

Taller de GraphQL, Apollo Server y Prisma

[Introducción \(Primera Parte\)](#)

[Básico](#)

[Ejecuciones en GraphQL Server Express](#)

[Clientes GraphQL](#)

[Ejemplo Pasando Argumento](#)

[Pasando valores a los parámetros](#)

[Pasando variables desde consola](#)

[Usando variables desde playground](#)

[Tipos de objetos](#)

[Mutations Inputs y Tipos](#)

[Middleware](#)

[Javascript - Apollo - Prisma \(Segunda Parte\)](#)

[Intro](#)

[Crear proyecto, primeras queries](#)

[Sobre GraphQL schema](#)

[Una Query Simple](#)

[Ejemplo](#)

[Resolver de tipo: Agregando un elemento calculado](#)

[Código del ejemplo:](#)

[Proceso de resolución de las queries](#)

[Una Mutation Simple](#)

[Extendiendo la definición del schema](#)

[Creamos un archivo para las definiciones de schema graphql:](#)

[Código:](#)

[Probando la mutation](#)

[Ejercicio](#)

[Código Ejercicio](#)

[Agregando una base de datos con Prisma](#)

[Prisma](#)

[Configuración](#)

[Configurar la base SQLite](#)

[Migration: Establecer los esquemas en la base de datos DDL](#)

[Generar el Prisma Client](#)

[Conectando el Server y la Database con Prisma Client](#)

[El argumento de resolver GraphQL context](#)

[Actualizar las functions resolver para poder usar Prisma Client](#)

[Usar Prisma Studio](#)

[Autenticación](#)

[Agregando un User al model](#)

[Generar nueva migration y Actualizar el prisma client](#)

[Modificar nuestro schema GraphQL](#)

[Implementar los resolvers](#)

[Archivo Query.js](#)

[Archivo Mutation.js \(explicadas las nuevas mutations\)](#)

[Archivo utils.js](#)

[Archivo index.js](#)

[Nota sobre Autenticaciones, roles y permisos. graphql-shield](#)

[Pruebas](#)

[Subscriptions en GraphQL](#)

[Implementar subscriptions GraphQL](#)

[Subscribir nuevos elementos Link](#)

[Añadir las subscriptions a los resolvers de las mutations](#)

[1- Agregar el publish del evento en la mutation](#)

[2- No olvidar incorporar los resolvers de subscriptions \(el archivo de Subscription\) a nuestro index.js](#)

[Probar las subscriptions](#)

[1- Ir al Playground, abrir un nuevo tab y subscribirse al event](#)

[2- Crear un nuevo link mediante la mutation post](#)

[Agregando la feature de votación](#)

[1- Actualizar nuestro esquema de base según la nueva necesidad \(schema.prisma\)](#)

[2- Actualizar esquema de la base y prismaClient](#)

[3- Modificar el schema.graphql. \(Nueva mutation y modificar types\)](#)

[4- Implementar las functions resolver](#)

[a. Resolver de la mutation](#)

[b. Resolvers de tipo](#)

[c. Agregar los nuevos files de Resolver al index](#)

[5- Probar la las mutations y subscriptions](#)

[Filtrado, Paginación y Ordenamiento](#)

[Filtrado](#)

[Paginación](#)

[Limit-Offset](#)

[Cursor-based](#)

[Ordenamiento \(sorting\)](#)

[Retornar la cantidad total de elementos en la query paginadas](#)

[Enlaces](#)

Introducción (Primera Parte)

Básico

<https://graphql.org/graphql-js/>

- GraphQL establece un modo de comunicación entre server y cliente diseñado para ser escalable.
- Permite representación de los datos de una manera natural (un [grafo](#))
- Las comunicaciones con GraphQL se hace a través de:
 - Mutations: Requests que implican una alteración de datos, o un procesamiento y modificación. Si hacemos una analogía con APIs rest, los verbos POST / PUT / PATCH / DELETE en GQL se resuelven con Mutations
 - Queries: Requests de consultas a la app, que generalmente se resuelven mediante consultas a base, archivos, cálculo, u otro (no se persisten cambios). Sería el concepto de GET.
 - Subscriptions: Similar generalmente implementado con webSocket, similar a Signal-R, etc. Permite al server enviar datos a sus clientes cuando un evento determinado ocurre.
- Mutations y queries de GQL se enrutan a un **único POST** que entra por / . Generalmente esto es transparente.
- Las mutations y queries son resueltas por functions denominadas “resolvers”. (Son controladores).
- Se indica al server qué se necesita en el response, según el Schema disponible, pudiendo traer objetos anidados, arrays, con distinto nivel de complejidad según lo disponga el server y el schema.

Ejemplo simple usando librería **graphql**

server.js

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

// Esta sección construye un schema en la sintaxis GraphQL
//En este ejemplo estamos creando un schema de un tipo de datos Query que tiene una funcionalidad
saludo que retorna un String. Básicamente indicamos que "La query saludo, retorna un String"
const schema = buildSchema(`
  type Query {
    saludo: String
  }
`);

// El objeto rootValue provee una función "resolver" para cada endpoint de la API
const rootValue = {
  saludo: () => {
```

```

    return 'Bueeenas, cómo andan..';
  },
};

// Ejecuta la query '{ hello }' e imprime el response. La llamada a la misma está definida por el
resolver hello()

graphql({
  schema,
  source: '{ saludo }',
  rootValue
}).then((response) => {
  console.log(response);
});

```

si instalamos y ejecutamos

```
npm install graphql --save
```

node server.js

Obtendremos:

```
{ data: [Object: null prototype] { saludo: 'Bueeenas, cómo andan..' } }
```

Ejecuciones en GraphQL Server Express

Instalamos las dependencias de express-graphql y graphql

```
npm install express express-graphql graphql --save
```

Modificamos nuestro código fuente un poco, levantando express-graphql

server02.js

```

const express = require('express');
const { graphqlHTTP } = require('express-graphql');
const { buildSchema } = require('graphql');

// Construimos un schema con la sintaxis de GraphQL
const schema = buildSchema(`
  type Query {
    saludo: String
  }
`);

// El root provee un resolver
const root = {
  saludo: () => {
    return 'Bueeenas, cómo andan..';
  },
};

// Levantamos el cliente en http://localhost:4000/graphql

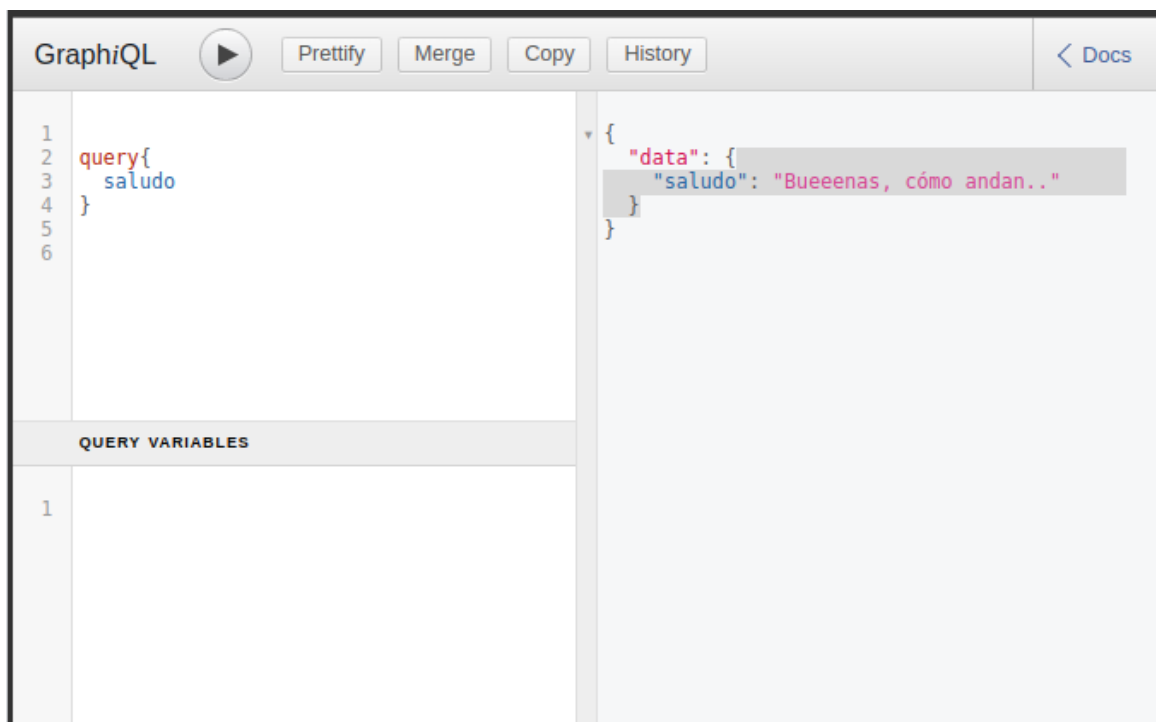
```

```
const app = express();
app.use('/graphql', graphqlHTTP({
  schema: schema,
  rootValue: root,
  graphiql: true,
}));
app.listen(4000);
console.log('Running a GraphQL API server at http://localhost:4000/graphql');
```

Siempre que hayamos configurado **graphiql: true**, podemos entrar a la herramienta en **http://localhost:4000/graphql**

Veremos una interfaz de prueba muy básica provista por la librería express-graphql, hay interfaces muy más completas (que veremos más adelante) que tiene algunas secciones simples, pero ya muy potentes:

- Sección de **queries y mutations**: Basándose en los Schemas actuales es posible ejecutar queries y mutations
- Sección de **Responses**
- **Docs**: La documentación de los Schemas es generada automáticamente, y cada vez que se modifica el Schema se modifica la documentación. Tiene información de Schemas de las Queries, Mutations, Tipos, Inputs, Enums, entre otros, existentes en nuestra App.
- Variables de queries. Pueden indicarse aquí valores para parámetros en las queries



Cientes GraphQL

GraphQL tiene una estructura más robusta que una REST API, existen clientes más potentes los cuales pueden manejar de manera eficiente lotes y capturas de datos entre otros. No es necesario un cliente complejo para hacer llamada GraphQL. Con express-graphql, podríamos enviar sólo un request al endpoint POST sobre el que se monta el server GraphQL, pasando la query como un field de **query** en un payload JSON

Si hacemos

```
curl -X POST \
-H "Content-Type: application/json" \
-d '{"query": "{ saludo }"}' \
http://localhost:4000/graphql
```

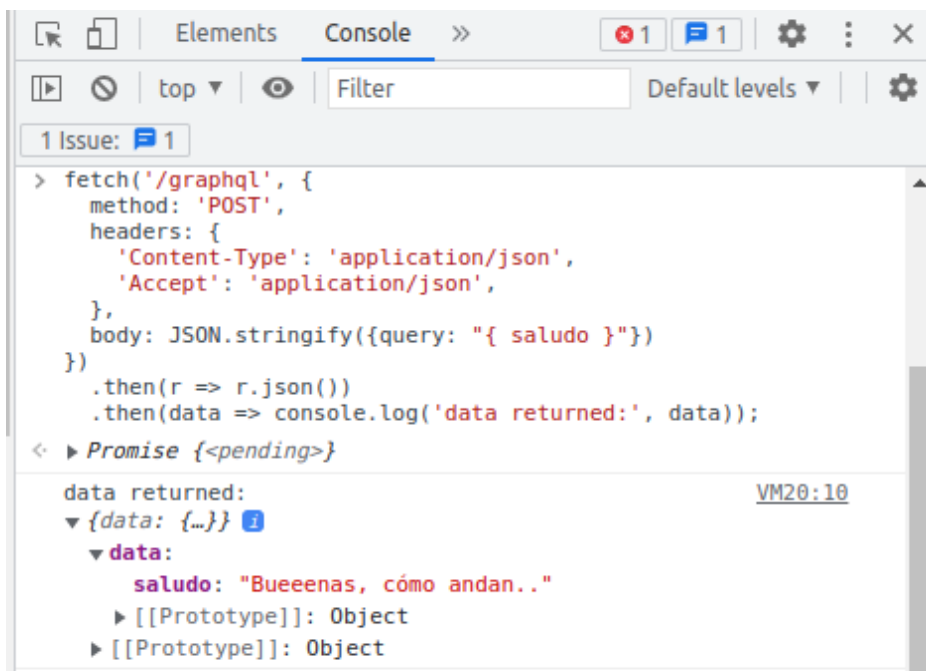
Obtendremos como respuesta el siguiente JSON

```
{"data":{"saludo":"Bueeenas, cómo andan.."}}}
```

También es posible usar clientes como:

- GraphiQL
- Insomnia
- Altair
- Playground
- Postman
- Clientes javascript, librerías node como apollo-client
- Explorador:
 - Si vamos a la url <http://localhost:4000/graphql>,
 - Abrimos consola de desarrollador
 - Hacemos el siguiente fetch:

```
fetch('/graphql', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    'Accept': 'application/json',
  },
  body: JSON.stringify({query: "{ saludo }"})
})
  .then(r => r.json())
  .then(data => console.log('data returned:', data));
```



Ejemplo Pasando Argumento

<https://graphql.org/graphql-js/passing-arguments/>

Tomamos el ejemplo de server del link

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

// Construct a schema, using GraphQL schema language
const schema = buildSchema(`
  type Query {
    rollDice(numDice: Int!, numSides: Int): [Int]
  }
`);

// The root provides a resolver function for each API endpoint
const root = {
  rollDice: ({numDice, numSides}) => {
    const output = [];
    for (let i = 0; i < numDice; i++) {
      output.push(1 + Math.floor(Math.random() * (numSides || 6)));
    }
    return output;
  }
};

const app = express();
app.use('/graphql', graphqlHTTP({
  schema: schema,
  rootValue: root,
  graphiql: true,
}));
app.listen(4000);
console.log('Running a GraphQL API server at localhost:4000/graphql');
```

Notemos que el schema de Queries declara una query rollDice, dicha query recibe dos enteros como parámetros, y retorna una array de Int

```
type Query {
  rollDice(numDice: Int!, numSides: Int): [Int]
}
```

Observaciones:

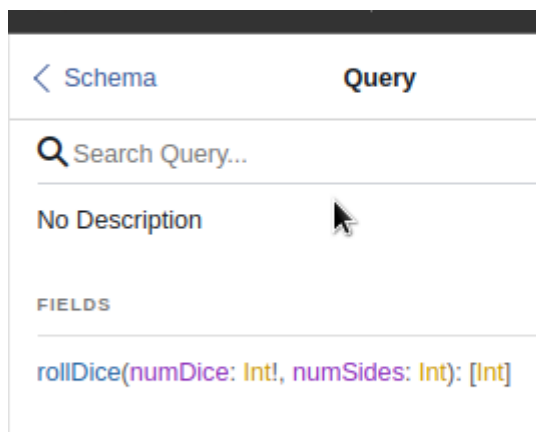
- El único parámetro que debe ser numérico y es obligatorio es numDice, esto se indica con el signo de admiración luego del tipo !
- numSides podría ir como null, o no ir en la query, el response podría devolver null o un arreglo vacío.
 - Si quisiéramos validar los responses y en el proceso dejar documentado haber puesto:

- [Int!] → Indica que la query debe retornar un Array con 0 o más elementos de tipo Int
- [Int!]! → Indica que la query debe retornar un Array 1 o más elementos Int (el array debe tener elementos)

Hay varias formas de consumir queries con parámetros desde el cliente:

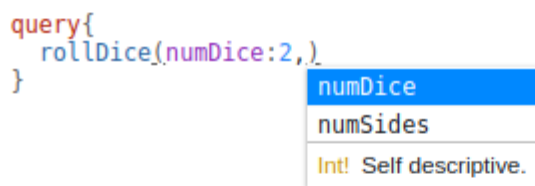
- Enviando los valores directamente a los parámetros
- Estableciendo valores en variables

Desde la interfaz Gráfica simple, ya podemos analizar la documentación
Ya en la parte de Docs, podemos ver la estructura de la query



La interfaz a medida que escribimos la query nos muestra qué posibilidades para hacer el request tenemos, y cómo debemos realizar la proyección de la misma (similar a un intelliSense)

