Xtratego

ACADEMY

UTE EC


UTE
EC
tratego
ACADEMY

UTE EC | Xtratego ACADEMY

Modulo 2

CURSO

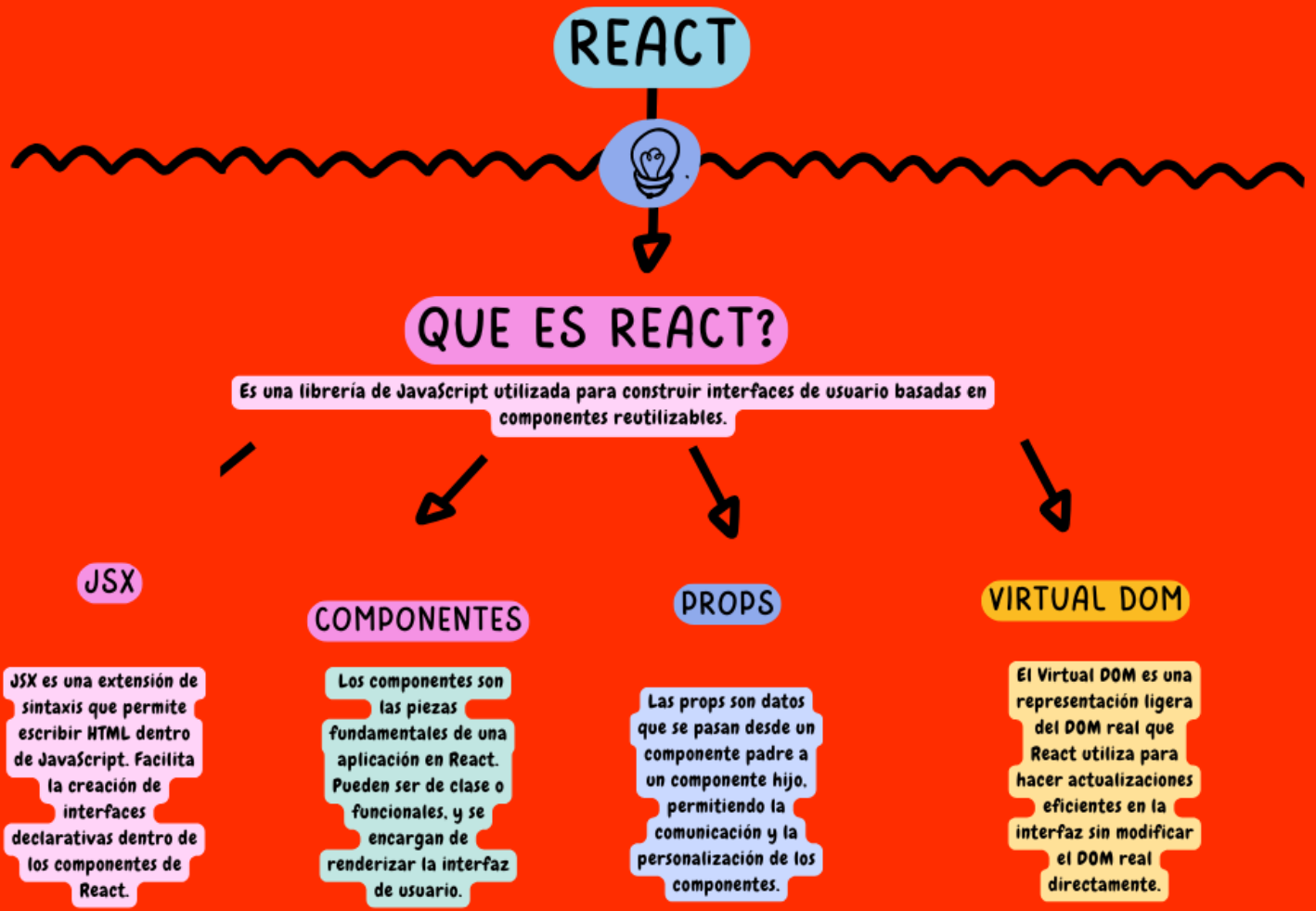
DESARROLLO FRONT END

xtratego.ec

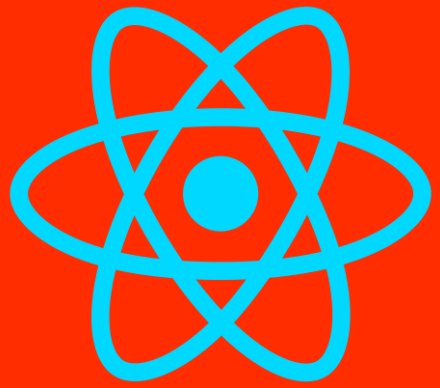
Unidad 1

Introducción a React.js

Introducción a React.js



Origen de React.js



Creación en Facebook en 2011 por Jordan Walke.

