UNIVERSIDAD TÉCNICA DEL NORTE

FACULTAD DE INGENIERIA EN CIENCIAS APLICADAS

INGENIERIA DE SOFTWARE

TEMA: Documentación de uso y aplicación del Asset “Cliente”

1. Instalación de aplicación, dependencias, módulos para su uso

PERN STACK: Postgresql, Express, React y Node

1.1. Base de datos

- Postgresql

1.2. Backend Api Rest

- Node js versión
- Editor de texto “Visual Studio Code” (elección)
- Framework “Express”

1.3. Frontend

Extensiones:

- Reac Snippets Es7/React/Redux/GraphQL
- Material Icons
- Material Theme

2. Configuración del entorno Backend

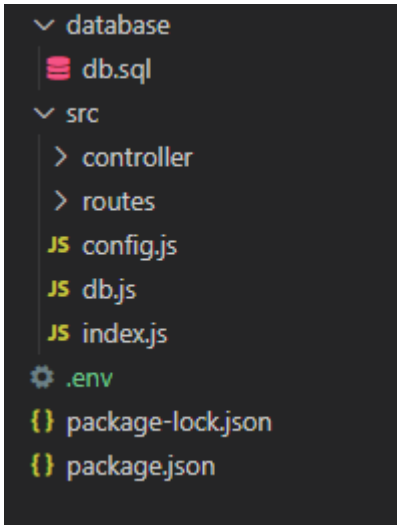
Servidor: npm i express

Vista por consola: npm i Morgan

Comunicar servidores en dominios distintos: npm i cors

Escuchar cambios: npm i nodemon -D

2.1. Arquitectura



2.2. Base de datos

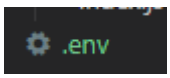
```
CREATE DATABASE clientedb

CREATE TABLE cliente(
    id serial
    ced_cliente int PRIMARY KEY,
    nombre_cliente varchar(255),
    telefono_cliente int,
    direccion_cliente varchar(255),
    email-cliente varchar(255)
);
```

Para conexión de la base de datos se ha creado variables de entorno que permite la protección de las credenciales de la base de datos.

Archivo de configuración:

```
const{config} = require('dotenv')
config()

module.exports = {
    db: {
        user: process.env.DB_USER,
        password: process.env.DB_PASSWORD,
        host: process.env.DB_HOST,
        port: process.env.DB_PORT,
        database : process.env.DB_DATABASE
    }
}
```

Variables de entorno se encuentra en este archivo que no ha sido importado por protección de datos.



2.3. Controlador

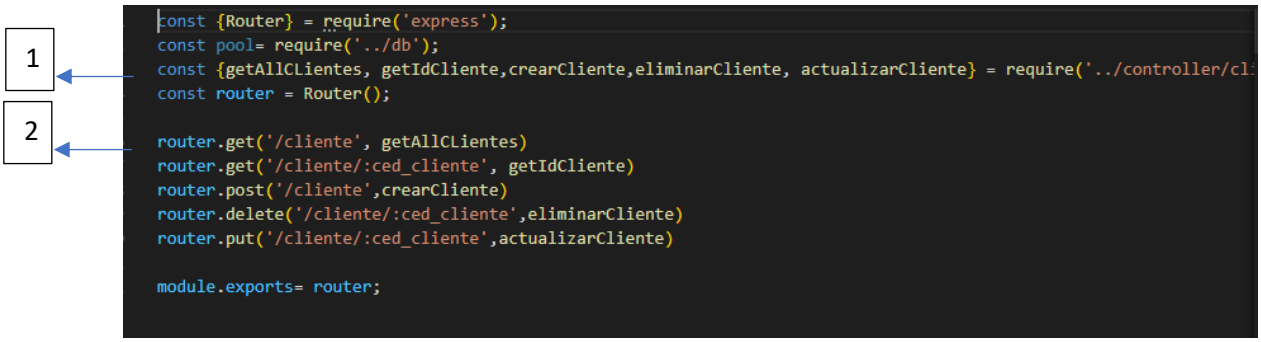
El controller permite crear las peticiones en este caso contiene 5 que permite controlar los métodos los cuales son : Get, Put, Delete y Update donde nos permite tener el CRUD de la aplicación.

1: Se crea el método con el nombre “getAllClientes” donde me va a permitir visualizar todos los clientes que se encuentren registrados en la base de datos.

2: La línea número 2 se encuentra la consulta de clientes en la base de datos.

