



20

S2 | Props, eventos & state

Después de esta lección podrás:

1. Compartir información entre componentes
2. Enviar información hacia otros componentes para lanzar eventos.

React props

Hasta el momento hemos unido varios componentes dentro de otro para crear un árbol de nodos dentro del HTML y el DOM. Esto ha permitido que bajo el componente `<App />` tengamos una estructura (recuerda el CV de la sesión anterior) que está formada con varios elementos (como si fuesen piezas de Lego) como por ejemplo:

```
class App extends React.Component {  
  render() {
```

```
        return (
          <Header />
          <Section />
          <Footer />
        );
    }
}
```

Pero, ¿y si quisieramos enviar información desde App a otro componente?

Para ello utilizaremos los `props`. Por ejemplo, vamos a crear información base para compartir entre componentes.

Imagina que tenemos una App que, dado un usuario que ha iniciado sesión, muestra en la Navbar la información del usuario y su perfil:

```
const user = {
  fistName: 'Juan',
  lastName: 'González',
  age: 30,
  image: 'https://bit.ly/3e7XhiJ',
  about: 'Soy una persona que siempre está aprendiendo cosas nuevas',
  education: [
    { title: 'Ingeniero civil', company: 'Universidad de Salamanca' },
    { title: 'Máster en Big Data', company: 'Universidad de Madrid' },
  ],
  experience: [
    { title: 'Junior data scientist', company: 'Upgrade Hub' },
    { title: 'Senior data scientist', company: 'Upgrade Hub Labs' },
  ],
};

class App extends React.Component {
  render() {
    return (
      <Header image={user.image} fullName={user.firstName + user.lastName} />
      <Section elements={user.education} />
      <Section elements={user.experience} />
    );
  }
}
```

Ahora cada uno de los componentes recibirá parte de la información del usuario, por ejemplo en el componente `<Header />` :

```
class Header extends React.Component {
  render() {
    return (
      <header>
        <img src={this.props.image} alt="User image" />
        <h1>{this.props.fullName}</h1>
      </header>
    );
  }
}
```

Esta información que recibimos la podremos obtener por medio de `this.props`, que será un objeto con una clave por cada campo que envíamos al componente.

Es decir, como `<Header image={user.image} />` envía la propiedad con nombre `image` al componente `<Header />` con un valor igual a `this.user.image`, tendremos un objeto `this.props` de esta manera:

```
// ¡Atención! ¡Esto es pseudocódigo no válido para tu proyecto!

// Para Header de esta forma:
<Header image={user.image} fullName={user.firstName + user.lastName} />

// En Header, tendremos this.props de esta forma:
this.props = {
  image: user.image, // https://bit.ly/3e7XhiJ
  fullName: user.firstName + user.lastName // Juan González
}
```

En las secciones, podremos extender esto a un uso más avanzado para poder reutilizar el mismo componente:

```
class Section extends React.Component {
  render() {
    return (
      <section>
        <ul>
          {this.props.elements.map(el => {
            return (
              <li key={el.title}>
                <p>Title: {el.title}</p>
                <p>Company: {el.company}</p>
              </li>
            );
          })}
        </ul>
      </section>
    );
  }
}
```

Con este componente podremos reutilizar una lista con elementos que sean objetos con las claves `title` y `company`. De esta forma, podemos utilizar `<Section />` en distintos lugares y todos se verán de una forma similar.

React PropTypes

Como puedes observar, los componentes pueden recibir diferentes props, de muchos tipos y con diferentes valores. Esto puede generar problemas en caso de confusión o en caso de que no se envíen props necesarios para el correcto funcionamiento de nuestros componentes.

Para solucionar este problema tenemos una librería con un uso muy extendido, `prop-types`.

```
npm install prop-types
```

Para utilizar esta librería, añadiremos unas nuevas propiedades a nuestros componentes, `propTypes` y `defaultProps`.

