**Manual: Creación de una Aplicación React con Vite para un Sistema de RRHH**

**Introducción**

Este manual te guiará paso a paso en la creación de una aplicación de frontend utilizando **React y Vite**. 🚀 Vite es una herramienta de compilación moderna que mejora significativamente la velocidad y la experiencia de desarrollo. La aplicación servirá como la interfaz de usuario para un sistema de Recursos Humanos, diseñada para interactuar con un backend existente.

**Prerrequisitos**

Antes de comenzar, asegúrate de tener instalado lo siguiente:

* **Node.js y npm:** Esencial para ejecutar Vite y gestionar los paquetes del proyecto.
* **Visual Studio Code:** Un editor de código para gestionar los archivos y usar la terminal integrada.

**Paso 1: Preparación del Entorno**

Primero, organiza tu espacio de trabajo creando una carpeta dedicada para la aplicación.

1. **Navega a tu directorio de trabajo** (p. ej., Cursos/Java).
2. **Crea la carpeta del proyecto** con el nombre: Recursos-humanos-app.

**Paso 2: Abrir el Proyecto en Visual Studio Code**

Abre la carpeta que acabas de crear como tu espacio de trabajo en VS Code.

1. Abre **Visual Studio Code**.
2. Ve a **Archivo > Abrir carpeta...** (File > Open Folder...).
3. Busca y selecciona la carpeta Recursos-humanos-app y haz clic en **"Seleccionar carpeta"**.

**Paso 3: Creación del Proyecto con Vite**

Ahora, en lugar de create-react-app, usaremos el comando de Vite para generar la estructura del proyecto. El proceso es interactivo y muy rápido.

1. **Abre la terminal integrada** en VS Code (**Terminal > Nueva terminal**).
2. **Ejecuta el comando de creación de Vite:** En la terminal, escribe el siguiente comando y presiona Enter. El punto . al final indica que el proyecto se debe crear en la carpeta actual.

Bash

npm create vite@latest .

1. **Sigue las instrucciones interactivas:** Vite te hará algunas preguntas para configurar el proyecto. Responde de la siguiente manera:
   * Need to install the following packages: create-vite@latest Ok to proceed? (y) -> Presiona y y Enter.
   * Select a framework: -> Usa las flechas para seleccionar **React** y presiona Enter.
   * Select a variant: -> Selecciona **JavaScript** y presiona Enter.

✨ ¡Listo! En segundos, Vite habrá creado la estructura base de tu proyecto.

**Paso 4: Instalar las Dependencias**

A diferencia de create-react-app, Vite crea los archivos del proyecto pero no instala las dependencias (los paquetes de npm) automáticamente. Debes hacerlo con un comando separado.

1. En la misma terminal, ejecuta:

Bash

npm install

Este comando leerá el archivo package.json y descargará todas las librerías necesarias (como React) en la carpeta node\_modules.

**Paso 5: Iniciar la Aplicación Vite**

El comando para iniciar el servidor de desarrollo también es diferente.

1. Para iniciar la aplicación, ejecuta:

Bash

npm run dev

1. **Observa el resultado:**
   * La terminal te mostrará que el servidor está corriendo y te dará una URL local.
   * Automáticamente (o si haces Ctrl+clic en el enlace) se abrirá tu navegador.
   * La aplicación se ejecutará, por defecto, en la dirección **http://localhost:5173**.
   * Verás una página de inicio simple de React + Vite.

**Paso 6: Verificación y Conexión con el Backend**

Has creado y ejecutado con éxito tu aplicación. Ahora, es crucial ajustar la configuración del backend para que se comunique correctamente con tu nueva aplicación de Vite.

* **Puerto de Ejecución:** Tu aplicación Vite se ejecuta en el **puerto 5173**, no en el 3000.
* **Ajuste de CORS:** Debes asegurarte de que tu backend (hecho en Spring) permita peticiones desde este nuevo origen. La anotación en tus controladores de Spring debe ser actualizada:

Java

// Antes (con create-react-app)

// @CrossOrigin(origins = "http://localhost:3000")

// Ahora (con Vite)

@CrossOrigin(origins = "http://localhost:5173")

Sin este cambio, el backend rechazará las llamadas desde tu frontend.

**Resumen y Próximos Pasos**

¡Felicidades! Has configurado un proyecto de React moderno y eficiente con Vite.

* **Logros:**
  1. Creaste un proyecto React usando **Vite**.
  2. Instalaste las dependencias.
  3. Lanzaste el servidor de desarrollo con npm run dev.
  4. Identificaste el cambio de puerto (5173) y el ajuste de **CORS** necesario.

El siguiente paso es comenzar a construir tu interfaz, para lo cual puedes empezar a editar el archivo **src/App.jsx**.

**Manual (Parte 2): Limpieza del Proyecto y Adición de Bootstrap**

**Introducción**

En esta segunda parte, vamos a preparar nuestro proyecto para el desarrollo real. Los objetivos son: optimizar nuestro entorno de trabajo con una extensión de VS Code, limpiar los archivos y el código de ejemplo que Vite genera por defecto, e integrar **Bootstrap** para estilizar nuestra aplicación de manera rápida y profesional.

**Paso 1: Instalar Extensión de VS Code para React (Opcional pero recomendado)**

Para agilizar la creación de componentes y código repetitivo en React, instalaremos una extensión de "snippets" (fragmentos de código).

1. En la barra lateral de VS Code, haz clic en el icono de **Extensiones** (cuatro cuadrados, uno separado).
2. En la barra de búsqueda, escribe ES7+ React/Redux/React-Native snippets.
3. Selecciona la extensión (generalmente es la primera, del autor dsznajder) y haz clic en **Instalar**.

Esta herramienta te permitirá generar estructuras de componentes, hooks y más con solo escribir atajos como rfce (React Functional Component Export).

**Paso 2: Limpieza de Archivos del Proyecto Vite**

Un proyecto recién creado con Vite incluye archivos de ejemplo que no necesitaremos. Vamos a eliminarlos para empezar con un lienzo limpio.

1. Abre el explorador de archivos en VS Code (el primer icono en la barra lateral).
2. Elimina los siguientes archivos:
   * Dentro de la carpeta public:
     + vite.svg
   * Dentro de la carpeta src/assets:
     + react.svg (puedes eliminar la carpeta assets completa si está vacía después de esto).
   * Dentro de la carpeta src:
     + App.css
     + index.css

Tu estructura de archivos en src ahora debería ser mucho más simple, conteniendo principalmente App.jsx y main.jsx.

**Paso 3: Limpieza del Código y Referencias Quebradas**

Al eliminar los archivos, nuestra aplicación se romperá porque el código todavía intenta importarlos. Ahora solucionaremos eso.

1. **Abre el archivo src/main.jsx:**
   * Elimina la siguiente línea, ya que el archivo index.css ya no existe:

JavaScript

import './index.css' // <--- ELIMINAR ESTA LÍNEA

1. **Abre el archivo src/App.jsx:**
   * Este archivo contiene varias referencias a los archivos que borramos y al estado de ejemplo. Elimina los imports que ya no se usan:

JavaScript

// ELIMINAR ESTAS LÍNEAS

import { useState } from 'react'

import reactLogo from './assets/react.svg'

import viteLogo from '/vite.svg'

import './App.css'

Después de estos cambios, la aplicación mostrará errores, lo cual es normal. Lo solucionaremos en el siguiente paso al reescribir el componente.

**Paso 4: Personalizar el Componente Principal (App.jsx)**

Vamos a reemplazar todo el contenido de App.jsx con una estructura básica para nuestro sistema.

1. **Borra todo el contenido** actual del archivo App.jsx.
2. **Pega el siguiente código:**

JavaScript

function App() {

return (

<div>

<h3>Sistema de Recursos Humanos</h3>

</div>

)

}

export default App

**Nota sobre .jsx:** La extensión .jsx le indica a las herramientas de desarrollo que este archivo contiene sintaxis de "JSX", que es la que nos permite mezclar HTML directamente dentro de JavaScript. Es la convención estándar en proyectos de React con Vite.

1. **Reinicia el servidor (si es necesario):** Si la terminal muestra muchos errores, a veces es más fácil detener el servidor de desarrollo (con Ctrl + C, y luego S o Y para confirmar) y volver a iniciarlo:

Bash

npm run dev

Ahora, si visitas http://localhost:5173, deberías ver únicamente el título "Sistema de Recursos Humanos" sin ningún estilo.

**Paso 5: Integrar Bootstrap mediante CDN**

Agregar Bootstrap nos dará acceso a un sistema de diseño profesional (rejillas, botones, tipografía, etc.) sin tener que escribir CSS desde cero. La forma más sencilla es usar su CDN (Content Delivery Network).

1. **Abre el archivo index.html** que se encuentra en la **raíz de tu proyecto**.
2. **Copia el link del CSS de Bootstrap:** Justo antes del cierre de la etiqueta </head>, pega la siguiente línea:

HTML

<head>

<meta charset="UTF-8" />

<link rel="icon" type="image/svg+xml" href="/vite.svg" /> <meta name="viewport" content="width=device-width, initial-scale=1.0" />

<title>Sistema RRHH</title>

<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-QWTKZyjpPEjISv5WaRU9OFeRpok6YctnYmDr5pNlyT2bRjXh0JMhjY6hW+ALEwIH" crossorigin="anonymous">

</head>

1. **Copia el script de JavaScript de Bootstrap:** Algunos componentes de Bootstrap (como menús desplegables o modales) requieren JavaScript. Pega el siguiente script justo antes del cierre de la etiqueta </body>:

HTML

<body>

<div id="root"></div>

<script type="module" src="/src/main.jsx"></script>

<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js" integrity="sha384-YvpcrYf0tY3lHB60NNkmXc5s9fDVZLESaAA55NDzOxhy9GkcIdslK1eN7N6jIeHz" crossorigin="anonymous"></script>

</body>

**Consejo de Productividad: Auto-Guardado** Para que los cambios se reflejen al instante sin tener que guardar manualmente, activa el auto-guardado en VS Code: **Archivo > Auto Guardar** (File > Auto Save). Asegúrate de que tenga una marca de verificación.

**Paso 6: Verificar la Instalación de Bootstrap**

Para confirmar que Bootstrap se ha cargado correctamente, aplicaremos algunas de sus clases de utilidad a nuestro componente.

1. Regresa al archivo src/App.jsx.
2. Modifica el div para que incluya las clases container y text-center:

JavaScript

function App() {

return (

<div className="container text-center"> // 'class' en HTML es 'className' en JSX

<h3 className="mt-3">Sistema de Recursos Humanos</h3>

</div>

)

}

export default App

1. Guarda el archivo (o deja que se auto-guarde).

Ahora, al ver tu aplicación en el navegador, notarás dos cambios:

1. La tipografía ha cambiado a la fuente por defecto de Bootstrap.
2. El título "Sistema de Recursos Humanos" está centrado en la página y tiene un pequeño margen superior (mt-3).

**Resumen y Próximos Pasos**

¡Excelente trabajo! Has limpiado y configurado tu proyecto, y ahora tienes el poder de Bootstrap a tu disposición.

* **Logros:**
  1. Eliminaste archivos innecesarios para una base de código limpia.
  2. Corregiste las referencias rotas y simplificaste el componente principal.
  3. Integraste exitosamente Bootstrap (CSS y JS) vía CDN.
  4. Verificaste que los estilos se aplican correctamente.

Ahora estás listo para comenzar a construir los componentes de tu aplicación, como la lista de empleados y los formularios de navegación.

