El futuro digital es de todos

MinTIC

Ciclo 4a:

Desarrollo de aplicaciones web

UNIVERSIDAD NACIONAL DE COLOMBIA

MisiónTIC2022

VERSIÓN 1.0

Unidad de educación continua y permanente
Facultad de Ingeniería

Unidad Camilo Torres
Calle 44 # 45-67
Bloque B5 piso 1

(57) + 316 5000
uec_fibog@unal.edu.co

Actividad Práctica
(API Gateway - Parte 2)

Una vez se han desplegado los microservicios *auth_ms*, y *account_ms*, y luego de comprender los conceptos básicos sobre el funcionamiento de un API Gateway, se puede realizar su implementación. Para ello, se debe tener en cuenta que en el API Gateway se integrarán las funcionalidades desarrolladas en ambos microservicios para exponer las funcionalidades finales al usuario final, de forma que este no note la existencia de múltiples microservicios en el sistema.

Datos del Servidor

A lo largo del microservicio se utiliza cierta información sujeta a cambios, como las URL de los microservicios. Para facilitar el manejo de esta información se agrupa en el archivo server.js de la siguiente manera:

```
module.exports = {
    auth_api_url: 'https://mision-tic-auth-ms.herokuapp.com',
    account_api_url: 'https://mision-tic-account-ms.herokuapp.com',
};
```

Estos valores son exportados y se pueden utilizar en cualquier parte del proyecto cuando se necesiten. Al estar ubicados en un único lugar cuando se realice un cambio en estos, solo bastará con modificar el valor en *server.js*.

Authentication Context

Dentro del entorno de desarrollo de *Apollo Server* se tiene el concepto de *Context*, para comprenderlo, se debe tener en cuenta que durante la ejecución de un API Gateway este recibe múltiples peticiones. Una vez llegan dichas peticiones, se les realiza un preprocesamiento con un *Context*, es decir, con una función que le agrega o resta información a la petición según sea el caso. De esta forma, Un *Context* se puede ver como una función que permite brindarle un contexto adicional a las peticiones.

Para implementar el mecanismo de autenticación en el API Gateway, con ayuda del microservicio *auth_ms*, se buscará validar cada una de las peticiones que llegan al API Gateway. Para ello, se utilizará el token generado por el microservicio. De esta forma, todas las peticiones deberán incluir un token que identifique al usuario que está realizando la petición. Hay algunas peticiones que no requieren verificación, por ejemplo, las peticiones que permiten generar o refrescar el token y la petición que permite realizar el registro de un nuevo usuario.

En la práctica para llevar a cabo el proceso de autenticación en el API Gateway, se creará un contexto de autenticación. La función correspondiente a dicho contexto tomará la petición entrante y revisará si esta tiene un token asociado en sus headers o no, en caso de tenerlo realizará una petición al microservicio *auth_ms* para corroborar la validez del token. En caso de ser válido, agrega la información extraída del token a la petición; cuando se procese la petición se solicitará la información adicional para validar la autenticación del usuario, y si no la posee, se retorna algún tipo de error. Las funciones que no necesiten verificación simplemente omitirán este proceso.

El anterior proceso es implementado con el siguiente fragmento de código, y debe ir en el archivo *src/utils/authentication.js*:

```
const { ApolloError } = require('apollo-server');
const serverConfig = require('../server');
const fetch = require('node-fetch');

const authentication = async ({ req }) => {
    const token = req.headers.authorization || '';

    if (token == '')
        return { userIdToken: null }

    else {
        try {
            let requestOptions = {
                method: 'POST', headers: { "Content-Type": "application/json" },
                body: JSON.stringify({ token }), redirect: 'follow'
            };

            let response = await fetch(
                `${serverConfig.auth_api_url}/verifyToken/`,
                requestOptions)

            if (response.status != 200) {
                console.log(response)
                throw new ApolloError(`SESION INACTIVA - ${401}` + response.status, 401)
            }

            return { userIdToken: (await response.json()).UserId };
        }
        catch (error) {
            throw new ApolloError(`TOKEN ERROR: ${500}: ${error}`, 500);
        }
```

```
    }
}

module.exports = authentication;
```

