VISTAS

VISTAS

- **Que aprenderemos en este apartado?**

1. **Renderizado Condicional**
2. **Listas, Claves, Formularios**
3. **Todo Application**

RENDERIZADO CONDICIONAL

El renderizado condicional en React funciona de la misma forma que lo hacen las condiciones en JavaScript.

```
function Greeting(props) {
  const isLoggedIn = props.isLoggedIn;
  if (isLoggedIn) {
    return <UserGreeting />;
  }
  return <GuestGreeting />;
}

ReactDOM.render(
  // Intentar cambiando isLoggedIn={true}:
  <Greeting isLoggedIn={false} />,
  document.getElementById('root')
);
```

RENDERIZADO CONDICIONAL

Puedes usar variables para almacenar elementos. Esto puede ayudarte para renderizar condicionalmente una parte del componente mientras el resto del resultado no cambia.

```
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>
  );
}
```

RENDERIZADO CONDICIONAL

Puedes incluir **expresiones en JSX** envolviéndolas en llaves. Esto incluye el operador lógico && de JavaScript.

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

RENDERIZADO CONDICIONAL

Esto funciona porque en JavaScript, true && expresión siempre evalúa a expresión, y false && expresión siempre evalúa a false.

Por eso, si la condición es true, el elemento justo después de && aparecerá en el resultado.

Si es false, React lo ignorará.

RENDERIZADO CONDICIONAL

Otro método para el renderizado condicional de elementos en una línea es usar el operador condicional **condición ? true : false** de JavaScript.

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

```
 render() {
   const isLoggedIn = this.state.isLoggedIn;
   return (
     <div>
       {isLoggedIn
         ? <LogoutButton onClick={this.handleLogoutClick} />
         : <LoginButton onClick={this.handleLoginClick} />
       }
     </div>
   );
 }
```

LISTAS

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

class App extends Component {
  constructor(props) {
    super(props);

    this.state = {
      tasks: ['task1', 'task2', 'task3', 'task4', 'task5', 'task6'],
    };
  }

  render() {
    const { tasks } = this.state;

    return (
      <ul>
        {tasks}
      </ul>
    );
  }
}

export default App;
```

LISTAS

map() recibe una función callback como argumento, esta función es invocada una vez sobre cada elemento del arreglo, cambiando cada uno de ellos. Y después de iterar por todo el arreglo, **retorna un nuevo arreglo**, con los nuevos elementos.

```
return (
  <ul>
    {tasks.map(task => <li>{task}</li>)}
  </ul>
);
```

KEY

Ahora, debemos tener en cuenta que React debe reconocer, para su correcto funcionamiento, cuales son los elementos que cambian para poder agregar, modificar o eliminar elementos dentro de nuestra lista.

Para facilitarle esto a React, debemos pasarle una propiedad extra a nuestro elemento de la lista. Esta propiedad es key, y le permite identificar y diferenciar cada uno de los elementos de una lista, por lo que este debe ser único. Se acostumbra usar el id del elemento, aunque en este caso, por facilidad usaremos el mismo nombre de la tarea.

KEY

```
return (
  <ul>
    {tasks.map(task => {
      return <li key={`task_${task}`}>{task}</li>
    )}
  </ul>
);
```

KEY

No se recomienda usar el indice de los elementos dentro del arreglo, ya que estos pueden cambiar de orden, se pueden agregar o incluso se pueden haber eliminado uno o varios elementos de nuestra lista.

FORMULARIOS

- Los elementos de formulario HTML funcionan de forma un poco diferente del resto de elementos DOM en React, porque los elementos de formulario ya mantienen un estado interno.
- El comportamiento predeterminado de los formularios de HTML es navegar a una nueva página cuando el usuario envía el formulario. Si este es comportamiento deseado no es necesario hacer nada en React.
- En la mayoría de los casos, es conveniente tener una función de JavaScript que envíe los datos del formulario sin enviar el formulario, lo que desencadenaría navegar a una nueva página.
- La función debe tener acceso a los datos que el usuario introdujo en el formulario para enviarla al servidor vía Ajax o WebSocket.
- React permite utilizar dos técnicas diferente:
  - componentes controlados
  - componentes no controlados.

COMPONENTES CONTROLADOS

- En HTML, los elementos del formulario <input>, <textarea> y <select> mantienen su propio estado y lo actualizan en función de las entradas del usuario. En React, el estado mutable se mantiene en la propiedad state de los componentes y solo se actualiza con setState().
- Para mantener sincronizados ambos estados, el componente React que representa un formulario también debe controlar lo que sucede en ese formulario después de la entrada del usuario mediante el uso de eventos.
- Un elemento de entrada de formulario cuyo valor es controlado por React de esta manera se denomina "componente controlado".
- La propiedad value de <input> permite establecer el valor inicial y recuperar el valor actual. React a incorporado la pseudo-propiedad value a <textarea> y <select>, de tal forma que los tres tengan el mismo comportamiento.
- El evento onChange se dispara con cada pulsación de tecla o selección en el control. El controlador del evento recibe un argumento SyntheticEvent que, a través de su propiedad target, permite acceder al HTMLElement que disparo el evento.

INICIALIZACIÓN

- Inicialización del estado:

```
constructor(props) {
  super(props);
  this.state = {
    nombre: props.nombre,
    // ...
  };
  this.handleChange = this.handleChange.bind(this);
}
```

- Para que sea un "componente controlado" se debe asignar al atributo value el state apropiado e interceptar el onChange.

