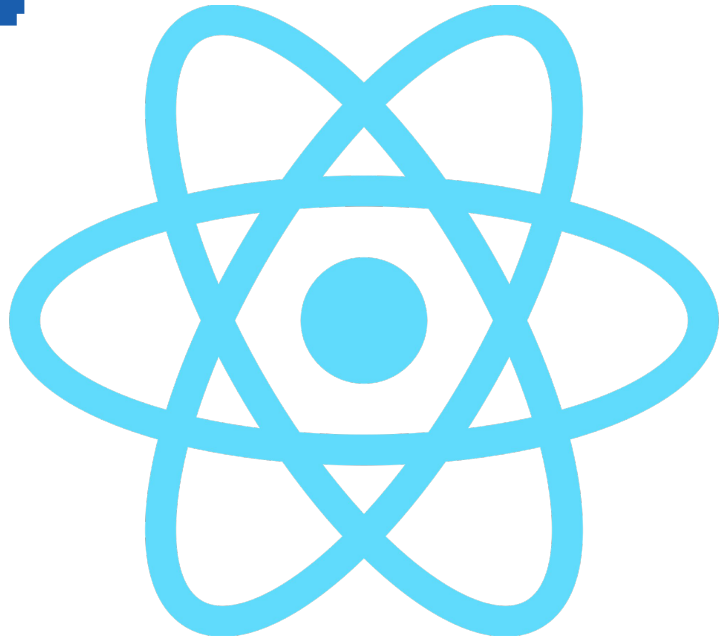
Clase 4

REACT



Temario

1. React
2. React Components and Props
3. Composing React Components and Stateful React Components
4. **Events, conditional rendering**
5. Lists and forms
6. React Router I
7. React Router II
8. Debugging and Testing

Eventos

La gestión de eventos en React es similar a cómo se manejan en elementos DOM.

Con eventos nos referimos a las acciones a ejecutar cuando algo sucede, algunos ejemplos de eventos son:

- onclick: acción de hacer un click sobre un elemento, por ejemplo un botón.
- onchange: acción que se debe ejecutar cuando un elemento cambia, por ejemplo un input.
- onblur: cuando el foco se posiciona en un elemento, por ejemplo al clicar en un input el aspecto puede cambiar
- onkeypress: cuando el usuario pulsa una tecla
- onkeydown: cuando el usuario está presionando una tecla
- onkeyup: cuando el usuario acaba de pulsar una tecla
- onmouseover: cuando el ratón pasa por encima de un elemento

Podéis consultar todos los eventos aquí: https://www.w3schools.com/tags/ref_eventattributes.asp

Eventos

Algunas de las diferencias de sintaxis son:

- Los eventos de React se nombran usando camelCase, en vez de minúsculas. **onclick => onClick**
- Con JSX pasas una función como el manejador del evento en vez de un string. **"activateLasers()" => {activateLasers}**
- En React, generalmente no necesitas llamar a ***addEventListener*** para agregar un manejador del evento del DOM, en cambio, debes proveer un manejador de eventos cuando el elemento se renderiza.
- En React no se puede retornar ***False*** para prevenir el comportamiento por defecto, se debe llamar explícitamente a ***preventDefault.*** Por ejemplo en un HTML sin React, para prevenir el comportamiento por defecto de enviar un formulario, puedes hacer así:

```
<form onsubmit="console.log('You clicked submit.'); return false">
  <button type="submit">Submit</button>
</form>
```

Sin embargo con React tienes que hacerlo así =>

```
<button onclick="activateLasers()">
  Activate Lasers
</button>
```

```
<button onClick={activateLasers}>
  Activate Lasers
</button>
```

```
function Form() {
  function handleSubmit(e) {
    e.preventDefault();
    console.log('You clicked submit.');
  }

  return (
    <form onSubmit={handleSubmit}>
      <button type="submit">Submit</button>
    </form>
  );
}
```

Eventos

Al usar componentes de clase, un patrón muy común es que los manejadores de eventos sean un método de la clase.

Los métodos de clase no están ligados por defecto a la clase, hay que hacerlo en el constructor a través del método ***bind()***. Si no se liga y se pasa el método ***this.handleClick*** como manejador en un evento, ***this*** será ***undefined*** cuando se llame al método. Este funcionamiento no es específico de React, es como operan las funciones Javascript.

Existen 2 maneras de evitar el uso de ***bind():***

1. Declarando un campo público de clase:

   campoPublico = () => {}

2. Si el manejador es una función normal, se debe llamar en el evento a una función flecha que llame () al manejador:

   onClick={ () => this.metodoDeClase() }

   Este método no es recomendable.

```
class EventExample extends React.Component {
  constructor(props) {
    super(props);
    // Este enlace es necesario para hacer que `this` funcione en el callback
    this.manejadorClick = this.manejadorClick.bind(this);
  }
  manejadorClick() {
    // Manejador ligado en el constructor
    console.log("this is:", this);
  }
  campoPublico = () => {
    // Campo público a usar como manejador
    console.log("this is:", this);
  };
  metodoDeClase() {
    // Método de clase
    console.log("this is:", this);
  }
  render() {
    return (
      <>
        <button onClick={this.manejadorClick}>
            Manejador ligado con bind
        </button>
        <button onClick={this.campoPublico}>
            Manejador con campo público
        </button>
        <button onClick={() => this.metodoDeClase()}>
            Manejador con método de clase
        </button>
      </>
    );
  }
}
```

Eventos

En muchas ocasiones es necesario pasar argumentos a los manejadores de eventos, argumentos como el id de un usuario o una fila, cualquier dato que sea requerido por el manejador.

