COMPONENTES & PROPS

Componentes & Props

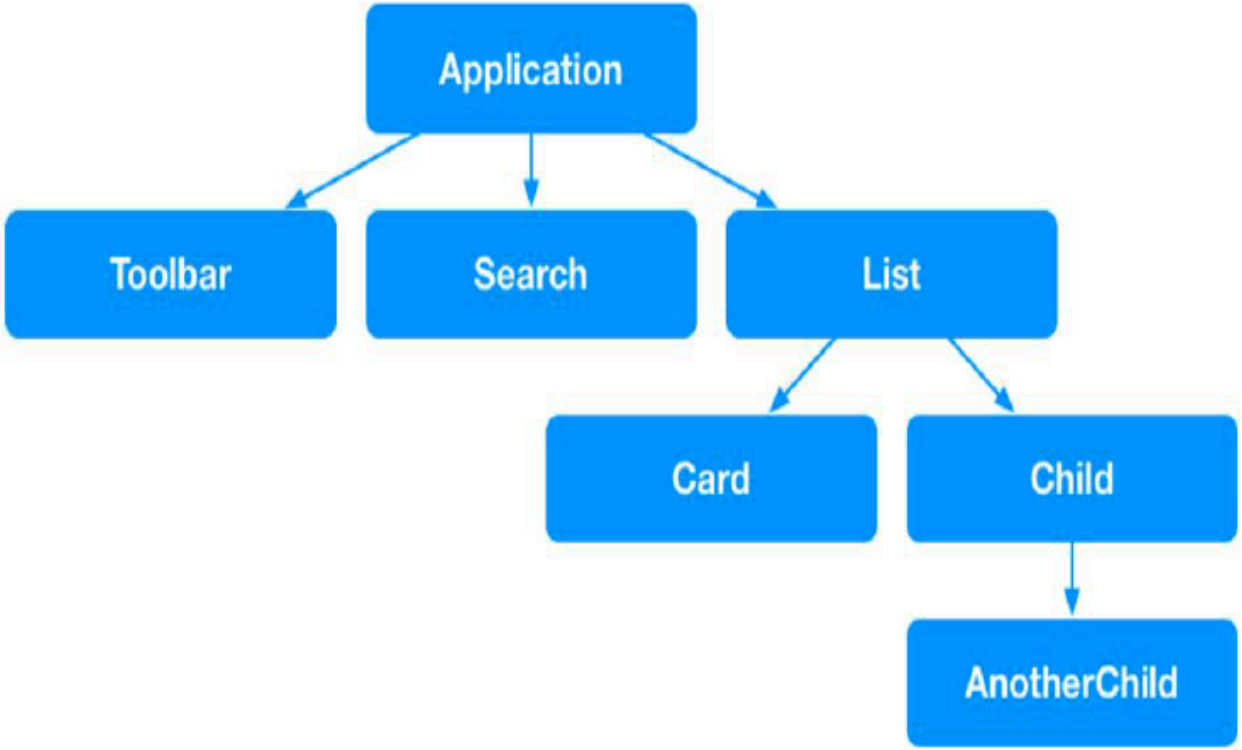
- **Que aprenderemos en este apartado?**

1. **Elementos React**
2. **Component structure**
3. **Controlled components**
4. **Props, State and lifecycle**
5. **Style**

COMPONENTES

- Los sistemas de hoy en día son cada vez más complejos, deben ser construidos en tiempo récord y deben cumplir con los estándares más altos de calidad.
- El paradigma de ensamblar componentes y escribir código para hacer que estos componentes funcionen entre si se conoce como Desarrollo de Software Basado en Componentes.
- Los Web Components nos ofrecen un estándar que va enfocado a la creación de todo tipo de componentes utilizables en una página web, para realizar interfaces de usuario y elementos que nos permitan presentar información (o sea, son tecnologías que se desarrollan en el lado del cliente). Los propios desarrolladores serán los que puedan, en base a las herramientas que incluye Web Components crear esos nuevos elementos y publicarlos para que otras personas también los puedan usar.
- Un componente puede estar compuesto por componentes que a su vez se compongan de otros componentes y así sucesivamente. Sigue un modelo de composición jerárquico con forma de árbol de componentes. La división sucesiva en componentes permite disminuir la complejidad funcional favoreciendo la reutilización y las pruebas.
- Los componentes establecen un cauce bien definido de entrada/salida para su comunicación con otros componentes

