Actividad evaluable UT2A4 – CRUD con acceso a base de datos

RA1, RA3, RA4

En esta actividad evaluable realizarás la conexión con la base de datos para el Login y además, de forma guiada, la interfaz del CRUD, que estará alojada en localhost:5173/home. Esta interfaz nos debe permitir INSERTAR y BORRAR datos de la base de datos. Contendrá un menú, un formulario donde el usuario introducirá los datos con un botón para insertarlos en la base de datos y una tabla donde se mostrarán los datos de la base de datos en filas. Cada fila tendrá un botón para eliminar el registro de dicha fila de la base de datos.

¿Qué tengo que entregar?

¿Qué tengo que entregar? (Requisito)

- Enlace al github en la rama **crud**.
- Documento en formato PDF donde expliques en forma de **manual de usuario** (ver manual de usuario de ejemplo que está en el campus) cómo funciona tu aplicación. Se debe explicar de cara al uso de la aplicación por parte de un usuario. Debe tener portada e índice, el texto debe tener alineación justificada, la expresión escrita debe ser correcta.
- El proyecto será descargado por la profesora desde el github: es requisito indispensable que la aplicación se pueda ejecutar en cualquier equipo para poder ser corregida.
- No se admite código generado por IA (y sí, se nota ...)

¿Cómo identifico mi trabajo? (Requisito)

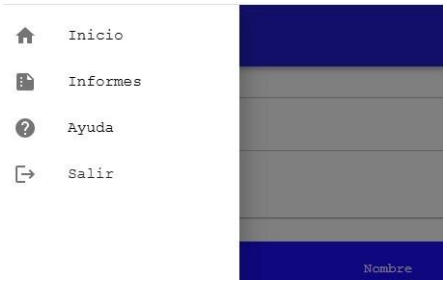
- Debes poner tu nombre y apellidos en el fichero README.md
- Debes nombrar el PDF como sigue:

PrimerApellido_SegundoApellido_Nombre_UT2A4

Aspecto final de la interfaz

El aspecto final de la página localhost:5173/home tiene que ser algo parecido a esto, con el estilo que ustedes hayan escogido:

Al picar en el botón del menú (arriba a la izquierda) aparece el menú de la aplicación:



Vemos que hay tres partes diferenciadas:

→ la **barra superior**: con un botón en la parte superior izquierda que nos lleva al menú con los enlaces a las diferente páginas y la opción de Salir de la aplicación, en el centro está el nombre de usuario que está logueado en ese momento, y en la parte superior derecha hay un icono que representa al usuario.

→ el **formulario**: para introducir los datos que luego insertaremos en la base de datos al picar en el botón insertar.

→ la **tabla**: donde se ven los registros de la base de datos y donde hay un botón que nos permite eliminar dicho registro.

El formulario y la tabla es recomendable que vayan en el mismo fichero puesto que el hecho de insertar datos en la base de datos hará que se actualice la tabla y los muestre. Pero la parte superior debe ir en un componente separado para luego reutilizarlo en otras páginas.

Desarrollo de la actividad

1. Creación de una nueva rama

Crea una nueva rama en tu proyecto: te sitúas en la carpeta proyecto*Tusiniciales* y ahí escribes el siguiente comando:

git checkout -b crud

Con esto estás creando una rama nueva llamada crud y además, te estás cambiando a esa rama.

Comprueba que estás en la nueva rama con el comando:

git branch

2. Conexión del login con la base de datos

Con lo que se explicó en clase sobre el backend y los endpoints, reemplaza tu comprobación de usuario y login correcto con la llamada al endpoint /login para conectarte a la base de datos y hacer la autenticación del usuario.

3. Creación del menú común a todas las páginas

El menú será común a todas las páginas de la aplicación: tanto si estamos en Inicio (en /home) como si estamos en Informes (/reports) querremos ver el mismo menú. Por lo tanto lo vamos a programar en un fichero que estará dentro de la carpeta components: en el **frontend/src/components/Menu.tsx**.

Tendrás que trasladar la parte de la navegación entre páginas (useNavigate, useDispatch, useSelector) a esa parte del código, puesto que será común en todas las páginas.

El menú debe ser *responsive* y debe aparecer el nombre del usuario que está logueado junto con un icono representativo del rol del usuario. Necesitarás usar, entre otros, los componentes AppBar para la barra superior (https://mui.com/material-ui/react-app-bar/) y Drawer que es lo que se abre cuando picas el botón hamburguesa (https://mui.com/material-ui/react-drawer/). Ayúdate de la documentación de MUI, hay infinidad de ejemplos. Tienes libertad para usar otros tipos de menús, siempre y cuando que seas capaz de modificar el código sin problema.

