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_ibog@unaleduco

Actividad Práctica

(Componente Web - Parte 2)

El componente web desarrollado en el ciclo anterior consumía una *API* de tipo *REST*, por lo cual las funcionalidades implementadas como el registro, autenticación y obtener información de usuarios, fueron desarrolladas usando herramientas que permiten el consumo de servicios *REST*, pero dado que en este ciclo se tiene una *API* de tipo *GraphQL*, se deben reemplazar algunas piezas de código con el fin de permitir el consumo de los servicios expuestos por el *API Gateway*.

A lo largo de esta guía se editarán los archivos *SignUp.vue*, *LogIn.vue* y *Home.vue*, para poder consumir los servicios del API Gateway de manera correcta. En el caso del registro y la autenticación, se mantendrá la lógica del componente y únicamente cambiará la implementación de algunas funciones. En el caso del home, este se modificará para mostrar al usuario información que antes no se mostraba, lo cual requiere hacer una reestructuración completa del componente.

Modificando el Registro de Usuarios

El registro de usuarios fue implementado en el archivo *SignUp.vue*, en donde se tiene un formulario encargado de recolectar la información necesaria para el registro del usuario, este formulario está conectado a través de un *v-model* con el objeto *user*. Esto quiere decir que toda la información que recolecte el formulario en cada uno de sus campos, será almacenada en dicho objeto *user*. Además, el formulario utiliza una función para procesar su envío, esta función básicamente toma el objeto *user*, consume el servicio del correspondiente *Back-End*, y procesa la respuesta.

La lógica anterior se mantendrá intacta, y el cambio principal que se realizará será implementar la función que procesa el envío del formulario, pues ahora se consumirá un servicio de tipo *GraphQL*. Además, se realizarán algunos pequeños cambios sobre el objeto user y los campos del registro.

De esta forma, el código correspondiente al *template* del archivo *SignUp.vue* es el siguiente:

```
<template>

    <div class="signUp_user">
        <div class="container_signUp_user">
            <h2>Registrarse</h2>

            <form v-on:submit.prevent="processSignUp" >
                <input type="text" v-model="user.username" placeholder="Usuario">
                <br>

                <input type="password" v-model="user.password" placeholder="Contraseña">
                <br>
```

```
                <input type="text" v-model="user.name" placeholder="Nombre">
                <br>

                <input type="email" v-model="user.email" placeholder="Correo">
                <br>

                <input type="number" v-model="user.balance" placeholder="Saldo Inicial">
                <br>

                <button type="submit">Registrarse</button>
            </form>
        </div>

    </div>

</template>
```

El cambio es bastante sutil, ya que solo se cambia el *v-model* del campo balance, anteriormente se tenía asociado con el campo *user.account.balance* y ahora se tiene asociado con el campo *user.balance*. Esto se debe a un cambio en los datos que recibe el *Back-End* (en este caso, el API Gateway).

Ahora, el código correspondiente al *script* del archivo *SignUp.vue* es el siguiente:

```
<script>
import gql from "graphql-tag";

export default {
    name: "SignUp",

    data: function() {
        return {
        user: {
            username: "",
            password: "",
            name: "",
            email: "",
            balance: 0,
        },
        };
    },

  methods: {
    processSignUp: async function() {
```

```
      await this.$apollo
        .mutate({
          mutation: gql`
            mutation($userInput: SignUpInput!) {
              signUpUser(userInput: $userInput) {
                refresh
                access
              }
            }
          `,
          variables: {
            userInput: this.user,
          },
        })
        .then((result) => {
          let dataLogIn = {
            username: this.user.username,
            token_access: result.data.signUpUser.access,
            token_refresh: result.data.signUpUser.refresh,
          };

          this.$emit("completedSignUp", dataLogIn);
        })
        .catch((error) => {
          alert("ERROR: Fallo en el registro.");
        });
    },

  },
}
</script>
```

A pesar de que la estructura general del código es muy similar a la estructura implementada en el ciclo 3, existen grandes diferencias:

- La definición del objeto *user* se simplificó, esto se debe a que el *API Gateway* modificó la estructura del objeto que se recibe en la petición.
- Anteriormente se usaba la librería *axios* para consumir los servicios del microservicio *auth_ms*, en este caso se usará la librería *graphql* para consumir los servicios del *API Gateway*.
- El principal cambio es la implementación de la función *proccessSignUp*, esta se modifica completamente ya que *GraphQL* maneja un mecanismo personalizado para realizar peticiones, en este se debe definir de manera explícita la estructura de la petición, ya sea una *Query* o una *Mutation*, también se debe definir los datos de entrada y de donde se obtendrán. En este caso, el

dato de entrada será el objeto *user*, con esto se puede realizar la petición y procesar la respuesta. Este procesamiento es realmente sencillo y muy similar al anterior, simplemente se obtiene la información retornada por la petición, y se procesa el resultado (emitiendo un método del componente padre).

