**¿Qué es el routing?**

Actualmente todas las apps creadas son de una sola pantalla.

Con una librería de Routing puedes tener diferentes URL’s y mostrar diferentes componentes, así como restringir acceso a ciertas páginas.

Un proyecto grande es mejor manejarlo en múltiples pantallas, en lugar de un solo componente que revise múltiples condiciones.

**React Router DOM ¿Qué es?**

**React Router DOM** es una librería para crear aplicaciones con routing (diferentes URL’s).

Al tener diferentes páginas tendremos un mejor orden, podemos seguir re-utilizando componentes y más.

A partir de la versión 6.4, se convierte en un framework con manejo de rutas, peticiones HTTP, formularios y datos.

**155. Creado el router y definiendo primeras rutas.**

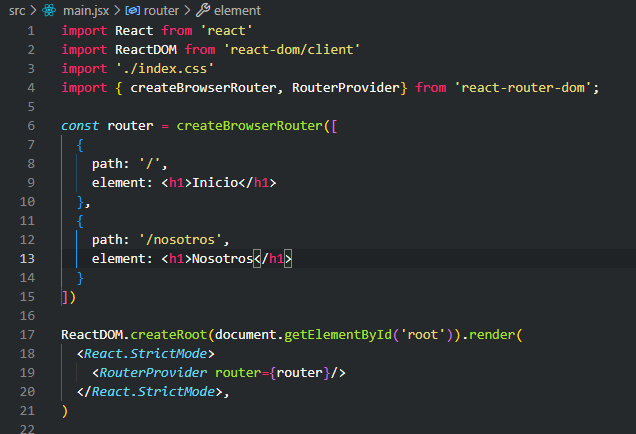
Instalamos la dependencia



Y en el main.jsx hacemos los cambios

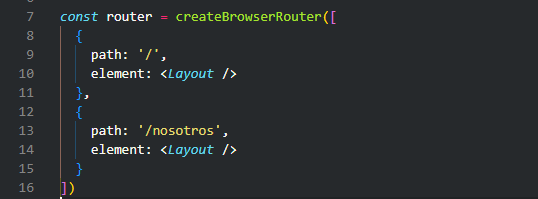


Otra página

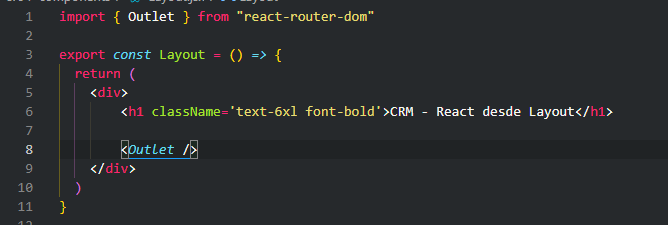


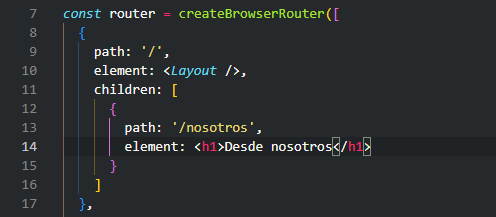
**156. Creando páginas y componentes y que es outlet**

Como en otras ocasiones si queremos tener componentes y usarlas en el router, vamos a importar el componente y usarla en la parte de element.



Ahora si queremos añadir código HTML en ese componente usamos la etiqueta <Outlet /> y de esa etiqueta en adelante ponemos el código, el código a inyectar viene desde element



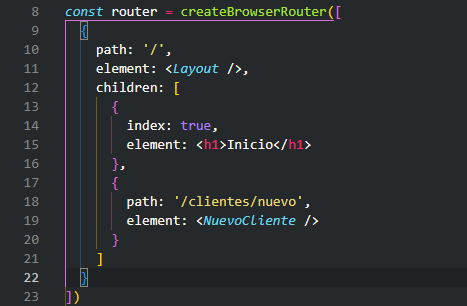


**OJO AL ELEMENTO CHILDREN**

Lo de element se va a renderizar después de la etiqueta Outlet en layout pero en este caso solo en el path de nosotros.

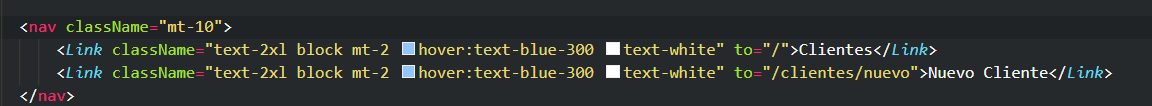


Y en el element podemos igual incluir componentes.



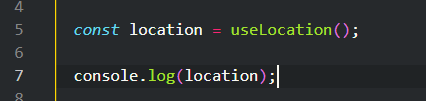
**158. Navegar entre componentes con Link**

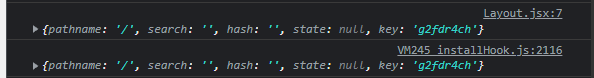
Tenemos que importar Link de react-router-dom y en lugar de usar <a href=””> usamos la etiqueta Link como si pueda un componente, con apertura y cierre. Hay otra etiqueta que se comporta igual que se llama NavLink que igual se tendría que importar.



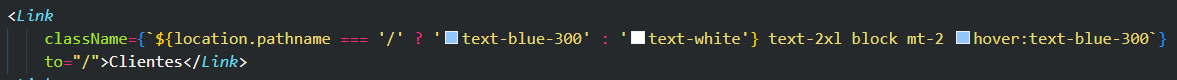
**159. Resaltar la página actual**

Vamos a usar un hook de react-router-dom, que se llama useLocation, lo importamos en layout y lo instanciamos



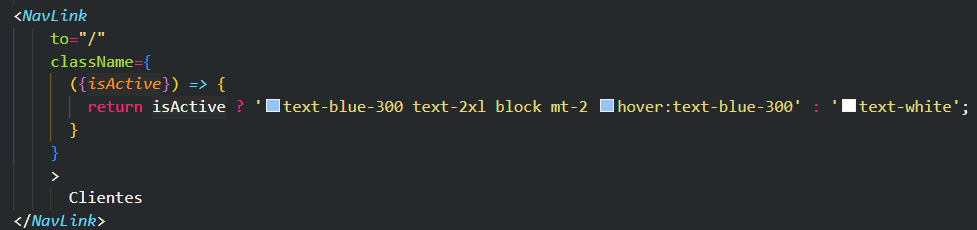


Entonces para poder resaltar el link actual hacemos uso de ternarios para evaluar el pathname que es la ruta de la página actual o en la que nos encontremos, entonces la evaluamos con el parámetro que queremos recibir



Y si cumple le agregamos la clase de color, si no un text-white.

