Xseed

Curso de React _ Technisys

Xseed

Clase 05 -

PropTypes, DefaultProps, Forms en React

by Diego Cáceres

Repaso

Webpack

Webpack es un sistema de bundling que prepara el código desarrollado en una aplicación para *producción*.

Se puede considerar como un *Browserify* avanzado ya que tiene muchas opciones de configuración. También es similar a Grunt y Gulp, ya que permite de alguna manera automatizar los procesos principales que son transpilar y preprocesar código de `.scss` a `.css`, de ES7 a ES5/6, etc.

Webpack

En resumen, Webpack necesita saber:

1. Punto de entrada de la aplicación.
2. Transformaciones a realizar al código.
3. Ubicación donde guardar el código transformado.

Esto es lo que se escribe en el archivo de configuración de Webpack, y aunque al principio el código de este archivo parezca muy complicado, no es más que indicar los tres datos mencionados.

Ejercicio 1 (7)

Ahora se creará un archivo de configuración de Webpack.
Crear un archivo `webpack.config.js`.

Hay que asegurarse de exportar un objeto desde el archivo, conteniendo la configuración:

```
// En webpack.config.js
module.exports = {}
```

Luego hay que indicarle el primero de los 3 pasos, es decir, cual es el archivo de ingreso al proyecto:

```
// En webpack.config.js
module.exports = {
   entry: './index.js',
}
```

Ejercicio 1 (9)

Cada regla que se agrega tiene dos partes; la primera indica sobre qué archivos aplicarla y la segunda indica qué *loader* utilizar sobre esos archivos:

```
module.exports = {
   entry: './index.js',
   module: {
       rules: [
           { test: /\.js$/,use: "babel-loader" },
       ]
   },
}
```

En este caso se está indicando que en todos los archivos JavaScript, aplique el `babel-loader`, que es uno de los instalados previamente por `npm`.

Ejercicio 1 (12)

El último paso es indicar en qué ruta dejar el código transformado. Para esto se agrega la propiedad `output` (hay que hacer un `require` del path, que ya está instalado en node, para trabajar con el file system):

```
var path = require("path");
module.exports = {
   entry: './index.js',
   module: {
       rules: [
           { test: /\.js$/, use: "babel-loader" },
           { test: /\.css$/, use: ["style-loader", "css-loader"] }
       ]
   },
   output: {
       path: path.resolve(__dirname, 'dist'),
       filename: 'index_bundle.js'
   }
```

Lists – `map` (2)

Si se quiere crear una lista de elementos en la UI en React, se puede simplemente utilizar `map` y retornar elementos en la función. Por ejemplo, el siguiente componente recibe por *`props`* una lista de nombres de amigos:

```
const ShowList = (props) => {
   return (
       <div>
           <h3>Amigos:</h3>
           <ul>
               {props.names.map((friend) => {
                   return <li> {friend} </li>;
               })}
           </ul>
       </div>
   );
}
```

Lists – `key` (1)

Warning: Each child in an array or iterator should have a unique "key" prop

- Este es un *warning* que se verá siempre que un componente de React renderice una lista de elementos y no se le asigne una `key` propia a cada uno de ellos. Esto es un requerimiento de React para poder ser más eficiente a la hora de actualizar la UI.
- React utiliza esta `key` para identificar exactamente qué elemento fue modificado, eliminado o agregado en una lista de elementos, sin necesidad de volver a renderizar toda la lista.
- Esta `key` debe ser única entre los elementos de la lista. Si se cuenta con un `id` claramente es la mejor opción, pero si no, se puede utilizar el índice del elemento en el array como último recurso.

Lists – `key` (3)

Las `key` sólo son necesarias entre los "hermanos" de la lista que se renderiza. Es decir, si se extrae un componente `ListItem`, el `key` pasaría del `<li>` al `<ListItem>`.

```
function ListItem(props) {
    return <li>{props.value}</li>;
}
function NumberList(props) {
    const numbers = props.numbers;
    const listItems = numbers.map((number, index) =>
        <ListItem key={index} value={number} />
    );
    return (
        <ul>{listItems}</ul>
    );
}
```

Props.children (1)

Cuando las expresiones JSX tienen tag de apertura y de cierre, el contenido entre ellas es enviado al componente como una propiedad especial, que se accede mediante `props.children`.