React fue diseñado para mejorar el rendimiento y la gestión de interfaces de usuario complejas en grandes aplicaciones

Ventajas de React

- **Modularidad:** Desarrollar aplicaciones utilizando componentes reutilizables.
- **Virtual DOM:** Mejora la eficiencia al actualizar solo los elementos necesarios en la interfaz.
- **Comunidad activa:** Ecosistema rico en herramientas y librerías.

UTE EC | Xtratego ACADEMY

Filosofía declarativa

React adopta un enfoque declarativo, lo que significa que los desarrolladores describen cómo debería verse la interfaz en un momento dado, y **React** se encarga de actualizar el DOM.

JSX

Sintaxis que permite escribir **HTML** dentro de **JavaScript**.

Aunque se parece al **HTML**, **JSX** permite el uso de lógica **JavaScript** dentro de las etiquetas

Virtual DOM

Es una representación ligera del **DOM** real. **React** actualiza de manera eficiente solo los componentes que han cambiado, lo que mejora el rendimiento en comparación con las manipulaciones directas del **DOM**

https://nodejs.org/es

Node.js es un entorno de ejecución de JavaScript del lado del servidor.

```
node --version
```

Ejemplo Node + Express

1

```
npm init
```

2

```
npm install express
```

4

```
node server.js
```

3

```
// Importar Express
const express = require('express');

// Crear una instancia de Express
const app = express();

// Configurar el puerto en el que escuchará el servidor
const port = 3000;

// Ruta básica de prueba
app.get('/', (req, res) => {
  res.send('¡Hola Mundo! Servidor Express en funcionamiento.');
});

// Iniciar el servidor
app.listen(port, () => {
  console.log(`Servidor escuchando en http://localhost:${port}`);
});
```

Configuración inicial React

```
npx create-react-app my-app
```

Herramienta para generar rápidamente un proyecto con la estructura base de **React**, sin la necesidad de configurar manualmente **Webpack, Babel, etc.**

Archivos clave en un proyecto React

index.html: Página HTML donde se montará la aplicación React.

index.js: Punto de entrada principal, donde se renderiza el componente raíz (App.js).

App.js: Componente raíz de la aplicación.

package.json: Información sobre dependencias y scripts del proyecto.

Entorno de desarrollo

React Developer Tools: Extensión para depurar aplicaciones React.

Hot Reloading: Recarga automática de la aplicación tras cambios en el código.

UTE EC | Xtratego ACADEMY

Componentes en React

Un **componente** es una función o clase de **JavaScript** que acepta **props** como entrada y devuelve elementos **React** que describen cómo debe verse la interfaz.

Los componentes permiten dividir la **UI** en piezas independientes y reutilizables

Componentes funcionales

Los componentes funcionales son funciones de **JavaScript** que aceptan **props** como argumento y devuelven **JSX**

```
function Welcome(props) {
  return <h1>Hola, {props.name}</h1>;
}
```

Componentes de clase

Son clases de JavaScript que extienden de React.Component y pueden tener estado interno.

```
class Welcome extends React.Component {
  render() {
    return <h1>Hola, {this.props.name}</h1>;
  }
}
```

Aunque los componentes de clase eran comunes en versiones anteriores de React, hoy en día se recomienda usar componentes funcionales con hooks

Modularidad y reutilización

Los componentes permiten crear interfaces UI modulares y mantener una separación clara de responsabilidades.
Los componentes pueden anidarse y reutilizarse en diferentes partes de la aplicación

UTE EC | Xtratego ACADEMY

¿Qué son las props?:

Las props son el mecanismo mediante el cual los componentes reciben datos del componente padre. Son inmutables dentro del componente que las recibe.

```
function App() {
  return <Welcome name="Carlos" />;
}
```

Uso de Props en componentes

Se accede a las **props** en los componentes funcionales como argumentos **(props.name)** y en los de clase como **this.props.name**

```
function App() {
  return <Welcome name="Carlos" />;
}
```

Validación de Props

React proporciona la biblioteca **PropTypes** para validar las props que se pasan a los componentes y asegurar que sean del tipo correcto

```
import PropTypes from 'prop-types';

function Welcome(props) {
  return <h1>Hola, {props.name}</h1>;
}
Welcome.propTypes = {
  name: PropTypes.string.isRequired,
};
```

Ant Design (abreviado como **Antd**) es un **framework de componentes de interfaz de usuario (UI)** que se utiliza en el desarrollo de aplicaciones web con **React**

```
npm install antd
```

Ejemplo: modulo2/react-1

Desarrollar un componente que reciba una lista de nombres como props y los renderice dentro de un componente de lista.
Utilizar un componente User que reciba cada nombre individual como prop

Unidad 2

Estado y **Ciclo de Vida** de Componentes

Unidad 2.1

Estado en React (State)

Definición de estado (State)

El **estado** en React es un objeto que almacena datos dinámicos de un **componente**, permitiendo que el componente responda a cambios en la interfaz de usuario UI.

