GraphQL

Introducción

Introducción

Limitaciones de las APIs REST

- Cada URL representa un recurso *completo*
  - No siempre necesitamos toda la información
- Recursos anidados se representan como enlaces
  - Recoger múltiples niveles de datos exige múltiples peticiones

Introducción

/character/1

Introducción

/character/1


```
{
    "id": "https://swapi.co/api/character/1/",
    "name": "Luke Skywalker",
    "height": "172",
    "mass": "77",
    "birth_year": "19BBY",
    "gender": "male",
    "homeworld": "https://swapi.co/api/planets/1/",
    "films": [
        "https://swapi.co/api/films/1/",
        "https://swapi.co/api/films/2/",
        "https://swapi.co/api/films/3/"
    ],
    "starships": [
      "https://swapi.co/api/starships/12/",
      "https://swapi.co/api/starships/22/"
    ]
}
```

Introducción

/character/1


```
{
    "id": "https://swapi.co/api/character/1/",
    "name": "Luke Skywalker",
    "height": "172",
    "mass": "77",
    "birth_year": "19BBY",
    "gender": "male",
    "homeworld": "https://swapi.co/api/planets/1/",
    "films": [
        "https://swapi.co/api/films/1/",
        "https://swapi.co/api/films/2/",
        "https://swapi.co/api/films/3/"
    ],
    "starships": [
      "https://swapi.co/api/starships/12/",
      "https://swapi.co/api/starships/22/"
    ]
}
```

Introducción

/character/1


```
{
    "id": "https://swapi.co/api/character/1/",
    "name": "Luke Skywalker",
    "height": "172",
    "mass": "77",
    "birth_year": "19BBY",
    "gender": "male",
    "homeworld": "https://swapi.co/api/planets/1/",
    "films": [
        "https://swapi.co/api/films/1/",
        "https://swapi.co/api/films/2/",
        "https://swapi.co/api/films/3/"
    ],
    "starships": [
      "https://swapi.co/api/starships/12/",
      "https://swapi.co/api/starships/22/"
    ]
}
```

Introducción

```
/character/1?fields=url,name

/character/1?include=films,starships

/character/1?include=films.id,films.title,
 starship.id,starship.name,...
```

Introducción

Para esto se inventa GraphQL

- Especificar exáctamente *los campos* que necesitamos
- Especificar qué *datos anidados* necesitamos
- Especificar los *campos de los datos anidados*
- ¡Tantos niveles como queramos!

(demo)

Conceptos Básicos

Conceptos Básicos

- Lenguaje para expresar *consultas de datos*
- Infraestructura para ejecutar las consultas
  - Independiente de base de datos
  - Independiente de lenguaje
- Múltiples implementaciones

Conceptos Básicos

Un servicio GraphQL se crea...

- Definiendo *tipos de datos* y sus campos
- Definiendo una *función para cada campo*

Conceptos Básicos

```
const { buildSchema, graphql } = require('graphql')

const typeDefinition = `
  type Query {
    date: String
  }
`

const schema = buildSchema(typeDefinition)
```

Conceptos Básicos

```
const { buildSchema, graphql } = require('graphql')

const typeDefinition = `
  type Query {
    date: String
  }
`

const schema = buildSchema(typeDefinition)
```

Conceptos Básicos

```
const { buildSchema, graphql } = require('graphql')

const typeDefinition = `
  type Query {
    date: String
  }
`

const schema = buildSchema(typeDefinition)
```

Conceptos Básicos

```
const { buildSchema, graphql } = require('graphql')

const typeDefinition = `
  type Query {
    date: String
  }
`

const schema = buildSchema(typeDefinition)
```

Conceptos Básicos

```
const typeDefinition = `
  type Query {
    date: String
  }
`

const schema = buildSchema(typeDefinition)

const root = {
  date: () => new Date().toJSON()
}
```

Conceptos Básicos

```
const typeDefinition = `
  type Query {
    date: String
  }
`

const schema = buildSchema(typeDefinition)

const root = {
  date: () => new Date().toJSON()
}
```

