CODER HOUSE

Clase 44. Programación Backend

GRAPHQL

OBJETIVOS DE LA CLASE

- Conocer y aprender sobre GraphQL.
- Identificar las diferencias entre GraphQL y REST.
- Desarrollar una API utilizando GraphQL.
- Aprender sobre la herramienta GraphiQL.

CODER HOUSE

CRONOGRAMA DEL CURSO

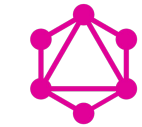
Clase 43	Clase 44	Clase 45
Documentación de APIs	**GRAPHQL**	**Introducción a frameworks de desarrollo backend - Parte I**

CODER HOUSE

GRAPHQL

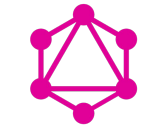
CODER HOUSE

DE QUÉ SE TRATA?



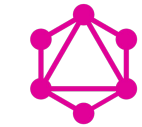
¿De qué se trata?

- GraphQL fue creada por Facebook como una alternativa a las API REST.
- Es un *lenguaje de consulta* y un tiempo de *ejecución* del servidor para las interfaces de programación de aplicaciones (API). Su función es brindar a los clientes exactamente los datos que solicitan y nada más.
- Los desarrolladores pueden realizar:
  - Consultas GraphQL: permiten consumir datos, especificando cuáles y cómo se desea recibirlos.
  - Mutaciones GraphQL: permiten escribir o modificar datos en el servidor.
- Con GraphQL, las API son rápidas, flexibles y sencillas para los desarrolladores. Incluso se pueden implementar en un IDE conocido como GraphiQL.



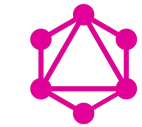
Esquemas

- Los desarrolladores de API utilizan GraphQL para crear un esquema que describa todos los datos posibles que los clientes pueden consultar a través del servicio.
- Un **esquema** de GraphQL está compuesto por tipos de objetos, que definen qué clase de objetos puede solicitar y cuáles son sus campos.
- A medida que ingresan las consultas, GraphQL las aprueba o rechaza en función del esquema, y luego ejecuta las validadas.
- El desarrollador de API adjunta cada campo de un esquema a una función llamada resolución. Durante la ejecución, se llama a la resolución para que genere el valor.



Ventajas en entorno empresarial

- Un esquema de GraphQL establece una fuente única de información en una aplicación de GraphQL. Ofrece a la empresa una forma de unificar toda su API.
- Las llamadas a GraphQL se gestionan en un solo recorrido de ida y vuelta. Los clientes obtienen lo que solicitan sin que se genere una sobrecarga.

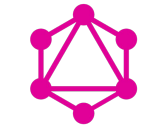


Ventajas en entorno empresarial

- Los tipos de datos bien definidos reducen los problemas de comunicación entre el cliente y el servidor.
- GraphQL es una herramienta introspectiva. Un cliente puede solicitar una lista de los tipos de datos disponibles. Esto es ideal para la generación automática de documentación.

Ventajas en entorno empresarial

- GraphQL permite que las API de las aplicaciones evolucionen sin afectar las consultas actuales.
- Muchas extensiones de GraphQL open source ofrecen características que no están disponibles con las API REST.
- GraphQL no exige una arquitectura de aplicación específica. Puede incorporarse sobre una API REST actual y funcionar con una herramienta de gestión de API también actual.