Estos "hijos" pueden ser de distintos tipos.

En el siguiente caso, `props.children` será simplemente un string:

```
<MyComponent>Hello world!</MyComponent>
```

Props.children (2)

En realidad, `props.children` es un array, lo cual es útil para poder brindar varios elementos como hijos.

Esto permite crear componentes que agreguen comportamiento a la aplicación y simplemente envuelvan al resto de los componentes. Pero para esto el "container" debe explícitamente renderizar a sus hijos, de la siguiente forma:

```
<MyContainer>
    <MyFirstComponent />
    <MySecondComponent />
</MyContainer>
```

```
class MyContainer extends Component {
    render() {
        return <div>{this.props.children}</div>
    }
}
```

PropTypes

PropTypes (1)

```
class Users extends Component {
   render() {
     return (
       <ul>
          {this.props.list.map((friend) => {
                    return <li>{friend}</li>
          })}
       </ul>
     )
   }
}
```

Este componente espera una lista de nombres y realiza un `map` para mostrarlos como una lista no ordenada (`<ul>`).

¿Qué pasaría si en vez de pasarle una lista en la **prop** `list`, se le pasa un string?

```
<Users list="Diego, Juan, Pedro" />
```

PropTypes (2)

Las PropTypes permiten declarar el tipo (string, número, función, etc) de cada ***prop*** que se pasa a un componente.

Si se pasa una ***prop*** que no es del tipo declarado, se verá una advertencia en la consola. Obviamente que la utilidad es durante el desarrollo de una aplicación, pero es muy útil para trabajar en equipos con más de un desarrollador o simplemente para escribir componentes más "seguros".

Nota: Desde React v15.5 Prop Types se movió a un paquete separado, por lo que para poder utilizarlo es necesario instalarlo por `npm` o `yarn`:

- `npm install --save 'prop-types'`
- `yarn add 'prop-types'`

PropTypes (3)

El ejemplo visto anteriormente, declarando las PropTypes sería así:

```
import React, { Component } from "react";
import PropTypes from 'prop-types';
class Users extends Component {
   render() {
     return (
       <ul> {this.props.list.map((friend) => {
           return <li>{friend}</li>
         })} </ul>
     )
   }
}
Users.propTypes = {
   list: PropTypes.array.isRequired
}
```

PropTypes (4)

En el caso de definir los PropTypes, y cometer un error al utilizar un componente, se mostrará un *warning* en la consola como el siguiente:

```
▶Warning: Failed prop type: Invalid prop `user` of type `string` supplied to `Avatar`, expected `object`.
    in Avatar (at UserInfo.js:8)
    in UserInfo (at Comment.js:8)
    in div (at Comment.js:7)
    in Comment (at App.js:44)
    in div (at App.js:42)
    in div (at App.js:37)
    in App (at index.js:7)
```

PropTypes (5)

La **API** completa se puede encontrar en https://github.com/facebook/prop-types.

Por defecto, siempre son opcionales. Estos son los tipos básicos, notar que las **funciones** y los **booleanos** son especiales.

```
MyComponent.propTypes = {
  optionalArray: PropTypes.array,
  optionalBool: PropTypes.bool,
  optionalFunc: PropTypes.func,
  optionalNumber: PropTypes.number,
  optionalObject: PropTypes.object,
  optionalString: PropTypes.string,
  optionalSymbol: PropTypes.symbol,
```

PropTypes (6) – Más usos

```
MyComponent.propTypes = {
  // Se puede agregar que la prop sea requerida
  requiredFunc: PropTypes.func.isRequired,
  // Un Elemento de React
  optionalElement: PropTypes.element,
  // Se puede declarar que una prop es una instancia de una clase
  optionalMessage: PropTypes.instanceOf(Message),
  // Se puede limitar que la prop esté limitada a valores especificados, como un enum
  optionalEnum: PropTypes.oneOf(['News', 'Photos']),
```