Método 1: llamar al manejador dentro de una función de flecha. Si además queremos recibir el evento, añadimos el argumento "**e**" a la función flecha y la enviamos a nuestro manejador. Ejemplo:

```
class EventExample extends React.Component {
  deleteRow = (id, e) =>{
    console.log(e, id);
  };
  render() {
    const id = 10;
    return (
        <button onClick={(e) => this.deleteRow(id, e)}>
            Eliminar con función flecha
        </button>
    );
  }
}
```

Método 2: haciendo el bind en la creación del prop. Es necesario pasarle el objeto "**this**" y luego los argumentos necesarios. En este método el evento "**e**" se pasa siempre como último argumento de nuestra función.

```
class EventExample extends React.Component {
  deleteRow = (id, e) =>{
    console.log(e, id);
  };
  render() {
    const id = 10;
    return (
        <button onClick={this.deleteRow.bind(this, id)}>
            Eliminar con bind
        </button>
    );
  }
}
```

Renderizado condicional

En React, puedes crear distintos componentes que encapsulan el comportamiento que necesitas. Entonces, puedes renderizar solamente algunos de ellos, dependiendo del estado de tu aplicación.

El renderizado condicional funciona de la misma manera que lo hacen las condiciones en Javascript. Podéis usar **if**, el operador condicional ternario: condition **?** exprIfTrue **:** exprIfFalse o el operador **&&**.

Imaginemos que queremos crear un componente **Greeting** que reciba una propiedad **isLoggedIn** para indicarnos si el usuario ha iniciado sesión o no. Si lo ha hecho (isLoggedIn=true), queremos devolver el saludo "Bienvenido de nuevo!", en otro caso, queremos mostrar el mensaje: "Por favor, haz login!". **Ejemplo**:

- Creamos una app my-tenth-app
- Creamos un componente **UserGreeting** con el saludo
- Creamos un componente **SignUp** con el segundo mensaje
- Creamos el componente **Greeting** que según el valor del prop **isLoggedIn** muestre un componente u otro
- Renderizamos el componente Greeting dentro del componente App y jugamos con el prop **isLoggedIn**

```
// src/Greeting.js
const UserGreeting = () => {
  return <h1>Bienvenido de nuevo!</h1>;
};

const SignUp = () => {
  return <h1>Por favor, haz login!</h1>;
};

const Greeting = ({ isLoggedIn }) => {
  if (isLoggedIn) {
    return <UserGreeting />;
  } else {
    return <SignUp />;
  }
};

export default Greeting;
```

```
import './App.css';
import Greeting from './Greeting';

function App() {
  return (
    <div className="App">
      {/* cambiar el valor a true */}
      <Greeting isLoggedIn={false} />
    </div>
  );
}
export default App;
```

Renderizado condicional: if

Sobre la misma aplicación my-tenth-app, realizar las siguientes acciones:

- Creamos un componente **LoginButton** que reciba como prop **onClick** y devuelva un botón que ponga **Login**
- Creamos un componente **LogoutButton** que reciba como prop **onClick** y devuelva un botón que ponga **Logout**
- Creamos un componente **LoginControl** con una variable de estado **isLoggedIn** inicializada a false
- En el componente LoginControl añadimos el método **handleLoginClick** que cambie el estado isLoggedIn a **true**
- En el componente LoginControl añadimos el método **handleLogoutClick** que cambie el estado isLoggedIn a **false**
- En el método render de **LoginControl**, renderizamos el componente **Greeting** y le pasamos como prop **isLoggedIn** la variable de estado isLoggedIn
- Debajo del componente **Greeting** renderizaremos el botón **LoginButton** si la variable de estado isLoggedIn es **false,** en otro caso, mostramos el botón **LogoutButton**
- En el componente App eliminamos el componente **Greeting** y mostramos el componente **LoginControl**

```
const LoginButton = ({ onClick }) => {
  return <button onClick={onClick}>Login</button>;
};

export default LoginButton;
```

```
const LogoutButton = ({ onClick }) => {
  return <button onClick={onClick}>Logout</button>;
};

export default LogoutButton;
```

```
import './App.css';
import LoginControl from './LoginControl';

function App() {
  return (
    <div className="App">
      <LoginControl />
    </div>
  );
}
export default App;
```

Renderizado condicional: if

El componente LoginControl tiene las siguientes características:

- Tiene un método constructor donde bindea los manejadores de los botones para poder asignarlos a los eventos onClick de los botones
- El state es inicializado en el constructor
- Los manejadores cambian el estado con setState
- En la función render asigno en la variable button el botón correspondiente según la condición isLoggedIn
- Los componentes Greeting, LoginButton y LogoutButton han sido creados en archivos independientes e importados en este archivo.

```
import React from "react";
import Greeting from "./Greeting";
import LoginButton from "./LoginButton";
import LogoutButton from "./LogoutButton";

class LoginControl extends React.Component {
  constructor(props) {
    super(props);
    this.handleLoginClick = this.handleLoginClick.bind(this);
    this.handleLogoutClick = this.handleLogoutClick.bind(this);
    this.state = { isLoggedIn: false };
  }
  handleLoginClick() {
    this.setState({ isLoggedIn: true });
  }
  handleLogoutClick() {
    this.setState({ isLoggedIn: false });
  }
  render() {
    const isLoggedIn = this.state.isLoggedIn;
    let button;
    if (isLoggedIn) {
      button = <LogoutButton onClick={this.handleLogoutClick} />;
    } else {
      button = <LoginButton onClick={this.handleLoginClick} />;
    }
    return (
      <div>
        <Greeting isLoggedIn={isLoggedIn} />
        {button}
      </div>
    );
  }
}

export default LoginControl;
```

Renderizado condicional: &&

El efecto condicional se puede conseguir con el operador &&.

En Javascript **expr1 && expr2** siempre evalúa a **expr2 si expr1** es **true**. Mientras que **expr1 && expr2** siempre evalúa a **expr1** si **expr1** es **false.**

Si usamos una expresión así dentro de JSX, si es **true** el elemento detrás de **&&** se mostrará en el resultado, y si es **false**, React lo omitirá.

Por otro lado, dentro del código JSX se pueden incluir expresiones envolviéndolas con llaves. Luego, podemos usar la expresión && dentro de JSX de la siguiente manera.

Hay que tener en cuenta que si expr1 es un 0, en Javascript se considera una expresión falsa, en ese caso React omitirá el componente pero devolverá **0.** Ejemplo:

Para evitar ese comportamiento lo mejor sería poner:

```
return <div>{count > 0 && <h1>Messages: {count}</h1>} </div>;
```

```
function App() {
  const unreadMessages = [];
  return (
    <div>
      <h1>Hello!</h1>
      {unreadMessages.length >0 &&
        <h2>
          You have {unreadMessages.length} unread messages.
        </h2>
      }
    </div>
  )
}
```

```
function App() {
  const count = 0;
  return <div>{count && <h1>Messages: {count}</h1>} </div>;
}
```

Renderizado condicional: ternario

En React podemos utilizar el operador ternario **condition ? expr1 : expr2** para hacer un renderizado condicional.

```
function App(props) {
  const isLoggedIn = props.isLoggedIn;
  return (
    <div>
      The user is <b>{isLoggedIn ? 'currently' : 'not'}</b> logged in.
    </div>
  );
}
```

Este operador lo podemos usar también para renderizar componentes. En nuestro ejemplo de LoginControl, el renderizado de los botones también podría realizarse así:

```
return (
  <div>
    <Warning warn={!isLoggedIn} />
    <Greeting isLoggedIn={isLoggedIn} />
    {isLoggedIn
        ? <LogoutButton onClick={this.handleLogoutClick} />
        : <LoginButton onClick={this.handleLoginClick} />
    }
  </div>
);
```

Renderizado condicional: No renderizar

Es posible que en alguna de nuestras aplicaciones necesitemos que según un valor, ya sea de props o de state, un componente no se muestre. La manera de hacerlo es retornando un **null.**

- En nuestra app my-tenth-app creamos un nuevo componente **Warning** que espere recibir una prop **warn**, si no la recibe no tiene que mostrarse, pero si recibe la prop debe renderizar un div con clase “warning” y el texto “Warning!!”
- Añadir un estilo en App.css para “warning” con color: orange;
- Importar el componente **Warning** en el componente **LoginControl.** En el renderizado del componente añadir el componente **Warning** con el valor de prop warn={!isLoggedIn}
- Comprobar lo que ocurre

```
// src/Warning.js
const Warning = (props) => {
  if (!props.warn) {
    return null;
  }
  return <div className="warning">Warning!</div>;
};

export default Warning;
```

```
// src/LoginControl.js
return (
  <div>
    <Warning warn={!isLoggedIn} />
    <Greeting isLoggedIn={isLoggedIn} />
    {button}
  </div>
);
```

Ejercicios

1. Replicar el código de clase:
   - App my-ninth-app con el componente **EventExample** que se renderice en dentro del componente App. En el componente EventExample deben estar todos los manejadores, incluido el manejador **deleteRow** con argumentos.
   - App my-tenth-app con todos los componentes, incluido el componente Warning.
2. Leer el artículo sobre el anti-patrón de usar index como key
3. Probar el ejemplo del anti-patrón
4. Leer la documentación de los Componentes no controlados y crear los componentes **NameForm** y **FileInput** y renderizarlos en el componente App.js

ALBAÑILESDIGITALES

Developing your new career

servicio navarro de empleo

das·nano

VeriDas