**Manual (Parte 3): Creación del Componente ListadoEmpleados y Estilización con Bootstrap**

En esta tercera parte del manual, te guiaré en la creación de tu primer componente funcional en React, ListadoEmpleados, y cómo integrarlo en tu aplicación principal. Además, aprenderás a aplicar estilos de Bootstrap de manera efectiva dentro de un entorno React.

**Paso 1: Creación del Componente ListadoEmpleados**

Comenzaremos organizando nuestros componentes en una estructura de carpetas lógica y luego crearemos el archivo JavaScript para nuestro nuevo componente.

1. **Crea una nueva carpeta:**
   * Dentro de la carpeta src, haz clic derecho y selecciona **"New Folder"**.
   * Nombra la carpeta Empleados y presiona Enter.
2. **Crea el archivo del componente:**
   * Dentro de la carpeta Empleados, haz clic derecho y selecciona **"New File"**.
   * Nombra el archivo ListadoEmpleados.js y presiona Enter.
3. **Genera la estructura del componente:**
   * Abre ListadoEmpleados.js.
   * Si tienes instalada la extensión "ES7+ React/Redux/React-Native snippets" (Paso 1 de la Parte 2 del manual), escribe rfc y selecciona la opción rfce (React Functional Component Export). Presiona Enter.
   * Esto generará automáticamente la estructura básica de un componente funcional de React:

JavaScript

import React from 'react'

function ListadoEmpleados() {

return (

<div>ListadoEmpleados</div>

)

}

export default ListadoEmpleados

**Paso 2: Integración del Componente en App.jsx**

Ahora, vamos a reemplazar el contenido del componente principal (App.jsx) para que muestre nuestro nuevo ListadoEmpleados.

1. **Abre src/App.jsx.**
2. **Importa el componente ListadoEmpleados:**
   * Agrega la siguiente línea al principio del archivo:

JavaScript

import ListadoEmpleados from './Empleados/ListadoEmpleados';

1. **Reemplaza el contenido del div:**
   * Modifica la función App para que renderice el componente ListadoEmpleados en lugar del título directo:

JavaScript

import ListadoEmpleados from './Empleados/ListadoEmpleados';

function App() {

return (

<ListadoEmpleados></ListadoEmpleados>

);

}

export default App;

* + También puedes usar la sintaxis abreviada para componentes que no tienen hijos: <ListadoEmpleados />.

1. **Verifica en el navegador:**
   * Guarda los cambios. Deberías ver que la página ahora muestra el texto "ListadoEmpleados", lo que confirma que tu nuevo componente se está renderizando correctamente.

**Paso 3: Aplicando Estilos de Bootstrap al Componente ListadoEmpleados**

Dentro de React, para aplicar clases de CSS de Bootstrap, utilizamos el atributo className en lugar de class, ya que class es una palabra reservada en JavaScript. También veremos cómo aplicar estilos en línea.

1. **Abre src/Empleados/ListadoEmpleados.js.**
2. **Agrega un contenedor principal y el título:**
   * Modifica el div principal para que envuelva el contenido y tenga las clases container y text-center.
   * Agrega un título h3 con la clase mt-3 (margin-top de Bootstrap) y un estilo en línea para el margen superior.

JavaScript

import React from 'react'

function ListadoEmpleados() {

return (

<div className='container text-center'> {/\* Se utiliza 'className' en lugar de 'class' \*/}

<div className="container">

{/\* Estilo en línea: se usa doble llave {{ }} y propiedades en camelCase \*/}

<h3 className="mt-3" style={{ margin: "30px" }}>Sistema de Recursos Humanos</h3>

</div>

</div>

)

}

export default ListadoEmpleados

* + **Nota sobre estilos en línea:** En React, los estilos en línea se definen como objetos JavaScript. Las propiedades CSS con guiones (como margin-top) se convierten a camelCase (como marginTop), y los valores de las propiedades se escriben como cadenas.

**Paso 4: Integración de una Tabla de Bootstrap**

Ahora, incorporaremos una tabla básica de Bootstrap para mostrar la lista de empleados.

1. **Obtén el código de la tabla de Bootstrap:**
   * Ve a la documentación oficial de Bootstrap (puedes buscar "Bootstrap tables" en Google).
   * Busca la primera tabla de ejemplo que aparece. Copia el código HTML de esa tabla.
2. **Pega y adapta el código en ListadoEmpleados.js:**
   * Pega el código de la tabla debajo del div del título.
   * **Importante:** Reemplaza todas las instancias de class por className en el código de la tabla.

JavaScript

import React from 'react'

function ListadoEmpleados() {

return (

<div className='container text-center'>

<div className="container">

<h3 className="mt-3" style={{ margin: "30px" }}>Sistema de Recursos Humanos</h3>

</div>

<div className="container"> {/\* Nuevo div para envolver la tabla \*/}

<table className="table table-striped table-hover align-middle"> {/\* Clases de Bootstrap \*/}

<thead className="table-dark"> {/\* Clases de Bootstrap \*/}

<tr>

<th scope="col">ID</th>

<th scope="col">Empleado</th>

<th scope="col">Departamento</th>

<th scope="col">Sueldo</th>

</tr>

</thead>

<tbody>

<tr>

<th scope="row">1</th>

<td>Larry</td>

<td>Administración</td>

<td>$5000</td>

</tr>

<tr>

<th scope="row">2</th>

<td>Mark</td>

<td>Sistemas</td>

<td>$7000</td>

</tr>

{/\* Dejamos solo dos filas por ahora, se generarán dinámicamente más adelante \*/}

</tbody>

</table>

</div>

</div>

)

}

export default ListadoEmpleados

1. **Actualiza los encabezados de la tabla (<th>):**
   * Cambia los nombres de las columnas para que coincidan con los datos de los empleados: ID, Empleado, Departamento y Sueldo.
2. **Añade clases de estilo a la tabla (<table>):**
   * Agrega las siguientes clases de Bootstrap al elemento <table> para mejorar su apariencia:
     + table-striped: Alterna el color de fondo de las filas.
     + table-hover: Resalta la fila al pasar el cursor sobre ella.
     + align-middle: Alinea verticalmente el contenido de las celdas al centro.
   * Añade la clase table-dark al <thead> para un encabezado oscuro.
3. **Verifica en el navegador:**
   * Guarda los cambios. Ahora deberías ver una tabla bien formateada con los encabezados y las dos filas de ejemplo, aplicando los estilos de Bootstrap que acabas de configurar.

**Resumen y Próximos Pasos**

¡Felicidades! Has creado con éxito tu primer componente de React, lo has integrado en la aplicación principal y has empezado a estilizarlo usando las poderosas clases de Bootstrap.

**Logros:**

* Creaste el componente ListadoEmpleados.js y lo organizaste en la carpeta Empleados.
* Integraste ListadoEmpleados en App.jsx, haciendo que tu aplicación lo renderice.
* Aprendiste a usar className en lugar de class para aplicar estilos de Bootstrap en JSX.
* Aplicaste estilos en línea en React usando objetos JavaScript.
* Incorporaste una tabla de Bootstrap y la personalizaste con clases como table-striped, table-hover, align-middle, y table-dark.

En el próximo vídeo, comenzaremos a agregar funcionalidad dinámica a esta tabla, haciendo que los datos de los empleados se carguen de forma programática.

¿Estás listo para darle vida a esta tabla con datos reales?

**Manual (Parte 4): Conexión con el Backend y Gestión de Estado**

En esta parte del manual, nos enfocaremos en la integración de nuestro frontend React con el backend. Utilizaremos **hooks de React** (useState y useEffect) para manejar el estado de los datos y realizaremos peticiones HTTP al backend usando la librería **Axios**.

**Paso 1: Configuración de la URL Base y Ajuste de CORS**

Antes de iniciar la conexión, necesitamos definir la URL de nuestro backend y asegurarnos de que la comunicación entre el frontend (puerto 5173) y el backend (puerto 8080) sea posible.

1. **Define la URL Base en ListadoEmpleados.js:**
   * Dentro de la función ListadoEmpleados, pero antes de la sentencia return, define una constante para la URL base de tu API de empleados:

JavaScript

import React, { useEffect, useState } from 'react' // Asegúrate de importar los hooks aquí

function ListadoEmpleados() {

const URL\_BASE = "http://localhost:8080/rh-app/empleados"; // Define la URL base de tu backend

// ... el resto de tu código

1. **Verifica y Corrige el Puerto en el Backend (Spring Boot):**
   * **¡Importante!** El video menciona un error común: si tu backend (Spring Boot) está configurado para permitir peticiones solo desde http://localhost:3000 (el puerto por defecto de create-react-app), debes cambiarlo al puerto que usa Vite (http://localhost:5173).
   * **En tu controlador de Spring Boot**, la anotación @CrossOrigin debe apuntar al puerto de tu aplicación React:

Java

// En tu controlador de Spring Boot

@CrossOrigin(origins = "http://localhost:5173") // Asegúrate de que coincida con el puerto de Vite

public class EmpleadoControlador {

// ...

}

* + Si realizaste este cambio, **detén y reinicia tu aplicación Spring Boot** para que los cambios de configuración surtan efecto.

**Paso 2: Declaración del Estado con useState**

Vamos a usar el hook useState para declarar una variable de estado que almacenará la lista de empleados que obtendremos del backend.

1. **Declara el estado para los empleados:**
   * Dentro de la función ListadoEmpleados y después de la definición de URL\_BASE, agrega la siguiente línea:

JavaScript

import React, { useEffect, useState } from 'react'

function ListadoEmpleados() {

const URL\_BASE = "http://localhost:8080/rh-app/empleados";

const [empleados, setEmpleados] = useState([]); // 'empleados' será nuestro arreglo, 'setEmpleados' la función para actualizarlo. Inicializamos con un arreglo vacío.

// ... el resto de tu código

* + **useState([])**: Inicializa empleados como un arreglo vacío. setEmpleados es la función que usaremos para actualizar este arreglo.

**Paso 3: Instalación e Importación de Axios**

Para hacer peticiones HTTP de forma sencilla y eficiente, utilizaremos la librería Axios.

1. **Abre una nueva terminal en VS Code:**
   * Haz clic en el icono de **"Terminal"** en la barra superior y selecciona **"New Terminal"**.
   * También puedes usar el atajo Ctrl + Shift + (la tilde, debajo de Esc).
   * Esto abrirá una segunda terminal, mientras la primera sigue ejecutando tu aplicación React (npm run dev).
2. **Instala Axios:**
   * En la nueva terminal, ejecuta el siguiente comando:

Bash

npm install axios

* + Espera a que la instalación se complete.

1. **Importa Axios en ListadoEmpleados.js:**
   * Vuelve a ListadoEmpleados.js y agrega la línea de importación al principio del archivo:

JavaScript

import React, { useEffect, useState } from 'react'

import axios from 'axios'; // Importa Axios

// ... el resto de tu código

**Paso 4: Creación de la Función cargarEmpleados**

Esta función se encargará de realizar la petición GET a nuestro backend para obtener la lista de empleados.