Por otro lado, el código correspondiente al *style* del archivo *SignUp.vue* no es modificado.

Modificando la Autenticación de Usuarios

El mecanismo de autenticación fue implementado en el archivo *LogIn.vue*, y sigue la misma lógica del mecanismo de registro, un formulario que recolecta la información y la guarda en un objeto *user*, y una función encargada de procesar el envío del formulario.

De esta forma, el código correspondiente al *template* del archivo *LogIn.vue* es el siguiente:

```
<template>

    <div class="logIn_user">
        <div class="container_logIn_user">
            <h2>Iniciar sesión</h2>

            <form v-on:submit.prevent="processLogInUser" >
                <input type="text" v-model="user.username" placeholder="Usuario">
                <br>
                <input type="password" v-model="user.password" placeholder="Contraseña">
                <br>
                <button type="submit">Iniciar Sesión</button>
            </form>
        </div>

    </div>

</template>
```

El cambio realizado no es funcional, pues tiene que ver únicamente con los textos que se muestran en la página web. Es importante tener en cuenta factores como la ortografía y el idioma utilizado a la hora de desarrollar una página web, pues son este tipo de aspectos los que pueden ver los usuarios y, por ende, los que demuestran la atención prestada por los desarrolladores en los detalles de la página web.

Por otro lado, los cambios realizados en el *script* del componente *LogIn.vue* son análogos a aquellos cambios realizados en el componente *SignUp.vue*, por lo cual, el código correspondiente a dicho *script* es el siguiente:

```
<script>
import gql from "graphql-tag";

export default {
    name: "LogIn",

  data: function() {
    return {
      user: {
        username: "",
        password: "",
      },
    };
  },

  methods: {
    processLogInUser: async function() {
      await this.$apollo
        .mutate({
          mutation: gql`
            mutation($credentials: CredentialsInput!) {
              logIn(credentials: $credentials) {
                refresh
                access
              }
            }
          `,
          variables: {
            credentials: this.user,
          },
        })
        .then((result) => {
          let dataLogIn = {
            username: this.user.username,
            token_access: result.data.logIn.access,
            token_refresh: result.data.logIn.refresh,
          };

          this.$emit("completedLogIn", dataLogIn);
        })
        .catch((error) => {
          alert("ERROR 401: Credenciales Incorrectas.");
        });
    },
  },
}
```

```
</script>
```

La estructura de este código es similar a la del componente *SignUp.vue*, es decir, que se crea una función en la que se define la *Mutation* correspondiente y se procesa su respuesta.

En cuanto al código correspondiente al *style* del archivo *LogIn.vue*, este no debe ser modificado.

Modificando el Home

La estructura actual del componente *Home* es bastante simple, ya que únicamente muestra un pequeño mensaje de bienvenida con el *username* obtenido del *localStorage*. Por esta razón, se modificará este componente para obtener y mostrar más información del usuario, como su nombre de usuario, nombre y correo electrónico. Para ello, es necesario reemplazar cada una de las partes del componente, incluido el *style*, a continuación, se muestran y explican cada una de las partes.

El código correspondiente al *template* del componente *Home.vue* es el siguiente:

```
<template>

  <div class="information">
    <h1>
      ¡Bienvenido
      <span>{{ userDetailById.name }}</span
      >!
    </h1>

    <div class="details">
      <h3>Su información es la siguiente</h3>

      <h2>
        Nombre de usuario:
        <span>{{ userDetailById.username }}</span>
      </h2>

      <h2>
        Correo electrónico:
        <span>{{ userDetailById.email }}</span>
      </h2>
    </div>
  </div>

</template>
```

El template cambia por completo, pero con los conocimientos actuales es realmente sencillo de comprender, simplemente se están mostrando una serie de valores usando los dobles corchetes (*{{ }}*), los cuales serán definidos en el *script*.

