INTRODUCCIÓN AL DESARROLLO FRONTEND

CON REACT

2023

ANDÉN Centro de Innovación y Emprendimientos Tecnológicos

SECRETARÍA DE EXTENSIÓN UNIVERSITARIA UTN - FRC SEU

UTN Facultad Regional Córdoba

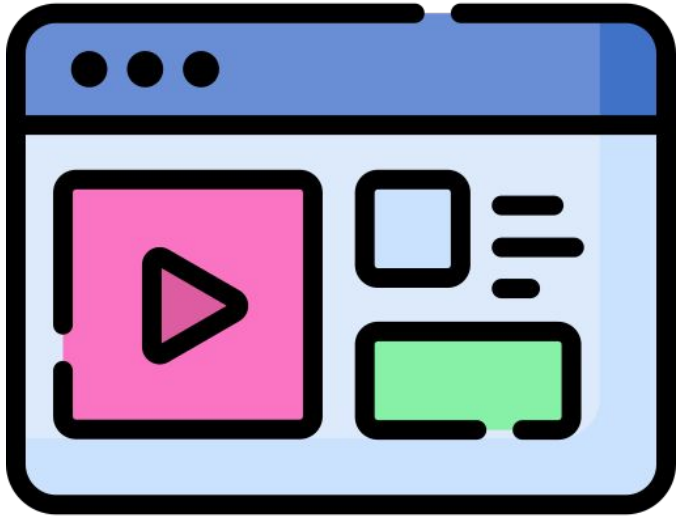
Agencia CÓRDOBA JOVEN

CÓRDOBA entre todos

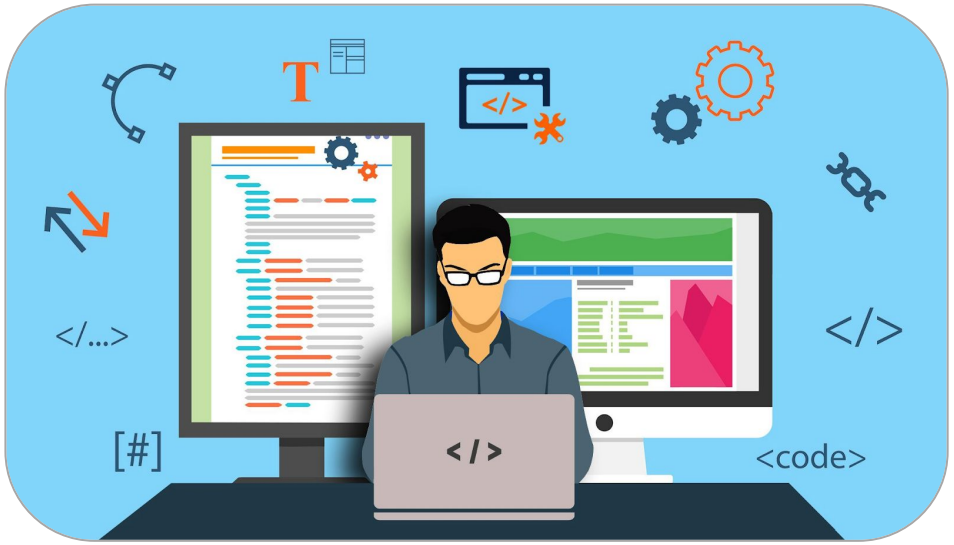
Clase 10 - Contenido

Desarrollo avanzado en React

- Técnicas avanzadas de React
- Optimización de rendimiento
- Mejoras en la experiencia del usuario



Objetivos de la Clase



- **Comprender la Importancia de una Estructura de Carpetas Adecuada:**
  - Los alumnos deberán entender cómo una estructura de carpetas bien organizada contribuye a la manejabilidad y escalabilidad de un proyecto React.
- **Adquirir Habilidades de Optimización y Gestión del Rendimiento:**
  - Aprender técnicas de optimización esenciales para garantizar un rendimiento óptimo de la aplicación, incluyendo la memoización y la optimización de renders.
- **Desarrollar Competencia en Técnicas Avanzadas de React:**
  - Familiarizarse con técnicas avanzadas como Lazy Loading, Suspense, y el manejo avanzado del estado usando Context API y Redux.
- **Aprender sobre Testing:**
  - Adquirir habilidades en la realización de pruebas unitarias y de integración.
- **Mejorar la Experiencia del Usuario y la Accesibilidad:**
  - Aprender a implementar mejoras en la experiencia del usuario como animaciones y transiciones, y a asegurar la accesibilidad en las aplicaciones React.

