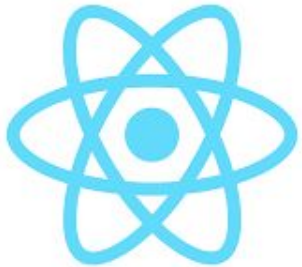
Curso de React.js



Daniel Rojo Pérez

Temario

Introducción a React.js

JSX

Eventos

Gestión de formularios

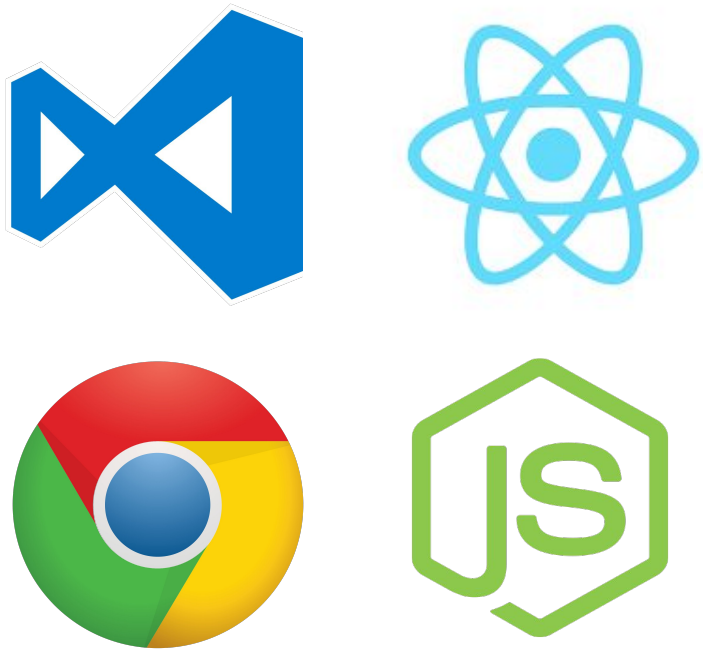
React.js y Servidor

Qué es React

Es una librería Javascript de código abierto para crear interfaces de usuario del tipo SPA (single page application). React.js es mantenida por Facebook, Instagram y una comunidad de desarrolladores independientes y compañías.

React.js implementa el patrón MVC (modelo-vista-controlador), únicamente se encarga de la interfaz de usuario y es muy común combinarla con otras librerías como Redux.

Herramientas



Primer proyecto con React

Con el cliente React

(Node >= 6 y npm >= 5.2)

> npx create-react-app my-app

> cd my-app

> npm start

Con un generador como Yeoman

> npm install -g yo

> npm install -g generator-react-webpack

> mkdir new-project && cd new-project

> yo react-webpack

> npm start

JSX

Es una extensión de la sintaxis de Javascript para trabajar con elementos del DOM como si fueran objetos de Javascript.

Con JSX podemos trabajar con estructuras de código HTML de una forma más accesible para el programador.

Además, JSX nos proporciona herramientas para mejorar la gestión del DOM de nuestra página web como un mecanismo para evitar ataques por inyección de código.

JSX - Databinding

Es una forma de llevar el valor de una variable de nuestro código JS al DOM, la actualización de esta variable se realizará a lo largo del ciclo de vida del componente de React, que veremos más adelante. Para primitiva tenemos la nomenclatura **{variable}**, para objetos **{{objeto}}** y para arrays **{[array]}**

```
const name = 'Curso de React';
const element = <h1>Este es el {name}</h1>;

ReactDOM.render(
 element,
 document.getElementById('root')
);
```

JSX - Expresiones JS

```
function formatName(user) {
   return user.firstName + ' ' +
user.lastName;
 }
  const user = {
   firstName: 'Harper',
   lastName: 'Perez'
 };
  const element = (
   <h1>
     Hello, {formatName(user)}!
   </h1>
 );
  ReactDOM.render(
   element,
   document.getElementById('root')
 );
```

JSX - Propiedades

Una parte importante de React y de la programación orientada a componentes son las propiedades de los elementos HTML que actúan como los parámetros de las funciones para introducir información en el componente.