1. **Define la función cargarEmpleados:**
   * Después de la declaración del estado (useState), define esta función asíncrona:

JavaScript

// ...

const [empleados, setEmpleados] = useState([]);

const cargarEmpleados = async () => {

try {

const resultado = await axios.get(URL\_BASE); // Realiza la petición GET

console.log("Resultado de cargar empleados:", resultado.data); // Imprime los datos recibidos

setEmpleados(resultado.data); // Actualiza el estado 'empleados' con los datos del backend

} catch (error) {

console.error("Error al cargar empleados:", error);

}

};

// ... el resto de tu código

* + **async y await**: Se usan para manejar operaciones asíncronas de manera más legible. await pausa la ejecución de la función hasta que la promesa (axios.get) se resuelva.
  + **axios.get(URL\_BASE)**: Realiza una petición GET a la URL definida.
  + **resultado.data**: La respuesta de Axios contiene la propiedad data, que es donde se encuentran los datos reales del backend.
  + **setEmpleados(resultado.data)**: Esta es la clave. Al llamar a setEmpleados, React detecta un cambio en el estado empleados y **re-renderiza** el componente, lo que es esencial para mostrar los datos actualizados en la tabla.

**Paso 5: Uso del Hook useEffect para Cargar Datos al Iniciar**

El hook useEffect nos permite ejecutar código después de que el componente se renderiza. Lo usaremos para llamar a cargarEmpleados una vez cuando la página se carga.

1. **Usa useEffect para llamar a cargarEmpleados:**
   * Después de la función cargarEmpleados, agrega el useEffect:

JavaScript

// ...

const cargarEmpleados = async () => { /\* ... \*/ };

useEffect(() => {

cargarEmpleados(); // Llama a la función para cargar los empleados

}, []); // El arreglo vacío `[]` asegura que el efecto se ejecute solo una vez al montar el componente

// ... el resto de tu código

* + **[] (Arreglo de dependencias vacío)**: Este segundo argumento es crucial. Le dice a React que este efecto solo debe ejecutarse **una vez** después del primer renderizado del componente, similar al ciclo de vida componentDidMount en componentes de clase. Si omites el arreglo, el efecto se ejecutaría en cada renderizado (lo que podría llevar a bucles infinitos en este caso).

**Paso 6: Verificación de la Conexión**

Ahora, vamos a comprobar si todo funciona correctamente.

1. **Guarda ListadoEmpleados.js.**
2. **Abre las herramientas de desarrollador de tu navegador:**
   * Presiona F12 o Ctrl + Shift + I (o Cmd + Option + I en Mac).
   * Ve a la pestaña **"Consola"**.
3. **Observa la salida en la consola:**
   * Deberías ver un mensaje como "Resultado de cargar empleados:" seguido de un objeto o arreglo de JavaScript que contiene los datos de tus empleados provenientes del backend. Esto confirma que la conexión es exitosa y que el frontend está recibiendo los datos.
4. Resultado de cargar empleados: (2) [{...}, {...}]
   * Si ves errores CORS (Cross-Origin Resource Sharing) en la consola, revisa el **Paso 1** de este manual y asegúrate de que tu backend esté configurado para permitir peticiones desde http://localhost:5173.

**Resumen y Próximos Pasos**

¡Excelente! Has dado un paso gigante al conectar tu aplicación React con el backend de Spring Boot. Ahora tu frontend es capaz de obtener datos de forma asíncrona.

**Logros:**

* Configuraste la **URL base** para la conexión al backend.
* Ajustaste la configuración **CORS** en tu backend para permitir peticiones desde el puerto de Vite.
* Utilizaste el hook **useState** para manejar el estado de los empleados en tu componente.
* Instalaste e importaste la librería **Axios** para realizar peticiones HTTP.
* Creaste una función **asíncrona (cargarEmpleados)** para obtener datos del backend.
* Empleaste el hook **useEffect** para cargar los datos de los empleados automáticamente cuando el componente se monta por primera vez.
* Verificaste la conexión exitosa observando los datos en la consola del navegador.

El siguiente paso es iterar sobre el arreglo empleados que ya tienes en el estado y mostrar cada empleado dinámicamente en la tabla HTML, en lugar de las filas estáticas.

**Manual (Parte 5): Renderizado Dinámico de Empleados y Formato de Sueldo**

En esta sección, daremos vida a nuestra tabla al **iterar sobre el arreglo de empleados** obtenido del backend y mostraremos cada uno de ellos dinámicamente. Además, aplicaremos un **formato monetario** al campo del sueldo.

**Paso 1: Iterar sobre el Arreglo de Empleados en la Tabla**

Vamos a modificar el cuerpo de nuestra tabla (<tbody>) para que renderice una fila por cada empleado en nuestro estado.

1. **Abre src/Empleados/ListadoEmpleados.js.**
2. **Localiza el <tbody>:**
   * Dentro del <tbody>, elimina las filas estáticas (<tr>...</tr>) que agregamos previamente.
3. **Agrega el código de iteración:**
   * Dentro del <tbody>, abre llaves {} para insertar código JavaScript.
   * Utiliza el método **map()** para iterar sobre el arreglo empleados. Cada iteración te dará un objeto empleado y su index.

|  |
| --- |
| JavaScript  import React, { useEffect, useState } from 'react'  import axios from 'axios';  // No hemos importado NumericFormat aún, lo haremos en el Paso 4.  function ListadoEmpleados() {  const URL\_BASE = "http://localhost:8080/rh-app/empleados";  const [empleados, setEmpleados] = useState([]);  const cargarEmpleados = async () => {  try {  const resultado = await axios.get(URL\_BASE);  console.log("Resultado de cargar empleados:", resultado.data);  setEmpleados(resultado.data);  } catch (error) {  console.error("Error al cargar empleados:", error);  }  };  useEffect(() => {  cargarEmpleados();  }, []);  return (  <div className='container text-center'>  <div className="container">  <h3 className="mt-3" style={{ margin: "30px" }}>Sistema de Recursos Humanos</h3>  </div>  <div className="container">  <table className="table table-striped table-hover align-middle">  <thead className="table-dark">  <tr>  <th scope="col">ID</th>  <th scope="col">Empleado</th>  <th scope="col">Departamento</th>  <th scope="col">Sueldo</th>  </tr>  </thead>  <tbody>  {/\* Iteramos el arreglo de empleados \*/}  {  empleados.map((empleado, index) => (  <tr key={index}> {/\* Importante: Cada elemento en una lista debe tener una 'key' única \*/}  <th scope="row">{empleado.idEmpleado}</th> {/\* Accedemos a la propiedad 'idEmpleado' \*/}  <td>{empleado.nombre}</td>  <td>{empleado.departamento}</td>  <td>{empleado.sueldo}</td>  </tr>  ))  }  </tbody>  </table>  </div>  </div>  )  } |

JavaScript

import React, { useEffect, useState } from 'react'

import axios from 'axios';

// No hemos importado NumericFormat aún, lo haremos en el Paso 4.

function ListadoEmpleados() {

const URL\_BASE = "http://localhost:8080/rh-app/empleados";

const [empleados, setEmpleados] = useState([]);

const cargarEmpleados = async () => {

try {

const resultado = await axios.get(URL\_BASE);

console.log("Resultado de cargar empleados:", resultado.data);

setEmpleados(resultado.data);

} catch (error) {

console.error("Error al cargar empleados:", error);

}

};

useEffect(() => {

cargarEmpleados();

}, []);

return (

<div className='container text-center'>

<div className="container">

<h3 className="mt-3" style={{ margin: "30px" }}>Sistema de Recursos Humanos</h3>

</div>

<div className="container">

<table className="table table-striped table-hover align-middle">

<thead className="table-dark">

<tr>

<th scope="col">ID</th>

<th scope="col">Empleado</th>

<th scope="col">Departamento</th>

<th scope="col">Sueldo</th>

</tr>

</thead>

<tbody>

{/\* Iteramos el arreglo de empleados \*/}

{

empleados.map((empleado, index) => (

<tr key={index}> {/\* Importante: Cada elemento en una lista debe tener una 'key' única \*/}

<th scope="row">{empleado.idEmpleado}</th> {/\* Accedemos a la propiedad 'idEmpleado' \*/}

<td>{empleado.nombre}</td>

<td>{empleado.departamento}</td>

<td>{empleado.sueldo}</td>

</tr>

))

}

</tbody>

</table>

</div>

</div>

)

}

export default ListadoEmpleados

* + **key={index}**: En React, cuando se renderizan listas de elementos, cada elemento debe tener una prop key única. Esto ayuda a React a identificar qué elementos han cambiado, se han añadido o se han eliminado, optimizando las actualizaciones del DOM. Aunque index funciona, en aplicaciones más grandes con reordenamiento o eliminación de elementos, es preferible usar un ID único del propio dato (e.g., empleado.idEmpleado) si está disponible y es estable. Para este caso, index es suficiente.
  + **empleado.idEmpleado**, **empleado.nombre**, etc.: Accedemos a las propiedades de cada objeto empleado que recibimos del backend. Asegúrate de que los nombres de las propiedades (idEmpleado, nombre, departamento, sueldo) coincidan exactamente con los nombres de las propiedades en los objetos JSON que devuelve tu API.

1. **Verifica en el navegador:**
   * Guarda los cambios. Deberías ver que la tabla ahora muestra los datos de los empleados que provienen de tu backend.

**Paso 2: Instalación de react-number-format**

Para darle un formato adecuado al sueldo (ej. "$20,000.00"), instalaremos una librería dedicada.

1. **Abre una nueva terminal en VS Code** (si no tienes una abierta aparte de la que ejecuta npm run dev).
2. **Instala la librería:**

Bash

npm install react-number-format

* + Espera a que la instalación se complete.

**Paso 3: Aplicar Formato Monetario al Sueldo**

Ahora, usaremos el componente NumericFormat de la librería instalada para formatear el campo del sueldo.

1. **Abre src/Empleados/ListadoEmpleados.js.**
2. **Importa NumericFormat:**
   * Agrega la siguiente línea al principio del archivo:

JavaScript

import React, { useEffect, useState } from 'react'

import axios from 'axios';

import { NumericFormat } from 'react-number-format'; // Importa el componente

1. **Usa NumericFormat en la celda del sueldo:**
   * Reemplaza <td>{empleado.sueldo}</td> por el componente NumericFormat con sus atributos:

JavaScript

// ...

<tbody>

{

empleados.map((empleado, index) => (

<tr key={index}>

<th scope="row">{empleado.idEmpleado}</th>

<td>{empleado.nombre}</td>

<td>{empleado.departamento}</td>

<td>

{/\* Aplicamos el formato numérico \*/}

<NumericFormat

value={empleado.sueldo}

displayType={"text"}

thousandSeparator={","}

prefix={"$"}

decimalScale={2}

fixedDecimalScale={true}

/>

</td>

</tr>

))

}

</tbody>

// ...

* + **value={empleado.sueldo}**: El valor numérico que se va a formatear.
  + **displayType={"text"}**: Indica que el componente debe mostrar el valor como texto (no como un campo de entrada editable).
  + **thousandSeparator={","}**: Usa una coma como separador de miles.
  + **prefix={"$"}**: Agrega el símbolo de dólar como prefijo.
  + **decimalScale={2}**: Muestra dos dígitos decimales.
  + **fixedDecimalScale={true}**: Asegura que siempre se muestren dos decimales, incluso si son ceros (ej., 20000.00).

1. **Verifica en el navegador:**
   * Guarda los cambios. Ahora, los sueldos en tu tabla deberían aparecer con el formato monetario ($XX,XXX.XX).

**Resumen y Próximos Pasos**

¡Excelente trabajo! Has completado el listado de empleados de forma dinámica y con un formato profesional para los valores monetarios.

**Logros:**

* **Renderizado dinámico**: Iteraste sobre el arreglo de empleados y generaste una fila de tabla (<tr>) por cada empleado.
* **Prop key**: Entendiste y aplicaste la importancia de la prop key para elementos de lista en React.
* **Acceso a propiedades**: Accediste a las propiedades de los objetos empleado (idEmpleado, nombre, departamento, sueldo).
* **Formato monetario**: Instalaste e implementaste la librería react-number-format para dar un formato profesional al campo del sueldo, incluyendo separadores de miles, prefijo de moneda y control de decimales.

