crmReact

v151. crmReact completo

- El proyecto incluye
  - React Router Dom (versión 6)
  - Tailwind CSS
  - Formularios hechos con Formik y con Yup (?)
- Tiene varias páginas
- ¿Qué es el routing?
- Crearemos un una API con JSON Server
- Veremos como enviar peticiones a un servidor
- La aplicación incluye un CRUD hecho en React al 100%
- Los formularios incluyen diversos tipos de validación

v152. Primeros pasos

- Instala contexto con vite
- Lo limpia

v153. Instala Tailwind CSS

crm-react> npm i autoprefixer postcss tailwindcss crm-react> npx tailwindcss init -p

- En *tailwind.config.js*

```
module.exports = {
  content:
    ['index.html', 'src/**/*.jsx'],
  ...
};
```

(En el video pone *purge* en vez del *content* de la v3 de Tailwind)

- En *index.css* importa:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

- Reinicia el servidor vite
- En el `<body>` del index.html agrega:

```
<body class='min-h-screen bg-gray-100'>
```

v154. Routing

- Con una **librería de Routing** pueden tenerse varias URLs
- Se pueden mostrar varios componentes
- Se puede restringir el acceso a ciertas páginas
- Los proyectos grandes se manejan mejor en múltiples pantallas

Librerías de Routing

- React Router + Reach Router (v6)
- React Location
- Gatsby
- Next.js
- Instala React Router v6
  - cmr-react> npm i react-router-dom
- En *App* importa:

```
import {BrowserRouter, Routes, Route} from 'react-router-dom';
```

Cada página está en *Route*

v155. Define rutas y usa

- Crea `...src/layout`
  - `...src/layout/IniciarSesion.jsx`
  - `...src/layout/Layout.jsx`
- Crea `...src/pages`
  - `...src/pages/Inicio.jsx`
  - `...src/pages/EditarCliente.jsx`
  - `...src/pages/NuevoCliente.jsx` (Pongo *pages* por el icono de la carpeta. En el video pone *paginas*)
- Los componentes en *layout* son los principales. Los que están en *pages* son los que se usan en las rutas y se accede a ellos con `<Route>` a medida que vamos navegando.
- En *App* usamos BrowserRouter para registrar diferentes endpoints (crear un nuevo routing).

```
<BrowserRouter>
  <Routes>
    // Rutas anidadas
    <Route path='/' element={...}>
      // todas las rutas aquí pertenecen a la ruta raíz que las encierra
      <Route />
    </Route>
```

```
    </Routes>
  </BrowserRouter>
```

- Cuando el usuario visite una ruta, React Router va a encontrar el componente que le corresponde a través de `<Route />`
- Para acceder al `element` *IniciarSesion* primero hay que importarlo: `import IniciarSesion from './layout/IniciarSesion.jsx'`

```
<BrowserRouter>
      <Routes>
        <Route path="/" element={<IniciarSesion />}>

        </Route>
      </Routes>

      <Routes>
        {/* Agrupa las rutas en su interior */}
        <Route path="/clientes" element={<Layout />}>
          <Route element={<Inicio />}/>
        </Route>
      </Routes>
    </BrowserRouter>
```

- Cuando el usuario visita la página principal (`/`) va a cargar el componente *IniciarSesion*.
- Cuando visita `/clientes` cargará el componente *Layout*.
- Para que se muestre el componente *Inicio* dentro del marco de *Layout*, en *Layout.jsx* hay que poner la etiqueta `<Outlet />`: ese es el lugar en el que aparece el Route anidado *Inicio*. Pero también hay que ponerle el atributo *index* en *App.jsx* para que se cargue por defecto:

```
  <Route ... >
    <Route index element={<Inicio />}/>
  </Route>
```

- Para editar clientes (asociar la página con un ítem) se hace:

```
  <Route path="/clientes/:id" element={<EditarCliente />}/>
```

:id es un placeholder, un parámetro de la ruta.