Pasando valores a los parámetros



Ejemplo de response



Pasando variables desde consola

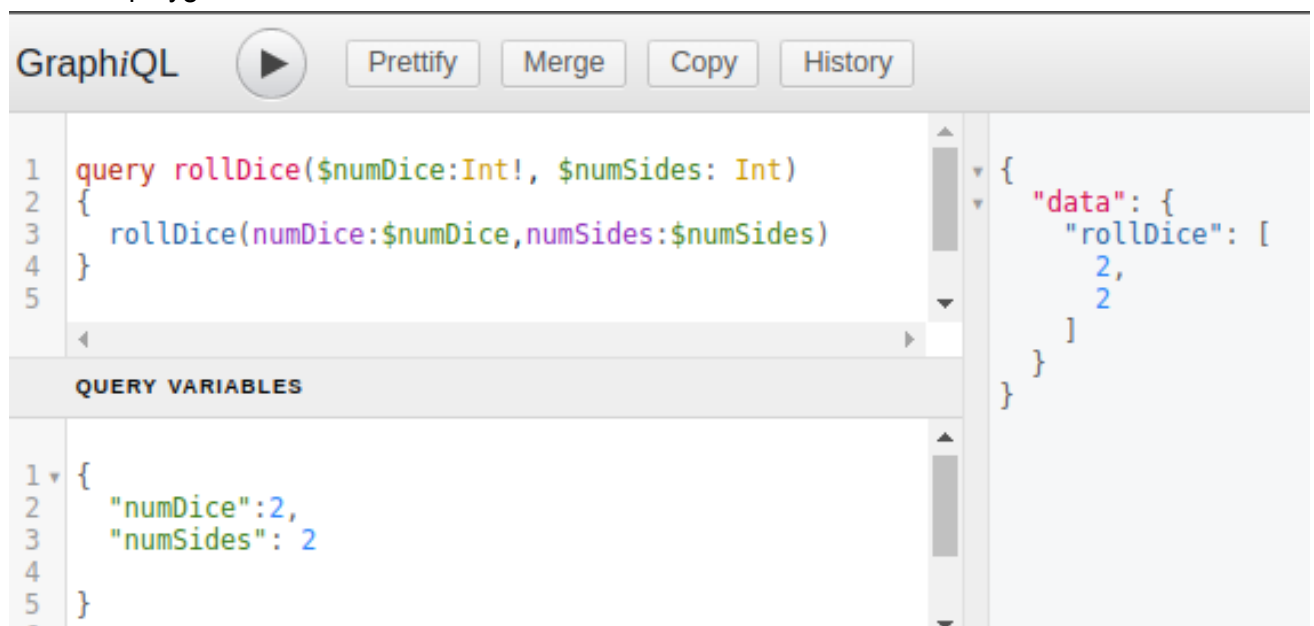
Es posible hacer la query graphql desde la **consola del explorador**:

```
var dice = 3;
var sides = 6;
var query = `query RollDice($dice: Int!, $sides: Int) {
  rollDice(numDice: $dice, numSides: $sides)
}`;

fetch('/graphql', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    'Accept': 'application/json',
  },
  body: JSON.stringify({
    query,
    variables: { dice, sides },
  })
})
.then(r => r.json())
.then(data => console.log('data returned:', data));
```

Usando variables desde playground

Desde el playground:



Tipos de objetos

<https://graphql.org/graphql-js/object-types/>

Es posible definir en nuestro schema distintos tipos de objetos, basados en tipos nativos soportados por GraphQL, esto es tanto para los parámetros de entrada de las Queries y Mutations (input) como para definir qué tipo de dato será el retornado en cada response de cada query o mutation.

En gql la manera en la que definimos tipos es la misma manera en la que se define el tipo query

```
type Query {  
  rollDice(numDice: Int!, numSides: Int): [Int]  
}
```

Por ejemplo podríamos implementar un objeto del tipo RandomDie que represente un dado

```
type RandomDie {  
  numSides: Int!  
  rollOnce: Int!  
  roll(numRolls: Int!): [Int]  
}  
  
type Query {  
  getDie(numSides: Int): RandomDie  
}
```

```
var express = require('express');  
var { graphqlHTTP } = require('express-graphql');  
var { buildSchema } = require('graphql');  
  
// Construct a schema, using GraphQL schema language  
var schema = buildSchema(`  
  type RandomDie {  
    numSides: Int!  
    rollOnce: Int!  
    roll(numRolls: Int!): [Int]  
  }  
  
  type Query {  
    getDie(numSides: Int): RandomDie  
  }  
`);  
  
// This class implements the RandomDie GraphQL type  
class RandomDie {  
  constructor(numSides) {  
    this.numSides = numSides;  
  }  
}
```

```

rollOnce() {
  return 1 + Math.floor(Math.random() * this.numSides);
}

roll({numRolls}) {
  var output = [];
  for (var i = 0; i < numRolls; i++) {
    output.push(this.rollOnce());
  }
  return output;
}
}

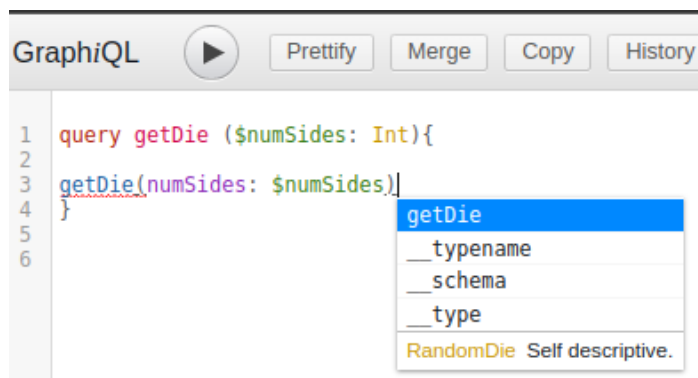
// The root provides the top-level API endpoints
var root = {
  getDie: ({numSides}) => {
    return new RandomDie(numSides || 6);
  }
}

var app = express();
app.use('/graphql', graphqlHTTP({
  schema: schema,
  rootValue: root,
  graphiql: true,
}));
app.listen(4000);
console.log('Running a GraphQL API server at localhost:4000/graphql');

```

La query se ejecuta de la misma manera. La definición de tipos importante para poder definir parámetros y respuestas más complejos, pudiendo ser reutilizado por otras mutations y queries, sin necesidad de redefinir schemas.

La documentación nos indica que la query retorna un RandomDie, por lo que del tipo RandomDie, como consumidor de la query, podemos proyectar sólo los datos que requerimos en el response.



:



Otro ejemplo podemos proyectar invocando a las subqueries



El código de cada subquery se ejecutará sólo si está en la proyección

Por lo tanto si sólo queremos conocer un field del (o de los) objetos retornados, proyectamos sólo lo necesario. Así mismo un solo field podría ser de otro tipo definido y es posible proyectar de manera anidada sobre el mismo (similar a Mongo).

Por ejemplo:

```
type Type1{
  field1: String!
  field2: Int
  address: String
}

type Type2 {
  id: Int!
  field1: String
  field2: String
  field3: String
  field4: String
  objsType1: [Type1!]
}
```

Teniendo definida la siguiente query:

```
type Query {
  getType2: Type2
}
```

En caso de requerir consultar sólo los address por cada id de type2

```
query ejemplo{
  getType2:{
    id
    objType1: {
      address
    }
  }
}
```

Nota: Para esto es necesario que el código retorne el detalle anidado deseado, es posible que se reutilicen tipos pero no se aniden en su totalidad, dependiendo de la complejidad. Hay casos en los que dependiendo de la query necesaria y lo que se requiere buscar, el back no busque y ponga a disposición todo el anidamiento.

Mutations Inputs y Tipos

<https://graphql.org/graphql-js/mutations-and-input-types/>

Si se requiere un endpoint para realizar cambios persistentes en base de datos, ya sea insertar, actualizar o eliminar cualquier información, se deben usar **Mutations** (no query), es posible hacer una analogía de **Mutations** con los verbos *POST / PUT / PATCH / DELETE* y **Queries** con *GET*

```
type Mutation {
  setMessage(message: String): String
}
type Query {
  getMessage: String
}
```

Lo único que hace falta es implementar los “resolvers” (los resolvers son functions que resuelven la query o mutation)

```
var fakeDatabase = {};
var root = {
  setMessage: ({message}) => {
    fakeDatabase.message = message;
    return message;
  },
  getMessage: () => {
    return fakeDatabase.message;
  }
};
```

Si se requiere complejizar los parámetros de las mutations en algún esquema específico, una forma de simplificar y reutilizar es usando **inputs** (no types) en nuestra definición de schema. Los types se usan para indicar returns tanto para queries como para mutations

```
input MessageInput {
  content: String
  author: String
}
```

```

}

type Message {
  id: ID!
  content: String
  author: String
}

type Query {
  getMessage(id: ID!): Message
}

type Mutation {
  createMessage(input: MessageInput): Message
  updateMessage(id: ID!, input: MessageInput): Message
}

```

Ejemplo:

```

var express = require('express');
var { graphqlHTTP } = require('express-graphql');
var { buildSchema } = require('graphql');

// Construct a schema, using GraphQL schema language
const schema = buildSchema(`
  input MessageInput {
    content: String
    author: String
  }

  type Message {
    id: ID!
    content: String
    author: String
  }

  type Query {
    getMessage(id: ID!): Message
  }

  type Mutation {
    createMessage(input: MessageInput): Message
    updateMessage(id: ID!, input: MessageInput): Message
  }
`);

// If Message had any complex fields, we'd put them on this object.
class Message {
  constructor(id, {content, author}) {
    this.id = id;
    this.content = content;
    this.author = author;
  }
}

// Maps username to content

```

```

var fakeDatabase = {};

var root = {
  getMessage: ({id}) => {
    if (!fakeDatabase[id]) {
      throw new Error('no message exists with id ' + id);
    }
    return new Message(id, fakeDatabase[id]);
  },
  createMessage: ({input}) => {
    // Create a random id for our "database".
    var id = require('crypto').randomBytes(10).toString('hex');

    fakeDatabase[id] = input;
    // Abajo la invocación a un controller, service command, etc que va a una base,
    //envía un mail, hace un request a una API guarda un file, etc
    return new Message(id, input);
  },
  updateMessage: ({id, input}) => {
    if (!fakeDatabase[id]) {
      throw new Error('no message exists with id ' + id);
    }
    // This replaces all old data, but some apps might want partial update.
    // Abajo la invocación a un controller, service command, etc que va a una base,
    //envía un mail, hace un request a una API guarda un file, etc
    fakeDatabase[id] = input;
    return new Message(id, input);
  },
};

var app = express();
app.use('/graphql', graphqlHTTP({
  schema: schema,
  rootValue: root,
  graphiql: true,
}));
app.listen(4000, () => {
  console.log('Running a GraphQL API server at localhost:4000/graphql');
});

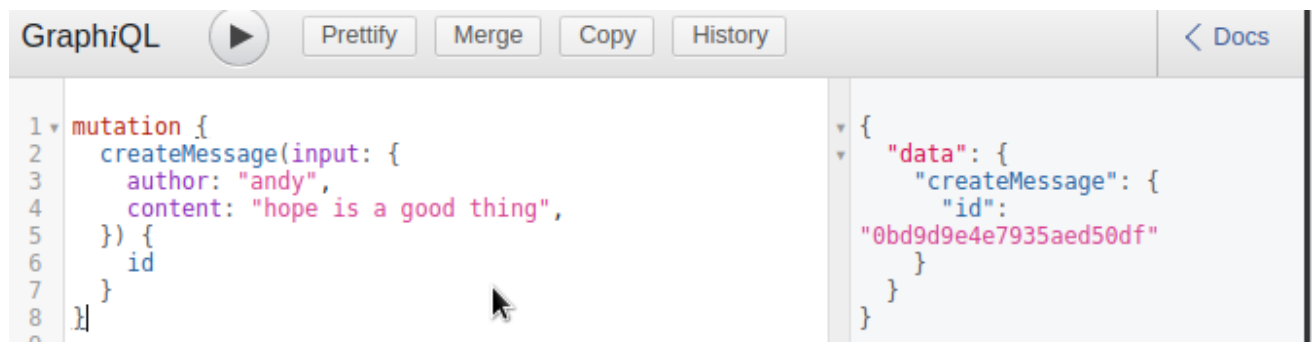
```

Ejemplo:

```

mutation {
  createMessage(input: {
    author: "andy",
    content: "hope is a good thing",
  }) {
    id
  }
}

```



Middleware

<https://graphql.org/graphql-js/authentication-and-express-middleware/>

Ejemplo

```
var express = require('express');
var { graphqlHTTP } = require('express-graphql');
var { buildSchema } = require('graphql');

var schema = buildSchema(`
  type Query {
    ip: String
  }
`);

const loggingMiddleware = (req, res, next) => {
  console.log('ip:', req.ip);
  next();
};

var root = {
  ip: function (args, request) {
    return request.ip;
  }
};

var app = express();
app.use(loggingMiddleware);
app.use('/graphql', graphqlHTTP({
  schema: schema,
  rootValue: root,
  graphiql: true,
}));
app.listen(4000);
console.log('Running a GraphQL API server at localhost:4000/graphql');
```

Solo es necesario usar el middleware como se haría en cualquier app express. El objeto **request** estará disponible como argumento en cualquier resolver.

links para middlewares de autenticación:

- [Passport](#),
- [express-jwt](#)
- [express-session](#).

Javascript - Apollo - Prisma (Segunda Parte)

<https://www.howtographql.com/graphql-js/0-introduction/>

A continuación se describe cada punto del tutorial del link, con algunas anotaciones adicionales.

Intro

En esta parte se usarán las siguientes tecnologías:

- [Apollo Server 2.18](#): Server GQL enfocado en fácil configuración y una alta performance
- [Prisma](#): Reemplaza a un ORM tradicional, provee capa de repositorio autogenerada mediante los schemas. Se utiliza un cliente prisma para acceder a la base desde GQL
- [GraphQL Playground](#): Es un “GraphQL IDE” que permite interactuar con una GraphQL API enviándole queries y mutations (similar a la interfaz graphiql vista anteriormente). Otras opciones a playground son playground de escritorio, Altair entre otros.