Árbol de componentes



COMPONENTES

- Los componentes le permiten dividir la IU en piezas independientes y reutilizables, y pensar en cada pieza aisladamente.
- Conceptualmente, los componentes son como las funciones de JavaScript. Aceptan entradas arbitrarias (llamadas "props" abreviatura de propiedades) y devuelven elementos React que describen lo que debería aparecer en la pantalla.
- La forma más sencilla de definir un componente es escribir una función de JavaScript:

```
function Saludo(props) {
  return <h1>Hola {props.name}</h1>;
}
```

Esta función es un componente React válido porque acepta un único argumento de objeto "props" con datos y devuelve un elemento React.

- También puede usar una clase ES6 para definir un componente:

```
class Saludo extends React.Component {
  render() {
    return <h1>Hola {this.props.name}</h1>;
  }
}
```

- Los dos componentes anteriores son equivalentes desde el punto de vista de React. Las clases tienen algunas características adicionales.

COMPOSICIÓN

- Un componente puede estar compuesto por componentes que a su vez se compongan de otros componentes y así sucesivamente.
- Sigue un modelo de composición jerárquico con forma de árbol de componentes.
- La división sucesiva en componentes permite disminuir la complejidad funcional favoreciendo la reutilización y las pruebas.
- Un botón, un formulario, un listado, una página: en las aplicaciones React, todos ellos se expresan comúnmente como componentes.

```
function App(props) {
  return (
    <div>
      <Saludo name="Don Pepito" />
      <Saludo name="Don Jose" />
    </div>
  );
}
```

RENDERIZAR EL COMPONENTE

- Los nombres de los componentes deben comenzar siempre con una letra mayúscula.
- Los componentes se utilizan como si fueran nuevos tipos de etiquetas del HTML dentro de los elementos.
- Las propiedades toman la sintaxis de los atributos HTML, cuando React ve un elemento que representa un componente definido por el usuario, pasa los atributos JSX al componente como propiedades de un solo objeto.

```
const element = <Saludo name="mundo" />;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

Tipos de componentes

En ReactJS existen dos categorías recomendadas para los componentes:

- Los **componentes de presentación** son aquellos que simplemente se limitan a mostrar datos y tienen poca o nula lógica asociada a manipulación del estado, es preferible que la mayoría de los componentes de una aplicación sean de este tipo porque son más fáciles de entender y analizar.

- Los **componentes contenedores** tienen como propósito encapsular a otros componentes y proporcionarles las propiedades que necesitan, además se encargan de modificar el estado de la aplicación por ejemplo usando Flux o Redux para despachar alguna acción y que el usuario vea el cambio en los datos.

CARACTERÍSTICAS

Características de los componentes de presentación
–Orientados al aspecto visual
–No tienen dependencia con fuentes de datos (e.g. Flux)
–Reciben callbacks por medio de props
–Pueden ser escritos como componentes funcionales.
–Normalmente no tienen estado

Características de los componentes contenedores
–Orientados al funcionamiento de la aplicación
–Contienen componentes de presentación y/o otros contenedores
–Se comunican con las fuentes de datos
–Usualmente tienen estado para representar el cambio en los datos

PROPIEDADES

- Las propiedades permiten personalizar los componentes.
- Las propiedades toman la sintaxis de los atributos HTML.
- Las propiedades se encapsulan en un único objeto (array asociativo o diccionario), donde las claves son los nombres de los atributos y los valores son los valores de los atributos, y se almacenan en la propiedad heredada this.props.
- Las propiedades son inmutables, de solo lectura. React es bastante flexible pero tiene una sola regla estricta: Los componentes no deben modificar las propiedades.
- Los componentes deben comportarse como “funciones puras“ porque no deben cambiar sus entradas y siempre devuelven el mismo resultado para las mismas entradas.

PROPIEDADES

- Los valores de las propiedades pueden ser valores constantes o expresiones:

```
const element = <Saludo name={formatName(user)} />;
```

- Si no se asigna valor al atributo se establece de manera predeterminada con el valor true.

