React

JSX

Enrique Barra Arias
Álvaro Alonso González

JSX - Especificación

- **Extensión de JavaScript similar a XML, sin alterar la semántica**
- Simple y "familiar" para los desarrolladores y diseñadores. El objetivo es definir fácilmente estructuras de árbol con atributos
- Podríamos usar React sin JSX, con JS vanilla (JS simple), pero el JSX hace que React sea mucho más simple y elegante
- Como XML, las etiquetas JSX tienen un nombre, atributos e hijos

```
<MyButton color="blue" shadowSize={2}>
  Click Me
</MyButton>
```

- Specs: https://facebook.github.io/jsx/
- Intro a JSX: https://reactjs.org/docs/introducing-jsx.html
- JSX en profundidad: https://reactjs.org/docs/jsx-in-depth.html

JSX – Tipos de elementos

- La primera parte de la etiqueta JSX (nombre de la etiqueta) determina el tipo de elemento de React
  - Utiliza la convención de mayúsculas y minúsculas
  - Los componentes empiezan por mayúsculas
  - Las etiquetas HTML comienzan en minúsculas

```
<MyButton color="blue" shadowSize={2}>
  Click Me
</MyButton>
```

```
<div className="sidebar" />
```

- JSX nos permite definir un componente de React como un elemento HTML personalizado
- Orientado a objetos. Cada componente de React es una clase o una función
- Definiremos un método render() (o return si es función) para cada componente que especificará cómo se visualizará el mismo. En este método render podemos combinar elementos HTML y otros componentes que hayamos definido anteriormente o que importemos de una librería.

JSX - Atributos y propiedades

- Al igual que los atributos de HTML (id, class, style, onclick), JSX nos permite definir nuestros propios atributos (props)
- Si un atributo va entre comillas, su valor será un String. Si está entre llaves, se trata de una expresión JavaScript

```
var myDivElement = <div className="foo" />;

var myElement = <SomeComponent someProperty={true} />;

var otherElement = <App age={7+10+8} />;
//App will have a prop.age that is 25 because 7+10+8 gets evaluated
```

- El operador ternario se utiliza muy a menudo

```
var person = <Person name={window.isLoggedIn ? window.name : ''} />;
```

- El valor por defecto de las props es true. Estas dos declaraciones son equivalentes:

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

Integrando expresiones en JSX

Podemos incrustar expresiones dentro de JSX envolviéndolas en llaves

```jsx
export default function Saludo(props) {
  const elem = <h1>Esto es un titular</h1>;
  const nombre = "Enrique Barra";
  const usuario = { nombre: "Enrique", apellido: "Barra", edad: "40" };

  function formatName(user) {
    return user.nombre + " " + user.apellido + " de " + user.edad + " años";
  }

  return (
    <div>
      {elem}
      <p>Hola {nombre}</p>
      <p>Hola {formatName(usuario)}</p>
    </div>
  );
}
```

https://github.com/REACT-UPM/ejemplos_simples/blob/main/src/Saludo.js

ADVERTENCIA

- Como JSX es JavaScript hay algunas palabras reservadas que no podremos usar como atributos del XML. Como "class" o "for"
  - Tendremos que usar “className” o “htmlFor” en su lugar

- React DOM utiliza la convención de nombres de propiedades camelCase en lugar de nombres de atributos HTML
  - Por ejemplo, “tabindex” se convierte en “tabIndex”, “onclick” se convierte en “onClick”, …
  - La excepción son los atributos de accesibilidad aria-* y los atributos data-*, que siguen siendo en minúscula.

JSX – estilos en línea

- El atributo **style** de HTML es un objeto de JavaScript
- Las propiedades CSS son las claves y los valores son strings
- Recuerda, las claves se escriben en camelCase:
  - background-color -> backgroundColor
  - margin-left -> marginLeft

```jsx
export default function SaludoConEstilo(props) {
  const elem = <h1>Esto es un titular</h1>;
  const nombre = "Enrique Barra";
  const usuario = { nombre: "Enrique", apellido: "Barra", edad: "40" };

  function formatName(user) {
    return user.nombre + " " + user.apellido + " de " + user.edad + " años";
  }

  const divStyle = {
    backgroundColor: "blue",
    color: "white",
    fontFamily: "Arial",
    padding: "10px",
  };

  return (
    <div style={divStyle}>
      {elem}
      <p>Hola {nombre}</p>
      <p>Hola {formatName(usuario)}</p>
    </div>
  );
}
```

https://github.com/REACT-UPM/ejemplos_simples/blob/main/src/SaludoConEstilo.js

Renderizando múltiples Components

- El método render debe devolver o bien un componente o un array de componentes

- MAL!:

```
//INCORRECTO
return (
    <p>Hola {nombre}</p>
    <p>Hola {formatName(usuario)}</p>
);
```

- BIEN:

```
return (
  <div>
    <p>Hola {nombre}</p>
    <p>Hola {formatName(usuario)}</p>
  </div>
);
```

- BIEN:

```
return [
    <p>Hola {nombre}</p>,
    <p>Hola {formatName(usuario)}</p>
  ];
```