- Elimina el grupo `<Routes>` que contiene el componente *IniciarSesion*, *IniciaresSesion.jsx* y *LoginForm.jsx*

v157. Tailwind CSS al Layout principal

...

v158. Enlaces a través de `<Link>`

`<a>` -> `<Link>` `href` -> `to`

v159. Hook useLocation()

> Cómo detectar la página en la que se encuentra el usuario

- Importa `useLocation()` de `react-router-dom` en *Layout.jsx*
- La sintaxis es

```
const location = useLocation()
```

- useLocation() devuelve un objeto con todas las propiedades de la ruta actual (pathname, search, hash, etc.).

v160. Páginas del proyecto

- *Formulario.jsx*

v161. Formik y otras librerías para formularios

- Las librerías se usan con formularios grandes, complejos o múltiples.
- Tienen validaciones robustas.
- Formik usa Yup para validar los datos.
- React Hook Form tiene validaciones propias
- En este video se usa Formik. Lo instala con `npm i formik yup`
- En *Formulario.jsx* importa:

```
import {Formik, Field, Form} from 'formik'
```

- Para convertir un *Field* en un textarea hay que ponerle el atributo `as="textarea"`

```
<Field as="textarea" ... />
```

v162. Valores iniciales en Formik

```
<Formik
  initialValues={{
    nombre: '',
    email: '',
    telefono: '',
    notas: ''
  }}
```

```
    ...
  />
```

- Cada una de las keys debe ser el *name* del input correspondiente.
- El *Form* debe ser rodeado por una arrow function.

v163. Leer un submit y administrarlo en Formik

```
  <Formik
    ...
    onSubmit={values => {
      console.log(values)
    }}
  />
```

`values` es un objeto con todos los valores llenados en el formulario.

v164. Validación con Yup

```
import * as Yup from 'yup'
```

en *Formulario.jsx*

- Crea una variable *schema* con el objeto *Yup* con los todos los campos y la forma que deben tener esos campos.
- En *Formik.jsx* ponemos el objeto *schema* como parámetro de `validationSchema`

```
  const nuevoClienteSchema = Yup.object().shape({
    nombre: Yup.string().min(3, 'El nombre es muy corto').max(20, 'El nombre es
muy largo').required('El nombre es obligatorio'),
    email: Yup.string().email('El email no es válido').required('El email es
obligatorio'),
    telefono: Yup.string().required('El teléfono es obligatorio'),
    notas: Yup.string()
  })
```

A la constante *nuevoClienteSchema* hay que asociarla a los valores del formulario. En *Formik* ponemos el objeto *nuevoClienteSchema* como parámetro de `validationSchema` en `<Formik ... validationSchema={nuevoClienteSchema}>`

- En *Formulario.jsx* importa `ErrorMessage` de *Formik* con el texto para mostrar errores. En este caso, se usa `<ErrorMessage name="nombre" />` que es el error de `nombre`.

- Igual, no lo usa(!) porque a *ErrorMesage* no se le pueden poner atributos CSS. En cambio, usa `errors.nombre` para mostrar el error de `nombre`, etc.

- *Formik* trae muchísima información sobre los campos del formulario, una parte son los mensajes de error `errors`. Como en Yup está definido `nombre: ...` en *nuevoClienteSchema*, en *Formik* se puede acceder a esos mensajes de error con `errors.nombre`.
- En la arrow function que rodea a `<Form ...>` se le pasa la desestructuración `{errors}` del *data* completo de Formik.
- Debajo del *Field* nombre pone

```
  {errors.nombre && touched.nombre ? (
    <div className="text-center my-4 bg-red-600 text-white font-bold p-3
uppercase">
      {errors.nombre}
    </div>
    ) : null
  }
```

- `touched` es un objeto que devuelve `true` si el campo fue tocado pero no se escribió nada. Hay que pasárselo en la desestructuración `{errors, touched}` en la arrow function.
- Las clases CSS se pueden poner en el div.
- Mueve el mensaje de error a su propio componente *Alerta.jsx* que define en la carpeta *components* que levanta `{children}`
- En *Formulario.jsx* importa *Alerta.jsx*
- `{children}` obliga a poner una etiqueta de apertura y otra de cierre en *Formulario*: `<Alerta> {errors.nombre} </Alerta>`).

v165. JSON-Server

¿Qué es una REST API?

