Hacé click acá para dejar tu feedback sobre esta clase.

Hacé click acá completar el quiz teórico de esta lecture.

REACT

Qué es React?

React es una librería de JavaScript que es declarativa, eficiente y flexible y sirve para construir interfaces de usuarios. Esta librería fue creada por el equipo de *facebook* e *instagram*, que fue liberada y ahora es un proyecto **open source**.

> La diferencia entre programación declarativa o imperativa, es que en la primera le *decimos* a la computadora **qué** queremos hacer (ella se encarga de saber cómo), mientras que en programación *imperativa* le decimos exactamente **cómo** queremos que se hagan las cosas. Puede que parezcan dos cosas iguales, pero veamos la diferencia con un ejemplo:

```
const numbers = [4,2,3,6];
//imperativo (le decimos COMO queremos que se hagan las cosas)
let total = 0;
for (let i = 0; i < numbers.length; i++){
  total += numbers[i]
}
//declarativo (decime QUE queremos que se haga)
numbers.reduce(function(p, c){
  return p + c;
})
```

Al usar *React* vamos a tener que pensar en términos de **componentes**. Cuando armemos una página con *react* vamos a tener que pensar nuestro sitio o página como una serie de **pequeños componentes**. En realidad, podemos decir que todo es un *componente*, de hecho vamos a tener *componentes* que estén formados por otros *componentes*. Esto último va a suceder cuando tengamos un problema que sea grande, y

la única forma (o la mejor) para resolver en *react* es dividirlo en pequeños problemas. Esto además hace que los componentes sean altamente **reusables**.

> Esta forma de desarrollar se conoce como **component driven development**.

React también es muy bueno en términos de performance, cuenta con un feature llamado **Virtual DOM**, con lo cual logra renderizar muy rápido las páginas manteniendo el código entendible y fácil de manejar.

Virtual DOM

Cuando queremos rastrear cambios en algún modelo y luego trasladarlos al DOM (rendering), tenemos que tener en cuenta dos cosas importantes:

1. Cúando se cambiaron los datos.
2. Qué elementos del DOM necesitan ser actualizados.

Para el primer punto react utiliza un modelo de observador (esto quiere decir que no tiene que estar revisando continuamente por cambios). Por lo tanto cuando algo cambia este le *avisa* a react inmediatamente.

Para el segundo punto, React construye una representación del DOM en memoria y calcula que elementos del DOM van a cambiar. Hacer cambios en el DOM consume muchos recursos, es por eso que se concentraron tanto en minimizar al máximo los cambios en el DOM real. Muy básicamente, cuando algo cambia en el estado del modelo, la idea es no tocar el DOM e ir haciendo cambios en el Virtual DOM, luego se computan las diferencias entre el DOM real y el virtual DOM (para esto se utilizan algoritmos de diferencia bastante copados), y finalmente se realizan la menor cantidad de cambios posibles al DOM real.

Component Driven Development

Miremos la imagen de abajo, cada cajita con un color particular representa un componente. Esta es una de las muchas formas de poder dividir un solo elemento o feature de nuestro sitio. Según esta división tendríamos la jerarquía de componentes que se muestran a la derecha de la imagen:


componentes

Qué debería contener un *Componente*?

Para diseñar componentes es importante tener en cuenta el principio de diseño llamado *single responsabilty principle*, o *princio de responsabilidad única*, basicamente deberíamos diseñar cada componente para que sea responsable de *sólo* una cosa. Pensar de este modo no es fácil, acá hay un tutorial de *facebook* para empezar a pensar en componentes.

De todos modos, la mejor forma de aprender es la práctica. Al usar React, te vas a ir dando cuenta cuando te conviene subdividir un componente en otros o no. De hecho, no te deberías preocupar tanto por asumir esta mentalidad antes de empezar, aceptá el hecho que mientas desarrolles con react vas a ir cambiando solo la mentalidad, y no al revés.

Usando React

Empezemos por *instalar* React y usarlo en una página estática, para que veamos cómo *se siente*. Primero vamos a comenzar con la forma *sencilla* de empezar con React, consiste en hacerlo en un documento HTML.

> Por ahora no se preocupen en entender el código en su totalidad. Luego veremos én que consiste cada cosa que usamos aquí, por ahora lo imporante es aprender a instalar lo necesario para poder empezar a desarrollar.

Luego veremos la mejor manera de usar react, que consiste generar un pipeline con algunas herramientas para poder tener el *ambiente* preparado para React, estas herramientas pueden ser: `glup`, `grunt`, o `webpack`, nosotros usaremos el último.

