Introducción a GraphQL

Javier Cremona

13 de Marzo de 2020

REST: Representational State Transfer

REST es un estilo de arquitectura que define un conjunto de restricciones a usarse en la creación de Servicios Web.

- Permite la comunicación entre distintos sistemas.
- Usa estándares (HTTP, XML, JSON, etc.), lo que permitió que los desarrolladores lo adopten rápidamente.

REST: Representational State Transfer

En general, REST se usa sobre HTTP. Facilita la implementación de APIs, desacoplando el servidor y el cliente.

- Emplea los principales métodos HTTP: GET, POST, PUT, DELETE, PATCH.
- El intercambio de datos se suele realizar en formato JSON o XML.
- Se emplean los códigos HTTP estándar en la respuesta. Por ejemplo: 200 para una petición correcta, 404 es «Not Found».

REST: Representational State Transfer

GET /movies/

```
{"movies":
    [{"name": "Titanic",
      "id": 217360881,
      "director": "James Cameron"
      "cast": [{"name": "Leonardo DiCaprio",
                "age": 45,
                "awards": [...]},
               {"name": "Kate Winslet",
                "age": 44,
                "awards": [...]}]},
     {"name": "Kill Bill",
      "id": 184766018,
       ...}
    ]}
```

REST: Representational State Transfer

GET /movies/<id>/

```
{"movie":
    {"name": "Kill Bill",
      "id": 184766018,
      "director": "Quentin Tarantino"
      "cast": [{"name": "Uma Thurman",
                  "age": 49,
                  "awards": [...]}
                ]
    }
}
```

REST: Representational State Transfer

Características:

- Arquitectura cliente-servidor.
- El identificador de cada recurso es la URI.
- Stateless: la petición contiene toda la información necesaria para ejecutarla.
- HATEOAS: en la respuesta se retornan links a otros recursos asociados.

REST: Representational State Transfer

Ventajas de implementar un API con REST:

- Desacople entre cliente y servidor, lo que permite que evolucionen independientemente.
- Escalabilidad: en sistemas distribuidos, cualquier nodo puede responder a una petición ya que no mantienen información de sesión durante la interacción.
- Emplea estándares como HTTP, JSON y XML.

REST: Representational State Transfer

Desventajas:

- Overfetching.
- Underfetching.
- Múltiples queries para obtener los datos.

Introducción a GraphQL

GraphQL es un lenguaje de consulta y manipulación pensado para comunicar clientes y servidores.

- Desarrollado inicialmente por Facebook.
- Provee un enfoque y herramientas para desarrollar APIs web.
- Permite a los clientes definir la estructura de datos que los mismos requieran.

GraphQL permite la consulta de datos mediante un enfoque *declarativo*. Un cliente específica exactamente qué datos desea obtener del servidor.

A diferencia de REST, donde se exponen múltiples endpoints que retornan una estructura de datos fija, GraphQL define un único endpoint. En dicho endpoint responde con la información solicitada por el cliente.

Schema Definition Language (SDL)

GraphQL posee su propio sistema de tipos que permite definir el esquema de un API.

```
type Employee {
    name: String
    age: Int
}
```

Se pueden expresar relaciones entre los tipos definidos.

```
type Department {
    name: String
    employees: [Employee]
}
```

Queries

Ejemplo de una consulta básica:

```
{
  allEmployees {
    name
  }
}
```

Resultado:

```
{
  "allEmployees":
      [{"name": "Cecilia"},
       {"name": "Lucas"},
       {"name": "Javier"}]
}
```

Queries

Y la edad de cada empleado? Con GraphQL, el cliente puede consultar solamente los fields que desee obtener!

En GraphQL, mediante una Query, los clientes puede obtener datos del servidor de una forma más flexible que con REST.

