



Clase 44. Programación Backend

Desarrollo de un servidor web basado en capas - Parte 2



OBJETIVOS DE LA CLASE

- Reconocer funcionalidades de GraphQL.
- Identificar las diferencias entre GraphQL y REST.
- Desarrollar una GraphQL API.
- Utilizar las GraphQL API con GraphQL.

CRONOGRAMA DEL CURSO

Clase 43



Desarrollo de un servidor web basado en capas - Parte 1



SERVIDOR MVC COMPLETO



CONSUMIR NUESTRA API REST



ESQUEMA API RESTful

Clase 44



Desarrollo de un servidor web basado en capas - Parte 2



SERVIDOR MVC CON GRAPHQL



PROBANDO GRAPHQL



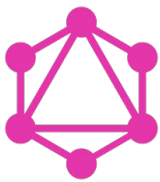
REFORMAR PARA USAR GRAPHQL

Clase 45



Introducción a frameworks de desarrollo backend - Parte 1

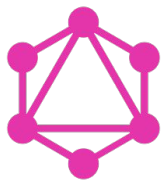
GRAPHQL API



¿De qué se trata?



- Como vimos en clases anteriores, GraphQL es un lenguaje de consulta y manipulación de datos para API, desarrollado por Facebook. Admite lectura, escritura y actualizaciones en tiempo real. Es de código abierto y es una alternativa a las API REST. En pocas palabras, se basa en:
 - Consultas GraphQL: esto permite al cliente leer y manipular cómo se deben recibir los datos.
 - Mutaciones GraphQL: así es como escribe datos en el servidor.
- Brinda una aproximación para el desarrollo de APIs y permite definir a los consumidores de los datos la estructura de los mismos para evitar que los servidores envíen datos innecesarios.



Ventajas de GraphQL



- Ofrece una mayor facilidad para el desarrollo de los proyectos. La reducción de las solicitudes para la visualización de datos, hace más sencilla la programación del frontend.
- Mejora el rendimiento de las aplicaciones, ya que reduce el número de peticiones al servidor y posibilita acceder a los datos que realmente se necesitan.
- Las solicitudes hechas con GraphQL permiten definir qué datos se van a recibir y en qué formato se va a consumir, por lo que se aseguran que el servidor no nos entregue más datos que los consultados.
- Da la posibilidad de tener separado el backend de un proyecto, lo que permite tener un único servicio web que funcione a través de varias plataformas.

COMPARACIÓN REST - GRAPHQL

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 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.

APLICACIÓN GRAPHQL API

Desarrollamos nuestra App



- Basándonos en la misma aplicación que desarrollamos la clase pasada, con un servidor API REST y un cliente en React, vamos a hacer los cambios necesarios para que ahora el servidor sea GraphQL API.
- Como el servidor que hicimos está hecho siguiendo la arquitectura en capas, debemos hacer muy pocas modificaciones para este cambio.
- Como lo que se cambia es la capa de presentación, donde cambiamos REST por GraphQL, lo que debemos modificar es la capa de rutas y algunas cosas del controlador.

Cambios generales

Ejemplo
en vivo



- En primer lugar, agregamos una variables de entorno, llamada GRAPHIQL ya que solo queremos habilitar esta herramienta cuando estemos en el ambiente de desarrollo (y no en producción).
- Entonces, así las inicializamos en los archivos **.env** de cada ambiente:

```
development.env X
server > development.env
1 // development.env
2 NODE_ENV=development
3 HOST=localhost
4 PORT=8080
5 //MEM - FILE - MONGO
6 TIPO_PERSISTENCIA=FILE
7 GRAPHIQL=true
```

```
production.env X
server > production.env
1 // production.env
2 NODE_ENV=production
3 HOST=localhost
4 PORT=9000
5 //MEM - FILE - MONGO
6 TIPO_PERSISTENCIA=MONGO
7 GRAPHIQL=false
```

