Curso de React y React Native

Clase 02

Estas diapositivas son de uso exclusivo para alumnos y ex-alumnos de Hack Academy.

Agenda de la clase

Agenda

- Repaso.
- Tipos de componentes en React.
- Más sobre las Props.
- Manejo del State.
- Keyword `this` en JavaScript.
- Conditional Rendering
- Ejercicios.

Repaso

Tipos de componentes en React

Tipos de componentes

```
import React, { Component } from 'react';
import ReactDOM from "react-dom";
class HelloWorld extends React.Component {
    render () {
        return (
            <div>
                Hello {this.props.name}
            </div>
        )
    }
}
ReactDOM.render(<HelloWorld name='Diego' />,
   document.getElementById('app'))
```

Class Component

```
import React from 'react';
import ReactDOM from "react-dom";
function HelloWorld (props) {
      return (
        <div>Hello {props.name}</div>
   );
}
ReactDOM.render(<HelloWorld name='Diego' />,
   document.getElementById('app'))
```

Functional Component

Class Component

Al declarar un Class Component se está utilizando la sintaxis de `class` incorporada en ES6 a JavaScript. Siempre es necesario que tenga al menos un método `render`, pero puede tener otros.

Una de las características que tienen estos componentes es que permiten utilizar los Lifecycle Methods de React, que son métodos de ciclo de vida de los componentes (se verá en detalle más adelante).

Class Component – Ejemplo

```
class HelloWorld extends React.Component {
    componentDidMount() {
        callSomeApiToGetInfo();
    }
    render () {
        return (
            <div>
                Hello {`${this.props.name} ${this.props.lastName}`}
            </div>
        )
    }
}
```

Functional Component

Como el nombre lo dice, son **funciones**. Reciben un único parámetro de tipo Object, y retornan “UI”.

Todos los componentes propios debemos nombrarlos con mayúscula, al igual que con los Class Components, de forma que React no piense que son tags html.

Functional Component – Ejemplo

```
// Podemos deconstruir (destructuring) los parámetros del objeto props
function HelloWorld ({ name }) {
    return (
        <div>Hello {name}</div>
    );
}
// Tambien podemos usar arrow functions (ES6)
const HelloWorld = (props) => {
    return (
        <div>Hello {props.name}</div>
    );
}
```

Más sobre las *props*

Props son Read-Only

No importa si se trabaja con Class Component o Functional Component, en cualquiera de los casos los valores que se reciben en *props* no se pueden modificar, son **sólo de lectura**.

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

Las funciones puras son un concepto relacionado con la programación funcional. Una función es pura cuando no modifica sus *inputs*, y devuelve siempre el mismo resultado para el mismo *input*.

Funciones puras e impuras

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

// Función Impura
function withdraw(account, amount) {
    account.total -= amount;
}
```

¿Cómo se modifica un componente?

Como las *props* son *Read-Only*, hasta ahora es perfecto para construir sitios estáticos, pero claramente no siempre es el caso deseado.

La forma para que los componentes y sitios desarrollados con React sean más interesantes, y que se conviertan en sitios dinámicos, que cambian su información a lo largo del tiempo, es utilizando un nuevo concepto de React llamado *state* (estado).

State – Estado

El *state* permite que los componentes de React modifiquen su *output* a lo largo del tiempo en respuesta a acciones del usuario, a respuestas de la red, y cualquier otro evento, sin violar la regla de que tienen de actuar como funciones puras con respecto a sus *props*.

Es un simple **objeto JavaScript** que contiene las variables necesarias para el componente, por lo tanto a partir del *state* y las *props*, el componente determina qué UI (User Interface) renderizar.

Manejo del State

Manejo del *State* (1)

El *state* es similar a las *props*, pero es **privado** para el componente y **controlado** completamente por el componente.

No se puede acceder al *state* de un componente desde otro; y si esto se logra mediante algún truco de JavaScript, igualmente va en contra de los conceptos de React.

Manejo del *State* (2)

Podemos definir el **Estado Inicial** del componente como un **Atributo de Clase**.