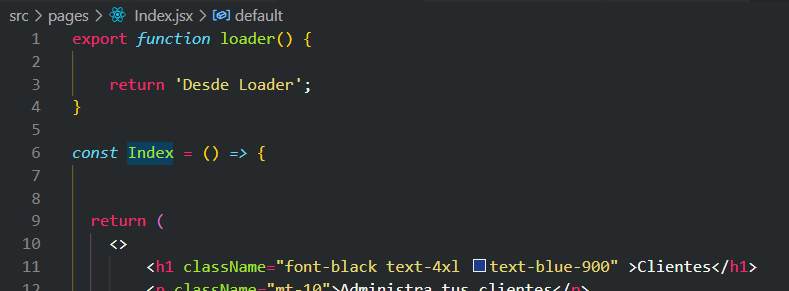
También se puede hacer con NavLink, solo que para evaluar usa una función



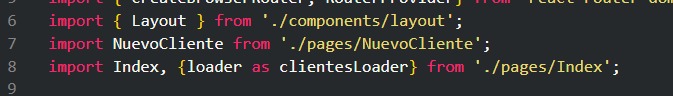
**160. Creando un Loader**

Un loader es como un useEffect.

Creamos la función loader en el archivo donde lo vamos a ocupar, en este caso Index.jsx y siempre tiene que retornar algo

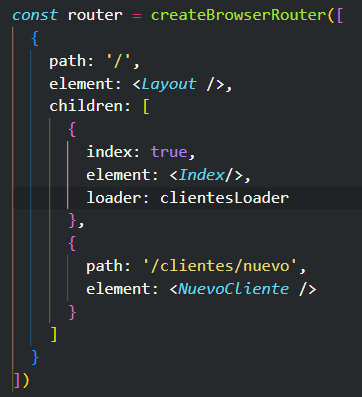


En el main.jsx lo importamos con un alias para que no tengamos problemas si en otro componente usamos un loader.





Y para poderlo usar directamente en el index, en el router del index le indicamos que el loader es clientesLoader



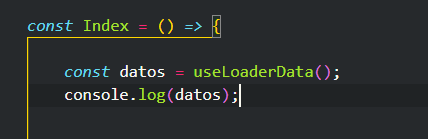


**161. Obtener los datos del loader con useLoaderData**

Se necesita importar un Hook



Y con el accedemos a lo que retorna el loader





El loader puede retornar cualquier tipo de datos, en este caso con el guist vamos a retornar un arreglo de objetos con clientes

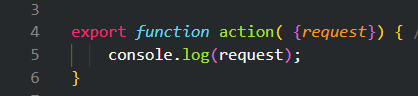
**Actions y Loaders**

Utiliza **loaders** para obtener datos de una API o de un objeto, similar a un state

Utiliza **Actions** para procesar la entrada de datos de un form.

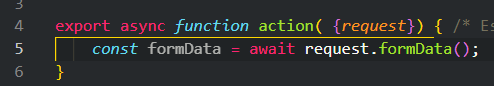
**167. Leer la información del Form con formData**

En el action para el form recibe un request o petición, la cual en consola nos da todas las opciones que incluye





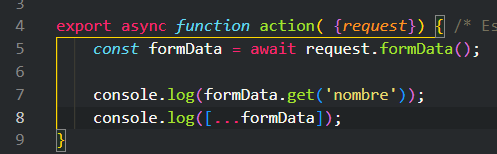
En el prototype tenemos formData

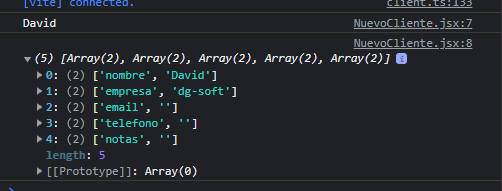
q

Hacemos la función asíncrona porque el request puede tardar en retornar los datos del Form

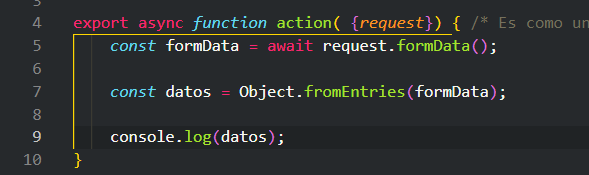
**OJO: Hacerle console.log al formData no va a imprimir nada**

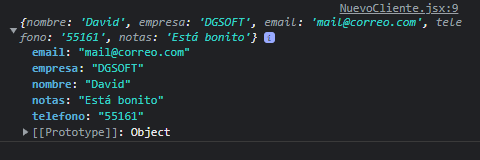
Las formas de leerlo



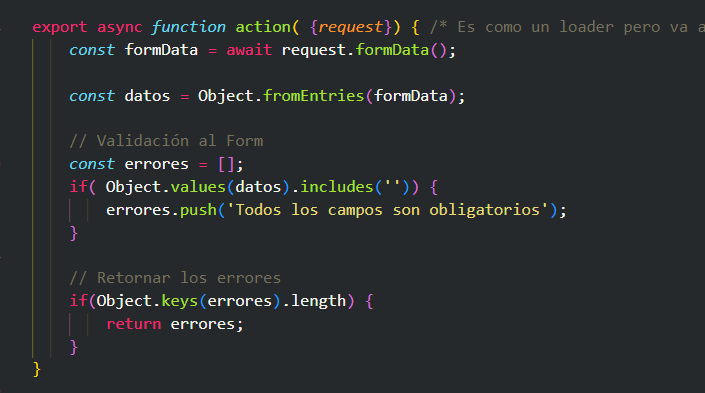


**Otra forma aún mejor**

****

****

**168. Añadiendo validación al formulario**

****

El primer if evalua si los datos incluyen un string vacio, en caso afirmativo se añade al arreglo de errores ese mensaje.

En el segundo if se valida si errores tiene algo y los retorna.

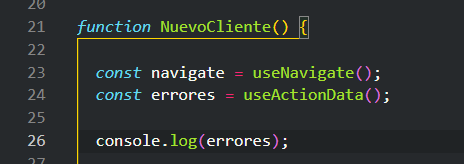
Pero… **¿Cómo recuperamos esos errores?**

Al igual que con loaders usamos useLoaderData, tenemos useActionData

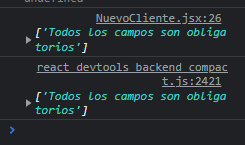
Primero se importa



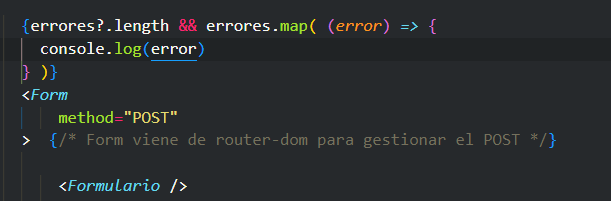
Y se instancia



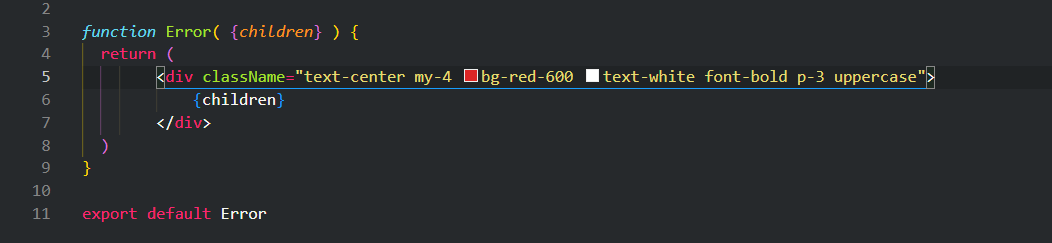
En el console.log al dar click



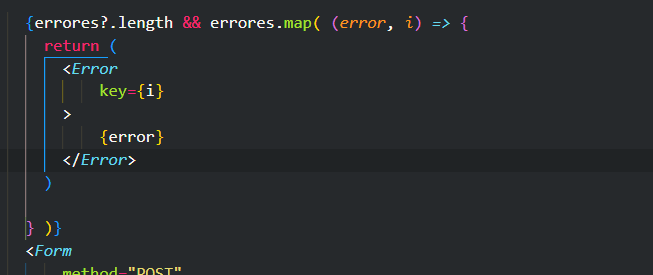
Entonces antes del <Form> podemos agregar el error, aquí se hace una prueba