¿Qué ocurre si no enviamos un array al componente `<Section />`?

```
import React from 'react';
import PropTypes from 'prop-types';

class Section extends React.Component {
  render() {
    return (
      <section>
        <ul>
          {this.props.elements.map(el => {
            return (
              <li key={el.title}>
                <p>Title: {el.title}</p>
                <p>Company: {el.company}</p>
              </li>
            );
          })}
        </ul>
      </section>
    );
  }
}

// Explicaremos este ejemplo en clase al ser una demostración de props complejos.
Section.propTypes = {
  elements: PropTypes.arrayOf(PropTypes.shape({
    title: PropTypes.string,
    company: PropTypes.string,
  }))
}

// Este valor será el que obtenga this.props.elements si no enviamos nada.
Section.defaultProps = {
  elements: []
}
```

Estos son los tipos de PropTypes disponibles para usar y tipar:

```
optionalArray: PropTypes.array
optionalBool: PropTypes.bool
optionalFunc: PropTypes.func
optionalNumber: PropTypes.number
optionalObject: PropTypes.object
optionalString: PropTypes.string
optionalNode: PropTypes.node
optionalEnum: PropTypes.oneOf(['News', 'Photos'])
optionalArrayOf: PropTypes.arrayOf(PropTypes.number)
```

React state & eventos

Ahora que hemos visto los props y podemos compartir información hacia abajo desde un componente a otro, ¿cómo persistimos la información en el componente superior y la alteramos mediante eventos?

Para solucionar esto, utilizaremos `React.state` en nuestros componentes de clases.

Vamos a crear un contador para ver directamente como cambian los valores:

```
import React from 'react';

class App extends React.Component {
  constructor(props) {
    super(props);

    // Podemos inicializar el estado fuera del constructor
    this.state = { count: 0 }
  }
}
```

```

// Usamos arrow functions para mantener la referencia a this
add = e => {
  const newCount = this.state.count + 1;
  this.setState({
    count: newCount
  });
}

// Usamos arrow functions para mantener la referencia a this
subtract = e => {
  const newCount = this.state.count - 1;
  this.setState({
    count: newCount
  });
}

render() {
  return (
    <div>
      <h2>The count is {this.state.count}</h2>

      <button onClick={this.subtract}>Subtract one</button>
      <button onClick={this.add}>Add one</button>
    </div>
  );
}
}

```

Vamos a explicar todo lo que está ocurriendo aquí:

- A través de `constructor` estamos creando un nuevo estado en nuestro componente. Podemos aprovecharlo para crear variables propias de esta clase aparte del estado.

Podríamos inicializar el estado también sin hacer uso del constructor haciendo directamente `state = { count: 0 }` en el cuerpo de la clase.

- Los eventos de los botones serán `onClick`. En `React` todos los eventos tienen un nombre distinto al que tienen en HTML, los inputs usarán `onChange`, usaremos también `onKeyPress` o también `onKeyDown`. Hay eventos para hacer scroll mediante `onScroll` y demás.

- La razón de que las funciones de la clase estén definidas mediante arrow functions está detrás de que el uso de un eventos en React conlleva perder el contexto de la referencia `this` si están definidas las funciones de forma normal.

Es recomendable también que no utilices las arrow functions en el render, ya que esta función se invocará cada vez que cambie el estado de nuestro componente o cambien sus props. En cambio, si hemos declarado la función dentro del cuerpo de la clase, esto ocurrirá solo cuando la instanciamos por primera vez.

Ejemplo sobre un objeto complejo

Volvemos al primer ejemplo, donde definimos el objeto `user` pero ahora lo declararemos en el estado de la siguiente forma:

```
class App extends React.Component {
  // Declaramos el state sin necesidad de constructor
  state = {
    user: {
      fistName: 'Juan',
      lastName: 'González',
      age: 30,
      image: 'https://bit.ly/3e7XhiJ',
      about: 'Soy una persona que siempre está aprendiendo cosas nuevas',
      education: [
        { title: 'Ingeniero civil', company: 'Universidad de Salamanca' },
        { title: 'Máster en Big Data', company: 'Universidad de Madrid' },
      ],
      experience: [
        { title: 'Junior data scientist', company: 'Upgrade Hub' },
        { title: 'Senior data scientist', company: 'Upgrade Hub Labs' },
      ],
    }
  }

  render() {
```

```

// Obtenemos los valores necesarios del estado aquí (no hacer lógica excesiva)
const { user } = this.state;

return (
  <Header image={user.image} fullName={user.firstName + user.lastName} />
  <Section elements={user.education} />
  <Section elements={user.experience} />
);
}

```

Tenemos exactamente el mismo resultado que antes pero utilizando `state`. ¿Qué ventajas ofrece este caso?