```
class MyComponent extends React.Component {
    state = {
      name: 'Soy un ejemplo',
    };
    render() { return <p>Hola Hack Academy!</p> }
}
```

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

Manejo del *State* (3)

Para acceder al valor de *state* se puede hacer a través de la keyword `this`

```
class MyComponent extends React.Component {
    state = {
      name: 'Soy un ejemplo',
    };
    render() { return <p>Hola {this.state.name}!</p> }
}
```

Manejo del *State* (4)

Como se mencionó anteriormente, no es posible actualizar el *state* simplemente utilizando la asignación con =.

```
// Incorrecto!
this.state.comment = 'Hello';
```

El motivo de esto, es que si se hace de esta forma React no va a detectar el cambio y no aplicará el cambio en Componente, por lo que no se verá el cambio en la UI. Una de las ventajas de React es que es eficiente a la hora de actualizar el DOM (lo que el usuario ve en la pantalla). Por eso es importante que si el estado interno de un componente cambia, esto se refleje correctamente en pantalla al usuario.

Manejo del *State* (5)

La forma correcta es utilizar la función `setState`, que recibe un objeto con el cual actualiza el estado del componente. **Sólo** actualiza las propiedades que contiene el objeto pasado por parámetros, por lo que **no es necesario que contenga las propiedades que no se modificaron**.

```
state = {
  description: "Soy un ejemplo",
  comment: "Comentario inicial"
};
// En algún lugar del componente:
this.setState({
    comment: "Comentario actualizado"
})
```

No se pierde ni se modifica el valor de `description`!

`setState` es asíncrono (1)

React puede juntar varias llamadas al `setState` de forma de ejecutarlas juntas y ser más eficiente (rápido), lo que quiere decir que el **`setState` no es garantizado que se ejecute de inmediato**.

Por eso es que no hay que basarse en `this.state` para calcular el siguiente estado.

```
// Esto puede fallar, hay que evitarlo.
this.setState({
    counter: this.state.counter + this.props.increment,
});
```

`setState` es asíncrono (2)

Por suerte, hay otra forma de utilizar `setState` que nos permite modificar el estado en función del estado anterior. **Le pasamos una función** en lugar del objeto. Esta función recibirá el estado anterior como primer parámetro y las *props* como segundo. Luego, simplemente retornamos el nuevo estado calculado en función de los parámetros.

```
// Esto garantiza que tendra los valores correctos
this.setState((prevState, props) => (
  { counter: prevState.counter + props.increment }
));
```

Inmutabilidad en React (1)

En programación en general, hay dos formas de actualizar información: La primera, consiste en mutarla, modificando directamente el valor de una variable. La segunda consiste en reemplazar los datos en una **copia** del objeto que también incluye los cambios deseados.

Mutando los datos

```
let player = this.state.player; // {score: 1, name: 'Jeff'};
player.score = 2;
this.setState({ player }) // Es lo mismo que: this.setState({ player: player })
```

Sin mutar los datos

```
const player = Object.assign({}, this.state.player, {score: 2});
// o... const player = { ...this.state, score: 2 } // Object Spread Operator
this.setState({ player }) // De nuevo, es lo mismo que: this.setState({ player: player })
```

Inmutabilidad en React (2)

Ventajas de la inmutabilidad: Seguimiento de cambios

Determinar si un objeto **mutado** ha cambiado es **complejo**: los cambios se realizaron directamente en el objeto. Esto requiere comparar el objeto actual con una copia anterior, recorrer todo el árbol de objetos y comparar cada variable y valor. Este proceso puede llegar a ser cada vez más complejo.

Determinar cómo ha cambiado un objeto inmutable es considerablemente más fácil: Si el objeto al que se hace referencia es diferente al de antes, entonces el objeto ha cambiado.

Inmutabilidad en React (3)

El state en React debe ser tratado como Inmutable.

El mayor beneficio de la inmutabilidad en React viene cuando se construyen componentes puros simples. Dado que los datos inmutables pueden determinar con más facilidad si se han realizado cambios, también ayuda a determinar cuándo un componente necesita ser renderizado nuevamente.

Keyword `this` en JavaScript

Keyword `this`

Es importante entender este concepto ya que en JavaScript el valor de `this` depende del contexto, lo cual puede generar problemas al utilizarlo.

Si se tiene la siguiente función, ¿cuál es el valor de `name`?

```
var sayName = function(name) {
    console.log('Hello, ' + name);
}
```

No se sabe hasta que alguien la invoca.