Creamos un component para el Error.jsx



Y en el NuevoCliente.jsx quedaría así



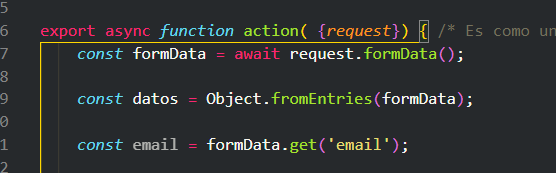
**169. Diferencias de useLoaderData y useActionData y más validaciones.**

¿Cuándo utilizar useLoaderData y useActionData?

**useLoaderData** cuando quieras obtener el resultado de un loader.

**useActionData** cuando quieras obtener el resultado de un action.

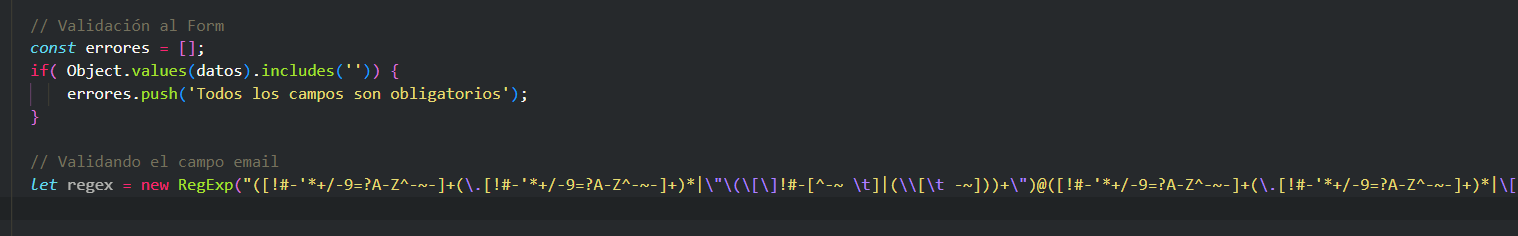
Vamos a validar un campo en específico(email)

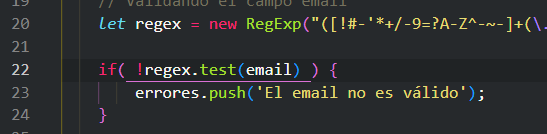




Primero obtenemos el valor de email

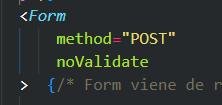
Y para validar vamos a usar una expresión regular

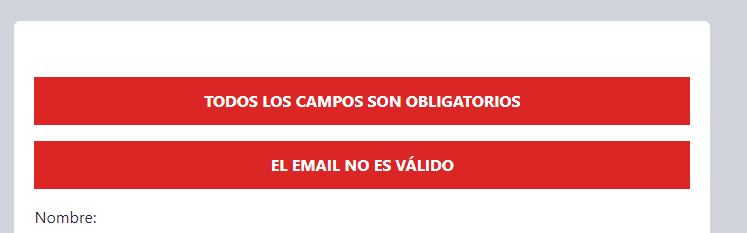




Evaluamos diciendo “si no se cumple”

Pero en el Form le añadimos un noValidate para que desactive las validaciones automáticas que hace el navegador





**170. Introducción a JSON-Server**

En una herramienta de nmp que crea una fake API así que la tenemos que instalar en una consola de powershell como administrador

npm i json-server

**¿Qué es una REST API o REST?**

REST = Representational State Transfer

Puede ser Diseñada en cualquier lenguaje

Debe responder a los Request HTTP: GET, POST, PUT, PATCH, DELETE

Tiene una forma ordenada y estructurada de poner a disposición los recursos

**Verbos HTTP**

**GET –** Obtener registros

**POST –** Enviar nuevos datos al servidor / Creación

**PUT / PATCH -** Actualizar

**DELETE –** Eliminar un registro.

Una REST API sigue cierto formato en sus rutas o url, se le conoce como endpoints.

**Endpoints de una REST API**

Una REST API cuenta con Endpoints (o Urls) para hacer operaciones CRUD

**C**reate **R**ead **U**pdate **D**elete

Listar todos los clientes GET /clientes

Obtener un solo cliente GET /cliente/id

Crear un nuevo cliente POST /clientes

Editar un cliente PUT /clientes/id

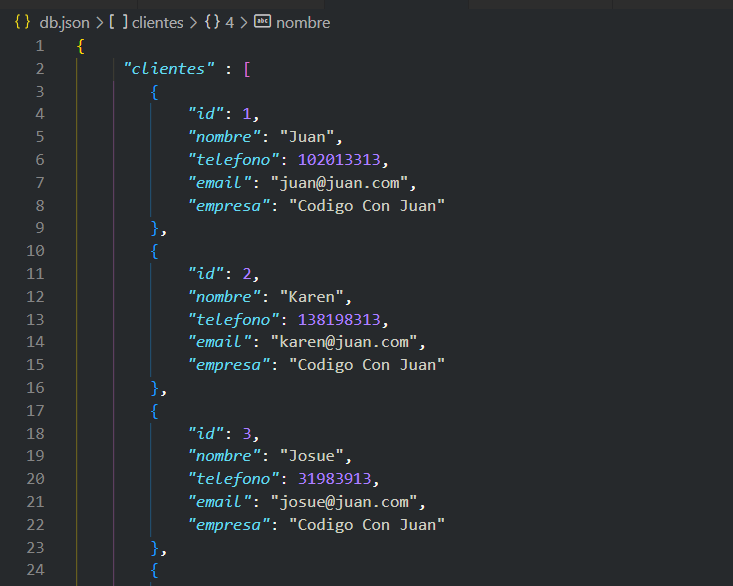
Borrar un cliente DELETE /clientes/id

**Vamos a crear una REST API con JSON Server**

En la raíz del proyecto creamos un archivo llamado db.json

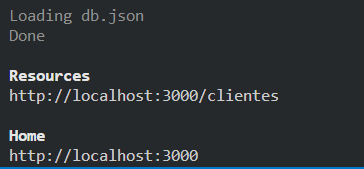
JSON = JavaScript Object Notation

Y ahí colocamos los clientes que teníamos en el index



Para tener disponible nuestro json con una url ejecutamos en consola

Json-server –watch db.json



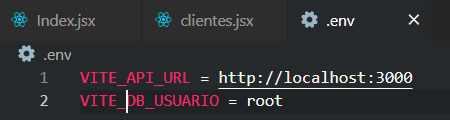
**171. Conectando JSON Server a nuestro proyecto**

Vamos a empezar por el index.jsx eliminando los clientes, dejando el loader pero para no tener todo el consumo de la API ahí creamos una carpeta data dentro del src y un archivo clientes.jsx

Usaremos una función con async y await, para consumir la URL, usaremos variables de entorno, son aque|las que en el entorno de desarrollo tiene un valor y en entorno de producción tiene un valor diferente.