Cambios generales

Ejemplo
en vivo



- Debemos agregar también esta variable de entorno en el archivo **config.js**. El resto de este archivo queda igual.

```
JS config.js X
server > JS config.js > ...
1  // config.js
2  import dotenv from 'dotenv';
3  import path from 'path';
4
5  dotenv.config({
6    path: path.resolve(process.cwd(), process.env.NODE_ENV + '.env')
7  });
8
9  export default {
10   NODE_ENV: process.env.NODE_ENV || 'development',
11   HOST: process.env.HOST || 'localhost',
12   PORT: process.env.PORT || 8080,
13   //MEM - FILE - MONGO
14   TIPO_PERSISTENCIA: process.env.TIPO_PERSISTENCIA || 'MEM',
15   GRAPHIQL: process.env.GRAPHIQL || 'true'
16 }
```


Cambios generales

Ejemplo
en vivo



```
JS server.js X
server > JS server.js > ...
1 import config from './config.js';
2 import express from 'express'
3 import cors from 'cors'
4 import RouterNoticias from './router/noticias.js'
5
6 const app = express()
7
8 if(config.NODE_ENV == 'development') app.use(cors())
9
10 app.use(express.static('public'))
11 app.use(express.json())
12
13 const routerNoticias = new RouterNoticias()
14
15 /* ----- */
16 /*          ZONA DE RUTAS MANEJADAS POR EL ROUTER          */
17 /* ----- */
18 app.use('/noticias', routerNoticias.start())
19
20 /* ----- */
21 /*          Servidor LISTEN          */
22 /* ----- */
23 const PORT = config.PORT || 8000
24 const server = app.listen(PORT,
25   () => console.log(
26     `Servidor express GRAPHQL escuchando en el puerto ${PORT}
27     \rConfig: [Modo: ${config.NODE_ENV}, Persistencia: ${config.TIPO_PERSISTENCIA}, GRAPHQL: ${config.GRAPHQL=='true'? 'Si': 'No'}]`
28   ))
29 server.on('error', error => console.log('Servidor express en error:', error))
```

- Y para chequear si estamos trabajando o no con GraphQL, agregamos la variable de entorno al console.log que teníamos en el servidor, en el archivo **server.js**. El resto de este archivo queda igual.

Cambios en capa de rutas

Ejemplo
en vivo



- Debemos tener instalados los módulos **graphql** y **express-graphql** que son los que vamos a utilizar en este caso.
- En el archivo `noticias.js` de la carpeta `rutas`, primero importamos el método `graphqlHTTP` del módulo `express-graphql` y el método `buildSchema` del módulo `graphql`.
- Importamos lo demás que necesitamos y creamos la clase con su constructor.

```
JS noticias.js X
server > router > JS noticias.js > ...
1  import { graphqlHTTP } from 'express-graphql';
2  import { buildSchema } from 'graphql';
3  import config from '../config.js'
4
5  import ControladorNoticias from '../controlador/noticias.js'
6
7
8  class RouterNoticias {
9
10     constructor() {
11         this.controladorNoticias = new ControladorNoticias()
12     }
13 }
```

Cambios en capa de rutas

Ejemplo
en vivo



```
14 start() {
15   // GraphQL schema
16   const schema = buildSchema(`
17     type Query {
18       noticias(_id: String!): [Noticia]
19     }
20     type Mutation {
21       guardarNoticia(
22         titulo: String!,
23         cuerpo: String!,
24         autor: String!,
25         imagen: String!,
26         email: String!,
27         vista: Boolean!,
28       ): Noticia,
29       actualizarNoticia(
30         _id: String!,
31         vista: Boolean!,
32       ): Noticia,
33       borrarNoticia(
34         _id: String!,
35       ): Noticia,
36     },
37     type Noticia {
38       _id: String!
39       titulo: String!
40       cuerpo: String!
41       autor: String!
42       imagen: String!
43       email: String!
44       vista: Boolean!
45     }
46   `);
```

- Luego, en el método `start()` creamos el esquema de GraphQL.
- En *Query* será el array con las noticias. Luego en *Mutation* tenemos los métodos de *guardarNoticia*, *actualizarNoticia* y *borrarNoticia* con sus variables.
- En *Noticia* tenemos las variables de una noticia, sus propiedades.