Podemos pasar un valor estático:

```
const element = <div tabIndex="0"></div>;
```

También podemos combinarlo con databinding:

```
const element = <img src={user.avatarUrl}></img>;
```

JSX - Representación de objetos

Estos dos ejemplos hacen lo mismo:

```
const element = (
   <h1 className="greeting">
     Hello, world!
   </h1>
 );
```

```
const element = React.createElement(
   'h1',
   {className: 'greeting'},
   'Hello, world!'
 );
```

En el fondo son este objeto de Javascript:

```
const element = {
   type: 'h1',
   props: {
     className: 'greeting',
     children: 'Hello, world!'
   }
 };
```

Renderización de elementos HTML

Para indexar en el DOM un elemento de React, empleamos ReactDOM.render()

```
<div id="root"></div>
```

```
const element = <h1>Hello, world</h1>;
ReactDOM.render(element, document.getElementById('root'));
```

Un elemento renderizado en el DOM es inmutable, esto es, que para modificar la información que muestra, tenemos que volver a invocarlo y sobreescribirlo en el DOM esta operación es muy costosa para el navegador en un número alto de iteraciones, conviene evitar esta situación, para ello tenemos los componentes.

Componentes y propiedades

Un componente de react es una entidad de código reutilizable que tiene un ciclo de vida propio y gestiona la información de entrada mediante propiedades y envía información de salida mediante eventos. Podemos crear un componente a partir de una función, el parámetro ‘props’ siempre apunta a las propiedades del componente en el código HTML

```
function Welcome(props) {
   return <h1 > Hello, {props.name}
< /h1>;}
```

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

Componentes ES6

Podemos usar la sintaxis de ES6 para definir un componente, que es una clase que extiende de React.Component. Es importante inicializar las propiedades y los estados.

```
class Welcome extends React.Component {
   constructor(props) {
     super(props);
     this.state = {};
   }
   render() {
     return <h1>Hello,
{this.props.name}</h1>;
   }
 }
```

Combinar componentes

```jsx
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
        <Avatar
user={props.author} />
        <div
className="UserInfo-name">
          {props.author.name}
        </div>
      </div>
      <div
className="Comment-text">
        {props.text}
      </div>
      <div
className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

```jsx
function UserInfo(props) {
  return (
    <div className="UserInfo">
      <Avatar user={props.user}
/>
      <div
className="UserInfo-name">
        {props.user.name}
      </div>
    </div>
  );
}
```

```jsx
function Comment(props) {
  return (
    <div className="Comment">
      <UserInfo user={props.author} />
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

Estado y ciclo de vida de componentes

Recordemos el caso de forzar el método ReactDOM.Render() para actualizar el valor de una variable:

```
function tick() {
   const element = (
     <div>
       <h1>Hello, world!</h1>
       <h2>It is {new
Date().toLocaleTimeString()}.</h2>
     </div>
   );
   ReactDOM.render(
     element,
     document.getElementById('root')
   );
 }
  setInterval(tick, 1000);
```

¡Está mal!
Cambia el DOM cada vez que hace una iteración

Usar las propiedades del componente

El componente se actualiza automáticamente cuando cambia el valor de alguna de sus propiedades

```
function Clock(props) {
   return (
     <div>
       <h1>Hello, world!</h1>
       <h2>It is
{props.date.toLocaleTimeString()}.</
h2>
     </div>
   );
 }
  function tick() {
   ReactDOM.render(
     <Clock date={new Date()} />,
     document.getElementById('root')
   );
 }
  setInterval(tick, 1000);
```

Variable state de los componentes

State es un objeto interno que contiene las variables que queramos gestionar en nuestro componente

