Actividad 13: Agregar una API GraphQL a Next.js

Objetivo: Aprender a integrar una API GraphQL en una aplicación Next.js para optimizar la gestión y obtención de datos, mejorando la eficiencia, flexibilidad y escalabilidad del desarrollo web. Esta integración permite a los desarrolladores definir esquemas robustos, realizar consultas precisas que minimizan la sobrecarga de datos, aprovechar herramientas avanzadas como Apollo Server y Apollo Client, y facilitar una colaboración más efectiva entre frontend y backend.

Además, al combinar las capacidades de renderizado de Next.js con las ventajas de GraphQL, se potencia el rendimiento de la aplicación, se optimiza la experiencia del usuario y se asegura una arquitectura moderna y mantenible para proyectos web de alto rendimiento.

Introducción

Vamos a reestructurar la API de nuestra aplicación meteorológica para usar GraphQL. Para hacerlo, primero debemos agregar GraphQL al proyecto. GraphQL no es un patrón, sino un entorno que consta de un servidor y un lenguaje de consulta, ambos los cuales debemos agregar a Next.js.

Instalaremos el servidor independiente Apollo, uno de los servidores GraphQL más populares, que también proporciona una integración con Next.js. Abre tu terminal y navega a la aplicación refactorizada que construiste en la actividad 11.

En el nivel superior del directorio, junto al archivo package.json, ejecuta este comando:

```
$ npm install @apollo/server @as-integrations/next graphql graphql-tag
```

Este comando también instala el lenguaje GraphQL y los módulos graphql-tag que necesitaremos.

Creando el esquema

Como discutimos, cada API GraphQL comienza con una definición de esquema. Crea una carpeta llamada graphql junto a la carpeta pages en el directorio Next.js. Aquí es donde agregaremos todos los archivos relacionados con GraphQL.
Ahora crea un archivo llamado schema.ts y pega el código que escribiste en el listado siguiente:

```
export const typeDefs = gql`
   type LocationWeatherType {
       zip: String!
       weather: String!
       tempC: String!
       tempF: String!
       friends: [String]!
```


```
}
input LocationWeatherInput {
       zip: String!
       weather: String
       tempC: String
       tempF: String
       friends: [String]
}
type Query {
       weather(zip: String): [LocationWeatherType]!
}
type Mutation {
       weather(data: LocationWeatherInput): [LocationWeatherType]!
}
`;
```

Ya hemos definido y discutido la definición de tipos utilizada aquí. Simplemente agrega una línea en la parte superior del archivo:

```
import gql from "graphql-tag";
```

Esta línea importa el literal de plantilla etiquetada qql que usamos para definir el esquema.

Agregando datos

Queremos que nuestra API devuelva datos diferentes dependiendo de los parámetros y propiedades de las consultas enviadas a ella. Por lo tanto, necesitamos agregar conjuntos de datos a nuestro proyecto. GraphQL puede consultar cualquier base de datos, incluso datos JSON estáticos. Así que implementemos un conjunto de datos JSON. Crea el archivo data.ts dentro del directorio graphql y agrega el código siguiente:

```
export const db = [
    {
        zip: "96815",
        weather: "sunny",
        tempC: "25C",
        tempF: "70F",
        friends: ["96814", "96826"]
    },
    {
        zip: "96826",
        weather: "sunny",
        tempC: "30C",
        tempF: "86F",
        friends: ["96814", "96814"]
    },
```


```
    {
        zip: "96814",
        weather: "sunny",
        tempC: "20C",
        tempF: "68F",
        friends: ["96815", "96826"]
    }
];
```

Este JSON define tres ubicaciones meteorológicas y sus propiedades. Un consumidor podrá consultar nuestra API para obtener estos conjuntos de datos.