```
<MyTextBox autocomplete />
<MyTextBox autocomplete={true} />
```

- Si ya se dispone de las propiedades en un objeto se puede pasar utilizando el operador de "propagación" ... (ES6):

```
const attr = {init: 10, delta: '1'};
return <Contador {...attr} />; // <Contador init={10} delta="1" />
```

- El operador de "propagación" también permite la des-estructuración:

```
const { tipo, ...attr } = props;
if (tipo)
   return <Contador {...attr} />;
return <Slider {...attr} />;
```

CONTENIDO DEL COMPONENTE

- Los componentes tienen una etiqueta de apertura y otra de cierre, el contenido entre estas etiquetas se pasa como una propiedad especial: props.children.

```
<MiContenedor>Hola Mundo ...</MiContenedor>
```

- Puede contener otros componentes, literales, etiquetas, expresiones, funciones o cualquier otra cosa valida dentro de cualquier etiqueta.
- El contenido puede ser múltiple y mezclado, en cuyo caso props.children toma la forma de un array.

```
<MiContenedor>
  Texto
  <Saluda nombre="mundo" />
  <Contador init={10} delta={1} />
  <h1>Etiqueta</h1>
  {this.props.nombre}
  {(item) => <div>Formato para {item}.</div>}
</MiContenedor>
```

- Antes de pasar el contenido a props.children automáticamente se elimina: el espacio en blanco al principio y al final de una línea, las líneas en blanco, las nuevas líneas adyacentes a las etiquetas, los salto de línea en medio de los literales cadena que se condensan en un solo espacio.

PROPTYPES

- A medida que la aplicación crece, se pueden detectar muchos errores con la verificación de tipos de las propiedades de los componentes.
- PropTypes exporta una gama de validadores que se pueden usar para garantizar que los datos que reciba sean válidos. Cuando se proporciona un valor no válido, se mostrará una advertencia en la consola del navegador pero, por razones de rendimiento, solo se verifica en modo de desarrollo.
- Las validaciones se establecen como una estructura que se asigna al static propTypes del componente.

```
import PropTypes from 'prop-types';
class Contador extends React.Component {
  // ...
}
Contador.propTypes = {
  init: PropTypes.number.isRequired,
  delta: PropTypes.any,
  onCambia: PropTypes.func
};
const element = <Contador init={10} delta="1" />;
```

PROPTYPES

- Tipos primitivos: array, bool, func, number, object, string, symbol, any, element (de React), node (cualquier valor que pueda se renderizado).
- Tipos definidos:
  - instanceOf(MyTipo): instancia de una determinada clase.
  - arrayOf(PropTypes.???): Array con todos los valores de un determinado tipo.
  - objectOf(PropTypes.???): Un objeto con valores de propiedad de cierto tipo.
  - oneOf([...]): Valor limitado a valores especificados en el array tratándolo como una enumeración.
  - oneOfType([...]): Tipo unión, uno de los tipos especificados en el array.
  - shape({...}): Objeto que al menos tiene las propiedades y tipos indicados en el array asociativo.
- También se puede especificar un validador personalizado que debería devolver un objeto Error si la validación falla. "console.warn" o throw no funcionarán dentro de "oneOfType":

```
customyProp: PropTypes.arrayOf(function(propValue, key, componentName, location,
propFullName) {
  if (!/matchme/.test(propValue[key])) {
    return new Error('...');
  }
 })
```

PROPTYPES

- Se puede definir una propiedad como obligatoria marcándola con .isRequired

```
Contador.propTypes = {
  init: PropTypes.number.isRequired,
  delta: PropTypes.any,
  onCambia: PropTypes.func
};
```

- Los valores predeterminados de la propiedades se establecen como una estructura que se asigna al static defaultProps del componente.