PropTypes (7) – Más usos

```
MyComponent.propTypes = {
   // Un objeto de uno de varios tipos
   optionalUnion: PropTypes.oneOfType([
       PropTypes.string,
       PropTypes.number,
       PropTypes.instanceOf(Message)
   ]),
   // Un objeto con determinada forma
   optionalObjectWithShape: PropTypes.shape({
       color: PropTypes.string,
       fontSize: PropTypes.number
   }),
   // Un array de un tipo de datos
   optionalArrayOf: PropTypes.arrayOf(PropTypes.number),
```

Default Prop Values

Default Prop Values

Se pueden definir valores por defecto para las **Props**, para el caso en que no sean especificadas, utilizando `defaultProps`.

El chequeo de los tipos definidos en las PropTypes sucede luego de interpretar las Default Props, por lo que también validará la estructura y tipo de estos valores contra las reglas definidas.

Default Prop Values – Ejemplo

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

// El valor por defecto:
Greeting.defaultProps = {
   name: 'Stranger'
};
Greeting.propTypes = {
   name: PropTypes.string
};
```

Xseed

Forms en React

Forms

En React, los Forms son un poco distintos a los demás elementos del DOM, ya que en sí los Form mantienen un estado interno.

En el siguiente ejemplo, se mantiene el valor del dato ‘name’ en el estado interno.

```
<form>
    <label>
        Name:
        <input type="text" name="name" />
    </label>
    <input type="submit" value="Submit" />
</form>
```

Si se quisiera lograr el comportamiento normal de los Form en HTML, de navegar a una nueva página al hacer `submit`, se puede tenerlo, pero lo normal es querer invocar una función en el `submit`, que acceda a los datos ingresados y se encargue de hacer el `submit`.

Forms – Controlled Components (1)

Para lograr esto en React se utiliza una técnica llamada Controlled Components.

Normalmente en HTML los elementos de formulario como `<input>`, `<textarea>` y `<select>` mantienen su propio estado y lo actualizan en función al `input` del usuario.

Esta técnica consiste básicamente en mantener al **state** de React como la fuente única de verdad, por lo que es React quien se encarga de mantener el estado de estos elementos, y además de actualizarlo en función al `input` del usuario.

En la siguiente slide se verá el código anterior pero aplicando estos conceptos.

Forms – Controlled Components (2)

```
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>
        );
    }
}
```

Forms – Text Area

En HTML un elemento `<textarea>` define su contenido mediante sus hijos.

```
<textarea>
    Texto en un textarea, como hijo el elemento html
</textarea>
```

En React, se utiliza el atributo `value` (al igual que con un `input`):

```
<form onSubmit={this.handleSubmit}>
    <label>
        Name:
        <textarea value={this.state.value} onChange={this.handleChange} />
    </label>
    <input type="submit" value="Submit" />
</form>
```

Forms – Select

En HTML para indicar la opción seleccionada en un `select` se puede utilizar el atributo `selected` dentro de un `option`.

```
<select>
    <option value="grapefruit">Grapefruit</option>
    <option value="lime">Lime</option>
    <option selected value="coconut">Coconut</option>
    <option value="mango">Mango</option>
</select>
```

En React, en vez de esto, se utiliza el `value` en el tag raíz `select`, de forma de que solo haya que actualizar el valor seleccionado en un lugar. A continuación, en las próximas slides, se muestra un ejemplo.

Forms – Select (ej. Parte 1)

```
// PARTE 1
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();
    }
```

Forms – Select (ej. Parte 2)

```
    // PARTE 2
    render() {
    return (
        <form onSubmit={this.handleSubmit}>
            <label>
                Pick your favorite Icecream 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>
        );
    }
}
```

Forms – Manejando múltiples inputs (1)

En el caso de tener que manejar varios inputs en un componente, no es necesario crear un *handler* para cada uno de ellos, sino que se puede aprovechar el atributo `name`, asignándole un `name` distinto a cada elemento, y luego en el *handler* decidir qué actualizar en función del `event.target.name`.