Estructura de Carpetas

- Una estructura de carpetas bien organizada es crucial para la manejabilidad y escalabilidad del proyecto.
- Facilita la navegación y comprensión del código a los desarrolladores.
- **Estructura Básica:**

  - src/
  - components/: Componentes reutilizables.
  - containers/: Contenedores o páginas de la aplicación.
  - assets/: Imágenes, estilos y otros recursos estáticos.
  - utils/: Funciones de utilidad y librerías auxiliares.
  - public/
  - tests/: Directorio para archivos de pruebas.

- **Buenas Prácticas:**
  - **Nombramiento consistente:** Utilizar un sistema de nombramiento coherente para archivos y carpetas.
  - **Organización lógica:** Agrupar archivos relacionados según funcionalidad o característica.
  - **Separación de concerns:** Separar la lógica, vista y estilos de manera clara.

Diferenciación entre Components y Pages

- **Carpeta components/:**
  - Aloja componentes reutilizables en diferentes partes de la aplicación.
  - Ejemplos: botones, tarjetas, barras de navegación.
- **Carpeta pages/:**
  - Aloja las "páginas" o "vistas" específicas de la aplicación.
  - Cada página representa una ruta o vista completa.
  - Componentes específicos de esa página se almacenan aquí.
- **Beneficios:**
  - Organización clara que facilita la navegación y comprensión del código.
  - Escalabilidad y manejabilidad mejoradas para proyectos en crecimiento.

```
src/
  - components/
    - Button.js
    - Card.js
    - Navbar.js
  - pages/
    - Home/
      - Home.js
      - HomeComponent.js
    - About/
      - About.js
      - AboutComponent.js
```

Componentes Controlados y No Controlados

La principal diferencia es cómo y dónde se maneja el estado de los elementos de entrada.

- Los componentes **controlados** mantienen el estado en React.
- Los componentes **no controlados** lo mantienen en el DOM.

Esto afecta cómo interactuamos con los valores de los elementos de entrada en tu código.

Componentes Controlados

```jsx
import React, { useState } from 'react';

function ControlledForm() {
  const [inputValue, setInputValue] = useState('');

  const handleChange = (event) => {
    setInputValue(event.target.value);
  };

  return (
    <input type="text" value={inputValue} onChange={handleChange} />
  );
}

export default ControlledForm;
```

Componentes No Controlados

```jsx
import React, { useRef } from 'react';

function UncontrolledForm() {
  const inputRef = useRef(null);

  const handleSubmit = (event) => {
    event.preventDefault();
    alert('Submitted: ' + inputRef.current.value);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input type="text" ref={inputRef} />
      <button type="submit">Submit</button>
    </form>
  );
}

export default UncontrolledForm;
```

Manejo de Estado Avanzado: Context API

```jsx
import React, { useContext, useState } from 'react';

const MyContext = React.createContext();

function ChildComponent() {
  const value = useContext(MyContext);
  return <div>{value}</div>;
}

function ParentComponent() {
  const [value, setValue] = useState('Hello, Context API!');
  return (
    <MyContext.Provider value={value}>
      <ChildComponent />
    </MyContext.Provider>
  );
}

export default ParentComponent;
```

Proporciona una forma de pasar datos a través del árbol de componentes sin tener que pasar props manualmente en cada nivel.

Manejo de Estado Avanzado: Redux

- **Redux** es una librería para gestionar el estado de la aplicación de manera predecible, centralizando el estado y permitiendo gestionarlo de manera eficaz.
- Facilita la gestión de estado en aplicaciones grandes y complejas.
- Middleware en Redux:
  - Amplía las capacidades de Redux permitiendo la gestión de efectos secundarios, como las llamadas a API.