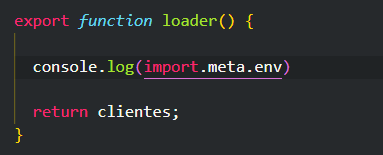
Cada framework tiene su forma de crear sus variables de entorno. En vite crearemos un archivo .env

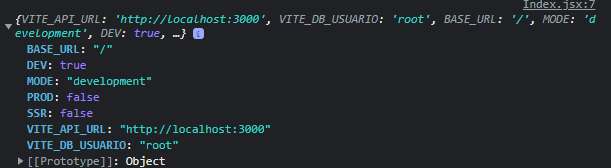
Siempre las variables de entorno en vite se deben nombrar iniciando por **VITE\_**



Como creamos un nuevo archivo lo recomendable es detener el servidor de vite y volverlo a ejecutar

Y en el loader de index hacemos console.log a esas variables

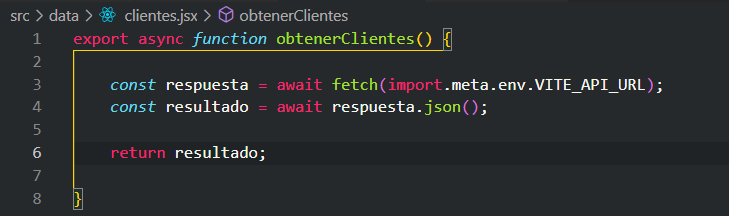




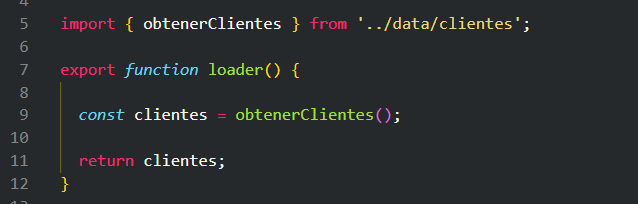
La mayoría las crea vite pero las nuestras son las últimas (VITE\_)

La segunda variable fue de ejemplo. No la requerimos

Así queda la consulta a la API en data>clientes.jsx



La importamos en Index.jsx y la llamamos desde el loader

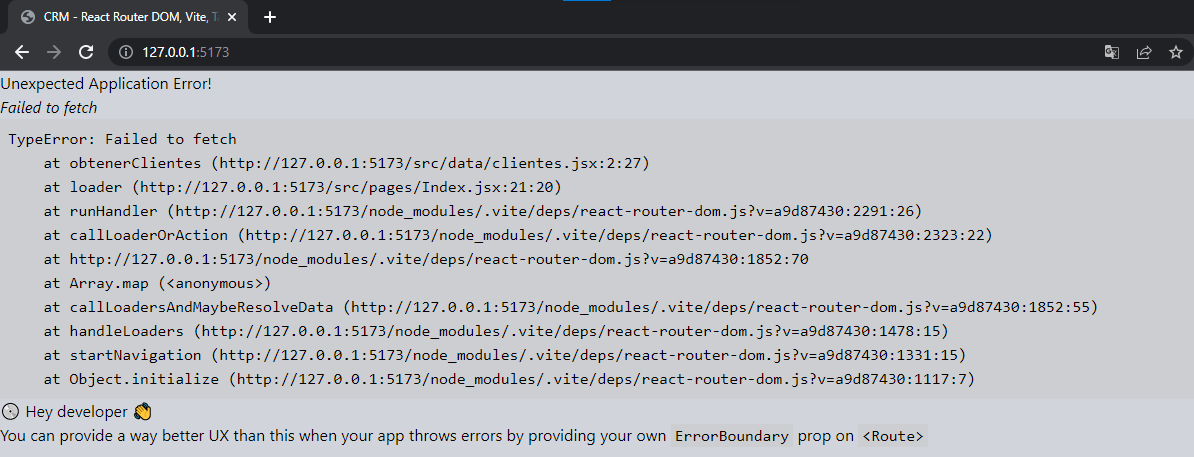




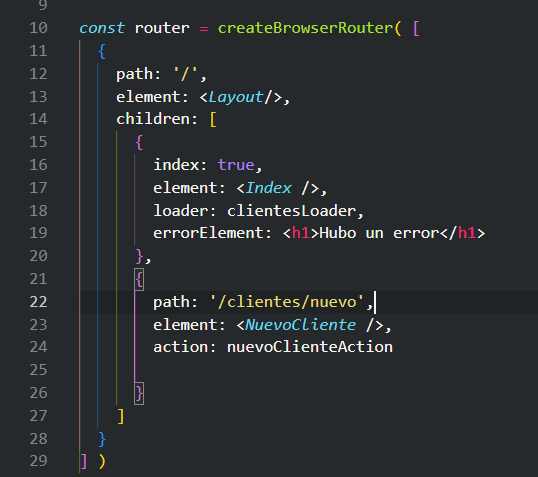
Con esa const que retornamos, va a quedar el programa como al inicio cargando los clientes, se recuperan los datos con useLoaderData y se listan gracias al map que aplica más abajo.

**172. Crear una pantalla de error.**

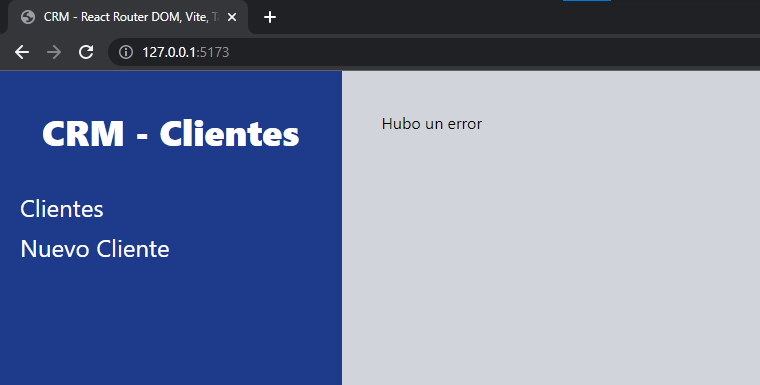
Cuando hay algún error la pantalla se ve así



Podemos mejorar esos errores con ErrorBoundary. En el index.jsx le pasamos un errorElement

Este es de prueba, se le puede pasar un componente como en el element, pero lo que hará es poner ese código html en la parte donde está pasando el error y ya no en toda la pantalla como antes

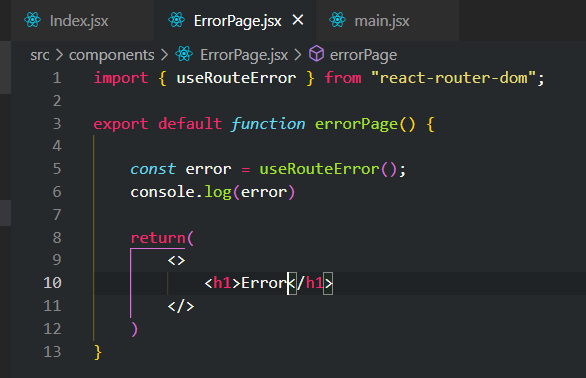




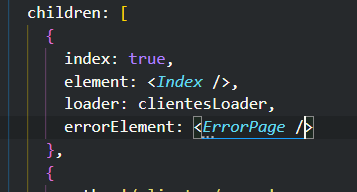


Pero vamos a hacerlo más inteligente y que se pueda hacer un debug para saber el error.