A diferencia de las **props**, que son inmutables y se pasan desde el componente padre, el estado es mutable y está gestionado internamente por el componente.

Estado en componentes de clase

En los **componentes** de **clase**, el **estado** se inicializa en el constructor y se actualiza mediante el método **setState**.

```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

  increment = () => {
    this.setState({ count: this.state.count + 1 });
  }

  render() {
    return (
      <div>
        <p>Contador: {this.state.count}</p>
        <button onClick={this.increment}>Incrementar</button>
      </div>
    );
  }
}
```

Ejemplo: modulo2/contador-antd

Estado en componentes de clase

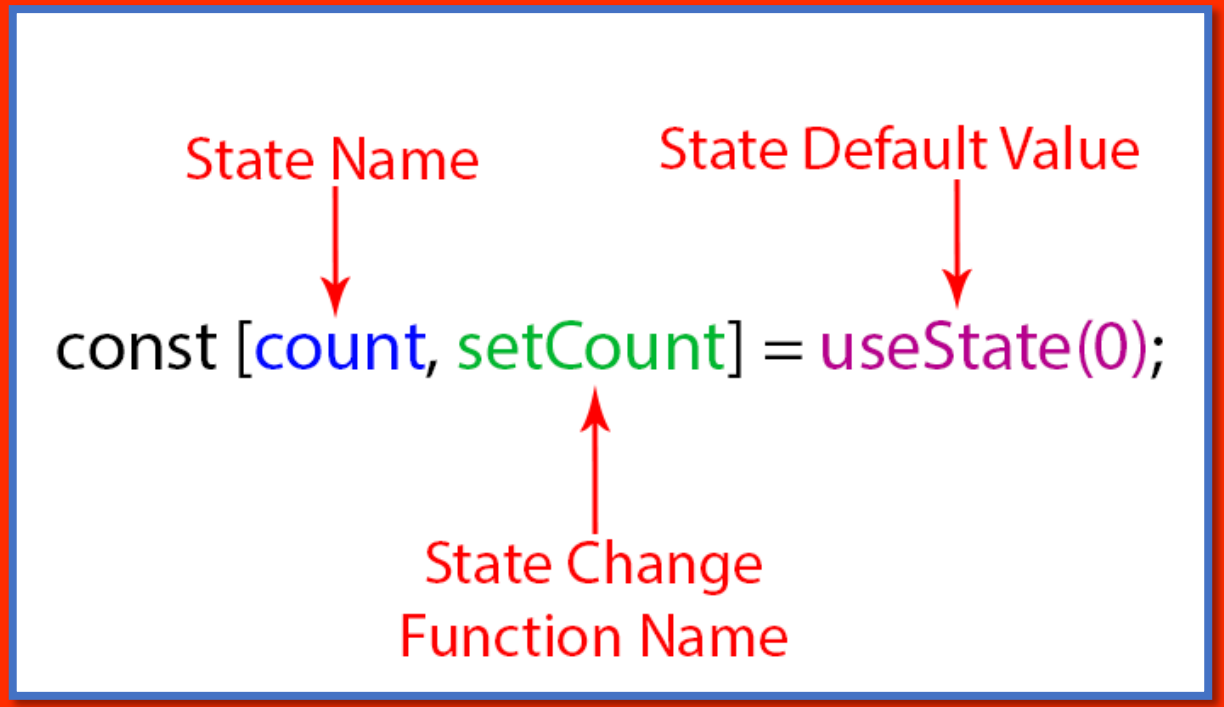
Desde la introducción de **hooks** en **React 16.8**, el estado puede manejarse en componentes funcionales utilizando el hook **useState**

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

function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Contador: {count}</p>
      <button onClick={() => setCount(count + 1)}>Incrementar</button>
    </div>
  );
}
```

Ejemplo: modulo2/contador-antd-2

Estado en componentes de clase

useState devuelve un array con dos elementos: el valor actual del estado y una función para actualizarlo



Unidad 2.2

Actualización del Estado y Renderizado

Actualización del estado

Cuando se actualiza el estado de un componente, **React** vuelve a renderizar dicho componente para reflejar los cambios en la interfaz de usuario UI.