Con esto, la funcionalidad básica de "listar empleados" está completa visualmente. El siguiente paso en una aplicación de Recursos Humanos sería agregar la capacidad de **añadir nuevos empleados**, lo que implicaría crear un formulario y enviar datos al backend.

¿Qué te gustaría hacer a continuación? ¿Crear un formulario para añadir empleados, o quizá explorar la edición y eliminación?

**Guía Docente: Implementación de Barra de Navegación con Bootstrap y React**

**1. Introducción**

¡Hola y bienvenidos! En esta sesión, continuaremos desarrollando nuestra aplicación de sistema de recursos humanos. Ya tenemos nuestra lista de empleados y ahora vamos a implementar la opción para **agregar un nuevo empleado**. Sin embargo, antes de abordar esa funcionalidad, vamos a incorporar un **menú de navegación** para mejorar la usabilidad de nuestra aplicación. Para ello, utilizaremos el framework de CSS **Bootstrap**.

**2. Creación de la Carpeta de Plantillas y Componente de Navegación**

Vamos a organizar nuestros archivos para que los componentes compartidos puedan ser fácilmente accesibles.

1. **Cerrar Archivos Abiertos:** Si tienes archivos abiertos en tu entorno de desarrollo, ciérralos para tener una vista más limpia.
2. **Crear Carpeta Plantilla:**
   * En la carpeta Source de tu proyecto, haz clic derecho.
   * Selecciona "Nuevo Folder" y nómbralo Plantilla. Esta carpeta contendrá todos los archivos de plantilla que se compartirán entre diferentes páginas.
3. **Crear Archivo Navegacion.js:**
   * Dentro de la carpeta Plantilla, haz clic derecho.
   * Selecciona "Nuevo Archivo" y nómbralo Navegacion.js.
4. **Crear Componente Funcional de React:**
   * Dentro de Navegacion.js, escribe R.f.c. (React Functional Component) y presiona **Tab** o **Enter**. Esto generará automáticamente la estructura básica de un componente funcional de React.

JavaScript

import React from 'react';

export default function Navegacion() {

return (

<div>

</div>

)

}

**3. Integración de la Barra de Navegación de Bootstrap**

Ahora, vamos a incorporar el código de una barra de navegación de Bootstrap en nuestro componente.

1. **Añadir div Contenedor:**
   * Elimina el div existente dentro del return de Navegacion.js.
   * Agrega un nuevo div con el className de container. Este div servirá como contenedor para nuestra barra de navegación.

JavaScript

import React from 'react';

export default function Navegacion() {

return (

<div className='container'>

{/\* Aquí irá el código de la barra de navegación de Bootstrap \*/}

</div>

)

}

1. **Obtener Código de Bootstrap:**
   * Abre tu navegador y ve a la documentación de Bootstrap (puedes buscar "Bootstrap Docs").
   * En la sección de búsqueda, escribe "Navbar".
   * Haz clic en el enlace "How it Works" (o similar) para ver la documentación de la barra de navegación.
   * Busca un ejemplo de barra de navegación que sea simple y que contenga un título y algunas opciones. El ejemplo que utilizaremos tiene un título y algunos enlaces.
   * Copia el código HTML de esa barra de navegación.
2. **Pegar Código en Navegacion.js:**
   * Pega el código copiado de Bootstrap dentro del div con className='container' en tu archivo Navegacion.js.
3. **Reemplazar class por className:**
   * En React, los atributos class de HTML deben ser reemplazados por className.
   * Presiona Ctrl + F (o Cmd + F en Mac) para abrir la barra de búsqueda y reemplazo.
   * Busca el atributo class.
   * Haz clic en la flecha para expandir la opción de reemplazo.
   * En el campo "Reemplazar con", escribe className.
   * Haz clic en el icono para reemplazar todas las ocurrencias de class por className.

Tu código dentro de Navegacion.js debería lucir similar a esto (con el código de Bootstrap pegado y los class cambiados a className):

JavaScript

import React from 'react';

export default function Navegacion() {

return (

<div className='container'>

<nav className="navbar navbar-expand-lg bg-body-tertiary">

<div className="container-fluid">

<a className="navbar-brand" href="#">Navbar</a>

<button className="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarSupportedContent" aria-controls="navbarSupportedContent" aria-expanded="false" aria-label="Toggle navigation">

<span className="navbar-toggler-icon"></span>

</button>

<div className="collapse navbar-collapse" id="navbarSupportedContent">

<ul className="navbar-nav me-auto mb-2 mb-lg-0">

<li className="nav-item">

<a className="nav-link active" aria-current="page" href="#">Home</a>

</li>

<li className="nav-item">

<a className="nav-link" href="#">Link</a>

</li>

{/\* ... otros elementos ... \*/}

</ul>

{/\* ... formulario de búsqueda si existe ... \*/}

</div>

</div>

</nav>

</div>

)

}

**4. Mostrar el Componente de Navegación en App.js**

Para que nuestra barra de navegación sea visible en la aplicación, necesitamos importarla y renderizarla en el componente principal, App.js.

1. **Abrir App.js:**
   * Navega a tu archivo App.js.
2. **Importar Navegacion:**
   * Al inicio del archivo, importa el componente Navegacion:

JavaScript

import Navegacion from './Plantilla/Navegacion'; // Asegúrate de la ruta correcta

1. **Renderizar Navegacion y ListadoEmpleados:**
   * Antes del componente ListadoEmpleados, agrega el componente Navegacion.
   * **Importante:** Las expresiones JSX deben tener un solo elemento principal. Para resolver esto, envolveremos ambos componentes dentro de un div con el className de container.

JavaScript

import './App.css';

import ListadoEmpleados from './empleados/ListadoEmpleados';

import Navegacion from './Plantilla/Navegacion';

function App() {

return (

<div className='container'> {/\* Elemento primario \*/}

<Navegacion/>

<ListadoEmpleados/>

</div>

);

}

export default App;

1. **Verificar en el Navegador:**
   * Guarda los cambios y verifica tu aplicación en el navegador. Deberías ver la barra de navegación en la parte superior, ajustándose automáticamente al tamaño de la pantalla.

**5. Personalización de la Barra de Navegación**

Ahora, vamos a personalizar los estilos y los enlaces de nuestra barra de navegación.

1. **Modificar Estilos de la Barra de Navegación (Navegacion.js):**
   * En el componente Navegacion.js, busca la etiqueta <nav>.
   * Modifica el className para usar un tema oscuro (navbar-dark) y un color de fondo primario (bg-primary).

JavaScript

<nav className="navbar navbar-expand-lg navbar-dark bg-primary">

{/\* ... \*/}

</nav>

* + Verifica en el navegador: la barra de navegación debería ahora ser azul con texto blanco.

1. **Personalizar el Título de la Barra de Navegación:**
   * En el <a> con className="navbar-brand", cambia el texto a "Sistema de Recursos Humanos".
   * El href de este enlace será la página de inicio, simplemente /.

JavaScript

<a className="navbar-brand" href="/">Sistema de Recursos Humanos</a>

1. **Configurar Enlaces de Navegación:**
   * Dentro del <ul className="navbar-nav">, deja solo dos elementos <li>. Elimina los demás.
   * **Primer Enlace (Inicio):**
     + Cambia el texto a "Inicio".
     + El href será /.
     + Asegúrate de que tenga las clases nav-link y active (si quieres que esté resaltado por defecto).

JavaScript

<li className="nav-item">

<a className="nav-link active" aria-current="page" href="/">Inicio</a>

</li>

* + **Segundo Enlace (Agregar Empleado):**
    - Cambia el texto a "Agregar Empleado".
    - El href será /agregar. Por ahora, este enlace no funcionará, ya que configuraremos las rutas en el siguiente video.

JavaScript

<li className="nav-item">

<a className="nav-link" href="/agregar">Agregar Empleado</a>

</li>

El bloque completo de navegación dentro de Navegacion.js debería lucir así:

JavaScript

import React from 'react';

export default function Navegacion() {

return (

<div className='container'>

<nav className="navbar navbar-expand-lg navbar-dark bg-primary">

<div className="container-fluid">

<a className="navbar-brand" href="/">Sistema de Recursos Humanos</a>

<button className="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarSupportedContent" aria-controls="navbarSupportedContent" aria-expanded="false" aria-label="Toggle navigation">

<span className="navbar-toggler-icon"></span>

</button>

<div className="collapse navbar-collapse" id="navbarSupportedContent">

<ul className="navbar-nav me-auto mb-2 mb-lg-0">

<li className="nav-item">

<a className="nav-link active" aria-current="page" href="/">Inicio</a>

</li>

<li className="nav-item">

<a className="nav-link" href="/agregar">Agregar Empleado</a>

</li>

</ul>

</div>

</div>

</nav>

</div>

)

}

1. **Verificar Cambios:**
   * Guarda los cambios y regresa a tu aplicación. Deberías ver el menú con las opciones "Inicio" y "Agregar Empleado".
   * Haz clic en "Inicio" y "Agregar Empleado". Observarás que la página se refresca. Por ahora, el enlace de "Agregar Empleado" no redirige a una página diferente; esto lo configuraremos con el concepto de rutas en el próximo video.

**6. Conclusión y Próximos Pasos**

Hemos implementado con éxito nuestra barra de navegación utilizando Bootstrap y React. Esta barra de navegación nos permitirá estructurar mejor nuestra aplicación y facilitar la navegación entre diferentes secciones.

En el próximo video, aplicaremos el concepto de **rutas (routing)** para que los enlaces de nuestra barra de navegación nos permitan movernos entre distintas vistas de nuestra aplicación.

¡Eso es todo por este video! Programen lo que hemos realizado y continuamos en el siguiente. ¡Saludos!

**Guía Docente: Implementación de la Funcionalidad "Agregar Empleado" en el Backend (Spring Boot)**

**1. Introducción**

¡Hola y bienvenidos nuevamente! En esta sesión, vamos a continuar con nuestra aplicación de sistema de recursos humanos. Hasta el momento, hemos logrado **listar todos los objetos de tipo Empleado** tanto desde el *backend* (Spring Boot) como desde el *frontend* (React), estableciendo una comunicación efectiva a través de Axios.

Ahora, nos centraremos en la funcionalidad para **agregar un nuevo empleado**. Para ello, modificaremos nuestro controlador en el *backend* para que pueda recibir y procesar peticiones de tipo **POST**.

**2. Modificación del Controlador EmpleadoControlador (Spring Boot)**

Vamos a agregar un nuevo método a nuestra clase EmpleadoControlador en el proyecto de Spring Boot para manejar las solicitudes de tipo POST.

1. **Asegúrate de que el Proyecto de Spring Boot esté Ejecutándose:**
   * Para que los cambios que realicemos sean efectivos, tu aplicación de Spring Boot debe estar en ejecución. Si no lo está, iníciala.
2. **Agregar el Método agregarEmpleado:**
   * Abre el archivo EmpleadoControlador.java.
   * Dentro de la clase, añade el siguiente método:

Java

import org.springframework.web.bind.annotation.PostMapping;

import org.springframework.web.bind.annotation.RequestBody;

import org.springframework.http.ResponseEntity; // Opcional, pero buena práctica para envolver la respuesta

// ... otras importaciones ...