Argumentos

Volvamos al ejemplo de las películas.

```
query {
    movie(name:"Titanic"){
        name
        releaseDate
    }
}
```

Resultado:

```
{
    "movie": {
        "name": "Titanic",
        "releaseDate": "05-02-1998"
    }
}
```

Argumentos

Mejor aún, cada field puede tener sus propios argumentos.

```
query {
    movie(name:"Titanic"){
        name
        releaseDate(format: ONLY_YEAR)
    }
}
```

Resultado:

```
{
    "movie": {
        "name": "Titanic",
        "releaseDate": "1998"
    }
}
```

Alias

Cuando consultamos un mismo field, usamos un alias para renombrar el resultado.

```
query {
  titanic: movie(name:"Titanic"){
            name
            releaseDate
  }
  wws: movie(name:"The Wolf of Wall Street"){
            name
            releaseDate
  }
}
```

Alias

Resultado:

```
{
    "titanic": {
        "name": "Titanic",
        "releaseDate": "05-02-1998"
    },
    "wws": {
        "name": "The Wolf of Wall Street",
        "releaseDate": "25-12-2013"
    }
}
```

Fragments

Podemos usar *fragments* para evitar repetir campos.

```
query {
  titanic: movie(name:"Titanic"){
        ...movieFields
  }
  wws: movie(name:"The Wolf of Wall Street"){
        ...movieFields
  }
}

fragment movieFields on Movie {
    name
    releaseDate
}
```

Variables

```
query ($name: String) {
  movie(name: $name) {
    name
    releaseDate
  }
}
```

Sistema de Tipos

Cómo se relacionan las *queries* mostradas recientemente con el sistema de tipos y los esquemas?

```
type Query {
    movies: [Movie]
    movie(name: String!): Movie
}

type Movie {
    name: String!           # non-nullable
    releaseDate: String
    director: Director
    actors: [Actor]
}
```

Sistema de Tipos: Listas

```
type Director {
    fullName: String!
    age: Int
}

type Actor {
    fullName: String!
    age: Int
    appearsIn: [Movie]
}
```

Queries: consultando objetos

```
query {
    movie(name:"Titanic"){
        name
        director {
            fullName
        }
} }
```

```
{
    "movie": {
        "name": "Titanic",
        "director": {
            "fullName": "James Cameron"
        }
    }
}
```

Sistema de Tipos: Scalars

GraphQL viene con un conjunto de tipos por defecto:

- String
- Int
- Float
- Boolean
- ID

Sistema de Tipos: Enums

GraphQL permite definir tipos enumerados.

```
enum Day {
    MONDAY
    TUESDAY
    WEDNESDAY
    THURSDAY
    FRIDAY
    SATURDAY
    SUNDAY
}
```

Sistema de Tipos: Interfaces

GraphQL permite definir *Interfaces*.

```
interface Person {
    name: String!
    age: Int
}
type Director implements Person {
    name: String!
    age: Int
    isProducer: Boolean
}
type Actor implements Person {
    name: String!
    age: Int
    appearsIn: [Movie]
}
```

Sistema de Tipos: Interfaces

Podemos tener una query que consulte por una lista de personas:

```
type Query {
    allDirectorsAndActors: [Person]
}
```

Y consultar por información específica de un director:

```
query {
    allDirectorsAndActors {
        name
        ... on Director {
            isProducer
        }
    }
}
```

Resolvers

Pero cómo se implementa en el servidor?
Si repasamos los esquemas que hemos visto, cada field tiene asociado un tipo. Un servidor GraphQL tiene un resolver por cada field. Si consideramos a una query como una collección de fields, la tarea del servidor para responder a la petición es invocar cada uno de los resolvers correspondientes a los fields de dicha query.

Por ejemplo, en Python:

```
class Query(graphene.ObjectType):
    movies = graphene.List(MovieType)
    def resolve_movies(self, info, **kwargs):
        return get_movies_from_db()
```

Mutations

Hasta acá hemos visto cómo consultar información. Podemos además modificar el estado del servidor con *mutations*.
La sintaxis es similar:

```
mutation {
    addMovie(name: "Pulp Fiction", director:
        {fullName: "Quentin Tarantino"}) {
          name
          ok
    }
}
```

Podemos especificar qué campos retornar. Al igual que en una query, addMovie tiene un tipo asociado. Este tipo puede ser un tipo definido específicamente para esta mutation y los fields que retorna no tienen por qué guardar relación con los campos de entrada.