```
this.state.count = this.state.count + 1; // Incorrecto
```

```
this.setState({ count: this.state.count + 1 }); // Correcto
```

Unidad 2.3

Ciclo de Vida de los Componentes

Concepto de ciclo de vida

El ciclo de vida de un componente en **React** abarca desde su creación hasta su destrucción.

React proporciona **métodos** para que los desarrolladores controlen diferentes fases del ciclo de vida en los componentes de clase.

Montaje (Mounting)

Ocurre cuando las props o el estado cambian

componentDidUpdate(prevProps, prevState): Se ejecuta justo después de una actualización. Ideal para realizar acciones basadas en cambios de estado o props

```
componentDidUpdate(prevProps) {
  if (this.props.id !== prevProps.id) {
    this.fetchData(this.props.id);
  }
}
```

Actualización (Updating)

Ocurre cuando el componente se inserta en el DOM

componentDidMount(): Se ejecuta inmediatamente después de que el componente ha sido montado. Ideal para realizar tareas como llamadas a APIs

```
componentDidMount() {
  fetch('https://api.example.com/data')
    .then(response => response.json())
    .then(data => this.setState({ data }));
}
```

https://jsonplaceholder.typicode.com/

Desmontaje (Unmounting)

Ocurre cuando el componente es eliminado del DOM

componentWillUnmount(): Se ejecuta justo antes de que el componente sea destruido. Útil para limpiar suscripciones o temporizadores.

```
componentWillUnmount() {
  clearInterval(this.timer);
}
```

Ejemplo: modulo2/api-antd-project

Ciclo de vida en componentes funcionales con hooks

En los componentes funcionales, el hook **useEffect** reemplaza la funcionalidad de varios métodos del ciclo de vida

```
useEffect(() => {
  document.title = `Has hecho clic ${count} veces`;
}, [count]); // Se ejecuta solo cuando cambia el valor de count
```

useEffect: Se ejecuta después de que el componente se ha renderizado

Ciclo de vida en componentes funcionales con hooks

Se puede **configurar** para que:

- Se ejecute después de cada renderizado.
- Se ejecute solo cuando cambien determinadas variables **(similar a componentDidUpdate).**
- Limpie recursos **(similar a componentWillUnmount).**

```
useEffect(() => {
  const timer = setInterval(() => {
    setCount(count + 1);
  }, 1000);

  return () => clearInterval(timer); // Limpieza similar a componentWillUnmount
}, []);
```

Unidad 2.4

Manejo de Eventos en React

Captura de eventos

React utiliza una sintaxis basada en camelCase para eventos, como **onClick, onChange, onSubmit**, etc.

Los manejadores de eventos se pasan como funciones en **JSX**

```
function Button() {
  const handleClick = () => {
    alert('¡Botón presionado!');
  };

  return <button onClick={handleClick}>Presiona aquí</button>;
}
```

Unidad 2.5

Formularios Controlados y No Controlados

Formularios no controlados

Los formularios no controlados utilizan referencias a los elementos del **DOM** directamente para manejar los datos de entrada. En este caso, **React** no gestiona el valor del campo.

```
function Form() {
  const inputRef = React.useRef();

  const handleSubmit = (event) => {
    event.preventDefault();
    alert(`El nombre ingresado es: ${inputRef.current.value}`);
  };

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Nombre:
        <input type="text" ref={inputRef} />
      </label>
      <button type="submit">Enviar</button>
    </form>
  );
}
```

Formularios controlados

En los formularios controlados, los datos de entrada del formulario están vinculados al estado del componente. Esto permite controlar el valor de los campos de formulario mediante **React**

```jsx
function Form() {
  const [name, setName] = useState('');

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

  const handleSubmit = (event) => {
    event.preventDefault();
    alert(`El nombre ingresado es: ${name}`);
  };

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Nombre:
        <input type="text" value={name} onChange={handleChange} />
      </label>
      <button type="submit">Enviar</button>
    </form>
  );
}
```

Ejemplo: modulo2/calcularedad-react

Unidad 2.6

Ejercicio Práctico

Contador interactivo

Crear un **componente** que muestre un contador con **dos botones**: uno para **incrementar** y otro para **decrementar** el valor del contador.
Añadir un mensaje que se **actualice** dinámicamente en función del valor del **contador**