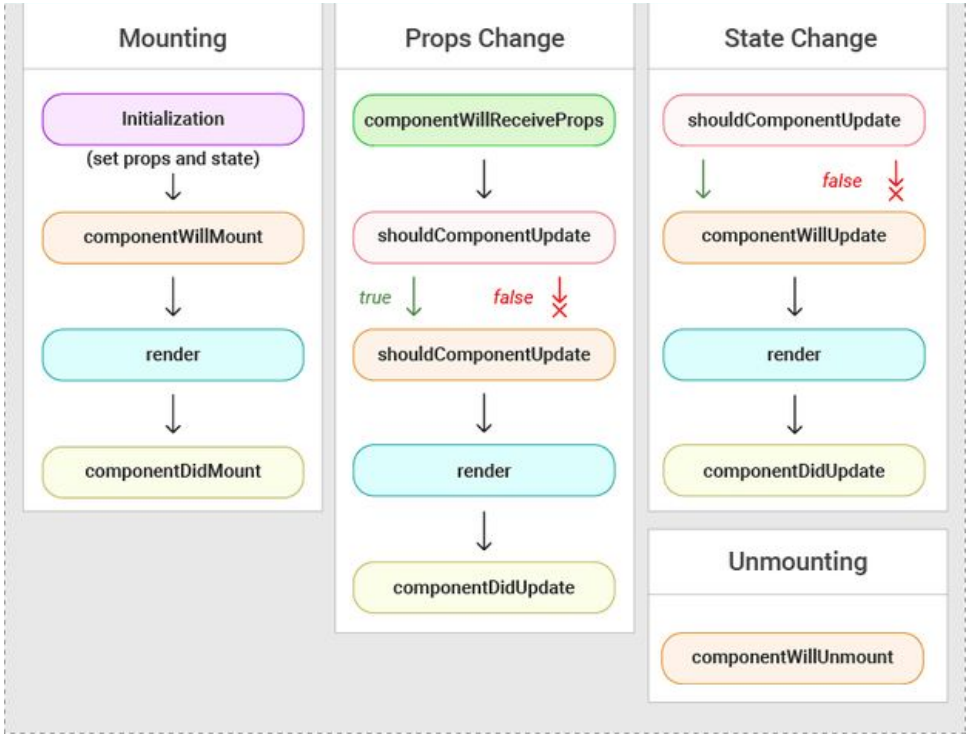
```
class Clock extends React.Component {
   constructor(props) {
     super(props);
     this.state = {date: new Date()};
   }
    render() {
     return (
       <div>
         <h1>Hello, world!</h1>
         <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
       </div>
     );
   }
 }
  ReactDOM.render(  <Clock />, document.getElementById('root'));
```

Ciclo de vida del componente

Podemos sobreescribir los métodos del ciclo de vida del componente (no todos), tenemos que tener en cuenta que es el navegador el que llama a estos métodos y puede variar bastante el número de llamadas a estos métodos en función los separamos en dos categorías:

- Métodos de creación del componente
- Métodos de actualización del componente

Métodos en el ciclo de vida el componente



Métodos de creación del componente

constructor (props)	Se llama antes de renderizar
componentWillMount()	Antes de renderizar , no es conveniente usarlo
render()	Método de renderización, hay que usarlo, pero con cuidado
componentDidMount()	Después de renderizar el componente
componentWillUnmount()	Antes de eliminar el componente del DOM
componentDidCatch()	Se llama cuando se produce un error en el componente

Métodos de actualización del componente

componentWillReceiveProps (newProps)	Este método se llama antes de actualizar los cambios de propiedades del componente
shouldComponentUpdate (newProps, newState)	Bloquea la renderización si devuelve ‘false’
componentWillUpdate (newProps, newState)	No podemos usar setState() aquí
render()	Renderiza el componente, usar con cuidado
componentDidUpdate (prevProps, prevState)	Se llama después de actualizar el componente

Eventos de React

React hace su propia implementación de los eventos de los elementos HTML, podemos llamar a métodos definidos en nuestro componente.

```
class LoggingButton extends React.Component {
   handleClick(e) {
     console.log('this is:', this);
   }
    render() {
     return (
       <button onClick={(e) => this.handleClick(e)}>
         Click me
       </button>
     );
   }
 }
```

Renderizado condicional

En muchas ocasiones queremos modificar el HTML que renderizamos en función del valor de ciertas variables

```
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 “inline”

En ocasiones queremos poner condiciones en el propio código HTML para que se renderice siempre en función del valor de un boolean

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

Renderizado de los contenidos de una lista

En la programación de frontend son muy importantes las listas de componentes, la inmensa mayoría de las aplicaciones trabajan con listas. Vamos a hacer el renderizado de listas en dos pasos:

1. Usar la función map() para realizar las iteraciones de componentes de la lista
2. Indexar las las iteraciones dentro de nuestro objeto JSX para renderizar