Conceptos Básicos

```
const typeDefinition = `
  type Query {
    date: String
  }
`

const schema = buildSchema(typeDefinition)

const root = {
  date: () => return new Date().toJSON()
}

const query = '{ date }'
```

Conceptos Básicos

```
const typeDefinition = `
type Query {
  date: String
}
`
const schema = buildSchema(typeDefinition)

const root = {
  date: () => return new Date().toJSON()
}

const query = '{ date }'

graphql(schema, query, root).then(console.log)
```

Conceptos Básicos

```
const typeDefinition = `
type Query {
  date: String
}
`

const schema = buildSchema(typeDefinition)

const root = {
  date: () => return new Date().toJSON()
}

const query = '{ date }'

graphql(schema, query, root).then(console.log)
```

Conceptos Básicos

Si ejecutamos el script...

```
{
  data: {
    date: '2018-11-20T11:02:00.545Z'
  }
}
```

Conceptos Básicos

GraphQL entiende 5 tipos de datos primitivos:

- `String`
- `Int`
- `Float`
- `Boolean`
- `ID`

Ejercicio

Escribe una API con GraphQL...

- Con un campo `dice` de tipo entero
- Que devuelve números aleatorios entre 1 y 6
- **TDD**

Conceptos Básicos

Podemos pasar *parámetros* a una API GraphQL

- Cada parámetro tiene *tipo* y *nombre*
- Comprobación de tipos automática
- Pueden ser *obligatorios* u *opcionales*

Conceptos Básicos

Definimos los parámetros en la declaración de tipos

```
type Query {
  hello(name: String): String
}
```

Conceptos Básicos

Recibimos los parámetros en el resolver

```
const root = {
  dice: (args) => {
    const { name } = args
    return `Hello, ${name}`
  }
}
```

Conceptos Básicos

Les damos valor en la query

```
{
  hello(name: "World")
}
```

```
const typeDefinition = `
  type Query {
    hello(name: String): String
  }
`

const schema = buildSchema(typeDefinition)

const root = {
  hello: (args) => {
    const { name } = args
    return `Hello, ${name}`
  }
}

const query = '{ hello(name: "World") }'

graphql(schema, query, root)
  .then(console.log)
```

Conceptos Básicos

GraphQL ofrece dos *modificadores de tipo*

- `[]` para *listas*
  - `[Int]`
- `!` para indicar que *no puede ser nulo*
  - `!String`

Ejercicio

Modifica el código del ejercicio anterior...

- Para que el campo `dice` reciba dos parámetros:
  - `numDice`, de tipo entero
  - `numSides`, de tipo entero
- Y devuelva un *array de números*
  - `numDice` elementos
  - cada elemento con un número aleatorio entre 1 y `numSides`
- **TDD**

Tipos

Tipos

GraphQL nos permite definir *nuestros propios tipos*

- Un conjunto de *campos*
- Cada capo puede recibir *parámetros*
- Representados con *objetos*

Tipos

```
type RandomDie {
  roll(numRolls: Int!): [Int]
}

type Query {
  dice(numSides: Int!): RandomDie
}
```

Tipos

La *query* selecciona los campos que necesita

```
{
  dice(numSides: 6) {
    roll(numRolls: 3)
  }
}
```

Tipos

La *respuesta* devuelve sólo los campos indicados

```
{
  "dice": {
    "roll": [1, 5, 2]
  }
}
```

Tipos

En el resolver, un campo de tipo complejo...

- Tiene que devolver un *objeto*...
- ...con un *método* para cada *campo* del tipo

Tipos

```
const random = n => Math.floor(Math.random() * n) + 1

const root = {
  dice({ numSides }) {
    return {
      roll({ numRolls }) {
        return new Array.from({ length: numRolls }).map(() => random(numSides))
      }
    }
  }
}
```

Tipos

La manera recomendada de implementar el resolver es:

- Definir una clase para cada tipo que hayamos creado nosotros
- Con un método para cada campo del tipo
- Devolver un valor de ese tipo ⇒ devolver una instancia de la clase

