Hacé click acá para dejar tu feedback sobre esta clase.

Hacé click acá completar el quiz teórico de esta lecture.

React Estado LifeCycle

La mejor manera: Webpack

Cuando empiezas a trabajar en un proyecto la manera anterior de incluir React no es la mejor, de hecho no puede escalar. Cuando tenemos muchas lineas de código en un sólo archivo, o muchos archivos chicos de Js, el hecho de tener que juntar todo se empieza a complicar. Por suerte, desde hace varios años existen herramientas que nos van a automatizar este proceso, haciendo que todo el workflow sea óptimo y sobre todo mantenible.

Algunas tareas de las que se encargan estos gestores de proceso pueden ser:

- Juntar código de varios archivos en uno sólo.
- Transpilar código. Por ejemplo, de CoffeeScript, o TypeScript a Javascript.
- Minificar código.
- Concatenar archivos.
- Correr los tests automáticamente.
- etc...

> Existen varios gestores de procesos buenos y populares, los más usados son: Grunt, Gulp! y Webpack. Nosotros vamos a usar *Webpack*, pero podrían hacer lo mismo con los otros.

Ahora también se utiliza el concepto de **módulos** en el frontend, para lograrlo se utilizan librerías como Browserify o CommonJS. Básicamente en vez de incluir librerías usando HTML, lo hacemos en el mismo JS, después estas librerías que mencionamos se encarga de incluir realmente el código necesario para que funcione.

Un ejemplo en el frontend se vería algo así:

```
//algunModulo.js
module.exports.doSomething = function() {
  return 'foo';
};
//algunOtroModulo.js
const someModule = require('someModule');
module.exports.doSomething = function() {
  return someModule.doSomething() + 'bar';
};
```

Vamos a empezar instalando y configurando Webpack. Voy a aclarar al principio que Webpack es una herramienta muy poderosa, por ende compleja, y lamentablemente su documentación no es la mejor. Por lo tanto, nos va a parecer complejo al principio, pero rápidamente nos vamos a encariñar con todas las cosas que podemos hacer con Webpack.

```
npm i -D webpack webpack-cli
```

Como dijimos, Webpack es una herramienta que va a aplicar ciertas *transformaciones* a nuestro código, por ende para funcionar webpack necesita saber:

1. Conocer el starting point de nuestra app, o el archivo javascript raíz.
2. Debe saber qué transformaciones tiene que hacer al código.
3. Tiene que saber dónde guardar el nuevo código transformado.

Todo esta información va a estar contenida en un archivo de configuración llamado `webpack.config.js`, que deberíamos crear en la raíz del directorio de nuestro proyecto. Este archivo va a ser en realidad un módulo, que va a exportar un objeto con las configuraciones de webpack, asi que podríamos empezar escribiendo lo siguiente en ese archivo:

```
// dentro de webpack.config.js
module.exports = {}
```

Ahora empezemos a agregar la información que mencionamos antes, empezemos por el punto 1 : el entry point.

```
module.exports = {
  entry: [
    './app/index.js'
  ]
}
```

Como ven, los entry points se definen dentro del objeto que exportamos bajo el nombre `entry`, y cuyo valor es un arreglo. Dentro de este explicito los paths de todos los archivos que sirvan como entry points de

nuestra app. Por ahora vamos a escribir sólo uno.

Bien, ahora para el segundo punto, tenemos que definir qué tipo de transformaciones vamos a hacer, para esto entran en juego los `loaders`, estos son los módulos encargados de realizar tranformaciones, existen varios tipos de `loaders`, ya que la comunidad va creando nuevos a medida que surgen nuevas necesidades.

> Para usar un loader, es necesario tenerlo instalado antes. Para eso vamos a usar `npm`. Por ejemplo, si quiero usar el `loader` de babel debería hacer: `npm i -D @babel/core @babel/preset-env @babel/preset-react babel-loader`.

```
module.exports = {
  entry: [
    './app/index.js'
  ],
  module: {
    loaders: [
      {test: /\.coffee$/, exclude: /node_modules/, loader: "coffee-loader"}
    ]
  },
}
```

En el ejemplo, usamos un `loader` de *coffeeScript*. Como se ve, también agreamos una propiedad llamada `module` que será un objeto dentro del cual aparecerá la propiedad `loaders` que será un arreglo de objetos. Cada objeto dentro de este arreglo representa una transformacion. Vemos que ese objeto tiene tres propiedades: `test`, `exclude`, y `loader`. La primera hace referencia a qué archivos deberán pasar por la transformación, y recibe como valor una **expresión regular**, en nuestro ejemplo estamos diciendo que pasarán por la transformación todos los archivos terminados en `.coffee`. La segunda, `exclude` le indica a webpack qué directorios excluir, en nuestro ejemplo (y siempre lo haremos) excluimos `node_modules`, donde sabemos que no habrá código para transformar. Finalmente, en la propiedad `loader` vamos a poner el nombre del loader que queremos usar, en este caso el nombre es "coffee-loader".