Optimización: Uso Eficiente del Estado y Props

El renderizado innecesario en React ocurre cuando un componente se renderiza sin que haya habido cambios en su estado o props. Esto puede afectar negativamente a la performance de la aplicación, especialmente en componentes grandes o en aplicaciones con una gran cantidad de datos dinámicos.

React.memo:

- React.memo es una función de orden superior que memoriza el resultado del renderizado de un componente y solo vuelve a renderizar el componente si las props han cambiado.
- En este código, MyComponent solo se volverá a renderizar si props.value cambia.

```
import React from 'react';

const MyComponent = React.memo((props) => {
  console.log('Component rendered');
  return <div>{props.value}</div>;
});

export default MyComponent;
```

Optimización: SEO

- Significa Optimización del Motor de Búsqueda (Search Engine Optimization en inglés)
- Es un conjunto de estrategias y técnicas utilizadas para aumentar la visibilidad de un sitio web en los resultados de búsqueda orgánica (no pagada) de los motores de búsqueda, como Google, Bing, y Yahoo.
- El objetivo principal es aumentar el tráfico del sitio web al mejorar su posición en los resultados de búsqueda para ciertas palabras clave relevantes.

Optimización: Lazy Loading

El lazy loading es una técnica que permite cargar componentes en demanda, en lugar de cargar todos los componentes al iniciar la aplicación, lo que puede mejorar la velocidad de carga inicial.

React.lazy y Suspense:

- React.lazy permite cargar un componente de forma perezosa.
- Suspense permite mostrar un contenido de respaldo mientras se carga el componente.
- En este código, MyLazyComponent se cargará sólo cuando sea necesario renderizarlo, y se mostrará "Loading..." mientras se carga.

```
import React, { Suspense } from 'react';

const MyLazyComponent = React.lazy(() => import('./MyLazyComponent'));

const MyComponent = () => {
  return (
    <Suspense fallback={<div>Loading...</div>}>
      <MyLazyComponent />
    </Suspense>
  );
};

export default MyComponent;
```

Optimización: Optimización de Renderizado

Optimizar el renderizado implica evitar cálculos y operaciones innecesarias que pueden afectar la performance.

React.useMemo y React.useCallback:

- React.useMemo: Memoriza el resultado de un cálculo y solo lo recalcula si las dependencias cambian.
- React.useCallback: Memoriza una función y solo la recrea si las dependencias cambian.
- En este código, expensiveCalculation sólo se ejecutará de nuevo si values cambia, y handleClick solo se recreará si sus dependencias cambian (en este caso, nunca, ya que no tiene dependencias).

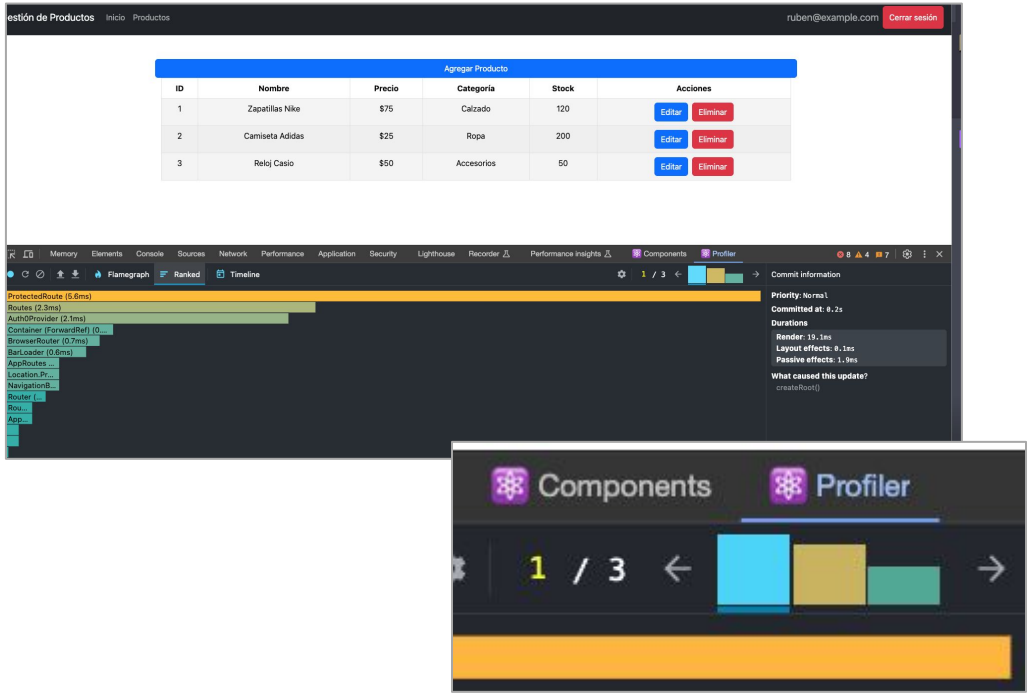
```
import React, { useMemo, useCallback } from 'react';

const MyComponent = ({ values }) => {
  const calculatedValue = useMemo(() => expensiveCalculation(values), [values]);
  const handleClick = useCallback(() => {
    // handle click
  }, []);

  return (
    <button onClick={handleClick}>{calculatedValue}</button>
  );
};

export default MyComponent;
```