Lo único que debes tener en cuenta aquí es que para navegar entre las diferentes páginas (por ahora Inicio y Informes) cuando piques en el menú tienes que usar el componente <Link to /> de la librería react-router-dom. Este componente debe envolver al componente donde vayas a hacer la navegación. Te pongo dos ejemplos:

Ejemplo 1

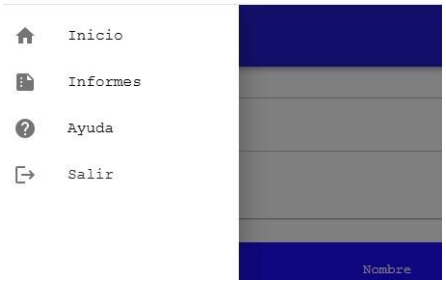
Si quieres poner un link en alguna parte del código que te lleve a la página http://localhost:5173/home, sería así:

```
<Link to='/home'>Inicio</Link>
```

En este caso te aparecerá en el código la palabra Inicio que será un enlace hay la página /home.
Es decir, lo que va dentro de `<Link to='/página a la que queremos ir'>` *Trozo de código que nos redirigirá a la página requerida cuando piquemos sobre él* `</Link>`

Ejemplo 2

En este ejemplo vamos a envolver tanto un botón de icono como un texto para que ambos me lleven a la página requerida cuando pique en ellos. Para verlo de forma más concreta: cuando se despliegue la lista del componente <Drawer/> quiero que al picar bien en el icono de la casa como en la palabra Inicio se vaya a la página http://localhost:5173/home



Debo envolver con el componente <Link /> la parte de código donde está el elemento de la lista de Inicio (Home):

```
const DrawerList = (
  <Box sx={{ width: 250 }} role="presentation" onClick={toggleDrawer(false)}>
    <List>
  //El componente <Link> es de la librería: react-router-dom. Sirve para ir a una página.
      <Link to={'/home'} style={{textDecoration:'none', color:'black'}}>
        <ListItem disablePadding>
          <ListItemButton>
            <ListItemIcon>
              <HomeIcon />
            </ListItemIcon>
          <ListItemText primary="Inicio" />
          </ListItemButton>
        </ListItem>
      </Link>
    // RESTO DE CÓDIGO
  </Box>
);

return (
  //RESTO DE CÓDIGO
)
}
```

4. Cómo evitar navegar a /home sin estar logueado: hook useEffect()

Hasta ahora si ponemos en la URL http://localhost:5173/home el enrutador nos montará el componente <Home>, que tendrá a su vez dentro el componente <Menu /> y el resto de componentes que tenga la página /home. Lo mismo pasa si ponemos a mano en el navegador la URL http://localhost:5173/reports , que el enrutador nos montará el componente <Reports> que a su vez tiene dentro el componente <Menu> y el resto de componente que tenga esa página. Es decir, entramos directamente en cualquiera de las dos páginas sin poner usuario ni contraseña.

Para evitar que pase eso se usa el *hook* useEffect(). (https://react.dev/reference/react/useEffect)

Como el componente común a ambas páginas (la de /home y la de /reports) es el **<Menu /> que creamos en el punto 3, es ahí donde tendremos que colocar el hook useEffect()**.

Este hook permite sincronizar un componente cuando cambie algo externo. Ese algo externo se representa con las dependencias. Es decir, si cambian las dependencias, yo haré una acción en mi componente. Esas dependencias deben estar dentro de la lógica del useEffect(). En nuestro caso, si no estamos autenticados, queremos que se navegue a /. Por lo tanto las dependencias son estar o no autenticados y navegar.