Cambios en capa de rutas

Ejemplo
en vivo



- Tenemos el Root Resolver, que contiene el mapeo de acciones a funciones, que en este caso llaman a los métodos del controlador correspondientes.
- Finalmente, devolvemos el método de graphqlHTTP con el objeto de parámetro con sus 3 propiedades.

```
48 // Root resolver
49 const root = {
50   noticias : _id => this.controladorNoticias.obtenerNoticias(_id),
51   guardarNoticia : this.controladorNoticias.guardarNoticia,
52   actualizarNoticia: (_id,noticias) => this.controladorNoticias.actualizarNoticia(_id,noticias),
53   borrarNoticia : _id => this.controladorNoticias.borrarNoticia(_id)
54 };
55
56 return graphqlHTTP({
57   schema: schema,
58   rootValue: root,
59   graphiql: config.GRAPHIQL == 'true'
60 })
61 }
62 }
63
64 export default RouterNoticias
```

Cambios en controlador

Ejemplo
en vivo



- En el archivo **noticias.js** de la carpeta **controlador**, tenemos prácticamente lo mismo que con API REST. Tenemos dos diferencias.
- La primera, es que como parámetro de los métodos se pasa directo el *id* (en REST se pasa *req* y *res*). Y además, como respuesta del método se pone directo “*return variable*” en lugar de “*res.json(variable)*” como en REST.

```
JS noticias.js X
server > controlador > JS noticias.js > ...
1 import ApiNoticias from '../api/noticias.js'
2
3 class ControladorNoticias {
4
5   constructor() {
6     this.apiNoticias = new ApiNoticias()
7   }
8
9   obtenerNoticias = async ({_id}) => {
10     try {
11       //console.log(_id)
12       let noticias = await this.apiNoticias.obtenerNoticias(_id)
13
14       return noticias
15     }
16     catch(error) {
17       console.log('error obtenerNoticias', error)
18     }
19   }
20
21   guardarNoticia = async ({titulo,cuerpo,autor,imagen,email,vista}) => {
22     try {
23       let noticia = {titulo,cuerpo,autor,imagen,email,vista}
24       let noticiaGuardada = await this.apiNoticias.guardarNoticia(noticia)
25
26       return noticiaGuardada
27     }
28     catch(error) {
29       console.log('error obtenerNoticias', error)
30     }
31   }
32
33   actualizarNoticia = async ({_id,vista}) => {
34     try {
35       //console.log(noticia)
36       let noticiaActualizada = await this.apiNoticias.actualizarNoticia(_id,{vista})
37       return noticiaActualizada
38     }
39     catch(error) {
40       console.log('error obtenerNoticias', error)
41     }
42   }
43 }
```

```
44   borrarNoticia = async ({_id}) => {
45     console.log(_id)
46     try {
47       let noticiaBorrada = await this.apiNoticias.borrarNoticia(_id)
48       return noticiaBorrada
49     }
50     catch(error) {
51       console.log('error obtenerNoticias', error)
52     }
53   }
54 }
55
56 export default ControladorNoticias
```

CODER HOUSE

CAMBIOS EN EL CLIENTE PARA CONSUMIR GraphQL API

Usar Axios con GraphQL



- En este caso, desde nuestro proyecto del lado cliente, vamos a consumir la API usando Axios como hicimos anteriormente. Pero ahora la API es GraphQL.
- Es por eso, que al consumir este tipo de APIs, Axios cambia un poco su forma de uso.
- Debemos pasarle como parámetros, además de la URL, la **Query** con las variables que estamos pidiendo en la *request* y el **headers** con el *content-type*.
- En las siguientes diapositivas, vemos ejemplos de esto con los cambios al proyecto de React que hicimos la clase pasada.

Cambios en el cliente

Ejemplo
en vivo



- Las modificaciones del lado del cliente son en el componente **Noticias.js** ya que es en el cual están hechos los llamados a la API con Axios.
- Como mencionamos, cambian las llamadas a la API de los métodos de este componente, ya que ahora es una GraphQL API la que estamos consumiendo.
- En primer lugar, importamos lo que vamos a necesitar (igual que en REST), creamos la URL genérica como hicimos antes, y creamos la clase del componente con las mismas variables de estado.

```
JS Noticias.js X
cliente > src > componentes > JS Noticias.js > ...
1  import React from 'react'
2  import Noticia from '../Noticia'
3
4  import axios from 'axios'
5
6  import '../Noticias.css'
7  import generarNoticia from '../generador'
8
9  const URL_NOTICIAS = (process.env.NODE_ENV === 'production'? '': 'http://localhost:8080') + '/noticias/'
10
11 class Noticias extends React.Component {
12
13   state = {
14     noticias : [],
15     idObtenerNoticia : '',
16     pedidas : false
17   }
18 }
```