Crear proyecto, primeras queries

- 1- Crear carpeta nuevo proyecto y ejecutar `npm init -y`
- 2- Crear folder src y file `src/index.js`
- 3- Instalar apollo server:

```
npm install apollo-server@^2 graphql@^14.6.0
```

(el tutorial sobre el que se basa el taller está hecho para la versión de apollo server 2, no obstante desde julio 2021 está disponible apollo-server 3)

Características de apollo-server

- Cumple las especificaciones de GraphQL
- Funcionalidades con **subscriptions** GraphQL en tiempo real
- Soporte listo para Playground
- Extensible mediante middlewares de express
- Resuelve directivas personales en el Schema GraphQL
- Trazabilidad de performance de queries
- Puede ejecutarse en múltiples plataformas

<https://www.howtographql.com/graphql-js/1-getting-started/>

```
const { ApolloServer } = require('apollo-server');
```

```
// 1
const typeDefs = `
  type Query {
    info: String!
  }
`

// 2
const resolvers = {
  Query: {
    info: () => `Esta es la API del taller`
  }
}

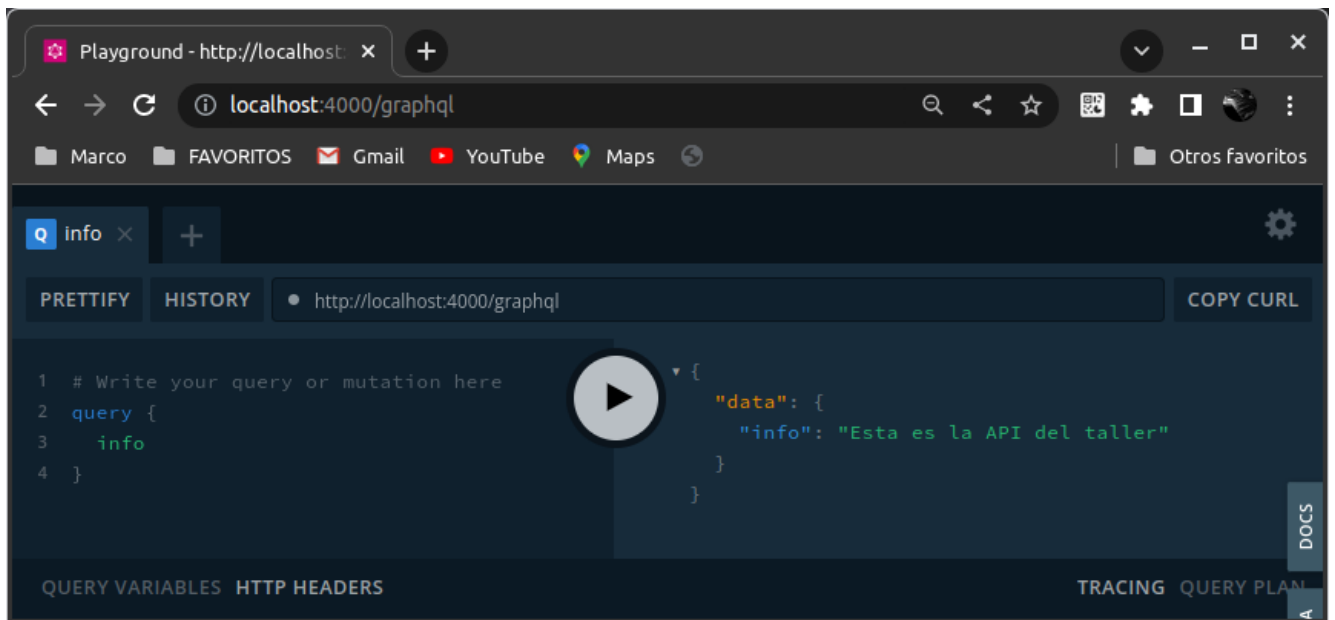
// 3
const server = new ApolloServer({
  typeDefs,
  resolvers,
})

server
  .listen()
  .then(({ url }) =>
    console.log(`Server is running on ${url}`)
  );
```

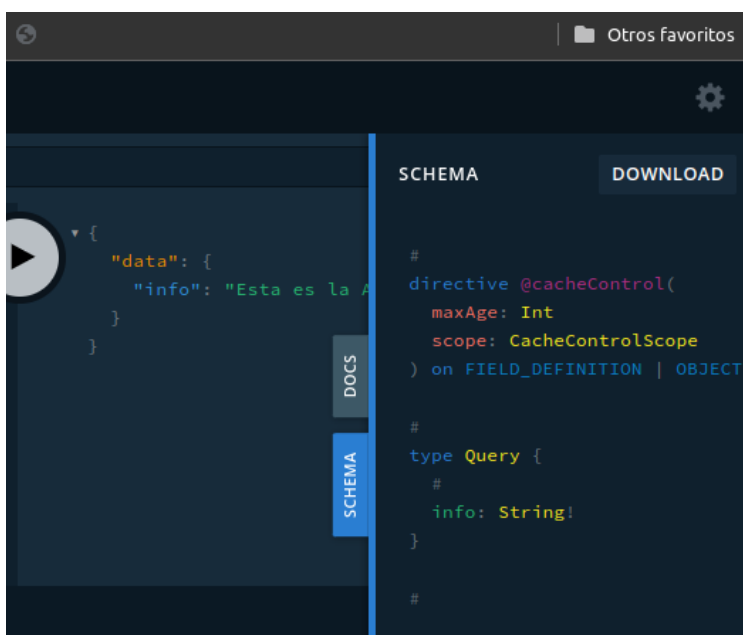
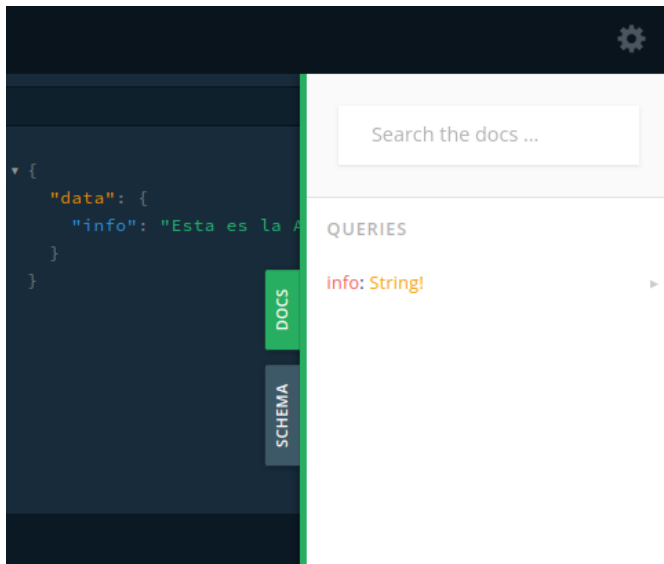
Notas:

- Observar que la configuración del server es mucho más simple y limpia respecto a la librería nativa (vista en la introducción de este taller)
- Como ya se ha visto en la introducción hay definición de schema, resolvers y el propio server graphql iniciado mediante la librería ApolloServer:
 - la constante typeDefs define el **schema** GraphQL, donde se define en este ejemplo una sola query con un único field “info” (podríamos decir que la Query es llama info o es “la query info”)
 - El objeto **resolvers** es la implementación actual del schema GraphQL definido. Llamamos resolvers al conjunto de implementaciones en functions o métodos que “resuelven” el schema (las queries y mutations). La estructura del resolver que respeta a la de definición de tipo `typeDefs: Query.info`.
 - El schema y resolver son empaquetados y pasados a ApolloServer (importado desde la librería apollo-server)

Al ejecutar, vemos la interfaz de Playground



Similar a graphql, podemos ver la documentation generada a través de playground



Sobre GraphQL schema

Los schemas GraphQL están escritos usualmente en el lenguaje de definición de GraphQL [Schema Definition Language](#) (SDL). Tiene un sistema de tipos que, como vimos, permite definir estructura de datos (al igual que cualquier lenguaje de programación tipado como java, TypeScript).

Como vimos todo schema GraphQL tiene “tres principales tipos raíz”: *Query*, *Mutations* y *Subscriptions*. Estos tipos de raíz se corresponden con esos tres posibles tipos de operación. Se denominan fields de raíz (root fields) y definen las operaciones disponibles en la API.

Consideremos el siguiente schema

```
type Query {
  users: [User!]!
  user(id: ID!): User
}

type Mutation {
  createUser(name: String!): User!
}

type User {
  id: ID!
  name: String!
}
```

En este ejemplo tenemos tres fields a nivel de root.

Hay 2 queries, una de users, cuyo schema determina que sí o sí debe retornar un array de al menos un user, y otra de user por id, la cual retorna uno o ningún user

Algunos ejemplo de operaciones posibles con las definiciones anteriores

```
# Query for all users
query {
  users {
    id
    name
  }
}
```

En este caso proyectamos los fields que queremos recibir, pero en la definición del schema del type User, el id y el name son obligatorios. Retorna [User!]!, en caso de no retornar un array de Users, dará error. Con esto aseguramos que no puede venir null o un array vacío.

- Si fuera [User]! quiere decir que el response sí o sí debe retornar un array, pero el mismo puede contener elementos User o Undefined
- Si fuera [User!], quiere decir que puede o no retornar un Array de Users, si lo retorna, debe tener al menos un User

```
# Query a single user by their id
query {
  user(id: "user-1") {
```

```

      id
      name
    }
  }
}

```

Recibe el parámetro obligatorio id, que seguramente se resolverá como filtro.

```

# Create a new user
mutation {
  createUser(name: "Bob") {
    id
    name
  }
}

```

Una Query Simple

Fuente: <https://www.howtographql.com/graphql-js/2-a-simple-query/>

Ejemplo

Modificamos nuestro index, agregando una query feed y el tipo Link a los typeDefs

```

const typeDefs = `
  type Query {
    info: String!
    feed: [Link!]!
  }

  type Link {
    id: ID!
    description: String!
    url: String!
  }
`

```

- Implementar resolvers (por el momento seguimos trabajando sobre el mismo archivo index).
- Agregamos un objeto link en memoria.

```

let links = [{
  id: 'link-0',
  url: 'www.howtographql.com',
  description: 'Fullstack tutorial for GraphQL'
}]

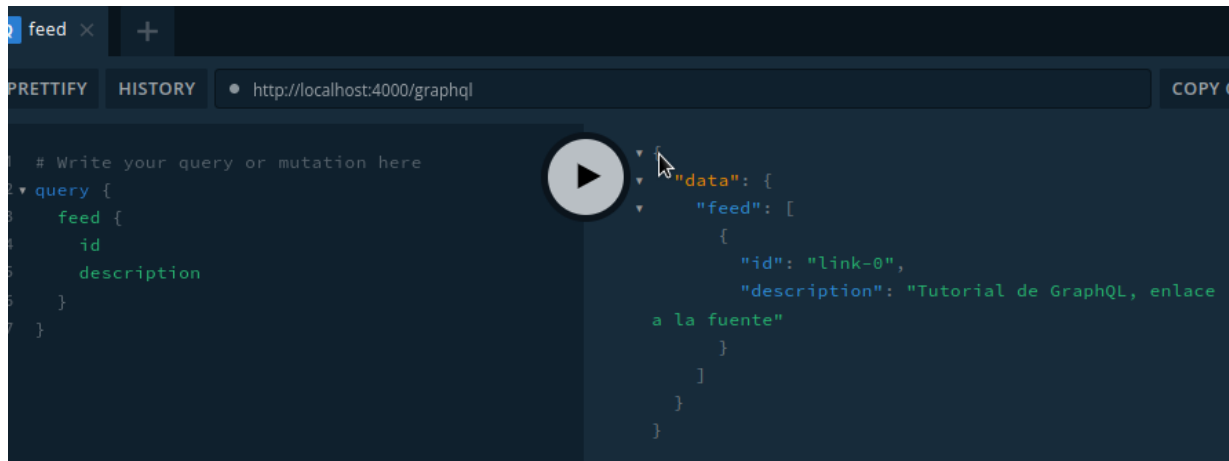
```

```

const resolvers = {
  Query: {
    info: () => `This is the API of a Hackernews Clone`,
    feed: () => links,
  },
  Link: {
    id: (parent) => parent.id,
    description: (parent) => parent.description,
    url: (parent) => parent.url,
  }
}

```

Levantamos el server y accedemos al playground de apollo, vemos que autocompleta la documentación.



Nota: La definición del **resolver de tipo** Link, no siempre es necesaria, se usa generalmente para agregar fields, cálculos o limpieza de datos a nivel documento row, u objeto que se realizan en tiempo de ejecución antes de retornar el resultado. Por ejemplo resolver siempre una relación para evitar establecerlas en cada response de query o mutation

La query feed espera Array de Link, al procesar el json retornado con Link ejecuta las acciones definidas en el resolver de tipo. Este json viene en el objeto parent o root, el cual es el resultado de la ejecución del resolver de nivel previo (en nuestro caso el resolver feed definido en resolvers.Query)

Podríamos agregar un field en el schema GraphQL, y Link definido en resolvers. Siempre en cada mutation, antes de devolver objetos Link resolverá el nuevo field. Al invocar el nombre del field, si hay una function se ejecuta (sería una forma de agregar “virtuals” a nivel de código). De esta manera en query están los resolvers al estilo controllers y al poner el mismo nombre de los objetos retornados, podemos definir nuevos comportamientos antes de retornarlo.

Resolver de tipo: Agregando un elemento calculado

Por ejemplo: Queremos agregar un field nuevo opcional que concatena la fecha de hoy a la descripción y lo retorna en un nuevo field. No tocamos el objeto ni el resolver de la query, podemos hacerlo sólo modificando el schema y el objeto Link retornado

Al schema agregamos el nuevo field (lo hago opcional)

```
type Link {
  id: ID!
  description: String!
  url: String!
  dateAndDescription: String
}
```

```
query {
  feed {
    id
    description
    dateAndDescription
  }
}
```

```
}  
}
```

Si consultamos así la query, sin ningún otro cambio nos retornará:

```
{  
  "data": {  
    "feed": [  
      {  
        "id": "link-0",  
        "description": "Tutorial de GraphQL, enlace a la fuente",  
        "dateAndDescription": null  
      }  
    ]  
  }  
}
```

Al haberlo establecido como valor opcional, puede retornar null si lo cambiamos en la definición por:

```
type Link {  
  id: ID!  
  description: String!  
  url: String!  
  dateAndDescription: String!  
}
```

nos dará el siguiente error:

```
"errors": [  
  {  
    "message": "Cannot return null for non-nullable field Link.dateAndDescription.",  
  }  
]
```

Añadimos el comportamiento deseado que se ejecutará en cada elemento antes de ser retornado

Quedándonos Link dentro de resolvers:

```
Link: {  
  id: (parent) => parent.id,  
  description: (parent) => parent.description,  
  url: (parent) => parent.url,  
  dateAndDescription: (parent) =>  
    `${new Date().toISOString().split("T")[0]}: ${parent.description}`,  
},
```

De hecho, podríamos eliminar los comportamientos por default de los otros fields ya que no hacen más que retornar el valor que entra en parent.

```
Link: {  
  dateAndDescription: (parent) =>  
    `${new Date().toISOString().split("T")[0]}: ${parent.description}`,  
},
```

Si ejecutamos nuevamente la query obtenemos

```
{
  "data": {
    "feed": [
      {
        "id": "link-0",
        "description": "Tutorial de GraphQL, enlace a la fuente",
        "dateAndDescription": "2022-10-17: Tutorial de GraphQL, enlace a la fuente"
      }
    ]
  }
}
```

En este caso Link es un **resolver de tipo** (no es de query ni de mutation)

Código del ejemplo:

```
const { ApolloServer } = require("apollo-server");

const typeDefs = `
  type Query {
    info: String!
    feed: [Link!]!
  }

  type Link {
    id: ID!
    description: String!
    url: String!
    dateAndDescription: String!
  }
`;

let links = [
  {
    id: "link-0",
    url: "www.howtographql.com",
    description: "Tutorial de GraphQL, enlace a la fuente",
  },
];

const resolvers = {
  Query: {
    info: () => `Esta es la API del taller`,
    feed: () => links,
  },
  Link: {
    dateAndDescription: (parent) =>
      `${new Date().toISOString().split("T")[0]}: ${parent.description}`,
  },
};
```



```

    },
  };

  const server = new ApolloServer({
    typeDefs,
    resolvers,
  });

  server.listen().then(({ url }) => console.log(`Server is running on ${url}`));

```

Proceso de resolución de las queries

Considerando la siguiente query

```

query {
  feed {
    id
    url
    description
  }
}

```

Todos estos fields son los que se pueden ver en la definición del schema. Vimos que cada field es “resuelto” por un resolver

El server GraphQL invoca a las functions resolver para los fields contenidos en la query, y los empaqueta de acuerdo a la capa de la query.

El resolver del tipo Link

```

Link: {
  id: (parent) => parent.id,
  description: (parent) => parent.description,
  url: (parent) => parent.url,
}

```

Todos los resolvers reciben 4 argumentos (siendo parent el primero):
los mismos son: **parent**, **args**, **context** e **info**

El primer argumento **parent**, (o también llamado **root**) es “**el resultado de la ejecución del resolver de nivel previo**”:

- En primer lugar invoca al resolver **feed** y retorna los datos completos almacenados, en nuestro ejemplo, en links
- En un segundo nivel de ejecución, el server GraphQL tiene la inteligencia suficiente para asociar cada elemento retornado por la primer ejecución con el resolver de tipo Link, ya que por la definición de schema antes de retornar un tipo invoca a su resolver de tipo. Por lo tanto, por cada elemento retornado por la ejecución del primer resolver en el array [Link], invocará un resolver de tipo Link, iterando dicho array y enviando al primer argumento **parent** cada objeto.

Cabe la **advertencia** de performance, por ejemplo al retornar una gran cantidad de elementos y efectuar operaciones costosas o con gran demora. (ya que se ejecutará por cada uno).

