazione 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 considerazione sull'utenza e sono stati tralasciati completamente aspetti quali estetica grafica, accessibilità e responsive design.

#### Architettura

L'architettura del frontend 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 richieste e risposte HTTP.

### Funzionalità

Di seguito si approfondisce la parte riguardante l'entità Employee, le altre entità sono state trattate in maniera analoga.

In linea con le query rese disponibili dal backend, devono essere presenti le seguenti funzionalità:

- \* **Visualizzazione degli Employee:** devono poter essere visualizzata la lista degli impiegati permettendo la ricerca per: 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 sezione successiva verrà mostrato come sono state implementate le funzionalità appena introdotte.

## Realizzazione

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.

A seguire è stata implementata l'interfaccia 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 le scelte:

### Web Application Prototype

Seleziona il tipo per il quale si vuole eseguire l'azione

☐ Employee  
☐ Project  
☐ Department  
☐ Site

Seleziona il tipo di richiesta

☐ Get  
☐ Insert/Update  
☐ Delete

**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. 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 4.15 abbiamo il caso in cui è stata selezionata l'entità Employee con operazione GET:

List of all employees:

ID	Name	Surname	Salary	Birth Date
14A51DS	Marco	Rossi	1450	05/05/1998
DA134F3	Francesca	Bianchi	1600	26/11/1987
PFHI32A	Lucio	Verdi	1890	06/12/1978

Seleziona il campo da specificare nella ricerca

☒ Name  
☐ Surname  
☐ Date Birth  
☐ Department

Name:

**Figura 4.15:** Opzioni disponibili con operazione GET selezionata.

Viene dunque esposta una lista contenente tutti gli impiegati, per ciascun impiegato vengono mostrati id, nome e cognome, salario e data di nascita. Successivamente 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 selezionando 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:



Enter employee fields:

Id:

First name:

Surname:

Salary:

Birth:

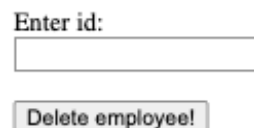
Department:

Projects:

**Figura 4.17:** Opzioni disponibili con operazione INSERT/UPDATE selezionata.

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 impiegato 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:



Enter id:

**Figura 4.18:** Opzioni disponibili con operazione DELETE selezionata.

## Testing

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 retrieve all employees by name', () => {--
  }
  it('should retrieve all employees by surname', () => {--
  }
  it('should retrieve 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 *HttpClientTestingModule*, necessario al fine di importare il servizio iniettabile *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'esecuzione 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: "AIF075", name: "Giulio", surname: "Gialli", salary: 1500, birth: new Date("02/01/1988")},
    {id: "LOSFD2", name: "Serafino", surname: "Bianchi", salary: 1500, birth: new Date("22/11/1979")},
    {id: "PIFAZA", 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 successivamente si verifica il risultato atteso tramite la funzione *expect*. Per finire si controlla che l'url sia corretto, che venga effettuata una sola chiamata GET e che si tratti effettivamente di una richiesta GET.

#### 4.1.5 Migrazione da REST a GraphQL

In questa sezione si affronta la migrazione sia del backend che del frontend 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.

È stata adottata la tecnica *Schema First* che prevede la costruzione del GraphQL Schema ed, a seguire, l'implementazione dei resolver delle API. Nonostante i resolver fossero già realizzati per il prototipo in REST è comunque necessario adattarli all'utilizzo di questo nuovo protocollo.

Le quattro entità presenti, ovvero Employee, Project, Department e Site devono essere riportate tutte nel GraphQL schema poiché sono tutte accessibili dal client; esse saranno definite sia come entità di input che di output poiché GraphQL differenzia i due tipi nelle query, mutation e subscription. In figura 4.21 è possibile visualizzare l'implementazione dei tipi *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: i campi *projects* e *department* di *Employee* non sono presenti in *EmployeeInput*, o meglio sono presenti ma in diversa forma. Il motivo di tale differenza si può sintetizzare in questo modo:

- \* il tipo *Employee* verrà ritornato al client su richiesta; dovrà contenere le varie istanze dei progetti a cui un impiegato sta partecipando come pure l'oggetto dipartimento;
- \* il tipo *EmployeeInput* indica come fornire i dati necessari all'aggiunta/aggiornamento dell'impiegato; è sufficiente dunque passare gli id dei progetti coinvolti come pure l'id del dipartimento dell'impiegato.

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
  employeeByDepartment(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.

Il dato ritornato è di tipo *Employee* per tutte le operazioni mentre quello di input è *EmployeeInput*.

È stata aggiunta inoltre una subscription per sfruttare a pieno le funzionalità di GraphQL. Questa subscription permette al client di ricevere i nuovi impiegati aggiunti senza per forza richiederli.

**Migrazione dei controller** La migrazione non coinvolge gli strati di servizio e repository in quanto non è necessario apportare modifiche alle logiche e alla gestione del database. L'unico strato da riscrivere parzialmente è lo strato del controller. Come prima cosa si realizzano le classi Java corrispondenti ai tipi di input; questo è necessario poiché nei metodi del controller è previsto come parametro di input la versione di input dell'impiegato.