> Siempre busquen los loaders que necesiten dentro del ecosistema npm.

Por último, vamos a agregar donde queremos que webpack deposite los archivos luego de la transformación:

```
module.exports = {
  entry: [
    './app/index.js'
  ],
  module: {
    loaders: [
      {test: /\.coffee$/, exclude: /node_modules/, loader: "coffee-loader"}
    ]
  },
  output: {
    path: __dirname + '/dist',
    filename: 'bundle.js'
  },
}
```

Ya podrán imaginarse que quieren decir las cosas que hemos agregado ahora. Por empezar una nueva propiedad `output` que contiene un objeto, en este último vamos a especificar el nombre del archivo de salida (`filename`) y la carpeta donde queremos que se guarde (`path`).

Bien, entonces intentemos reproducir el mismo ejemplo que hicimos en el HTML, pero ahora usando este proceso. Por lo tanto, lo primero que hacemos es sacar el código escrito en React que estaba embebido en el HTML y lo pasamos a un archivo `js`. El primer cambio que tenemos que hacer es importar los módulos `react` y `react-dom` que antes requeríamos a través del tag `script`:

```
import React from 'react';
import ReactDOM from 'react-dom';

class HelloWorld extends React.Component {
  render(){
    return (
      <div>
        Hola, Soy Henry!!
      </div>
    )
  }
};

function HelloWorldFunction() {
  return(
    <div>
      Hola, Soy Henry!
    </div>
  )
};
ReactDOM.render(<HelloWorld />, document.getElementById('app'));
```

Genial, ahora tenemos que construir el archivos de configuración de webpack, para que funcione con `babel`. Básicamente tenemos que transformar el código que usa EcmaScript6 y JSX a JS plano.

```
// webpack.config.js

module.exports = {
  entry: [
    './app/index.js'
  ],
  output: {
    path: __dirname + '/dist',
    filename: 'bundle.js'
  },
  module: {
    rules: [
      {
        test: /\.(js|jsx)$/,
```

```
        exclude: /node_modules/,
        use: {
          loader: 'babel-loader',
          options: {
            presets: ['@babel/preset-react', '@babel/preset-env']
          }
        }
      }
    ]
  }
}
```

Por último vamos a tener que usar `npm` para instalar las dependencias:

```
npm install -D @babel/core @babel/preset-env @babel/preset-react babel-loader
```

```
npm install react react-dom --save
```

Para poder ejecutar webpack, debemos agregar dentro de `scripts` en nuestro `package.json` lo siguiente:

```
"scripts": {
    "build": "webpack -w",
}
"devDependencies": {
    "@babel/core": "^7.9.0",
    "@babel/preset-env": "^7.9.0",
    "@babel/preset-react": "^7.9.4",
    "babel-loader": "^8.1.0",
    "webpack": "^4.42.1",
    "webpack-cli": "^3.3.11"
  },
  "dependencies": {
    "react": "^16.13.1",
    "react-dom": "^16.13.1"
  }
```

Para probar si todo funciona bien, iremos a la carpeta donde tenemos definidos todos estos archivos, y vamos a escribir `npm run build`.

 Webpack

Si todo funcionó bien, veremos un mensaje como el de la imagen! Y además encontraremos un archivo nuevo en la carpeta `dist`.