Optimización: Profiling

El Profiling es una técnica utilizada para entender el comportamiento de una aplicación y encontrar áreas de optimización.

React DevTools Profiler:

- Permite identificar renderizados costosos.
- Ofrece una vista detallada de cómo y cuándo se están renderizando los componentes, ayudando a detectar posibles optimizaciones.



Optimización: Virtualización

La virtualización es una técnica que consiste en renderizar solo los elementos que caben en el área visible, mejorando el rendimiento en listas largas.

```
import { FixedSizeList as List } from 'react-window';

const MyList = ({ items }) => (
  <List
    height={500}
    itemCount={items.length}
    itemSize={35}
    width={300}
  >
    {({ index, style }) => (
      <div style={style}>
        {items[index]}
      </div>
    )}
  </List>
);
```

Estrategias de renderizado: Server Side Rendering (SSR)

SSR es una técnica que implica renderizar los componentes de React en el servidor en lugar de en el cliente, lo que resulta en una página completamente renderizada que se envía al navegador.

Beneficios:

- Mejora la SEO (Optimización del Motor de Búsqueda) ya que los crawlers pueden indexar el sitio más fácilmente.
- Proporciona un TTFB (Time To First Byte) más rápido, lo que mejora la percepción de rendimiento.

Implementación en React:

- Utilizando frameworks como Next.js que facilitan el SSR.

```
// pages/index.js en Next.js
function HomePage() {
  return <div>Welcome to Next.js!</div>
}

export default HomePage
```

Estrategias de renderizado: Static Site Generation (SSG)

SSG es una técnica que genera páginas estáticas en tiempo de construcción que luego se pueden servir sin necesidad de un servidor.

Beneficios:

- Extremadamente rápido ya que no hay tiempo de renderizado en el servidor ni tiempo de espera para las API en el cliente.
- Mejora la SEO.

Implementación en React:

- También se puede realizar con frameworks como Next.js.

```
// pages/posts/[id].js en Next.js
export async function getStaticProps(context) {
  const { id } = context.params;
  const postData = await getPostData(id);
  return {
    props: {
      postData
    }
  }
}

function Post({ postData }) {
  // Renderizar datos del post
  return (
    <div>
      {postData.title}
      {/* ... */}
    </div>
  );
}

export default Post;
```

Testing: Introducción

- Tipos de Pruebas:
  - **Pruebas Unitarias:** Verifican la funcionalidad de partes individuales de la aplicación.
  - **Pruebas de Integración:** Verifican la interacción entre diferentes partes de la aplicación.
- Bibliotecas de Testing:
  - **Jest:** Framework de testing muy popular para JavaScript.
  - **React Testing Library:** Biblioteca que proporciona utilidades para probar componentes de React.