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:

```

@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 si tratta di un controller simile a quello REST nei metodi, tuttavia le annotazioni e alcune implementazioni differiscono. Vengono utilizzate le annotazioni:

- \* **@Controller**: indica che si tratta di un generico controller (non più REST controller);
- \* **@QueryMapping**: attribuito ai metodi che risolvono le query;
- \* **@MutationMapping**: attribuito ai metodi che risolvono le mutation;
- \* **@SubscriptionMapping**: attribuito al metodo che risolve la subscription;
- \* **@Argument**: utilizzato per specificare gli argomenti.

Queste annotazioni sono fondamentali poiché permettono di effettuare il mapping tra i tipi, le query, le mutation e le 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 simile 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*:

```

@SubscriptionMapping
Publisher<List<Employee>> newEmployeeAdded(){
    return subscriber -> Executors.newScheduledThreadPool( corePoolSize: 1).scheduleAtFixedRate(() -> {
        List<Employee> employees= employeeService.selectAll();
        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 al verificarsi di un evento. L'oggetto *Publisher<>* permette di inviare la lista aggiornata a tutti i subscribers in tempo reale, così facendo il client non dovrà richiedere periodicamente la lista con la query *allEmployee*. **Mancano test in graphQL**

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

- \* **Data:** ritornato nella risposta solo in caso di successo, assume valore *null* al presentarsi di un errore;
- \* **Errors:** ritornato solo ed esclusivamente al verificarsi di uno o più errori, contiene le informazioni necessarie all'interpretazione dell'anomalia;
- \* **Extensions:** campo facoltativo a disposizione degli sviluppatori di backend, 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 individuano tre categorie:

- \* **Errori di sintassi:** si tratta di errori nella sintassi della query. Se la query è sintatticamente scorretta l'errore sarà di tipo "Syntax Error"; in questo caso l'esecuzione della query non parte nemmeno;
- \* **Errori di validazione:** si verificano quando vengono specificati nella richiesta tipi non coerenti con lo schema. 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 errori di data fetching;
- \* **Errori nei resolver:** si verificano durante la risoluzione della query e non sono legati al fatto che la query sia formulata male o con tipi sbagliati. Questi errori devono essere gestiti lato server;

Quando si verificano errori di sintassi o validazione la risposta l'errore ritornato indica perfettamente la natura dell'errore. La terza tipologia di errore quasi sempre risulta incomprensibile dal client. Questo accade perché la gestione delle eccezioni durante l'esecuzione di una richiesta è presa in carico dall'interfaccia predefinita *DataFetcherExceptionHandler* la quale permette la dichiarazione di più risolutori di eccezioni detti *DataFetcherExceptionResolver*. Questi ultimi vengono invocati sequenzialmente fino a quando uno di essi è in grado di risolvere l'eccezione; in caso contrario viene ritornata una informazione di errore incomprensibile con 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 GraphQLExceptionHandler extends DataFetcherExceptionHandlerAdapter {
    @Override
    protected GraphQLError resolveToSingleError(Throwable ex, DataFetchingEnvironment env) {
        if (ex instanceof EmployeeNotFoundError) {
            return GraphQLErrorBuilder.newError()
                .errorType(ErrorType.NOT_FOUND)
                .message(ex.getMessage())
                .path(env.getExecutionStepInfo().getPath())
                .location(env.getField().getSourceLocation())
                .build();
        } else {
            return null;
        }
    }
}

```

**Figura 4.25:** Classe *GraphQLExceptionHandler* 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. Al presentarsi dell'eccezione il client sarà ora in grado di comprendere il motivo di 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* utilizzato nel fetching dei dati da un server GraphQL.

In figura 4.26 è possibile visualizzare l'implementazione del metodo *allEmployees*.

```

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'endpoint del server GraphQL. Il metodo ritorna un oggetto di tipo *Observable* ed il subscriber dovrà occuparsi di gestire quanto ritornato dal servizio. Nel caso in cui la query sia andata a buon fine dovrà leggere le informazioni presenti nel campo *"data"*; nel caso contrario dovrà occuparsi di mostrare il messaggio ed il tipo dell'errore presente nel campo *"errors"*.

## 4.2 SushiLab

### 4.2.1 Confronto con stakeholder

Circa dopo un mese dall'inizio dello stage è stato realizzato un confronto con gli stakeholder. Con il tutor interno dell'azienda SyncLab, ovvero Fabio Pallaro sono stati valutati alcuni applicativi già implementati che seguissero l'architettura server - client per effettuarne la migrazione da REST API a GraphQL API.

Tra tutti l'applicativo selezionato è stato **SushiLab**, il quale è stato realizzato da alcuni stagisti nei mesi precedenti. Il motivo che ci ha portato a concordare questa scelta è stato principalmente il fatto che fosse un applicativo con una struttura API sviluppata.