@RestController

@RequestMapping("/rh-app") // Asegúrate de tener esta anotación o la que corresponda para el path base

public class EmpleadoControlador {

private static final Logger logger = LoggerFactory.getLogger(EmpleadoControlador.class);

@Autowired

private EmpleadoServicio empleadoServicio;

// ... Método para listar empleados (GetMapping) ...

@PostMapping("/empleados") // Anotación para peticiones POST a /empleados

public Empleado agregarEmpleado(@RequestBody Empleado empleado) { // @RequestBody mapea el JSON del cuerpo a un objeto Empleado

logger.info("Empleado a agregar: " + empleado); // Imprime el empleado recibido para depuración

return empleadoServicio.guardarEmpleado(empleado); // Llama al servicio para guardar el empleado

}

}

**Explicación del Código:**

* + @PostMapping("/empleados"): Esta anotación indica que este método manejará las **peticiones HTTP de tipo POST** dirigidas a la URL /empleados. Es la misma URL que usamos para GET, pero el tipo de método (POST vs. GET) las diferencia.
  + public Empleado agregarEmpleado(@RequestBody Empleado empleado):
    - Empleado: El método devolverá un objeto de tipo Empleado (el empleado guardado con su ID generado).
    - @RequestBody Empleado empleado: Esta anotación es crucial. Le indica a Spring que el **objeto Empleado viene en el cuerpo (body) de la petición HTTP** en formato JSON, y que debe ser mapeado automáticamente a un objeto Empleado de Java.
  + logger.info("Empleado a agregar: " + empleado);: Imprimimos la información del empleado recibido en los logs del *backend* para verificar que los datos se están recibiendo correctamente.
  + return empleadoServicio.guardarEmpleado(empleado);: Delegamos la lógica de guardado al servicio EmpleadoServicio. Recordemos que el método guardarEmpleado del servicio (save del JpaRepository) sirve tanto para **insertar** (si el ID es nulo) como para **actualizar** (si el ID es diferente de nulo).

**3. Prueba de la Funcionalidad POST con Postman**

Ahora, vamos a probar nuestro nuevo *endpoint* POST utilizando Postman para asegurarnos de que la comunicación y el guardado funcionan correctamente.

1. **Detener y Volver a Ejecutar la Aplicación de Spring Boot:**
   * Es **fundamental** que detengas tu aplicación de Spring Boot y la vuelvas a iniciar. Los cambios en los *endpoints* del controlador no se cargan automáticamente sin un reinicio.
2. **Abrir Postman y Crear una Nueva Petición:**
   * Abre Postman.
   * Haz clic en el icono para crear una **nueva petición** (generalmente un signo + o "New").
3. **Configurar la Petición:**
   * **Método:** Selecciona **POST** en el menú desplegable del método HTTP.
   * **URL:** Ingresa la URL de tu *endpoint*: http://localhost:8080/rh-app/empleados (Asegúrate de que el puerto y el *context-path* sean los correctos para tu configuración).
4. **Configurar el Cuerpo de la Petición (Body):**
   * Haz clic en la pestaña **Body**.
   * Selecciona la opción **raw**.
   * En el menú desplegable a la derecha, elige **JSON**.
5. **Preparar el JSON del Empleado:**
   * Para enviar un objeto de tipo Empleado, necesitamos un JSON válido. Puedes copiar un objeto existente de tu lista de empleados o crear uno desde cero.
   * Copia un objeto JSON de un empleado existente y pégalo en el área de texto del cuerpo (body).
   * **Elimina el atributo idEmpleado** (o déjalo como null). Para una inserción, el ID debe ser nulo para que la base de datos lo genere automáticamente.
   * **Modifica los valores** de nombre, departamento y sueldo para el nuevo empleado que deseas agregar.

**Ejemplo de JSON para el cuerpo de la petición:**

JSON

{

"nombre": "Carlos Lara",

"departamento": "Contabilidad",

"sueldo": 30000.00

}

1. **Enviar la Petición:**
   * Haz clic en el botón **Send**.
2. **Verificar la Respuesta:**
   * Deberías recibir un **estatus 200 OK**.
   * En la sección de respuesta de Postman, observarás el objeto Empleado que acabas de guardar, incluyendo el idEmpleado generado por la base de datos (por ejemplo, idEmpleado: 3).

JSON

{

"idEmpleado": 3,

"nombre": "Carlos Lara",

"departamento": "Contabilidad",

"sueldo": 30000.0

}

1. **Verificar en el Backend (Logs):**
   * Revisa la consola de tu aplicación de Spring Boot. Deberías ver el mensaje de log: Empleado a agregar: Empleado(idEmpleado=null, nombre=Carlos Lara, departamento=Contabilidad, sueldo=30000.0)
   * También deberías ver un *log* de la consulta INSERT a la tabla empleado ejecutada por Hibernate/JPA.
2. **Verificar en el Frontend (React):**
   * Vuelve a tu aplicación de React en el navegador.
   * **Refresca la página**. Deberías ver el nuevo registro de "Carlos Lara, Contabilidad, 30000" añadido a la lista de empleados.

**4. Conclusión y Próximos Pasos**

¡Excelente! Con esto, hemos confirmado que la funcionalidad para **agregar un empleado en el *backend* está funcionando correctamente**. Nuestro controlador puede recibir peticiones POST con los datos del empleado en formato JSON, y el servicio se encarga de persistirlos en la base de datos.

En el siguiente video, continuaremos con esta funcionalidad, pero esta vez desde el ***frontend* en React**. Implementaremos la interfaz de usuario para que los usuarios puedan introducir los datos de un nuevo empleado y enviarlos a nuestro *backend*.

¡Nos vemos en el siguiente video! ¡Saludos!

**Guía Docente: Implementación de Rutas en el Frontend con React Router DOM**

**1. Introducción**

¡Hola y bienvenidos nuevamente! Vamos a continuar con el caso de **agregar un empleado** en nuestra aplicación. Ya tenemos esta funcionalidad implementada en el *backend* (Spring Boot), así que ahora es el momento de integrarla en nuestro *frontend* con **React**.

Para ello, necesitamos una forma de **cambiar entre diferentes vistas o componentes** en nuestra aplicación cuando el usuario hace clic en los enlaces de navegación. Actualmente, al hacer clic en "Agregar Empleado", la URL cambia a /agregar (como se puede ver en la parte inferior del navegador), pero la aplicación no responde a ese cambio de ruta.

Para manejar las rutas en React, vamos a utilizar una librería externa muy popular llamada **React Router DOM**.

**2. Instalación de React Router DOM**

El primer paso es instalar el módulo de React Router DOM en nuestro proyecto.

1. **Abrir la Consola/Terminal:**
   * Abre tu consola o terminal en el directorio raíz de tu proyecto de React.
2. **Instalar el Componente:**
   * Ejecuta el siguiente comando:

Bash

npm install react-router-dom

* + Presiona Enter y espera a que la instalación se complete.

**3. Configuración de Rutas en App.js**

Una vez instalado react-router-dom, vamos a configurar nuestras rutas en el archivo App.js, que es el componente principal de nuestra aplicación.

1. **Abrir App.js:**
   * Navega a tu archivo App.js.
2. **Importar Browser Router, Routes y Route:**
   * Al inicio del archivo, importa los componentes necesarios de react-router-dom. Estos componentes nos permitirán definir y manejar las rutas de nuestra aplicación.

JavaScript

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

* + **Nota:** Anteriormente, importábamos ListadoEmpleados y Navegacion directamente. Ahora, BrowserRouter actuará como un contenedor principal para todas nuestras rutas.

1. **Estructurar las Rutas:**
   * Reemplaza el div que contiene Navegacion y ListadoEmpleados por la estructura de BrowserRouter, Routes y Route.

JavaScript

import './App.css';

import ListadoEmpleados from './empleados/ListadoEmpleados';

import Navegacion from './Plantilla/Navegacion';

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

function App() {

return (

<div className='container'>

<BrowserRouter> {/\* Contenedor principal de las rutas \*/}

<Navegacion /> {/\* El componente de navegación se mostrará en todas las rutas \*/}

<Routes> {/\* Contenedor para definir nuestras rutas \*/}

<Route exact path='/' element={<ListadoEmpleados />} /> {/\* Define la ruta raíz \*/}

</Routes>

</BrowserRouter>

</div>

);

}

export default App;

**Explicación de la Estructura:**

* + **<BrowserRouter>:** Este es el componente principal que habilita la funcionalidad de enrutamiento en tu aplicación de React. Debe envolver toda la lógica de enrutamiento.
  + **<Navegacion />:** Colocamos el componente de Navegacion fuera de <Routes> pero dentro de <BrowserRouter>. Esto asegura que la barra de navegación **se muestre en todas las páginas**, independientemente de la ruta activa.
  + **<Routes>:** Este componente es un contenedor para todas las definiciones individuales de Route. Routes renderiza la primera Route que coincide con la URL actual.
  + **<Route exact path='/' element={<ListadoEmpleados />} />:**
    - **Route:** Define una ruta individual.
    - **exact:** (Opcional, pero recomendado para la ruta raíz) Asegura que la ruta solo coincida si la URL es **exactamente** la especificada en path. Sin exact, / coincidiría con cualquier ruta que comience con / (como /agregar).
    - **path='/':** Especifica la URL para esta ruta. En este caso, la ruta raíz (/).
    - **element={<ListadoEmpleados />}:** Indica qué componente debe renderizarse cuando la URL coincide con path. Aquí, se mostrará el ListadoEmpleados.

1. **Verificar en el Navegador:**
   * Guarda los cambios en App.js.
   * Abre tu aplicación en el navegador. Deberías seguir viendo el menú de navegación y la lista de empleados como antes.
   * Haz clic en el enlace "Inicio" en tu barra de navegación. La URL debería ser /, y el ListadoEmpleados seguirá visible.

**4. Conclusión y Próximos Pasos**

¡Perfecto! Hemos sentado las bases para el enrutamiento en nuestra aplicación de React utilizando react-router-dom. Hemos configurado la ruta raíz (/) para mostrar el componente ListadoEmpleados.

En el siguiente video, vamos a **configurar la ruta para "Agregar Empleado"** (/agregar) y empezaremos a crear el componente que permitirá a los usuarios ingresar los datos de un nuevo empleado.

¡Nos vemos en el siguiente video! ¡Saludos!

**Guía Docente: Implementación del Componente "Agregar Empleado" en React**

**1. Introducción**

¡Hola y bienvenidos nuevamente! Vamos a continuar con nuestro sistema de recursos humanos. Ahora, el objetivo es agregar el componente que procesará nuestra petición de "Agregar empleado".

**2. Creación del Componente AgregarEmpleado.js**

1. **Crear el Archivo:**
   * Dentro de la carpeta empleados, crea un nuevo archivo llamado AgregarEmpleado.js.
2. **Estructura Básica del Componente:**
   * En AgregarEmpleado.js, utiliza el atajo RFC (React Functional Component) y presiona Enter para generar la estructura básica de un componente funcional de React.

JavaScript

import React from 'react';

export default function AgregarEmpleado() {

return (

<div>

</div>

);

}

**3. Configuración de la Nueva Ruta en App.js**

1. **Importar el Componente AgregarEmpleado:**
   * En App.js, importa el componente AgregarEmpleado.

JavaScript

import AgregarEmpleado from './empleados/AgregarEmpleado';

1. **Agregar la Ruta:**
   * Dentro del componente <Routes>, después de la ruta existente para la página de inicio (/), agrega una nueva ruta para /agregar.