2.4. Rutas

Importar función llamada “Router” lo cual nos devuelve un objeto donde nos va a permitir crear Url para poder navegar por las distintas peticiones.



1: Contiene el método que es llamado de la clase controller

2: Se crea la ruta con el método asignado

Nota: Si aun no se encuentra corriendo la aplicación Frontend , se puede visualizar todas las rutas mediante la aplicación “POSTMAN” para ver su correcto funcionamiento.

La aplicación contiene exactamente 5 rutas que funcionan de la siguiente manera:

`router.get('/cliente', getAllClientes)` : Sirve para visualizar todos los clientes existentes.

`router.get('/cliente/:ced_cliente', getIdCliente)`: Visualiza cliente mediante su ID

`router.post('/cliente',crearCliente)`: Crea el cliente

```
router.delete('/cliente/:ced_cliente',eliminarCliente)
router.put('/cliente/:ced_cliente',actualizarCliente)
```
: Elimina y actualiza un cliente mediante el número de identidad.

2.5. Funcionamiento de aplicación

Del funcionamiento se encarga la clase Index.js donde se encuentra instanciado las dependencias que se usa mas las rutas creadas.

```
const express = require('express');
const morgan = require('morgan');
const cors = require('cors');

const clientesRoutes = require('./routes/clientes.routes');

const app= express();

app.use(cors());
app.use(morgan('dev'));
app.use(express.json());

app.use(clientesRoutes)
```

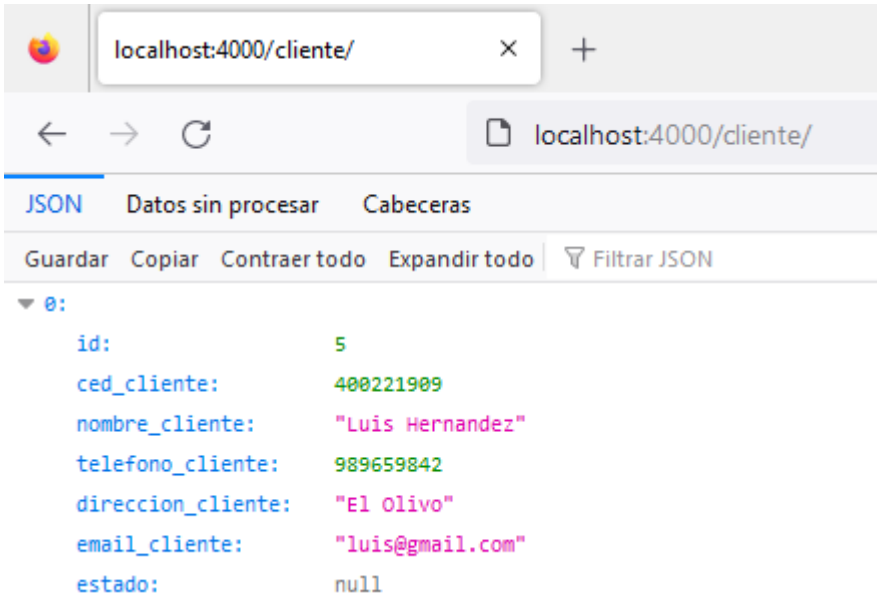
Todo esto va a correr en el puerto: 4000

```
app.listen(4000)
console.log('Server on port 4000')
```

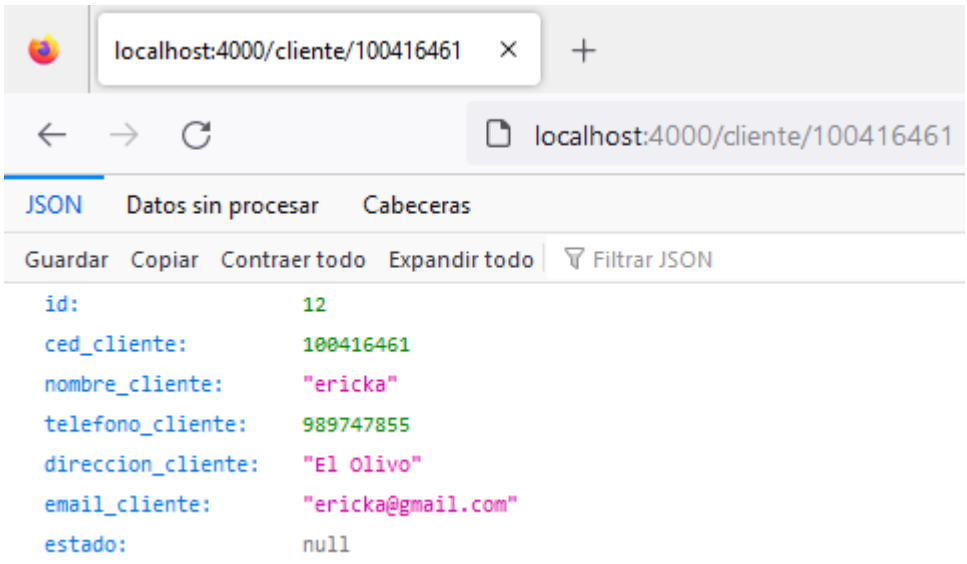
Comando: npm run dev

```
PS D:\Fabrica_Proyect\CopiaFuncional\Cliente> npm run dev

> cliente@1.0.0 dev
> nodemon src/index.js

[nodemon] 2.0.20
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node src/index.js`
Server on port 4000
```