### 4.2.2 Comprensione dell'applicativo e obiettivi

SushiLab è un applicativo che mira ad automatizzare il processo di tracciamento degli ordini nei ristoranti di sushi all you can eat. Si tratta da un applicativo composto da:

- \* **Backend:** realizzato un backend in Java con il framework Spring Boot, gestisce un inoltre database PostgreSQL;
- \* **Frontend:** realizzato in Typescript con il framework Angular;
- \* **Componente di Machine Learning:** utilizzata per il riconoscimento dei piatti tramite la fotocamera del telefono;
- \* **Applicativo mobile:** la versione mobile della web app sotto forma di applicazione.



Il mio intervento sull'applicativo prevede di lavorare sulla comunicazione tra backend e frontend, effettuando una migrazione completa delle REST API sviluppate portandole a GraphQL API, esattamente come è stato fatto per il prototipo.

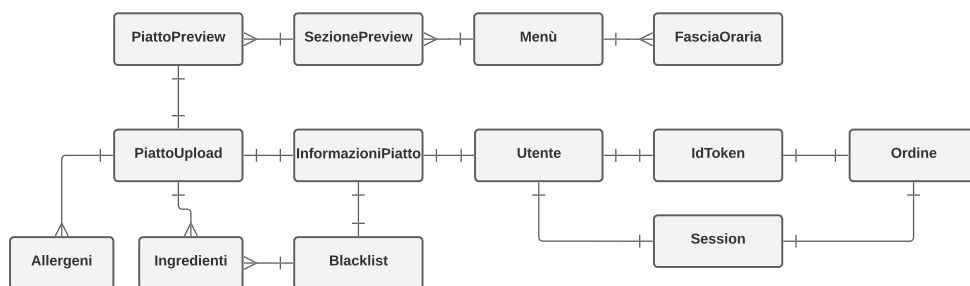
Di seguito nelle sottosezioni verranno analizzate le strutture del backend e del frontend, tralasciando gli aspetti implementativi specifici già visualizzati per il prototipo e concentrandosi principalmente sulla parte di comunicazione tramite API.

### 4.2.3 Panoramica del backend

Il backend sviluppato in Java con il framework Spring Boot segue l'architettura proposta per il prototipo, ovvero il pattern repository - service - controller. Di seguito verranno analizzate e descritte brevemente le componenti principali degli strati di persistenza e di business logic, mentre verrà approfondito lo strato di controller.

#### Entità

In figura 4.27 è possibile visualizzare lo schema ER delle entità presenti e le relazioni che intercorrono. Vengono tralasciati i campi per non entrare eccessivamente nel dettaglio.



**Figura 4.27:** Diagramma ER di *SushiLab*.

Sono dunque presenti le entità:

- \* **PiattoUpload:** una tipo di piatto con allergeni e ingredienti;
- \* **PiattoPreview:** l'anteprima di piatto associata ad un *PiattoUpload*;
- \* **SezionePreview:** un insieme di piatti di una particolare categoria;
- \* **Menù:** un insieme di sezioni di piatti, si tratta del menù di un ristorante;
- \* **Utente:** l'utente registrato, può anche essere un gestore e dunque essere il proprietario del ristorante e dunque poter apportare delle modifiche ai menù, ai piatti, ecc..;
- \* **FasciaOraria:** una fascia oraria in cui può essere disponibile un Menù;
- \* **InformazioniPiatto:** le caratteristiche di un singolo utente riguardo ingredienti che non può mangiare, se gli è piaciuto un piatto specifico o se ha preferenze per particolari tipi di menù;
- \* **Blacklist:** l'insieme di ingredienti che un cliente preferisce non mangiare;

- \* **IdToken:** un token identificativo associato ad un utente quando partecipa ad un tavolo;
- \* **Session:** la situazione di un singolo tavolo in un determinato momento con determinati clienti, quando un cliente si siede ad un tavolo può aprire una sessione e invitare gli amici allo stesso tavolo;
- \* **Ordine:** gli ordini effettuati in una specifica sessione o da una specifica persona;
- \* **Allergeni:** le sostanze che tipicamente possono causare allergie;
- \* **Ingredienti:** i possibili ingredienti nei vari piatti;

Ciascuna di queste entità ha la corrispettiva classe Java nel backend. Per ciascuna entità è stata realizzata la corrispettiva repository estendendo l'interfaccia *JPARespository* $\langle T, ID \rangle$  per la gestione dei dati e delle risorse nel database.

### Servizi

Lo strato di servizio presente prevede la definizione di solo due classi per la gestione delle logiche del backend. Di seguito vengono elencate le funzioni dei due servizi.

**ClientService** Il primo servizio gestisce le logiche delle possibili richieste del client. Questo servizio permette la gestione di:

- \* **Utenti:** permette tutte le operazioni di login, registrazione di un nuovo utente, aggiunta dell'IdToken ad un utente all'apertura di una nuova sessione ad un tavolo e infine il recupero e modifica della password;
- \* **Preferenze Utente:** permette di modificare le preferenze di un utente e modificare, aggiungere o eliminare le valutazioni attribuite ai piatti; inoltre permette di visualizzare gli ingredienti nella blacklist di uno specifico utente;
- \* **Sessioni:** permette di creare ed eliminare le sessioni;
- \* **Ordini:** permette la gestione degli ordini da parte di un utente, di aggiungerne di nuovi o di visualizzarli;
- \* **Gestore:** permette di verificare che un utente corrisponda o meno al gestore;
- \* **Menu:** permette di visualizzare il menù di un ristorante.