Tipos

```
const random = n => Math.floor(Math.random() * n) + 1

class RandomDie {
  constructor(numSides) {
    this.numSides = numSides
  }

  roll({ numRolls }) {
    return new Array
        .from({ length: numRolls })
        .map(() => random(this.numSides))
  }
}

const root = {
  dice({ numSides }) {
    return new RandomDie(numSides)
  }
}
```

Ejercicio

Incluye el tipo `RandomDie` en el código del ejercicio anterior

- Modifica `dice` para que el parámetro `numSides` sea opcional
  - valor por defecto: 6
- Añade un campo `bestOf` a `RandomDie` que...
  - reciba un parámetro `numRolls`, de tipo entero
  - lanza el dado `numRolls` veces y devuelve la mejor
- **TDD**

Ejercicio

Utilizando datos aleatorios, implementa (TDD) el resolver para esta API:

```
type Player {
  name: String!,
  number: Int!
}

type Team {
  name: String!,
  points: Int!,
  matchesPlayed: Int!,
  players: [Player]!
}

type Query {
  teams(nameFilter: String): [Team]!
}
```

Mutaciones

Mutaciones

Para definir una *inserción* o una *modificación*

- Definimos un campo en el tipo **`Mutation`**
- Igual que los campos de **`Query`**
- *Buena práctica*: nombrarlo con **`set`**, **`update`** o **`create`**

Mutaciones

```
type Mutation {
  setDate(unixTime: Int!): String
}

type Query {
  date: String
}
```

Mutaciones

Las *mutaciones* se implementan en el *resolver*

```
let time = new Date()

const root = {
  setDate: ({ unixTime }) => {
    time = new Date(unixTime * 1000)
    return root.date()
  },
  date: () => time.toString().slice(0, 15)
}
```

Mutaciones

Para *ejecutar* una mutación

- Indicamos que la consulta es de tipo `mutation`
- Especificamos qué operaciones queremos aplicar

Mutaciones

```
const stamp = 1533232122
const result = graphql(
  schema,
  `mutation { setDate(unixTime: ${stamp}) }`,
  root
)
result.then(console.log)
```

Mutaciones

```
{ data: { setDate: 'Thu Aug 02 2018' } }
```

Ejercicio

Implementa una *lista de la compra* con una API que permita:

- Añadir elementos (nombre, cantidad)
- Modificar elementos (por `ID`)
- Eliminar elementos
- Marcar elementos como "comprados"
- Listar los elementos añadidos
  - filtrar por: nombre, comprado

Mutaciones

A veces tenemos mutaciones que reciben parámetros similares

```
type Mutation {

  createItem(
    name: String!, category: String!, quantity: Int!
  ): [Item]

  updateItem(
    id: ID!, name: String!, category: String!, quantity: Int!
  ): [Item]

}
```

Mutaciones

Definir un *input type* nos permite factorizar

```
input ItemInput {
  name: String!
  category: String!
  quantity: Int!
}

type Mutation {
  createItem(item: ItemInput): [Item]
  updateItem(id: ID!, item: ItemInput): [Item]
}
```

Mutaciones

Definir un *input type* nos permite factorizar

```
mutation {
  createItem(item: {
    name: "huevos",
    category: "food",
    quantity: 12
  })
}
```

Mutaciones

Un *input type* es un conjunto de campos escalares

- Sólo para parámetros (de *mutations* o *queries*)
- Sólo pueden contener *valores escalares*
- Se representan como *objetos*
- *Buena práctica*: nombrar el tipo con el sufijo `Input`

Errores

Errores

GraphQL captura los *errores* automáticamente

- En caso de error, la respuesta contiene...
  - un campo `errors`
  - el `resolver` que ha fallado aparece con valor `null`

Errores

```
const root = {
  date: ({ unixTime }) => {
    throw new Error('Oh, noes!')
    return new Date().toString().slice(0, 15)
  }
}
```