Implementando resolvers

Ahora podemos definir nuestros resolvers. Agrega el archivo resolvers.ts al directorio graphql y pega el código siguiente. Esto es similar al código que discutimos anteriormente cuando introdujimos los resolvers, pero en lugar de devolver el mismo objeto JSON estático al consumidor, consultamos nuestro nuevo conjunto de datos.

```
import {db} from "./data";

declare interface WeatherInterface {
    zip: string;
    weather: string;
    tempC: string;
    tempF: string;
    friends: string[];
}

export const resolvers = {
    Query: {
        weather: async (_: any, param: WeatherInterface) => {
            return [db.find((item) => item.zip === param.zip)];
        }
    },
    Mutation: {
        weather: async (_: any, param: {data: WeatherInterface}) => {
            return [db.find((item) => item.zip === param.data.zip)];
        }
    }
};
```

Importamos el array de objetos JSON que creamos anteriormente y definimos una interfaz para los resolvers. El resolver de consulta encuentra un objeto utilizando el código postal pasado a él y lo devuelve al servidor Apollo. La mutación hace lo mismo, excepto que la estructura del parámetro es ligeramente diferente: es accesible a través de la propiedad data.

Lamentablemente, no podemos cambiar realmente los datos utilizando la mutación, ya que los datos son un archivo JSON estático. Hemos implementado la mutación aquí solo con fines ilustrativos.

Creando la ruta de la API

El servidor Apollo GraphQL expone un endpoint, graphql/, que implementaremos ahora. Crea un nuevo archivo, graphql.ts, en la carpeta api y agrega el código siguiente.

Este código inicializa el servidor GraphQL y agrega un encabezado CORS para que podamos acceder a la API desde diferentes dominios y usar el explorador sandbox de GraphQL integrado para jugar con GraphQL más tarde. Viste este encabezado en las respuestas cURL anteriores.

```
import {ApolloServer} from "@apollo/server";
import {startServerAndCreateNextHandler} from "@as-integrations/next";
import {resolvers} from "../../graphql/resolvers";
import {typeDefs} from "../../graphql/schema";
import {NextApiHandler, NextApiRequest, NextApiResponse} from "next";

//@ts-ignore
const server = new ApolloServer({
    resolvers,
    typeDefs
});

const handler = startServerAndCreateNextHandler(server);

const allowCors =
    (fn: NextApiHandler) => async (req: NextApiRequest, res: NextApiResponse) => {
        res.setHeader("Allow", "POST");
        res.setHeader("Access-Control-Allow-Origin", "*");
        res.setHeader("Access-Control-Allow-Methods", "POST");
        res.setHeader("Access-Control-Allow-Headers", "*");
        res.setHeader("Access-Control-Allow-Credentials", "true");

        if (req.method === "OPTIONS") {
            res.status(200).end();
        }
        return await fn(req, res);
    };

export default allowCors(handler);
```

Este código es todo lo que necesitamos para crear el punto de entrada de GraphQL. Primero importamos los módulos necesarios, incluidos nuestro esquema GraphQL y los resolvers, ambos los

cuales creamos anteriormente. Luego inicializamos un nuevo servidor GraphQL con typedefs y resolvers.