Esta es la estructura de uso del *hook* useEffect:

```
useEffect(() => {
   //Lo que queremos hacer
}, [dependencias])
```

Para saber si estamos o no autenticados, usamos los datos que obtuvimos del store con el useSelector(). Esos datos están almacenados en el objeto userData (o como lo hayan llamado ustedes en su código). Recordemos que ese objeto tiene los siguientes elementos, que nos dicen el nombre de usuario, el rol del usuario y si está o no autenticado:

```
userData.userName
userData.userRol
userData.isAutenticated
```

Con el hook useEffect() evitamos que se monte la página /home o la página /reports, puesto que reaccionará cuando se cambie el userDara.isAutenticated, comprobará si es false y si lo es, navegaremos a http://localhost:5173/ (es decir, la ruta padre /).

```
//Para usar el useEffect debemos importarlo
import { useEffect } from 'react'

//Trozo de código donde vamos a usar el useEffect(): siempre los hooks van al principio del componente
//Antes de esto tendremos que coger del store los datos
const isLoggedin = userData.isAutenticated

useEffect(() => {

 if (!isLoggedin) {
   navigate('/')
 }
 }, [isLoggedin, navigate])
```

Comprueba ahora que al poner http://localhost:5173/home o http://localhost:5173/reports directamente en la URL no te deja entrar puesto que no estás logueado.

5. Creación del componente <Dashboard> que usarás en Home.tsx

Crea un componente llamado Dashboard en el fichero **frontend/src/components/Dahsboard.tsx**. Este componente tendrá el formulario y la tabla que se usarán en la página /home.

Es decir, ahora el fichero Home.tsx renderizará los componentes <Menu> y <Dashboard> en su return:

```
//Aquí importamos el componente <Menu>
//Aquí importamos el componente <Dashboard>

export default function Home(){
return (
  <>
    <Menu />
    <Dashboard />
  </>
)
}
```

6. Creación del formulario en Dashboard.tsx

Ya has creado varios formularios a lo largo del curso, así que esto no debería ser complicado. Ten en cuenta que el formulario debe ser *responsive*.

Recuerda que todos los elementos del formulario: nombre, tipo, marca y precio los debes poner en un useState. Como estamos trabajando con Typescript deben indicar los tipos de los diferentes elementos, para ello hay que crear una interface:

```
///Creamos el tipo itemtype. Este tipo será un objeto con un id opcional de tipo number
//nombre, marca y tipo de tipo string y el precio de tipo number
interface itemtype {
  id?: number
  nombre: string
  marca: string
  tipo: string
  precio: number
}

//Inicializo los valores del item. Aquí no pongo el id porque no lo necesito
const itemInitialState: itemtype = {
    nombre: '',
    marca: '',
    tipo: '',
    precio: 0
}

//Cuando declaremos el useState del item en nuestro código:
const [item, setItem] = useState(itemInitialState)
```

El botón de + INSERTAR DATOS de nuestro formulario debe insertar datos en la tabla **coleccion** de nuestra base de datos.

Para ello debemos responder al evento onSubmit realizando una llamada al *endpoint* correspondiente.

7. Creación del fichero backend/services/items.js

Ahora vamos a saltar al backend de la aplicación para **crear las consultas que manejarán la comunicación con la tabla coleccion** de la base de datos bdgestion.sql. Contenido incompleto del fichero que irán rellenando:

```
const db = require('./db')
const helper = require('../helper')
const config = require('../config')

//Función con la consulta para insertar datos en la base de datos: INSERT
async function insertData (req, res) {
    //data tiene los datos que vamos a insertar en la base de datos. Los coge de req.query
    //Para acceder a cada uno de los datos: data.nombre, data.precio, ...
   const data = req.query
   const result = await db.query(
     `AQUÍ VA LA CONSULTA`
)
/*En la variable result se almacena lo que devueve la consulta. Si accedemos a effectedRow nos da el número de filas de la base de datos que ha sido modificado o añadido. Si ese número es mayor que cero es que ha habido inserción en la base de datos.*/
  return result.affectedRows
}

//Función con la consulta de obtener datos de la base de datos: select * from coleccion
async function getData (req, res) {
 //La variable rows almacena los datos obtenidos de la consulta select.
 const rows = await db.query(`
     `AQUÍ VA LA CONSULTA`
)
/*Los datos obtenidos de la consulta del select los paso por la función helper para que en el caso de que no haya datos devueltos, me devuelva un array vacío. */
const data = helper.emptyOrRows(rows)
return {
//Devolvemos el resultado del Select, que está almacenado en la variable data
  data
   }
}

//Función con la consulta para borrar datos de la base de datos: DELETE
async function deleteData (req, res) {
  //En data almaceno los datos que me pasan para poder realizar el delete, me pasarán el id.
  const data = req.query
  const result = await db.query(
    `AQUÍ VA LA CONSULTA`
)
/*En la variable result se almacena lo que devueve la consulta. Si accedemos a effectedRow nos da el número de filas de la base de datos que ha sido borrado. Si ese número es mayor que cero es que ha habido borrado en la base de datos.*/
  return result.affectedRows
}
//Al final del fichero exporto las funciones getData, insertData y deleteData
module.exports = {
   getData,
   insertData,
   deleteData
}
```