- REST API: REpresentational State Transfer
- REST API: Representación estado de transferencia
- API: Application Programming Interface
- API: Interfaz de programación de aplicaciones
- Debe responder a los Request HTTP: GET, POST, PUT, PATCH, DELETE
- Es una forma ordenada y estructurada de poner a disponsición los recursos
- Es una forma de comunicación entre dos o más aplicaciones
- **HTTP:**
  - GET: Obtener datos
  - POST: Enviar datos al servidor o para crearlos
  - PUT / PATCH: Actualizar datos
  - DELETE: Borrar datos
- **Endpoints:** Son los recursos que se pueden acceder a través de una API
  - Una REST API tiene *Endpoints* o URLs para hacer operaciones CRUD
  - Listar todos los clientes -> `GET /clientes`

    - Obtener un solo cliente -> `GET /clientes/15`
    - Crear un cliente -> `POST /clientes`
    - Actualizar / editar un cliente -> `PUT /clientes/5`
    - Borrar un cliente -> `DELETE /clientes/8`

Creación de una API REST con JSON-Server

- Para crear una API REST, se usa una herramienta llamada *JSON-Server* y se puede hacer en menos de 30 segundos!
- system32> npm install -g json-server
    - en Powershell con atributos de administrador

v166. Base de datos en la API

- Crea *db.json* en la raíz del proyecto

```
{
  "clientes": []
}
```

- crm-react> json-server --watch db.json --port 3001
    - --watch: cuando se modifique el archivo, se actualiza la API. Recarga cada vez que hay cambios
    - --port 3001: la API corre en el puerto 3001
    - Resources http://localhost:3001/clientes: los recursos están en la URL http://localhost:3001/clientes
- *db.json* se irá llenando con los datos que se inserten de la API

v168. Envía los datos desde Formulario a la API

- Convierte el handleSubmit de *Formulario.jsx* en una función asíncrona
- Usa un try - catch para capturar los errores (debbuguear)
- Para crear un cliente, se usa `POST /clientes` a la URL http://localhost:3001/clientes

```
  handleSubmit = async (values) => {
    try {
      const url = 'http://localhost:3001/clientes'
      const respuesta = await fetch(url, {
        method: 'POST',
        body: JSON.stringify(values),
        headers: {
          'Content-Type': 'application/json'
        }
      })
      console.log(respuesta)
      const resultado = await respuesta.json()
      console.log(resultado)
    } catch (error) {
      console.log(error)
```

```
    }
  }
```

v169. Redirecciona al usuario para que no pueda a cargar los mismos datos una y otra vez

- En *Formulario.jsx*, en el argumento de onSubmit de *Formik* escribe `{resetForm}` al lado de `values` y resetea luego de correr `handleSubmit(values)`:

```
<Formik
  ...
  onSubmit={async(values, {resetForm}) => {
    await handleSubmit(values)
    resetForm()
  }}
  ...
>
  ...
```

Espera que se complete `handleSubmit(values)` y luego ejecuta `restetForm()` para resetear el formulario.

- Luego de resetear se puede dirigir al usuario a la página principal de la API (*Inicio*), donde se muestran todos los clientes cargados.
- Para redirigir se usa *useNavigate* de *react-router-dom*
- Importa *useNavigate* en *Formulario.jsx*
- lo usa:

```
const navigate = useNavigate()
...
const resultado = await respuesta.json()
navigate('/clientes')
```

v170. Recupera los clientes consultando la API

- La lista de clientes se muestra en la pestaña de *Clientes* (en *Inicio.jsx*)
- Para recuperar los clientes, se usa `GET /clientes` a la URL http://localhost:3001/clientes
- Se usa *useEffect* para recuperar los clientes

```
useEffect(() => {
  const consultarApi = async () => {
    const url = 'http://localhost:3001/clientes'
    const respuesta = await fetch(url)
    const clientes = await respuesta.json()
    setClientes(clientes)
  }
```

```
    consultarApi()
  }, [])
```

