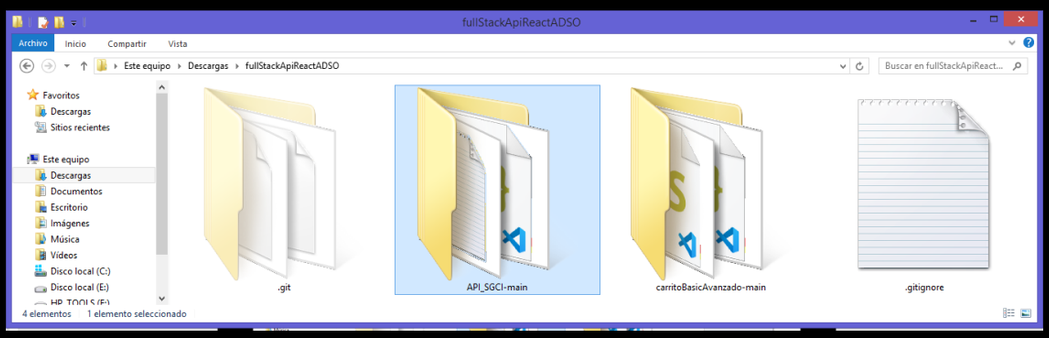
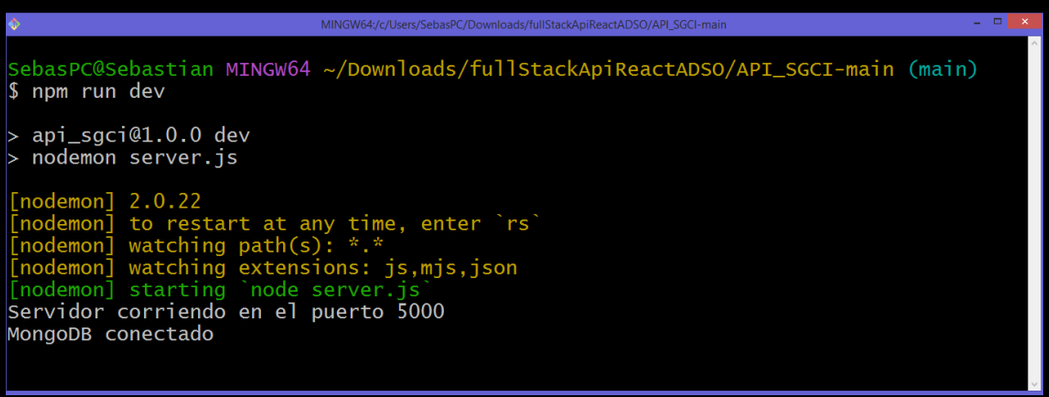
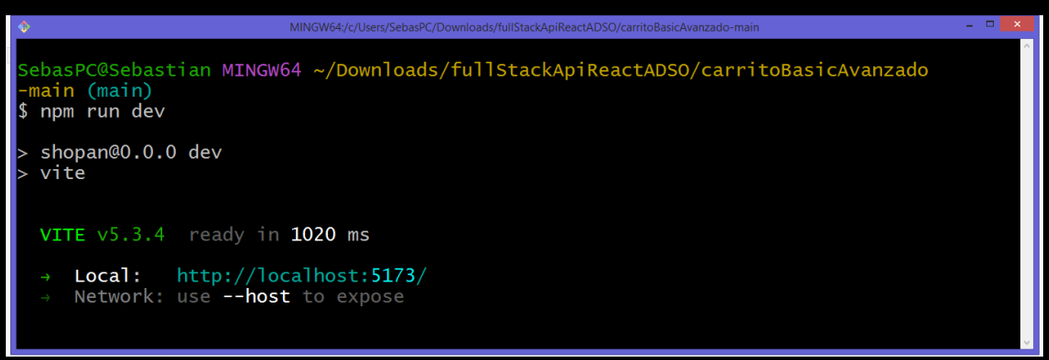
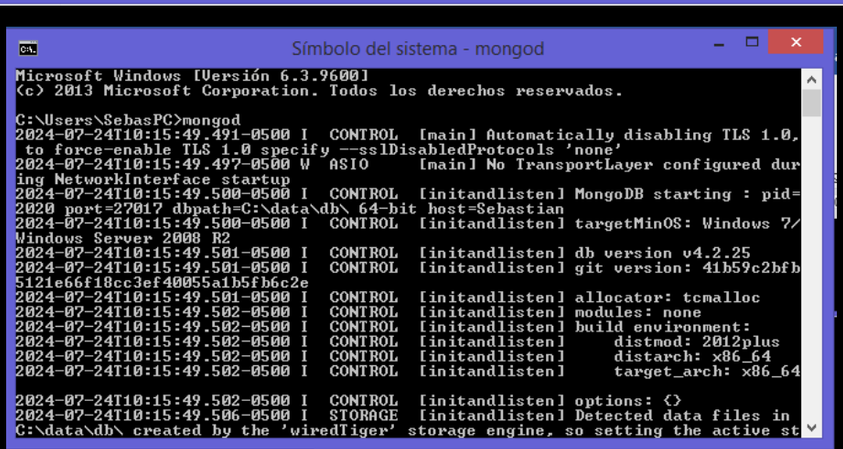
Parte Dos

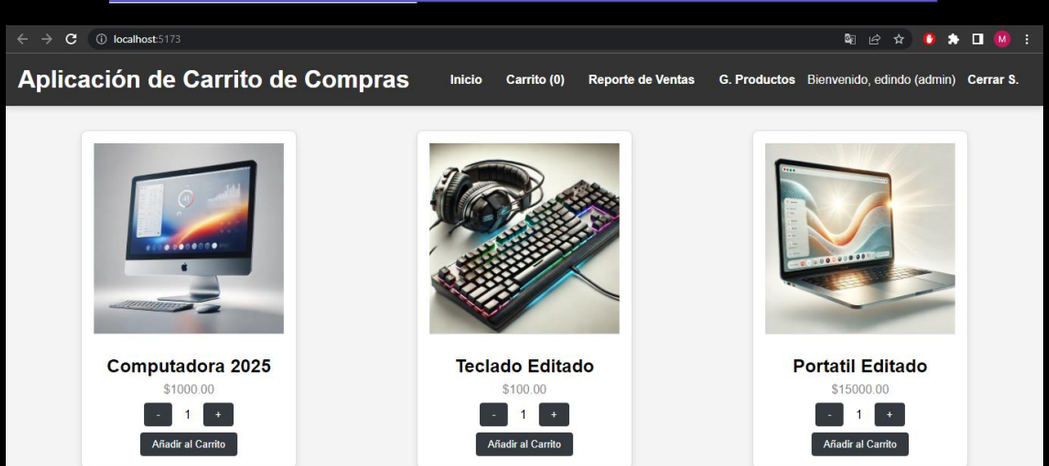
Continuamos con nuestra carpeta fullStackApiReactADSO, revisamos que los archivos este en orden y todo esté funcionando correctamente.