```
sayName('Diego');
```

Keyword `this` – Implicit Binding

Esta regla consiste simplemente en mirar a la izquierda del punto al momento de la invocación. Este es el caso más común.

```
const me = {
 name: 'Diego',
 age: 27,
 sayName: function() {
     console.log(this.name);
 }
};

me.sayName()
```

Se mira a la izquierda del "." cuando se invoca y se ve 'me', entonces es me.name => Diego

JSBin - Ejemplo 1 JSBin - Ejemplo 2

Keyword `this` – Explicit Binding

Otra opción es utilizar el `bind`, que devuelve una nueva función con el contexto que recibe por parámetros, aplicado a la función.

```
var sayName = function(food1, food2) {
    console.log("My name is " + this.name + " and I like " + food1 + " and " + food2);
};
var tom = {
    name: "Tom",
    age: 33
};
var foods = ["pizza", "chocolate"];
var newFn = sayName.bind(tom, foods[0], foods[1]);
newFn();
```

JSBin - Este Ejemplo

State y uso del `bind`

En React es común que se utilice el `bind` para que las funciones definidas dentro de un Class Component, siempre estén 'bindeadas' al mismo contexto, es decir, al contexto del componente.

```
class InputExample extends React.Component {
    state = { text: "" };
    constructor(props) {
        super(props);
        this.change = this.change.bind(this);
    }
    handleChange(event) {
        this.setState({ text: event.target.value });
    }
    render() {
        return <input type="text" value={this.state.text} onChange={this.change} />;
    }
}
```

Si no se "bindea" change, al utilizar `this.setState` no estaría definido

...Más facil

Si tuviéramos muchos handlers de cambios en inputs como en el ejemplo anterior, se imaginan tener que hacer todo eso a mano todas las veces? Mucho trabajo.
Por suerte, conocemos las Arrow Functions, que **no** tienen `this` propio.

```
class InputExample extends React.Component {
 state = { text: '' };
 render() {
   return (
     <input
       type="text"
       value={this.state.text}
       onChange={event => this.setState({ text: event.target.value })}
     />
   );
 }
}
```

Conditional Rendering

Conditional Rendering (1)

En React se pueden crear distintos componentes para encapsular en cada uno el comportamiento necesario. Luego, se puede utilizar Conditional Rendering para procesar solo alguno de ellos, dependiendo del estado de la aplicación.

```
const UserGreeting = () => <h1>Welcome back!</h1>;

const GuestGreeting = () => <h1>Please sign up.</h1>;


const Greeting = ({ isLoggedIn }) => isLoggedIn ? <UserGreeting /> : <GuestGreeting />;
```

Conditional Rendering (2)

También se pueden utilizar variables para almacenar los componentes y después mostrar (o no) parte de un componente sin que afecte el resto del *output*.

```
render() {
    const isLoggedIn = this.state.isLoggedIn;
    let body = null;
    if (this.state.isLoggedIn) {
        body = <App  />;
    } else {
        body = <LoginComponent />;
    }
    return (
        <div>
            {body}
        </div>
    );
}
```

Conditional Rendering – Operador Lógico

Como todo lo que se escriba entre `{ }` será evaluado como una expresión JavaScript, también se puede utilizar el operador lógico de JavaScript `&&` lo que permite lograr condicionales más cortos.

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

Conditional Rendering – Inline `if/else`

También se puede utilizar el operador condicional inline:

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

Conditional Rendering – Inline `if/else`

Se pueden agregar expresiones un poco más complejas, aunque hay que tener cuidado de que el código siga siendo legible.

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

Ejercicio

Ejercicio: Countdown!

1. Crear un nuevo proyecto de React. ¡Ya saben hacerlo!
2. En el proyecto recién creado, poner un `<input>` que reciba un valor numérico como **prop** llamada `defaultCount`. Dicha prop será el **valor por defecto** del `<input>`.
3. Luego, agregar un `<button>` que diga START.
4. El usuario podrá ingresar en dicho `<input>` un valor alternativo, que reemplazará el valor que el componente tenía por defecto al ser invocado.
5. Cuando el usuario clickea en START (lo cual ejecutará el evento onClick) aparecerá en pantalla un nuevo componente: el cual irá mostrando una cuenta regresiva de segundos. Dicha cuenta regresiva comenzará, sí, lo adivinaron; con el valor del `<input>` anterior, hasta llegar a 0.