Una Mutation Simple

Fuente: <https://www.howtographql.com/graphql-js/3-a-simple-mutation/>

Extendiendo la definición del schema

Modificamos nuestro typeDefs:

```
const typeDefs = `
  type Query {
    info: String!
    feed: [Link!]!
  }

  type Mutation {
    post(url: String!, description: String!): Link!
  }

  type Link {
    id: ID!
    description: String!
    url: String!
  }
`
```

Creamos un archivo para las definiciones de schema graphql:

Como vemos el schema se está haciendo grande como para dejarlo todo dentro de un mismo js,

1- Dentro de scr, creamos un nuevo file llamado **schema.graphql**

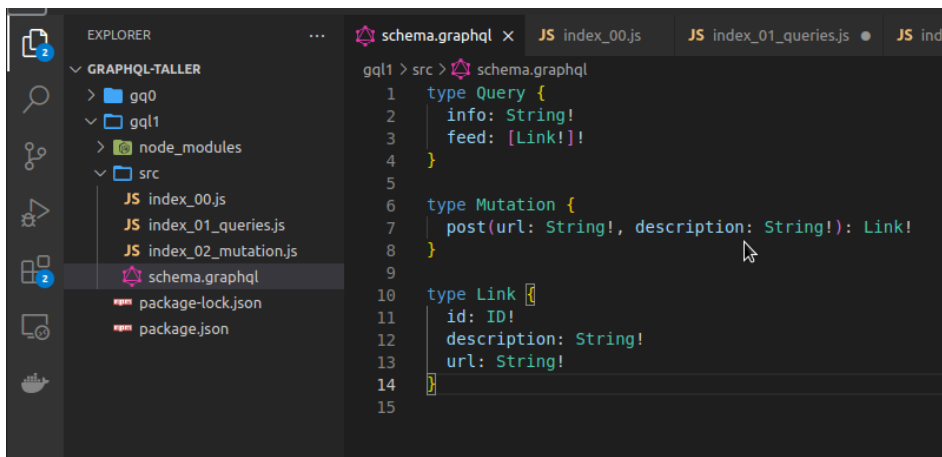
2- Copiamos el contenido de typeDefs y lo pegamos en el file

```
type Query {
  info: String!
  feed: [Link!]!
}

type Mutation {
  post(url: String!, description: String!): Link!
}

type Link {
  id: ID!
  description: String!
  url: String!
}
```

Nota: Podemos usar extensiones en node como GraphQL: Language Feature Support y GraphQL, Syntax Highlighting



Modificamos index, usando la librería fs, para leer el nuevo archivo

```
const fs = require('fs');
const path = require('path');

const server = new ApolloServer({

  typeDefs: fs.readFileSync(
    path.join(__dirname, 'schema.graphql'),
    'utf8'
  ),

  resolvers,
})
```

Código:

```
const fs = require("fs");
const path = require("path");
const { ApolloServer } = require("apollo-server");

let links = [
  {
    id: "link-0",
    url: "www.howtographql.com",
    description: "Tutorial de GraphQL, enlace a la fuente",
  },
];

const resolvers = {
  Query: {
    info: () => `Esta es la API del taller`,
    feed: () => links,
  },
}
```

```

Mutation: {
  post: (parent, args) => {
    let idCount = links.length;
    const link = {
      id: `link-${idCount++}`,
      description: args.description,
      url: args.url,
    };
    links.push(link);
    return link;
  },
},
};

const server = new ApolloServer({
  typeDefs: fs.readFileSync(path.join(__dirname, "schema.graphql"), "utf8"),

  resolvers,
});

server.listen().then(({ url }) => console.log(`Server is running on ${url}`));

```

Hemos añadido una mutation para poder agregar elementos al array links. Como vimos anteriormente (en la intro) las mutations reciben el argumento args, el cual aporta los argumentos de la operación definidos en el schema de la mutation.

Probando la mutation

```

mutation {
  post(url: "www.prisma.io", description: "Prisma replaces traditional ORMs") {
    id
  }
}

```

Devolverá:

```

{
  "data": {
    "post": {
      "id": "link-1"
    }
  }
}

```

Luego podemos usar las queries ya definidas para buscar los links existentes

```

query {
  feed {
    id
    description
  }
}

```

```

    }
  }
}

{
  "data": {
    "feed": [
      {
        "id": "link-0",
        "description": "Tutorial de GraphQL, enlace a la fuente"
      },
      {
        "id": "link-1",
        "description": "Prisma replaces traditional ORMs"
      },
      {
        "id": "link-2",
        "description": "Prisma replaces traditional ORMs"
      }
    ]
  }
}

```

Ejercicio

Implementar los resolver con la siguiente ampliación del schema:

```

type Query {
  # Fetch a single link by its `id`
  link(id: ID!): Link
}

type Mutation {
  # Update a link
  updateLink(id: ID!, url: String, description: String): Link

  # Delete a link
  deleteLink(id: ID!): Link
}

```

Código Ejercicio

```

const fs = require("fs");
const path = require("path");
const { ApolloServer } = require("apollo-server");

let links = [
  {
    id: "link-0",
    url: "www.howtographql.com",
    description: "Tutorial de GraphQL, enlace a la fuente",
  },
];

```

```

const resolvers = {
  Query: {
    info: () => `Esta es la API del taller`,
    feed: () => links,
  },

  Mutation: {
    post: (parent, args) => {
      let idCount = links.length;
      const link = {
        id: `link-${idCount++}`,
        description: args.description,
        url: args.url,
      };
      links.push(link);
      return link;
    },

    updateLink: (parent, args) => {
      const { id, url, description } = args;
      const link = links.find((x) => x.id == id);
      if (!link) return undefined;
      link.url = url;
      link.description = description;
      return link;
    },

    deleteLink: (parent, args) => {
      const { id } = args;
      const idx = links.indexOf(links.find((x) => x.id == id));
      if (idx >= 0) {
        const deleted = links.splice(idx, idx)[0];
        return deleted;
      }
    },
  },
};

const server = new ApolloServer({
  typeDefs: fs.readFileSync(path.join(__dirname, "schema.graphql"), "utf8"),

  resolvers,
});

server.listen().then(({ url }) => console.log(`Server is running on ${url}`));

```

Agregando una base de datos con Prisma

<https://www.howtographql.com/graphql-js/4-adding-a-database/>

Se usará [Prisma](#) como ORM seteando un [SQLite](#)

Ver también las [Bases de datos soportadas por Prisma](#)

Prisma

Prisma es una herramienta open source para gestionar acceso a la baes de datos. Consiste principalmente en 3 herramientas

- Prisma Client
- Prisma Migrate
- Prisma Studio

Configuración

- Instalar prisma CLI y prisma client

```
npm install prisma --save-dev
```

```
npm install @prisma/client
```

- Usar prisma CLI para inicializar prisma en el cliente

```
npx prisma init
```

Así como GraphQL, prisma tiene su propia definición de schema. Dentro del directorio prisma creado en el paso anterior, veremos un archivo llamado **schema.prisma**. A fines prácticos podemos pensar en él como el schema de la base de datos

Tiene tres componentes principales

- 1- **Data Source**: Especifica la conección a la base de datos
- 2- **Generator**: Indica que queremos generar
- 3- **Data model**: Define los modelos de nuestra aplicación. Cada model será mapeado a una tabla (o collection) en la base.

Dependiendo de la versión de prisma y su configuración es posible conectarse a:

- PostgreSQL
- MongoDB
- SQL Server
- MySQL
- sqlite

Los SGDB pueden cambiar según la versión de node y de prisma (no todas las versiones de prisma soportan todos los motores de base de datos)

- Abrir schema.prisma y añadir el siguiente código

```
// 1
datasource db {
  provider = "sqlite"
  url      = "file:./dev.db"
}
```

```
// 2
generator client {
  provider = "prisma-client-js"
}

// 3
model Link {
  id          Int          @id @default(autoincrement())
  createdAt   DateTime     @default(now())
  description String
  url         String
}
```

Configurar la base SQLite

Migration: Establecer los esquemas en la base de datos DDL

A los efectos de enfocarse en prisma se usará sqlite que no requiere configuración y no requiere un server de base funcionando.

Una vez establecido el datasource, ejecutamos la migración:

```
npx prisma migrate dev
```

Lo que hace una migration, configura, crea en la base de datos las entidades descritas en nuestro schema.prisma, por lo que en nuestro ejemplo creará la tabla Links. Es decir que Prisma ejecuta por nosotros las instrucciones DDL (Data Definition Language) del motor por nosotros.

Nota: es posible hacer un script con una secuencia de comandos sh y prisma, por ejemplo para preparar un ambiente de desarrollo, un deploy, etc.

Nos pedirá un nombre para la migration, colocamos init

Veremos que nos crea un folder de migracion y datos en la base por cada migration que realizamos, para mantener trazabilidad entre el schema de prisma, y las migrations realizadas

Generar el Prisma Client

El prisma client ofrece el acceso a cada elemento de la base de datos definido, similar a utilizar patrón repository, sin la necesidad de programarlo enteramente.

Ejecutamos el siguiente comando:

```
npx prisma generate
```

Esto incluso nos permitirá trabajar con intellisense respecto a los modelos existentes en prisma.

Vamos a escribir nuestra primer query con Prisma Client momentáneamente en un archivo **script.js**

[.../hackernews-node/src/script.js](#)

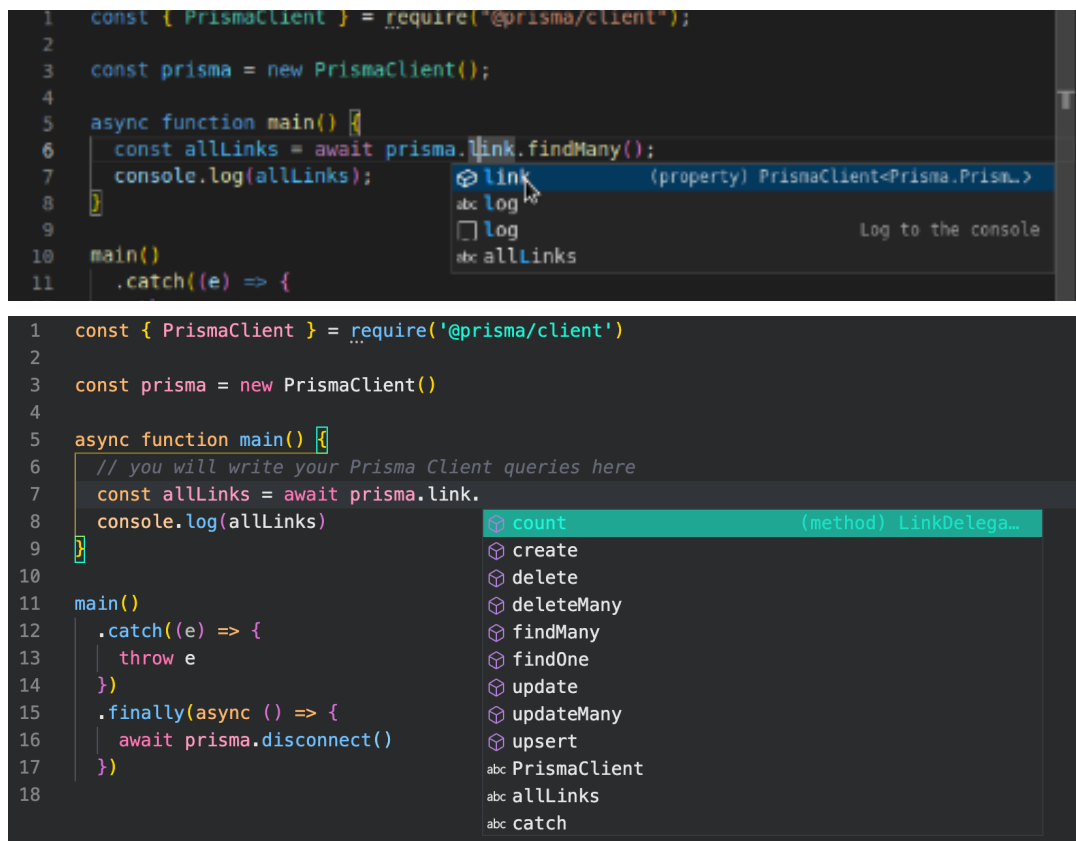

```
// 1 Importamos PrismaClient
const { PrismaClient } = require("@prisma/client")

// 2 Instanciamos PrismaClient
const prisma = new PrismaClient()

// 3 Definimos la function async main
async function main() {
  const allLinks = await prisma.link.findMany()
  console.log(allLinks)
}

// 4 Llamamos a la function main definida para enviar las queries a la base
main()
  .catch(e => {
    throw e
  })
// 5 Cerramos la conexión a la base cuando el script finaliza
  .finally(async () => {
    await prisma.$disconnect()
  })
}
```

Observemos lo que mencionamos del intellisense que autocompleta con las entidades generadas, y con las operaciones posibles:



Veamos qué ocurre al ejecutar este script.js de ejemplo

```
node src/script.js
```

Retorna [], esto es porque realiza la query a la base exitosamente, pero no hay elementos aun

Para hacer una pequeña prueba borrador podemos agregar un código que inserte un elemento justo antes del findMany

```

const { PrismaClient } = require("@prisma/client");

const prisma = new PrismaClient();

async function main() {
  const newLink = await prisma.link.create({
    data: {
      description: "Fullstack tutorial for GraphQL",
      url: "www.howtographql.com",
    },
  });
  const allLinks = await prisma.link.findMany();
  console.log(allLinks);
}

main()
  .catch((e) => {
    throw e;
  })
  .finally(async () => {
    await prisma.$disconnect();
  });

```

Luego, al volver a ejecutar veremos los resultados

```

node src/script.js
[
  {
    id: 1,
    createdAt: 2022-10-17T21:29:26.898Z,
    description: 'Fullstack tutorial for GraphQL',
    url: 'www.howtographql.com'
  }
]

```

Conectando el Server y la Database con Prisma Client

Link: <https://www.howtographql.com/graphql-js/5-connecting-server-and-database/>

Necesitamos importar nuestra librería Prisma Client generada y asociarlo con el server GraphQL para poder acceder a las queries de la base de datos expuestas por el nuevo Prisma Client.

El argumento de resolver GraphQL **context**

Recordemos que los resolvers siempre reciben cuatro argumentos. El argumento context es el 3º argumento recibido

El argumento `context` es un objeto JavaScript al cual puede acceder cualquier resolver para leer o escribir datos en él. Es un mecanismo para que los resolvers puedan comunicarse. Es posible escribir en el objeto `context` una vez que el server GraphQL se ha inicializado.

De esta manera podemos establecer una instancia del Prisma Client dentro del `context` al inicializar el server, y acceder desde los resolvers a los argumentos del `context` y al Prisma Client.