```
Contador.defaultProps = {
  delta: 5
};
```

ESTADO

- El estado es similar a las propiedades pero puede cambiar, es privado y está completamente controlado por el componente.
- El estado local es una característica disponible solo para las clases.
- El estado es la propiedad this.state, que se hereda de la clase componente, y es un objeto que encapsula todas la propiedades de estado que se vayan a utilizar en el render().
- El estado se inicializa en el constructor, que obligatoriamente tiene que invocar al constructor del padre para propagar las propiedades:

```
class Contador extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      contador: +props.init,
      delta: +props.delta
    };
  }
```

ESTADO

- El estado es accesible a través de las propiedades de this.state:

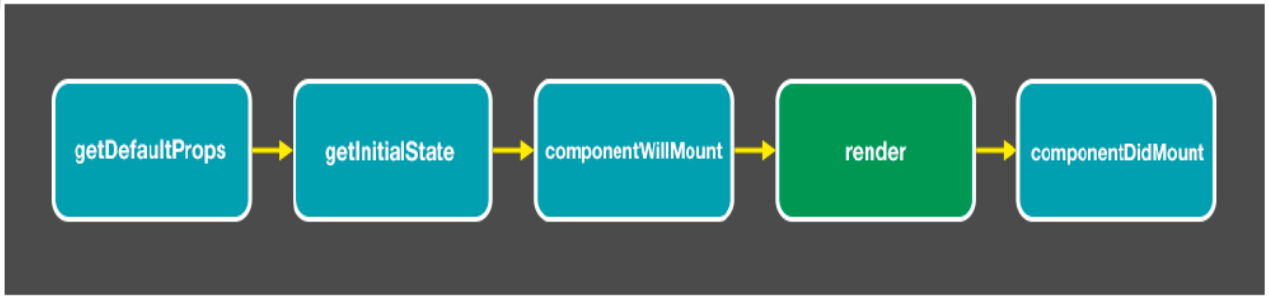
```
render() {
  return (
    <div>
      <h1>{this.state.contador}</h1>
      <p>
        <button onClick={this.sube}>Sube</button> 
        <button onClick={this.baja.bind(this, 1, 2)}>Baja</button>
      </p>
    </div>
  );
 }
}
```

- Los cambios en el estado provocan que se vuelva a ejecutar el método render.

CICLO DE VIDA DE UN COMPONENTE

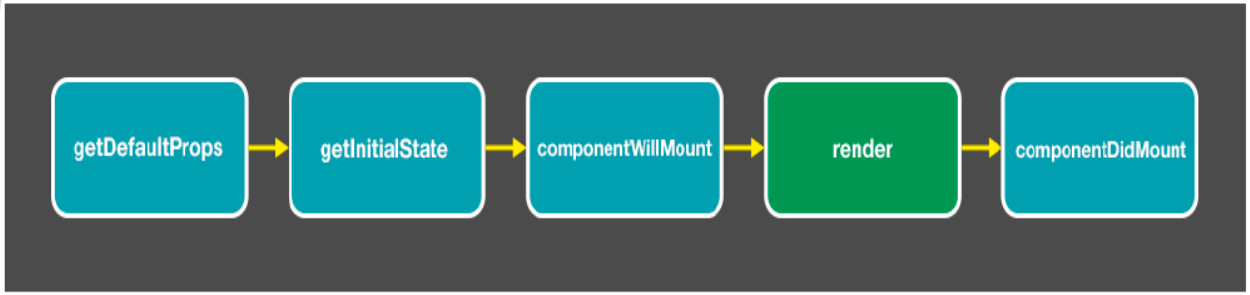
- Para comprender el comportamiento de los componentes de React es importante comprender su ciclo de vida y los métodos que intervienen en dicho ciclo. Lo interesante de esto es que los métodos se pueden sobreescribir para conseguir que el componente se comporte de la manera que se espera.
- Podemos plantear cuatro fases que pueden darse en el ciclo de vida de un componente.
  - Montaje o inicialización
  - Actualización de estado
  - Actualización de propiedades
  - Desmontaje
- Cada método tiene un prefijo will o did dependiendo de si ocurren antes o después de cierta acción.

MONTAJE



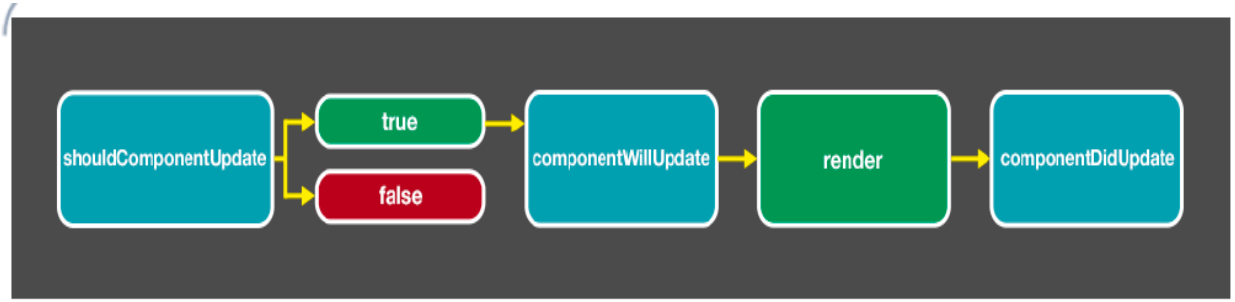