Desventajas en entorno empresarial

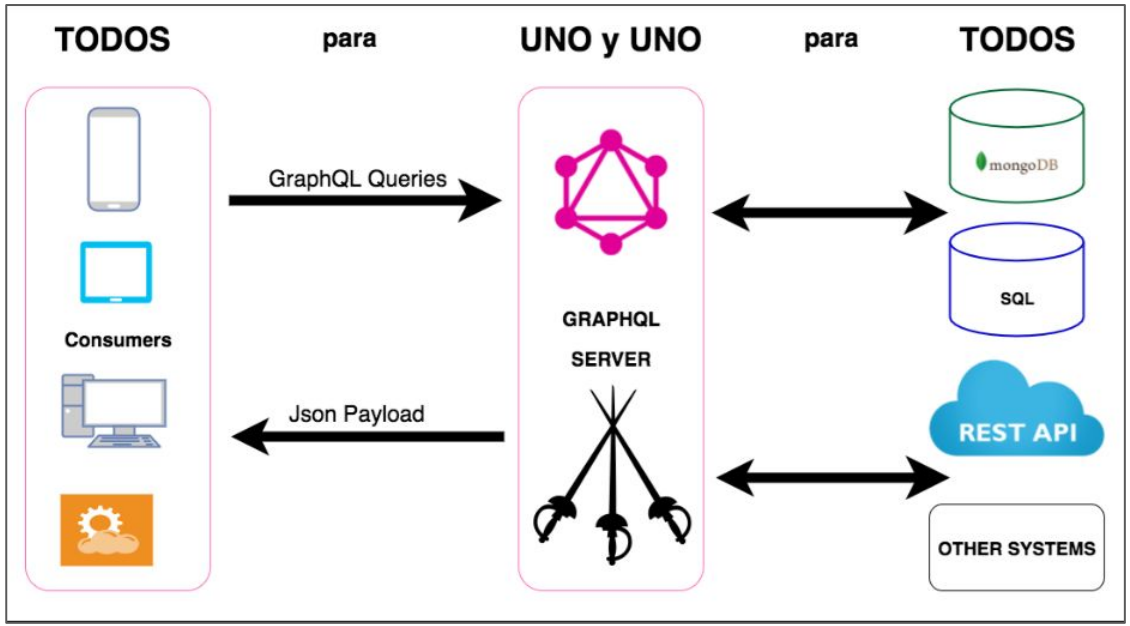
- GraphQL presenta una curva de aprendizaje para desarrolladores que tienen experiencia con las API REST.
- Delega gran parte del trabajo de las consultas de datos al servidor, lo cual representa una mayor complejidad para los desarrolladores de servidores.

Desventajas en entorno empresarial

- Según su implementación, GraphQL podría requerir estrategias de gestión de API diferentes a las de API REST, sobre todo si se tienen en cuenta los precios y los límites de frecuencia.
- El almacenamiento en caché es más complejo que con REST.
- Los encargados del mantenimiento de las API tienen la tarea adicional de escribir un esquema de GraphQL fácil de mantener.

Todos para uno y uno para todos

Esta conocida frase en cierto modo define una de las bases de GraphQL, la existencia de un único punto de acceso a los datos como vemos en el diagrama.



Todos para uno y uno para todos

Todos para uno:

- El “todos” serían los consumidores que acceden a las operaciones mediante un único punto de entrada, sea cual sea el tipo de integración que se haga.
- El “uno” sería nuestro GraphQL server.

Todos para uno y uno para todos

Uno para todos:

- El “Uno” sería nuestro GraphQL Server.
- El “Todos” serían todos los subsistemas a los cuales ha de acceder el GraphQL Server para resolver las consultas y operaciones requeridas por el usuario.

COMPARACIÓN REST - GRAPHQL

CODER HOUSE

Características de REST

- **Protocolo cliente/servidor sin estado**: Las peticiones HTTP contienen toda la información necesaria, por lo que ni el cliente ni el servidor necesitan recordar ningún estado previo para responder a la petición.
- **Se apoya sobre el protocolo HTTP:** HTTP permite realizar una serie de operaciones bien definidas entre las que están: POST, GET, PUT y DELETE.
- **La manipulación de los objetos es a partir de URI:** La URI es el identificador único de cada recurso en REST. Es la que permite, además, acceder a los recursos y manipularlos.
- **Sistema de capas:** Los componentes se organizan siguiendo una arquitectura jerárquica. Cada capa tiene una función dentro del sistema y el cliente solo debe acceder a la capa con la que interactúa.