Bien, ahora por último tenemos que agregar ese archivo generado a un HTML para poder correrlo:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Descubre React</title>
  </head>
  <body>
    <div id="app"></div>
    <script src="./dist/index_bundle.js"></script>
  </body>
</html>
```

Si abren este archivo van a ver que vemos exactamente lo mismo que en el primer archivo HTMl que creamos. La ventaja de esta forma, es que tenemos separado el código JS de todo lo demás, podemos tener múltiples archivos y Webpack se encargará de unirlos y depositarlos en el archivo de salida.

Bien, ahora que lo hicimos funcionar y más o menos vimos cómo se escribe, aprendamos un poco más sobre React.

Introduccion a Hooks

Los Hooks en React fueron introducidos en su version 16.8, nos permiten el uso de estados y otras caracteristicas sin el uso de clases. Estas además de dificultar la reutilización y organización del código, pueden ser una gran barrera para el aprendizaje de React. Se tiene que entender como funciona el 'this' en JavaScript, que es muy diferente a cómo funciona en la mayoría de los lenguajes. Agregar bind a tus manejadores de eventos hace que,por lo general, el código sea muy verboso. Para resolver estos problemas, Hooks te permiten usar más de las funciones de React sin clases. Conceptualmente, los componentes de React siempre han estado más cerca de las funciones. Los Hooks abarcan funciones, pero sin sacrificar el espíritu práctico de React.

Estado de un Componente

Dijimos que las *props* era la *configuración inicial* del Componente y que no se pueden cambiar, que son *inmutables*. Van a existir muchos casos donde un Componente mantenga adentro suyo algún dato que pueda cambiar con el tiempo, como por ejemplo lo que queremos hacer ahora de cambiar el nombre del saludo. Para hacer esto, cada Componente es capaz de mantener datos guardados en lo que React llama el *Estado* de un Componente. Un Componente por lo tanto va a poder actualizar su propio *Estado* sin restricciones. Para empezar, comencemos dandole un *Estado* inicial al Componente, lo hacemos definiendo dentro del constructor de la clase el `this.state` que va a ser un objeto con todas las propiedades que queramos almacenar como estado interno. Cada propiedad puede almacenar cualquier tipo de dato que quedamos. Para nuestro ejemplo, vamos a darle como *Estado* inicial un objeto que contenga la propiedad `name` y que sea igual a `this.props.name`, o sea que vamos a hacer que una *prop* sea un *estado*, esto es así porque sabemos que va a cambiar en el futuro.

Genial, ya tenemos *Estado*, ahora lo único que nos falta es *cambiar* el *Estado* cuando el usuario haga click. Para hacerlo vamos a tener que usar una función llamada `setState`. No podemos simplemente asignarle un valor nuevo a `this.state.name`, esto es así por matener la arquitectura del Virtual DOM de react. Veamos cómo quedaría el ejemplo:

```
import React from 'react';
import ReactDOM from 'react-dom';

class HelloWorld extends React.Component {
  constructor(props) {
    super(props);
    this.state = {name: this.props.name}
    this.onButtonClick = this.onButtonClick.bind(this);
  }
  onButtonClick(e){
    e.preventDefault();
    const name = this.refs.name.value;
    this.setState({
      name: name
    });
  }
  render(){
    return (
      <div>
        <form onSubmit={this.onButtonClick}>
          <input type='text' ref='name' />
          <button>Poner Nombre</button>
        </form>
        Hola, {this.state.name}!!
      </div>
    )
  }
};
ReactDOM.render(<HelloWorld name='Soy Henry' />, document.getElementById('app'));
```

Como vemos, ahora tenemo un Componente que maneja un *Estado* interno, que se inicializa usando una *prop* y que está pensado en cambiar en el futuro. Siguiendo el ejemplo usando un componente de funcion:

```
import React, { useState, useRef } from 'react';
import ReactDOM from 'react-dom';

function HelloWorld(props) {
  const nameState = useState(props.name)
  const textInput = useRef(null);

  const onButtonClick = (e) => {
    e.preventDefault();
    const name = textInput.current.value
    nameState[1](name)
  }
  return (
    <div>
      <form onSubmit={onButtonClick}>
        <input type='text' ref={textInput} />
        <button>Poner Nombre</button>
```

```
        </form>
        Hola, {nameState[0]}!!
      </div>
    )
};
ReactDOM.render(<HelloWorld name='Soy Henry' />, document.getElementById('app'));
```

Aca vemos el uso de el Hook `useState`. Lo llamamos dentro de un componente funcional para agregarle un estado local. React mantendrá este estado entre re-renderizados. useState devuelve un array con 2 elementos: el valor de estado actual y una función que le permite actualizarlo. Puedes llamar a esta función desde un controlador de eventos o desde otro lugar. Es similar a this.setState en una clase, excepto que no combina el estado antiguo y el nuevo. El único argumento para useState es el estado inicial. En el ejemplo anterior, es 'props.name'. Ten en cuenta que a diferencia de this.state, el estado aquí no tiene que ser un objeto — aunque puede serlo si quisieras. El argumento de estado inicial solo se usa durante el primer renderizado. Ahora veremos el mismo ejemplo escribiendo `useState` de una forma mas entendible, usando `array destructuring`. Para darle el primer valor del array nuestro state inicial y al segundo valor sera nuestra funcion para actualizar el estado.

```
import React, { useState, useRef } from 'react';
import ReactDOM from 'react-dom';

function HelloWorld(props) {
  const [name, setName] = useState(props.name)
  const textInput = useRef(null);

  const onButtonClick = (e) => {
    e.preventDefault();
    const name = textInput.current.value
    setName(name)
  }
  return (
    <div>
      <form onSubmit={onButtonClick}>
        <input type='text' ref={textInput} />
        <button>Poner Nombre</button>
      </form>
      Hola, {name}!!
    </div>
  )
};
ReactDOM.render(<HelloWorld name='Soy Henry' />, document.getElementById('app'));
```

Anidando Componentes

Como dijimos antes, en React *todo* es un copomente, por lo tanto es lógico pensar que vamos a tener *componentes dentro de componentes* todo el tiempo. Veamos con un ejemplo como funciona esto, y como se le pueden pasar datos (en React le decimos *props*) a los componentes.

Ahora armemos un ejemplo un poco más complejo, en este vamos a tener dos componentes. Uno va a llamar al otro y le va a pasar algunas propiedades, veamos como hacerlo:

```
class ContenedorAmigos extends React.Component {
  render(){
    const name = 'Soy Henry';
    const amigos = ['Toni', 'Franco', 'Emi', 'Solano'];
    return (
      <div>
        <h3> Nombre: {name} </h3>
        <MostrarLista names={amigos} />
      </div>
    );
  }
};
```

En este componente hemos definido un método `render` un poco más complejo, en el tenemos dos variables (`name` y `amigos`) y retornamos un XML que utiliza estas dos variables. Como ven, podemos acceder a un *prop* del mismo Componente usando los `{}`, de esta forma `{name}` va a ser reemplazado por `Soy Henry`. Luego llamamos a un componente que todavía no hemos definido con el nombre de `mostrarLista` y le pasamos como propiedad el arreglo `amigos`. Por lo tanto dentro de `mostrarLista` vamos a disponer de ese arreglo como una *prop*.

Definamos el elemento hijo o *child*:

```
class MostrarLista extends React.Component {
  render(){
    const lista = this.props.names.map(amigo => <li> {amigo} </li>);
    return (
      <div>
        <h3> Amigos </h3>
        <ul>
          {lista}
        </ul>
      </div>
    )
  }
};
```

Lo primero que notamos es que usamos *JS* para crear elementos HTMl más complejos. En esta caso usamos la función `map`, para crear un elemento `<li>` por cada *amigo* en la lista o arreglo. Viendo el ejemplo anterior usando funciones:

```
function ContenedorAmigos() {
  const name = 'Soy Henry';
  const amigos = ['Santi', 'Guille', 'Facu', 'Solano'];
  return (
```

```
      <div>
        <h3> Nombre: {name} </h3>
        <MostrarLista names={amigos} />
      </div>
    )
  };

  function MostrasLista({ names }) {
    const lista = names.map(amigo => <li> {amigo} </li>);
    return (
      <div>
        <h3> Amigos </h3>
        <ul>
          {lista}
        </ul>
      </div>
    )
  };
```

Aca podemos usar `destructuring` para pasar las props directamente con el nombre de la variable `names`

> Por si no se acuerdan como funciona map:

```
const amigos = ['Santi', 'Guille', 'Facu', 'Solano'];
const lista = amigos.map(amigo => "<li> " + amigo + "</li>");
console.log(lista); //['<li> Santi</li>', '<li> Guille</li>', '<li> Facu</li>',
'<li> Solano</li>'];
```

Justamente ese nuevo conjunto de `li`s que hemos creado, lo vamos a usar envuelto en tags `<ul>` para formar la lista de amigos.

Etiquetas HTML y Componentes

Los componentes que creamos en React despues los usamos escribiendolos como un tag HTML, en realidad es un tag XML. Por ejemplo: el tag `MostrarLista` es un Componente que creamos antes y lo usamos así:

```
<div>
  <h3> Nombre: {name} </h3>
  <MostrarLista names={amigos} />
</div>
```

Luego ese tag se renderizará a lo que sea que hayamos escrito en el método `render` de ese componente, transformandose así en HTML finalmente. Existe una convención en React para distinguir entre Componentes React y elemento HTML nativos. Para el primero usamos BumpyCase y lowercase para el último. Por ejemplo:

```
<MostrarLista /> BumpyCase
<div>            lowercase
```

Separando Componentes

Seguro estarán pensando que si tenemos Componentes, lo mejor sería poder tenerlos también en archivos y carpetas distintas. Lo bueno de React es que es TODO JavaScript, asi que vamos a poder usar `CommonJS` para exportar Componentes como módulos y luego requerirlos:

```
import React from 'react';

export class Lista extends React.Component {
  render(){
    const lista = this.props.names.map(amigo => <li> {amigo} </li>);
    return (
      <div>
        <h3> Amigos </h3>
        <ul>
          {lista}
        </ul>
      </div>
    )
  }
};
```

Luego en el archivo donde lo necesitemos simplemente lo requerimos y lo empezamos a usar:

```
import { Lista } from './MostrarLista.js';
```

En este caso usamos las llaves en '{ Lista }' porque le dimos nombre a nuestro export:

```
export class Lista extends React.Component
```

De haber sido un export default podemos hacer un import simple porque le indica que es lo unico que se importará.