En nuestro FrontEnd - carritoBasicAvanzado-main

Vamos actualizar el archivo App.jsx importando el componente pedidos.

```
import Pedidos from './components/Pedidos';
```

Y creamos la ruta:

```
<Route path="/pedidos" element={<Pedidos />} /> {/* Nueva ruta para el componente Pedidos */}
```

Pueden ver la actualización en este link:

LINK: https://github.com/mellotak/fullStackApiReactADSO/commit/ec55545fc9765784d7fb5217369ad4d14e4a5a79#diff-93157c85182eb7c769f77d53d10c8c8ca73118946ace3337b1caffa14c4a575e

En nuestro FrontEnd – Carpeta Components

Vamos actualizar el archivo header.jsx , agregando el nuevo link de acceso al componente pedidos en nuestro menu.

```
            <Link to="/pedidos">Mis Pedidos</Link> {/* Enlace a los pedidos */}
```

Y vamos actualizar: la Palabra: “Aplicación de Carrito de Compras” por SGCI

Pueden ver la actualización en este link:

LINK: https://github.com/mellotak/fullStackApiReactADSO/commit/ec55545fc9765784d7fb5217369ad4d14e4a5a79#diff-1768cf7e2f1e55b795489fe3e26f9f9fd9ac5e65aa0175776eddf28109b8a8e7

En nuestro FrontEnd– Carpeta Components

Vamos a crear el nuevo componente llamado Pedidos.jsx,
Pueden ver el código en este link:

LINK: https://github.com/mellotak/fullStackApiReactADSO/blob/main/carritoBasicAvanzado-main/src/components/Pedidos.jsx

Explicación del código:

El código define un componente de React llamado Pedidos. Este componente es responsable de mostrar la lista de pedidos de un usuario específico, obteniendo estos datos de una API. Aquí están los pasos clave y funcionalidades que realiza:

1. **Importaciones**: Se importan las bibliotecas necesarias de React (useEffect y useState) y axios para realizar solicitudes HTTP.

2. **Estado del componente**: Se definen dos estados mediante useState: uno para almacenar los pedidos (pedidos) y otro para almacenar el ID del usuario (userId), que se obtiene del almacenamiento local del navegador (localStorage).
3. **useEffect**: Este hook se utiliza para ejecutar la lógica de obtención de pedidos cuando el componente se monta o cuando el userId cambia.
    - Dentro de useEffect, se define una función asincrónica fetchPedidos que realiza una solicitud GET a la API para obtener los pedidos del usuario.
    - Los pedidos obtenidos se ordenan por fecha de creación en orden descendente.
    - Los pedidos ordenados se almacenan en el estado pedidos.
    - Si ocurre algún error durante la solicitud, se captura y se imprime en la consola.
4. **Condición de existencia del usuario**: Antes de llamar a fetchPedidos, se verifica si userId existe. Si no existe, no se realiza la llamada a la API.
5. **Renderizado del componente**:
    - Se renderiza un contenedor con el título "Mis Pedidos".
    - Si hay pedidos en el estado, se mapean (iteran) para crear una representación visual de cada pedido en el DOM.
    - Cada pedido incluye detalles como el ID del pedido, la fecha del pedido, el estado, el código de pago, los productos dentro del pedido (incluyendo imagen, nombre, cantidad y precio), y la información de envío (nombre, teléfono, dirección, barrio, municipio, departamento y total del pedido).
    - Si no hay pedidos, se muestra un mensaje indicando "No tienes pedidos."
6. **Exportación del componente**: Finalmente, el componente Pedidos se exporta como el valor predeterminado del módulo, para que pueda ser utilizado en otras partes de la aplicación.

En resumen, este componente de React se encarga de obtener y mostrar una lista de pedidos del usuario, organizándolos y presentando información detallada sobre cada pedido de manera clara y ordenada.

En nuestro BackEnd - API_SGCI-main

Vamos actualizar el archivo pedidosRoutes.js que se encuentra en la subcarpeta ROUTES de la carpeta principal API_SGCI-main

En el archivo pedidosRoutes.js vamos a crear una nueva ruta con el siguiente código:

```
// Muestra todos los pedidos del cliente por su ID
router.get('/pedidos/cliente/:idCliente', pedidosController.mostrarPedidosCliente);
```

Pueden ver la actualización en este link:

LINK: https://github.com/mellotak/fullStackApiReactADSO/commit/ec55545fc9765784d7fb5217369ad4d14e4a5a79#diff-e424c68d62f345d8e17d884be147db4f8518fa228e2c1f729c55bf1c1b673b5a

En nuestro BackEnd - API_SGCI-main

Vamos actualizar el archivo pedidosController.js que se encuentra en la subcarpeta CONTROLLERS de la carpeta principal API_SGCI-main

En el archivo pedidosController.js vamos a crear un nuevo controlador llamado: mostrarPedidosCliente, así:

```
// Muestra todos los pedidos de un cliente específico
exports.mostrarPedidosCliente = async (req, res, next) => {
    try {
        const pedidos = await Pedido.find({ cliente: req.params.idCliente })
            .populate('cliente', '-password')
            .populate('pedido.producto');
        res.json(pedidos);
    } catch (error) {
        console.log(error);
        next();
    }
};
```

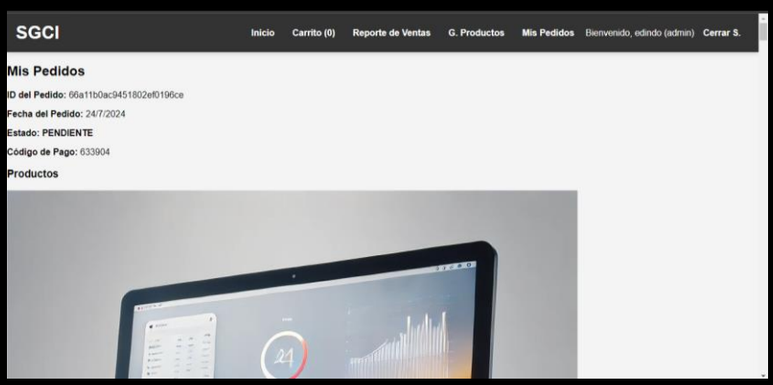
Pueden ver la actualización en este link:

LINK: https://github.com/mellotak/fullStackApiReactADSO/commit/ec55545fc9765784d7fb5217369ad4d14e4a5a79#diff-c05c79dcb65e72a8ec48660ab88018cf01c3ed16798d93741a76e4e68cd0f8db

Este código define una función llamada mostrarPedidosCliente que pertenece a un controlador en una aplicación de Node.js/Express. La función se utiliza para obtener y devolver todos los pedidos de un cliente específico desde la base de datos.