**GestoreService** Il servizio *GestoreService* gestisce le logiche riguardanti le operazioni realizzabili da un utente gestore, dunque permette:

- \* **Piatti:** la gestione dei piatti, è possibile visualizzarli, eliminarli, modificarli e aggiungerne di nuovi;
- \* **Menu:** la gestione del menù, è possibile riorganizzarlo diversamente e attribuirgli differenti fasce orarie di disponibilità.

### Controller

Infine abbiamo lo strato più importante per gli scopi della tesi. Si tratta dello strato di controller il quale prevede cinque controller principali, di seguito verranno analizzati tutti i metodi di ciascun controller tralasciandone gli aspetti implementativi.

**LoginController** Il seguente controller dichiara una dipendenza con il servizio *ClientService*. Di seguito la tabella dei metodi:

Metodo	Parametri	Descrizione	Azione
<b>eseguiLogin</b>	Utente utente	Controlla ed effettua il login di un utente	POST

**Tabella 4.1:** Metodi del controller *LoginController*.

**UtenteController** Questo controller dichiara una dipendenza con il servizio *ClientService* ed è composto dai metodi nella specificati nella seguente tabella:

**TavoloController** Anche il seguente controller dichiara una dipendenza con *ClientService* e i suoi metodi gestiscono gli utenti, gli ordini ed i tavoli.

- \* **UtenteController**: espone le API per la gestione degli utenti;
- \* **TavoloController**: espone le API per la gestione delle sessioni nei tavoli e dunque degli ordini;
- \* **MenuController**: espone le API per la visualizzazione dei menù
- \* **LoginController**: espone le API per il login;
- \* **GestoreController**: espone le API per tutte quelle funzionalità disponibili al gestore;

### 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).

#### 4.2.4 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.

#### 4.2.5 Migrazione del FE da REST a GraphQL

Metodo	Parametri	Descrizione	Azione
<b>ottieniUtente</b>	String idPersona	Ricerca un utente per id	GET
<b>registraUtente</b>	Utente utente	Aggiunge un nuovo utente	POST
<b>modificaStatoPreferiti</b>	String idPersona String idPiatto	Modifica i preferiti di un utente, con gli dei piatti	POST
<b>modificaValutazione</b>	String idPersona String idPiatto String param	Modifica la valutazione di una persona rispetto ad un piatto con nuovi parametri	POST
<b>recuperoPassword</b>	String param	Ritorna la password recuperata se vengono forniti i parametri necessari	POST
<b>verificaCodice</b>	String param	Verifica il codice di recupero password	POST
<b>reimpostaPassword</b>	String param	Imposta una nuova password per un utente	POST
<b>aggiornaBlacklist</b>	String idPersona Ingredienti ingredienti	Aggiorna la blacklist di una persona con i nuovi ingredienti	POST
<b>ottieniBlacklist</b>	String idPersona	Ritorna la lista di ingredienti nella blacklist di un utente	GET

Tabella 4.2: Metodi del controller *UtenteController*.

Metodo	Parametri	Descrizione	Azione
<b>creaSessione</b>	String idPersona	Permette la creazione di una sessione da parte di una persona	POST
<b>ottieniSessione</b>	String idTavolo	Ritorna la sessione attiva in un determinato tavolo	GET
<b>chiudiSessione</b>	String idTavolo	Chiude la sessione in un determinato tavolo	DELETE
<b>ottieniOrdiniPersona</b>	String idTavolo String idPersona	Ritorna gli ordini di una persona in uno specifico tavolo	GET
<b>modificaOrdiniPersona</b>	String idTavolo String idPersona	Modifica gli ordini di una persona in uno specifico tavolo	POST
<b>ottieniOrdiniTavolo</b>	String idTavolo String idPersona	Ritorna tutti gli ordini effettuati dalle persone di un determinato tavolo	GET
<b>ottieniGliOrdiniInArrivo</b>	String idTavolo String idPersona	Ritorna gli ordini che sono quasi pronti per essere consegnati	GET
<b>spostaGliOrdiniInArrivo</b>	String idTavolo	Imposta come in arrivo gli ordini di un determinato tavolo	POST