Creamos un componente que se llame ErrorPage.jsx, importamos el Hook useRouterError y lo “instanciamos”

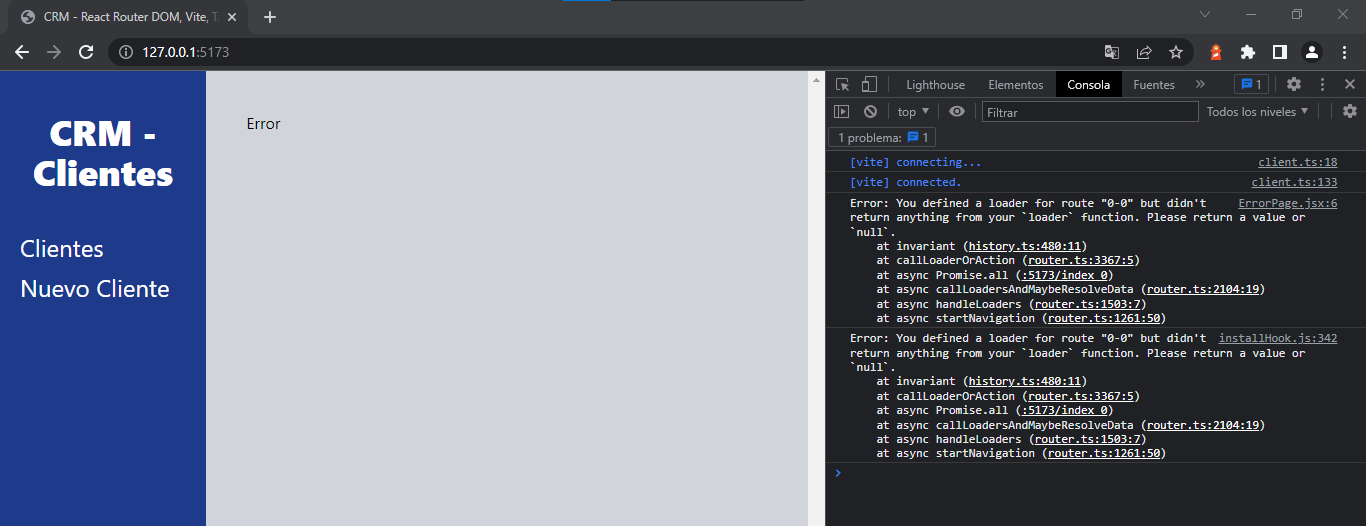


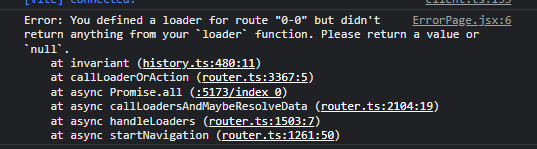
Y ese ErrorPage lo importamos en el main.jsx y se lo pasmos como errorElement en lugar del HTML que teníamos





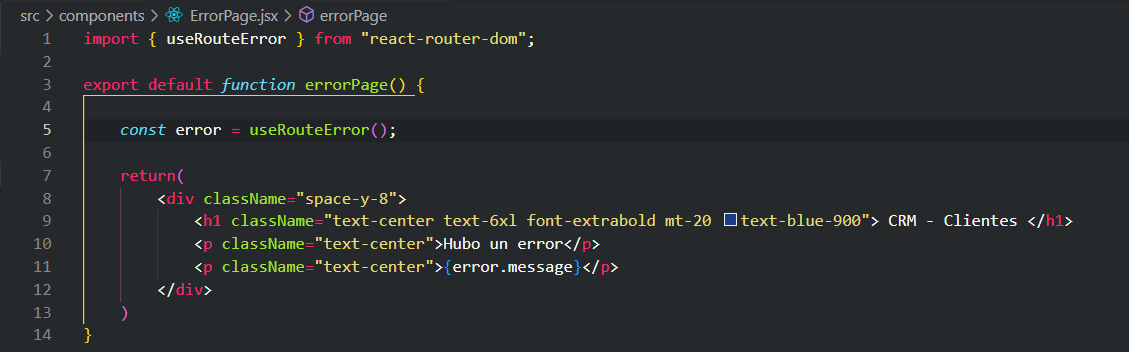
Y así podemos debuggear el error en consola

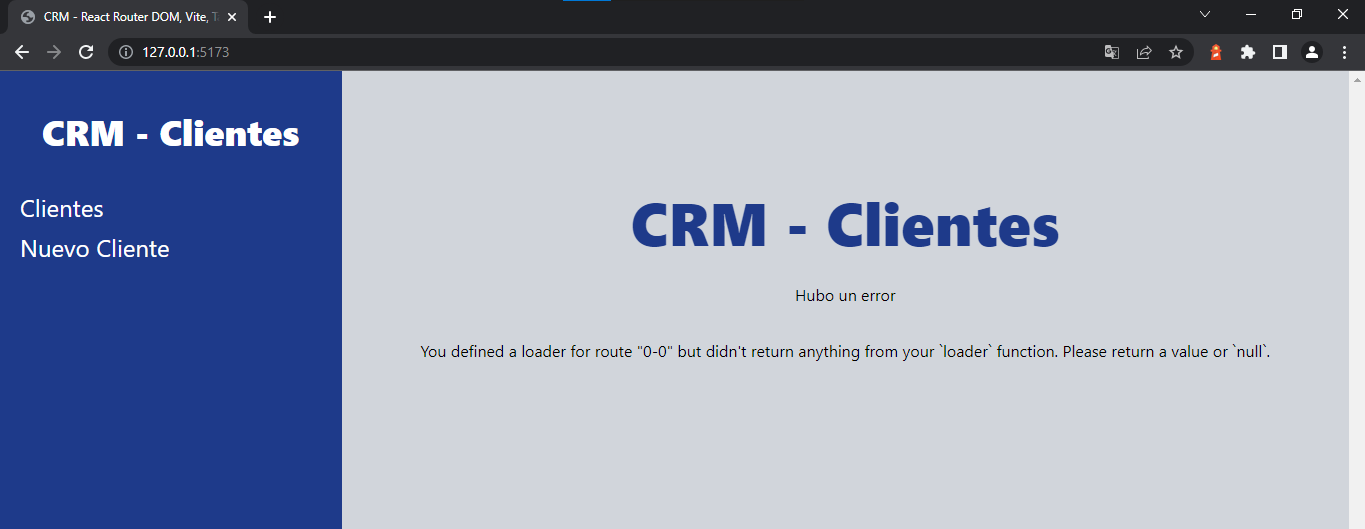




El error está en el Loader de index.js ya que por regla debe tener un return y en este caso no se lo pusimos para probar esos errores