El código correspondiente al *script* del componente *Home.vue* es el siguiente:

```
<script>
import gql from "graphql-tag";
import jwt_decode from "jwt-decode";

export default {
  name: "Home",

  data: function () {
    return {
      userId: jwt_decode(localStorage.getItem("token_refresh")).user_id,
      userDetailById: {
        username: "",
        name: "",
        email: "",
      },
    };
  },

  apollo: {
    userDetailById: {
      query: gql`
        query ($userId: Int!) {
          userDetailById(userId: $userId) {
            username
            name
            email
          }
        }
      `,
      variables() {
        return {
          userId: this.userId,
        };
      }
    },
  }
};
</script>
```

En este, para obtener la información básica del usuario se realiza una *Query*. Como se puede notar, la declaración de esta y su uso, es un poco diferente a la declaración y uso de una *Mutation*, pues *Apollo* define un mecanismo simple e innovador para obtener la información. Para construir una *Query*, se deben definir múltiples elementos, el primero es un objeto en la *data* del componente llamado *userDetailById*, en este se almacenará la información retornada por la petición, y se accederá desde el template para mostrar los datos al usuario; el segundo elemento que se debe definir es una variable en la *data* del componente llamada *userId*, la cual servirá para conocer el *id* del usuario autenticado, a partir de los datos desencriptados del *refresh token*; finalmente, el último elemento que se debe definir es la estructura de la *Query* dentro de la llave *apollo* del componente, para ello se debe tener en cuenta varios aspectos:

- La llave del objeto declarado dentro de la llave *apollo* y el nombre del objeto en el que se almacenará la respuesta (definido en la *data*) debe ser **exactamente el mismo** que el nombre de la *Query* (El mismo nombre asignado en el API Gateway).
- En el objeto declarado dentro de la llave *apollo* (*userDetailById*) se configuran los parámetros de la *Query*, entre estos se encuentran la declaración detallada de la *Query* y la definición de las variables que va a utilizar la petición.
- Para definir las variables que va a utilizar la petición, se debe crear una función que retorne un objeto con las variables necesarias.
- Para declarar detalladamente la Query se utiliza *gql*, una dependencia que se encarga de traducir la sentencia para que *apollo* la interprete y la realice.

De esta forma se define una *Query* con *Apollo Client*. En cuanto a su funcionamiento, *apollo* realiza **automáticamente** la petición en 2 situaciones: cuando se carga la página por primera vez, o cuando detecta algún cambio en la(s) variable(s) de la *Query* (en este caso, la variable *userId*). Cuando la petición finaliza, el resultado se asigna al objeto con el mismo nombre de la petición (en este caso, el objeto *userDetailById*), por ello es importante que se utilice el mismo nombre en la declaración del objeto en la *data* del componente.

El código correspondiente al *style* del componente *Home.vue* es el siguiente:

```
<style>

.information {
  margin: 0;
  padding: 0%;
  width: 100%;
  height: 100%;

  display: flex;
  flex-direction: column;
  justify-content: center;
  align-items: center;
```

```
}

.information h1 {
  font-size: 60px;
  color: #283747;
}

.information h2 {
  font-size: 40px;
  color: #283747;
}

.information span {
  color: crimson;
  font-weight: bold;
}

.details h3 {
  font-size: 35px;
  color: #283747;
  text-align: center;
}

.details h2 {
  font-size: 35px;
  color: #283747;
}
.details {
  border: 3px solid rgba(0, 0, 0, 0.3);
  border-radius: 20px;
  padding: 30px 80px;
  margin: 30px 0 0 0;
}

</style>
```

En este caso es necesario modificar los *estilos* del *template* porque este último se rediseñó completamente.

En este momento no es posible ejecutar el servidor porque aún no se ha modificado el componente *Account.vue*, sin embargo, para analizar el *style* del componente *Home.vue*, a continuación se muestra el resultado de los cambios realizados:

Conclusiones

En el desarrollo de un sistema de software es muy común realizar *modificaciones* sobre *código existente*, ya sea porque existe una nueva versión de la aplicación, porque se busca mejorar el rendimiento o porque se usa una nueva herramienta, como en este caso. Es importante tener en cuenta que a pesar de que el código puede ser modificado por completo, muchas veces la *lógica* detrás del código antiguo es muy similar a la lógica del código nuevo, es importante entender estos detalles ya que facilitarán el proceso de evolución del sistema.

Nota: de ser necesario, en el material de la clase se encuentra un archivo llamado *C4a.AP.11. bank_fe.rar*, con el desarrollo del componente realizado en esta guía.