```
render() {
  // ...
  <input type="text" name="nombre" value={this.state.nombre} onChange={this.handleChange} />
  // ...
}
```

- Es conveniente que el valor del atributo name coincida con el nombre de la propiedad en el state para facilitar el tratamiento posterior.

TRATAMIENTO DE CAMBIOS

- El tratamiento del onChange debe incluir el paso del value al state.

```
handleChange(event) {
   this.setState(nombre: event.target.value});
}
```

- O de una forma genérica que atiende a múltiples controles:

```
handleChange(event) {
   this.setState({[event.target.name]: event.target.value});
}
```

- El tratamiento puede incluir transformaciones y validaciones pero si no se pasa el value al state se anulan los cambios.

```
this.setState({nombre: event.target.value.toUpperCase()});
```

VALIDACIONES

- HTML5 ya cuenta con validadores, pero cuando la validación sea personalizado o el navegador no soporte las validaciones HTML5 se incluirán en la captura de datos y se pasaran al estado para poder mostrar al usuario los errores:

```
let errors = {};
if  (event.target.name === 'nombre') {
   if (event.target.value && event.target.value !== event.target.value.toUpperCase()) {
      errors.nombre = 'Debe ir en mayúsculas.';
   } else
      errors.nombre = null;
}
this.setState({ msgErr: errors, invalid: invalid });
```

- Cuando se mezcla la validación HTML5 y la personalizada hay que capturar los mensajes HTML para poder implementar una experiencia de usuario coherente y no mostrarlos de formas diferentes en función a la detección:

```
errors[event.target.name] = event.target.validationMessage;
if (!errors.nombre && event.target.name === 'nombre') {
   // ...
}
```

MOSTRAR ERRORES

- Los errores se pueden acumular dentro del estado en un objeto con los pares nombre/valor donde:
  - Nombre: name del control
  - Valor: cadena con el mensaje de error

  this.state = { ... msgErr: {} ... };
- Es recomendable crear un componente que muestre los mensaje de validación:

```
class ValidationMessage extends React.Component {
  render() {
    if (this.props.msg) {
      return <span className="errorMsg">{this.props.msg}</span>;
    }
    return null;
  }
}
```

- Para posteriormente asociar los mensajes a los controles:

```
Nombre: <input type="text" id="nombre" name="nombre" value={this.state.elemento.nombre}
              required minLength="2" maxLength="10" onChange={this.handleChange} />
        <ValidationMessage msg={this.state.msgErr.nombre} />
```

ENVIAR EL FORMULARIO

- Si los datos del formulario no se van a enviar directamente al servidor como respuesta del formulario, es necesario interceptar y detener el evento submit con preventDefault.

```
sendClick(e) {
   e.preventDefault();
   // ...
}

<form name="miForm" onSubmit={this.sendClick}>
   // ...
   <input type="submit" value="Enviar" />
```

- O no hacer submit:

```
<button onClick={this.sendClick} >Enviar</button>
```

- Se puede deshabilitar el botón de envío si no pasa la validación, manteniendo en el estado la validez del formulario:

```
<button disabled={this.state.invalid} onClick={this.sendClick} >Enviar</button>
```

CONTROL DE ERRORES

```
handleChange(event) {
    const cmp = event.target.name;
    const valor = event.target.value;
    const errMsg = event.target.validationMessage;
    this.setState((prev, props) => {
      const ele = prev.elemento;
      let errors = prev.msgErr;
      let invalid = false;
      ele[cmp] = valor;
      errors[cmp] = errMsg;
      if (!errors.nombre && cmp === 'nombre') {
        if (valor && valor !== valor.toUpperCase()) {
          errors.nombre = 'Debe ir en mayúsculas.';
        } else
          errors.nombre = '';
      }
      for (var c in errors) {
        invalid = invalid || invalid = invalid || (errors[c] !== '' && errors[c] !== null && typeof(errors[c]) !==
"undefined");
      }
      return { elemento: ele, msgErr: errors, invalid: invalid }
    })
  }
```

COMPONENTES NO CONTROLADOS

- En la mayoría de los casos, es recomendable utilizar componentes controlados para implementar formularios. La alternativa son los componentes no controlados, donde el DOM maneja los datos del formulario.
- Dado que un componente no controlado mantiene la estado en el DOM, a veces es más fácil integrar el código React y no React cuando se usan componentes no controlados.
- Para escribir un componente no controlado, en lugar de escribir un controlador de eventos para cada actualización de estado se pueden usar un campos referencia para obtener del DOM los valores de formulario.
- En el ciclo de vida del render de React, el atributo value de los elementos del formulario anula el valor en el DOM. Con un componente no controlado, a menudo solo desea que React especifique el valor inicial, pero deje las actualizaciones posteriores sin control. Para manejar este caso, se debe especificar el atributo defaultValue en lugar de value.

```
<input type="text" ref={(tag) => this.txtNombre = tag}
      defaultValue={this.props.nombre} />
```