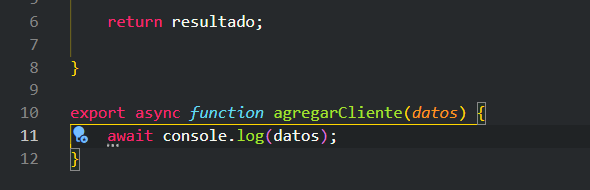
Podemos darle estilo a la página de errores para que se muestre de forma más amigable con el usuario



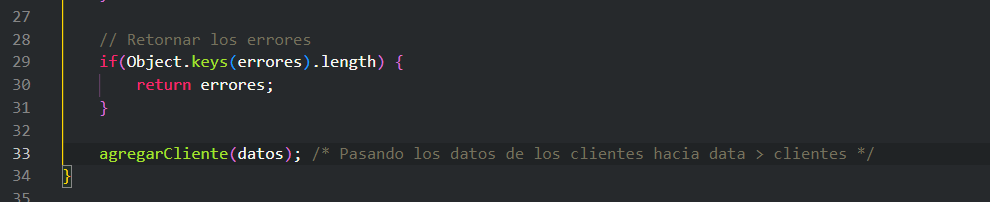


**173. Añadir nuevos clientes en nuestro formulario**

En el archivo clientes de la carpeta data creamos una nueva función



Y la importamos en NuevoCliente.jsx, para llamarla después de que haya pasado las validaciones y le enviamos como argumento los datos

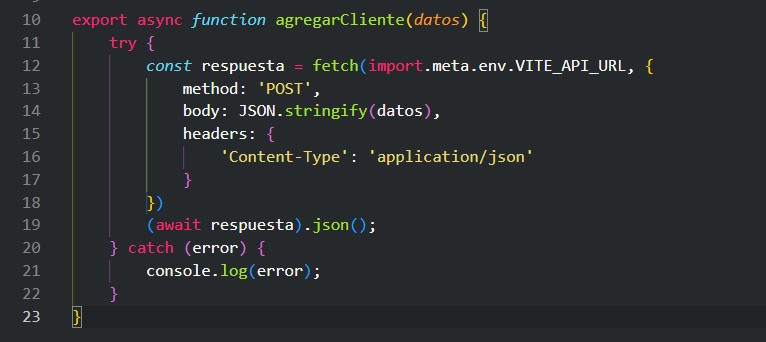




Y al llenar y enviar el formulario en consola tendremos el objeto con los datos y su origen viene desde clientes.js



En la función de agregarClientes vamos a trabajar con un try catch para en caso de error, tomarlo y debuggearlo



Aquí pasan varias cosas:

1. Tenemos que hacerle fetch de nuevo a la url porque vamos a meter los datos
2. En esta función el fetch tiene la finalidad de enviar los datos del form, por lo que debe llevar un objeto de configuración
3. El objeto de configuración es para indicar el método que se va a usar, por default es GET, por eso cuando consultamos una API para obtener datos no es necesario el objeto, pero en este caso el method va a ser POST

El objeto de configuración

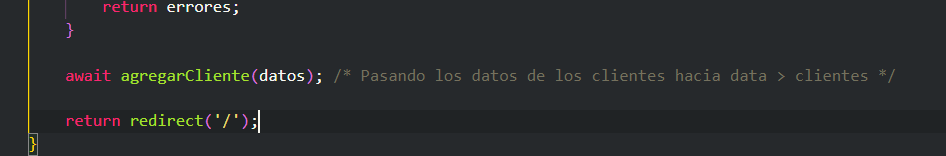
El objeto de opciones que se pasa como segundo argumento a fetch() especifica

* Que el método de solicitud **HTTP sea POST**
* El cuerpo de la solicitud sea un objeto JavaScript datos que se convierte en una cadena JSON utilizando el método JSON.stringify().
* También especifica que el tipo de contenido de la solicitud sea "application/json".

La función fetch() devuelve una promesa que resuelve en la respuesta de la solicitud HTTP. En este caso, la respuesta se almacena en la constante respuesta.

**Nota importante: Los action también tienen que retornar algo**

El return en nuevocliente.jsx va a ser un redirect, que es una característica de router-dom así que se tiene que importar



Hemos visto diferentes formas para dirigir o saltar entre páginas: **Link, Navigate, Redirect** pero se usan en acciones en específico

**Link o NavLink** Es para crear una barra de navegación en lugar de <a href=””>

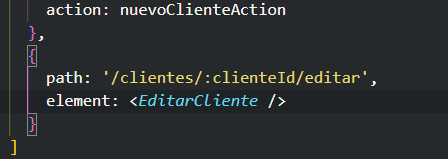
**Navigate** Es buena opción cuando quieres redireccionar por medio de un botón

**Redirect** Cuando vayas a redireccionar al usuario en actions y loaders

**174. Primeros pasos con editar cliente y useParams para recuperar el id**

Creamos un nuevo children en el main.jsx

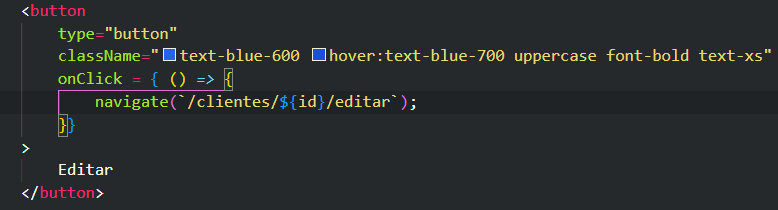
Cuando editamos un registro debemos tener routerdinamico porque tenemos varios clientes, cada uno tiene un id y ese id puede ser variable.

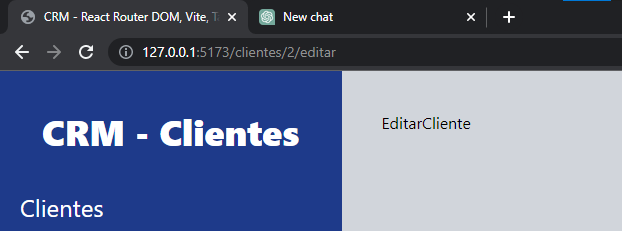


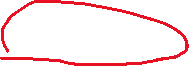
La sintáxis es con dos puntos **:** y ponemos el nombre de lo que se va a recuperar, se recomienda usar un nombre significativo acorde a lo que se está recuperando, porque react-router-dom va a crear una variable con ese nombre.

Creamos un archivo en pages llamado EditarCliente.jsx para usarlo como element del nuevo children

Para ser dirigidos a esa página dinámicamente usamos useNavigate en ListadoClientes.jsx con un onClick en Editar

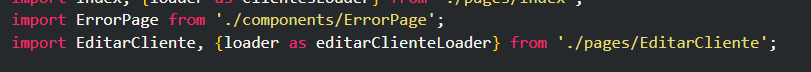
Donde dinámicamente se le está pasando el id a la url, eso en conjunto con el **:clienteId** nos dará una URL que contenga ese id



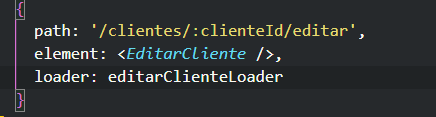


¿Cómo obtener el valor de la URL?