Vamos a ver los siguentes componentes:

- React: La librería en si.
- React Router: Nos permite mapear componentes a URLs, de esta forma podremos armar páginas tipo SPA.
- Webpack: Es una herramienta que agrupa modulos de JavaScript (entre otras cosas que hace), o mejor dicho, agrupa código, lo vamos a usar para crear el pipeline para desarrollar con React.
- Babel: Es una librería/herramienta que nos permite transformar nuestro código. Con React vamos a usar JSX, que es un lenguaje construido sobre JS, Babel lo va a transformar a JS normal.

La Manera Sencilla

La forma más fácil y rápida para poder empezar a ver *como es* React es a traves de un documento HTML. Ojo, esta no es la *mejor* forma ni la que usaremos en adelante. Aca veremos dos formas de representar un componente en React. Componente basado en clases y funciones. Hasta hace poco con las clases era la unica forma que teniamos de utilizar caracteristicas como el estado (state), hasta la introduccion de Hooks (lo veremos en profundidad mas adelante), que nos permiten usar state dentro de un componente de funcion.

```
<!DOCTYPE html>
<html>
  <head>
    <title>React</title>
    <script crossorigin src="https://unpkg.com/react@16/umd/react.development.js">
</script>
    <script crossorigin src="https://unpkg.com/react-dom@16/umd/react-
dom.development.js"></script>
    <script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
  </head>
  <body></body>
</html>
```

Como puedes comprobar solo estamos haciendo referencia a dos ficheros ficheros JavaScript:

- **react.js**: La librería principal de ReactJS.
- **react-dom.js**: Desde React 0.14 el manejador de DOM de tus componentes React se realiza con esta libreria JavaScript.
- **browser-min.js**: Una version minima de BabelJS para el navegador.

Ahora agreguemos un poco de código de React en el body:

```
<body>
  <div id="app"></div>
  <script type="text/babel">
    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'));
  </script>
</body>
```

Como ven el tag script tiene el atributo type con el valor **text/babel** y no el tan conocido text/javascript, esto es debido a que a partir de ahora vamos a escribir código EcmaScript6 y *traducirlo* usando Babel. Lo empezamos a hacer definiendo nuestra primera clase *Hola* extendiendo de la clase primitiva React.Component. Como habrás podido adivinar BabelJS ejecuta toda la transpilación entre ES6 y JavaScript desde tu navegador y es por eso que en la siguiente sección comprenderás porque es mejor utilizar un gestor de procesos que genere ficheros JavaScript precompilados.

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.

Componentes

Como dijimos los bloques básicos con lo que vamos a construir todo en React se llaman *Components*. Podemos pensar a un *Componente* como una colección de HTML, CSS y JS, y un estado o datos especificos para ese componente. Estos componentes están definidos o en JavaScript puro, o usando lo que se conoce como **JSX**.

JSX

Básicamente, **JSX** es *syntactic sugar* para una función en particular de React: `React.createElement(component, props, ...children)`. Lo que sucede es que está función recibe un objeto que describe un elemento tipo XML (parecido a HTML, pero con tags que podemos inventar). Es contraintuitivo escribir un elemento XML, con sus propiedades y demás, codificado en un objeto, por eso mismo es que crearon este *lenguaje* que nos permite escribir directamente código **XML** embebido en *JS* y que luego será transformado a *JS* por algún loader, como *babel*.

Veamos un ejemplo:

Sin **JSX**:

```
React.createElement(
  MyButton,
  {color: 'blue', shadowSize: 2},
  'Click Me'
)
```

Con **JSX**:

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

Esta es la razón de ser de **JSX**, podemos leer más en detalle las demás features que nos ofrece acá.

Podemos ir a la página de `babel` y ver cómo se transforme el código *JSX* en Javascript plano:

```
// Código en JSX, componente basado en clases
class HelloWorld extends React.Component{
  render(){
    return (
      <div>
        Hola, Soy Henry!!
      </div>
    )
  }
};
// Codigo en JSX, componente de funcion
function HelloWorldFunction() {
  return(
    <div>
      Hola, Soy Henry!
    </div>
  )
};

// Código en Javascript con un componente de clases
"use strict";

var HelloWorld = function (_React$Component) {

  _createClass(HelloWorld, [{
    key: "render",
    value: function render() {
      return React.createElement(
```

```
          "div",
          null,
          "Hola, Soy Henry!!"
        );
      }
    }]);

    return HelloWorld;
  }(React.Component);

  // Código en Javascript con un componente de funciones
  "use strict";

  function HelloWorld() {
    return React.createElement(
      'div',
      null,
      "Hola, Soy Henry!!"
    );
  }
```

Como podemos ver, la transformación toma los datos y lo transforma a código Javascript. Se llamó a `React.createElement` en la función `render`, esta función crea un elemento HTML según los parámetros que les pasamos. Como podemos ver, escribir código JavaScript para crear elementos HTML puede ser engorroso. Si bien, podríamos codear todas nuestras apps de react escribiendo JS nativo, lo mejor y más productivo va a ser usar **JSX**.