Actualizar las functions resolver para poder usar Prisma Client

1- Agregar PrismaClient en index.js

```
const { PrismaClient } = require("@prisma/client");
```

2- Podemos atachar una instancia de PrismaClient al `context` cuando el serer GraphQL se inicializa:

En `index.js` guardamos una instancia de `PrismaClient` en una variable y actualizamos la instanciación de `GraphQLServer`

```
const { PrismaClient } = require("@prisma/client");
const prisma = new PrismaClient();
// ...
const server = new ApolloServer({
  typeDefs: fs.readFileSync(path.join(__dirname, "schema.graphql"), "utf8"),
  resolvers,
  context: {
    prisma,
  }
});
```

Ahora todos nuestros resolvers tienen acceso a `context`

3- Refactorizar resolvers

Modificamos nuestro `index`, usando `prisma`

```
const fs = require("fs");
const path = require("path");
const { ApolloServer } = require("apollo-server");
const { PrismaClient } = require("@prisma/client");
const prisma = new PrismaClient();

const resolvers = {
  Query: {
    info: () => `Esta es la API del taller`,
    feed: async (parents, args, context) => {
      return context.prisma.link.findMany();
    },
  },
}
```

```

    },

    Mutation: {
      post: (parent, args, context, info) => {
        const newLink = context.prisma.link.create({
          data: {
            url: args.url,
            description: args.description,
          },
        });
        return newLink;
      },

      updateLink: async (parent, args, context, info) => {
        const { id, url, description } = args;
        const result = await context.prisma.link.update({
          where: { id: parseInt(id) },
          data: {
            url: url ? url : undefined,
            description: description ? description : undefined,
          },
        });
        return result;
      },

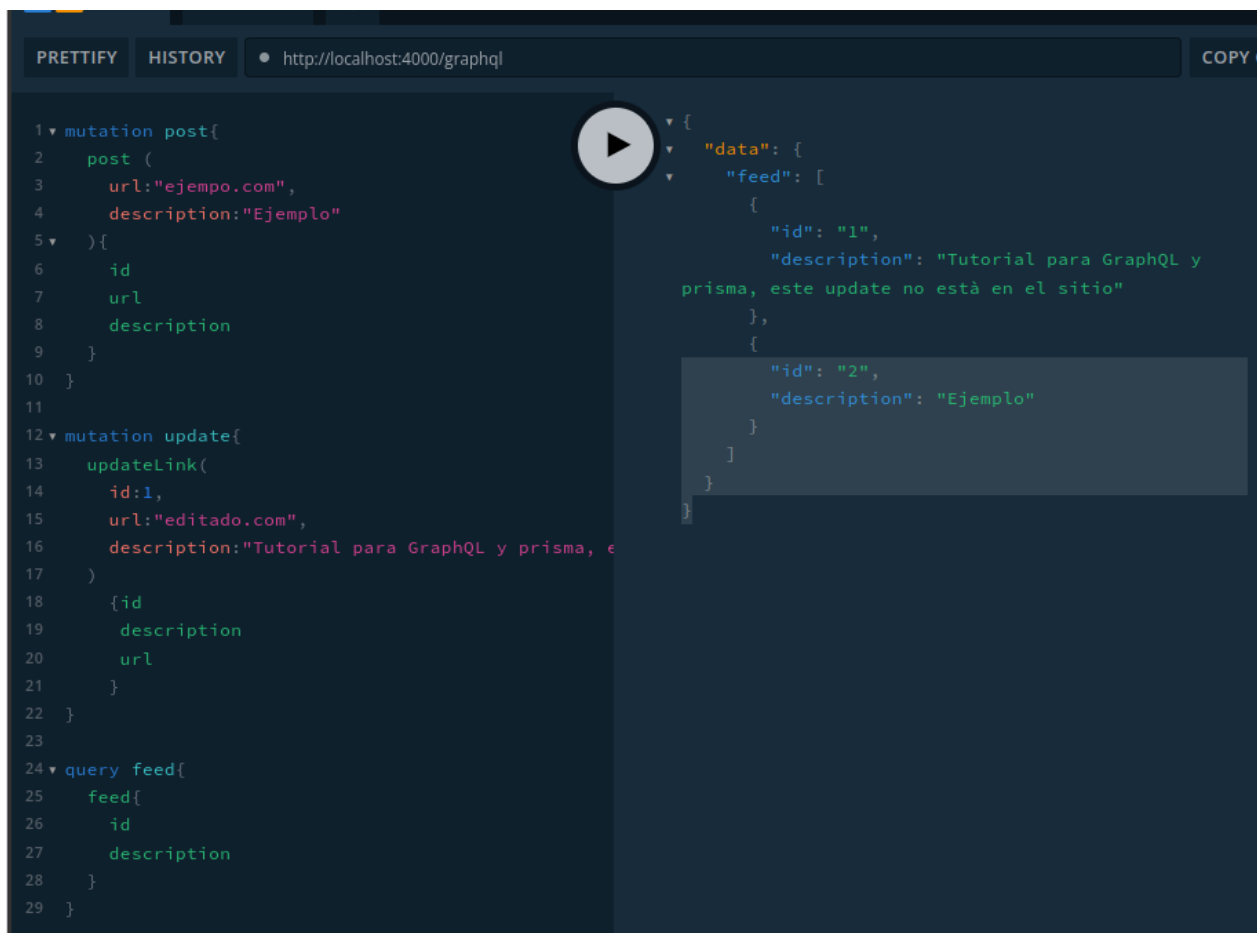
      deleteLink: async (parent, args, context, info) => {
        const { id } = args;
        return await context.prisma.link.delete({
          where: { id: parseInt(id) },
        });
      },
    },
  },
};

const server = new ApolloServer({
  typeDefs: fs.readFileSync(path.join(__dirname, "schema.graphql"), "utf8"),
  resolvers,

  context: {
    prisma,
  },
});

server.listen().then(({ url }) => console.log(`Server is running on ${url}`));

```



Usar Prisma Studio

Una GUI que permite interactuar con los datos. En la práctica no es tan común usar estos GUI para acceder a las distintas bases, ya que existen IDEs específicos con set de herramientas completos según el motor. Pero si la interacción con la base no es tan incentiva, puede resultar útil

En la raíz de nuestro proyecto ejecutamos

`npx prisma studio`

Una podemos acceder a la visualización de los datos.

The screenshot shows a web browser window with the address bar displaying `localhost:5555`. The browser has several tabs open, including Google, Prisma, and others. The Prisma Studio interface is visible, showing a table with 2 records. The table has columns for `id`, `createdAt`, `description`, and `url`. The first record has `id` 1, `createdAt` 2022-10-17T21:29:26.8..., `description` Tutorial para GraphQL..., and `url` editado.com. The second record has `id` 2, `createdAt` 2022-10-17T23:19:25.4..., `description` Ejemplo, and `url` ejemplo.com.

id #	createdAt	description	url
1	2022-10-17T21:29:26.8...	Tutorial para GraphQL...	editado.com
2	2022-10-17T23:19:25.4...	Ejemplo	ejemplo.com

Autenticación

<https://www.howtographql.com/graphql-js/6-authentication/>

Se implementará un registro y login de autenticación de usuarios.

En el repo que se comparte ver en branch dev/01_gql_prisma_auth el código final

Agregando un User al model

En ejemplos en branch dev/01_gql_prisma_auth

Si estamos trabajando con GraphQL y Prisma, es recomendable al definir entidades de negocio, es posible diseñar el schema de la base, y modelar las entidades directamente sobre el schema de prisma

- Abrir el schema.prisma y agregar los datos de User y link

```
model Link {
  id          Int          @id @default(autoincrement())
  createdAt   DateTime     @default(now())
  description String
  url         String
  postedBy    User?        @relation(fields: [postedById], references: [id])
  postedById  Int?
}

model User {
  id          Int          @id @default(autoincrement())
  name        String
  email       String       @unique
  password    String
  links       Link[]
}
```

Nota: con esto creamos una relación 1 Usuario a N links, estando la Foreign Key llamada `postedById` en la tabla Link. Prisma permite modelar diferentes mapeos de relaciones: 1 a N y N a M

Al establecer la relación es necesario establecer que un objeto de un tipo (en nuestro caso **postedBy** del type User), y el field que tendrá la relación **postedById**. En el field correspondiente al atributo objeto se configura la relación.

Esto es:

```
postedBy    User?        @relation(fields: [postedById], references: [id])
postedById  Int?
```

La notation `@relation` configura la constraint en la base destino.

Esto en una base de datos relacional se traducirá en el DDL correspondiente generando las constraints, a modo de ejemplo podría generar una sentencia en la base de datos destino similar a la siguiente:

```
CREATE TABLE Link(
  id INT NOT NULL PRIMARY KEY,
  createdAt DATE DEFAULT GETDATE()
  description VARCHAR(255)
  url VARCHAR(255)

  postedById INT NOT NULL,
  CONSTRAINT FK_postedById FOREIGN KEY (postedById) REFERENCES User(id)
)
```

También permite diferentes tipos de constraints, indexados, valores defaults, valores guid, uuid, ejecución de ciertas functions por default, etc.

Generar nueva migration y Actualizar el prisma client

En este punto es necesario hacer 2 cosas:

- 1. Los cambios en el modelo deben impactar en la base destino
Para esto ejecutamos una migration

```
npx prisma migrate dev --name "add-user-model"
```

- 2. El modelo que levanta el cliente prisma, debe ser actualizado
Para esto necesitamos regenerar el PrismaClient

```
npx prisma generate
```

Modificar nuestro schema GraphQL

- Agregaremos las nuevas mutations y tipos necesarios.

```
type Mutation {
  post(url: String!, description: String!): Link!
  updateLink(id: ID!, url: String, description: String): Link
  deleteLink(id: ID!): Link
  signup(email: String!, password: String!, name: String!): AuthPayload
  login(email: String!, password: String!): AuthPayload
}
```

Ahora debemos agregar las definiciones de tipo **User** y **AuthPayload**

```
type User {
  id: ID!
  name: String!
  email: String!
  links: [Link]!
}
```

```
type AuthPayload {
  token: String
  user: User
}
```

Las mutations **signup** y **login**: Ambas mutations retornarán el *token* que puede ser usado para autenticaciones posteriores. Este token, es el que utilizaremos en el header de autenticación para poder ejecutar mutations y queries comunes.

- Podemos agregar, según sea necesario los datos del User asociados al link (según la necesidad). Agregamos un field postedBy del tipo User

```
type Link {
  id: ID!
  description: String!
  url: String!
  postedBy: User
}
```

Nota: Si en Prisma realizamos bien las configuraciones ambos objetos pueden ser obtenidos con la cláusula include.

Implementar los resolvers

Para ordenar un poco el proyecto, vamos a separar los resolvers en un archivo **Mutation.js** y otro **Query.js**. Esta estructura podría ajustarse a las necesidades de nuestro proyecto o arq. Por ejemplo podría tener folders con los nombres de las entidades y en cada uno un archivo mutation y otro query, los cuales pueden consumir otros helpers, servicios, etc.

- Creamos los siguiente archivos

```
mkdir src/resolvers
touch src/resolvers/Query.js
touch src/resolvers/Mutation.js
touch src/resolvers/User.js
touch src/resolvers/Link.js
```

Nota: Personalmente recomiendo no usar mutations como functions compartidas dentro del proyecto entre distintos .js, se puede hacer pero se presta a errores, mejor definir servicios, helpers, etc. Donde podamos desarrollar estas inclusiones de casos de uso o fragmento de funcionalidad compartida. Es decir podríamos pensar en los resolvers como controllers. En general, para pequeños comportamientos y accesos simples a base, se suele hacer como en los siguientes ejemplos.

Archivo Query.js

```
function info() {
  return `Esta es la API del taller`;
}
```



```
function feed(parents, args, context) {
  return context.prisma.link.findMany();
}

module.exports = {
  info,
  feed,
};
```

Archivo Mutation.js (explicadas las nuevas mutations)

Crearemos los resolvers de las mutations definidas en el schema signup y login, y también agregaremos la información de las relaciones. Prestar atención en a la mutation post

```
function post(parent, args, context, info) {
  const { userId } = context;
```

El userId lo tenemos inyectado en el context por la autenticación con token ([ver Archivo index](#))

```
const bcrypt = require("bcryptjs");
const jwt = require("jsonwebtoken");
const { APP_SECRET, getUserId } = require("../utils");

async function post(parent, args, context, info) {
  const { userId } = context;
  return await context.prisma.link.create({
    data: {
      url: args.url,
      description: args.description,
      postedBy: { connect: { id: userId } },
    },
  });
}

async function updateLink(parent, args, context, info) {
  const { id, url, description } = args;
  const result = await context.prisma.link.update({
    where: { id: parseInt(id) },
    data: {
      url: url ? url : undefined,
      description: description ? description : undefined,
    },
  });
  return result;
}

async function deleteLink(parent, args, context, info) {
  const { id } = args;
```

```

return await context.prisma.link.delete({
  where: { id: parseInt(id) },
});
}

async function signup(parent, args, context, info) {
  // 1 Encriptamos el password
  const password = await bcrypt.hash(args.password, 10);

  // 2 Creamos el usuario en la base con PrismaClient
  const user = await context.prisma.user.create({
    data: { ...args, password },
  });

  // 3 Generamos un Json Web Token firmado con un APP_SECRET
  const token = jwt.sign({ userId: user.id }, APP_SECRET);

  // 4 Retornamos el token y los datos del usuario
  return {
    token,
    user,
  };
}

async function login(parent, args, context, info) {
  //1 Buscamos el user a través del unique (esta versión tiene esta function, en
  otras versiones se usa findOne)
  const user = await context.prisma.user.findUnique({
    where: { email: args.email },
  });

  if (!user) {
    throw new Error("No such user found");
  }

  //2 Comparamos el password encriptado
  const valid = await bcrypt.compare(args.password, user.password);
  if (!valid) {
    throw new Error("Invalid password");
  }

  // 3 Generamos un Json Web Token firmado con un APP_SECRET
  const token = jwt.sign({ userId: user.id }, APP_SECRET);

  return {
    token,
    user,
  };
}

module.exports = {
  post,
  updateLink,

```

```
deleteLink,
signup,
login,
};
```

- Instalamos jsonwebtoken y bcryptjs

npm install jsonwebtoken bcryptjs

Archivo utils.js

- Creamos un archivo **utils.js** en la carpeta src

```
const jwt = require('jsonwebtoken');
const APP_SECRET = 'GraphQL-is-aw3some';

function getTokenPayload(token) {
  return jwt.verify(token, APP_SECRET);
}

function getUserId(req, authToken) {
  if (req) {
    const authHeader = req.headers.authorization;
    if (authHeader) {
      const token = authHeader.replace('Bearer ', '');
      if (!token) {
        throw new Error('No token found');
      }
      const { userId } = getTokenPayload(token);
      return userId;
    }
  } else if (authToken) {
    const { userId } = getTokenPayload(authToken);
    return userId;
  }

  throw new Error('Not authenticated');
}

module.exports = {
  APP_SECRET,
  getUserId
};
```

Archivo index.js

Debemos quitar las Mutations y Queries que hemos movido a los files correspondientes, pero más importante implementar la autorización:

```
const { getUserId } = require('./utils');

const server = new ApolloServer({
  typeDefs: fs.readFileSync(
```

```

    path.join(__dirname, 'schema.graphql'),
    'utf8'
  ),
  resolvers,
  context: ({ req }) => {
    return {
      ...req,
      prisma,
      userId:
        req && req.headers.authorization
          ? getUserId(req)
          : null
    };
  }
});

```

El Archivo index.js queda de la siguiente manera:

```

const fs = require("fs");
const path = require("path");
const { ApolloServer } = require("apollo-server");
const { PrismaClient } = require("@prisma/client");
const prisma = new PrismaClient();
const { getUserId } = require("./utils");

const Query = require("./resolvers/Query");
const Mutation = require("./resolvers/Mutation");
const User = require("./resolvers/User");
const Link = require("./resolvers/Link");

const resolvers = {
  Query,
  Mutation,
  User,
  Link,
};

const server = new ApolloServer({
  typeDefs: fs.readFileSync(path.join(__dirname, "schema.graphql"), "utf8"),
  resolvers,

  context: ({ req }) => {
    return {
      ...req,
      prisma,
      userId: req && req.header.authorization ? getUserId(req) : null,
    };
  },
});

server.listen().then(({ url }) => console.log(`Server is running on ${url}`));

```

Nota sobre Autenticaciones, roles y permisos. graphql-shield

Existe un middleware llamado graphql-shield que permite establecer chains de rules y sus validaciones para mutations, queries y resolvers de tipo, administrar roles y permisos, sin necesidad de validar en cada query y mutation si un usuario está autenticado y su rol.