1. **Función Asincrónica**: mostrarPedidosCliente es una función asincrónica (async), lo que permite usar await para manejar operaciones asincrónicas como consultas a la base de datos.
2. **Parámetros de la Función**:
    - req (request): Objeto de solicitud HTTP que contiene información sobre la solicitud, incluyendo parámetros.
    - res (response): Objeto de respuesta HTTP que se utiliza para enviar una respuesta al cliente.
    - next: Función que se llama para pasar el control al siguiente middleware en caso de error.
3. **Consulta a la Base de Datos**:
    - Utiliza Pedido.find para buscar todos los pedidos que coincidan con el ID del cliente proporcionado en req.params.idCliente.
    - Usa populate para incluir detalles del cliente en los pedidos, excluyendo la contraseña por razones de seguridad.
    - También usa populate para incluir detalles de los productos en cada pedido.
4. **Enviar Respuesta**:
    - Si la consulta es exitosa, los pedidos obtenidos se envían como una respuesta JSON usando res.json(pedidos).
5. **Manejo de Errores**:
    - Si ocurre un error durante la consulta, se captura en el bloque catch, se imprime en la consola para depuración, y se llama a next() para pasar el control al siguiente middleware de manejo de errores.

Si ingresamos al menu “Mis pedidos”, tendremos:



De acuerdo a lo anterior es necesario agregar estilos

En nuestro FrontEnd – carritoBasicAvanzado-main

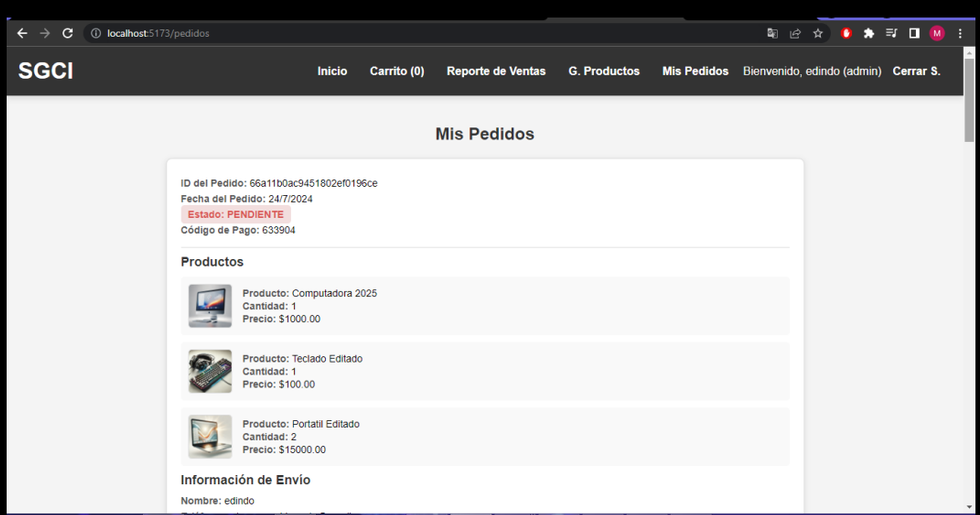
Vamos agregar los estilos /* Estilos para el componente Pedidos.jsx */

Pueden ver los estilos actualizados en este link:

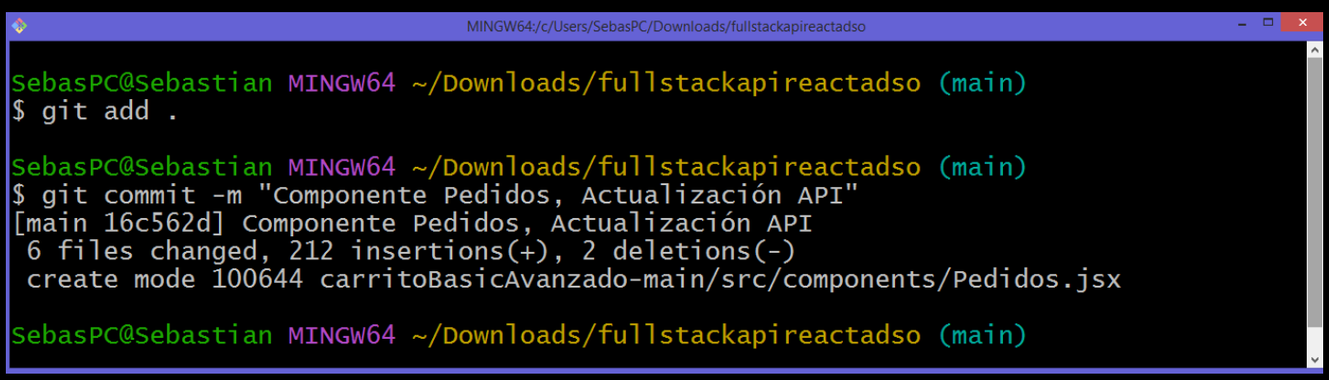
LINK: https://github.com/mellotak/fullStackApiReactADSO/commit/ec55545fc9765784d7fb5217369ad4d14e4a5a79#diff-e2d0e2470494feabfc8f05f1631d6c7f7502f00fae9d882acc1c630a1d7227a0

Explicación: Este código define los estilos CSS para un componente de React llamado Pedidos.jsx. El contenedor principal (.pedidos-container) tiene un ancho máximo de 900px y un diseño centrado, con márgenes y relleno específicos. Cada tarjeta de pedido (.pedido-card) tiene un borde, radio, sombra y efectos de transición para un aspecto moderno, y cambia su posición al pasar el cursor por encima. El encabezado y el pie de la tarjeta tienen bordes inferiores y rellenos para separar los elementos, y los párrafos dentro de estos tienen márgenes y tamaños de fuente ajustados. Los títulos dentro de la tarjeta están estilizados con tamaños de fuente y colores específicos. Los productos dentro de la tarjeta (.pedido-item) se muestran en un diseño flexible con alineación central, fondo claro y márgenes. Las imágenes de los productos tienen tamaños fijos, bordes redondeados y ajustes de borde. Los párrafos de los detalles del producto tienen márgenes y tamaños de fuente definidos. El título principal de la página de pedidos está centrado y estilizado con un margen inferior. Los textos en negrita tienen un color específico, y el estado del pedido (.estado) tiene estilos diferentes según su clase (pendiente, pagado, enviado), con colores de texto y fondo distintos para cada estado.