8. Añadir funcionalidad al botón Insertar del formulario.

Cuando se pique en el botón Insertar, hay que insertar los datos en la base de datos y avisar al usuario que se han insertado correctamente.

a) Programar la consulta **INSERT** en la función insertData del fichero **backend/services/items.js**

b) Añadir en el fichero **backend/index.js** un *endpoint* para añadir un registro en la base de datos. Yo al *endpoint* le puse el nombre /addItem pero lo pueden nombrar como quieran. Fíjate en el *endpoint* /login, es muy parecido el código. El código incompleto sería así:

```
app.get('/addItem', async function(req, res, next) {
  try {
    res.json(await AQUÍ LLAMO A LA FUNCIÓN insertData que está en el fichero items y le paso req)
  } catch (err) {
    console.error(`Error while inserting items `, err.message);
    next(err);
  }
})
```

c) En el frontend llamar al backend con la función fetch() cuando se pique el botón de Insertar. En la función que maneje el botón Insertar, debemos hacer un fetch al endpoint /addItem. Aquí fijarse en el fetch que hicieron en el login, es muy parecido. Si la respuesta del fetch es > 0, lanzamos un alert indicando que hemos insertado los datos correctamente (alert('Datos guardados con éxito')).

9. Creación del resto de los *endpoints*: /getItems y /deleteItem

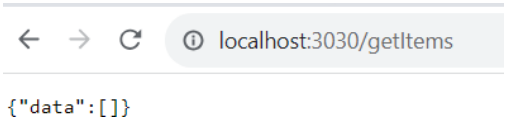
9.1 Obtener datos de la base de datos (/getItems)

a) Programar la consulta SELECT en la función getData del fichero backend/services/items.js

b) Añadir en el fichero backend/index.js el *endpoint* para seleccionar los datos de la tabla colección. Yo al *endpoint* le puse el nombre /getItems pero lo pueden nombrar como quieran. El código incompleto sería así:

```
app.get('/getItems', async function(req, res, next) {
  try {
    res.json(await AQUÍ LLAMO A LA FUNCIÓN getData que está en el fichero items)
items.getData())
  } catch (err) {
    console.error(`Error while getting items `, err.message);
    next(err);
  }
})
```

Para probar que funciona vamos al *endpoint* /getItems a través del navegador (recuerden que los endpoints están en el puerto 3030): http://localhost:3030/getItems
Veremos cómo son los datos que nos devuelve la base de datos. Si en la tabla **coleccion** la base de datos aún está vacía nos devuelve un objeto vacío:



Si insertamos algún registro en la tabla **coleccion** veremos que nos devuelve un objeto data que tiene dentro un array de objetos. Cada objeto es un registro de la tabla colección:

9.2 Borrar un registro de la base de datos (/deleteItem)

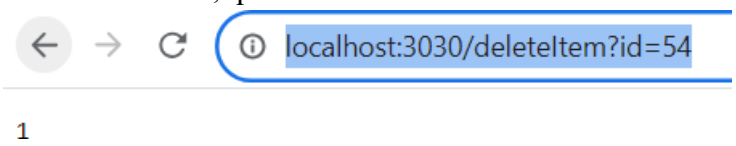
a) Programar la consulta DELETE en la función `deleteData` del fichero backend/services/items.js

b) Añadir en el fichero backend/index.js el *endpoint* para seleccionar los datos de la tabla **coleccion**. Yo al *endpoint* le puse el nombre /deleteItem pero lo pueden nombrar como quieran. El código incompleto sería así:

```
app.get('/deleteItem', async function(req, res, next) {
  try {
    res.json(await AQUÍ LLAMO A LA FUNCIÓN deleteData que está en el fichero items y le paso req)
  } catch (err) {
    console.error(`Error while deleting items `, err.message);
    next(err);
  }
})
```

Para probar que funciona vamos al *endpoint* /deleteItem a través del navegador en el puerto 3030 y le indicamos que queremos borrar el registro con id 54: http://localhost:3030/deleteItem?id=54

Veremos que nos devuelve un número, que es el número de filas borradas:



10. Creación de la tabla en el fichero Dashboard.tsx: lógica para mostrar los datos en la tabla y lógica para borrar un registro de la tabla

10.1 Lógica para mostrar los datos en la tabla

Con la tabla mostramos los datos que están almacenados en la tabla **coleccion** de nuestra base de datos. Para mostrar los datos tenemos que hacer un SELECT de la tabla colección, almacenar dichos datos en una variable (que será un array) y luego mostrar los datos de la variable en la tabla.