- Almacena todo lo que trae en el feth en el state `clientes`

v171. Muestra los clientes en una tabla

v173. Elegir un cliente de la tabla y mostrarlo

- Cuando el usuario hace click en Ver hay que dirigirlo a otra pantalla.
- En *Cliente* importa `{useNavigate}` de react-router-dom
- Declara el hook: `const navigate = useNavigate()`. `navigate` es una función.
- Define el `onClick` del botón Ver:

```
...
onClick={() => navigate(`/clientes/${id}`)}
...
```

- Crea el componente *VerCliente.jsx* en `...src/pages/VerCliente.jsx`
- En *App.jsx* importa el componente *VerCliente* y define una nueva ruta hacia él:

```
...
<Route path="/clientes" element={<Layout />} />
  ...
  <Route path=":id" element={<VerCliente />} />
  ...
</Route>
...
```

Como id tiene dos puntos al inicio (`:id`) se trata como una variable.

- *VerCliente* es el componente que visitará el usuario cuando escriba `/clientes/:id`
- ¿Cómo se recupera el id del cliente al hacer click en el botón Ver?
    - Se hace con el hook *useParams* del *react-router-dom* que recupera los parámetros de la URL.
    - Importa useParams en *VerCliente* y define el hook: `const params = useParams()`
        - `params` siempre contiene actualizados los parámetros de la URL
        - Desestructura `params` para obtener el id del cliente: `const {id} = useParams()`

react-router-dom, lo que hay que saber

- Cómo crear el Router
- Rutas anidadas
- Layouts principales
- Uso de los hooks que llegan a ser muy útiles para redireccionar (useNavigate) o leer datos de una URL (useParams)

v174. Muestra toda la información de un cliente

- Importa useEffect y useState en *VerCliente*
- Una vez que está listo el componente se hace la consulta asíncrona de `obtenerClienteAPI` a la API por única vez (con un try catch). Eso se logra con useEffect
- Obtiene una `respuesta` con un await fetch apuntando a la URL de la API específica de ese cliente `.../clientes/${id}` y recupera el `resultado` en forato JSON
- Lo que queda en `resultado` se almacena en el objeto state `cliente`
- Con los datos en el state se renderiza como siempre

v175. Soluciona parpadeo al mostrar los datos

- *VerCliente* hace un parpadeo al mostrar los datos por primera vez o al recargarlos
- Lo soluciona mostrando `Cargando...` (se podría usar un spinner)que controla a traves del state `cargando`
- Cuando el componente está listo se van a buscar los datos. Ahí se produce el parpadeo.
- Para solucionarlo, se usa una condición en el render:
    - Si `cargando` es `true` se muestra `Cargando...`
    - Si `cargando` es `false` se muestra el contenido

v178 - ... Edición de clientes

`enableReinitialize={true}` permite que los campos del formulario Formik se rellenen con los datos que vienen de "afuera". Si está en `false` por más que se asignen los datos a los campos del formulario Formik, estos no se llenarán.

v183. Actualiza registros en la API

- Trabaja dentro del try-catch del *Formulario*
- Si `cliente` viene vacío está creando, si `cliente` viene lleno está editando.

v184. Eliminar registros

- El registro debe eliminarse tanto del DOM como de la API.
- Para que desaparezca del DOM se usa `setClientes()`.
- Para confirmar la eliminación existe en JavaScript la función `confirm()`:

```
const confirmar = confirm('¿Estás seguro de eliminar este cliente?')
```

que saca una alerta con el texto que se le pasa y devuelve `true` o `false` según lo que presione el usuario.

- Para borrar de la API se usa el método *DELETE* del `fetch` (lo usual, con async-await, etc.). Este método exige pasarle el id (`fetch(`http://localhost.../${id}`, {method: 'DELETE'})`)
- Una vez borrado el registro de la API se podría actualizar la página volviendo a cargar los datos de la API, pero esto hace pedidos innecesarios a la base de datos. Mejor actualizar el state.
- Para actualizar el state se puede usar `.filter(...)` sobre el array que contenga la lista de clientes.

v185 - 186. Deployment de la API