Testing: Pruebas Unitarias

- Las pruebas unitarias son esenciales para verificar que cada parte de tu código funcione como se espera en aislamiento del resto de la aplicación.
- Ejemplo:
  - En el ejemplo proporcionado, se está realizando una prueba unitaria en un componente llamado **MyComponent**.
- Importaciones:
  - Se importa la función render de **'@testing-library/react',** que es utilizada para renderizar un componente en un entorno de prueba.
  - Se importa **MyComponent** desde su archivo respectivo.
- **Función test:**
  - Se define una nueva prueba utilizando la función test de Jest.
  - El primer argumento es una descripción de lo que se está probando.
  - El segundo argumento es una función que contiene la lógica de la prueba.
- **Renderizado:**
  - Se renderiza **MyComponent** utilizando la función render.
  - render devuelve un objeto que contiene varias funciones de utilidad para consultar el DOM resultante.
- **Consulta y Aserción:**
  - Se utiliza la función **getByText** para buscar un elemento en el DOM que contenga el texto "learn react".
  - Se utiliza la función **expect** junto con el matcher toBeInTheDocument para verificar que el elemento encontrado esté presente en el DOM.

```
import { render } from '@testing-library/react';
import MyComponent from './MyComponent';

test('renders learn react link', () => {
  const { getByText } = render(<MyComponent />);
  const linkElement = getByText(/aprendiendo react/i);
  expect(linkElement).toBeInTheDocument();
});
```

Testing: Pruebas de Integración

- Las pruebas de integración ayudan a asegurar que varias partes de la aplicación trabajen juntas como se espera. Esto puede incluir múltiples componentes, o incluso cómo interactúan los componentes con hooks y el estado de la aplicación.
- **Ejemplo:**
  - En el ejemplo proporcionado, se está realizando una prueba unitaria en un componente llamado **MyComponent**.
- **Importaciones**:
  - Se importan las funciones render y fireEvent de '@testing-library/react', que son utilizadas para renderizar un componente y simular eventos del usuario, respectivamente.
  - Se importa App desde su archivo respectivo.
- **Función test:**
  - Se define una nueva prueba utilizando la función test de Jest.
  - El primer argumento es una descripción de lo que se está probando.
  - El segundo argumento es una función que contiene la lógica de la prueba.
- **Renderizado:**
  - Se renderiza App utilizando la función render.
  - render devuelve un objeto que contiene varias funciones de utilidad para consultar el DOM resultante.
- **Simulación de Eventos:**
  - Se utiliza la función fireEvent para simular un clic en un botón que incrementa un contador.
  - Se busca el botón con el texto "increment" usando getByText y luego se simula un clic en ese botón.
- **Consulta y Aserción:**
  - Después de simular el clic, se busca un elemento en el DOM que contenga el texto "count: 1" usando getByText.
  - Se utiliza la función expect junto con el matcher toBeInTheDocument para verificar que el elemento encontrado esté presente en el DOM y que el contador se haya actualizado correctamente.

```
import { render, fireEvent } from
'@testing-library/react';
import App from './App';

test('renders and updates a counter', () => {
  const { getByText } = render(<App />);
  const button = getByText(/increment/i);
  fireEvent.click(button);
  const count = getByText(/count: 1/i);
  expect(count).toBeInTheDocument();
});
```

Testing: Mocking

- El **mocking** implica simular comportamientos o dependencias externas para aislar las pruebas y obtener resultados predecibles.
- **Utilidad:** Es útil cuando se desea evitar llamadas a APIs externas, bases de datos, o cualquier otra dependencia que podría hacer que la prueba sea menos predecible o más lenta.
- **En este ejemplo,** se utiliza jest.mock para simular el módulo ./api. Luego, dentro de una prueba, se especifica un valor de retorno falso para la función **fetchData** usando **mockResolvedValue**.

```
jest.mock('./api');  // Mocking del módulo './api'

// ... dentro de una prueba:

api.fetchData.mockResolvedValue(mockData);
// Definir un valor de retorno falso para fetchData
```

Bueno, Vamo a Codea!!!!

