Los componentes son entidades independientes. Estos solo atienden a las propiedades (**props**) que reciben y a su propio estado (**state**).

Los **props** son un objeto que contiene los valores que un componente recibe cuando se instancia. Estos pueden ser funciones, objetos, variables o clases. Los valores pueden cambiar a lo largo del tiempo, pero nunca será el componente que recibe dichos valores el que los modifique.

El **state** es un objeto en el se almacenan todos los valores que representan a un componente en un instante determinado. Tal y como indica su traducción, es el estado de un componente. Este se puede inicializar cada vez que se instancia el componente y modificar durante su ciclo de vida. El estado es immutable.

Un componente sólo se actualiza cuando cambian sus props o su state. Cada cambio en estos valores provocará una llamada al método *render* del componente.

Immutabilidad

Ya hablamos en la primera sección sobre inmutabilidad, pero es el momento de volver a explicarla en contexto ya que es un concepto muy importante en React. Cuando hablamos de objetos inmutables no nos referimos a la librería ImmutableJS, **sino a objetos que no pueden ser modificados directamente**. Cada cambio en estos objetos **implica una copia nueva con el valor modificado**.

De esta manera, siempre tendremos la versión original y la modificada del objeto. Como iremos viendo a lo largo del curso, esta característica nos permite comparar los cambios en nuestros componentes. Gracias a las optimizaciones que este tipo de datos tienen en la máquina virtual de JavaScript, React puede realizar comparaciones y aplicar los cambios necesarios en la interfaz. **Todo ello proporciona un gran rendimiento a nuestras aplicaciones**.

Definición y uso

Props

Las props son valores que se utilizan para enviar información a un componente. En JSX se escriben como atributos, tal y como vimos en el apartado anterior. **El propio componente nunca debe cambiar estos valores**, será el padre el encargado de modificarlos.

Para la sintaxis de clases, los props se encuentran disponibles en `this.props`.

```
class Title extends React.Component {
  render() {
    return <h1>{ this.props.title }</h1>;
  }
}
```

Para enviar un título a nuestro componente utilizamos el atributo que hemos definido:

```
let title = <Title title="Trabajando con Props!" />;
```

En el caso definir el componente como una función, las props se pasan como parámetro de esta.

```
const Alert = props =><div className={ `Alert Alert--${props.type}` }>{ props.text }</div>;
```

Del mismo modo, pasamos las props a nuestro componente mediante atributos en JSX:

```
let errors = <Alert type="error" text="El usuario no existe" />;
```

Código en Codepen.

PropTypes

Las props de los objetos deben de ser del tipo que estos esperan. Por ejemplo, podríamos transformar nuestro título a mayúsculas en el componente del apartado anterior.

```
class Title extends React.Component {
  render() {
    return <h1>{ this.props.title.toUpperCase() }</h1>;
  }
}
```

Al llamar a `toUpperCase` estamos suponiendo que el prop `title` es de tipo `string` o que este define el método. Si al instanciar el componente pasamos un número en lugar de una `string` en `title`, este lanzará una excepción. No obstante, no todos estos errores son tan obvios y sencillos de corregir.

Para evitar este tipo de errores, React nos permite definir los `PropTypes` de un componente. Mediante este objeto definimos el tipo de cada prop que el componente espera recibir. De esta manera nos aseguramos que obtener un mensaje `warning` en consola cada vez que nuestro componente recibe un prop del tipo que no espera. **La finalidad de estos errores es de depuración. Estos no se muestran en producción**. Si utilizamos la versión minificada de React estos errores nunca se mostrarán.

Los `PropTypes` se pueden definir de distintas maneras dependiendo de si el componente está definido como una clase o una función. El único requisito es que esté definida como una propiedad estática. Veamos algunos ejemplos.

```
class Title extends React.Component {
  // Definimos los propTypes como una propiedad estática
  static propTypes = {
    // Definimos el tipo de title
    title: React.PropTypes.string.isRequired
  };

  render() { return <h1>{ this.props.title.toUpperCase() }</h1> }
}
```

Antes de continuar con los ejemplos, vamos a analizar la definición del tipo de `title`. `React.PropTypes` incluye todos los `PropTypes` disponibles en la librería. `string` es el tipo que se espera recibir. `isRequired` indica que este prop debe de estar siempre definido. En el caso de que su valor sea `undefined` o no sea una `string`, el componente lanzará una excepción en consola. Si eliminamos `isRequired`, el prop será opcional. Tenéis todos los tipos disponibles en la documentación de React.