Similitudes entre REST y GraphQL

- Ambos posibilitan intercambiar datos entre el cliente y el servidor en diferentes formatos, JSON es el formato predeterminado para ambos.
- La implementación del lado del servidor puede hacerse con cualquier lenguaje, ambos funcionan independientemente del lenguaje escogido.
- Dan la libertad de implementar el frontend en cualquier lenguaje, el consumo de datos también es indiferente al lenguaje escogido para la implementación de la interfaz de la aplicación.
- Solo se limitan a definir las peticiones y la forma en que es devuelta la información, no almacenan datos de los clientes.

Diferencias entre REST y GraphQL

Característica	REST	GraphQL
Definición	Es un estilo de arquitectura de software.	Es un lenguaje de consulta y manipulación de datos.
Respuesta del servidor	Puede hacer overfetching, es decir enviar más información de la que necesita.	Envía solo lo necesario: controlan los datos que deben ser enviados desde el servidor.
Obtención de datos	El servidor expone los datos, los clientes deben adecuarse a la forma en que están representados.	Los clientes definen la estructura de los datos que reciben como respuesta del servidor.
Peticiones	Hace múltiples peticiones por vista lo que disminuye el rendimiento.	Hace una sola petición por vista, y en esta se pueden obtener todos los datos necesarios.
Almacenamiento en caché	implementa almacenamiento en caché para evitar repetir búsquedas de un mismo recurso.	El almacenamiento en caché es responsabilidad de los clientes.
Versionado de una API	Para dar soporte a nuevas versiones de un API generalmente se deben crear nuevos endpoints.	El cambio de la versión del API no afecta, ya que se pueden quitar o adicionar campos modificando la consulta.

Resumiendo

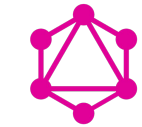
- GraphQL surge principalmente para solucionar problemas de REST.
- Ambas son de las formas más usadas para el diseño del funcionamiento de un API y la forma en que se accederá a los datos.
- GraphQL ofrece mayor flexibilidad gracias a sus con sus consultas, esquemas y solucionadores, además de un mejor rendimiento.
- Si nuestras necesidades son implementar y usar de forma fácil una API conviene elegir GraphQL. El desarrollo con el mismo es más sencillo, por lo que podemos acortar los tiempos de implementación. Si usamos microservicios en el backend de la aplicación, REST es más recomendable para este propósito.

Resumiendo

- A pesar de ser más eficiente realizando las búsquedas y obteniendo los datos, podemos ver afectado el rendimiento al usar GraphQL si no implementamos el almacenamiento en caché en los casos necesarios (en REST viene integrado).
- GraphQL está centrado en mejorar la capacidad de desarrollo de APIs y su adecuación al uso según las necesidades del cliente, agiliza el desarrollo y disminuye las modificaciones ante cambios realizados. Además, su mantenimiento es menos costoso que una API implementada con REST.
- Siempre debemos analizar con detenimiento los requisitos de la aplicación, el rendimiento y otros factores para escoger correctamente cómo vamos a implementar nuestra API.

IMPLEMENTACIÓN DE UNA API GRAPHQL

CODER HOUSE



Usando GraphQL en nuestra App

- Primero creamos un proyecto de Node, y creamos el servidor en el archivo *server.js*.
- Luego, instalamos los módulos ***graphql*** y ***express-graphql***:

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

- En el *server.js* requerimos además de express, el método ***buildSchema*** del módulo *graphql*, el método ***graphqlHTTP*** del módulo *express-graphql*, y la librería *crypto* para generar ids aleatorios:

```
import express from 'express';
import { graphqlHTTP } from 'express-graphql';
import { buildSchema } from 'graphql';
import crypto from 'crypto';
```