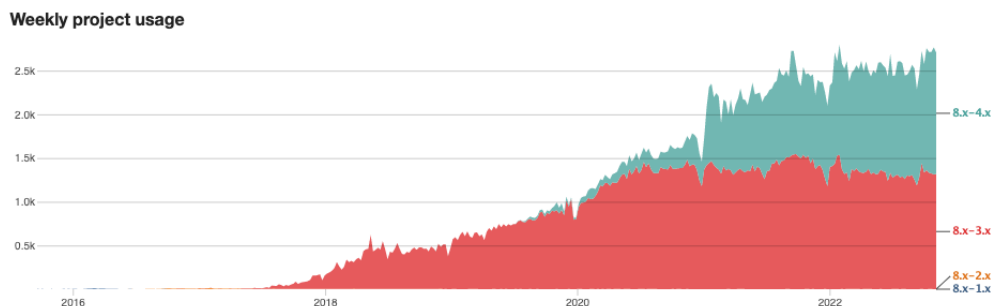
## Capitolo 5

# 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

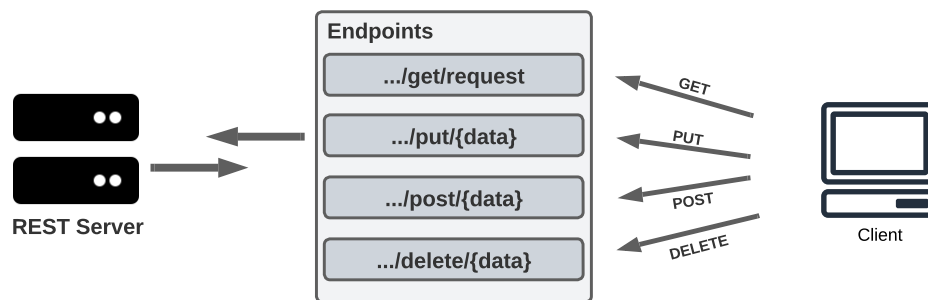
Come sottolineato in precedenza, GraphQL e REST hanno diversi aspetti che li differenziano. La più grande differenza tra questi due protocolli è legata alla loro natura: infatti quando si fa riferimento a REST, si sta parlando di uno stile architetturale, dunque di un modo di costruire le proprie API le quali, se rispettano i vincoli REST illustrati al punto 3.2.1, 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. Per endpoint si intendono i punti di accesso forniti dal server per permettere al client di eseguire richieste.

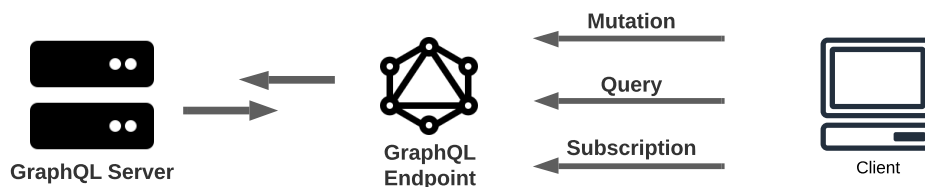
Lo stile REST prevede l'utilizzo di più endpoint, sfrutta infatti la molteplicità degli endpoint per differenziare le richieste possibili. Quando un client implementa una richiesta a delle REST API deve sapere esattamente a quale endpoint inviare la richiesta per ricevere i dati necessari. In figura 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 endpoint 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."*

Con questa fra L. Byron rappresenta perfettamente l'identità di GraphQL che si propone come sistema innovativo che prevede di distribuire le risorse ordinatamente su un grafo e non, come in REST, su diversi endpoint.

### 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 invia 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	JSON RETURNED
<pre> query {   allEmployees {     id     name   } } </pre>	<pre> "data": {   "allEmployees": [     {       "id": "3AFASDF12F",       "name": "Matteo"     },     {       "id": "GA14PL3FAV",       "name": "Marco"     }   ] } </pre>

In GraphQL questo approccio 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	JSON RETURNED
<pre> query {   allEmployees {     id     projects {       id       name     }   } } </pre>	<pre> "data": {   "allEmployees": [     {       "id": "3AFASDF12F",       "projects": [         {           "id": "7NFAISH280",           "name": "Progetto Beta"         },         {           "id": "N2A8F234SD",           "name": "Progetto Teta"         }       ]     }   ] } </pre>

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 riportati 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 introspezione, 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 moltissimi specifici 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 approcci classici, alcuni prediligono l'evoluzione delle API, altri ritengono necessario versionarle. GraphQL predilige un approccio 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 approccio 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 approcci, 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 dall'esigenza di poter utilizzare connessioni bidirezionali, non si tratta di una funzionalità prevista nella natura del protocollo come avviene nel caso di GraphQL.

**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 impostando 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 progettazione, la realizzazione e la migrazione del prototipo e dell'applicativo SushiLab sono risultati evidenti alcune caratteristiche dei protocolli analizzati che riguardano più l'aspetto implementativo legato alla mia esperienza, per questo motivo è stato scelto di riportarle in una sezione a sé stante.

**Documentazione online** Lo stile architetturale REST vanta una maggior quantità di documentazione online rispetto GraphQL. Questo è stato notato soprattutto durante lo sviluppo del backend, infatti sono state diverse le volte in cui è capitato di dover risolvere diversi tipi di errori. In REST risulta molto più semplice e immediato trovare la soluzione a quello che si sta cercando, in GraphQL invece non è così, infatti spesso alcuni problemi possono risultare complessi da risolvere in quanto non sono presenti nel web altri situazioni simili oppure capita spesso anche che siano questioni già sollevate, ma non ancora risolte. Questo è dovuto al fatto che REST sia un protocollo di data fetching affermato e utilizzato ampiamente dalla maggior parte degli sviluppatori, oltre ad essere utilizzato da molto più tempo rispetto a GraphQL, il quale viceversa è un protocollo giovane ed utilizzato ancora da una piccola, anche se in crescita, nicchia di sviluppatori.