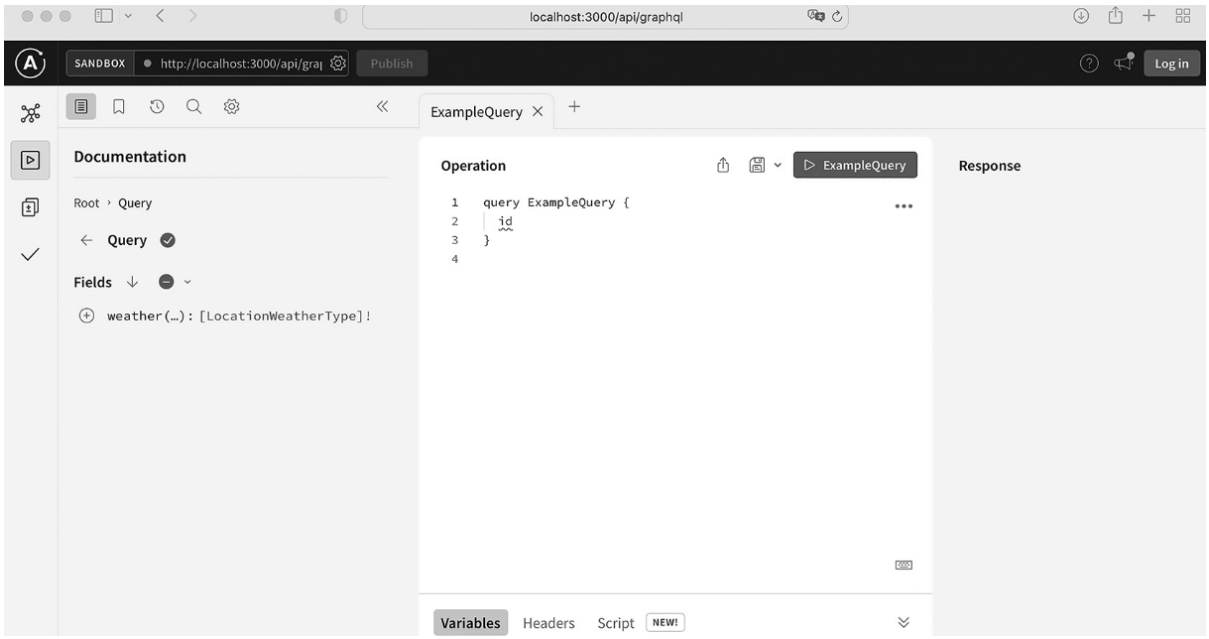
Iniciamos el servidor y continuamos creando el manejador de la API. Para hacerlo, utilizamos el ayudante de integración de Next.js para iniciar el servidor y devolver el manejador de Next.js. El ayudante de integración conecta la instancia sin servidor de Apollo al servidor personalizado de Next.js. Antes de definir la exportación predeterminada como una función async que toma los objetos de solicitud y respuesta de la API como parámetros, creamos un envoltorio para agregar los encabezados CORS a la solicitud.
El primer bloque dentro de la función configura los encabezados CORS, y limitamos la solicitud permitida a solicitudes POST. Necesitamos los encabezados CORS aquí para hacer que nuestra API GraphQL sea públicamente disponible. De lo contrario, no podríamos conectarnos a la API desde un sitio web que se ejecute en un dominio diferente o incluso usar el sandbox GraphQL incorporado del servidor.

Parte de la configuración CORS aquí es que inmediatamente devolvemos 200 para cualquier solicitud OPTIONS. Los patrones CORS utilizan solicitudes OPTIONS como verificaciones preliminares. Aquí, el navegador solicita solo encabezados y luego verifica los encabezados CORS de la respuesta para verificar que el dominio desde el cual llama a la API tiene permiso para acceder al recurso antes de realizar la solicitud real.

Sin embargo, el servidor Apollo solo permite solicitudes POST y GET y devolvería 405: Método No Permitido para la solicitud preliminar OPTIONS. Entonces, en lugar de pasar esta solicitud al servidor Apollo, terminamos la solicitud y devolvemos 200 con los encabezados CORS anteriores. El navegador debería proceder luego con el patrón CORS.
Finalmente, iniciamos el servidor y creamos el manejador de la API en la ruta deseada, api/graphql.