<https://www.npmjs.com/package/graphql-shield>

<https://www.the-guild.dev/graphql/shield>

Ejemplo simple:

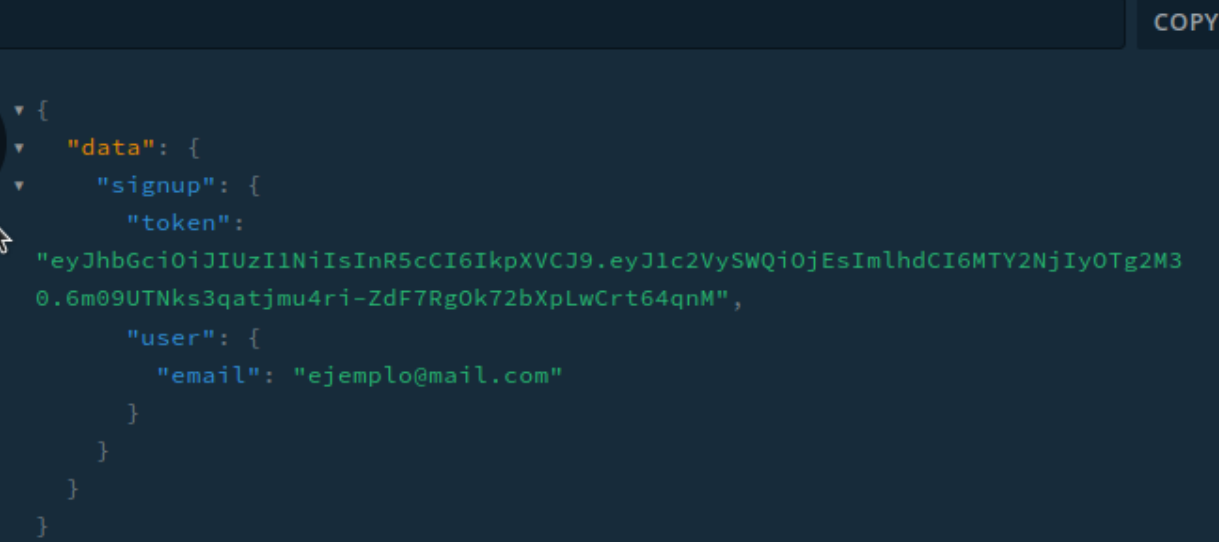
<https://www.albertgao.xyz/2019/12/01/how-to-use-graphql-shield-with-apollo-server-to-authorize-JWT/>

Pruebas

Probamos la mutation signup para crear un usuario

```
mutation signup{
  signup(
    email: "ejemplo@mail.com",
    password:"123",
    name:"marco"){
    token
    user{
      email
    }
  }
}
```

Recibimos el token el cual debemos especificar en el **header Authorization** en las próximas mutations que requieran autenticar al usuario

A screenshot of a JSON response from a GraphQL API. The response is displayed in a dark-themed editor with a 'COPY' button in the top right corner. The JSON structure is as follows:

```
{
  "data": {
    "signup": {
      "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOiJEsImhhdCI6MTY2NjIyOTg2M30.6m09UTNks3qatjmu4ri-ZdF7RgOk72bXpLwCrt64qnM",
      "user": {
        "email": "ejemplo@mail.com"
      }
    }
  }
}
```

Usamos el token para autenticarnos y tratar de realizar un post

Podemos probar autenticarnos

```
mutation login{
  login(email:"ejemplo@mail.com", password:"123"){
    token
```

```

    user{
      name
    }
  }
}

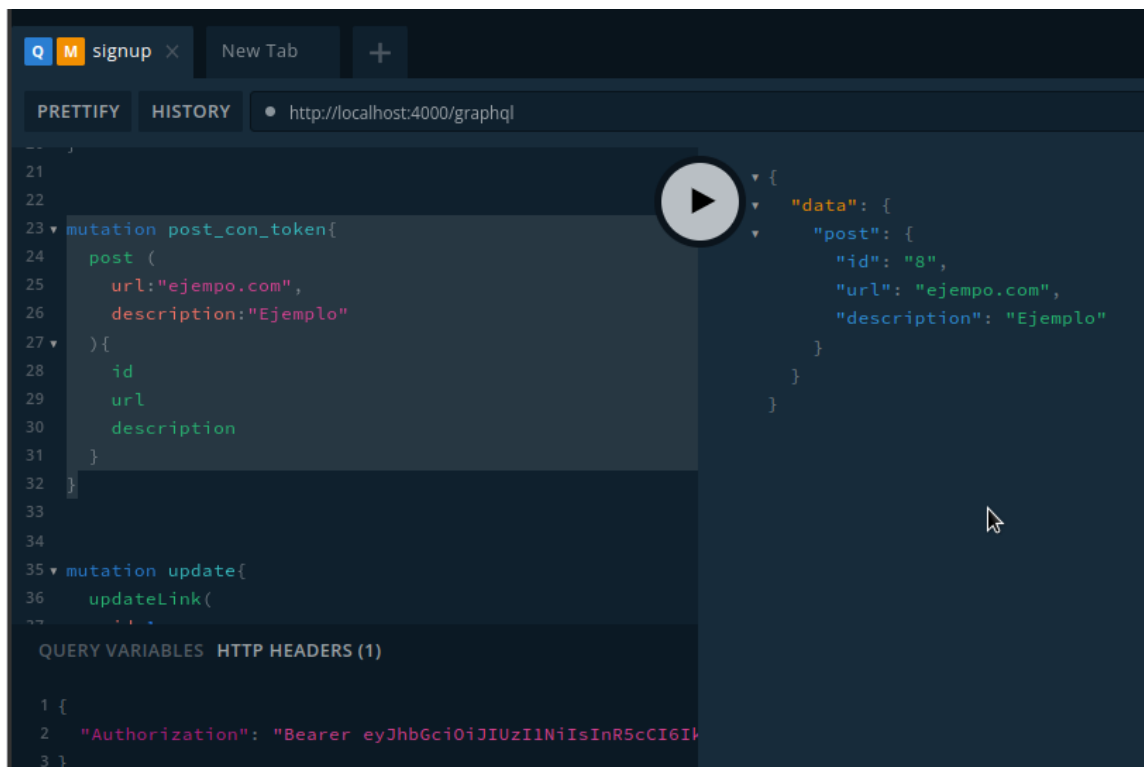
```

Probamos una mutation pasando el header Authorization el token:

```

mutation post_con_token{
  post (
    url:"ejemplo.com",
    description:"Ejemplo"
  )
{
  id
  url
  description
}
}

```



Recordar que cada vez que el server reciba una mutation ejecutará esta parte para userId que agregamos al instanciar ApolloServer en el archivo index.js

```

userId: req && req.headers.authorization ? getUserId(req) : null,

```

Ejemplo del código realizado (para esta parte en el branch **dev/01_gql_prisma_auth**)
https://github.com/marbriganti/taller-gql/tree/dev/01_gql_prisma_auth/workspace/gql1

Subscriptions en GraphQL

<https://www.howtographql.com/graphql-js/7-subscriptions/>

La característica de subscriptions en GraphQL nos permite enviar datos a clientes cuando un evento específico ocurre. Las subscriptions están implementadas usualmente con WebSockets. Este protocolo mantiene una conexión con su cliente suscrito.

El cliente realiza una conexión “long-lived” con el server enviando una query de subscription donde especifica en qué evento está interesado. Cuando ocurre este evento, el server usa la conexión para pushear el dato del evento al/los cliente(s) suscritos.

Implementar subscriptions GraphQL

La librería **apollo-server** provee al contexto de **PubSub** (como se hizo con PrismaClient).

- Incorporamos PubSub, instanciamos y lo proveemos al contexto:

```
const { ApolloServer, PubSub } = require("apollo-server");
const pubsub = new PubSub();

//. . .

const server = new ApolloServer({
  typeDefs: fs.readFileSync(path.join(__dirname, "schema.graphql"), "utf8"),
  resolvers,
  context: ({ req }) => {
    return {
      ...req,
      pubsub,
      prisma,
      userId: req && req.headers.authorization ? getUserId(req) : null,
    };
  },
});
```

Subscribir nuevos elementos Link

1- Necesitamos agregar al schema un tipo Subscription, de manera tal que permita a los clientes suscribirse a nuevas creaciones de elementos Link

dentro de nuestro schema.graphql

```
type Subscription {
  newLink: Link
```

```
}
```

(Los datos de este evento corresponden el tipo *Link* definido en el mismo schema)

2- Necesitamos implementar el **resolver de la subscription para el field newLink**. Resolvers de subscriptions son un poco diferente a los de Queries y Mutations.

- Retornan un **AsyncIterator** el cual será usado por el server GraphQL para pushear los datos del evento al cliente.
- Los resolvers de la subscription son envueltos dentro de un objeto (wrapper) y necesitan proporcionarse como un valor para un field llamado **subscribe**. También es necesario proveer de otro field llamado **resolve** que retorna el dato emitido por el **AsyncIterator**.

Recordar el resolver de una subscription esté en un objeto wrapper con 2 fields:

- **subscribe**: este field provee del resolver de la subscription.
- **resolve**: este field retorna los datos, payload, emitido por el AsyncIterator

```
function newLinkSubscribe(parent, args, context, info) {
  //Accedemos a pubsub desde el context y llamamos a la function asyncIterator
  //pasando el string "NEW_LINK". Esta function se usa para resolver la
  //subscription y pushar los datos del evento
  return context.pubsub.asyncIterator("NEW_LINK");
}

const newLink = {
  subscribe: newLinkSubscribe,
  resolve: (payload) => {
    return payload;
  },
};

module.exports = {
  newLink,
};
```

Añadir las subscriptions a los resolvers de las mutations

1- Agregar el publish del evento en la mutation

Para terminar la implementación de la subscription es necesario llamarla desde un resolver.

Ir al archivo de mutation, y agregar la llamada al **publish** de la librería pubsub que ahora tenemos en el context, con los datos del link creado antes de retornarlo en la mutation.

```
async function post(parent, args, context, info) {
  const { userId } = context;
  const createdLink = await context.prisma.link.create({
    data: {
```



```

    url: args.url,
    description: args.description,
    postedBy: { connect: { id: userId } },
  },
});

context.pubsub.publish("NEW_LINK", createdLink);

return createdLink;
}

```

Este método publish (el cual agregamos en la func Subscription->newLinkSubscribe) recibe el los datos como segundo argumento (la susbscription los devolverá en el payload del field resolve del wrapper)

2- No olvidar incorporar los resolvers de subscriptions (el archivo de Subscription) a nuestro index.js

```

const Subscription = require('./resolvers/Subscription'); //recordar agregarlo

const resolvers = {
  Query,
  Mutation,
  User,
  Link,
  Subscription //recordar agregarlo
};

```

Probar las subscriptions

1- Ir al Playground, abrir un nuevo tab y subscribirse al event

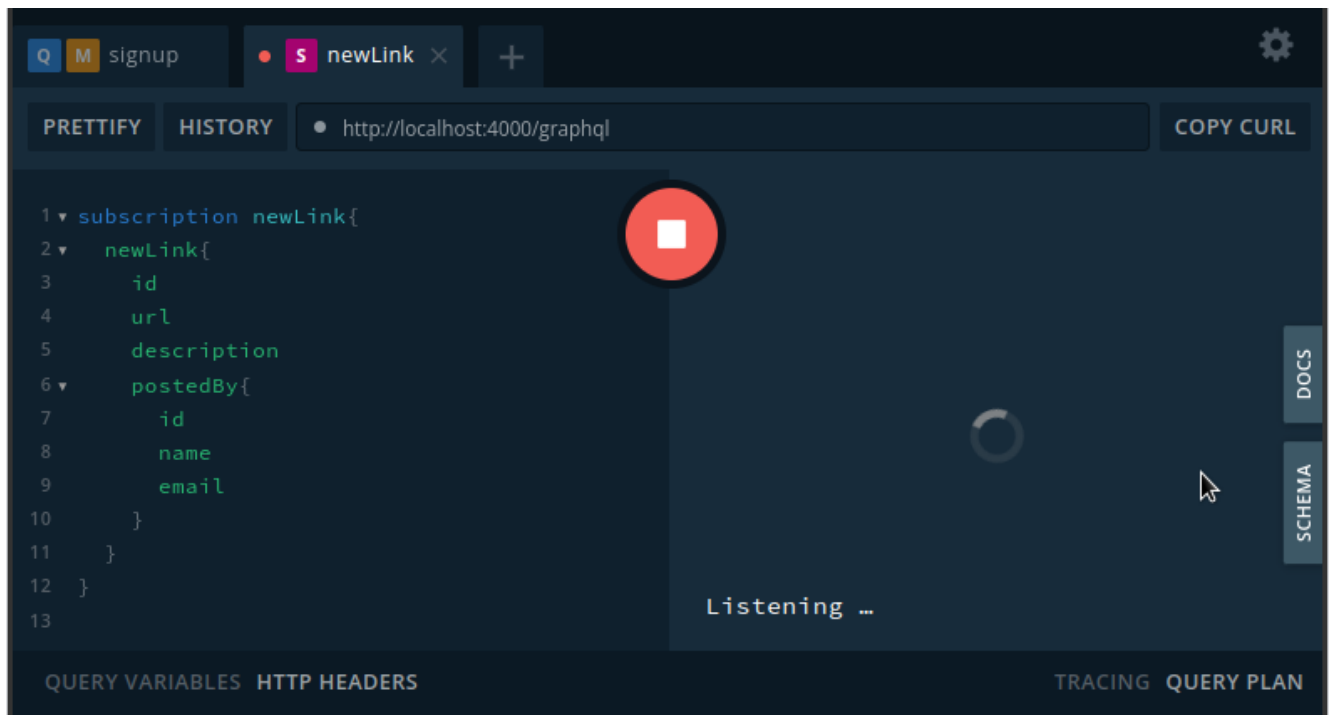
En este punto veremos la información sólo cuando ocurra el evento "NEW_LINK"

```

subscription newLink{
  newLink{
    id
    url
    description
    postedBy{
      id
      name
      email
    }
  }
}

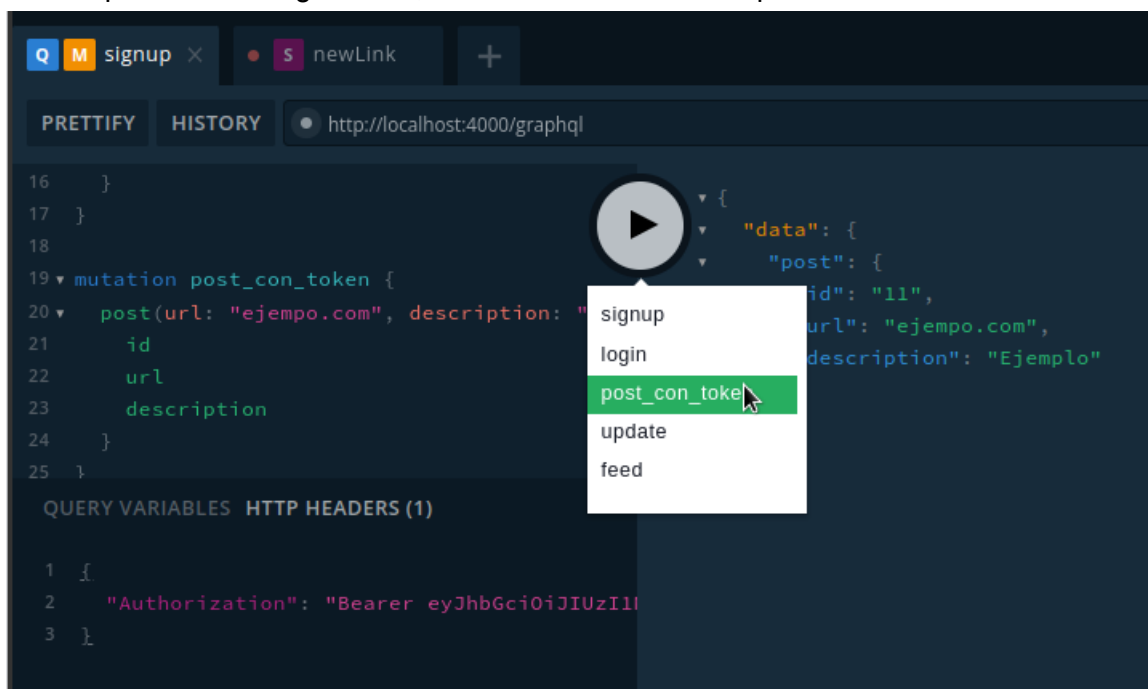
```

Cuando ejecutemos la subscription, veremos el mensaje “Listening...” indicando que el cliente está escuchando por nuevos eventos.