CODER HOUSE

Cambios en el cliente

Ejemplo
en vivo



```
23  /* ----- */
24  /*   GET noticia (query)   */
25  /* ----- */
26  async obtenerNoticias(_id) {
27    try {
28      let body = {
29        query: `
30          query {
31            noticias${_id?('__id: "'+_id+'"'):''} {
32              _id
33              titulo
34              cuerpo
35              imagen
36              email
37              vista
38            }
39          }
40        `,
41        variables: {}
42      }
43      let options = {
44        headers: {
45          'Content-Type': 'application/json'
46        }
47      }
48      let response = await axios.post(URL_NOTICIAS, body, options)
49      let { data: {data: {noticias}} } = response
50      this.setState({noticias: noticias? noticias : []})
51    }
52    catch(error) {
53      console.error(error)
54      this.setState({noticias: []})
55    }
56    this.setState({pedidas : true, idObtenerNoticia: ''})
57  }
```

- Ahora el método de *obtenerNoticias*, define primero un body con la **Query**, dentro de la cual tenemos todas las variables de las noticias que habíamos definido en el esquema en el servidor.
- Luego, tenemos las options con sus **headers** y el *content-type*.
- Una vez que llamamos a la ruta usando Axios, debemos pasarle además de la URL, como parámetros, el *body* y las *options*.

Cambios en el cliente

Ejemplo
en vivo



- En *mutation*, dentro de la *query*, especificamos los valores de las variables al crear noticia, actualizarla o borrarla.

```
/* ----- */
/* POSI noticia (mutation) */
/* ----- */
async incorporarNoticia() {
  try {
    let noticia = generarNoticia()

    let body = {
      query: {
        mutation guardarNoticia($titulo: String!, $cuerpo: String!, $autor: String!, $imagen: String!, $email: String!, $vista: Boolean!) {
          guardarNoticia( titulo: $titulo, cuerpo: $cuerpo, autor: $autor, imagen: $imagen, email: $email, vista: $vista) {
            _id
            titulo
            cuerpo
            autor
            imagen
            email
            vista
          }
        }
      },
      variables: noticia
    }

    let options = {
      headers: {
        'Content-Type': 'application/json'
      }
    }

    let response = await axios.post(URL_NOTICIAS, body, options)
    let { data: {data: {guardarNoticia: noticiaIncorporada}} } = response

    //incorporo noticia en local
    let noticias = [...this.state.noticias]
    noticias.push(noticiaIncorporada)
    this.setState({noticias})
  } catch(error) {
    console.error('IncorporarNoticia', error)
  }
}
```

```
/* ----- */
/* UPDATE noticia (mutation) */
/* ----- */
async actualizarComoLeida(_id) {
  try {
    let body = {
      query: {
        mutation actualizarNoticia($ _id: String!, $vista: Boolean!) {
          actualizarNoticia( _id: $ _id, vista: $vista) {
            _id
            titulo
            cuerpo
            autor
            imagen
            email
            vista
          }
        }
      },
      variables: {
        _id: _id,
        vista: true
      }
    }

    let options = {
      headers: {
        'Content-Type': 'application/json'
      }
    }

    let response = await axios.post(URL_NOTICIAS, body, options)
    let { data: {data: {actualizarNoticia: noticiaActualizada}} } = response
    console.log(noticiaActualizada)

    //actualizo noticia en local
    let noticias = [...this.state.noticias]
    noticias.find(noticia => noticia._id === _id).vista = true
    this.setState({noticias})
  } catch(error) {
    console.error(error)
  }
}
```

```
/* ----- */
/* DELETE noticia (mutation) */
/* ----- */
async borrarNoticia(_id) {
  try {
    let body = {
      query: {
        mutation borrarNoticia($ _id: String!) {
          borrarNoticia( _id: $ _id) {
            _id
            titulo
            cuerpo
            autor
            imagen
            email
            vista
          }
        }
      },
      variables: {
        _id: _id
      }
    }

    let options = {
      headers: {
        'Content-Type': 'application/json'
      }
    }

    let response = await axios.post(URL_NOTICIAS, body, options)
    let { data: {data: {borrarNoticia: noticiaBorrada}} } = response
    console.log(noticiaBorrada)

    //borro noticia en local
    let noticias = [...this.state.noticias]
    let index = noticias.findIndex(noticia => noticia._id === _id)
    noticias.splice(index, 1)
    this.setState({noticias})
  } catch(error) {
    console.error('borrarNoticia', error)
  }
}
```

- Finalmente tenemos el *render*, sin modificaciones.