```
function Counter() {
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>El valor del contador es: {count}</p>
      <button onClick={() => setCount(count + 1)}>Incrementar</button>
      <button onClick={() => setCount(count - 1)}>Decrementar</button>
    </div>
  );
}
```

Unidad 3

Gestión de Estado con Redux

Unidad 3.1

Introducción a Redux

Concepto clave de Redux

Redux es una librería de JavaScript para gestionar el estado global de una aplicación, ideal para aplicaciones con grandes volúmenes de datos o que requieren manejar el estado en múltiples componentes

UTE EC | Xtratego ACADEMY

Principios fundamentales de Redux

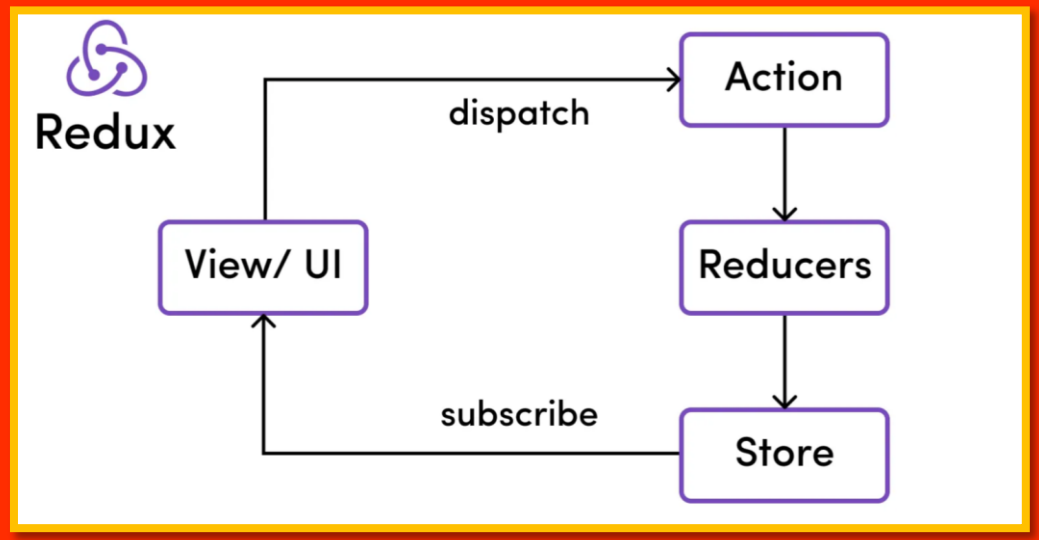
Single Source of Truth (Una sola fuente de la verdad): El estado de toda la aplicación se almacena en un único objeto, denominado **store**.

Estado inmutable: El estado no se modifica directamente, sino que se crean nuevas versiones del estado mediante **acciones** y **reducers**.

Flujo de datos unidireccional: Los datos siempre fluyen en una única dirección dentro de la aplicación.

Beneficios de Redux

Ayuda a gestionar el **estado** de manera centralizada, permitiendo que múltiples componentes compartan datos de forma consistente



Unidad 3.2

Estructura de Redux

Store

Es el único lugar donde reside el estado de la aplicación.
Se crea utilizando **createStore()**, que toma como argumento un **reducer**

```
import { createStore } from 'redux';

const initialState = { count: 0 };

function counterReducer(state = initialState, action) {
  switch (action.type) {
    case 'INCREMENT':
      return { count: state.count + 1 };
    default:
      return state;
  }
}
const store = createStore(counterReducer);
```

Actions (Acciones)

Son objetos simples que describen qué ocurrió en la aplicación.

Tienen al menos una propiedad **type**, que indica el tipo de acción, y pueden incluir datos adicionales

```
const incrementAction = {
  type: 'INCREMENT'
};
```

Reducers

Son funciones puras que reciben el estado actual y una acción, y devuelven un nuevo estado.
El **reducer** es responsable de definir cómo cambia el estado en respuesta a las acciones.

```
function counterReducer(state = initialState, action) {
  switch (action.type) {
    case 'INCREMENT':
      return { count: state.count + 1 };
    case 'DECREMENT':
      return { count: state.count - 1 };
    default:
      return state;
  }
```

Unidad 3.3

Flujo de datos en Redux

Dispatching actions

Las acciones se envían **(dispatch)** para notificar que algo ha ocurrido en la aplicación.
Se utilizan métodos como **store.dispatch()** para enviar acciones.

```
store.dispatch({ type: 'INCREMENT' });
```

Actualización del estado con reducers

Cuando una acción es enviada, el **reducer** procesa esa acción y devuelve un nuevo estado.

Redux garantiza que el flujo de datos sea predecible y que las actualizaciones ocurran de manera controlada

UTE EC | Xtratego ACADEMY

Subscription

Los **componentes** pueden **suscribirse** a la store para ser **notificados** cuando haya un cambio en el estado. Los **suscriptores** se actualizan cuando el **estado** cambia

```
store.subscribe(() => console.log(store.getState()));
```

Unidad 3.4

Integración de Redux con React

React-Redux

Para integrar Redux con React, se utiliza la biblioteca react-redux, que proporciona un conjunto de herramientas para conectar React con Redux de manera sencilla

```
npm install react-redux
```

Provider

El componente **<Provider>** es utilizado para envolver la aplicación **React**, permitiendo que los componentes accedan al store de **Redux**.

