GraphQL

SLIDE 1

è un meccanismo di accesso alle API creato da Facebook nel 2012, ma rilasciato open source solo nel 2015. Si tratta di un linguaggio di interrogazione (o query language)

GraphQL non rappresenta una rottura totale con REST, bensì prosegue sulla stessa via ereditandone i principali elementi fondanti:

fa leva sul protocollo HTTP sfruttando gli elementi che al suo interno lo predispongono naturalmente ad essere un vettore di richieste e risposte tra client e server;

la caratteristica principale è quella di recuperare i dati in maniera precisa e puntuale.

SLIDE 2

GRAPHQL VS REST

Quello che per GraphQL può essere considerato il principio cardine (aspetto in cui vuole contraddistinguersi da REST ) è la **capacità di permettere ai client di richiedere solo i dati cui si è interessati**, con la massima precisione.

Il primo è l’**over-fetching**, situazione in cui un endpoint risponde troppo genericamente offrendo una gran mole di dati che il client in buona parte non usa, ma che deve comunque elaborare. Il secondo problema, opposto al primo, prende nome di **under-fetching** per cui un endpoint risponde con informazioni troppo stringate per cui un client può avere un set completo di dati solo inoltrando più richieste con le immaginabili conseguenze in termini di efficienza e consumo di banda, tanto per sè stesso quanto per il server.

Oltre a questo aspetto, GraphQL presenta ulteriori punti di forza che ne hanno permesso un veloce successo tra un grande pubblico di sviluppatori:

* mentre REST si fonda sul principio della “mappatura” di risorse con endpoint di rete, GraphQL permette di accedere a più risorse attraverso una singola richiesta. Questo permette di **evitare richieste multiple** a più URL, come può capitare in REST: una caratteristica introdotta strizzando l’occhio al mondo mobile;
* si favorisce al massimo l’utilizzo dei **tipi di dato** per indurre una maggiore identificazione dei dati ed una migliore visibilità sugli errori;
* la flessibilità nella formulazione delle query permette di predisporre ulteriori campi senza compromettere la correttezza delle interrogazioni già in produzione. Questo è sempre stato un limite di REST nel quale, infatti, si ricorre spesso al *versioning* delle funzionalità per poter distinguere con quali API un client può dialogare. **Il versioning non trova pertanto applicazione in GraphQL**.
* GraphQL semplifica il mock, perché ogni backend GraphQL viene fornito con un sistema di tipi statici. I tipi possono essere condivisi tra il tuo backend e il tuo frontend e contengono tutte le informazioni necessarie per rendere il mocking incredibilmente veloce e conveniente.

**Perché MOCK?**

MOCK i dati che un backend restituirebbe è molto utile per due motivi principali:

1. Ti consente di iniziare a sviluppare una funzionalità di frontend quando non hai ancora un backend funzionante. Questo è fondamentale per i progetti in cui i componenti frontend e backend sono spesso sviluppati in parallelo.
2. Ti consente di eseguire test localmente senza connetterti a un vero backend, che è molto più veloce e sicuro. Man mano che la tua base di codice cresce e la tua app diventa più complessa, l'avvio di tutta l'infrastruttura del server solo per eseguire alcuni test non è fattibile.

Se deridere la tua API di backend ha vantaggi così evidenti, perché non lo fanno tutti? Penso che sia perché deridere spesso sembra troppo disturbo per valerne la pena.

CONTRO

 *Nessuna memorizzazione nella cache;* Poiché tutte le richieste a un endpoint GraphQL sono richieste POST, molte applicazioni Web non supportano la memorizzazione nella cache. La maggior parte di essi supporta solo la memorizzazione nella cache delle richieste GET.

 *Più difficile da ottimizzare;* Il cliente ha molta flessibilità per richiedere ciò di cui ha bisogno. Poiché l'API può ricevere molte richieste diverse, è più difficile ottimizzare le prestazioni.

 *Non adatto per la restituzione di file;* Anche se potrebbe essere possibile restituire file (codificati Base64 per esempio), ciò non è consigliato. Se è necessario, probabilmente è meglio restituire un URL al file (ospitato da qualche altra parte).