Usando GraphQL en nuestra App

```
const schema = buildSchema(`
  type Persona {
    id: ID!
    nombre: String,
    edad: Int
  }
  input PersonaInput {
    nombre: String,
    edad: Int
  }
  type Query {
    getPersona(id: ID!): Persona,
    getPersonas(campo: String, valor: String): [Persona],
  }
  type Mutation {
    createPersona(datos: PersonaInput): Persona
    updatePersona(id: ID!, datos: PersonaInput): Persona,
    deletePersona(id: ID!): Persona,
  }
`);
```

- En el *server.js* también creamos el esquema de GraphQL.
- Lo utilizamos para describir el sistema completo de tipos de API.
- Incluye el conjunto completo de datos y define cómo un cliente puede acceder a esos datos.
- En este caso, estos tipos son Query, Mutation, Persona, y PersonaInput (que es un DTO de persona sin ID).

Usando GraphQL en nuestra App

```
class Persona {
    constructor(id, { nombre, edad }) {
        this.id = id;
        this.nombre = nombre;
        this.edad = edad;
    }
}

const personasMap = {};
```

Como modelo usaremos una Persona, con nombre y edad, para mantener al ejemplo bien sencillo.

Para la persistencia usaremos un objeto simple, como diccionario.

Usando GraphQL en nuestra App

Definimos nuestras funciones de manejo de persistencia (que operan sobre el diccionario de personas):

```
function getPersonas({ campo, valor }) {
    const personas = Object.values(personasMap)
    if (campo && valor) {
        return personas.filter(p => p[ campo ] == valor);
    } else {
        return personas;
    }
}

function getPersona({ id }) {
    if (!personasMap[ id ]) {
        throw new Error('Persona not found.');
    }
    return personasMap[ id ];
}

function createPersona({ datos }) {
    const id = crypto.randomBytes(10).toString('hex');
    const nuevaPersona = new Persona(id, datos)
    personasMap[ id ] = nuevaPersona;
    return nuevaPersona;
}
```

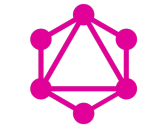
```
function updatePersona({ id, datos }) {
    if (!personasMap[ id ]) {
        throw new Error('Persona not found');
    }
    const personaActualizada = new Persona(id, datos)
    personasMap[ id ] = personaActualizada;
    return personaActualizada;
}

function deletePersona({ id }) {
    if (!personasMap[ id ]) {
        throw new Error('Persona not found');
    }
    const personaBorrada = personasMap[ id ]
    delete personasMap[ id ];
    return personaBorrada;
}
```

Usando GraphQL en nuestra App

```
const app = express();

app.use(express.static('public'));

app.use('/graphql', graphqlHTTP({
   schema: schema,
   rootValue: {
       getPersonas,
       getPersona,
       createPersona,
       updatePersona,
       deletePersona
   },
   graphiql: true,
}));

const PORT = 8080
app.listen(PORT, () => {
   const msg = `Servidor corriendo en puerto:
${PORT}`;
   console.log(msg)
});
```

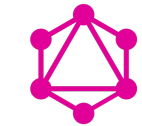
Finalmente cargamos nuestras funciones en nuestro middleware de graphql, haciéndolas disponibles en la ruta ***/graphql***.

CODER HOUSE



Usando GraphQL en nuestra App

- En este middleware pasamos como argumento un objeto con 3 propiedades:
  - ***schema***: el esquema GraphQL que creamos al principio.
  - ***rootValue***: El “root resolver” que contiene las funciones a publicar.
  - ***graphiql***: Si está en true, habilita la herramienta GraphiQL (que veremos más adelante) al acceder al endpoint desde un navegador.



Usando GraphQL en nuestra App

- Ya podemos iniciar nuestro servidor y empezar a enviarle consultas en formato GraphQL al endpoint creado.
- Para probar nuestras diversas consultas, utilizaremos un formulario html con un script que utiliza Fetch para enviar nuestras consultas al servidor.