Visualizando en la controla permite verificar su correcto funcionamiento, abriendo el navegados con la ruta http://localhost:4000/cliente/



Mediante su identidad:



3. Configuración del entorno Frontend

Configuración para las rutas: npm i react-router-dom

Material UI

3.1. Navegación

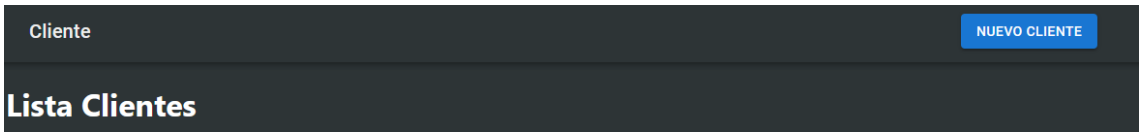
En Apps.js se ingresa las rutas por las cuales vamos a navegar dentro de la aplicación

```jsx
import { BrowserRouter, Route, Routes } from 'react-router-dom';
import ListaClientes from './components/listaClientes';
import FormClientes from './components/formClientes';
import Menu from './components/Navbar';
import { Container } from '@mui/material'

export default function App() {
  return (
    <BrowserRouter>
    <Menu/>
      <Container>
        <Routes>
          <Route path="/" element={<ListaClientes />} />
          <Route path="/cliente/nuevo" element={<FormClientes />} />
          <Route path="/cliente/:ced_cliente/edit" element={<FormClientes/>}/>
        </Routes>
      </Container>
    </BrowserRouter>
```

Tenemos un Navbar donde vamos a ingresar Titulo y a su vez enlaces para diferentes opciones.

```jsx
import { AppBar, Toolbar, Typography, Container, Button, Box } from "@mui/material"
import {Link, useNavigate} from "react-router-dom";

export default function Navbar() {

  const navigate = useNavigate()

  return (
    <Box>
      <AppBar position='static' color="transparent">
        <Container>
          <Toolbar>
            <Typography variant='h6' sx={{flexGrow:1}}>
              <Link to="/" style={{textDecoration: "none", color:"#eee"}}>Cliente</Link>
            </Typography >
            <Button variant='contained' color='primary' onClick={() => navigate('/cliente/nuevo')}>
              Nuevo Cliente
```

Asi se ve en la vista:



Lista Clientes

3.2. Lista de Clientes y como llamar a la ruta del Api Rest

Este es el método que va a permitir la visualización de los datos que se encuentran en la Api

```jsx
const loadClientes = async () => {
  const response = await fetch('http://localhost:4000/cliente')
  const data = await response.json()
  setClientes(data)
  setTablaCliente(data)
}
```

El nombre es a elección y en la variable response se encuentra la ruta con la que habíamos navegado en la Api.

En la tabla se encuentran todos los datos para su vista

```
{clientes.map((cliente) => (
  <StyledTableRow key={cliente.ced_cliente}>

    <StyledTableCell align="right">{cliente.ced_cliente}</StyledTableCell>
    <StyledTableCell align="right">{cliente.nombre_cliente}</StyledTableCell>
    <StyledTableCell align="right">{cliente.telefono_cliente}</StyledTableCell>
    <StyledTableCell align="right">{cliente.direccion_cliente}</StyledTableCell>
    <StyledTableCell align="right">{cliente.email_cliente}</StyledTableCell>
    <StyledTableCell align="right">

      <Button variant='contained' color='inherit' onClick={() => navigate(`/cliente/${cliente.ced_cl
        Actualizar
      </Button>

      <Button variant='contained' color='warning' onClick={() => handleDelete(cliente.ced_cliente)}
```