- **getDefaultProps y getInitialState** o el **constructor(props)** de la clase: se ejecuta cuando se instancia un componente. Nos permite definir el estado inicial del componente, hacer el bind de métodos y definir campos de clase.
- **componentWillMount()**: se ejecuta cuando el componente se va a renderizar. En este punto es posible modificar el estado del componente sin causar una actualización (y por lo tanto no renderizar dos veces el componente).

MONTAJE



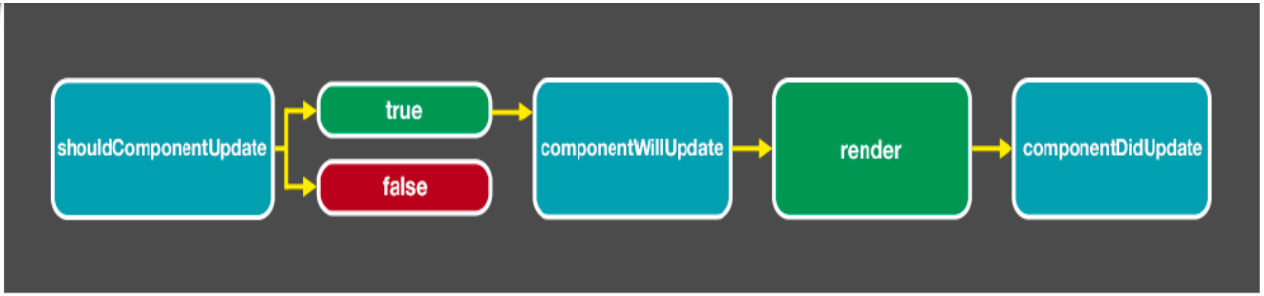
- **render()**: único obligatorio, en esta fase genera la UI inicial del componente. Debe ser una función pura (no puede tener efectos secundarios) y no debe modificar nunca el estado del componente.
- **componentDidMount()**: se ejecuta cuando el componente ya se renderizó en el navegador y permite interactuar con el DOM o las otras APIs del navegador (geolocation, navigator, notificaciones, etc.). Es el mejor sitio para realizar peticiones HTTP o suscribirse a diferentes fuentes de datos (un Store o un WebSocket) y actualizar el estado al recibir una respuesta. Cambiar el estado en este método provoca que se vuelva a renderizar el componente.

ACTUALIZACIÓN DE ESTADO



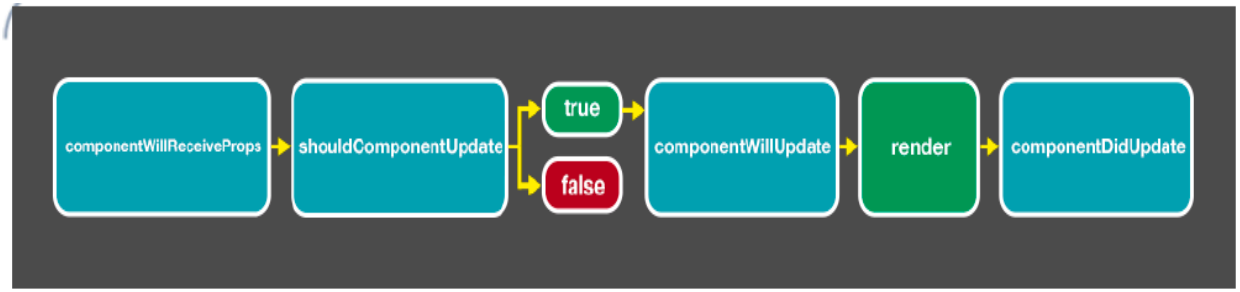
- **shouldComponentUpdate(next_props, next_state)**: decide si el componente se vuelve a renderizar o no. Recibe como parámetros las nuevas propiedades y el nuevo estado, debe devolver true para que se renderice. False puede afectar a sus componentes hijos que no reciben las nuevas propiedades.
- **componentWillUpdate(next_props, next_state)**: recibe como parámetros las nuevas propiedades y el nuevo estado para preparar al componente para su actualización por lo que no se debe modificar estados para evitar bucles infinitos.

ACTUALIZACIÓN DE ESTADO



- **render()**: en esta fase, genera una nueva versión UI del componente y modifica la versión actual con las discrepancias con la nueva versión. Debe ser una función pura (no puede tener efectos secundarios) y no debe modificar nunca el estado del componente.