BREAK

¡5/10 MINUTOS Y VOLVEMOS!

GRAPHiQL

¿De qué se trata?



- GraphQL 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 Postaman 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:** GraphQL 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. GraphQL 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/noticias>. Se nos abre entonces el GraphiQL para poder empezar a probar los endpoints.

```
1 # Welcome to GraphiQL
2 #
3 # GraphiQL is an in-browser tool for writing, validating, and
4 # testing GraphQL queries.
5 #
6 # Type queries into this side of the screen, and you will see intelligent
7 # typeahead suggestions of the current GraphQL type schema and live syntax and
8 # validation errors highlighted within the text.
9 #
10 # GraphQL queries typically start with a "{" character. Lines that start
11 # with a # are ignored.
12 #
13 # An example GraphQL query might look like:
14 #
15 # {
16 #   field(arg: "value") {
17 #     subfield
18 #   }
19 # }
20 #
21 # Keyboard shortcuts:
22 #
23 # Prettify Query:  Shift-Ctrl-P (or press the prettify button above)
24 #
25 # Merge Query:    Shift-Ctrl-M (or press the merge button above)
26 #
27 # Run Query:      Ctrl-Enter (or press the play button above)
28 #
29 # Auto Complete:  Ctrl-Space (or just start typing)
30 #
31 #
32 #
```


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 GraphQL 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.

The screenshot shows the GraphQL Playground interface. On the left, the query editor contains the following code:

```
1 mutation {  
2   guardarNoticia(  
3     titulo : "hola",  
4     cuerpo: "mundo",  
5     autor: "DS",  
6     imagen: "http://",  
7     email: "j@d",  
8     vista: false) {  
9       _id  
10      titulo  
11      cuerpo  
12      autor  
13      imagen  
14      email  
15      vista  
16    }  
17 }
```

A red box highlights the query editor. A large arrow points from the query editor to the right, where the JSON response is displayed:

```
{  
  "data": {  
    "guardarNoticia": {  
      "_id": "1",  
      "titulo": "hola",  
      "cuerpo": "mundo",  
      "autor": "DS",  
      "imagen": "http://",  
      "email": "j@d",  
      "vista": false  
    }  
  }  
}
```

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.

The screenshot shows the GraphQL Playground interface. On the left, the query is defined as follows:

```
1 {  
2   noticias {  
3     _id  
4     titulo  
5     cuerpo  
6     autor  
7     imagen  
8     email  
9     vista  
10  }  
11 }
```

On the right, the JSON response is displayed:

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

Probar nuestra GraphQL API

Ejemplo
en vivo



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

The screenshot shows the GraphQL Playground interface. On the left, the query editor contains the following code:

```
1 query{
2   noticias(_id: "2") {
3     _id
4     titulo
5     cuerpo
6     autor
7     imagen
8     email
9     vista
10  }
11 }
```

On the right, the JSON response is displayed:

```
{
  "data": {
    "noticias": [
      {
        "_id": "2",
        "titulo": "Prueba",
        "cuerpo": "probando una noticia",
        "autor": "DS",
        "imagen": "http://",
        "email": "jd@",
        "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:

The screenshot shows the GraphiQL interface with a query editor on the left and a response viewer on the right. The query is a mutation to update a news item. The response is a JSON object containing the updated data.