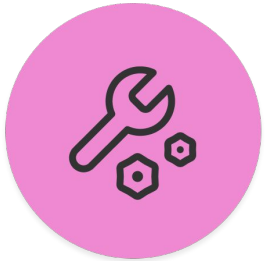
Usando GraphQL en nuestra App

Ahora podemos ejecutar (entre otras) las siguientes dos consultas de creación y lectura:

```
mutation {
 createPersona(datos: {
   nombre: "marian",
   edad: 35
 }) {
   id
 }
}
```

```
query {
 getPersonas {
   nombre
   edad
 }
}
```

Con la primera agregamos una persona, y vemos el id como resultado, y con la segunda vemos nombre y edad de las personas existentes.



SERVIDOR API GRAPHQL

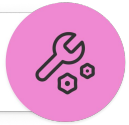
Tiempo: 10 minutos

CODER HOUSE

Servidor API GraphQL

Tiempo: 10 minutos

Desafío generico



Crear un servidor basado en GraphQL que me permita administrar recordatorios.

El mismo deberá permitirme realizar las siguientes operaciones:

- Agregar un recordatorio
- Ver todos los recordatorios

Un recordatorio está compuesto por un título, una descripción, y un timestamp del momento en que fue creado.

Verificar el correcto funcionamiento del servidor utilizando un script con fetch desde el navegador (se puede usar el ejemplo de clase).

No es necesario realizar un servidor en capas, y se puede mantener la persistencia en memoria para simplificar el desafío y concentrarse en la parte de GraphQL.

CODER HOUSE

BREAK

¡5/10 MINUTOS Y VOLVEMOS!

GRAPHiQL

¿De qué se trata?

- GraphiQL es el entorno de desarrollo integrado (IDE) de GraphQL. Es un editor interactivo para construir consultas y explorar la GraphQL API.
- Una de sus mayores ventajas es que ofrece asistencia contextual y proporciona mensajes de error en caso de que la sintaxis de la consulta sea errónea.
- Está basado en JavaScript, se ejecuta en el navegador y para su funcionamiento solo hay que proporcionarle el endpoint de la API a probar.

¿Para qué se utiliza?

- El propósito de este GraphiQL es darle a la Comunidad GraphQL:
  - un servicio de idioma oficial según las especificaciones,
  - un servidor LSP completo y un servicio CLI para usar con IDE,
  - un modo de espejo codificado,
  - un ejemplo de cómo utilizar este ecosistema con GraphiQL,
  - ejemplos de cómo implementar o extender GraphiQL.
- Su uso es similar al de Postman para REST, pero nos sirve para probar las GraphQL APIs.

