Coding Bootcamp

Sprint 4

Temario

Temario

- Conditional Rendering.
- List Rendering.
- Props son *read-only*.
- State.
- Event listeners.
- Formularios.

Conditional Rendering

Conditional Rendering (1/5)

Conditional Rendering hace referencia a la acción de renderizar (o no) un componente según el estado de la aplicación.

El Conditional Rendering en React funciona de la misma forma que lo hacen las condiciones en JavaScript. 💪

Por lo tanto, se puede usar `if/else`, `&&` y/o el operador ternario. No es necesario aprender una nueva sintaxis (como puede ser el caso de Vue.js o Angular).

👉 Ver los siguientes ejemplos.

Conditional Rendering (2/5) – Ej. con variable aux. + `if/else`

```
function ConditionalRendering({ estaLogueado }) {
  let body = null;
  if (estaLogueado) {
    body = <App />;
  } else {
    body = <ComponenteLogin />;
  }
  return <div>{body}</div>;
}
```

Se creó una variable auxiliar para almacenar el componente a mostrar. Luego se hizo un `if/else` para determinar qué componente contendrá dicha variable.

Conditional Rendering (3/5) – Ej. con operador lógico &&

```jsx
function Correo({ mensajesSinLeer }) {
  return (
    <div>
      <h1>¡Hola alumno!</h1>
      {mensajesSinLeer.length > 0 && (
        <h2>Tiene {mensajesSinLeer.length} mensajes sin leer.</h2>
      )}
    </div>
  );
}
```

Si la expresión a la izquierda del **&&** evalúa a `true`, se evalúa la expresión de la derecha.

Conditional Rendering (4/5) – Ej. con operador ternario

```
function ConditionalRendering ({ estaLogueado }) {
  return (
    <div>
      El usuario {estaLogueado ? "está logueado" : "no está logueado"}
    </div>
  );
}
```

Conditional Rendering (5/5) – Ej. con operador ternario

```
function ConditionalRendering ({ estaLogueado }) {
  return (
    <div>
      El usuario {estaLogueado ? <LogoutButton /> : <LoginButton />}
    </div>
  );
}
```

List Rendering

List Rendering – `map` (1/5)

En varios frameworks existe alguna sintaxis especial para crear listas de elementos en la UI. Por ejemplo, en Angular se usa `ng-repeat` y en Vue.js se usa `v-for`.

Sin embargo, en React se utiliza simplemente JavaScript “puro”.

Se usa el método `map` (que tienen todos los arrays). El mismo retorna un nuevo array con los elementos del array original habiéndose aplicado una función sobre cada uno de ellos.

```
const numbers = [1, 2, 3];

const numbersPlusTen = numbers.map(num => num + 10);

console.log(numbersPlusTen); // [11, 12, 13]
```

List Rendering – `map` (2/5)

Ejemplo:

```
function FriendsList({ friends }) {
  return (
    <div>
      <h3>Amigos:</h3>
      <ul>
        {friends.map((friend) => {
          return <li>{friend}</li>;
        })}
      </ul>
    </div>
  );
};
```

List Rendering – `map` (3/5)

Utilizando el componente anterior de la siguiente forma:

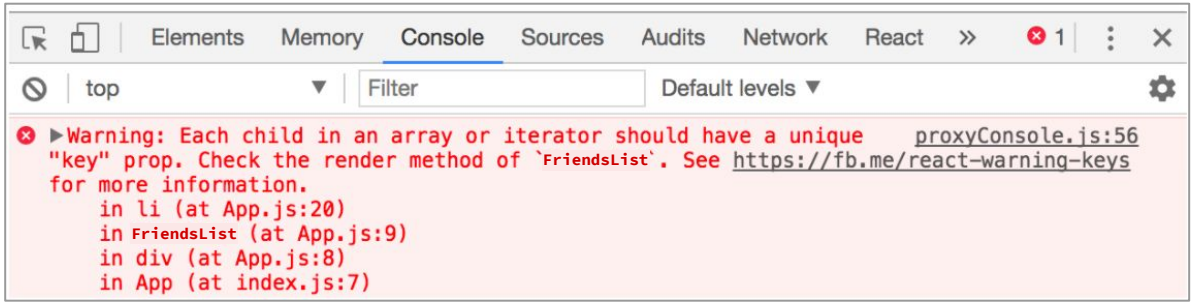
```
<FriendsList names={["María", "Juan", "Pedro"]} />
```

En la página se verá el siguiente resultado:

Amigos:

- María
- Juan
- Pedro

Y en la consola se verá:



List Rendering – `key` (4/5)

Warning: Each child in an array or iterator should have a unique "key" prop.

- Este es un *warning* que se verá siempre que un componente de React renderice una lista de elementos y no se le asigne una `key` propia a cada uno de ellos. Esto es un requerimiento de React para poder ser más eficiente a la hora de actualizar la UI.
- React utiliza esta `key` para identificar exactamente qué elemento fue modificado, eliminado o agregado en una lista de elementos, sin necesidad de volver a renderizar toda la lista.
- Esta `key` debe ser única. Si se cuenta con un `id` claramente es la mejor opción.

List Rendering – `filter` (5/5)

Otro método de los arrays que suele ser útil en React es `filter`. Funciona de forma muy similar a `map` pero, en vez de aplicar una función sobre cada elemento, permite filtrar ciertos elementos que no cumplan la condición dada. En el siguiente ejemplo se obtienen sólo los nombres que empiecen con E:

```
const friends = ["Emiliano", "María", "Pablo", "Ernesto", "Victoria"];
const newFriends = friends.filter((name) => {
   return name[0] === "E"
});

console.log(newFriends) // Retorna: ["Emiliano", "Ernesto"]
```

Props son *read-only*

Props son *read-only*

Independientemente de si se trabaja con Class Component o Functional Component, los valores que se reciben en *props* no deben ser modificados, son **sólo de lectura**. También se dice que las *props* deben ser "inmutables".