Quedando el siguiente resultado:



Creamos el commit y el resultado es:



COMPONENTE GESTOR DE PEDIDOS

En nuestro FrontEnd – carritoBasicAvanzado-main

Vamos actualizar el archivo App.jsx importando el componente ManageOrders.

```
import ManageOrders from './components/ManageOrders';
```

Y creamos la ruta:

```
<Route path="/manage-orders" element={<PrivateRoute><ManageOrders /></PrivateRoute>} /> {/* Ruta GP */}
```

Pueden ver la actualización en este link:

LINK: https://github.com/mellotak/fullStackApiReactADSO/commit/2f9bcc34f9d3ba89b1433ba949fed271d70390f6#diff-93157c85182eb7c769f77d53d10c8c8ca73118946ace3337b1caffa14c4a575e

Vamos actualizar el archivo header.jsx , agregando el nuevo link de acceso al componente pedidos en nuestro menu.

```
{user.role === 'admin' && <Link to="/manage-orders">G. Pedidos</Link>} {/* Link GP */}
```

Pueden ver la actualización en este link:

LINK: https://github.com/mellotak/fullStackApiReactADSO/commit/2f9bcc34f9d3ba89b1433ba949fed271d70390f6#diff-1768cf7e2f1e55b795489fe3e26f9f9fd9ac5e65aa0175776eddf28109b8a8e7

ManageOrders.jsx

Ahora vamos a construir el componente gestor de pedidos que se llamara: ManageOrders.jsx y lo construiremos en la carpeta components.

Pueden ver el código en este link:

LINK: https://github.com/mellotak/fullStackApiReactADSO/blob/main/carritoBasicAvanzado-main/src/components/ManageOrders.jsx

Explicación del código: Este código define un componente React llamado ManageOrders, que se encarga de gestionar y mostrar una lista de pedidos. El componente utiliza los hooks useState y useEffect para manejar el estado y los efectos secundarios.

Al principio, se importan las bibliotecas necesarias: React y axios, junto con un componente adicional ModalOrders que se usa para mostrar detalles de un pedido en un modal.

Dentro del componente, se definen varios estados:

- orders para almacenar todos los pedidos obtenidos de la API.
- filteredOrders para almacenar los pedidos que coinciden con el término de búsqueda.
- searchTerm para almacenar el texto ingresado en el campo de búsqueda.
- selectedOrder para almacenar el pedido que se selecciona para mostrar detalles.

El useEffect se utiliza para obtener los pedidos de la API al montar el componente. Se define una función asincrónica fetchOrders que realiza una solicitud GET a la API, ordena los pedidos por fecha de creación en orden descendente, y actualiza los estados orders y filteredOrders con los datos obtenidos. Si ocurre algún error durante la solicitud, se captura y se imprime en la consola.

La función handleSearch se encarga de filtrar los pedidos basándose en el término de búsqueda ingresado por el usuario. Convierte el término a minúsculas y filtra los pedidos por el nombre del cliente o el estado, actualizando filteredOrders.

El renderizado del componente incluye:

- Un contenedor principal con un título "Gestionar Pedidos".
- Un campo de entrada para buscar pedidos por nombre de cliente o estado.
- Una lista de pedidos filtrados, donde cada pedido se muestra en una tarjeta con detalles como ID del pedido, nombre del cliente, estado, código de pago y total. Al hacer clic en una tarjeta, se actualiza el estado selectedOrder para mostrar el modal.
- Un modal que se muestra si hay un pedido seleccionado, utilizando el componente ModalOrders, que permite ver y gestionar el pedido seleccionado, y actualizar los estados de los pedidos si se realizan cambios.

En resumen, este componente permite gestionar y visualizar pedidos, realizar búsquedas específicas, y mostrar detalles de un pedido seleccionado en un modal interactivo.

ModalOrders.jsx

Ahora vamos a construir el componente gestor de pedidos que se llamara: ModalOrders.jsx y lo construiremos en la carpeta components.

Pueden ver el código en este link:

LINK: https://github.com/mellotak/fullStackApiReactADSO/blob/main/carritoBasicAvanzado-main/src/components/ModalOrders.jsx

Este código define un componente de React llamado ModalOrders, que se utiliza para actualizar y eliminar pedidos específicos. A continuación, se explica cada sección del código:

Importaciones y Declaraciones Iniciales

1. **Importaciones**: Se importan las bibliotecas necesarias: React para manejar los componentes y axios para las solicitudes HTTP.
2. **Estados del Componente**:
    - orderToDelete: Estado que almacena el ID del pedido que se desea eliminar.

Función handleUpdateOrder

3. **Actualizar Pedido**: Esta función se llama cuando se hace clic en el botón "Actualizar". Realiza una solicitud PUT a la API para actualizar el pedido con la nueva información.
    - Si la solicitud es exitosa, actualiza los estados orders y filteredOrders con el pedido actualizado.
    - Después de la actualización, el modal se cierra estableciendo setSelectedOrder(null).

Función confirmDeleteOrder

4. **Confirmar Eliminación**: Esta función se llama cuando se hace clic en el botón "Eliminar". Establece orderToDelete con el ID del pedido actual para mostrar la confirmación de eliminación.

Función handleDeleteOrder

5. **Eliminar Pedido**: Esta función se llama cuando se confirma la eliminación del pedido. Realiza una solicitud DELETE a la API para eliminar el pedido.
    - Si la solicitud es exitosa, actualiza los estados orders y filteredOrders eliminando el pedido.
    - Después de la eliminación, el modal se cierra estableciendo setSelectedOrder(null) y orderToDelete(null).

Función handleCloseConfirmation

6. **Cerrar Confirmación**: Esta función se llama para cerrar el modal de confirmación sin eliminar el pedido, estableciendo orderToDelete(null).