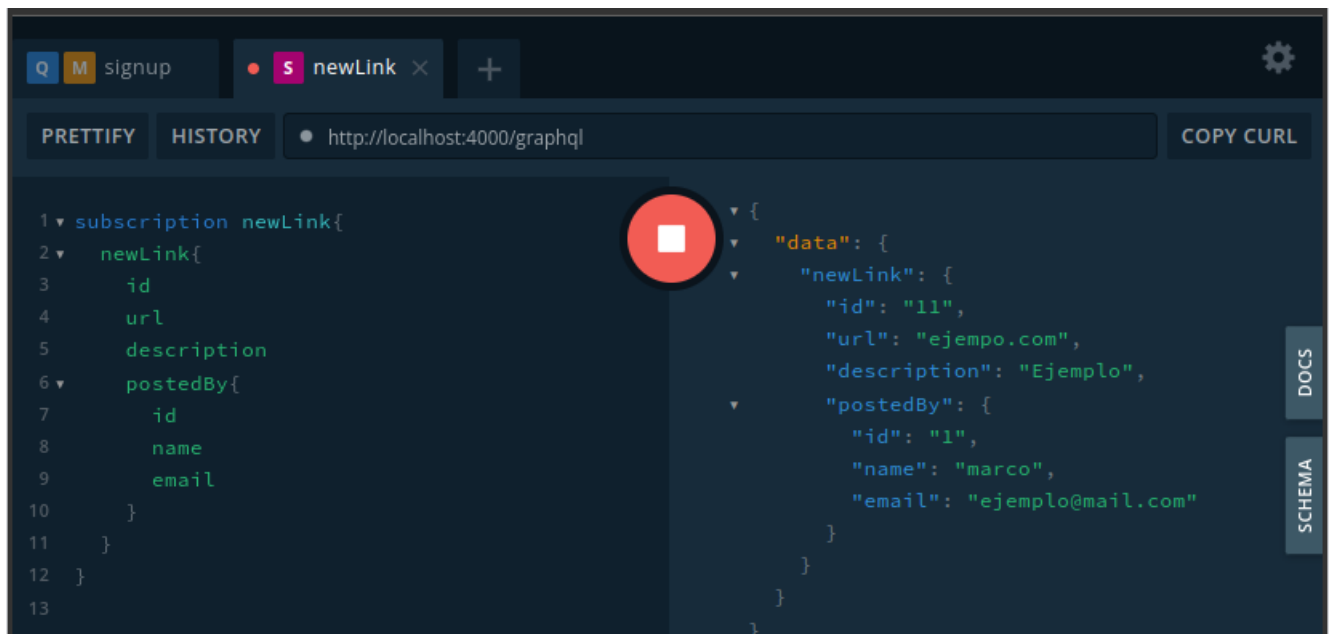
2- Crear un nuevo link mediante la mutation post

En otra pestaña nos logueamos, invocamos a la mutation post enviando el token



Vemos el resultado de la mutation correctamente

Vamos la pestaña de nuestra subscription que está en ejecución

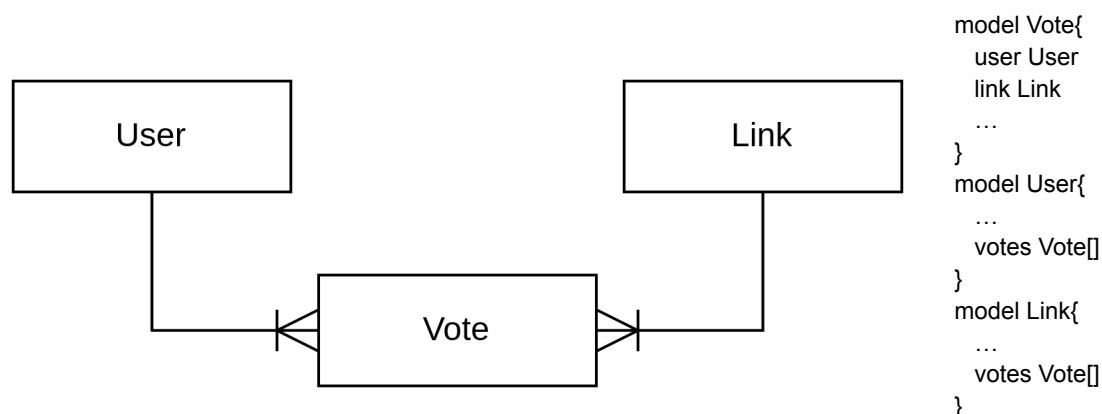


Vemos la información del datos provista por la subscription (resolve). Y el objeto es Link, ya que así lo definimos en el schema de la subscription, (el cual publica el resolver de la mutation donde se crea antes de retornarlo como vimos). Y como siempre, vemos que trae los datos que hemos proyectado al lanzar la subscription (o sea los fields del objeto que queremos).

Agregando la feature de votación

Siguiendo con el tutorial descrito en <https://www.howtographql.com/graphql-js/7-subscriptions> Se agrega esta feature, como repaso del circuito, sin repetir explicaciones.

Un “usuario” puede “crear un voto” por determinado “link”



1- Actualizar nuestro esquema de base según la nueva necesidad (schema.prisma)

```
// DATABASE MODEL
model Link {
  id          Int          @id @default(autoincrement())
  createdAt   DateTime     @default(now())
  description String
  url         String
  postedBy    User?        @relation(fields: [postedById], references: [id])
  postedById  Int?
  votes       Vote[]
}

model User {
  id          Int          @id @default(autoincrement())
  name        String
  email       String       @unique
  password    String
  links       Link[]
  votes       Vote[]
}

model Vote {
  id          Int          @id @default(autoincrement())
  Link        Link?        @relation(fields: [linkId], references: [id])
  linkId      Int?
  User        User?        @relation(fields: [userId], references: [id])
  userId      Int?

  @@unique([linkId, userId])
}
```

Nota1 Si tenemos la extensión de prisma en visual code, al completar el schema para User y Link con vote Vote[], solo nos indica el error, al crear el model Link, autocompleta con un esquema sugerido

Nota 2 La constraint **@@unique**, a parte de garantizar la unicidad, sirve para **findOne** o **findUnique** (según la versión de prisma) por clave, para **updates** y **upserts**, ya que estos últimos sólo se pueden hacer mediante primary key o uniques. Es muy útil en algunos casos, se debe establecer en la sección where del filtro nombre_constraint:{<condicion>} (no lo veremos en este taller

(el nombre por default puede variar según el motor de base de datos)

Por ejemplo:

```
await prisma.vote.findUnique({
  where: {
    linkId_userId: { // nombre de la propiedad del atributo @@unique default link_user
      linkId: 3
      userId: 4
    },
  },
})
```

Nota 3 Cada constraint `@unique`, crea una constraint en la base de datos destino y la nombra por default en caso de no establecerse un nombre, tanto Prisma como algunas versiones del motor de base de datos usado (MySQL, PostgreSQL, etc., pueden tener ciertas restricciones como ser la cantidad máximo de caracteres que se pueden emplear para nombrar una constraint (caso MySQL), o tal vez sea necesario añadir otra constraint con relaciones hacia las mismas entidades, etc. Para esto `@unique` ofrece opciones como ser la nomenclación.

Para más información de unives con Prisma: [Enlace Prisma Unique](#)

2- Actualizar esquema de la base y prismaClient

- a. Ejecutamos la migration para actualizar el esquema de la base conforme al schema.prisma

```
npx prisma migrate dev --name "add-vote-model"
```

```
npx prisma migrate dev --name "add-vote-mod

Environment variables loaded from .env
Prisma schema loaded from prisma/schema.prisma
Datasource "db": SQLite database "dev.db" at "file:./dev.db"

Applying migration `20221022172456_add_vote_model`

The following migration(s) have been created and applied from new schema changes:

migrations/
├─ 20221022172456_add_vote_model/
├─ migration.sql

Your database is now in sync with your schema.
```

- b. Actualizar nuestra API de Prisma Client para poder usar las operaciones CRUD

```
npx prisma generate
```

Necesitamos actualizar nuestro cliente prisma para que el cliente tenga los cambios de nuestro modelo. Crea las operaciones principales a nivel entidad y actualiza el schema.

```
~/workspace/taller-gql/workspace/gql1_apollo$ npx prisma generate
Environment variables loaded from .env
Prisma schema loaded from prisma/schema.prisma

✓ Generated Prisma Client (4.4.0 | library) to ./node_modules/@prisma/client in 102ms
You can now start using Prisma Client in your code. Reference:
https://pris.ly/d/client
...
import { PrismaClient } from '@prisma/client'
const prisma = new PrismaClient()
...

Update available 4.4.0 -> 4.5.0
Run the following to update
  npm i --save-dev prisma@latest
  npm i @prisma/client@latest
```

Nota: Los 2 pasos anteriores pueden establecerse en un script donde puedan agregarse más instrucciones, como limpieza de la base, etc

3- Modificar el schema.graphql. (Nueva mutation y modificar types)

```
type Mutation {  
  post(url: String!, description: String!): Link!  
  updateLink(id: ID!, url: String, description: String): Link  
  deleteLink(id: ID!): Link  
  signup(email: String!, password: String!, name: String!): AuthPayload  
  login(email: String!, password: String!): AuthPayload,  
  vote(linkId: ID!): Vote  
}
```

```
type Vote {  
  id: ID!  
  link: Link!  
  user: User!  
}
```

```
type Link {  
  id: ID!  
  description: String!  
  url: String!  
  postedBy: User  
  votes: [Vote!]!  
}
```

Observar que estamos estableciendo en el tipo Links votes como parámetro obligatorio en los payloads tanto el Array devuelto como que sea obligatorio que contenga al menos un Vote

4- Implementar las functions resolver

Agregamos nuevo resolver de la nueva mutation en **src/resolvers/Mutation.js**
También vamos a notificar el evento

a. Resolver de la mutation

```
async function vote(parent, args, context, info) {  
  const { userId } = context; //Recuperamos el id del user logueado  
  const linkId = Number(args.linkId); //Argumento de la mutation (ver schema)  
  
  //Hago un findUnique por la constraint unique  
  const vote = await context.prisma.vote.findUnique({  
    where: {  
      linkId_userId: {  
        linkId: linkId,  
        userId,  
      },  
    },  
  });  
}
```

```

    select: { linkId: true }, //No necesito todo, proyecto sólo linkId
  },
});

if (Boolean(vote)) {
  throw new Error(`Already voted for link: ${linkId}`);
}
const newVote = context.prisma.vote.create({
  data: {
    user: { connect: { id: userId } },
    link: { connect: { id: linkId } },
  },
});
context.pubsub.publish("NEW_VOTE", newVote);
return newVote;
}

```

b. Resolvers de tipo

b.1 Agregamos al resolver de tipo Link las relaciones de votes por default

Cuando busquemos un link devolveremos por default sus votos

src/resolvers/Link.js

```

function votes(parent, args, context) {
  return context.prisma.link.findUnique({ where: { id: parent.id } }).votes();
}

```

Otra forma de hacerlo directamente desde los votos:

```

function votes(parent, args, context) {
  return context.prisma.votes.findMany({where: { linkId:parent.id}});
}

```

b.2 Agregamos el resolver tipo Vote con las relaciones a Link y User

- Creamos **src/resolvers/Vote.js**

```

function link(parent, args, context) {
  return context.prisma.vote.findUnique({ where: { id: parent.id } }).link();
}

function user(parent, args, context) {
  return context.prisma.vote.findUnique({ where: { id: parent.id } }).user();
}

module.exports = {
  link,
  user,
};

```

- c. Agregar los nuevos files de Resolver al index

```
const Vote = require("../resolvers/Vote");
```

...

```
const resolvers = {  
  Query,  
  Mutation,  
  User,  
  Link,  
  Subscription,  
  Vote //Agregar a resolvers  
};
```

5- Suscribirse a los nuevos votos

- a. Agregar al **schema.graphql**

```
type Subscription {  
  newLink: Link  
  newVote: Vote  
}
```

- b. Agregar wrapper y resolver en **Subscription.js**

```
//resolver  
async function newVoteSubscribe(parent, args, context, info) {  
  return context.pubsub.asyncIterator("NEW_VOTE");  
}  
  
//wrapper  
const newVote = {  
  subscribe: newVoteSubscribe,  
  resolve: (payload) => {  
    return payload;  
  },  
};  
  
module.exports = {  
  newLink,  
  newVote //exportar wrapper  
};
```


5- Probar la las mutations y subscriptions

Nos subscribimos

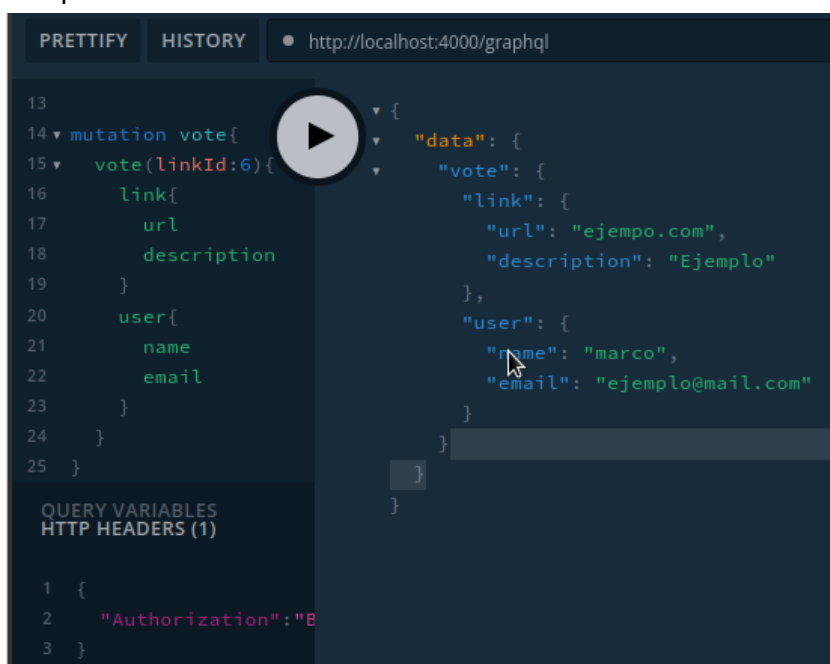
```
subscription newVote{
  newVote{
    id
    link{
      id
      description
      url
      postedBy{
        name
      }
    }
  }
}
```

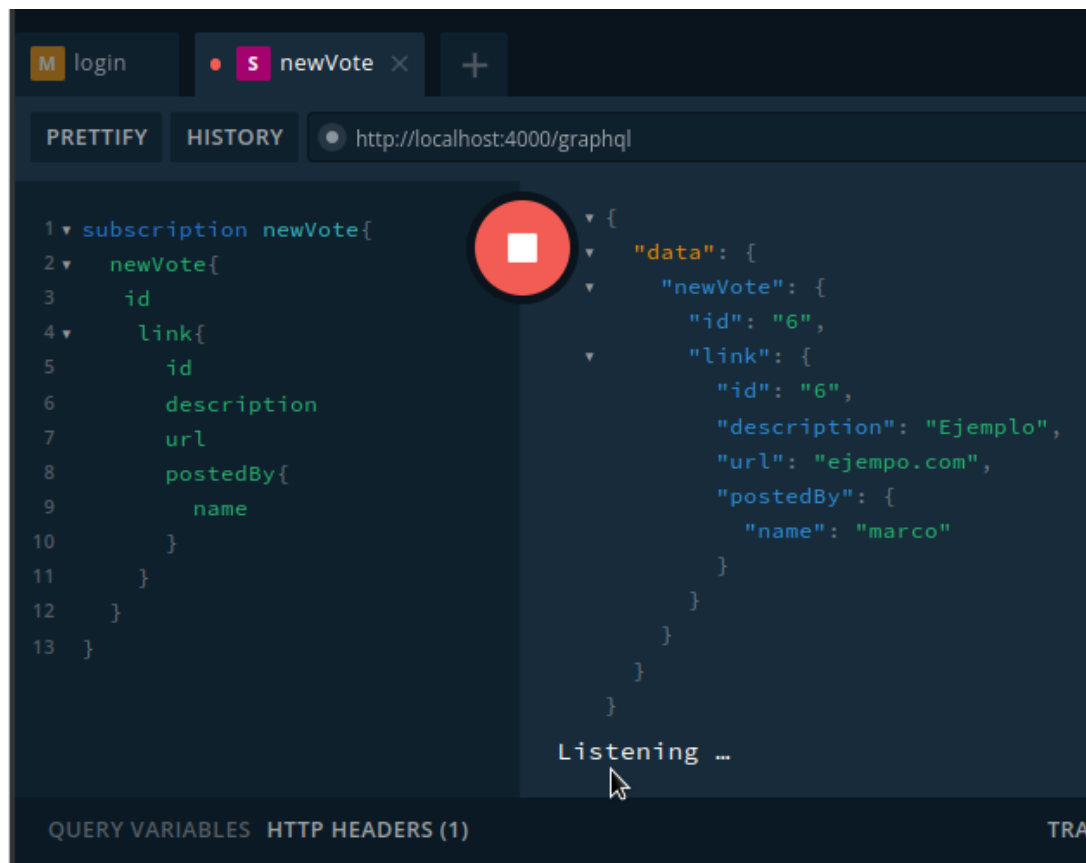
Nos logueamos y votamos por algún link creado

```
mutation vote{
  vote(linkId:6){
    link{
      url
      description
    }
    user{
      name
      email
    }
  }
}
```