```
1 mutation {  
2   actualizarNoticia(_id:"3",vista: true) {  
3     _id  
4     titulo  
5     cuerpo  
6     autor  
7     imagen  
8     email  
9     vista  
10  }  
11 }
```

```
{  
  "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



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

```
{
  "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
      }
    ]
  }
}
```

The screenshot shows the GraphQL IDE interface. On the left, the query editor contains a mutation to delete a news item with ID 2. On the right, the JSON response is displayed, showing the deleted item's details.

```
1 mutation {
2   borrarNoticia(_id:"2"){
3     _id
4     titulo
5     cuerpo
6     autor
7     imagen
8     email
9     vista
10  }
11 }
```

```
{
  "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).

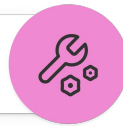


SERVIDOR MVC CON GRAPHQL

Tiempo: 15 a 20 minutos

SERVIDOR MVC CON GRAPHQL

Desafío
generico



Tiempo: 15 a 20 minutos

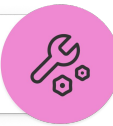
Realizar un esqueleto de servidor MVC basado en Node.js y express, que utilice GraphQL como lenguaje de Query.

- Su desarrollo debe estar separado en capas, basado en una estructura de carpetas y archivos que contenga los módulos necesarios para soportar:
- La capa de ruteo
 - El controlador
 - La lógica de negocio
 - Las validaciones de nuestros datos
 - La capa de persistencia (DAO, DTO)

Realizar una simple query y una mutación para pedir e incorporar palabras respectivamente a un array de strings persistidos en memoria, siguiendo la lógica de la separación del proceso en capas.

SERVIDOR MVC CON GRAPHQL

Desafío
generico



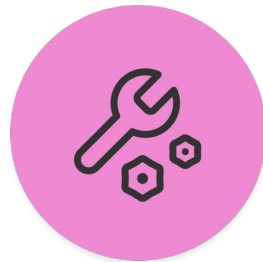
Tiempo: 15 a 20 minutos

Cada palabra que ingrese se debe almacenar en el array dentro de un objeto que contenga un timestamp. Ej.

```
[  
  { id: 1, palabra: "Hola", timestamp: 1624450180112 },  
  { id: 2, palabra: "que", timestamp: 1624450189685 },  
  { id: 3, palabra: "tal", timestamp: 1624450195068 }  
  ...  
]
```

Con el query en GraphQL se traerá la frase completa en formato string.

Probar la operación con GraphiQL.



PROBANDO GRAPHQL

Tiempo: 5 a 10 minutos

PROBANDO GRAPHQL

Tiempo: 5 a 10 minutos

Desafío
generico



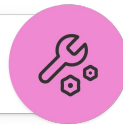
Realizar una sencilla página web front en HTML/JS (send.html) que al ejecutarse dentro del navegador, en un proceso independiente al servidor MVC GraphQL del desafío anterior (puede estar servida por el live server de visual studio code), le envíe a este mediante una mutación GraphQL una palabra al azar.

- ➔ No hace falta realizar la vista, el HTML estará para contener el script de ejecución.

PROBANDO GRAPHQL

Tiempo: 5 a 10 minutos

Desafío
generico



Utilizar axios en el front para emitir dicha mutación y los queries necesarios bajo el lenguaje GraphQL.

Así mismo, realizaremos otra página web (receive.html) similar a la anterior, que al ejecutar su script interno, genere un query GraphQL al mismo servidor para obtener la frase completa almacenada, representando por consola o en la vista del documento dicha frase.

- Considerar el uso de CORS en el servidor para permitir los request de dominios cruzados.



REFORMAR PARA USAR GRAPHQL

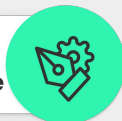
Refactoricemos el código del proyecto que venimos trabajando para cambiar de API RESTful a GraphQL API

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 GraphQL para realizar la prueba funcional de los querys y las mutaciones.

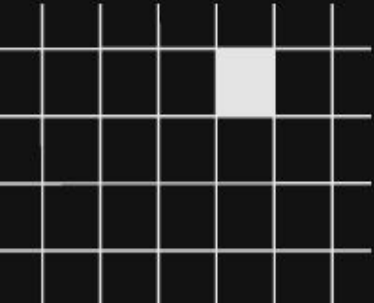
¿PREGUNTAS?





¡MUCHAS GRACIAS!

Resumen de lo visto en clase hoy:

- GraphQL API
 - Comparación entre GraphQL y REST
 - GraphiQL
- 



OPINA Y VALORA ESTA CLASE

#DEMOCRATIZANDO LA EDUCACIÓN

CODER HOUSE