Usando el sandbox de Apollo

Inicia tu servidor Next.js con npm run dev. Deberías ver la aplicación Next.js ejecutándose en http://localhost:3000. Si navegas a la API GraphQL en http://localhost:3000/api/graphql, encontrarás la interfaz sandbox de Apollo para consultar la API, como en la figura:

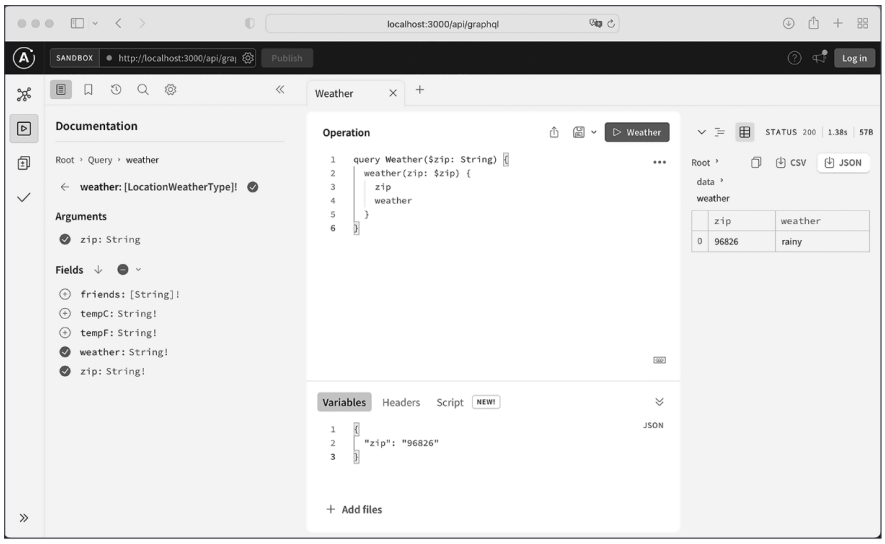


En el panel de documentación a la izquierda, vemos las consultas disponibles como campos del objeto de consulta que definimos anteriormente. Como era de esperar, vemos la consulta Weather aquí, y cuando hacemos clic en ella, aparece una nueva consulta en el panel de Operación en el medio. Al mismo tiempo, la interfaz cambia y vemos los argumentos y campos disponibles.

Hacer clic en cada uno proporciona más información. Usando el botón más (+), podemos agregar campos al panel de Consulta y ejecutarlos contra los datos.

Intenta crear una consulta Weather que devuelva las propiedades zip y weather. Esta consulta requiere un código postal como argumento; agrégalo a través de la interfaz de usuario en el lado izquierdo, y luego agrega el código postal 96826 como una cadena al objeto JSON en la sección de variables del panel inferior. Ahora ejecuta la consulta haciendo clic en el botón Weather en la parte superior del panel de Operación. Deberías recibir el resultado para este código postal en el panel de Respuesta a la derecha como JSON.

Compara tu pantalla con la figura siguiente:



Juega creando consultas, accediendo a propiedades y creando errores con argumentos inválidos para familiarizarte con GraphQL antes de pasar al siguiente tema.

Ejercicios teóricos

Conceptos básicos de GraphQL:

Pregunta: ¿Cuál es la diferencia principal entre GraphQL y REST en términos de cómo manejan las solicitudes y las respuestas?

Esquema de GraphQL:

Pregunta: Explica la importancia de definir un esquema (`typeDefs`) en GraphQL y cómo influye en las consultas y mutaciones.

Resolvers:

Pregunta: ¿Qué es un resolver en GraphQL y cuál es su función dentro de una API GraphQL?

CORS en APIs:

Pregunta: ¿Por qué es necesario configurar los encabezados CORS en la API GraphQL y qué problemas podrían surgir si no se hace correctamente?

Mutaciones vs consultas:

Pregunta: ¿Cuál es la diferencia entre una consulta (`Query`) y una mutación (`Mutation`) en GraphQL?

Ejercicios prácticos

Extender el Esquema con nuevos tipos:

Tarea: Agrega un nuevo tipo llamado `User` al esquema que incluya los campos `id`, `name`, `email` y `location` (que referencia a `LocationWeatherType`). Actualiza las consultas para permitir obtener usuarios junto con sus ubicaciones meteorológicas.