Tenemos que ver tanto el response en la solapa de la mutation como el payload enviado al subscriber al ejecutarse el evento, en la solapa donde nos subscribimos

Solapa de mutation





Observar que sigue “**Listening ...**” indicando que sigue escuchando eventos **newVote**

Filtrado, Paginación y Ordenamiento

Link: <https://www.howtographql.com/graphql-js/8-filtering-pagination-and-sorting/>

Prisma Client ya nos ofrece la opción de filtrado, ordenamiento y paginación. La idea es aprovechar estas características en la medida de lo posible

Nota: Existen situaciones donde se requieren combinar filtrados en base con filtrado y paginación en memoria, armar nuevos tipos a nivel graphql no necesariamente existentes en la base, ejecutar consulta complejas a prismaClient e incluso \$rawQuery en lenguaje nativo del motor de base desde prisma client (fuera del alcance de este taller). Comunmente son situaciones excepcionales, por ejemplo para reportes, búsquedas rápidas genéricas, etc.

Para el ejemplo vamos a modificaremos la query feed

Filtrado

Vamos a establecer un filtro básico “filter” que reciba String. Los filtros pueden ser por fields específicos que queramos, recibir un tipo input determinado.

1. En schema.graphql modificamos los argumentos de la query feed de modo que quede de la siguiente manera:

```

type Query {
  info: String!
  feed(filter:String!): [Link!]!
  link(id: ID!): Link
}

```

2. Modificamos resolver

Si viene algo en el filtro revisaremos si algo del texto está en description o la url de link, sino retornar todo

```

async function feed(parents, args, context) {
  const where = args.filter
    ? {
      OR: [
        { description: { contains: args.filter } },
        { url: { contains: args.filter } },
      ],
    }
    : {};

  return await context.prisma.link.findMany({
    where,
  });
}

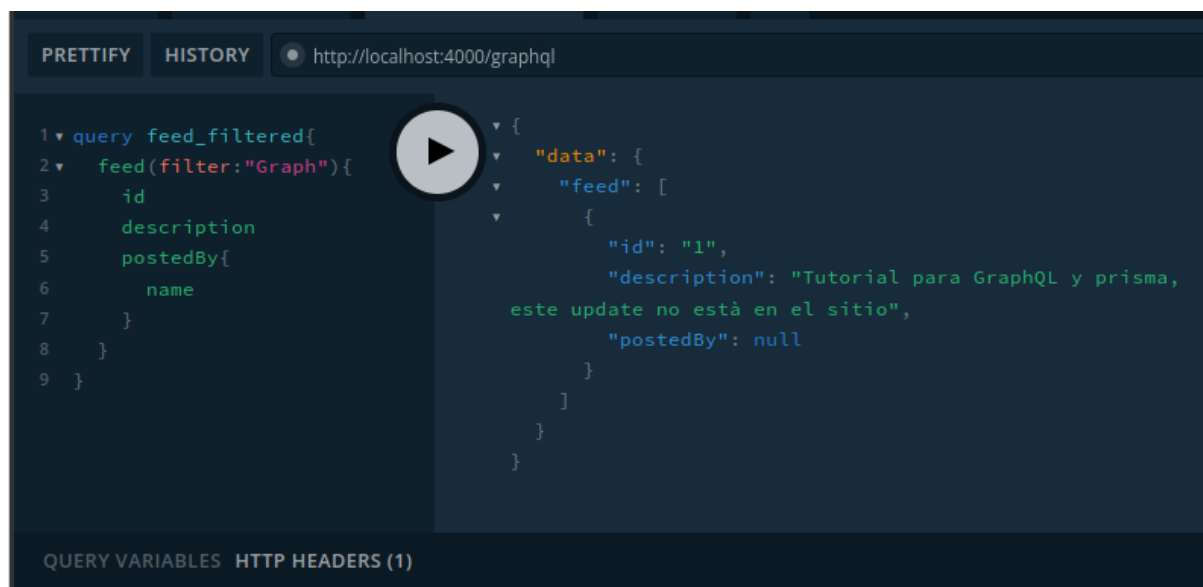
```

3. Prueba:

```

query feed_filtered{
  feed(filter:"Graph"){
    id
    description
    postedBy{
      name
    }
  }
}

```



Paginación

Es posible traer los datos paginados desde la base mediante las opciones de Prisma **take** y **skip**. Prisma soporta 2 modelos de paginación:

Limit-Offset

Solicitar una porción específica de la lista proporcionando los índices de los ítems a ser recibidos, un offset con un limit (es el que se implementará)

Cursor-based

Este modelo de paginación es un poco más avanzado asociado a un único id (cursor).

Para limit, en prisma usamos **take** (para tomar una cantidad de elementos)

Para indicar el “start index”, es decir a partir de qué índice queremos comenzar, usamos **skip** para que “salte” los no deseados. Su valor por default es 0

Es importante tener en cuenta la paginación, porque según la versión de prisma, una query tiene un máximo de elementos que puede retornar.

Ver más de paginación en Prisma en este [enlace](#)

1. Agregamos skip y take al schema

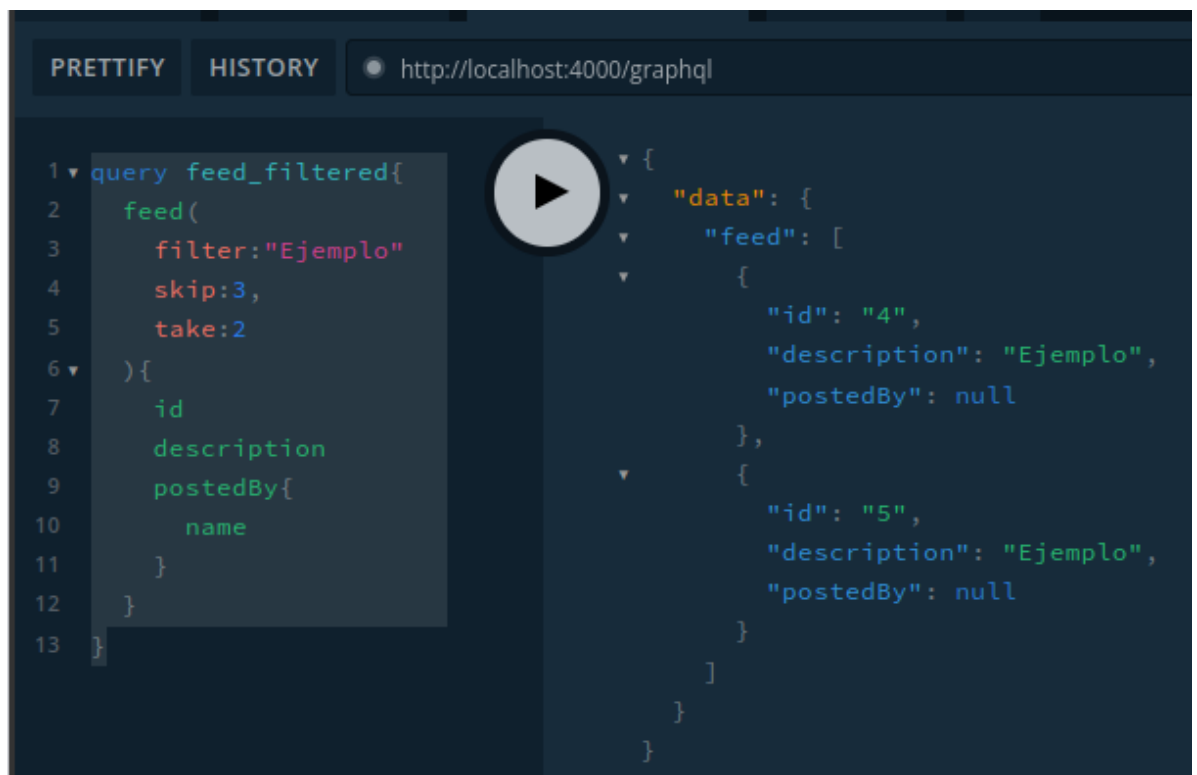
```
type Query {  
  info: String!  
  feed(filter: String, skip: Int, take: Int): [Link!]!  
  link(id: ID!): Link  
}
```

2. Modificamos el resolver

```
async function feed(parents, args, context) {  
  const { skip, take } = args;  
  const where = args.filter  
    ? {  
      OR: [  
        { description: { contains: args.filter } },  
        { url: { contains: args.filter } },  
      ],  
    }  
    : {};  
  
  return await context.prisma.link.findMany({  
    where,  
    skip,  
    take,  
  });  
}
```

3. Prueba: Filtrar según los elementos que hayamos insertado, podemos no filtrar

```
query feed_filtered{
  feed(
    filter:"Ejemplo"
    skip:3,
    take:2
  ){
    id
    description
    postedBy{
      name
    }
  }
}
```



Ordenamiento (sorting)

Link: <https://www.howtographql.com/graphql-js/8-filtering-pagination-and-sorting/>

Como vimos en [Mutations Inputs y Tipos](#), es posible definir un tipo input en el schema GraphQL, así mismo es posible definir enums.

Ya que Prisma permite el ordenamiento de elementos según un criterio específico, a través del field **orderBy**, podemos ya definir dicho criterio en el schema.graphql

1. Crearemos nuestro propio criterio de ordenamiento con el cual podrá ordenar el cliente en `schema.graphql`

```
input LinkOrderByInput {
  description: Sort
  url: Sort
  createdAt: Sort
}

enum Sort {
  asc
  desc
}
```

2. Añadimos el nuevo criterio a nuestra definición de query

```
type Query {
  info: String!
  feed(filter: String, skip: Int, take: Int, orderBy: LinkOrderByInput): [Link!]!
  link(id: ID!): Link
}
```

3. Añadir el criterio al resolver para que sea pasado a Prisma en el argumento **orderBy**

```
async function feed(parents, args, context) {
  const { skip, take, orderBy } = args;
  const where = args.filter
    ? {
        OR: [
          { description: { contains: args.filter } },
          { url: { contains: args.filter } },
        ],
      }
    : {};

  return await context.prisma.link.findMany({
    where,
    skip,
    take,
    orderBy,
  });
}
```

4. Ejemplo

```
query feed_filtered{
  feed(
    take:4
    orderBy:{
      description:desc
    }
  ){
    id
    description
    postedBy{
      name
    }
  }
}
```

Retornar la cantidad total de elementos en la query paginadas

Para este ejemplo vamos a crear un nuevo tipo (siguiendo el ejemplo lo llamaremos Feed, pero otro nombre apropiado podría ser PaginatedLinks).

Esto suele ser necesario para que un cliente (por ejemplo un front) pueda mostrar los datos paginados y el total de páginas, selección, etc.

Es muy importante que a la hora de establecer el count, **el criterio de búsqueda sea el mismo**

1. Agregaremos el nuevo tipo a nuestro schema.graphql y modificamos el tipo devuelto en la definición de query

```
type Feed {
  links: [Link!]!
  count: Int!
}
```

```
type Query {
  info: String!
  feed(filter: String, skip: Int, take: Int, orderBy: LinkOrderByInput): Feed!
  link(id: ID!): Link
}
```

Nota: observar que los datos de links estarán dentro de un feed, es decir la query no retorna un array sin un Feed que tiene el array de link y el count

2. Modificar el resolver de la query

```
async function feed(parents, args, context) {
  const { skip, take, orderBy } = args;
  const where = args.filter
  ? {
    OR: [
      { description: { contains: args.filter } },
      { url: { contains: args.filter } },
    ],
  }
  : {};
}
```

```

const links = await context.prisma.link.findMany({
  where,
  skip,
  take,
  orderBy,
});

const count = await context.prisma.link.count({
  where,
});

return {
  links,
  count: count,
};
}

```

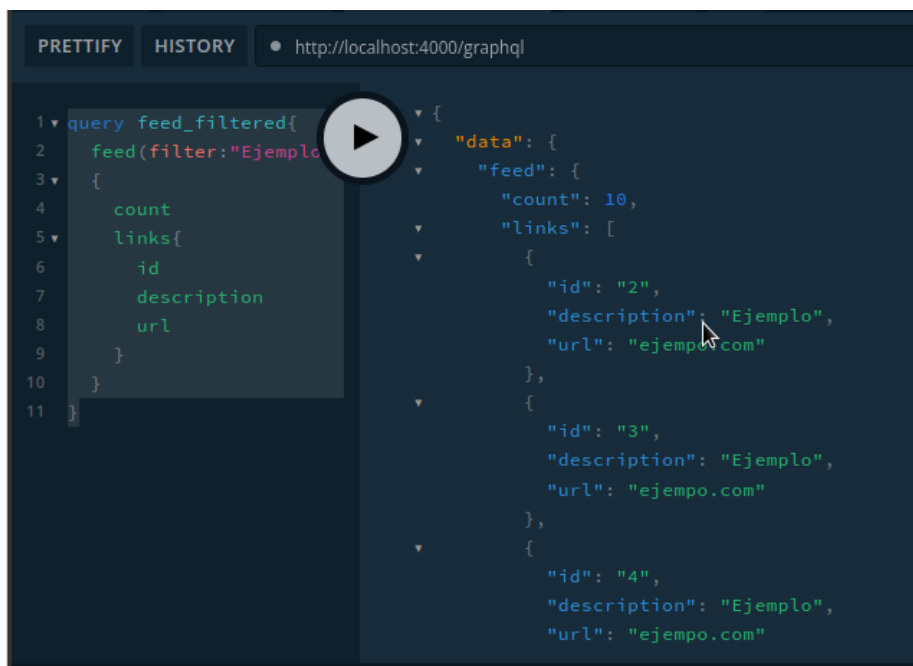
3. Prueba

Ahora nuestra definición cambió, por lo que debemos solicitar los elementos “del grafo” que requerimos acorde al nuevo schema de la query

```

query feed_filtered{
  feed(filter:"Ejemplo")
  {
    count
    links{
      id
      description
      url
    }
  }
}

```



Enlaces

Github Código ejemplo y explicaciones del taller <https://github.com/marbriganti/taller-gql>

Sitio GraphQL: <https://graphql.org>

Github Código fuente dell sitio howtographql completo: <https://github.com/howtographql/graphql-js>

Sitio npm jsonwebtoken: <https://www.npmjs.com/package/jsonwebtoken>

Github node-jsonwebtoken: <https://github.com/auth0/node-jsonwebtoken>

JWT libraries: <https://jwt.io/libraries>

WebSocket: <https://en.wikipedia.org/wiki/WebSocket>

Prisma: <https://www.prisma.io/docs/reference>

Bases de datos soportadas por Prisma: <https://www.prisma.io/docs/concepts/database-connectors>