```
import { Provider } from 'react-redux';
import { store } from './store';

function App() {
  return (
    <Provider store={store}>
      <MyComponent />
    </Provider>
  );
}
```

Conexión de componentes con Redux (Hooks)

Se utilizan los **hooks** useSelector para leer el estado desde la store y useDispatch para enviar acciones.

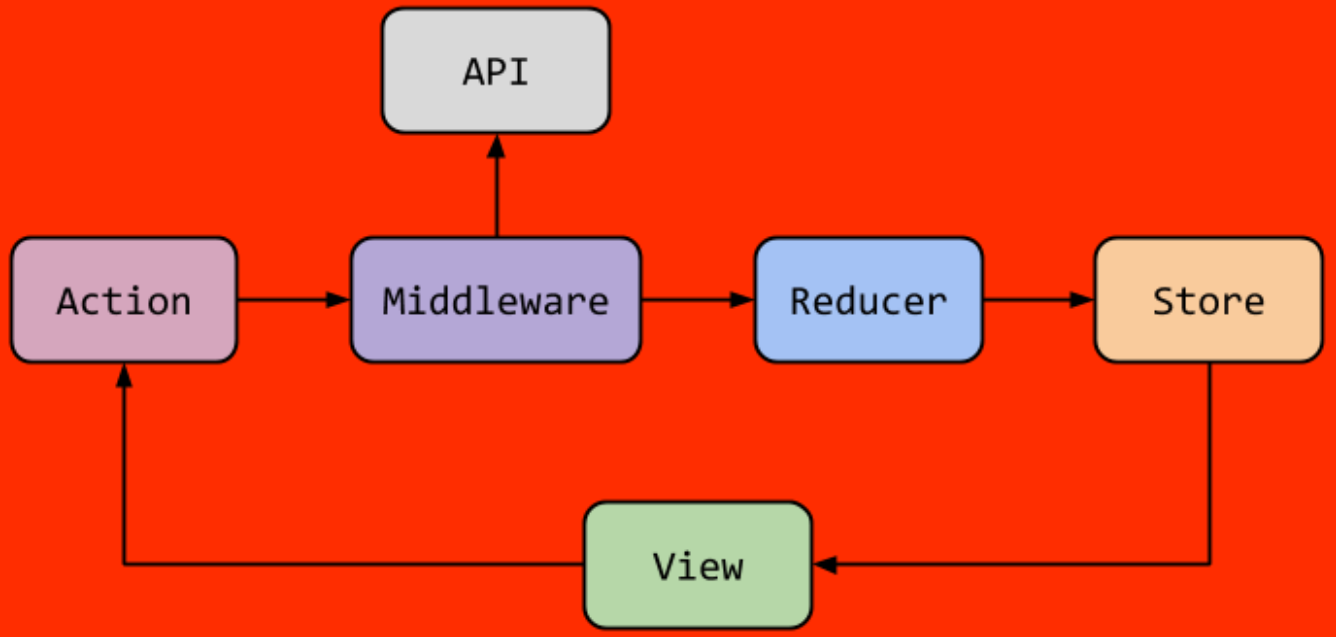
```jsx
import { useSelector, useDispatch } from 'react-redux';

function Counter() {
  const count = useSelector((state) => state.count);
  const dispatch = useDispatch();

  return (
    <div>
      <p>Contador: {count}</p>
      <button onClick={() => dispatch({ type: 'INCREMENT' })}>Incrementar</button>
      <button onClick={() => dispatch({ type: 'DECREMENT' })}>Decrementar</button>
    </div>
  );
}
```

Unidad 3.5

Middleware en Redux

Middleware

Los **middleware en Redux** interceptan las acciones antes de que lleguen al **reducer**, lo que permite realizar tareas adicionales como llamadas a **APIs** o manejar acciones **asíncronas**.



Redux Thunk

redux-thunk es un **middleware** que permite crear acciones **asíncronas**. Se utiliza para manejar **operaciones** que requieren esperar datos, como consultas a una API

```
npm install redux-thunk
```

UTE EC | Xtratego ACADEMY

Ejemplo de acción asíncrona con Thunk

```
function fetchData() {
  return function(dispatch) {
    fetch('https://api.example.com/data')
      .then(response => response.json())
      .then(data => {
        dispatch({ type: 'FETCH_SUCCESS', payload: data });
      });
  };
}
```

Unidad 3.6

Herramientas para Depuración

Redux DevTools

Redux DevTools es una extensión de navegador que permite monitorear las acciones y el estado de la aplicación en tiempo real.
Proporciona una visualización clara del flujo de datos y facilita la depuración de errores.

```
const store = createStore(
  rootReducer,
  window.__REDUX_DEVTOOLS_EXTENSION__ &&
window.__REDUX_DEVTOOLS_EXTENSION__()
);
```

Unidad 3.7

Ejercicio Práctico

Ejercicio: Contador global con Redux

Crear una aplicación que maneje el estado de un contador global utilizando **Redux**.

Utilizar **react-redux** para conectar la aplicación **React** con **Redux**.

Añadir botones para incrementar y decrementar el contador y mostrar el valor actualizado en todos los componentes que lo necesiten.

Extensión: Utilizar **Redux Thunk** para simular una operación asíncrona que actualice el **contador**.

Unidad 4

Herramientas de Desarrollo y Build (Webpack y Babel)

UTE EC | Xtratego ACADEMY

Unidad 4.1

Introducción a Webpack y Babel

Webpack

Es un **empaquetador** de módulos para aplicaciones **JavaScript modernas**. Permite combinar múltiples archivos **(JavaScript, CSS, imágenes, etc.)** en uno o más paquetes optimizados para su distribución

Babel

Es un **transpilador** de **JavaScript** que convierte código ES6+ en versiones anteriores del lenguaje para garantizar la **compatibilidad** con todos los **navegadores**

Objetivos

Optimizar el rendimiento de las aplicaciones **React**.

Hacer que el código **JavaScript** moderno sea compatible con todos los **navegadores**.

Gestionar y organizar eficientemente todos los activos del proyecto

UTE EC | Xtratego ACADEMY

Unidad 4.2

Configuración Básica de Webpack

Instalación de Webpack