Así que lo primero que tendremos que hacer es crear una variable en el fichero **frontend/src/components/Dashboard.tsx** en la que almacenaremos los datos obtenidos del SELECT. Como esa variable va a actualizarse cada vez que hagamos el select, la ponemos como un useState. Su valor inicial será un array vacío. Cuando hagamos el select, su valor se actualizará con los valores obtenidos de la base de datos.

```
const [tableData, setTableData] = useState([])
```

Para pensar ...
¿En qué parte del frontend hacemos fetch() al endpoint que hace el SELECT? ¿Cómo lo hacemos?

Una vez tenemos claro en qué parte hacemos el fetch() al *endpoint* que realiza el SELECT y tenemos ya almacenados los datos de la tabla coleccion en la variable tableData … vamos a crear la tabla.

10.2 Implementación de la tabla: componente Table

Para la tabla usamos el componente Table de la librería @MUI: https://mui.com/material-ui/react-table/

Una tabla tiene la siguiente estructura básica:

→ Contenedor de tabla (TableContainer)
→ La tabla en sí (Table)
→ La cabecera de la tabla (TableHead) con un componente fila (TableRow) y luego cada una de las celdas (TableCell) de la fila de la cabecera.
→ El cuerpo de la tabla (TableBody) que tendrá varios componentes fila con varias celdas cada uno de ellos.

En nuestro caso vamos a rellenar el cuerpo de la tabla con los datos de la base de datos, que vienen en un array de objetos. Para recorrer un array en javascript se usa el método map como ya saben. Así pues, vemos a continuación el esquema de una tabla y la parte donde se haría el .map del array:

```
<TableContainer>
  <Table aria-label='Nombre Tabla para accesibilidad'>
    <TableHead>
      <TableRow>
        <TableCell></TableCell>
          .
          .
          .
        <TableCell></TableCell>
  </TableRow>
</TableHead>
<TableBody>
        método .map para recorrer los objetos del array y ponerlos en las filas de la tabla.
</TableBody>
</Table>
<TableContainer>
```

Fijarse que tenemos una propiedad llamada **aria-label**, que nos sirve para temas de accesibilidad, eso facilita la lectura de la aplicación a una persona con problemas de visión. **Hay que ponerlo en el código.**

Repito aquí el ejemplo del punto 9.1 en donde vimos qué devuelve el *endpoint* /getItems dentro de data:

Lo que queremos hacer ahora es mostrar con el componente <Table> esos datos obtenidos del *endpoint* /getItems y almacenados en el array tableData.
Nos centramos ahora **en lo que se pondría dentro del Body de la tabla**, que es un .map con el que recorremos el array tableData (siempre y cuando haya datos, si no los hay no se entra en el map). La variable row toma el valor de los elementos del array a medida que se va recorriendo. Así que row.id será el valor del id, row.nombre será el valor del nombre, …
Fíjense que raw es de tipo itemtype, que fue el que definimos al principio del código de Home.tsx.
Fíjense también que para el key usamos el id que proviene de la base de datos.

```
<TableBody>
 {tableData.map((row: itemtype) => (
   <TableRow key={row.id}>
    <TableCell>
      <Button onClick={() => handleDeleteItem(row)}>
        <DeleteForeverIcon />
         </Button>
    </TableCell>
    <TableCell>{row.nombre}</TableCell>
    //AÑADIR EL RESTO DE CELDAS
   </TableRow>
 ))}
</TableBody>
```

10.3 Lógica para borrar un registro de la tabla

En el código del TableBody hemos insertado un componente <Button> con un componente icono (<DeleteForeverIcon />). Al picar en dicho botón icono, tendremos que implementar una función a la que pasamos el id del registro que queremos borrar. Dicha función hará el fetch al *endpoint* /deleteItem. Ahora les toca a ustedes …

PUNTUACIÓN

Entrega	Nota MÁXIMA
1) El ejercicio se entrega en plazo y resuelto correctamente.	10
2) Igual que el 1) pero fuera de plazo. Máximo 24 horas	6
3) No se admiten entregas fuera de plazo cuando se han pasado más de 24 horas de la fecha límite de entrega.	0
4) Si no se cumplen las especificaciones de entrega la tarea irá directamente a recuperación.	0

En la tabla se indican las notas máximas en cada situación.