Errores

```
await graphql(schema, '{ date }', root)
```

Errores

```
{
  "errors": [{
    "message": "Oh, noes!",
    "path": ["date"],
    "locations": [{
      "line": 1,
      "column": 3
    }]
  }],
  "data": { "date": null }
}
```

express-graphql

express-graphql

Nos permite *exponer* nuestra API GraphQL

- En una ruta *HTTP*
- Se encarga de *recibir*, *parsear* y *ejecutar* operaciones
- Configuramos el endpoint con un *schema* y un *resolver*
- Convive con otros *middlewares* de express

express-graphql

```
const { buildSchema, graphql } = require('graphql')
const graphqlHTTP = require('express-graphql')
const app = require('express')()

// graphql
const typeDefinition = `...`
const schema = buildSchema(typeDefinition)

const root = { /* ... */ }

// express
app.use('/graphql', graphqlHTTP({ schema, rootValue: root }))
app.listen(3000)
```

express-graphql

Podemos hacer peticiones contra */graphql*

```
$ curl "http://localhost:3000/graphql?query=%7Bdate%7D"

{ "data": { "date":"Wed Dec 05 2018" } }
```

express-graphql

Podemos hacer peticiones contra */graphql*

```
$ (
  export MUTATION="mutation%20%7BsetDate(unixTime:0)%7D";
  curl -X POST "http://localhost:3000/graphql" \
       -d "query=$MUTATION"
)

{ "data": { "setDate":"Thu Jan 01 1970" } }
```

express-graphql

Nos permite *exponer* nuestra API GraphQL

- Query urlencoded en el parámetro *query*
- Consultas con *GET*
- Mutaciones y consultas con *POST*

Ejercicio

Añade `express-graphql` a la API del ejercicio antedrior:

- Endpoint: `/graphql`
- Prueba a hacer consultas con `GET`
- Prueba a hacer modificaciones con `POST`

express-graphql

express-graphql **tiene** graphiql **integrado:**

```
app.use('/graphql', graphqlHTTP({
  schema,
  rootValue: root,
  graphiql: true
}))
```

express-graphql

Podemos añadir *documentación* a nuestro esquema

```
const typeDefinition = `
  """
  My great API
  """
  type Query {
    "Returns the active date"
    date: String
  }
`
```

express-graphql

Podemos añadir *documentación* a nuestro esquema

```
const typeDefinition = `
  type Mutation {
    "Modifies the active date"
    setDate(
      "unixTime: date in unix format"
      unixTime: Int!

    ): String
  }
`
```

Ejercicio

Documenta la API del ejercicio anterior

- Comprueba que aperece en GraphiQL (Docs)
- Comprueba que aparece en el "intellisense"

express-graphql

graphqlHTTP **no es más que otro** *middleware*

```
app.use(
  '/graphql',
  (req, res, next) => {
    console.log('Something running before!')
    next()
  },
  graphqlHTTP({
    schema, rootValue: root, graphiql: true
  })
)
```

Ejercicio

Protege tu API con un login (sesión)

- Sólo los usuarios autenticados pueden hacer queries
- Crea un usuario "a mano" en el código
- Pantalla de **login** para crear sesión
- Usuario *NO* autenticado = redirect a **login**
- ¿Se podría implementar el login como una mutacion GraphQL?

express-graphql

Resolvers reciben la *request* como segundo parámetro

```
const root = {
  date: (args, req) => {
    const path = req.baseUrl
    const date = time.toString().slice(0, 15)
    return `${date} (${path})`
  }
}
```

Ejercicio

Extiende tu API para que cada usuario tenga su propia lista

- Registro de nuevos usuarios (en memoria)
- Cada usuario sólo puede ver su propia lista
- Cada usuario sólo puede modificar los items de su lista

Ejercicio

Crea usuarios *administradores*

- Un flag en el objeto `user`
- Los administradores pueden listar, crear y borrar usuarios
- Los administradores pueden ver las lista de cualquier usuario
- Modifica el esquema para reflejar estas operaciones

express-graphql

Un resolver es *asíncrono* si devuelve una promesa

```
const root = {
  date: () => {
    return new Promise(resolve => setTimeout(
      () => resolve(time.toString().slice(0, 15)),
      1000
    ))
  }
}
```