"All React components must act like pure functions with respect to their props."

Es decir, los componentes deben ser "funciones puras". Una función es pura cuando no modifica sus *inputs*, y siempre retorna el mismo resultado para el mismo *input*.

Funciones puras e impuras

```
// Función Pura:
function suma(a, b) {
   return a + b;
}

// Función Impura:
function retiro(cuenta, cantidad) {
   cuenta.total -= cantidad;
}
```

State

State (1/2)

Las interfaces de usuario de las aplicaciones son dinámicas y cambian con el tiempo. Las *props* sirven para pasar información de un componente padre a un componente hijo, pero de poco sirven si no pueden modificar.

Para esto surge un concepto llamado *"state"* (estado).

El *state* le permite que los componentes de React cambien (modifiquen su *output*) a lo largo del tiempo en respuesta a acciones del usuario, respuestas de red y cualquier otra cosa, sin violar la regla de la inmutabilidad de los *props*.

El *state* no son más que variables o atributos definidos dentro del componente, que contienen datos que el componente necesita conocer. A partir del *state* y las *props*, el componente determina qué UI (User Interface) renderizar.

State (2/2)

- El *state* es privado para el componente y controlado completamente por él.
- No es posible acceder al *state* de un componente desde otro componente.
- Más adelante veremos cómo manejar un *state* global a toda la aplicación.

State en un Class Component (1/2)

En el caso de un Class Component, el estado se define como un atributo de clase:

```
class UnComponente extends React.Component {
  state = {
    nombre: "María",
    edad: 34,
  };
  render() {
    return <p>¡Hola {this.state.nombre}!</p>;
  }
}
```

El state es un Objeto que contendrá todas las propiedades que se quieran manejar localmente dentro del componente.

State en un Class Component (2/2)

No se debe actualizar el *state* utilizando la asignación `=`.

```
this.state.nombre = "Victoria";
```

Si se hace de esta forma, React no detectará el cambio y no aplicará el cambio en la UI.

La forma de hacerlo es utilizando el método `setState`.

```
this.setState({
    nombre: "Victoria"
})
```

Sólo es necesario especificar los atributos del state que se deseen modificar, en este caso: `nombre`.

State en un Functional Component (1/2)

En el caso de un Functional Component, el estado son variables las cuales van acompañadas de métodos (*setters*) para modificarlas.

Además se deben definir usando una función especial de React llamada `useState`.

```
const UnComponente = (props) => {
  const [nombre, setNombre] = React.useState("María");
  const [edad, setEdad] = React.useState(34);
  return <p>¡Hola {nombre}!</p>;
}
```

State en un Functional Component (2/2)

La función `useState` es lo que se conoce como un *"hook"*.

Los *hooks* son funciones especiales de React que permiten agregar funcionalidades a lo Functional Components. Ver documentación.

```
const [nombre, setNombre] = React.useState("María");
```

El *hook* `useState` recibe como parámetro el valor inicial que tendrá la variable.

Retorna un array conteniendo la variable y una función *setter* para poder modificarla.

```
setNombre("Victoria");
```

```
nombre = "Victoria";
```

Event listeners

Event listeners (1/2)

El manejo de eventos en React es muy similar a como se realiza en HTML. Las diferencias principales son de sintaxis:

```
<button onclick="buscarPelicula()">
  Buscar película
</button>
```

vs.

```
<button onClick={buscarPelicula}>
  Buscar película
</button>
```

React provee un set de *wrappers* de los eventos nativos de JS denominados Synthetic Events. Los mismos actúan de la misma manera que los eventos originales, solucionan algunos problemas y son más fáciles de utilizar.

Sólo se aplican a elementos del DOM: No van a funcionar en un Componente *custom*.

Lista de eventos soportados https://reactjs.org/docs/events.html#supported-events.

Event listeners (2/2)

Entonces, ¿cuándo utilizar `addEventListener`?

Si se quiere obtener comportamientos complejos, es necesario implementar un *event listener* propio. Por ejemplo: para detectar el resize de la ventana. Más info aquí.

Para el resto de los casos, usar eventos de React. Esto es porque:

- Maneja event pooling.
- Elimina los *listeners* por nosotros.
- Es más óptimo.
- Ofrece mayor compatibilidad entre navegadores.

Formularios

Formularios (1/2)

Los campos de un formulario en React se crean de la misma forma que cualquier formulario HTML. Por ejemplo, para crear un campo de texto se escribe:

```
<input type="text" />
```

Notar que, por la naturaleza de HTML, un campo mantiene un estado interno, el cual se actualiza cuando el usuario modifica los datos. 👉 Este estado es diferente del estado que maneja un componente en React.

Sin embargo, React propone tener una única fuente de verdad, y que sea React que controle el estado de los campos, de forma externa.

Ver documentación: https://reactjs.org/docs/forms.html.

Formularios (2/2)

Para poder controlar un campo con React, es necesario declarar una variable de estado que contenga el valor de dicho campo. Inicialmente, el valor del campo es vacío (""). A medida que el usuario escriba en el *input*, se guardará el texto ingresado en la variable `nombre`.

```
const [nombre, setNombre] = React.useState("");
```

```
<input
  type="text"
  value={nombre}
  onChange={(event) => setNombre(event.target.value)}
/>
```