Renderizado del Componente

7. **Renderizado del Modal**: El componente renderiza un modal con formularios para actualizar los detalles del pedido y botones para actualizar o eliminar el pedido.
    - **Formulario de Actualización**: Incluye campos para cambiar el estado del pedido y la información de envío (nombre, teléfono, dirección, barrio, municipio y departamento).
    - **Botones**: Incluye botones para actualizar el pedido y para mostrar la confirmación de eliminación.
8. **Confirmación de Eliminación**: Si orderToDelete tiene un valor, se renderiza un modal adicional para confirmar la eliminación del pedido con botones para confirmar o cancelar la acción.

INDEX.CSS

Ahora vamos Actualizar los estilos, en index.css, agregados estos:

```
/***************Estilos componente MangeOrders y ModalOrders.jsx ********************/
```

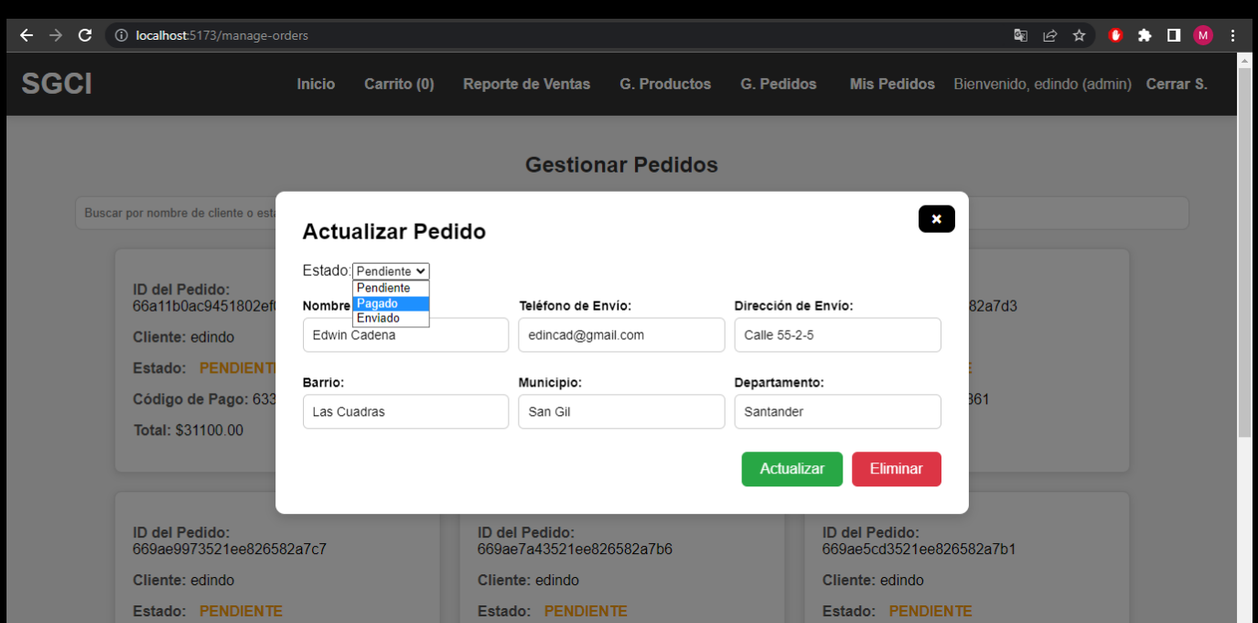
Pueden ver los estilos actualizados en este link:

LINK:

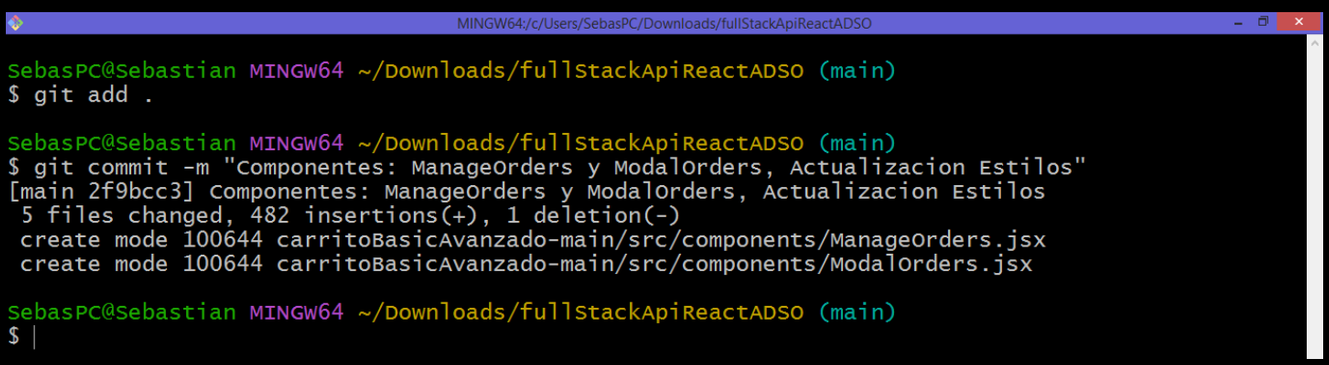
https://github.com/mellotak/fullStackApiReactADSO/commit/2f9bcc34f9d3ba89b1433ba949fed271d70390f6#diff-e2d0e2470494feabfc8f05f1631d6c7f7502f00fae9d882acc1c630a1d7227a0

Explicación del código: Este código define los estilos CSS para los componentes ManageOrders y ModalOrders.jsx. El contenedor principal (.gestionar-pedidos-contenedor) tiene un ancho máximo de 1200px, centrado con márgenes y relleno específicos. La barra de búsqueda (.buscar-input) ocupa el 100% del ancho, con relleno, margen inferior, borde y bordes redondeados. La lista de pedidos (.pedidos-lista) se muestra en un diseño flexible con espacio entre las tarjetas y justificación centrada. Cada tarjeta de pedido (.pedido-tarjeta) tiene un borde, bordes redondeados, relleno, ancho ajustado, fondo blanco, sombra y efectos de transición al pasar el cursor por encima. Las tarjetas cambian de color de fondo y se elevan ligeramente al ser seleccionadas. Para pantallas pequeñas, las tarjetas ocupan el ancho completo. El modal (.modal) se posiciona fijo, centrado, con un fondo semitransparente. El contenido del modal (.modal-contenido) tiene fondo blanco, relleno, bordes redondeados, sombra, ancho máximo de 700px y un botón de cierre con estilo específico y efectos de transición al pasar el cursor. El encabezado del modal (modal-contenido h3) tiene márgenes y tamaño de fuente específicos. Las filas del formulario (.form-fila) se muestran en un diseño flexible con espacio entre columnas, y cada columna (.form-columna) contiene etiquetas e inputs con estilos definidos. La fila de botones (.boton-fila) tiene botones con diferentes colores de fondo y efectos de opacidad al pasar el cursor. Los estados (.estado) tienen estilos específicos según su valor (pendiente, pagado, enviado). El modal de confirmación de eliminación (.modal-confirmacion) tiene estilos similares al modal principal, con un contenedor de contenido centrado, fondo blanco, relleno, bordes redondeados, sombra, y botones de confirmación y cancelación estilizados.