Ejercicio

Mueve la persistencia de tu API a `mysql`

- Utilizando la librería `mysql` del tema anterior

Paginación

Paginación

La solución más *simple* es añadir *un parámetro*.

```
type Item {
  name: String
}

type List {
  items(page: Int = 1): [Item]!
}

type Query {
  list: List
}
```

Ejercicio

Añade paginación a tu API utilizando un parámetro `page`

- Inicializa tu bbdd con items aleatorios
- Extra: recibe otro parámetro `pageCount`

Paginación

Pero: estamos mezclando *paginación* **con** *lógica de negocio*

- Estamos acoplando...
  - detalles de los datos
  - con detalles de la comunicación
- Nos gustaría desacoplarlo

Paginación

Tipo específico para encapsular la paginación

```
type Item {
  name: String
}

type ItemPage {
  page(num: Int = 1, length: Int = 10): [Item]!
}

type List {
  items: ItemPage!
}

type Query {
  list: List
}
```

Ejercicio

Implementa paginación con un tipo intermedio como `ItemPage`

- Utiliza el esquema del ejemplo anterior como referencia
- Extra: ¿Cómo podríamos generalizar la implementación?

Paginación

Para implementar la *solución definitiva* necesitamos...

- Poder definir tipos dinámicamente
  - En lugar de definir tipos como `ItemPage`...
  - ...queremos `PaginatedResults(Item)`
- ¡Sin tener que manipular strings!

Paginación

GraphQL **tiene un** *mecanismo alternativo* **para definir tipos:**

- Definir los tipos mediante objetos *Javascript*
  - En lugar de usar strings
- Expresividad equivalente

Paginación

```
// type Item {
//    name: String
// }

const ItemType = new GraphQLObjectType({
  name: 'Item',
  fields: {
    name: { type: GraphQLString }
  }
})
```

Paginación

```
// type List {
//    items: [Item]!
// }

const ListType = new GraphQLObjectType({
  name: 'List',
  fields: {
    items: { type: GraphQLList(ItemType) }
  }
})
```

Paginación

```
// type Query {
//   list: List
// }

// const schema = buildSchema(typeDefinition)

const QueryType = new GraphQLObjectType({
    name: 'Query',
    fields: {
      list: { type: ListType }
    }
  })

const schema = new GraphQLSchema({
  query: QueryType
})
```

Paginación

```
const ItemType = new GraphQLObjectType({
  name: 'Item',
  fields: {
    name: { type: GraphQLString }
  }
})

const ListType = new GraphQLObjectType({
  name: 'List',
  fields: {
    items: { type: GraphQLList(ItemType) }
  }
})

const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'Query',
    fields: {
      list: { type: ListType }
    }
  })
})
```

Paginación

Los tipos en *código* son fáciles de *manipular*

- ¡No son más que *objetos*!
- Podemos generarlos dinámicamente
- Utilizando factorías

Paginación

Así es como habíamos definido `ItemPage`:

```
type ItemPage {
  page(num: Int = 1, length: Int = 10): [Item]!
}
```

Paginación

```
const ItemPageType = new GraphQLObjectType({
  name: 'ItemPage',
  fields: {
    page: {
      type: new GraphQLList(ItemType),
      args: {
        num: { type: GraphQLInt, defaultValue: 1 },
        length: { type: GraphQLInt, defaultValue: 10 }
      }
    }
  }
})
```

Paginación

```
const PaginatedResultType = Type => GraphQLObjectType({
  name: `${Type.name}Page`,
  fields: {
    page: {
      type: new GraphQLList(Type),
      args: {
        num: { type: GraphQLInt, defaultValue: 1 },
        length: { type: GraphQLInt, defaultValue: 10 }
      }
    }
  }
})

const ItemPageType = new PaginatedResultType(ItemType)
```

Ejercicio

Traduce tu API a notación de objetos e implementa...