```
npm install --save-dev webpack webpack-cli
```

Archivo de configuración de Webpack

```
const path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist'),
  },
  mode: 'development',
};
```

- **entry**: Define el punto de entrada de la aplicación.
- **output**: Indica dónde se generará el archivo empaquetado.
- **mode**: Establece el modo (desarrollo o producción).

Loaders

Los **loaders** transforman archivos no JavaScript **(como CSS o imágenes)** en módulos que **Webpack** puede procesar.
Ejemplo de instalación y uso de un loader para CSS:

```
npm install --save-dev style-loader css-loader
```

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.css$/i,
        use: ['style-loader', 'css-loader'],
      },
    ],
  },
};
```

Plugins

Los **plugins** amplían las capacidades de **Webpack**, como la generación de archivos **HTML** o la optimización de los archivos de salida.

Ejemplo de instalación del plugin **HtmlWebpackPlugin**

```
npm install --save-dev html-webpack-plugin
```

Configuración del plugin en webpack.config.js

```
const HtmlWebpackPlugin = require('html-webpack-plugin');
module.exports = {
  plugins: [
    new HtmlWebpackPlugin({
      template: './src/index.html',
    }),
  ],
};
```

Unidad 4.3

Optimización con Webpack

Code Splitting

Técnica para **dividir** el **código** en varios archivos más pequeños, **mejorando** el rendimiento de la aplicación al cargar solo lo necesario.

```
module.exports = {
  optimization: {
    splitChunks: {
      chunks: 'all',
    },
  },
};
```

Lazy Loading

Permite **cargar componentes** bajo demanda en lugar de **cargarlos** todos al **inicio**

```
const LazyComponent = React.lazy(() => import('./LazyComponent'));
function App() {
  return (
    <React.Suspense fallback={<div>Cargando...</div>}>
      <LazyComponent />
    </React.Suspense>
  );
}
```

Minificación y Compresión

En modo de **producción**, Webpack **minifica** y **comprime** automáticamente el código **JavaScript**.
Para habilitar esto, se utiliza el modo **production**

```
module.exports = {
  mode: 'production',
};
```

Unidad 4.4

Configuración Básica de Babel

Instalación de Babel

Se instalan **Babel** y los presets necesarios para transpirar el código moderno a **ES5**

```
npm install --save-dev @babel/core @babel/preset-env babel-loader
```

Archivo de configuración de Babel (.babelrc)

Define los **presets** y **plugins** utilizados por Babel para transpirar el código

```
{
  "presets": ["@babel/preset-env", "@babel/preset-react"]
}
```

- **@babel/preset-env**: Transpila el código moderno (ES6+) a versiones anteriores compatibles con navegadores antiguos.
- **@babel/preset-react**: Permite utilizar JSX y otras características de React

Integración de Babel con Webpack

Para que Webpack utilice **Babel** para transpirar el código, se agrega el loader de Babel en el archivo **webpack.config.js:**

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        use: {
          loader: 'babel-loader',
        },
      },
    ],
  },
};
```

Unidad 4.5

Gestión de Recursos con Webpack

Carga de Imágenes

Para manejar imágenes, se utiliza el file-loader

```
npm install --save-dev file-loader
```

Configuración en webpack.config.js

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.(png|jpg|gif)$/i,
        use: [
          {
            loader: 'file-loader',
            options: {
              name: '[path][name].[ext]',
            },
          },
        ],
      },
    },
  },
};
```

Carga de Fuentes

Para manejar fuentes, se utiliza también file-loader o url-loader

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.(woff|woff2|eot|ttf|otf)$/i,
        type: 'asset/resource',
      },
    ],
  },
};
```

UTE EC | Xtratego ACADEMY

Unidad 4.6

Entornos de Desarrollo y Producción

Separación de configuraciones

Es buena práctica tener configuraciones separadas para desarrollo y producción, ya que en desarrollo se prioriza la facilidad de depuración y en producción la optimización.

Creación de archivos separados:

- **webpack.dev.js (desarrollo)**
- **webpack.prod.js (producción)**

Ejemplo de configuración para desarrollo (webpack.dev.js):

```
module.exports = {
  mode: 'development',
  devtool: 'inline-source-map',
  devServer: {
    contentBase: './dist',
  },
};
```