GRAPHQL VS REST PT. 2

Una cosa interessante da sottolineare è che GraphQL non è legato a nessun database specifico (o motore di archiviazione per quella materia) ed è invece supportato da codice esistente. Ciò significa che, a differenza delle API REST (in cui il client interagisce prima con codice arbitrario scritto dai programmatori e questo codice raggiunge il database); il client interagisce prima con GraphQL, che a sua volta interagisce con codice arbitrario e alla fine finisce di parlare con il database. Un diagramma più utile per rappresentare questa situazione è questo:

Questo cambiamento nell'architettura ha molti vantaggi ad esso legati, ad esempio è possibile ottenere tutti i dati di cui il cliente ha bisogno in una singola richiesta (mentre le API REST devono eseguire più richieste).

ASPETTI PRINCIPALI

Schema Definition Language

L’endpoint di GraphQL espone uno **schema** che descrive quali tipi di dati (query) possono essere utilizzati dal client. Oltre ai tipi di dati, espone anche le operazioni che possono essere eseguite. Fondamentalmente, le operazioni possono essere suddivise in 3 categorie: query, mutazioni e abbonamenti.

QUERY

L'idea alla base di GraphQL era quella di creare un'API che i client possono utilizzare per consumare i dati in modo efficiente, ed è esattamente ciò che le query ti consentono di fare. Solo i tipi di dati che vengono esposti tramite una query possono essere utilizzati dal client. Quando si utilizza una query, il client può richiedere una serie di campi invece di ottenere tutti i campi da quel tipo. Ciò consente alle richieste di essere molto più efficienti.

Quando un tipo di dati fa riferimento ad altri tipi di dati (tipo nidificato), puoi recuperare anche questi (sempre utilizzando una query). Anche se non esiste una query specifica che espone questo tipo.

MUTATION

Con le mutazioni il cliente può aggiungere, modificare o eliminare dati. Solo le mutazioni esposte dallo schema possono essere utilizzate per modificare i dati. Se non è presente alcuna mutazione nello schema GraphQL, il client non è in grado di manipolare i dati nell'API.

Esempio:  
basandosi sull'esempio precedente, sarebbe un'ottima aggiunta consentire all'utente di aggiornare il proprio stato utente. È qui che entra in gioco una mutazione. Esponendo una mutazione "ChangeUserStatus" nello schema GraphQL, il client può aggiornare lo stato tramite questa mutazione.

SUBSCRIPTION

Un’altra funzionalità di GraphQL si chiama Sottoscrizioni. Ciò consente a un client di ricevere notifiche sugli eventi che stanno accadendo. Con la crescita delle architetture guidate dagli eventi, questa può essere una funzionalità piuttosto potente. Per ottenere notifiche (quasi) in tempo reale, i WebSocket vengono utilizzati come protocollo.

**MQTT**[[1]](https://it.wikipedia.org/wiki/MQTT#cite_note-1) (**Message Queue Telemetry Transport**) è un [protocollo](https://it.wikipedia.org/wiki/Protocollo_di_rete) [ISO standard](https://it.wikipedia.org/wiki/International_Organization_for_Standardization) di messaggistica leggero di tipo [publish-subscribe](https://it.wikipedia.org/wiki/Publish/subscribe" \o "Publish/subscribe) posizionato in cima a [TCP/IP](https://it.wikipedia.org/wiki/Suite_di_protocolli_Internet).

Esempio:  
come accennato, gli abbonamenti sono un ottimo modo per ricevere notifiche sulle modifiche nell'API. Quando si esaminano esempi precedenti basati su GitHub, le notifiche di commenti su una richiesta di pull sarebbero un ottimo caso d'uso. Quando un client si iscrive a questo tipo di evento (nuovo evento di commento), l'API invierà una notifica utilizzando [WebSocket](https://translate.google.com/website?sl=en&tl=it&u=https://en.wikipedia.org/wiki/WebSocket) quando viene pubblicato un commento.