JavaScript

<Routes>

<Route exact path='/' element={<ListadoEmpleados />} />

<Route exact path='/agregar' element={<AgregarEmpleado />} />

</Routes>

* + exact path='/agregar': Define la ruta exacta para el componente AgregarEmpleado.
  + element={<AgregarEmpleado />}: Especifica que el componente AgregarEmpleado se renderizará cuando la URL sea /agregar.

**4. Modificación del Menú de Navegación (Navegacion.js)**

1. **Importar el Componente Link:**
   * En Navegacion.js, importa el componente Link de react-router-dom.

JavaScript

import { Link } from 'react-router-dom';

1. **Reemplazar el Enlace <a> por Link:**
   * En el enlace de "Agregar Empleado", reemplaza la etiqueta <a> por <Link>.
   * Reemplaza el atributo href por to.

JavaScript

<li className="nav-item">

<Link className="nav-link" to="/agregar">Agregar Empleado</Link>

</li>

* + to="/agregar": Especifica la ruta a la que se dirigirá el enlace.

**5. Creación del Formulario en AgregarEmpleado.js**

1. **Obtener Código de Bootstrap:**
   * Ve a la documentación de Bootstrap y busca "Form".
   * Copia el código de un formulario básico.
2. **Pegar el Código en AgregarEmpleado.js:**
   * Dentro del div contenedor en AgregarEmpleado.js, pega el código del formulario de Bootstrap.
3. **Ajustes del Código:**
   * Reemplaza todos los atributos class por className.
   * Modifica las etiquetas (<label>) y los campos de entrada (<input>) para que correspondan a los atributos de un empleado: nombre, departamento y sueldo.
   * Asegúrate de que el campo de sueldo sea de tipo number y permita valores decimales (usando step="any").
   * Agrega atributos name a los campos de entrada, con los mismos valores que los id.
   * Agrega el atributo required al campo de nombre.
   * Cambia el atributo form a htmlForm (ya que form es una palabra reservada en JavaScript).

JavaScript

import React from 'react';

import { Link } from 'react-router-dom'; // Asegúrate de importar Link aquí también

export default function AgregarEmpleado() {

return (

<div className="container">

<div className="container text-center" style={{ margin: '30px' }}>

<h3>Agregar Empleado</h3>

</div>

<form htmlForm>

<div className="mb-3">

<label htmlFor="nombre" className="form-label">Nombre del empleado</label>

<input type="text" className="form-control" id="nombre" name="nombre" required />

</div>

<div className="mb-3">

<label htmlFor="departamento" className="form-label">Departamento</label>

<input type="text" className="form-control" id="departamento" name="departamento" />

</div>

<div className="mb-3">

<label htmlFor="sueldo" className="form-label">Sueldo</label>

<input type="number" className="form-control" id="sueldo" name="sueldo" step="any" />

</div>

<div className="text-center">

<button type="submit" className="btn btn-warning btn-sm me-3">Agregar</button>

<Link to="/" className="btn btn-danger btn-sm">Regresar</Link>

</div>

</form>

</div>

);

}

**6. Verificación**

1. **Ejecutar la Aplicación:**
   * Asegúrate de que tu aplicación de React esté en ejecución.
2. **Navegar a la Página "Agregar Empleado":**
   * Haz clic en el enlace "Agregar Empleado" en la barra de navegación. Deberías ver el formulario.
3. **Probar el Formulario:**
   * Intenta enviar el formulario sin completar el campo de nombre. Deberías ver un mensaje de validación del navegador, indicando que el campo es requerido.
   * Completa el formulario y envíalo. Por ahora, los datos se enviarán en la URL (como una petición GET). En el siguiente video, modificaremos esto para enviar una petición POST.

**7. Conclusión**

En este video, hemos creado el componente AgregarEmpleado, configurado la ruta /agregar y agregado un formulario básico utilizando Bootstrap. En el próximo video, nos centraremos en la lógica para enviar los datos del formulario al *backend* utilizando una petición POST.

Saludos.

**Guía Docente: Conectando el Formulario de "Agregar Empleado" con el Backend en React**

**1. Introducción**

Bien, vamos a continuar con nuestra aplicación de sistema de recursos humanos. Ya tenemos la parte visual de nuestro formulario de "Agregar Empleado". Ahora, el siguiente paso es **relacionar este formulario con un estado en React** para capturar los datos de entrada, y luego **procesar el envío (submit) del formulario** para enviar esos datos a nuestro *backend* a través de una petición POST.

**2. Gestión del Estado del Formulario con useState**

Vamos a utilizar el *hook* useState para crear un objeto que represente los datos del empleado que se están ingresando en el formulario.

1. **Abrir AgregarEmpleado.js:**
   * Abre el archivo AgregarEmpleado.js.
2. **Importar useState y useNavigate:**
   * Al inicio del archivo, importa el *hook* useState de React y el *hook* useNavigate de react-router-dom. También necesitaremos axios para la petición HTTP.

JavaScript

import React, { useState } from 'react';

import { useNavigate } from 'react-router-dom';

import axios from 'axios'; // Asegúrate de tener axios instalado (npm install axios)

1. **Definir el Estado del Empleado:**
   * Dentro del componente funcional AgregarEmpleado, define una constante para el estado del empleado. Inicializaremos las propiedades nombre, departamento y sueldo como cadenas vacías.

JavaScript

export default function AgregarEmpleado() {

let navegacion = useNavigate(); // Hook para la navegación

const [empleado, setEmpleado] = useState({

nombre: "",

departamento: "",

sueldo: ""

});

// Extraer las propiedades del objeto empleado para facilitar el acceso

const { nombre, departamento, sueldo } = empleado;

// ... resto del componente ...

}

1. **Crear la Función onInputChange:**
   * Esta función se encargará de actualizar el estado del empleado cada vez que haya un cambio en los campos del formulario. Utilizaremos el operador *spread* (...) para mantener los valores existentes y solo actualizar la propiedad que ha cambiado.

JavaScript

const onInputChange = (e) => {

setEmpleado({ ...empleado, [e.target.name]: e.target.value });

};

**Explicación:**

* + e.target.name: Obtiene el valor del atributo name del campo de entrada que disparó el evento (ej. nombre, departamento, sueldo).
  + e.target.value: Obtiene el valor actual del campo de entrada.
  + [e.target.name]: e.target.value: Esta es una sintaxis de propiedades computadas en JavaScript. Permite usar una variable (el name del input) como clave para un objeto.

1. **Asociar los Campos del Formulario con el Estado:**
   * Para cada campo input en tu formulario, agrega los atributos value y onChange:
     + **value={nombre} (o departamento, sueldo):** Esto hace que el valor del input esté controlado por el estado de React.
     + **onChange={(e) => onInputChange(e)}:** Llama a la función onInputChange cada vez que el valor del input cambia.

JavaScript

<form htmlForm>

<div className="mb-3">

<label htmlFor="nombre" className="form-label">Nombre del empleado</label>

<input

type="text"

className="form-control"

id="nombre"

name="nombre"

required

value={nombre} // Asociado al estado 'nombre'

onChange={(e) => onInputChange(e)} // Actualiza el estado en cada cambio

/>

</div>

<div className="mb-3">

<label htmlFor="departamento" className="form-label">Departamento</label>

<input

type="text"

className="form-control"

id="departamento"

name="departamento"

value={departamento} // Asociado al estado 'departamento'

onChange={(e) => onInputChange(e)} // Actualiza el estado en cada cambio

/>

</div>

<div className="mb-3">

<label htmlFor="sueldo" className="form-label">Sueldo</label>

<input

type="number"

className="form-control"

id="sueldo"

name="sueldo"

step="any"

value={sueldo} // Asociado al estado 'sueldo'

onChange={(e) => onInputChange(e)} // Actualiza el estado en cada cambio

/>

</div>

{/\* ... botones ... \*/}

</form>

**3. Manejo del Envío del Formulario (onSubmit) y Petición POST**

Ahora, vamos a configurar el evento onSubmit del formulario para enviar los datos al *backend* cuando el usuario haga clic en "Agregar".

1. **Definir la Función onSubmit:**
   * Esta función será asíncrona porque realizará una petición HTTP.

JavaScript

const onSubmit = async (e) => {

e.preventDefault(); // Previene el comportamiento por defecto de recargar la página

const urlBase = "http://localhost:8080/rh-app/empleados"; // URL de tu backend

await axios.post(urlBase, empleado); // Envía el objeto empleado al backend

navegacion('/'); // Redirige a la página de inicio después de agregar

};

**Explicación:**

* + e.preventDefault(): Es crucial llamar a esta función para evitar que el navegador recargue la página, lo cual es el comportamiento por defecto de un formulario HTML.
  + const urlBase = "http://localhost:8080/rh-app/empleados";: Define la URL de tu *endpoint* POST en el *backend*.
  + await axios.post(urlBase, empleado);:
    - axios.post(): Realiza una petición POST.
    - urlBase: La URL a la que se envía la petición.
    - empleado: El objeto JavaScript empleado (que contiene nombre, departamento, sueldo con los valores del formulario) se envía como el cuerpo (body) de la petición en formato JSON. Axios lo convierte automáticamente a JSON.
  + navegacion('/');: Después de que la petición POST se completa (gracias a await), utilizamos el *hook* useNavigate para redirigir al usuario a la página de inicio (/), donde se mostrará la lista actualizada de empleados.

1. **Asociar onSubmit al Formulario:**
   * En la etiqueta <form>, agrega el atributo onSubmit.

JavaScript

<form onSubmit={(e) => onSubmit(e)}>

{/\* ... campos del formulario ... \*/}

</form>

**4. Código Completo de AgregarEmpleado.js**

Tu archivo AgregarEmpleado.js debería lucir así:

JavaScript

import React, { useState } from 'react';

import { useNavigate, Link } from 'react-router-dom'; // Importa Link si aún no lo has hecho

import axios from 'axios';

export default function AgregarEmpleado() {

let navegacion = useNavigate();

const [empleado, setEmpleado] = useState({

nombre: "",

departamento: "",

sueldo: ""

});

const { nombre, departamento, sueldo } = empleado;

const onInputChange = (e) => {

setEmpleado({ ...empleado, [e.target.name]: e.target.value });

};

const onSubmit = async (e) => {

e.preventDefault();

const urlBase = "http://localhost:8080/rh-app/empleados";

try {

await axios.post(urlBase, empleado);

navegacion('/');

} catch (error) {

console.error("Error al agregar empleado:", error);

// Puedes agregar una alerta o mensaje de error al usuario aquí

}

};

return (

<div className="container">

<div className="container text-center" style={{ margin: '30px' }}>

<h3>Agregar Empleado</h3>

</div>

<form onSubmit={(e) => onSubmit(e)}>

<div className="mb-3">

<label htmlFor="nombre" className="form-label">Nombre del empleado</label>

<input

type="text"

className="form-control"

id="nombre"

name="nombre"

required

value={nombre}

onChange={(e) => onInputChange(e)}

/>

</div>

<div className="mb-3">

<label htmlFor="departamento" className="form-label">Departamento</label>

<input

type="text"

className="form-control"

id="departamento"

name="departamento"

value={departamento}

onChange={(e) => onInputChange(e)}

/>

</div>

<div className="mb-3">

<label htmlFor="sueldo" className="form-label">Sueldo</label>

<input

type="number"

className="form-control"

id="sueldo"

name="sueldo"

step="any"

value={sueldo}

onChange={(e) => onInputChange(e)}

/>

</div>

<div className="text-center">

<button type="submit" className="btn btn-warning btn-sm me-3">Agregar</button>

<Link to="/" className="btn btn-danger btn-sm">Regresar</Link>

</div>

</form>

</div>

);

}

**5. Verificación Final**

1. **Asegúrate de que ambos proyectos estén en ejecución:**
   * Tu aplicación de Spring Boot (backend).
   * Tu aplicación de React (frontend).
2. **Prueba la funcionalidad en el navegador:**
   * Abre tu aplicación React.
   * Haz clic en el enlace "Agregar Empleado".
   * Rellena el formulario con los datos de un nuevo empleado (ej., "Laura Zavala", "Recursos Humanos", "25000").
   * Haz clic en el botón "Agregar".
3. **Observa el comportamiento:**
   * La aplicación no debería recargar la página.
   * Después de un breve momento, serás redirigido a la página de inicio.
   * El nuevo empleado debería aparecer en la tabla de listado.
   * Revisa los logs de tu *backend* de Spring Boot; deberías ver el mensaje de Empleado a agregar y la consulta INSERT ejecutada.