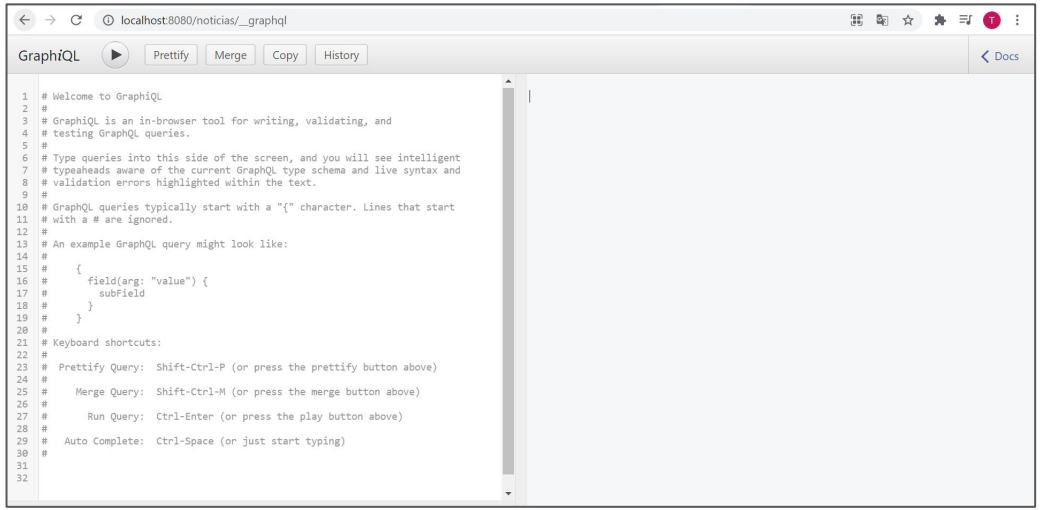
Características

- **Documentación**: El panel de la derecha existe para que exploremos las posibles consultas, mutaciones, campos, etc. Incluso si el servidor no implementa descripciones creadas por humanos, siempre podremos explorar el gráfico de posibilidades.
- **Debugging**: GraphiQL admite la depuración a medida que escribimos, dando pistas y señalando errores.
- **Visor JSON**: Las respuestas de GraphQL no tienen que ser JSON, pero se prefiere. GraphiQL viene con un visor JSON con todas las sutilezas que esperaríamos: plegado de código, sangría automática, soporte de copia y solo lectura.
- **Compartir**: Cuando editamos una consulta, la URL se actualiza inmediatamente. Todo se conserva: espacios en blanco, comentarios, incluso consultas no válidas. Se puede compartir fácilmente esta URL con colegas o de forma pública.

Probar nuestra GraphQL API

Ejemplo en vivo

- Para empezar a usar GraphiQL para probar nuestra API, primero prendemos nuestro servidor en el ambiente de desarrollo, con el comando `npm start` o `npm run dev`.
- Luego, vamos al navegador, e ingresamos a la url http://localhost:8080/graphql. Se nos abre entonces el GraphiQL para poder empezar a probar los endpoints.

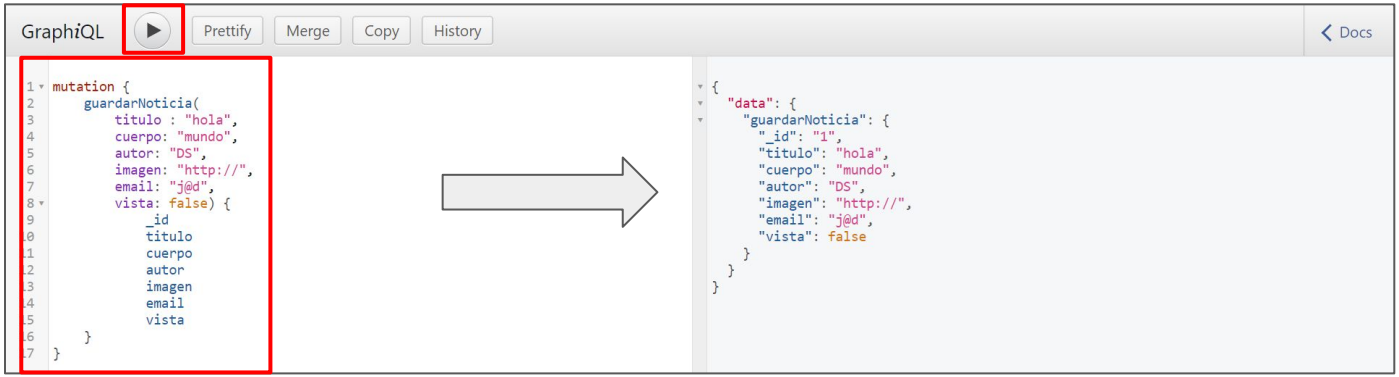


CODER HOUSE

Probar nuestra GraphQL API

Ejemplo en vivo

- Vamos primero a probar el endpoint por POST para guardar una nueva noticia.
- Para eso, en la consola del lado izquierdo de GrapihQL escribimos la Query que vemos en la imagen. Le podemos poner los valores que queramos a cada variable.
- Luego, hacemos click en el botón de play y se va a ejecutar la Query, obteniéndose el resultado en la parte derecha de la pantalla.