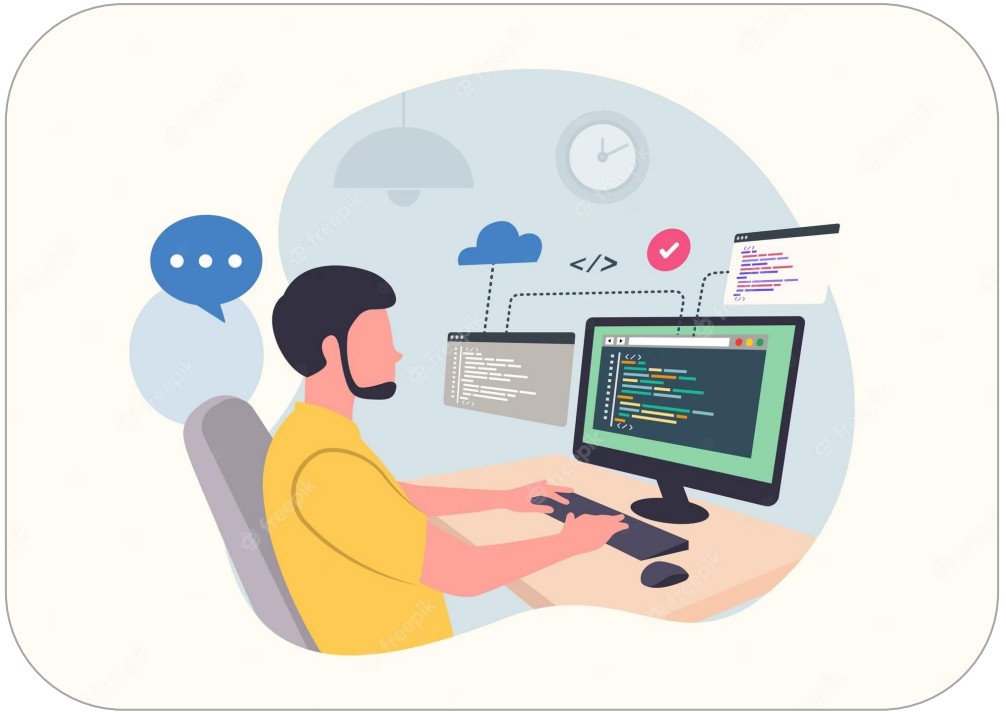
Enunciado:

Usar el repositorio como esqueleto de una aplicación de React. Su tarea será desarrollar una interfaz de usuario (UI) que permita a los usuarios interactuar con una lista de productos. Deberás implementar las siguientes características:

1. Listar Productos:
   - Mostrar una lista de productos existentes en una tabla o lista.
   - Cada entrada debe mostrar al menos el nombre del producto, precio y una opción para editar y eliminar el producto.
2. Crear Productos:
   - Implementar un formulario que permita a los usuarios agregar nuevos productos.
   - El formulario debe contener campos para el nombre del producto y el precio.
   - Validar que los campos no estén vacíos antes de permitir la sumisión.
3. Actualizar Productos:
   - Al seleccionar la opción de editar en un producto existente, mostrar un formulario con los detalles actuales del producto que permita modificar el nombre y/o el precio.
   - Validar que los campos no estén vacíos antes de permitir la sumisión.
4. Eliminar Productos:
   - Al seleccionar la opción de eliminar en un producto existente, mostrar un mensaje de confirmación antes de proceder con la eliminación.
   - Si el usuario confirma la eliminación, eliminar el producto de la lista.
5. Interacción con el Backend:
   - Todas las operaciones de creación, actualización y eliminación deben interactuar con un servidor backend para reflejar los cambios en la base de datos.
6. Manejo de Errores:
   - Implementar un manejo básico de errores que informe al usuario si algo sale mal durante las operaciones de creación, actualización o eliminación.
7. Estilización:
   - Aplicar estilos básicos para mejorar la apariencia de la interfaz de usuario.

Actividad: Gestión Productos

- Seguir las instrucciones de la actividad publicada en la UVE.



GRACIAS TOTALES

ANDÉN Centro de Innovación y Emprendimientos Tecnológicos

SECRETARÍA DE EXTENSIÓN UNIVERSITARIA UTN - FRC SEU

UTN Facultad Regional Córdoba

Agencia CÓRDOBA JOVEN

CÓRDOBA entre todos