Para que esto suceda al momento de que carga la página creamos un loader, en la cual la función en automático recibe params, pero necesario ir al main.jsx e indicar que ese children tiene un loader









Y con eso vamos a tener ya en consola esos params

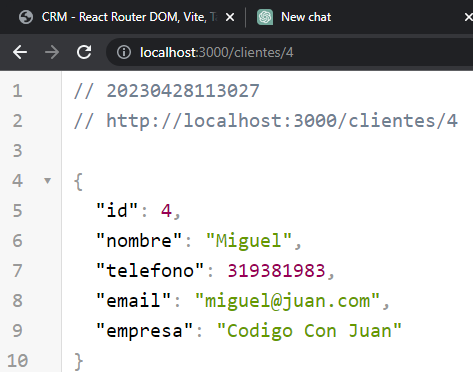


Y si nos damos cuenta la variable se llama clienteId tal cual la definimos en el path del children con **/:clienteId**

Como json-server se comporta como una API podemos ingresar a un cliente en específico por su id con la ruta

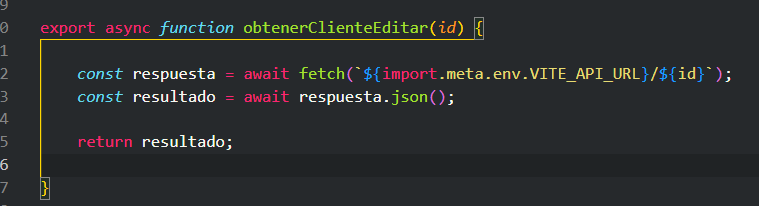
[http://localhost:3000/clientes/**IddeCliente**](http://localhost:3000/clientes/IddeCliente)

Ejemplo:

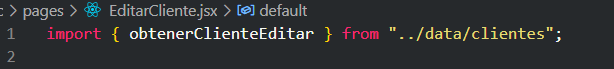


**175. Verificando que el cliente exista y mostrando error**

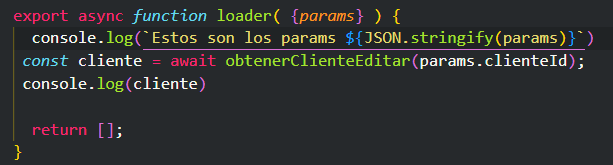
En data > clientes.js creamos una función para hacer fecth a la url pero esta vez inyectando en la url el id del cliente que estamos obteniendo, entonces la función toma como parámetro el id

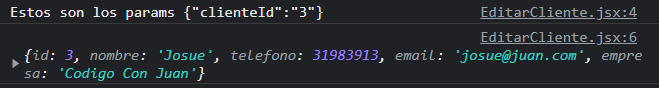


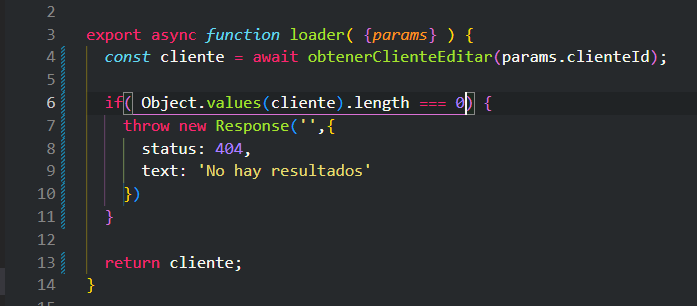
Y la importamos en EditarCliente.jsx,

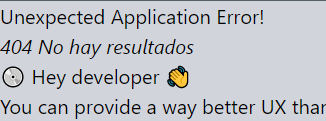


En editar cliente tenemos un loader que tiene como argumento params, ese params tiene ya el Id del cliente

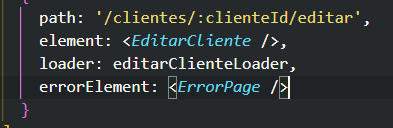




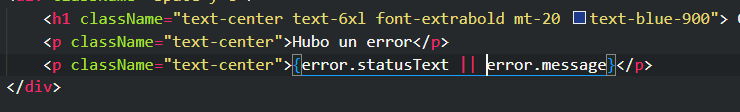
Validamos que no retorne un objeto vacío, y nos saldrá la pantalla de error

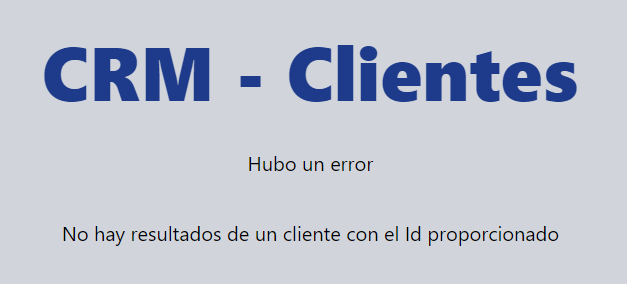


Pero se pueden poner mensajes de error personalizados, en el main.jsx le añadimos un errorElement



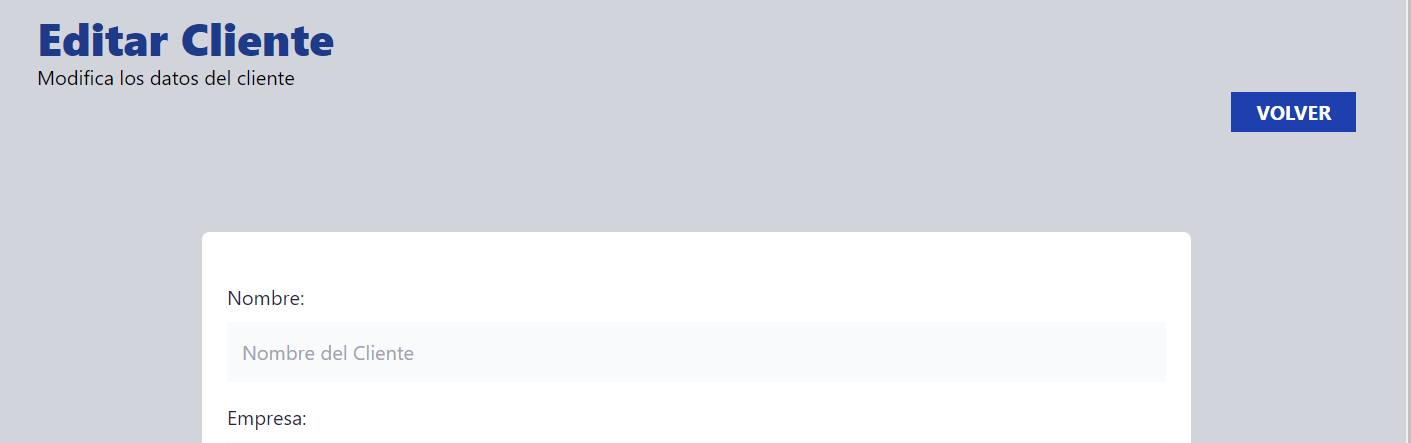
Pero errorPage leía un error.message, lo modificamos para que pueda leer ambos errores