**Complessità nell'apprendimento di REST e GraphQL** Durante l'apprendimento di questi due protocolli di data fetching è stata notata una differenza di complessità tra di essi. REST risulta molto semplice e intuitivo sia dal punto di vista del backend che del frontend, sono presenti molti meno vincoli rispetto a GraphQL ed è sufficiente seguire alcune linee guida e best practices per realizzare delle REST API complete e funzionanti. GraphQL invece può risultare un po' più complesso in quanto è necessario

comprenderne bene i meccanismi altrimenti si rischia facilmente di creare un sistema di API fallimentare. Una delle parti che sicuramente risulta essere complessa da apprendere è la realizzazione di un GraphQL Schema ben strutturato che sia in grado di esporre le logiche, le entità e relative relazioni del backend che necessitano di essere esposte e al contempo di oscurarne i punti più vulnerabili, infatti la flessibilità di questo linguaggio di query può risultare in molte circostanze un grosso vantaggio, ma può anche rivelarsi deleteria per il server realizzato. Per realizzare delle buone GraphQL API è sicuramente necessaria una buona padronanza del dominio e molta esperienza.

**Sviluppo e migrazione backend** Durante lo sviluppo e la migrazione dei backend con i due protocolli di data fetching sono state notate alcune differenze significative. Sicuramente il linguaggio di query GraphQL richiede un maggior sforzo lato backend per la realizzazione di buone API. Una delle principali criticità riscontrate durante la migrazione del backend da REST a GraphQL è stata sicuramente la questione del mapping tra i tipi definiti in Java e quelli definiti nel GraphQL Schema, infatti si è rivelato essere uno dei principali motivi dietro alla comparsa di frequenti errori. Oltre ad essere spesso necessario dichiarare in Java ulteriori classi per rappresentare diverse versioni della stessa entità, ad esempio la versione di input che, come visto nel capitolo ?? spesso risulta differente, è stato complesso anche comprendere come strutturare i tipi rispettando il mapping e, allo stesso tempo, realizzare un GraphQL Schema che non fosse dispersivo. Infatti spesso si rischia di introdurre nuovi tipi di vario genere per risolvere velocemente problemi di mapping, ma così facendo si rischia di introdurre disorganizzazione e di rendere più complessa la manutenibilità del server.

Queste problematiche non sono presenti in REST il quale risulta molto più semplice da questo punto di vista, infatti è sufficiente associare i vari endpoint ai vari resolver nello strato di controller, i quali comunicano con le stesse classi dello strato di servizio e di persistenza, dunque non si creano due sorte di ambienti differenti che è necessario far combaciare tra loro come invece avviene in GraphQL.

Un altro aspetto che sicuramente ha reso più complessa la realizzazione del server di backend in GraphQL rispetto a REST è stata la gestione degli errori. Come spiegato precedentemente la compatibilità di REST con il protocollo HTTP, il quale è stato utilizzato per la comunicazione tra server e client, ha permesso una più facile e veloce gestione degli errori, in quanto a parte qualche specifica necessaria, risultano essere già chiari e significativi. L'uso improprio che GraphQL fa del protocollo HTTP si riflette anche nella gestione degli errori i quali, se non trattati approfonditamente, rischiano di essere totalmente inespressivi.

**Sviluppo e migrazione frontend** Per quanto riguarda lo sviluppo del frontend con i due protocolli di data fetching non sono state riscontrate considerevoli differenze e criticità con un protocollo piuttosto che con l'altro. L'unico aspetto che è importante riportare è la questione legata all'overfetching e underfetching; principalmente per questo motivo lo sviluppo del frontend con protocollo REST risulta essere leggermente più complesso rispetto allo sviluppo del medesimo frontend in GraphQL. Infatti nel caso dell'underfetching è necessario un maggior lavoro lato frontend nella gestione dei dati i quali, dopo aver inviato più richieste, devono essere fatti combaciare per estrarne la forma richiesta per i propri scopi. Per quanto riguarda l'overfetching invece sarà necessario estrapolare dai dati ricevuti, in una forma più estesa di quanto richiesto, i campi necessari. In GraphQL questo non accade poiché nella query può essere direttamente specificato il tipo e la forma dei dati che si desidera ricevere. Il frontend in REST dunque risulta molto più limitato rispetto al medesimo in GraphQL poiché

deve accontentarsi delle API fornite dal backend lottando con problemi di overfetching e underfetching o, in casi estremi, richiedere l'implementazione di API specifiche.

### Analisi comparativa prestazionale

Fare riferimento al documento B4.

