**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?