- **componentDidUpdate(next_props, next_state)**: al igual que componentDidMount complementa al render con las operaciones en las que intervengan elementos del DOM y necesitan que ya estén creados.

ACTUALIZACIÓN DE PROPIEDADES



- **componentWillReceiveProps(next_props)**: Se ejecuta tan sólo cuando las propiedades del componente cambian (flujo del contenedor), recibiendo como parámetros las nuevas propiedades y puede actualizar el estado sin desencadenar que se vuelva a renderizar el componente.
- Los pasos siguientes en este escenario, a partir de este punto, son iguales que cuando se actualiza el estado del componente.

DESMONTAJE

- En el desmontaje de un componente React interviene un único método, **componentWillUnmount()**, y es invocado justo antes de que el componente se desmonte: se quite del árbol DOM.
- Es momento para realizar operaciones de limpieza como des suscribirse de los WebSocket o Stores, cancelar peticiones HTTP que hayan quedado pendientes, eliminar listeners, temporizadores o demás objetos que puedan quedar en memoria.
- Es muy importante para conservar los recursos y asegurar el rendimiento.

ESTILOS

- Hay distintas formas de agregar estilos a un componente:

  1.- Estilo general para toda la aplicación index.css
  2.- Estilo para un componente App.css

- Para usar los estilos es necesario importarlos con ***import***
- React funciona ligeramente diferente y para los atributos de clases no se utiliza *class* sino *className*

ESTILOS

- Es posible utilizar **Bootstrap** con React, sólo debe ser instalado con
  npm install bootstrap

y debe ser importado en el **index.js**

```
import React from 'react';
import ReactDOM from 'react-dom';
import 'bootstrap/dist/css/bootstrap.css';

import './global.css';
import Badge from './components/Badge';
```