```
// Presentational
export default class Lista extends React.Component

// Container
import Lista from './MostrarLista.js';
```

De esta forma vamos a poder organizar muy bien nuestros componentes en distintos archivos y carpetas.

React y Funciones Puras

Como vimos recién vamos a poder usar todo lo que sabemos de JS para codear con React. Pensemoslo así, en vez de tener *funciones* que tomen argumentos y retornen valores y objetos, en React vamos a tener *funciones* que tomen argumentos y retornen **UI (user interfaces)**. Podemos resumir este concepto en `f(d) = V`, es decir, una función toma `d` argumentos y retorna una **View**. Esta es una buena forma de desarrollar interfaces, porque ahora toda tu interfaz este compuesta de invocaciones a funciones, que es la forma en que estamos acostumbrados a programar nuestras aplicaciones. Veamos este concepto en código:

```
const getFoto = function(username) {
  return 'https://photo.fb.com/' + username
}
const getLink = function(username) {
  return 'https://www.fb.com/' + username
}
const getData = function(username) {
  return {
    foto: getFoto(username),
    link: getLink(username)
  }
}
getData('atralice')
```

Si vemos el código de arriba, notamos que tenemos tres funciones y una invocación a una función. De esta forma logramos que el código sea modular y entendible. Cada función tiene un propósito específico y luego las componemos en otra función en donde generemos un comportamiento que utiliza cada una de estas para lograr el comportamiento que deseamos.

Bien, ahora modifiquemos ese código para que devuelvan un *UI* en vez de sólo datos:

```
import React from 'React';

class Foto extends React.Component {
  render() {
    return (
      <img src={'https://photo.fb.com/' + this.props.username} />
    )
  }
};

class Link extends React.Component {
  render() {
    return (
      <a href={'https://www.fb.com/' + this.props.username}
        {this.props.username}
      </a>
    )
  }
};

class Avatar extends React.Component {
```

```
  render(username) {
    return (
    <div>
      <Foto username={this.props.username}/>
      <Link username={this.props.username}/>
    </div>
  )
};

<Avatar username='atralice'/>
```

Ahora, en vez de crear componer funciones que retornen datos, estamos componiendo funciones que retornan *UIs*. Esta idea es tan importante que React en la versión `0.14` introdujo lo que se conoce como `Stateless Functional Components`, que nos permite escribir el código de arriba como simples funciones.

> Lee acá que tienen de bueno las `Stateless Functional Components`.

Reescribamos nuestro código usando `Stateless Functional Components`:

```
import React from 'React';

const Foto = function(props) {
  return <img src={'https://photo.fb.com/' + props.username} />
};

const Link = function(props) {
  return (
    <a href={'https://www.fb.com/' + props.username}
      {props.username}
    </a>
  )
}

const Avatar = function(props) {
  return (
    <div>
      <Foto username={props.username}/>
      <Link username={props.username}/>
    </div>
  )
};