Pasos sugeridos:

- Define el tipo `User` en `schema.ts`.
- Actualiza el tipo `Query` para incluir una consulta `users` que retorne una lista de `User`.
- Implementa los resolvers necesarios en `resolvers.ts`.
- Actualiza el archivo `data.ts` para incluir datos de usuarios.

Implementar una mutación para agregar una nueva ubicación:

Tarea: Crea una mutación que permita agregar una nueva ubicación meteorológica a la base de datos db.
Pasos sugeridos:

- Define una nueva mutación en el esquema para agregar una ubicación.
- Implementa el resolver correspondiente que añada la nueva ubicación al array db.
- Asegúrate de manejar posibles errores, como códigos postales duplicados.

Crear consultas anidadas:

Tarea: Modifica la consulta `weather` para que, además de los campos actuales, también retorne la información de los amigos (sus códigos postales y clima).
Pasos sugeridos:

- Actualiza el tipo `LocationWeatherType` para que el campo `friends` sea de tipo `[LocationWeatherType]`.
- Ajusta los resolvers para resolver los datos de los amigos correctamente.
- Realiza una consulta en el sandbox de Apollo que obtenga, por ejemplo, el clima de una ubicación y el clima de sus amigos.

Añadir autenticación básica:

Tarea: Implementa una autenticación básica en la API GraphQL que requiera un token de acceso en las solicitudes.

Pasos sugeridos:

- Configura middleware para verificar la presencia y validez del token en las solicitudes.

- Modifica los resolvers para que verifiquen la autenticidad antes de procesar las consultas o mutaciones.
- Prueba el acceso a la API con y sin el token válido.

Optimizar consultas con paginación:

Tarea: Implementa paginación en la consulta `weather` para manejar grandes conjuntos de datos.

Pasos sugeridos:

- Añade argumentos como `limit` y `offset` a la consulta `weather`.
- Ajusta el resolver para que retorne solo el subconjunto de datos solicitado.
- Realiza pruebas en el sandbox de Apollo para verificar la paginación.

Consultas adicionales

Consulta básica con selección de campos:

```
query {
  weather(zip: "96815") {
    zip
    weather
  }
}
```

2. Consulta con variables:

```
query GetWeather($zipcode: String!) {
  weather(zip: $zipcode) {
    zip
    weather
    tempC
    tempF
  }
}
```

Variables:

```
{
  "zipcode": "96826"
}
```

3. Mutación para agregar una nueva ubicación:

```
mutation AddWeather($data: LocationWeatherInput!) {
    weather(data: $data) {
      zip
      weather
      tempC
      tempF
      friends
    }
}
```

Variables:

```
{
  "data": {
    "zip": "97005",
    "weather": "cloudy",
    "tempC": "22C",
    "tempF": "72F",
    "friends": ["96815", "96826"]
  }
```


```
  }
```

4. Consulta anidada (si implementaste el tipo User):

```
query {
    users {
      id
      name
      email
      location {
        zip
        weather
      }
    }
  }
```

5. Consulta con paginación (si implementaste la paginación):

```
query {
    weather(limit: 2, offset: 1) {
      zip
      weather
      tempC
      tempF
    }
  }
```

6. Consulta con fragmentos para reutilizar campos:


```
fragment WeatherFields on LocationWeatherType {
    zip
    weather
    tempC
    tempF
  }

  query {
    weather(zip: "96814") {
      ...WeatherFields
    }
  }
```

7. Consulta con alias para diferenciar resultados:

```
query {
    firstWeather: weather(zip: "96815") {
      zip
      weather
    }
    secondWeather: weather(zip: "96826") {
      zip
      weather
    }
  }
```

8. Consulta con directivas (si estás familiarizado con ellas):

```
query GetWeather($includeTemp: Boolean!) {
```


```
    weather(zip: "96815") {
      zip
      weather
      tempC @include(if: $includeTemp)
      tempF @include(if: $includeTemp)
    }
  }
```

Variables:

```
  {
    "includeTemp": true
  }
```