```
handleInputChange(event) {
    const target = event.target;
    const value = target.type === 'checkbox' ? target.checked : target.value;
    const name = target.name;

    this.setState({
        [name]: value
    });
}
```

¡Computed Properties de ES6!

Forms – Manejando múltiples inputs (2)

```
class Reservation extends React.Component {
    constructor(props) {
        super(props);
        this.state = {  isGoing: true,  numberOfGuests: 2 };
        this.handleInputChange = this.handleInputChange.bind(this);
    }
    render() {
        return (
            <form>
                <label>
                    Is going: <input name="isGoing" type="checkbox" checked={this.state.isGoing}
                        onChange={this.handleInputChange} />
                </label><br/>
                <label>
                    Number of guests: <input name="numberOfGuests" type="number"
                        value={this.state.numberOfGuests} onChange={this.handleInputChange} />
                </label>
            </form>
        );
    }
```

El valor de name coincide con la clave del state

Ejercicios

Ejercicios

Descargar el zip `EjerciciosClase05`. Cada carpeta dentro corresponde a un ejercicio y es un proyecto creado con Create React App, por lo que para ejecutarlo es necesario:

- Si usan yarn:
  - `$ yarn install`
  - `$ yarn start`
- Si usan npm:
  - `$ npm install`
  - `$ npm start`

Se debería abrir automáticamente el explorador con el ejercicio corriendo en http://localhost:3000/

Ejercicio 1

- Editar el archivo `src/App.js` y realizar un refactoreo para mover el componente `UserInfo` y `Comment` a sus correspondientes archivos dentro de la carpeta `src/components`. Además, agregar los `imports` necesarios para que todo siga funcionando. Esto permitirá que el proyecto (y futuros proyectos) queden mejor organizados.
- Instalar el paquete de `prop-types`.
- Agregar la definición de **PropTypes** necesarios para cada componente: `Avatar`, `UserInfo` y `Comment`. (No olvidarse de importar el paquete instalado en los componentes).
- Agregar la definición de **PropTypes** en `App.js` declarando que debe recibir un array, de tipo objeto, con la forma necesaria.
- Agregar `DefaultProps` a `Avatar` y a `UserInfo`.

Ejercicio 2

- Editar el archivo `src/components/InputForm.js`
- Primero asegurarse que la **etiqueta** y el **placeholder** muestren los datos recibidos por **Props**.
- Modificar el componente para convertirlo en un "Controlled Component" que maneje el valor del `input`.
- Cuando se presione el botón, debe haber un `alert` con el contenido del `input`.
- El botón sólo puede funcionar cuando el `input` no es vacío.
- Agregar las **PropTypes** y **DefaultProps** que correspondan.

Ejercicio 3 (1)

En este ejercicio se implementa un sitio para poder ver dos usuarios de Github. Ya se cuenta con `InputForm`, casi igual al del ejercicio anterior, sólo que también se le agrega que reciba un `id`.

- Hay que completar `InputForm` para que reciba además una función `onSubmit` que invocará en el `handleSubmit`, pasándole tanto el `id` como el `value` ingresado. Esta función la recibirá de su padre: `GithubProfiles`.
- Luego, se deberá completar el componente `Profile`, declarando en las Prop Types que reciba un `Avatar` (para el `src` de la imagen), un `username` para mostrar, un `id` y una función `onReset` para invocar con el `id` cuando se presione el botón reset.

Ejercicio 3 (2)

- Implementar en `GithubProfiles` que maneje un *state* con 4 valores: el username de cada perfil, y la imagen de cada perfil y completar el `handleSubmit` y el `handleReset` para que actualicen el estado.
- Revisar que los elementos de `InputForm` y `Profile` del render de `GithubProfiles` estén recibiendo todas las props necesarias.
- Finalmente, utilizar Conditional Rendering para no mostrar el perfil cuando aún no se completó el formulario, y no mostrar el formulario si ya fue completado, hasta que no presione reset.