Otra manera de establecer los `PropTypes` en un componente de clase es mediante el uso de `get`.

```
class Title extends React.Component {
  // Definimos los propTypes como una propiedad estática
  static get propTypes() {
    return {
      // Definimos el tipo de title
      title: React.PropTypes.string.isRequired
    }
  }

  render() { return <h1>{ this.props.title.toUpperCase() }</h1> }
}
```

Por último, podemos definir los `PropTypes` en el propio prototipo de la clase o función. Esta forma es la única válida para los componentes definidos como funciones. También es válido para los definidos en clases.

```
class Title extends React.Component {
  render() { return <h1 className={ this.props.className }>{ this.props.title.toUpperCase() }</
h1> }
}

// Definimos los propTypes en el prototipo
Title.propTypes = {
  title: React.PropTypes.string.isRequired,
  className: React.PropTypes.string
}

// Igual para los componentes definidos en funciones
const RedText = props => <p style={ { color: 'red' } }>{ props.text }</p>;

RedText.propTypes = {
  text: React.PropTypes.string.isRequired
}

// Un ejemplo más complejo con array de objetos
const SumList = props => {
  let sum = 0;
  props.numbers.forEach(n => sum += n.number);
  return <p>La suma es igual a: { sum }</p>
}

SumList.propTypes = {
  numbers: React.PropTypes.arrayOf(React.PropTypes.shape({
    number: React.PropTypes.number.isRequired
  })).isRequired
}
```

Código en Codepen.

DefaultProps

React también nos permite definir valores por defecto para los distintos props. Esta funcionalidad es muy útil para modificar cómo se renderiza de un componente. Su definición es igual que los `PropTypes`, solo que el nombre de la propiedad es `defaultProps` en lugar de `propTypes`. Vamos a utilizar el componente `Alert` que definimos anteriormente:

```
// Las clases disponibles son: info, error y success.
const Alert = props =><div className={ `Alert Alert--${props.type}` }>{ props.text }</div>;

Alert.propTypes = {
  type: React.PropTypes.oneOf(['info', 'success', 'error'])
}
```

```
}

Alert.defaultProps = {
  type: 'info'
}
```

Código en Codepen;

State

Como su propio nombre indica, el estado de un componente almacena todos sus datos en un instante de tiempo. El propio componente puede modificar su estado posteriormente. Cada vez que este modifica el estado, se genera una nueva *versión* del componente, que se traduce en una llamada al método render para recargar el componente.

Para inicializar el estado utilizamos el `constructor` de la clase. Para acceder a los valores almacenados en el estado utilizamos el objeto `this.state`. Si queremos modificar algún valor del estado debemos de utilizar el método `setState`. Como hemos comentado, el objeto `state` es inmutable por lo que no podemos modificarlo directamente.

Con este sencillo contador podemos ver como iniciar, leer y actualizar el estado.

```
// Contador
class Counter extends React.Component {
  // Inicializamos nuestro estado
  constructor(props) {
    // ESTA LINEA SIEMPRE DEBE DE SER LA PRIMERA EN EL
    // CONSTRUCTOR
    super(props);

    // Inicializamos el estado. Aquí creamos el objeto
    // directamente.
    this.state = {
      counter: 0
    }
  }

  onClick = () => {
    // Actualizamos el valor del contador
    this.setState({ counter: this.state.counter + 1 })
  }

  // Renderizamos el componente
  render() {
    return <div>
      <h2>Counter</h2>
```

```
      <p>Current value: <b>{ this.state.counter }</b></p>
      <button onClick={ this.onClick }>Increase</button>
    </div>;
  }
}
```

Código en Codepen.

En este ejemplo hemos visto el estado en componentes de clases, pero ¿qué ocurre con los definidos como funciones? Esta es la diferencia entre ambos tipos de componentes. **Los componentes definidos como funciones no tienen estado. Por ello se suelen llamar componentes *Stateless***.

La finalidad de los componentes stateless suele ser representar datos, es decir, su función es puramente visual. No obstante, esto no siempre se cumple ya que estos también pueden incluir elementos con los que el usuario puede interaccionar. La clave es que ellos no almacenan ni procesan el resultado de esa interacción. Más adelante veremos como comunicar los componentes entre sí.