- Un tipo dinámico de resultados paginados (factoría)
- Un *resolver* **genérico** para gestionar paginación
  - Una **única clase** para resolver todos los tipos dinámicos
  - Recibe como **parámetros** la info necesaria

Conceptos Avanzados

Conceptos Avanzados

Una API más *compleja*

- Data muy interconectada
- ¡Grafo!
- Perfecta para *Graph*QL

Conceptos Avanzados

(demo)

Conceptos Avanzados

¿Qué pasa si queremos pedir *dos* `starships`?

```
{
  starship(id: "starships/1") {
    name
    pilots {
      name
    }
  }
}
```

Conceptos Avanzados

Alias: **varias consultas a un mismo campo**

- *Desacoplar* nombre del campo y del resultado
- Pedir *múltiples valores* de un mismo campo
- Útil, sobre todo, para campos que reciben *parámetros*

Conceptos Avanzados

```
{
  falcon: starship(id: "starships/1") {
    name
    pilots {
      name
    }
  }
  xwing: starship(id: "starships/12") {
    name
    pilots {
      name
    }
  }
}
```

Conceptos Avanzados

Fragments: **conjuntos de campos con nombre propio**

- Evita *duplicar* los campos en peticiones similares
- Nombre propio
- Asociados a *un tipo*. No aplicable a otros tipos.

Conceptos Avanzados

```
{
  falcon: starship(id: "starships/1") {
    ...shipSummaryFields
  }
  xwing: starship(id: "starships/12") {
    ...shipSummaryFields
  }
}

fragment shipSummaryFields on Starship {
  name
  pilots {
    name
  }
}
```

Conceptos Avanzados

Nombres de operación

- Da *nombre propio* a la operación
- No es necesario (excepto en casos puntuales)
- *Muy recomendable*. Facilita depuración y logs.

Conceptos Avanzados

```
query GetStarships {
  falcon: starship(id: "starships/1") {
    ...shipSummaryFields
  }
  xwing: starship(id: "starships/12") {
    ...shipSummaryFields
  }
}

fragment shipSummaryFields on Starship {
  name
  pilots {
    name
  }
}
```

Conceptos Avanzados

```
query GetStarships {
  falcon: starship(id: "starships/1") {
    ...shipSummaryFields
  }
  xwing: starship(id: "starships/12") {
    ...shipSummaryFields
  }
}

fragment shipSummaryFields on Starship {
  name
  pilots {
    name
  }
}
```

Conceptos Avanzados

Variables

- Se definen tras el nombre de la operación
- Empiezan con *$*
- Se especifican en un *objeto separado* de la query
- Aplicables también a *mutations*

Conceptos Avanzados

```
query GetStarships ($A: String, $B: String) {
  falcon: starship(id: $A) {
    ...shipSummaryFields
  }
  xwing: starship(id: $B) {
    ...shipSummaryFields
  }
}

fragment shipSummaryFields on Starship {
  name
  pilots {
    name
  }
}
```

Conceptos Avanzados

Directivas

- Configurar dinámicamente la query
- Mediante variables
- Sólo dos en la especificación:
  - *@include(if: Boolean)*
  - *@skip(if: Boolean)*

Conceptos Avanzados

```
query GetStarships ($A: String, $B: String, $withPilots: Boolean = false) {
  falcon: starship(id: $A) {
    ...shipSummaryFields
  }
  xwing: starship(id: $B) {
    ...shipSummaryFields
  }
}

fragment shipSummaryFields on Starship {
  name
  pilots @include(if: $withPilots) {
    name
  }
}
```

Conceptos Avanzados

Tipos compuestos:

- *Interfaces* abstraer
- *Unions* para combinar

Conceptos Avanzados

```
interface Character {
  id: ID!
  name: String!
  friends: [Character]
  appearsIn: [Episode]!
}
```

Conceptos Avanzados

```
type Human implements Character {
  id: ID!
  name: String!
  friends: [Character]
  appearsIn: [Episode]!
  starships: [Starship]
  totalCredits: Int
}
```

Conceptos Avanzados

```
type Droid implements Character {
  id: ID!
  name: String!
  friends: [Character]
  appearsIn: [Episode]!
  primaryFunction: String
}
```

Conceptos Avanzados

```
union SearchResult = Human | Droid
```

Conceptos Avanzados

```
{
  search(text: "an") {
    ... on Human {
      name
      height
    }
    ... on Droid {
      name
      primaryFunction
    }
  }
}
```