3.3. Creación de clientes

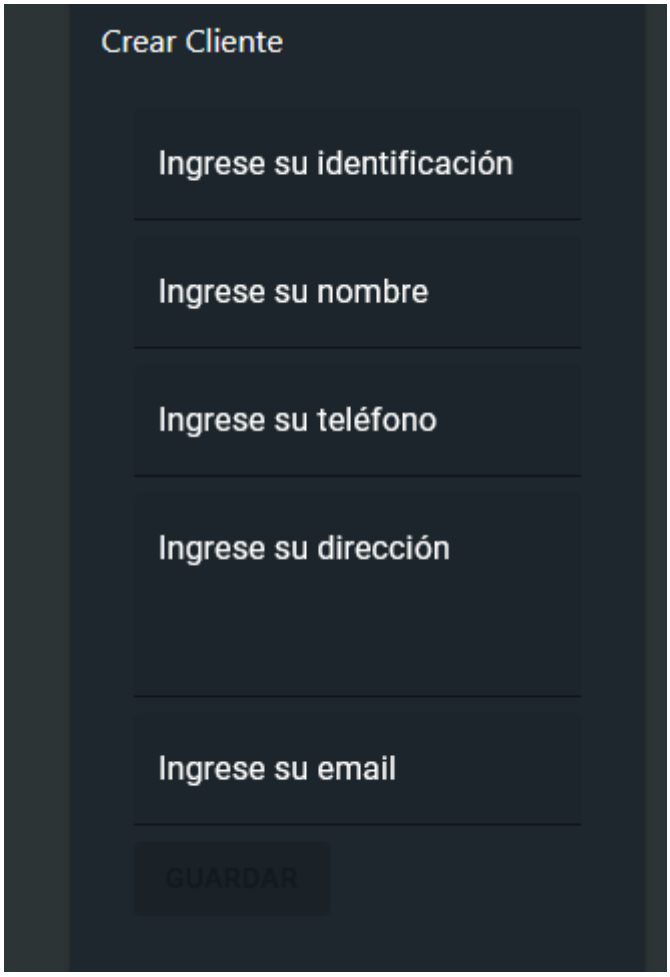
Estos son los datos que se va a recibir de cliente

```
const [cliente, setCliente] = useState({
  ced_cliente: "",
  nombre_cliente: "",
  direccion_cliente: "",
  telefono_cliente: "",
  email_cliente: "",
});
```

Formulario donde se van a ingresar los datos para la creación

```
<Card sx={{ mt: 5 }} style={{ backgroundColor: '#1e272e', padding: '1rem'
  <Typography variant='5' textAlign='center' color='white'>
    {editing? "Editar Cliente" : "Crear Cliente"}
  </Typography>
  <CardContent>
    <form onSubmit={handleSubmit}>
      <TextField
        variant='filled'
        label='Ingrese su identificación'
        sx={{ display: 'block', margin: '.5rem 0' }}
        name="ced_cliente"
        value={cliente.ced_cliente}
        onChange={handleChange}
        inputProps={{ style: { color: "white" } }}
        InputLabelProps={{ style: { color: "white" } }}
      />
```

Pantalla :



El botón se encuentra deshabilitado y al momento de llenar todos los datos automáticamente se habilita.

CRUD :

DELETE :

```
const handleDelete = async (ced_cliente) => {
  await fetch(`http://localhost:4000/cliente/${ced_cliente}`, {
    method: "DELETE",
  })
  setClientes(clientes.filter((cliente) => cliente.ced_cliente !== ced_cliente));
}
```

Actualizar o ingresar: Este método se encarga de verificar si esta llamando un dato con una ID se denominaría actualizar y nos llevaría al mismo formulario que comparte estos dos métodos.

Caso contrario si no obtiene ninguna ID, se creara un nuevo usuario.

```
if (editing) {
await fetch(`http://localhost:4000/cliente/${params.ced_cliente}`, {
    method: "PUT",
    headers: {
      "Content-Type": "application/json",
    },
    body: JSON.stringify(cliente),
  });
} else {
  await fetch("http://localhost:4000/cliente", {
    method: "POST",
    body: JSON.stringify(cliente),
    headers: { "Content-Type": "application/json" },
```

3.4. Pantalla final como debería ser su correcto funcionamiento.