Mutations

```
type Mutation {
  addMovie(name: String, director:
      DirectorInput): AddMovieOutput
}

input DirectorInput {
  fullName: String
}

type AddMovieOutput {
  name: String
  ok: Boolean
}
```

Introspection

GraphQL permite a los clientes hacer consultas para conocer el esquema.

```
query {
    __schema{
        types {
            name
            kind
            fields {
              name
            }
        }
    }
}
```

Introspection

GraphQL permite a los clientes hacer consultas para conocer el esquema.

```
{   "__schema": {
        "types": [
            {
                "name": "Movie",
                "kind": "OBJECT",
                "fields": [
                  {"name": "name"},
                  {"name": "director"},
                  {"name": "releaseDate"},
                  ...]
            },
        ...]
    }
}
```

Introspection

Podemos consultar información de un tipo específico:

```
query {
    __type(name: "String"){
        name
        kind
    }
}
```

Resultado:

```
{
    "__type": {
        "name": "String",
        "kind": "SCALAR"
    }
}
```

Subscriptions

Mediante una *subscription* podemos suscribirnos a un evento, mantener una conexión abierta con el servidor y recibir notificaciones cuando dicho evento ocurra:

```
type Subscription {
    newMovie: Movie
}
```

```
subscription {
    newMovie {
        name
        releaseDate
    }
}
```

Ejercicios

Algunos ejercicios...

Implementando el server en Python

En Python, utilizamos la librería graphene para definir un esquema e implementar el servidor.

```
from graphene import ObjectType, String, Field,
    List, Int, Schema, Mutation

class Movie(ObjectType):
    name = String(required=True)
    duration_in_minutes = Int()
    director = Field(Director)
    actors = List(Actor)
    release_date = String()
```

Implementando el server en Python

Veamos la implementación de algunos *resolvers:*

```
class Query(ObjectType):
    movies = List(Movie)
    movie = Field(Movie, name=String(required=True))

    def resolve_movies(self, info):
        return get_movies_from_db()

    def resolve_movie(self, info, name):
        return get_movie(name) # instance of Movie
```

Implementando el server en Python

Veamos la implementación de una *mutation*:

```
class Mutation(ObjectType):
    add_movie = AddMovie.Field()

class AddMovie(Mutation):
    name = String()
    ok = Boolean()
    class Arguments:
        movie = MovieInput()

    def mutate(self, info, movie):
        insert_to_db(movie)
        return AddMovie(name=movie.name, ok=true)
```

Implementando el server en Python

Podemos retornar errores de la siguiente forma:

```
from graphql import GraphQLError

def mutate(self, info, movie):
    try:
        insert_to_db(movie)
        return AddMovie(name=movie.name, ok=true)
    except Exception as e:
        raise GraphQLError("Unable to insert movie")
```

En la respuesta veremos algo similar a este resultado:

```
{
  "errors": [
    {
      "message": "Unable to insert movie",
      "locations": [{"line": 2,
                     "column": 3}],
      "path": ["addMovie"]
    }
  ]
}
```

Implementando el server en Python

Podemos acceder a información contextual y metadata en un *resolver* mediante el argumento *info*. En este ejemplo consultamos el método HTTP empleado y los headers HTTP*:*

```
class Query(ObjectType):
    movies = List(Movie)

    def resolve_movies(self, info):
        method = info.context["request"].method
        print(method) # print "POST"
        headers = info.context["request"].headers
        print(headers['host']) # print the hostname
        return get_movies_from_db()
```

GraphiQL

GraphiQL es una herramienta de interfaz gráfica que permite conectarnos a un servidor durante la etapa de desarrollo y ejecutar queries y mutations. Podemos también consultar el esquema de una forma interactiva.

Repositorio

El código utilizado, junto a las slides, se encuentra en el repositorio:

- https://github.com/jcremona/graphql-tutorial