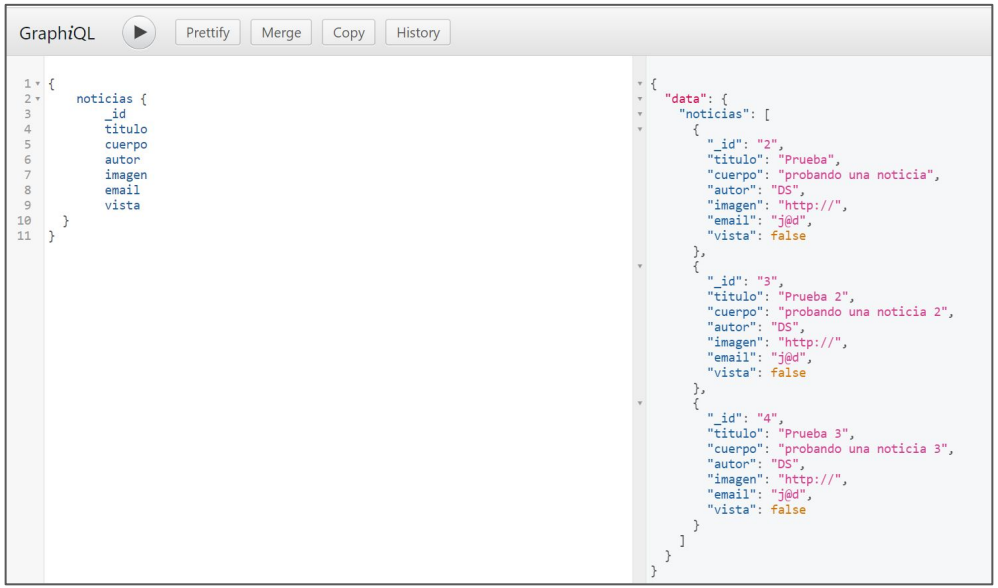


CODER HOUSE

Probar nuestra GraphQL API

Ejemplo en vivo

- Con la siguiente Query, obtenemos todas las noticias que tengamos almacenadas.
- De la misma forma que antes, escribimos la Query, clickeamos *play* y obtenemos la respuesta del lado derecho.



CODER HOUSE

Probar nuestra GraphQL API

Ejemplo en vivo

- Para obtener una noticia por su id la query es la siguiente:

GraphiQL | Prettify | Merge | Copy | History

```
query{
     noticias(_id: "2") {
        _id
        titulo
        cuerpo
        autor
        imagen
        email
        vista
    }
}
```

```
{
  "data": {
    "noticias": [
      {
        "_id": "2",
        "titulo": "Prueba",
        "cuerpo": "probando una noticia",
        "autor": "DS",
        "imagen": "http://",
        "email": "j@d",
        "vista": false
      }
    ]
  }
}
```

- Es igual que la anterior, pero le agregamos el id que queremos que traiga.

Probar nuestra GraphQL API

- Para actualizar y marcar como leída una noticia, la query queda como la siguiente:

GraphiQL ▶ Prettify Merge Copy History

```
mutation {
    actualizarNoticia(_id:"3",vista: true) {
        _id
        titulo
        cuerpo
        autor
        imagen
        email
        vista
    }
}
```

```
{
  "data": {
    "actualizarNoticia": {
      "_id": "3",
      "titulo": "Prueba 2",
      "cuerpo": "probando una noticia 2",
      "autor": "DS",
      "imagen": "http://",
      "email": "j@d",
      "vista": true
    }
  }
}
```

- Vemos en la respuesta a la derecha, que todo sigue igual y lo único que cambió fue *vista* a *true*.