El resultado será el siguiente:



Creamos el commit y el resultado es este:



Ahora bien, vamos a construir el componente UpdateUser.jsx que nos permitirá que el usuario pueda actualizar su información.

COMPONENTE Actualización de Usuario: updateuser.jsx

En nuestro FrontEnd – carritoBasicAvanzado-main

Actualizamos nuestro app.jsx

```
import UpdateUser from './components/UpdateUser'; // Importar el componente de actualización de usuario
```

Y creamos la ruta:

```
<Route path="/update-user" element={<UpdateUser />} /> {/* Nueva ruta para actualizar el usuario */}
```

Pueden ver la actualización en este link:

LINK:
https://github.com/mellotak/fullStackApiReactADSO/commit/0d1c8241df08075ea5359ca455704791a052cb12#diff-93157c85182eb7c769f77d53d10c8c8ca73118946ace3337b1caffa14c4a575e

Vamos a actualizar el archivo header.jsx ,

- Cambiaremos este código:

```
<span>Bienvenido, {user.username} ({user.role})</span>
```

- Por este:

```
<span>  <Link to="/update-user">Bienvenido, {user.username} ({user.role})</Link></span>
```

Pueden ver la actualización en este link:

LINK:
https://github.com/mellotak/fullStackApiReactADSO/commit/0d1c8241df08075ea5359ca455704791a052cb12#diff-1768cf7e2f1e55b795489fe3e26f9f9fd9ac5e65aa0175776eddf28109b8a8e7

UPDATEUSER.JSX

Ahora vamos a construir el componente gestor de pedidos que se llamara: updateuser.jsx y lo construiremos en la carpeta components.

Pueden ver el código en este link:

LINK: https://github.com/mellotak/fullStackApiReactADSO/blob/main/carritoBasicAvanzado-main/src/components/UpdateUser.jsx

Explicación del código: Este código define un componente React llamado UpdateUser, que permite a los usuarios actualizar su información personal. A continuación se explica cada sección del código:

Importaciones y Declaraciones Iniciales

1. **Importaciones**: Se importan React, useState, y useEffect para manejar el estado y efectos en el componente, axios para realizar solicitudes HTTP, y useNavigate de react-router-dom para la navegación.
2. **Estados del Componente**:
   - user: Estado que almacena los datos del usuario (username, email, y password).
   - error: Estado para manejar errores.
   - navigate: Hook para la navegación.

useEffect

3. **Inicialización del Usuario**: useEffect se utiliza para cargar los datos del usuario desde localStorage cuando el componente se monta, asegurando que password siempre esté inicializado como una cadena vacía.

Función handleChange

4. **Manejo de Cambios en el Formulario**: Esta función se llama cada vez que se cambia un campo del formulario, actualizando el estado user con el valor ingresado.

Función handleSubmit

5. **Manejo del Envío del Formulario**:
   - Previene la acción por defecto del formulario.
   - Obtiene el token de autenticación desde localStorage.
   - Crea un objeto updatedUser con solo los campos que deben ser actualizados, comparando los valores con los almacenados en localStorage.
   - Verifica si hay datos que actualizar, si no hay, establece un mensaje de error.
   - Realiza una solicitud PUT a la API para actualizar los datos del usuario, incluyendo el token en los encabezados.
   - Si la actualización es exitosa, actualiza localStorage con los nuevos datos del usuario y navega a la página de inicio.
   - Si ocurre un error, se maneja estableciendo el estado error.

Renderizado del Componente

6. **Renderizado del Formulario**:
   - Muestra un título "Actualizar Datos del Usuario".
   - Si hay un error, muestra un mensaje de error.
   - Renderiza un formulario con campos para username, email, y password, todos con valores controlados por el estado user y manejadores de cambios para actualizar el estado.
   - Incluye un botón para enviar el formulario.

En nuestro BackEnd - API_SGCI-main

Vamos actualizar el archivo authController.js que se encuentra en la subcarpeta CONTROLLERS de la carpeta principal API_SGCI-main

En la línea 57, vamos agregar: ,email: user.email

Quedando el resultado así:

```
res.status(200).json({ mensaje: 'Inicio de sesión exitoso', token, username: user.username,
role: user.role, userId: user._id , email: user.email });
```

También a nuestro try, catch, del método updateUser, le agregaremos otro condicional, el cual es:

```
// Validar si el nuevo email ya está en uso
if (updates.email) {
  const existingUser = await User.findOne({ email: updates.email });
  if (existingUser && existingUser._id.toString() !== req.user.userId) {
    return res.status(400).json({ error: 'El email ya está en uso, usar otro' });
  }
}
```

Pueden ver la actualización en este link:

LINK:
https://github.com/mellotak/fullStackApiReactADSO/commit/0d1c8241df08075ea5359ca455704791a052cb12#diff-3e35483d3b1052556c362f1fa7a2e6148ccec987f3e4968751cb0639699b6b51

Explicación del código: Este fragmento de código es una parte de una función en un servidor que verifica si un nuevo correo electrónico ya está siendo utilizado por otro usuario antes de permitir que el usuario actualice su información.

1. **Validación del Email**: Primero, el código verifica si se ha proporcionado un nuevo correo electrónico para actualizar.
2. **Búsqueda en la Base de Datos**: Si se proporciona un nuevo correo electrónico, el servidor busca en la base de datos para ver si ya existe un usuario con ese correo.
3. **Comparación de Usuarios**: Si encuentra un usuario con ese correo, el código compara el ID de ese usuario con el ID del usuario que está haciendo la solicitud de actualización.
4. **Evitar Duplicación de Emails**: Si el correo ya está en uso por un usuario diferente, el servidor responde con un error, indicando que el correo electrónico ya está en uso y que debe usarse otro correo.