**6. Conclusión**

¡Felicidades! Con esto, hemos completado la funcionalidad de **agregar un nuevo empleado** desde el *frontend* de React, enviando los datos a nuestro *backend* de Spring Boot a través de una petición POST. Hemos integrado el manejo de estado con useState, el control del formulario y la comunicación HTTP con axios.

Pongan en práctica lo que hemos visto y nos vemos en el siguiente video. ¡Saludos!

---

## Guía Docente: Implementación de Funcionalidad CRUD Completa (Editar y Eliminar) en el Sistema de Recursos Humanos

¡Hola y bienvenidos nuevamente a nuestra aplicación de sistema de Recursos Humanos!

En las secciones anteriores, logramos listar y agregar empleados. Ahora, vamos a completar las operaciones CRUD (Crear, Leer, Actualizar, Borrar) añadiendo las funcionalidades de \*\*editar y eliminar un empleado\*\*.

---

### 1. Funcionalidad "Editar Empleado" - Backend (Spring Boot)

Para editar un empleado, primero necesitamos una forma de buscarlo por su ID en el \*backend\* y luego un \*endpoint\* para actualizarlo.

#### 1.1. Buscar Empleado por ID (Backend)

1. \*\*Abrir `EmpleadoControlador.java`:\*\*

\* Ve a tu proyecto de Spring Boot y abre la clase `EmpleadoControlador`.

2. \*\*Agregar Método `obtenerEmpleadoPorId`:\*\*

\* Añade un nuevo método para manejar las peticiones GET con un ID en la URL. Este método devolverá un `ResponseEntity` que envuelve un objeto `Empleado`.

```java

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.server.ResponseStatusException; // Importar para manejar excepciones

// ... otras importaciones ...

@RestController

@RequestMapping("/rh-app")

public class EmpleadoControlador {

// ... otros métodos (listar, agregar) ...

@GetMapping("/empleados/{id}") // Petición GET a /empleados/{id}

public ResponseEntity<Empleado> obtenerEmpleadoPorId(@PathVariable Integer id) {

Empleado empleado = empleadoServicio.buscarEmpleadoPorId(id); // Suponiendo que tienes este método en EmpleadoServicio

if (empleado == null) {

// Si el empleado no se encuentra, lanzar una excepción personalizada

throw new RecursoNoEncontradoExcepcion("No se encontró el empleado con ID: " + id);

}

return ResponseEntity.ok(empleado); // Retorna el empleado encontrado con status 200 OK

}

}

```

\*\*Explicación:\*\*

\* `@GetMapping("/empleados/{id}")`: Mapea las peticiones GET que incluyen un ID en la URL.

\* `@PathVariable Integer id`: Extrae el `id` de la URL y lo asigna a la variable `id`.

\* `ResponseEntity<Empleado>`: Indica que el método devolverá una respuesta HTTP que contiene un objeto `Empleado` en el cuerpo.

\* `empleadoServicio.buscarEmpleadoPorId(id)`: Llama a un método en tu servicio para buscar el empleado por ID. Asegúrate de que este método exista y utilice el repositorio para buscar.

\* \*\*Manejo de Excepciones (`RecursoNoEncontradoExcepcion`):\*\* Es crucial manejar el caso en que el ID no exista.

#### 1.2. Crear Clase `RecursoNoEncontradoExcepcion`

1. \*\*Crear Paquete `Excepcion`:\*\*

\* Dentro del paquete principal de tu aplicación (`Gnrh` o similar), crea un nuevo paquete llamado `Excepcion`.

2. \*\*Crear Clase `RecursoNoEncontradoExcepcion.java`:\*\*

\* Dentro del paquete `Excepcion`, crea una nueva clase `RecursoNoEncontradoExcepcion`.

\* Esta clase debe extender de `RuntimeException` para no obligar a su manejo explícito (`try-catch`) y ser anotada con `@ResponseStatus` para devolver un estado HTTP 404.

```java

package com.gnrh.rhapp.excepcion; // Ajusta el paquete según tu proyecto

import org.springframework.http.HttpStatus;

import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus(HttpStatus.NOT\_FOUND) // Devuelve un 404 Not Found

public class RecursoNoEncontradoExcepcion extends RuntimeException {

public RecursoNoEncontradoExcepcion(String mensaje) {

super(mensaje); // Llama al constructor de la clase padre (RuntimeException)

}

}

```

#### 1.3. Actualizar Empleado (Backend - `PUT` Request)

Ahora, vamos a agregar el método para actualizar un empleado existente.

1. \*\*Abrir `EmpleadoControlador.java`:\*\*

\* Vuelve a la clase `EmpleadoControlador`.

2. \*\*Agregar Método `actualizarEmpleado`:\*\*

\* Añade un nuevo método que manejará las peticiones PUT. Recibirá el ID del empleado a actualizar desde la URL y el objeto `Empleado` con los nuevos datos en el cuerpo de la petición.

```java

import org.springframework.web.bind.annotation.PutMapping;

import org.springframework.web.bind.annotation.RequestBody;

// ... otras importaciones ...

@RestController

@RequestMapping("/rh-app")

public class EmpleadoControlador {

// ... otros métodos (listar, agregar, obtenerPorId) ...

@PutMapping("/empleados/{id}") // Petición PUT a /empleados/{id}

public ResponseEntity<Empleado> actualizarEmpleado(@PathVariable Integer id, @RequestBody Empleado empleadoRecibido) {

Empleado empleado = empleadoServicio.buscarEmpleadoPorId(id); // Buscar el empleado existente

if (empleado == null) {

throw new RecursoNoEncontradoExcepcion("El ID de empleado recibido no existe: " + id);

}

// Actualizar las propiedades del empleado encontrado con los datos recibidos

empleado.setNombre(empleadoRecibido.getNombre());

empleado.setDepartamento(empleadoRecibido.getDepartamento());

empleado.setSueldo(empleadoRecibido.getSueldo());

empleadoServicio.guardarEmpleado(empleado); // Guardar (actualizar) el empleado

return ResponseEntity.ok(empleado); // Retorna el empleado actualizado

}

}

```

\*\*Explicación:\*\*

\* `@PutMapping("/empleados/{id}")`: Mapea las peticiones PUT para actualizar un recurso específico.

\* `@RequestBody Empleado empleadoRecibido`: Mapea el JSON del cuerpo de la petición a un objeto `Empleado`.

\* \*\*Validación:\*\* Se busca el empleado por ID para asegurar que existe antes de intentar actualizarlo. Si no existe, se lanza la excepción `RecursoNoEncontradoExcepcion`.

\* \*\*Actualización de Propiedades:\*\* Las propiedades del objeto `empleado` obtenido de la base de datos se actualizan con los valores del `empleadoRecibido`.

\* `empleadoServicio.guardarEmpleado(empleado)`: Se reutiliza el método `guardarEmpleado` (que internamente usa `JpaRepository.save()`) para actualizar el registro, ya que el ID no es nulo.

#### 1.4. Probar con Postman (Backend)

1. \*\*Reiniciar la Aplicación de Spring Boot:\*\* \*\*¡Importante!\*\* Detén y vuelve a ejecutar tu aplicación de Spring Boot para que los nuevos \*endpoints\* sean reconocidos.

2. \*\*Probar `GET /empleados/{id}`:\*\*

\* En Postman, crea una nueva petición.

\* Método: `GET`

\* URL: `http://localhost:8080/rh-app/empleados/1` (cambia `1` por un ID existente).

\* Envía la petición. Deberías obtener un `200 OK` con los datos del empleado.

\* Prueba con un ID inexistente (ej., `http://localhost:8080/rh-app/empleados/99`). Deberías obtener un `404 Not Found`.

3. \*\*Probar `PUT /empleados/{id}`:\*\*

\* En Postman, crea una nueva petición.

\* Método: `PUT`

\* URL: `http://localhost:8080/rh-app/empleados/4` (asume que el ID 4 es un empleado existente).

\* Pestaña `Body`, selecciona `raw` y `JSON`.

\* Pega un JSON con los datos del empleado que deseas actualizar (el ID dentro del JSON puede omitirse o ser ignorado, el ID de la URL es el que se usa).

```json

{

"nombre": "Laura Zavala (Actualizado)",

"departamento": "Recursos Humanos (Nuevo)",

"sueldo": 25000.00

}

```

\* Envía la petición. Deberías obtener un `200 OK` con el empleado actualizado.

\* Verifica la lista de empleados en tu \*frontend\* de React para confirmar que los cambios se reflejaron.

---

### 2. Funcionalidad "Editar Empleado" - Frontend (React)

Ahora, vamos a implementar la parte del \*frontend\* para editar un empleado.

#### 2.1. Crear el Componente `EditarEmpleado.js`

1. \*\*Crear el Archivo:\*\*

\* En la carpeta `empleados`, crea un nuevo archivo llamado `EditarEmpleado.js`.

2. \*\*Copiar Contenido de `AgregarEmpleado.js`:\*\*

\* El formulario de edición será muy similar al de agregar. Copia \*\*todo el contenido\*\* de `AgregarEmpleado.js` y pégalo en `EditarEmpleado.js`.

3. \*\*Renombrar y Ajustar el Componente:\*\*

\* Cambia el nombre de la función exportada a `EditarEmpleado`.

\* Modifica el título `<h3>` a "Editar Empleado".