```
module.exports = {
  mode: 'production',
  optimization: {
    minimize: true,
  },
};
```

Unidad 4.7

Herramientas de Depuración

UTE EC | Xtratego ACADEMY

Source Maps

Los **source maps** facilitan la depuración al mapear el código empaquetado al código fuente original.
Para habilitar source maps en **Webpack**

```
module.exports = {
  devtool: 'source-map',
};
```

Webpack DevServer

Webpack DevServer proporciona un entorno de desarrollo rápido con recarga automática **(hot reloading).**

```
npm install --save-dev webpack-dev-server
```

Configuración básica en webpack.config.js:

```
module.exports = {
  devServer: {
    contentBase: './dist',
    hot: true,
  },
};
```

Modulo 2

Práctica de modulo

Practica

Descripción de la Práctica
Objetivo:
Desarrollar una aplicación de lista de tareas (To-Do List) utilizando React. El proyecto debe implementar conceptos clave del curso, como componentes, props, manejo de estado, ciclo de vida de componentes, y el uso de Redux para gestionar el estado global de la aplicación.
Instrucciones de la Práctica:
Configuración del Proyecto:
Utiliza create-react-app para generar el entorno inicial del proyecto.
Configura Webpack y Babel para empaquetar y transpilar el código.
Asegúrate de utilizar ESLint y Prettier para mantener la calidad del código.
Componentes:
Crea al menos tres componentes:
TodoList: Encargado de mostrar la lista de tareas.
TodoItem: Representa una tarea individual.
AddTodo: Un formulario para añadir nuevas tareas a la lista.
Props y Estado:
Usa props para pasar los datos desde TodoList a TodoItem.
Utiliza useState para gestionar el estado local de la lista de tareas y el formulario de ingreso de tareas en AddTodo.
Ciclo de Vida de Componentes:
Usa el hook useEffect para guardar las tareas en el almacenamiento local del navegador cada vez que la lista cambie.
Al cargar la aplicación, usa useEffect para cargar las tareas guardadas desde el almacenamiento local.
Manejo de Eventos:
Implementa el evento onSubmit para añadir una nueva tarea en el formulario de AddTodo.
Implementa un botón para eliminar tareas individuales, utilizando el evento onClick.
Uso de Redux:
Configura Redux en el proyecto para gestionar el estado global de las tareas.
Implementa actions y reducers para añadir y eliminar tareas en el estado global.
Usa useSelector y useDispatch para conectar los componentes a Redux.
Optimización:
Utiliza técnicas de optimización como code splitting y lazy loading para mejorar el rendimiento de la aplicación.

Rubrica

Criterio	Excelente (5 puntos)	Bueno (4 puntos)	Regular (3 Configuración del Proyecto puntos)	Insuficiente (1-2 puntos)
Configuración del Proyecto	Configuración completa y correcta de create-react-app, Webpack, Babel, ESLint, Prettier.	Faltan algunos detalles menores en la configuración del proyecto.	La configuración es funcional pero con errores que afectan el rendimiento.	El proyecto no se configura correctamente o presenta problemas graves de configuración.
Componentes	Todos los componentes funcionan correctamente, reutilizables y bien organizados.	Los componentes funcionan, pero faltan detalles en la organización o reusabilidad.	Los componentes funcionan parcialmente, pero con errores en su funcionamiento.	Los componentes no funcionan correctamente o no están bien estructurados.
Props y Estado	Uso correcto de props y estado con useState, manejo dinámico de datos.	Uso adecuado de props y estado, aunque hay detalles de optimización.	El manejo de props o estado tiene errores leves que afectan la dinámica de la aplicación.	Faltan props o el manejo de estado es incorrecto, afectando el funcionamiento.
Ciclo de Vida	Uso óptimo de useEffect para guardar y recuperar datos, manejo adecuado del ciclo de vida.	Uso correcto de useEffect, pero con algunas ineficiencias.	Se implementa useEffect, pero no gestiona bien los datos.	No se usa el ciclo de vida correctamente o no se implementa.
Manejo de Eventos	Todos los eventos se implementan correctamente y la interacción es fluida	Los eventos funcionan, pero con pequeños problemas en la interacción	Los eventos presentan errores que afectan la funcionalidad de la app.	No se implementan correctamente o los eventos no funcionan.
Uso de Redux	Configuración completa de Redux, con manejo eficiente del estado global.	Redux se configura correctamente, pero hay pequeños errores en el flujo de datos.	Redux está configurado, pero con problemas que afectan el manejo del estado global.	No se configura Redux o su implementación es incorrecta.
Optimización	Uso completo de técnicas de optimización como code splitting y lazy loading.	Se usan algunas técnicas de optimización, pero faltan mejoras.	La optimización es limitada y afecta el rendimiento de la aplicación.	No se implementan técnicas de optimización.

GRACIAS