En nuestro FrontEnd – carritoBasicAvanzado-main

Vamos actualizar el archivo login.jsx y agregaremos en nuestro try, catch el siguiente código, que permitirá que cuando el usuario realice el Login se almacene el correo, para utilizarlo.

```
      localStorage.setItem('email', response.data.email); // Almacena el email
```

Pueden ver la actualización en este link:

LINK:
https://github.com/mellotak/fullStackApiReactADSO/commit/0d1c8241df08075ea5359ca455704791a052cb12#diff-e3f5cb11665a9390d1097ab2a9d2220c50a767d2647e4e08b94e82c8168f4871

En nuestro FrontEnd – carritoBasicAvanzado-main

Vamos actualizar nuestros estilos con:

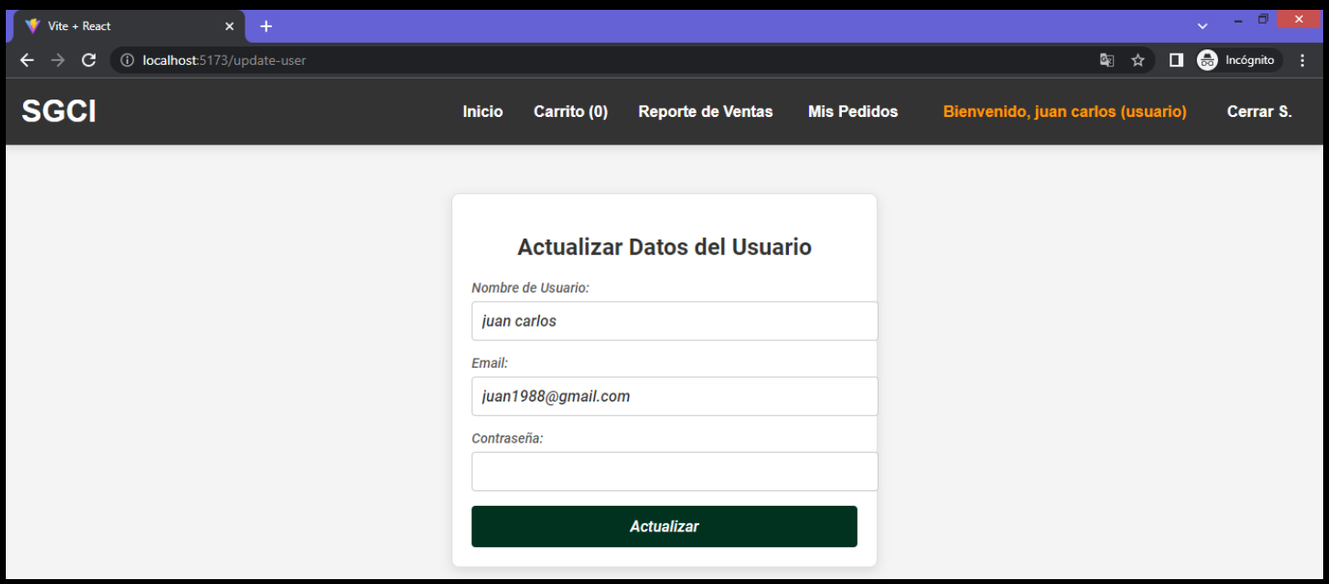
```
/***************Estilos componente UpdateUser.jsx ********************/
```

Pueden ver la actualización en este link:

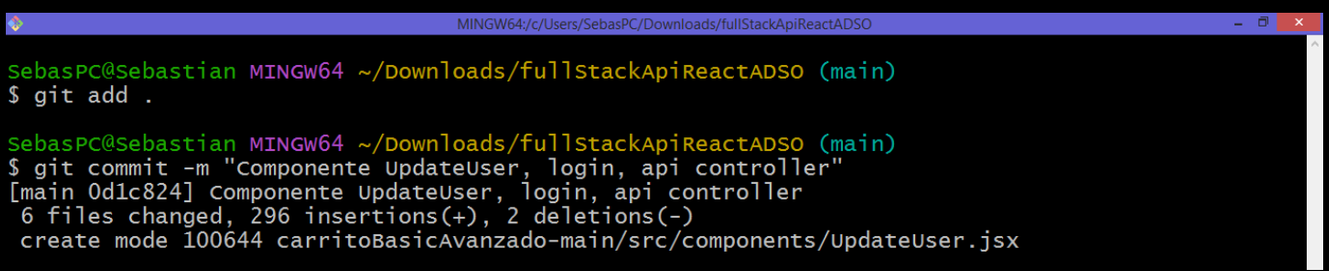
LINK:
https://github.com/mellotak/fullStackApiReactADSO/commit/0d1c8241df08075ea5359ca455704791a052cb12#diff-e2d0e2470494feabfc8f05f1631d6c7f7502f00fae9d882acc1c630a1d7227a0

Explicación del código: Este código CSS define los estilos para el componente UpdateUser.jsx. El contenedor principal del formulario tiene un ancho máximo de 400px, centrado con márgenes y relleno, fondo blanco, borde redondeado y sombra. El título del formulario está centrado, con un margen inferior y color específico. Los mensajes de error y éxito se muestran centrados y con colores rojo y verde, respectivamente. El formulario está diseñado para alinear sus elementos en el centro, con grupos de formularios que ocupan todo el ancho disponible y espaciados adecuadamente. Las etiquetas se muestran en bloque y con color y tamaño de fuente definidos. Los campos de entrada tienen relleno, borde redondeado, y cambian de color cuando están en foco. El botón de envío ocupa todo el ancho disponible, con un color de fondo verde oscuro y cambia a un verde más oscuro cuando se pasa el cursor. Los estilos responsivos aseguran que el formulario se adapte a pantallas pequeñas, ajustando márgenes, rellenos y tamaños de fuente. Los enlaces dentro de un span tienen un estilo específico, con texto en blanco y sin subrayado, cambiando de color al pasar el cursor.