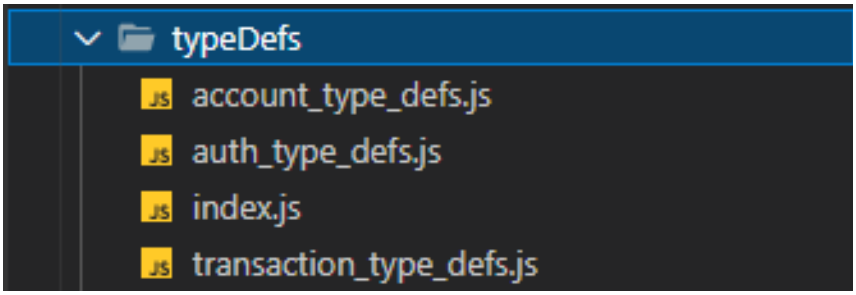
Para comprender el código, se debe notar que se define una función que recibe una petición como parámetro (*req*), y dentro de esta se comprueba si dicha petición posee un token; si lo tiene, se realiza una petición al microservicio *auth_ms* para comprobar si el token es válido, si lo es, retorna la información adicional (*userIdToken*) que utilizarán las peticiones que requieran de un token; si no lo es, retorna un error al componente web.

TypeDefs

GraphQL, a diferencia de REST, es más estricto con los datos que recibe y retorna su interfaz. En REST se pueden manejar distintos tipos de datos siempre y cuando sean soportados por el formato JSON. En cambio, en GraphQL se deben recibir y retornar estructuras de datos definidas previamente, esto permite un API más robusta y fácil de utilizar. Estas estructuras de datos son definidas en lo que se conoce como un TypeDef, en este se definen las *Queries* (consultas de datos) y *Mutations* (transformaciones de datos), y las estructuras de datos que estas operaciones utilizaran.

En este caso se tienen tres *TypeDef*. El primero es el *TypeDef* de *auth*, este representa los procesos necesarios para realizar los procesos de registro y autenticación, se puede decir que esta entidad está asociada al microservicio *auth_ms*, en segundo lugar, se tiene el *TypeDef* de *account*, este representa la cuenta bancaria de los usuarios, en tercer lugar, se tiene el *TypeDef* de *transaction*, este representa la entidad transacción que son los intercambios de dinero entre los usuarios, estos dos últimos TypeDefs están asociados al microservicio *account_ms*.

La implementación de los *TypeDefs* se realiza en los siguientes archivos. Los archivos con terminación *type_defs.js* definirán la estructura de las operaciones y el archivo *index.js* se encargará de crear una estructura única que reúna dichas operaciones.



El código correspondiente archivo auth_type_defs.js es el siguiente:

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

const authTypeDefs = gql `
    type Tokens {
        refresh: String!
        access: String!
    }

    type Access {
        access: String!
    }

    input CredentialsInput {
        username: String!
        password: String!
    }

    input SignUpInput {
        username: String!
        password: String!
        name: String!
        email: String!
        balance: Int!
    }

    type UserDetail {
        id: Int!
        username: String!
        password: String!
        name: String!
        email: String!
    }

    type Mutation {
        signUpUser(userInput :SignUpInput): Tokens!
        logIn(credentials: CredentialsInput!): Tokens!
        refreshToken(refresh: String!): Access!
    }

    type Query {
        userDetailById(userId: Int!): UserDetail!
    }
```

```
`;

module.exports = authTypeDefs;
```

El código es realmente sencillo, con ayuda del elemento *gql* provisto por *Apollo Server*, se define una serie de objetos como *Tokens*, *Access*, *CredentialsInput*, *SignUpInput* y *UserDetail*, estos representan el flujo de datos que existirán entre las operaciones. Con ayuda de estos se crea un objeto en el que se definen las *Mutations* y otro objeto en el que se definen las *Queries*. El símbolo de admiración (*!*) establece que el dato de entrada o salida es obligatorio.

El código correspondiente archivo account_type_defs.js es el siguiente:

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

const accountTypeDefs = gql `
    type Account {
        username: String!
        balance: Int!
        lastChange: String!
    }

    extend type Query {
        accountByUsername(username: String!): Account
    }
`;

module.exports = accountTypeDefs;
```

Este fragmento de código es muy similar al del *TypeDef* anterior, salvo por un pequeño detalle, y es que en este caso no se crea un objeto para agrupar las *Queries*, sino que se usa la palabra clave extend para indicar que ya existe uno (el definido en el *TypeDef* anterior), y se deben incluir las *Queries* a este.