Fragments

Los Fragments son otra alternativa para devolver múltiples componentes en el render, si no quieres envolverlos con un div o un array extra

```
return (
  <React.Fragment>
    <p>Hola {nombre}</p>
    <p>Hola {formatName(usuario)}</p>
  </React.Fragment>
);
```

```
return (
  <>
    <p>Hola {nombre}</p>
    <p>Hola {formatName(usuario)}</p>
  </>
);
```

Conditional rendering

- Se renderizan unos componentes u otros según alguna condición
- https://es.reactjs.org/docs/conditional-rendering.html

- Tenemos varias opciones, if-else, &&, operador ternario (? :)

- Veamos cada una de ellas con un ejemplo

Conditional rendering - if-else

- If-else con return dentro del if y el else
- Utilizado si el componente es muy simple

```
export default function SaludoCondicional(props) {
  const usuarioLogueado = true; // obtenido de las cookies o de una base de datos
  const nombre = "Enrique Barra";

  if(usuarioLogueado) {
    return <h1>Hola {nombre}</h1>;
  } else {
    return <h1>Hola usuario no logueado</h1>;
  }
}
```

Conditional rendering - if-else

- If-else con definición de componente dentro del if y el else
- Utilizado en componentes algo más complejos

```
export default function SaludoCondicional2(props) {
  const usuarioLogueado = true; // obtenido de las cookies o de una base de datos
  const nombre = "Enrique Barra";
  let misaludo;

  if(usuarioLogueado) {
    misaludo = <h2>Hola {nombre}</h2>;
  } else {
    misaludo = <h2>Hola usuario no logueado</h2>;
  }

  return(<div>
    <h1>Página de saludo con o sin loguear</h1>
    {misaludo}
  </div>)
}
```

Conditional rendering - operador &&

- Hacemos un if en una línea con el operador lógico &&
- Esto funciona porque en JavaScript, **true && expresión** siempre evalúa a expresión, y **false && expresión** siempre evalúa a false

```
export default function MailBox(props){
  const unreadMessages = [{ sender: "Enrique Barra", time: "17:50", content: "Hola que tal" },
  { sender: "Pepito Pérez", time: "17:51", content: "¿Vas a venir?" }];

  return (<div>
    <h1>Hola</h1>
    {unreadMessages.length>0 &&
      <h2>Tienes {unreadMessages.length} mensajes sin leer.</h2>
    }
  </div>)
}
```

Conditional rendering - operador ternario condicional

- Hacemos un if en una línea con el operador ternario condicional
- condición ? expr1 : expr2
- https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Operators/Conditional_Operator

```
export default function BotonCondicional(props){
  const añadido = false;

  return(<div>
    <button>
      {añadido ? "Quitar":"Añadir"}
    </button>
  </div>)
}
```

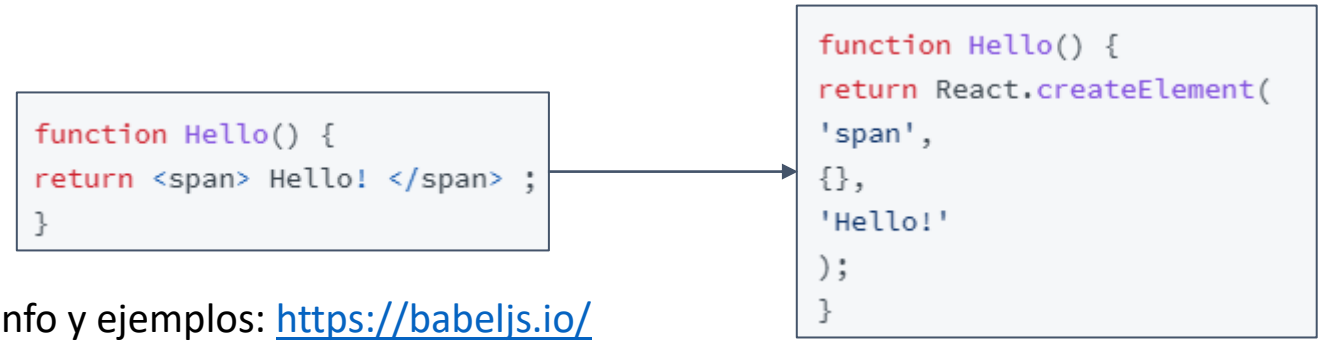
Conditional rendering - operador ternario condicional

- Hacemos un if en una línea con el operador ternario condicional
- condición ? expr1 : expr2
- https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Operators/Conditional_Operator

```
return(<div>
    {isLoggedIn ?
    <LogoutButton propiedad1={valor}/>:
    <LoginButton propiedad2={valor2}/> }
</div>)
```

Transpilar (Babel)

- Los navegadores sólo “entienden” JavaScript
  - Ni JSX
  - Ni Typescript
  - Ni ES6 ó ES10 por completo
- Por tanto, necesitamos transformar nuestro código a Javascript vanilla (estándar)
  - A esta acción se le llama ***transpilar***
- La idea es que los transpiladores/procesadores transformen el JSX en JavaScript estándar
- Para ello utilizaremos Babel (nos lo da ya configurado create-react-app)
- Babel tiene múltiples plugins (para distintas transformaciones)



- Más info y ejemplos: https://babeljs.io/