Vamos a crea un pequeño botón que permita comenzar el modo edición de nuestro usuario dentro del render:

```

return (
  <Header image={user.image} fullName={user.firstName + user.lastName} />
  <Section elements={user.education} />
  <Section elements={user.experience} />

  <button onClick={setIsEditing}>Editar perfil</button>
);

```

La función la declararemos de la siguiente forma (modificaremos también el state):

```

state = {
  isEditing: false,
  user: { ... } // Mantenemos el user anterior
}

setIsEditing = e => {
  const newIsEditing = !this.state.isEditing; // Invertimos isEditing
  this.setState({
    isEditing: newIsEditing
  })
}

```

```
    })  
}
```

Ahora podemos renderizar un formulario de edición para cambiar campos del usuario en caso de que queramos:

```
return (  
  {this.state.isEditing  
  ? <Form user={user} />  
  : (  
    <Header image={user.image} fullName={user.firstName + user.lastName} />  
    <Section elements={user.education} />  
    <Section elements={user.experience} />  
  )}  
  <button onClick={this.setIsEditing}>Editar perfil</button>  
)
```

Ahora verás que si hacemos click sobre el botón podemos cambiar entre el componente Form y el contenido original que teníamos. Vamos a crear el componente form:

```
class Form extends React.Component {  
  // Cambiamos el campo de user con el evento  
  handleChangeName = e => {  
    handleChangeUserValue('firstName', e.target.value)  
  }  
  
  // Desactivamos el submit del formulario para evitar recarga de la web  
  handleSubmit(e) {  
    e.preventDefault();  
  }  
  
  render() {  
    const { user, handleChangeUserValue } = this.props;  
  
    return (  
      <form onSubmit={this.handleSubmit}>  
        <label>  
          <p>Nombre de usuario</p>
```

```

        <input type="text" value={user.firstName} onChange={this.handleChangeName} />
      </label>
    </form>
  );
}
}

```

Ahora solamente nos falta añadir la función `handleChangeUserValue` a `App` para que el evento se transmita de abajo hacia arriba:

```

class App extends React.Component {
  // Declaramos el state sin necesidad de constructor
  state = {
    isEditing: false,
    user: {
      fistName: 'Juan',
      lastName: 'González',
      age: 30,
      image: 'https://bit.ly/3e7XhiJ',
      about: 'Soy una persona que siempre está aprendiendo cosas nuevas',
      education: [
        { title: 'Ingeniero civil', company: 'Universidad de Salamanca' },
        { title: 'Máster en Big Data', company: 'Universidad de Madrid' },
      ],
      experience: [
        { title: 'Junior data scientist', company: 'Upgrade Hub' },
        { title: 'Senior data scientist', company: 'Upgrade Hub Labs' },
      ],
    }
  }

  // Cambiamos los campos del usuario uno a uno
  handleChangeUserValue = (fieldName, value) => {
    const newUser = { ...this.state.user, [fieldName]: value };

    this.setState({
      user: newUser,
    })
  }

  setIsEditing = e => {
    const newIsEditing = !this.state.isEditing; // Invertimos isEditing
    this.setState({
      isEditing: newIsEditing
    })
  }
}

```

```
render() {
  // Obtenemos los valores necesarios del estado aquí (no hacer lógica excesiva)
  const { user } = this.state;

  return (
    {this.state.isEditing
      ? <Form user={user} handleChangeUserValue={this.handleChangeUserValue} />
      : (
        <Header image={user.image} fullName={user.firstName + user.lastName} />
        <Section elements={user.education} />
        <Section elements={user.experience} />
      )}
    <button onClick={this.setIsEditing}>Editar perfil</button>
  );
}
```

Con este conocimiento, podemos avanzar y crear diferentes eventos para cada input del usuario y así cambiar valores de forma dinámica. ¡Atrévete a cambiar el array de valores! 