<Avatar username='atralice'/>
```

Ahora cada componente es lo que llamamos una `pure function`. Este concepto viene de `Functional Programming`, básicamente, las funciones puras son consistente y predecible, porque tienen las siguientes características:

- Una función pura siempre retorna el mismo resultado para los mismos argumentos.
- La ejecución de una función pura **NO** depende del *estado* de la aplicación.

- Las funciones puras **NO** modifican el estado ni ninguna variable afuera de su scope.

En React el método `render` necesita ser una función pura, y por ende, todos los beneficios de programar funciones puras se transladan ahora a tu **UI**. De esta forma logramos tener lo que en React se conoce como: `Stateless Functional Components`. Si vemos le ejemplo, todos los Componentes que armamos no tiene estado, y que no hacen nada más que recibir datos a través de `proprs` y renderizar una **UI**, esto es, básicamente, Componentes que sólo tienen el método `render`. De esto nace un paradigma en el que se diferencian dos tipos de Componentes, los que acabamos de mencionar son los llamados `Presentational Components` y los segundos son `Containers Components`.

Como se pueden imaginar, los `Presentational Components` se preocupan en como **se ven las cosas** y los `Container Componentes` en **como funcionan las cosas**. Organizar nuestro código de esta forma trae varias ventajas:

- Mejor separación de temas. Vas a entender mejor tu aplicación y tu UI escribiendo Componentes de este modo.
- Mejor reusabilidad. Podes usar los mismos Presentational Componentes en distintos Containers.
- Podes tener a los diseñadores trabajando en los Presentational Componentes sin tener que meterse a la lógica de la aplicación.

> Esto es sólo un paradigma, seguramente hay otros que tengan otras características. Si querés poder leer más de este paradigma acá.

Organizando las carpetas de un Proyecto

Antes mencionamos el patrón de separar Componentes según mantengan *Estados* (**Containers**) o sólo sirvan para renderizar algo (**Presentational**). Vamos a organizar nuestra estructura de carpetas de proyecto alrededor de este.

Básicamente, vamos a guardar cada Componente en un archivo `.js` separado y *exportarlo*. Los *Presentational* van a ir en una carpeta llamada `components`, y los *Containers* en otra carpeta llamada `containers`. Como sabemos, los *Containers* van a incluir o *requerir* a los *Presentational* en su código, y desde código denuestra app vamos a *requerir* a los *Containers*.

> Por convención vamos a llamar a los archivos que contengan un componente con la primera letra en Mayúsculas. Por ejemplo: `Header.jsx` o `Profile.js`.

Cuando comenzamos un proyecto nuevo de React, en vez de empezar de cero, podemos guardar un esqueleto que ya tenga todas las tareas que deberíamos repetir en cada proyecto, en inglés esto se conoce como `boilerplates project`. De hecho, podemos hacer nuestro propio `boilerplate` o buscar online alguna que se ajuste a nuestras necesidades. En general que están publicados online traen muchas cosas que tal vez no vayamos a usar, aquí algunos ejemplos:

- react-webpack-boilerplate
- react-starter-kit
- react-native-starter
- react-webpack-boilerplate

> Hace poco salió Create React APP una mini app del equipo de Facebook que te ayuda a comenzar un proyecto nuevo de React en segundos (yo todavía no la probé pero parece interesante!).

Tambien pusimos nuestro propio ejemplo de un boilerPlate simplificado acá. Básicamente trae un archivo de *webpack* preconfigurado y un mini servidor *express* para levantar el archivo desde la carpeta del output de *webpack*, super simple!

> Como siempre, todo viene en muchos *sabores* y hay que probar y elegir el que mas le gusta a cada uno, ninguno es el mejor, todos van a tener pros y cons.

Propiedades y Estados

Ya vimos que en React las propiedades se pasan de componentes padres a hijos a través de la variable `props`. Veamos algunas propiedades más avanzadas del comportamiento de `props`.

Estados

La otra forma de que los Componentes de React tengan información es a través del `State`. Este `Estado` se encuentra disponible en el objeto `State` de cada Componente, es decir que podermos acceder a el a través de `this.State`. Los estados **no** son inmutables, es decir que estan pensados para *cambiar* eventualmente. Justamente por esto, es que los Componentes que tienen Estados son menos *performantes* que los que no.

Cuando creamos un Componente cualquiera, su estado o `this.State` es igual a `null`, o sea que no tiene Estado por defecto!. Para agregar un estado vamos a usar el método `getInitialState`, el cual retorna un objeto que contiene el estado inicial de ese Componente. Por ejemplo:

```
class Componente extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      nombre : "Toni",
      trabajo  : "Profesor"
    };
  }
  render() {
    return (
      <div>
        Mi nombre es {this.state.nombre}
        y soy el  {this.state.trabajo}.
      </div>
    )
  }
};
ReactDOM.render(<Componente />, document.getElementById('appEstado'));
```

En un componente de funcion la nueva forma de tener un `State` es a traves del Hook `useState` que vimos anteriormente. Siguiendo con el ejemplo anterior:

```
function Componente(props) {
  const [nombre, setNombre] = useState("Toni");
  const [trabajo, setTrabajo] = useState("Profesor");
```

```
  return (
    <div>
      Mi nombre es {nombre}
      y soy el  {trabajo}.
    </div>
  )
}
ReactDOM.render(<Componente />, document.getElementById('appEstado'));
```

Actualizando el Estado en una clase

Para actualizar o cambiar el estado de un Componente llamamos a la función `this.SetState`, pasandole por parámetros un nuevo estado, es decir un objeto con las nuevas propiedades:

```
// Agregamos este método al código anterior
...

handleClick() {
  this.setState({
    nombre : "Guille"
  });
}
render() {
  return(
    <div onClick={this.handleClick}>
      Mi nombre es {this.state.nombre}
      , y soy el  {this.state.trabajo}.
    </div>;
  )
}
```

Actualizando el Estado en una funcion

Para actualizar el `state` en un componente de function, como vimos, llamamos a la funcion que nos devuelve el Hook `useState`. Y le pasamos por parametro el nuevo `estado`. Por ejemplo:

```
// Agregamos este método al código anterior
...

const handleClick = () => {
  setNombre("Guille");
}

  return(
    <div onClick={handleClick}>
      Mi nombre es {nombre}
      , y soy el  {trabajo}.
    </div>
```