Probar nuestra GraphQL API

Ejemplo en vivo


```
{
  "data": {
    "noticias": [
      {
        "_id": "3",
        "titulo": "Prueba 2",
        "cuerpo": "probando una noticia 2",
        "autor": "DS",
        "imagen": "http://",
        "email": "j@d",
        "vista": true
      },
      {
        "_id": "4",
        "titulo": "Prueba 3",
        "cuerpo": "probando una noticia 3",
        "autor": "DS",
        "imagen": "http://",
        "email": "j@d",
        "vista": false
      },
      {
        "_id": "1",
        "titulo": null,
        "cuerpo": null,
        "autor": null,
        "imagen": null,
        "email": null,
        "vista": true
      }
    ]
  }
}
```


- Finalmente, para eliminar una noticia, la query queda como:

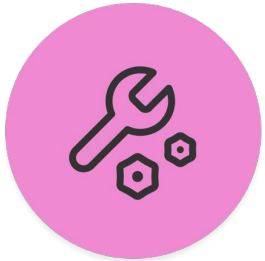
GraphiQL ▶ Prettify Merge Copy History

```
mutation {
    borrarNoticia(_id:"2"){
        _id
        titulo
        cuerpo
        autor
        imagen
        email
        vista
    }
}
```


```
{
  "data": {
    "borrarNoticia": {
      "_id": "2",
      "titulo": "Prueba",
      "cuerpo": "probando una noticia",
      "autor": "DS",
      "imagen": "http://",
      "email": "j@d",
      "vista": true
    }
  }
}
```


Si ejecutamos nuevamente la query para obtener todas las noticias almacenadas, vemos que ya no está la que borramos (_id = 2).

CODER HOUSE



SERVIDOR API GRAPHQL (continuación)

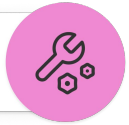
Tiempo: 10 minutos

CODER HOUSE

Servidor API GraphQL

Tiempo: 10 minutos

Desafío generico

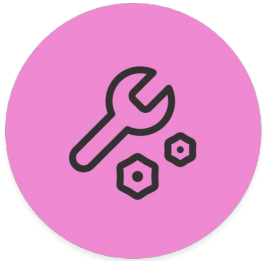


Continuando con el desafío anterior, agregar lo necesario para que nuestro servidor soporte también las siguientes operaciones:

- Marcar un recordatorio como leído
- Eliminar todos los recordatorios marcados como leídos

Verificar el correcto funcionamiento del servidor utilizando la herramienta GraphiQL

CODER HOUSE



SERVIDOR MVC CON GRAPHQL

Tiempo: 15 a 20 minutos

CODER HOUSE

SERVIDOR MVC CON GRAPHQL

Desafío generico

Tiempo: 15 a 20 minutos

Realizar las modificaciones para que el servidor desarrollado en los desafíos anteriores presente una correcta separación en capas. Esto debe incluir:

- La capa de ruteo (donde se levanta el servidor express)
- Los controladores (quienes conectan los root resolvers con los métodos de la api)
- La lógica de negocio (donde se realizan las validaciones y se interactua con la capa de persistencia)
- Los modelos (dentro de ellos se realizan las validaciones)
- La capa de persistencia (utilizando adecuadamente DAOs y DTOs, según corresponda, pero persistiendo aún en memoria)

REFORMAR PARA USAR GRAPHQL

Refactoricemos el código del proyecto que venimos trabajando para cambiar de API RESTful a GraphQL API

CODER HOUSE

REFORMAR PARA USAR GRAPHQL

Formato: link a un repositorio en Github con el proyecto cargado.
Sugerencia: no incluir los node_modules

Desafío entregable

>> Consigna:

- En base al último proyecto entregable de servidor API RESTful, reformar la capa de routeo y el controlador para que los requests puedan ser realizados a través del lenguaje de query GraphQL.
- Si tuviésemos un frontend, reformarlo para soportar GraphQL y poder dialogar apropiadamente con el backend y así realizar las distintas operaciones de pedir, guardar, actualizar y borrar recursos.
- Utilizar GraphiQL para realizar la prueba funcional de los querys y las mutaciones.

¿PREGUNTAS? 

¡MUCHAS GRACIAS!

Resumen de lo visto en clase hoy:

- GraphQL
- Comparación entre GraphQL y REST
- GraphiQL

OPINA Y VALORA ESTA CLASE

#DEMOCRATIZANDOLAEDUCACIÓN

CODER HOUSE