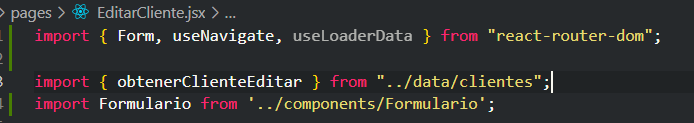


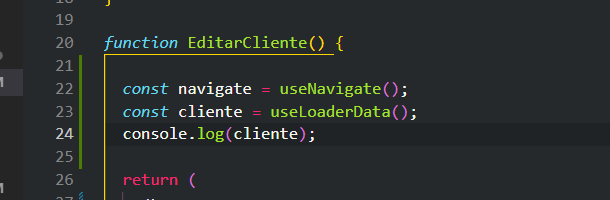
**176. Añadiendo el Formulario para Editar y Autocompletar el formulario.**

Empezamos copiando la parte de HTML de nuevoCliente y le hacemos los ajustes para que nos quede como página de edición

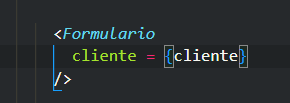


El loader de la página retorna el cliente, recordemos que para leer esos datos importamos useLoaderData

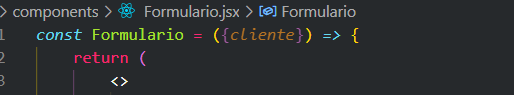
Y la instanciamos



Y le pasamos vía props al Formulario el cliente

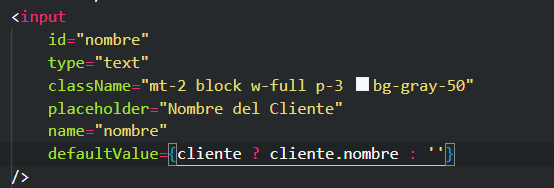


Y lo recuperamos en Formulario

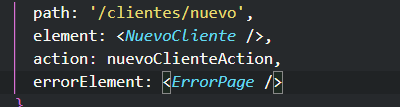


Para poner los datos en cada campo podemos usar un value=”” pero eso dará error ya que en cada campo se usa un onchange con un useState, pero aquí no hay como tal un state, por usar router-dom tenemos los action entonces usamos defaultValue= y ese no espera el onChange.

Pero eso nos dará un error en NuevoCliente porque querrá leer una propiedad que puede o no puede existir, entonces podemos solucionarlo con un chaining.



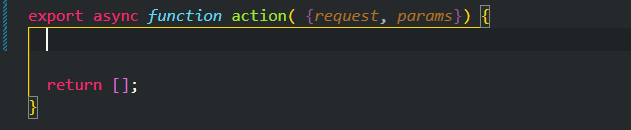
Como sale la pantalla de error por default de router-dom en donde tenemos el main.jsx editamos la parte de NuevoCliente



Para llenar todo el form hacemos lo mismo del optional chaining en cada campo. No hacemos destructuring en el Formulario porque puede que cliente exista o no, si es un nuevo cliente no existe.

**177. Creando la acción para editar un cliente.**

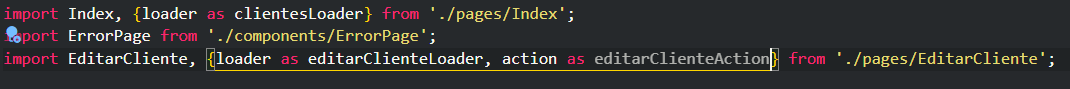
Necesitamos un action para mandar la información actualizada, creamos uno en EditarCliente

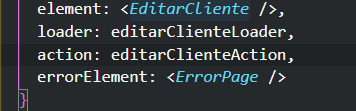


El request es para el formData y de los datos del objeto cliente, params es para el id del cliente, esos params vienen de más arriba, del loader

Copiamos lo del action de NuevoCliente y lo ajustamos a este caso

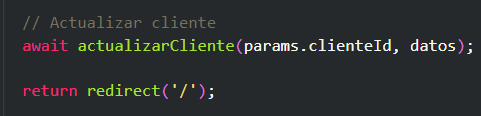
Y lo añadimos en el main.jsx el action





Y en vez de llamar a agregarCliente vamos a llamar la función que estará en data > cliente.js





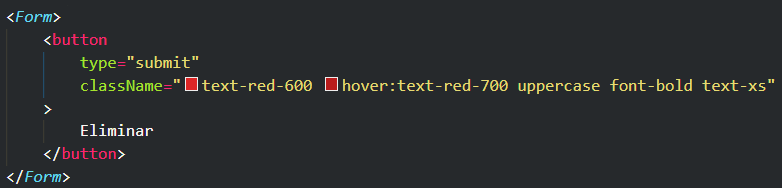
La función en cliente.js



La url a la que se le hace fetch debe llevar el ID, y en vez de GET se usa PUT porque es para actualizar.

**178. Eliminar un cliente**

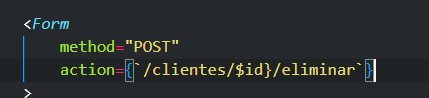
Tenemos que rodear el botón de eliminar con una etiqueta Form para poder asociarle una acción



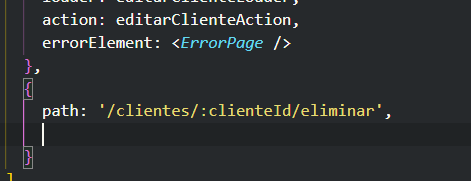
E importamos Form y redirect



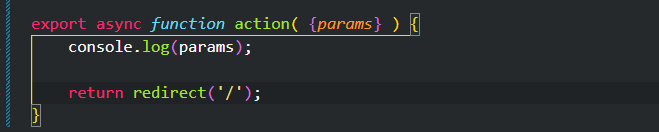
En la etiqueta Form añadimos



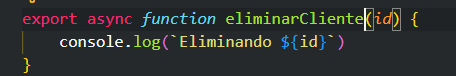
Creamos un nuevo children para la ruta



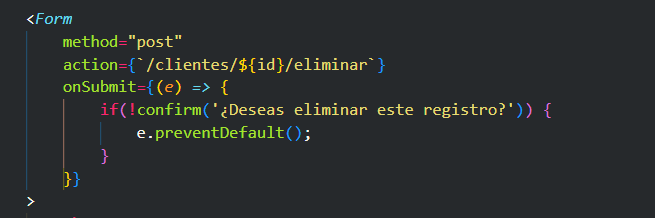
En ListaClientes.jsx creamos un action



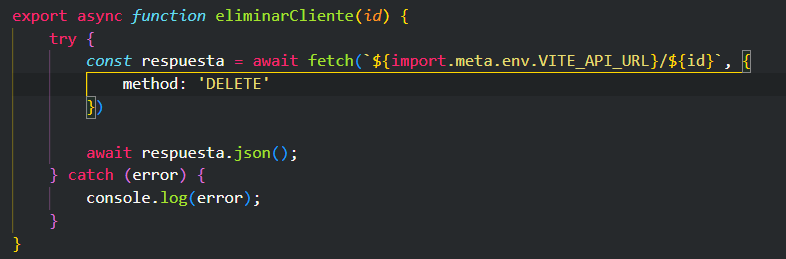
Vamos a crear la función en clientes.js para eliminar



Agregamos un onSubmit (que es de react) para validar que se quiera eliminar el registro



Y en la función de eliminar cliente solo tendrá el método delete sin body ni headers porque no estamos enviando nada



Y en el action colocamos await a la función que eliminará el cliente