```
    )
  }
```

> Cuando se cambia el estado de un Componente usando `setState` o el Hook 'useState', React re-renderiza todo el Componente. Hay que interntar hacerlo lo menos posible!

Diferencia entre Estados y props

Seguramente se preguntarán cúal es la diferencia entre props y estados, en realidad las dos mantiene información o datos que va a usar el Componente. La diferencia es el uso de cada una. Las pros de un Componente van a ser pasadas por el padre del mismo y *deberían* ser *inmutables*, es decir, que si se cambian las props (o sea que el padre las cambia), el Componente se debería renderizar de nuevo con las nuevas props. Podría decir que las props son un tipo de *configuración* del Componente.

Los estados, encambio, son manejados exclusivamente e internamente por un Componente, nada tiene que ver con el estado de otros Copmonentes, es decir el estado es *privado*, además los estados **no** son *inmutables*!

React Life Cycle Events

El Life Cycle Completo

Como sabemos, es critico para una app tener estados, es decir poder hacer Ajax requests, etc... Dijimos que el método render de los Componentes tiene que ser *stateless*, es decir que en él no vamos a poder agregar este comportamiento. Para poder hacerlo vamos a incorporar el concepto de ciclo de vida de React y sus métodos. Con ellos, vamos a poder dar comportamiento a nuestros Componentes según sucedan ciertos eventos en nuestra app y en nuestros Componentes (por ejemplo, cuando se renderiza el Componente, o cuando le llegan datos nuevos, etc).

En la imagen de abajo, vemos el ciclo de vida Completo de cualquier Componente de React. En ella vemos también los estados en los que puede estar un Componente y qué cosas o funciones activarán el paso de estados y por ende la invocación de los métodos que nos provee React:


LifeCycle

> Pueden ver este Gist y probarlo localmente para tener un mayor entendimiento de *cuando* se invoca cada método de React.

Veamos algunos de estos eventos, primero los vamos a separa en dos categorias que cubren la mayoría de ellos:

- Cuando un Componente es Montado o Desmontado en el DOM.
- Cuando un Componente recibe nuevos datos.

Montando / Desmontado

Cuando un Componente es agregado al DOM decime que fue Montado (mounted) y cuando es removido del DOM decimos que fue Desmontado (unmounted). Por definición estos eventos son llamados por React *solo* una vez en el ciclo de vida del Componente (cuando 'nace' y cuando 'muere'). Por lo tanto nos van a servir

para setear ciertas condiciones iniciales de un Compoente o bien, cerrar o eliminar ciertos listeners que sólo le servían al Componente en cuestión, en general la mayoría de las veces haremos lo siguiente en estos Eventos:

- Establecer algunas *props* por defecto.
- Establecer algunos Estados iniciales del Componente.
- Hacer alguna petición AJAX para traer datos necesarios para el Componente.
- Crear listeners si son necesarios.
- Remover listeners que ya no sirven más.

Para poder hacer uso de estos Eventos, React nos da una serie de método que son invocados según el momento del ciclo de vida del Componente. Dentro de estos métodos nosotros vamos a agregar la funcionalidad que necesitamos para nuestro Componente.

Establecer props por defecto

Varias veces vamos a tener Componentes que nos van a servir para varias cosas, por lo tanto, cuando los instanciamos vamos a querer darles distintas *props* por defecto (se toman estas props si no le pasamos una en particular). Para hacerlo, React nos provee el método `defaultProps`, en que asignamos un objeto con las *props* que finalmente tendrá nuestro Componente cuando se renderize, por ejemplo:

```
class Cargando extends React.Component {
  constructor(props) {
    super(props);
    ...
  }
  render() {
    ...
  }
};

Cargando.defaultProps = {
  text: 'cargando...',
}
```

Al igual que en una clase, tenemos el ejemplo en una `function`:

```
function Cargando(props) {
  return (
    ...
  )
}

Cargando.defaultProps = {
  text: 'cargando...',
}
```

En este ejemplo, si no le pasamos la *prop* `text` al Componente `Cargando`, esta tomará el valor por defecto: `cargando...`.

Establecer un estado Inicial

Cuando queremos que nuestro Componente maneje algún tipo de Estado le tenemos que setear el *Estado Inicial* del mismo. Esto lo podemos lograr desde el constructor mediante `this.state` que se le asigna un objeto con el estado inicial. Esta será llamada cuando el Componente es montado al DOM por primera vez.

```
class Login extends React.Components{
  constructor(props) {
    super(props)
    this.state = {
      email: '',
      password: ''
    }
  }
  render() {
    ...
  }
};
```

Al usar un componente de funcion, seteamos el *Estado Inicial* con el Hook `useState`, en donde el primer elemento del array que nos retorna es nuestro estado, y el segundo es una funcion para cambiar el estado:

```
function Login() {
  const [email, setEmail] = useState('')
  const [password, setPassword] = useState('')

  return (
    ...
  )
};
```

Hacer alguna petición AJAX para traer datos necesarios para el Componente

Este es un caso muy común, el Componente necesita datos que son traídos a través de un request tipo AJAX. En un componente de clase, React nos da el método `componentDidMount`, este método es llamado justo después que el componente se montó al DOM:

```
class Lista extends React.Component {
  componentDidMount() {
    return Axios.get(this.props.url).then(this.props.callback) // AJAX request con
Axios
  }
```

```
  render() {
    ...
  }
};
```

Al usar una funcion, introducimos el Hook `useEffect`. Este recibe un callback que se ejecuta despues de cada renderizado en el componente, y nos permite hacer peticiones de datos, establecimiento de suscripciones y actualizaciones manuales del DOM en componentes de React. Por lo general llamamos a estas operaciones "efectos secundarios" (o simplemente "efectos"). Este Hook equivale a los ciclos de vida de clase: componentDidMount, componentDidUpdate y componentWillUnmount combinados. El segundo argumento que recibe es un `array`, al pasar un array vacio `[]`, esto le indica a React que `useEffect` no depende de ningún valor proveniente de las props o el estado, de modo que no necesita volver a ejecutarse.