## 5.3 Conclusioni

L'analisi comparativa effettuata ha mostrato come entrambi i protocolli di data fetching REST e GraphQL abbiano vantaggi e svantaggi. In questa sezione si vuole definire quale sia il miglior approccio a seconda dell'applicativo che si vuole realizzare. Sicuramente non c'è un protocollo che può essere ritenuto migliore rispetto all'altro in maniera assoluta, infatti è necessario analizzare bene il caso d'uso e comprendere quando sia preferibile uno piuttosto che l'altro. Si tratta principalmente di una scelta che deve essere fatta lato backend, questo perché è nel server di backend che c'è l'impatto maggiore.

### 5.3.1 Linee guida prima di intraprendere uno sviluppo o la migrazione di un applicativo da/in REST VS GraphQL

Quando si vuole realizzare un nuovo applicativo è importante valutare a fondo quale delle due soluzioni può risultare migliore per il caso specifico. Si tratta di una scelta fondamentale in quanto l'utilizzo di un protocollo piuttosto che l'altro può influire consistentemente sul futuro dell'applicativo. Gli sviluppatori frontend sicuramente tendono a preferire GraphQL, grazie alla sua flessibilità semplifica ampiamente il loro lavoro. Viceversa gli sviluppatori del backend tendono a preferire REST, è molto più semplice, sono presenti molti più strumenti e GraphQL risulta ancora giovane come tecnologie, dunque è difficile scalare un server in GraphQL e rendere sicuri ed efficienti argomenti come l'autenticazione.

**Analisi dal punto di vista di uno sviluppatore frontend** Sicuramente per uno sviluppatore frontend GraphQL è nella maggior parte dei casi la soluzione migliore. Ci sono diversi motivi per i quali un developer frontend può preferire GraphQL a REST, tra questi troviamo:

- \* **Chiarezza delle API:** l'autodocumentazione di GraphQL è sicuramente un aspetto gradito dai frontend developer, è semplice comprendere come interagire con il server grazie alla chiarezza dell'interfaccia di facciata che viene necessariamente realizzata, ovvero il GraphQL schema. Inoltre anche nell'evoluzione delle GraphQL API è sempre chiaro quali campi sono deprecati e quali invece conviene utilizzare;
- \* **Flessibilità:** la flessibilità che le API GraphQL concedono al client è un privilegio non da poco per i frontend developer, infatti è possibile specificare esattamente il quantitativo, la forma e il tipo dei dati che si vuole ricevere; questo aspetto permette di non ritrovarsi in situazioni tipiche con le API REST dove spesso, per riuscire a ricevere i dati nella forma desiderata, è necessario eseguire più richieste al backend e, in seguito, modellare i dati per ottenere la forma finale;



- \* **Chiarezza delle operazioni:** in GraphQL le richieste vengono costruite in maniera più chiara e organizzata, in REST ci si può spesso trovare in situazioni in cui è necessario specificare diversi parametri nell'url per la richiesta e questo può causare confusione e richieste sbagliate; inoltre è possibile eseguire le richieste ad un singolo endpoint, cambiando solo il body della richiesta;
- \* **Dati in tempo reale:** risulta più semplice e performante realizzare Real Time Application grazie al tipo subscription permesso dal protocollo GraphQL, in REST, come spiegato precedentemente, è necessario qualche passaggio in più;

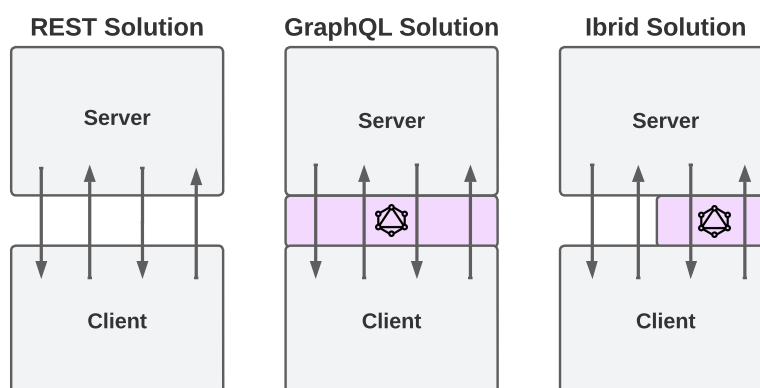
**Analisi dal punto di vista di uno sviluppatore backend** Per lo sviluppatore backend la questione è ben differente. Sviluppare un backend con API GraphQL risulta molto più complesso e dispendioso rispetto a REST. Tuttavia sono presenti alcune situazioni in cui GraphQL può essere una soluzione migliore:

- \* **Per certi tipi di client:** è fondamentale, prima di sviluppare il server di backend, fare un'analisi sulle varie tipologie di client che verranno serviti dalle proprie API. Di seguito vengono elencate delle situazioni in cui delle API GraphQL risulterebbero più adatte:
  - quando si ha a che fare con smartphone, smartwatch, socialmedia, dispositivi di IoT e tutti quei dispositivi per i quali la latenza e la larghezza della banda sono fondamentali. È stato ampiamente dimostrato che tempi di risposta lenti disincentivano gli utenti dall'utilizzo di un'applicazione, per questo GraphQL risulta un'ottima soluzione per i dispositivi mobili riducendone i tempi di latenza ed aiutando a prelevare esclusivamente i dati necessari;
  - con client in continua evoluzione, quando si servono diversi tipi di client e infine quando si servono client propensi a richiedere query specifiche e annidate. Infatti in questi casi spesso capita che anche le richieste possano variare nel tempo, che siano mandate da client differenti o che siano molto specifiche. In queste situazioni GraphQL lascia flessibilità al client il quale può specificare e modellare le sue query, mutation o subscription in qualsiasi momento, senza essere vincolati dalle API fornite dal backend;
- \* **Il dominio dei dati complesso:** il dominio dei dati che un server di backend deve trattare è fondamentale nella scelta del protocollo. Quando si trova a dover manipolare molti dati distribuiti in diverse entità con molte relazioni tra loro, GraphQL risulta un'ottima soluzione in quanto permette di specificare e organizzare i vari tipi che si vuole esporre in un unico posto, il GraphQL Schema. Questo permette di dare chiarezza e organizzazione alle API. Bisogna stare però attenti quando si trattano dati complessi perché GraphQL fornisce molti punti di accesso e questo può essere un problema per la sicurezza.
- \* **In un backend a microservizi:** in un backend di questo genere GraphQL può tornare molto utile, infatti ciascun microservizio gestisce le logiche e i dati di uno specifico dominio ed è possibile realizzare uno schema GraphQL per ciascun microservizio e, successivamente, un GraphQL Schema comprensivo di tutti i vari schemi GraphQL, così facendo si può esporre un'unica interfaccia ai client mascherando la divisione interna;
- \* **Per backend in continua evoluzione:** in questo caso l'approccio evolutivo delle GraphQL API spiegato precedentemente ritorna molto utile, in quanto

mantenendo un solo endpoint e semplicemente creando nuovi campi e deprecando i più vecchi è possibile mantenere una versione sempre aggiornata delle API senza complicare il lavoro al client;

In quasi tutti gli altri casi intraprendere lo sviluppo di un backend in GraphQL può risultare svantaggioso e dispendioso in fatti seguire lo stile architetturale REST per la realizzazione delle proprie API ancora oggi risulta essere vincente in una gran parte di applicativi.

**Migrazione** Intraprendere una migrazione da REST API a GraphQL API è una soluzione che richiede una grossa riflessione prima di essere effettuata. Si tratta innanzitutto di un'operazione molto dispendiosa e deve avere forti e valide motivazioni alla base. Ci sono molti aspetti che disincentivano la migrazione, in primis il fatto che servizi terzi, tool vari e librerie che permettono uno sviluppo veloce in REST non sono presenti in GraphQL e dunque realizzare le medesime funzionalità può risultare dispendioso. La sicurezza è un altro aspetto da considerare insieme al caching e la gestione degli errori, anche questi sono aspetti che richiedono molto più tempo in GraphQL per essere implementati. Inoltre molte compagnie evitano di intraprendere questo percorso, anche se magari può risultare vantaggioso a lungo termine, perché i clienti per i quali forniscono le API non sono pronti a migrare in GraphQL. Negli ultimi anni si stanno realizzando delle soluzioni ibride per andare in contro ai bisogni di tutti, come mostrato in figura 5.4



**Figura 5.4:** Soluzione ibrida mantenendo sia REST API che GraphQL API.

Questa soluzione prevede di mantenere le vecchie REST API e aggiungere un GraphQL Schema con nuovi resolver che, possono avere implementazioni differenti o, invocare a loro volta i resolver delle REST API. Questa soluzione permette di fornire ai clienti sia REST API che GraphQL API o, in caso sia un obiettivo degli sviluppatori backend, realizzare una migrazione da REST a GraphQL uno step alla volta.

## 5.3.2 Tabella di comparazione riassuntiva

Protocolli	REST	GraphQL
Endpoint	Endpoint multipli	Endpoint singolo
Data fetching	Problemi di overfetching e underfetching	Risolve l' overfetching e underfetching
Metodi HTTP	GET, POST, PUT e DELETE	POST
Codici di stato HTTP	Utilizza tutti i codici di stato	Utilizza solo il codice di stato 200
Caching	Sfrutta il caching previsto nel protocollo HTTP	Non sfrutta il caching HTTP, è responsabilità del client
Documentazione	Necessari tool di terze parti	Si autodocumenta
Sicurezza	Molto sicuro	Meno sicuro
Formato dati supportati	JSON, XML, YAML	JSON
Versionamento ed evoluzione	Possibili entrambi gli approcci	Possibili entrambi gli approcci, ma viene preferito il primo
Trasmissione dati in tempo reale	Possibile con escamotage	Previsto nella natura del protocollo
File uploading	Previsto	Non previsto, ma con le librerie si può ovviare il problema
Documentazione	Ampia documentazione	Documentazione ridotta
Apprendimento	Più semplice e veloce	Più complesso e lento
Sviluppo backend	Più semplice da realizzare	Più complesso per questioni di mapping e gestione errori
Sviluppo frontend	Problemi con overfetching e underfetching, è più limitato	Più semplice e flessibile



## 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](#)).