\* Cambia el texto del botón `Agregar` a `Guardar`.

```javascript

import React, { useState, useEffect } from 'react'; // Importa useEffect

import { useNavigate, Link, useParams } from 'react-router-dom'; // Importa useParams

import axios from 'axios';

export default function EditarEmpleado() {

let navegacion = useNavigate();

const { id } = useParams(); // Hook para obtener parámetros de la URL (ID)

const [empleado, setEmpleado] = useState({

nombre: "",

departamento: "",

sueldo: ""

});

const { nombre, departamento, sueldo } = empleado;

const onInputChange = (e) => {

setEmpleado({ ...empleado, [e.target.name]: e.target.value });

};

// Hook useEffect para cargar el empleado cuando el componente se monta

useEffect(() => {

cargarEmpleado();

}, []); // Array vacío para que se ejecute solo una vez al montar el componente

const cargarEmpleado = async () => {

const urlBase = "http://localhost:8080/rh-app/empleados";

const resultado = await axios.get(`${urlBase}/${id}`); // Petición GET para obtener el empleado por ID

setEmpleado(resultado.data); // Actualiza el estado del empleado con los datos cargados

};

const onSubmit = async (e) => {

e.preventDefault();

const urlBase = "http://localhost:8080/rh-app/empleados";

await axios.put(`${urlBase}/${id}`, empleado); // Cambia de POST a PUT y añade el ID a la URL

navegacion('/');

};

return (

<div className="container">

<div className="container text-center" style={{ margin: '30px' }}>

<h3>Editar Empleado</h3> {/\* Título modificado \*/}

</div>

<form onSubmit={(e) => onSubmit(e)}>

<div className="mb-3">

<label htmlFor="nombre" className="form-label">Nombre del empleado</label>

<input

type="text"

className="form-control"

id="nombre"

name="nombre"

required

value={nombre}

onChange={(e) => onInputChange(e)}

/>

</div>

<div className="mb-3">

<label htmlFor="departamento" className="form-label">Departamento</label>

<input

type="text"

className="form-control"

id="departamento"

name="departamento"

value={departamento}

onChange={(e) => onInputChange(e)}

/>

</div>

<div className="mb-3">

<label htmlFor="sueldo" className="form-label">Sueldo</label>

<input

type="number"

className="form-control"

id="sueldo"

name="sueldo"

step="any"

value={sueldo}

onChange={(e) => onInputChange(e)}

/>

</div>

<div className="text-center">

<button type="submit" className="btn btn-warning btn-sm me-3">Guardar</button> {/\* Texto del botón modificado \*/}

<Link to="/" className="btn btn-danger btn-sm">Regresar</Link>

</div>

</form>

</div>

);

}

```

#### 2.2. Configurar la Ruta de Edición en `App.js`

1. \*\*Abrir `App.js`:\*\*

\* Ve a tu archivo `App.js`.

2. \*\*Importar `EditarEmpleado`:\*\*

\* Importa el nuevo componente `EditarEmpleado`.

```javascript

import EditarEmpleado from './empleados/EditarEmpleado';

```

3. \*\*Agregar la Ruta para Editar:\*\*

\* Dentro de `<Routes>`, añade una nueva `Route` para `/editar/:id`. El `:id` indica que es un parámetro de la URL.

```javascript

<Routes>

<Route exact path='/' element={<ListadoEmpleados />} />

<Route exact path='/agregar' element={<AgregarEmpleado />} />

<Route exact path='/editar/:id' element={<EditarEmpleado />} /> {/\* Nueva ruta \*/}

</Routes>

```

#### 2.3. Añadir el Botón "Editar" en `ListadoEmpleados.js`

1. \*\*Abrir `ListadoEmpleados.js`:\*\*

\* Ve a tu archivo `ListadoEmpleados.js`.

2. \*\*Modificar la Tabla para Incluir la Columna de Acciones:\*\*

\* Añade un nuevo `<th>` en el encabezado de la tabla para "Acciones".

\* Dentro del `<td>` que itera sobre los empleados, añade un `div` con los botones "Editar" y "Eliminar".

```javascript

import { Link } from 'react-router-dom'; // Asegúrate de importar Link

// ... dentro del componente ListadoEmpleados ...

<table className="table table-striped table-hover align-middle">

<thead className='table-dark'>

<tr>

<th scope="col">ID</th>

<th scope="col">Empleado</th>

<th scope="col">Departamento</th>

<th scope="col">Sueldo</th>

<th scope="col">Acciones</th> {/\* Nueva columna \*/}

</tr>

</thead>

<tbody>

{

empleados.map((empleado, indice) => (

<tr key={indice}>

<th scope="row">{empleado.idEmpleado}</th>

<td>{empleado.nombre}</td>

<td>{empleado.departamento}</td>

<td>{empleado.sueldo}</td>

<td className='text-center'> {/\* Centrar contenido \*/}

<div>

{/\* Botón Editar \*/}

<Link to={`/editar/${empleado.idEmpleado}`}

className='btn btn-warning btn-sm me-3'>Editar</Link>

{/\* Botón Eliminar (se agregará después) \*/}

</div>

</td>

</tr>

))

}

</tbody>

</table>

```

#### 2.4. Probar la Funcionalidad "Editar" (Frontend)

1. \*\*Ejecutar Ambas Aplicaciones:\*\* Asegúrate de que tanto el \*backend\* como el \*frontend\* estén en funcionamiento.

2. \*\*Navegar y Probar:\*\*

\* Abre tu aplicación React.

\* Haz clic en el botón "Editar" junto a cualquier empleado en la lista.

\* Deberías ser redirigido al formulario de edición, y los campos del formulario deberían estar \*\*pre-llenados\*\* con los datos del empleado seleccionado.

\* Modifica algunos datos y haz clic en "Guardar".

\* Deberías ser redirigido de vuelta a la página de listado, y los cambios deberían verse reflejados en la tabla.

\* Verifica los logs del \*backend\* para confirmar que se realizó una petición PUT.

---

### 3. Funcionalidad "Eliminar Empleado" - Backend (Spring Boot)

Ahora, implementaremos el \*endpoint\* para eliminar un empleado en el \*backend\*.

#### 3.1. Eliminar Empleado (Backend - `DELETE` Request)

1. \*\*Abrir `EmpleadoControlador.java`:\*\*

\* Vuelve a tu clase `EmpleadoControlador`.

2. \*\*Agregar Método `eliminarEmpleado`:\*\*

\* Añade un nuevo método que manejará las peticiones `DELETE`. Recibirá el ID del empleado a eliminar desde la URL. Devolverá un `Map` (clave-valor) para indicar si la eliminación fue exitosa.

```java

import org.springframework.web.bind.annotation.DeleteMapping;

import java.util.HashMap;

import java.util.Map;

// ... otras importaciones ...

@RestController

@RequestMapping("/rh-app")

public class EmpleadoControlador {

// ... otros métodos (listar, agregar, obtenerPorId, actualizar) ...

@DeleteMapping("/empleados/{id}") // Petición DELETE a /empleados/{id}

public ResponseEntity<Map<String, Boolean>> eliminarEmpleado(@PathVariable Integer id) {

Empleado empleado = empleadoServicio.buscarEmpleadoPorId(id); // Buscar el empleado para validar

if (empleado == null) {

throw new RecursoNoEncontradoExcepcion("El ID de empleado recibido no existe: " + id);

}

empleadoServicio.eliminarEmpleado(empleado); // Suponiendo que tienes este método en EmpleadoServicio

Map<String, Boolean> respuesta = new HashMap<>();

respuesta.put("eliminado", Boolean.TRUE); // Indicamos que se eliminó

return ResponseEntity.ok(respuesta); // Retorna una respuesta JSON: {"eliminado": true}

}

}

```

\*\*Explicación:\*\*

\* `@DeleteMapping("/empleados/{id}")`: Mapea las peticiones DELETE para eliminar un recurso.

\* `ResponseEntity<Map<String, Boolean>>`: Indica que la respuesta será un JSON con una clave `String` y un valor `Boolean`.

\* \*\*Validación:\*\* Se verifica si el empleado existe antes de intentar eliminarlo.

\* `empleadoServicio.eliminarEmpleado(empleado)`: Llama a un método en tu servicio para eliminar el empleado. Asegúrate de que este método exista y utilice el repositorio para borrar.

\* `Map<String, Boolean> respuesta`: Se crea un mapa simple para la respuesta JSON.

#### 3.2. Probar con Postman (Backend)

1. \*\*Reiniciar la Aplicación de Spring Boot:\*\* \*\*¡Importante!\*\* Detén y vuelve a ejecutar tu aplicación de Spring Boot.

2. \*\*Probar `DELETE /empleados/{id}`:\*\*

\* En Postman, crea una nueva petición.

\* Método: `DELETE`

\* URL: `http://localhost:8080/rh-app/empleados/4` (cambia `4` por un ID de empleado que quieras eliminar).

\* Envía la petición.

\* Deberías obtener un `200 OK` con la respuesta `{"eliminado": true}`.

\* Verifica la lista de empleados en tu \*frontend\* de React o en tu base de datos para confirmar que el registro fue eliminado.

\* Prueba con un ID inexistente. Deberías obtener un `404 Not Found`.

---

### 4. Funcionalidad "Eliminar Empleado" - Frontend (React)

Finalmente, agregaremos el botón "Eliminar" en el \*frontend\* y la lógica para que haga la petición DELETE.

#### 4.1. Añadir el Botón "Eliminar" y el Método `eliminarEmpleado` en `ListadoEmpleados.js`

1. \*\*Abrir `ListadoEmpleados.js`:\*\*

\* Vuelve a tu archivo `ListadoEmpleados.js`.

2. \*\*Agregar el Botón "Eliminar":\*\*

\* Dentro del `div` de la columna "Acciones", después del botón "Editar", añade el botón "Eliminar".

```javascript

// ... dentro del <td> de acciones ...

<td className='text-center'>

<div>

<Link to={`/editar/${empleado.idEmpleado}`}

className='btn btn-warning btn-sm me-3'>Editar</Link>

{/\* Botón Eliminar \*/}

<button

onClick={() => eliminarEmpleado(empleado.idEmpleado)} // Llama al método eliminarEmpleado

className='btn btn-danger btn-sm'>Eliminar</button>

</div>

</td>

```

3. \*\*Definir el Método `eliminarEmpleado`:\*\*

\* Agrega la función `eliminarEmpleado` en tu componente `ListadoEmpleados`. Esta función será asíncrona y realizará la petición DELETE.

```javascript

// ... después del método cargarEmpleados ...

const eliminarEmpleado = async (id) => {

const urlBase = "http://localhost:8080/rh-app/empleados"; // Asegúrate de que URL\_BASE esté definida

await axios.delete(`${urlBase}/${id}`); // Petición DELETE

cargarEmpleados(); // Vuelve a cargar la lista para actualizar la vista

};

```

\*\*Explicación:\*\*

\* `onClick={() => eliminarEmpleado(empleado.idEmpleado)}`: Cuando el botón se hace clic, se llama a la función `eliminarEmpleado` y se le pasa el ID del empleado.

\* `axios.delete(`${urlBase}/${id}`):` Realiza la petición DELETE al \*backend\*, incluyendo el ID en la URL.

\* `cargarEmpleados()`: Una vez que el empleado se elimina en el \*backend\*, se llama nuevamente a `cargarEmpleados` para refrescar la lista de empleados en la interfaz de usuario, mostrando los datos actualizados (sin el registro eliminado).

#### 4.2. Verificación Final de la Funcionalidad "Eliminar" (Frontend)

1. \*\*Ejecutar Ambas Aplicaciones:\*\* Asegúrate de que tanto el \*backend\* como el \*frontend\* estén en funcionamiento.

2. \*\*Navegar y Probar:\*\*

\* Abre tu aplicación React.

\* Haz clic en el botón "Eliminar" junto a cualquier empleado.

\* El empleado debería desaparecer de la lista de inmediato.

\* Verifica los logs del \*backend\* para confirmar que se realizó una petición DELETE y luego una petición GET para recargar la lista.

---

### 5. Conclusión del Proyecto

¡Felicidades! Hemos terminado de implementar la funcionalidad CRUD completa en nuestra aplicación de sistema de Recursos Humanos.

\* Hemos creado nuestra aplicación \*\*Fullstack\*\* utilizando \*\*React\*\* para el \*frontend\*, donde las vistas se definieron en componentes y se comunicaron con el \*backend\* usando `axios`.

\* En el \*backend\*, utilizamos \*\*Spring Boot\*\* para crear nuestro controlador REST, capa de servicio, repositorio y entidad, comunicándonos con la base de datos para procesar peticiones HTTP (GET, POST, PUT, DELETE).

Esperamos que esta aplicación les haya sido tan agradable de crear como a nosotros el haberla desarrollado con ustedes. ¡Eso es todo por este proyecto y nos vemos en futuros proyectos!

¡Saludos!