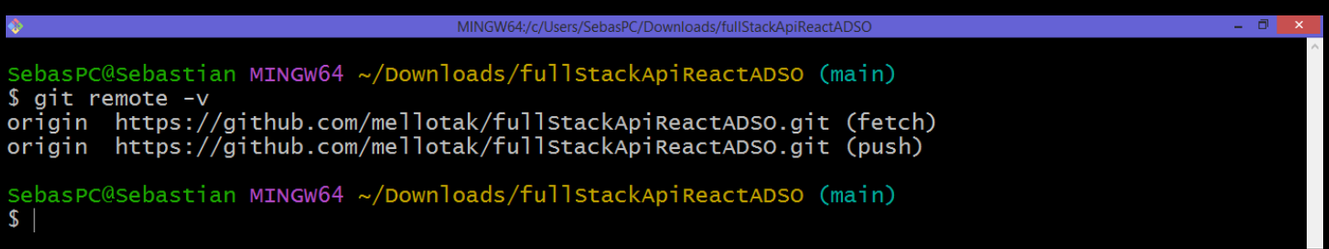
El resultado será algo como esto:



Creamos el commit y el resultado será el siguiente



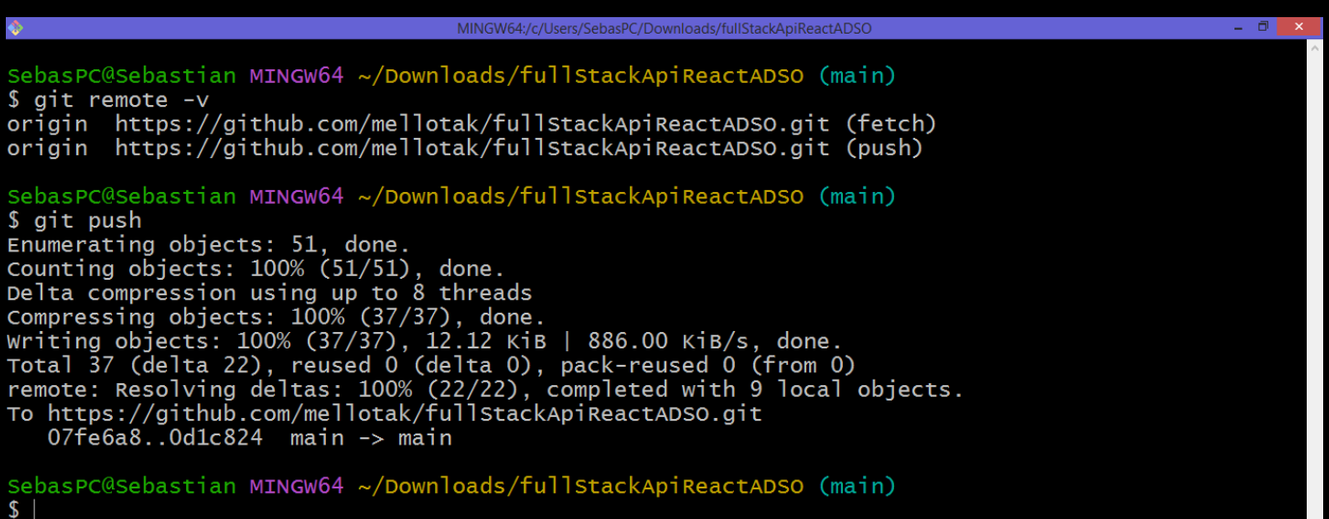
Ahora los vamos a subir a nuestro repositorio, para ello veamos donde lo subiría:



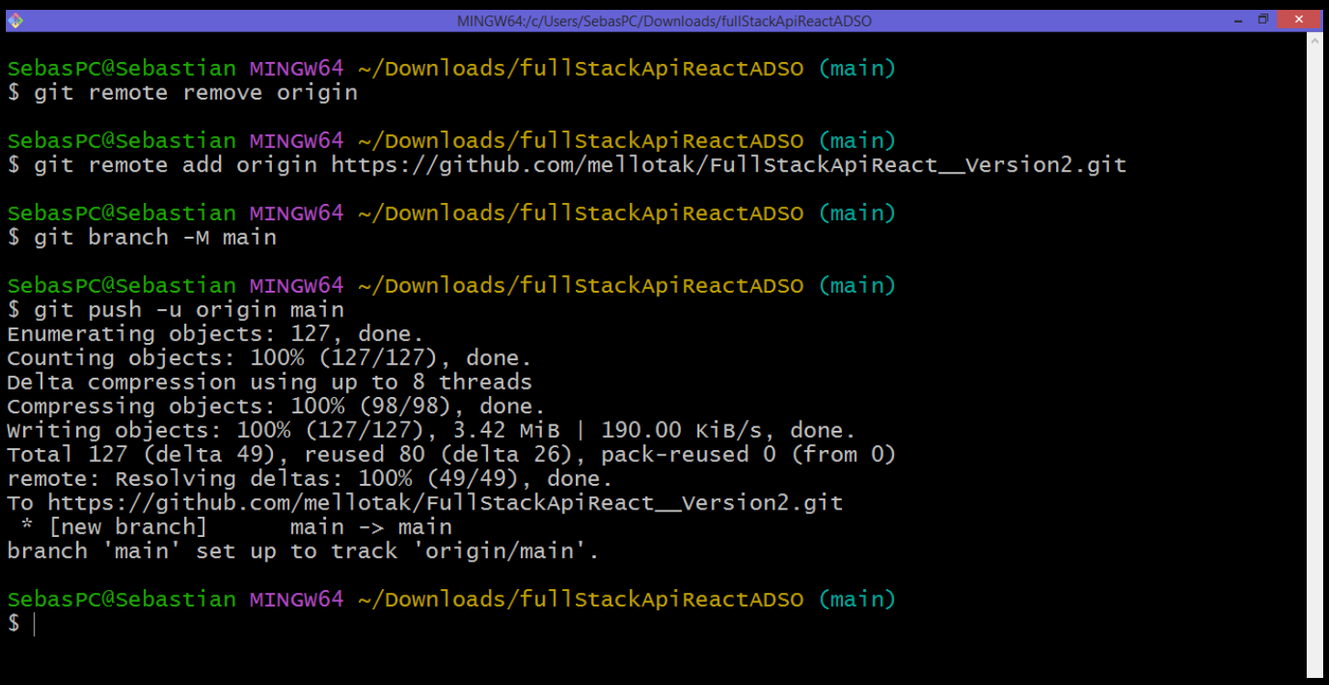
Revisamos si es nuestra ruta correcta, procedemos con el comando

```
git push
```

Resultado



Nota: Caso contrario la ruta no sea la deseada hacemos lo siguiente:



En este caso eliminamos la ruta de origen, creamos nuestro repositorio remoto y lo cargamos con los comandos correspondientes en la nube.