```
function Lista(props) {
  useEffect(() => {
    Axios.get(props.url).then(props.callback) // AJAX request con Axios
  },[])

  return (
    ...
  )
};

###### Crear listeners si son necesarios

Como se pueden imaginar, también vamos a hacerlo usando el método
`componentDidMount` y el Hook `useEffect`:

```javascript

class Lista extends React.Component {
  componentDidMount() {
    ee.on('evento', () => this.setState({ ... }) // creamos un event listener para
un 'evento'
  }
  render() {
    ...
  }
};
```

```
function Lista() {
  const [state, setState] = useState({})
  useEffect(() => {
    ee.on('evento', () => setState({ ... }) // creamos un event listener para un
'evento'
  },[])
```
```

```
  return (
    ...
  )
};
```

Remover listeners que ya no sirven más

Análogamente, existe el método `componentWillUnmount` que es invocado justo antes de remover el Componente del DOM:

```
class FriendsList extends React.Component {
  componentWillUnmount() {
    ee.off() //sacamos el listener que habiamos puesto.
  }
  render() {
    ...
  }
};
```

Para el caso de una function, usamos el mismo Hook, `useEffect`, Quizás puedas estar pensando que necesitaríamos un efecto aparte para llevar a cabo el remove del event. Pero el código para añadir y eliminarlo está tan estrechamente relacionado que useEffect está diseñado para mantenerlo unido. Si tu efecto devuelve una función, React la ejecutará en el momento correcto:

```
function FriendList() {
  const [state, setState] = useState({})
  useEffect(() => {
    ee.on('evento', () => setState({ ... })

    return () => {
      ee.off() //sacamos el listener que habiamos puesto.
    };
  },[])

  return (
  ...
  )
};
```

En este caso, cuando retornamos una funcion en `useEffect`, esta es ejecutada antes de que el componente sea removido de la UI.

Eventos cuando el componente recibe nuevos Datos

El primero método que veremos es `componentWillReceiveProps`, este evento se activa cuando el Componente recibe nuevas *props*.

El segundo, y más avanzado es `shouldComponentUpdate`. Ya sabiamos que React le pone mucho importancia a re-renderizar nuevos Componentes (ya que esto implica mucho trabajo para el cliente), por lo tanto nos da este método para que nosotros podamos controlar este comportamiento. Justamente, `shouldComponentUpdate` devuelve un **Booleano**, si es `true`, entonces se re-redenrizará el Componente (y por ende todos sus hijos), en caso de ser `false` no se hará tal cosa.

En el caso en donde tenemos un componente de funcion. Utilizaremos el Hook `useEffect` para el primer caso. Este Hook puede recibir como segundo parametro un array `[]`. Por ejemplo:

```
function Ejemplo() {
  const [name, setName] = useState("Toni");
  useEffect(() => {
    document.title = name;
  }, [name]);

  return (
    <input value={name} onChange={event => setName(event.target.value)} />
  );
};
```

En el ejemplo anterior, no necesitamos actualizar el título del documento (nuestro efecto) después de cada representación, sino solo cuando el nombre de la variable del state cambie su valor. Es por eso que pasamos array con el valor de `name` como segundo parámetro: Aca si solo queremos que nuestro Hook se invoque solo después del primer render, tenemos que pasar una matriz vacía [] (que nunca cambia) como segundo parámetro. En el segundo caso podemos utilizar `React.memo` que es un HOC (High Order Component) que ayuda en la performance de renderizado de un componente, evitar un re-renderizado. Si un componente devuelve el mismo resultado, es decir, no cambian sus props. Envolver el componente en `React.memo` puede ayudar mucho a la performance. Esta funcion puede recibir como segundo argumento una funcion de comparacion personalizada, que reciba las props viejas y las nuevas. Si retorna true, se obvia la actualización. Por ejemplo:

```
function Ejemplo(props) {
  return (
    <div>{props.name}</div>
  );
};

const fnComparacion = function(prevProps, nextProps) {
  return prevProps.name === nextProps.name;
};

export default React.memo(Ejemplo, fnComparacion);
```

Homework

Completa la tarea descrita en el archivo README