El código correspondiente archivo transacction_type_defs.js es el siguiente:

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

const transactionTypeDefs = gql `
    type Transaction {
        id: String!
```

```
        usernameOrigin: String!
        usernameDestiny: String!
        value: Int!
        date: String!
    }

    input TransactionInput {
        usernameOrigin: String!
        usernameDestiny: String!
        value: Int!
    }

    extend type Query {
        transactionByUsername(username: String!): [Transaction]
    }

    extend type Mutation {
        createTransaction(transaction: TransactionInput!): Transaction
    }
`;

module.exports = transactionTypeDefs;
```

Al igual que el *TypeDef* de *account*, no se crea un objeto para las *Mutations* o para las *Queries*, sino que usa *extend* para agregarlas a las ya existentes.

Por último, se tienen el código del archivo typeDefs/index.js, este permitirá unir los 3 *TypeDefs* en una única estructura:

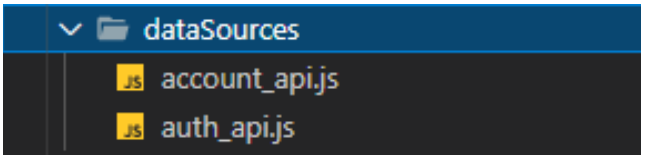
```
//Se llama al typedef (esquema) de cada submodulo
const accountTypeDefs = require('./account_type_defs');
const transactionTypeDefs = require('./transaction_type_defs');
const authTypeDefs = require('./auth_type_defs');

//Se unen
const schemasArrays = [authTypeDefs, accountTypeDefs, transactionTypeDefs];

//Se exportan
module.exports = schemasArrays;
```

Primero se importan los 3 *TypeDefs*, luego se unen en un único esquema y por último se exporta ese esquema.

Data Sources

Al desarrollar el API Gateway con *Apollo Server* se tienen algunos beneficios que no se tendrían si se trabajara solamente con *NodeJS*. Uno de estos beneficios son los *DataSources*; como es de esperarse el API Gateway debe comunicarse con los microservicios, una primera idea de cómo hacerlo sería realizar *peticiones HTTP* a los microservicios con ayuda de algún paquete como *Axios*, sin embargo, existe una manera más eficiente de hacerlo. *Apollo Server* ofrece el concepto de *DataSource*, el cual permite la creación de una clase que maneja de manera rápida y sencilla las peticiones a los microservicios, esta clase hereda de una clase base que agrega todo el entramado necesario para la comunicación con los microservicios. En resumen, un *DataSource* es una manera fácil y rápida de acceder a los microservicios.

En este caso se crearán dos *DataSources*, una para cada microservicio, en estas se definirán las peticiones necesarias. La implementación se realiza en los siguientes archivos:



El código correspondiente al archivo auth_api.js es el siguiente:

```
const { RESTDataSource } = require('apollo-datasource-rest');

const serverConfig = require('../server');

class AuthAPI extends RESTDataSource {

    constructor() {
        super();
        this.baseURL = serverConfig.auth_api_url;
    }

    async createUser(user) {
        user = new Object(JSON.parse(JSON.stringify(user)));
        return await this.post(`/user/`, user);
    }

    async getUser(userId) {
```

```
        return await this.get(`/user/${userId}/`);
    }

    async authRequest(credentials) {
        credentials = new Object(JSON.parse(JSON.stringify(credentials)));
        return await this.post(`/login/`, credentials);
    }

    async refreshToken(token) {
        token = new Object(JSON.parse(JSON.stringify({ refresh: token })));
        return await this.post(`/refresh/`, token);
    }

}

module.exports = AuthAPI;
```

S e tiene una única clase *AuthAPI* la cual se crea a partir de *RESTDataSource* provista por *Apollo*. Para crear dicha clase, únicamente se requiere definir la URL del microservicio, una vez definida la clase, dentro de esta se implementan una serie de funciones que representan todos y cada uno de los servicios que ofrece el microservicio. Para la definición de estas funciones se hace uso de métodos internos como *this.post* o *this.get*.

El único detalle del contenido de las funciones es que en algunos casos se realizan algunas transformaciones con ayuda de JSON, esto solamente se hace para mantener coherencia con los formatos aceptados por los microservicios.

El código correspondiente al archivo account_api.js es el siguiente:

```
const { RESTDataSource } = require('apollo-datasource-rest');

const serverConfig = require('../server');

class AccountAPI extends RESTDataSource {

    constructor() {
        super();
        this.baseURL = serverConfig.account_api_url;
    }

    async createAccount(account) {
        account = new Object(JSON.parse(JSON.stringify(account)));
```

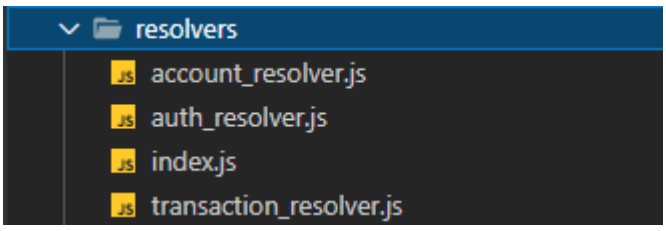
```
        return await this.post('/accounts', account);
    }

    async accountByUsername(username) {
        return await this.get(`/accounts/${username}`);
    }

    async createTransaction(transaction) {
        transaction = new Object(JSON.parse(JSON.stringify(transaction)));
        return await this.post('/transactions', transaction);
    }

    async transactionByUsername(username) {
        return await this.get(`/transactions/${username}`);
    }
}

module.exports = AccountAPI;
```

Este fragmento de código es similar al descrito anteriormente.

Resolvers

En los *TypeDefs* se definió la estructura del API Gateway, es decir, las operaciones que realizará (*Queries* y *Mutations*), pero aún no se han definido como resolver o manipular dichas operaciones, es aquí donde aparece el contexto de *Resolver*. Un *Resolver*, como su nombre lo indica, es el elemento encargado de determinar cómo resolver o procesar una operación. En el ambiente de *Apollo Server* se debe crear un *Resolver* para cada una de las operaciones (*Queries* o *Mutations*) definidas en los *TypeDefs*.

En este caso se crearán tres *Resolvers*, uno para cada *TypeDef*, y dentro de estos se definirán una serie de funciones con un formato especifico que permitirá resolver las *Queries* y *Mutations*. Estas funciones reciben algunos parámetros (cuyo origen se explicará más adelante). La implementación se realiza en los siguientes archivos:



Para comprender el código, se deben tener en cuenta los siguientes detalles:

- Las funciones ya sean *Mutations* o *Queries* reciben ciertos parámetros, el primero de ellos (segundo en orden ya que el primero solamente es el indicador '_') son los datos definidos en el *TypeDef*.

- Además de los datos definidos en el *TypeDef*, las funciones del resolver reciben algunos datos adicionales. El primero de ellos es un objeto llamado *DataSource*, el cual, como su nombre lo indica contiene una instancia de los *DataSources* y permitirá acceder de manera fácil y rápida a los microservicios desde el *Resolver*.

- La segunda información adicional que recibe las funciones de los resolver es *userIdToken*, esta es la información agregada por el contexto de autenticación. Cabe resaltar que únicamente lo recibirán las funciones que necesitan autenticación.

El código correspondiente al archivo auth_resolver.js es el siguiente:

```
const usersResolver = {
    Query: {
        userDetailById: (_, { userId }, { dataSources, userIdToken }) => {
            if (userId == userIdToken)
                return dataSources.authAPI.getUser(userId)
            else
                return null
        },
    },
    Mutation: {
        signUpUser: async(_, { userInput }, { dataSources }) => {
            const accountInput = {
                username: userInput.username,
                balance: userInput.balance,
                lastChange: (new Date()).toISOString()
            }
            await dataSources.accountAPI.createAccount(accountInput);

            const authInput = {
                username: userInput.username,
                password: userInput.password,
                name: userInput.name,
                email: userInput.email,
            }
            return await dataSources.authAPI.createUser(authInput);
        },
        logIn: (_, { credentials }, { dataSources }) =>
```

```
            dataSources.authAPI.authRequest(credentials),

        refreshToken: (_, { refresh }, { dataSources }) =>
            dataSources.authAPI.refreshToken(refresh),
    }
};

module.exports = usersResolver;
```

El código está conformado por dos objetos, en uno de ellos se definen las funciones que procesarán las respectivas *Queries*, y en el otro se definen las funciones que procesaran las respectivas *Mutations*, algunos detalles que se deben tener en cuenta de esta implementación son las siguientes:

- Algunas operaciones como *LogIn*, *SignUpUser* y *RefefeshToken* no requieren validar el id del usuario, pues no están accediendo a recursos protegidos. En el caso de *userDetailById*, si se requiere dicha validación, ya que es necesario verificar si el usuario que solicita los datos está autenticado.

- Algunas operaciones simplemente hacen uso de una sola funcionalidad de un microservicio. En estos casos la implementación de la función que las procesa es muy sencilla, y se puede realizar incluso en una única línea de código.

- La verdadera importancia del API Gateway se ve reflejada en la operación *signUpUser*, en esta, para poder registrar el usuario se requiere utilizar funcionalidades de ambos microservicios, primero crea una cuenta bancaria para el usuario usando la funcionalidad *createAccount* del microservicio *account_ms* y luego se completa el registro creando el usuario con la funcionalidad *createUser* del microservicio *auth_ms*, aquí se puede evidenciar claramente el proceso de orquestación de los microservicios que realiza el API Gateway.

El código correspondiente al archivo account_resolver.js es el siguiente:

```
const accountResolver = {
    Query: {
        accountByUsername: async(_, { username }, { dataSources, userIdToken }) => {
            usernameToken = (await dataSources.authAPI.getUser(userIdToken)).username
            if (username == usernameToken)
                return await dataSources.accountAPI.accountByUsername(username)
            else
                return null
        },
    },
```

```
    Mutation: {}
};

module.exports = accountResolver;
```

La estructura del código es muy similar a la del Resolver anterior, se puede observar que en este caso solo se define una *Query* que requiere autenticación.

El código correspondiente al archivo transaction_resolver.js es el siguiente:

```
const transactionResolver = {
    Query: {
        transactionByUsername: async(_, { username }, { dataSources, userIdToken }) => {
            usernameToken = (await dataSources.authAPI.getUser(userIdToken)).username
            if (username == usernameToken)
                return dataSources.accountAPI.transactionByUsername(username)
            else
                return null
        }
    },
    Mutation: {
        createTransaction: async(_, { transaction }, { dataSources, userIdToken }) => {
            usernameToken = (await dataSources.authAPI.getUser(userIdToken)).username
            if (transaction.usernameOrigin == usernameToken)
                return dataSources.accountAPI.createTransaction(transaction)
            else
                return null
        }
    }
};

module.exports = transactionResolver;
```

La estructura del código es muy similar a la de los *Resolvers* anteriores, se puede observar que en este caso se define una *Query* y una *Mutation* que requieren autenticación.

Por último, se tiene el código del archivo resolvers/index.js, este fragmento permite unir todos los resolvers en un único resolver:

```
const accountResolver = require('./account_resolver');
```

```
const transactionResolver = require('./transaction_resolver');
const authResolver = require('./auth_resolver');

const lodash = require('lodash');

const resolvers = lodash.merge(accountResolver, transactionResolver, authResolver);

module.exports = resolvers;
```

Para crear una única estructura que agrupe todo los *Resolvers* y facilite su exportación, se hace uso de la librería *lodash*, esta toma los *Resolvers* previamente importados y los agrupa en un único objeto, este objeto tendrá todas las funciones de las *Queries* en un sub-objeto llamado *Query* y las funciones de las mutaciones en un sub-objeto llamado Mutation.

API Gateway

En este punto se han definido todas las partes esenciales del API Gateway GraphQL de *Apollo Server*, pero aún no se han integrado, lo cual genera algunas desconexiones a nivel lógico, pero no debe haber preocupación por esto ya que *Apollo Server* realizará toda la orquestación. Para entender esto se debe analizar el proceso que realiza un petición, cuando se recibe la petición se pasa por el contexto de autenticación, luego se analiza el contenido de la petición y con ayuda de los *TypeDefs*, la información resultante del contexto y los *DataSources*, se definen los parámetros del *Resolver* correspondiente y se hace el llamado de este, luego se obtiene el resultado y se retorna, todo este proceso es orquestado y dirigido por *Apollo Server*.

Para lograr el anterior funcionamiento con *Apollo Server* es necesario definir una instancia de *Apollo Server* e indicarle el contexto, los *TypeDefs*, los *Resolvers* y los *DataSources*, y por último activar el servidor. Este proceso es realizado en el archivo src/index.js:

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

const typeDefs = require('./typeDefs');
const resolvers = require('./resolvers');
const AccountAPI = require('./dataSources/account_api');
const AuthAPI = require('./dataSources/auth_api');
const authentication = require('./utils/authentication');

const server = new ApolloServer({
    context: authentication,
    typeDefs,
```

```
    resolvers,
    dataSources: () => ({
        accountAPI: new AccountAPI(),
        authAPI: new AuthAPI(),
    }),
    introspection: true,
    playground: true
});

server.listen(process.env.PORT || 4000).then(({ url }) => {
    console.log(`🚀  Server ready at ${url}`);
});
```

Primero se importan los elementos necesarios, luego se crea la instancia de *Apollo Server*, y finalmente se realiza la ejecución.

Ejecución en Entorno Local

Si bien no es necesario correr la aplicación de manera local, esto se puede hacer, ejecutando en la raíz del proyecto el comando *node src/index.js*.

Nota: de ser necesario, en el material de la clase se encuentra un archivo llamado *C4a.AP.08. api_gateway.rar*, con el desarrollo del componente realizado en esta guía.