```
Const heroes = ['superman', 'batman'],
const listHeroes = this.state.heroes.map(
(hero, i) => <li key={i}>{hero}</li>)
```

```
ReactDOM.render(
        <ul>{listHeroes}</ul>,
        document.getElementById('root')
);
```

Formularios con React

Para trabajar con formularios en React, vamos a combinar las herramientas que ya hemos visto: escuchamos los eventos de cada uno de los componentes del formulario y modificamos los componentes en función de la información que le queramos mostrar al usuario. Tenemos 3 componentes básicos para los formularios:

1. Input
2. TextArea
3. Select

Ejemplo de input

```
class NameForm extends React.Component {
   constructor(props) {
     super(props);
     this.state = {value: ''};
      this.handleChange = this.handleChange.bind(this);
     this.handleSubmit = this.handleSubmit.bind(this);
   }
    handleChange(event) {
     this.setState({value: event.target.value});
   }
    handleSubmit(event) {
     alert('A name was submitted: ' + this.state.value);
     event.preventDefault();
   }
}
```

```
render() {
   return (
     <form onSubmit={this.handleSubmit}>
       <label>
         Name:
        <input type="text"
             value={this.state.value}
             onChange={this.handleChange} />
       </label>
       <input type="submit" value="Submit" />
     </form>
   );
 }
}
```

Ejemplo de Textarea

```
class EssayForm extends React.Component {
   constructor(props) {
     super(props);
     this.state = {
       value: 'Please write an essay about your
favorite DOM element.'
     };
      this.handleChange =
this.handleChange.bind(this);
     this.handleSubmit =
this.handleSubmit.bind(this);
   }
    handleChange(event) {
     this.setState({value: event.target.value});
   }
}
```

```
handleSubmit(event) {
   alert('An essay was submitted: ' + this.state.value);
   event.preventDefault();
 }
 render() {
   return (<form onSubmit={this.handleSubmit}>
        <label>
          Essay:
          <textarea value={this.state.value}
              onChange={this.handleChange} />
        </label>
        <input type="submit" value="Submit" />
      </form>
   );
 }
}
```

Ejemplo de Select

```
class FlavorForm extends React.Component {
   constructor(props) {
     super(props);
     this.state = {value: 'coconut'};
      this.handleChange = this.handleChange.bind(this);
     this.handleSubmit = this.handleSubmit.bind(this);
   }
    handleChange(event) {
     this.setState({value: event.target.value});
   }
    handleSubmit(event) {
     alert('Your favorite flavor is: ' +
this.state.value);
     event.preventDefault();
   }
 }
```

```
render() {
   return (
     <form onSubmit={this.handleSubmit}>
       <label>
         Pick your favorite flavor:
         <select value={this.state.value}
onChange={this.handleChange}>
           <option
value="grapefruit">Grapefruit</option>
           <option value="lime">Lime</option>
           <option value="coconut">Coconut</option>
           <option value="mango">Mango</option>
         </select>
       </label>
       <input type="submit" value="Submit" />
     </form>); }}
```

Peticiones rest con Fetch

En ES6 tenemos la función Fetch para hacer peticiones Rest. Fetch devuelve una promesa y podemos obtener el json como un objeto de la siguiente forma para hacer una petición GET

```
// GET
fetch('https://jsonplaceholder.typicode.com/todos')
  .then(response => response.json())
  .then(data => console.log(JSON.stringify(data)))
```

Un ejemplo de FETCH con el método DELETE

```
// DELETE
fetch('https://jsonplaceholder.typicode.com/users/1', {
 method: 'DELETE'
});
```

Peticiones POST y PUT con Fetch

```
// POST

fetch('https://jsonplaceholder.typicode.com/users',
{
 headers: { "Content-Type": "application/json;
charset=utf-8" },
 method: 'POST',
 body: JSON.stringify({
   username: 'Elon Musk',
   email: 'elonmusk@gmail.com',
 })
})
```

```
// PUT

fetch('https://jsonplaceholder.typicode.com/users/3'
, {
 headers: { "Content-Type": "application/json;
charset=utf-8" },
 method: 'PUT',
 body: JSON.stringify({
   username: 'Elon Musk',
   email: 'elonmusk@gmail.com',
 })
})
```