Pueden ver más transformaciones que realiza *babel* acá;

Creando nuestro primer componente

> Usemos el ejemplo de arriba, pero pensemos paso a paso cómo crear ese componente.

Como podemos imaginar, un *Componente* en react está representado por una clase o un objeto llamado *Component*. Este tiene incorporado una serie de propiedades y métodos, los cuales logran el comportamiento y le dan el poder de React.

Cuando nosotros creamos un componente nuevo, básicamente *heredamos* todas esas propiedades y métodos del objeto *Component* y luego customizamos el nuevo componente según nuestras necesidades. Veamos un ejemplo con un componente de clases:

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

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

```
    )
  }
};
ReactDOM.render(<HelloWorld />, document.getElementById('app'));
```

Del mismo modo podemos ver el ejemplo con un componente de funcion:

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

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

Primero importamos las librerías donde contienen la defincón de los objetos de React. Luego creamos una clase, en este caso llamada 'HelloWorld', y extendemos a `React.Component` para formar nuestro nuevo **Componente** (este es el paso donde heredamos toda la funcionalidad de React!). Dentro de este nuevo objeto `HelloWorld` vamos a definir el método `render`, que es un método **requirido** por todos los componentes de React. Este método define el **template** de nuestro componenete, por lo tanto es obligatorio!. Como verán, en este punto hemo incluido **JSX**, ya que escribimos código XML mezclado con JS, y justamente lo hemos usado para crear un nuevo tag `div` que contenga el String '*Hello World*'.

Bien, ahora ya tenemos nuestro propio componente creado. Para indicarle a React que lo incluya en la página, vamos a llamar al método `render` de `ReactDOM`. Este método recibe como parámetro el componente que queremos incluir al DOM, y en qué lugar del mismo. Por lo tanto le pasamos nuestro componente (`<HelloWorld />`), y un selector `document.getElementById('app')` para indicarle donde agregarlo.

Props

Una de las ventajas de separar todo en Componentes, es que estos pueden ser reutilizables. Para que lo sean, vamos a poder cambiar un poco su comportamiento pasandole algunos datos. En React estos datos se conocen como **props** (propiedades) de un Componente. Veamos la forma de pasar *props* a un Componente.

Las *props* funcionan como los *atributos* HTML, es decir, que cuando usamos un Componente podemos agregarle *props* escribiendo su nombre dentro del tag del mismo. Por ejemplo, para agregar la *prop* `name` al Componente que habiamos creado antes, cuando lo usamos escribimos `<HelloWorld name='Henry />`. De esta forma, podemos pasar una o varias *props* al mismo Componente. Ahora, para utilizarlas, dentro del Componente vamos a tener un objeto que se encuentra en `this.props` que va a contener todas las *props* que le hayamos dado a ese Componente. En el ejemplo, el `name` va a estar en `this.props.name` y va a tomar el valor de `Toni`. Por último, para poder acceder al contenido de `this.props.name` vamos a tener que escribir una *expresión JavaScript* dentro de *JSX*, para hacerlo tenemos que separarla con **{}**, en el ejemplo también usamos los *{}* para pasar una variable como *prop*:

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

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

const nombreVariable = 'Toni';

ReactDOM.render(<HelloWorld name={nombreVariable} />,
document.getElementById('app'));
```

En un componente de funcion:

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


function HelloWorldFunction(props) {
  return(
    <div>
      Hola, {props.name}!!
    </div>
  )
}

const nombreVariable = 'Toni';

ReactDOM.render(<HelloWorldFunction name={nombreVariable} />,
document.getElementById('app'));
```

Ahora tenemos un Componente que nos sirve para saludar! Sólo tenemos que pasarle una *prop* con el nombre de a quien va dirigido el saludo. 😒 Es un ejemplo simple, pero mostramos la forma de React de pasar *props* a sus Componentes. Aca vemos una de las diferencias entre un componente de clases y un componente de funciones. El uso de la palabra reservada 'this', esto nos hace mas facilita a la hora de hacer un debugging, no pensar a que hace referencia 'this' siempre es un plus. Ademas, no usar 'this' significa que no es necesario el uso de bindear los eventos para hacer referencia a eventos dentro de una clase.

> Mirá como quedaría este código traducido a JavaScript plano acá.

Una cosa muy importante de las *props* es que son **innmutables**, es decir, que cuando las seteamos no las vamos a poder cambiar en el futuro (por lo menos están pensadas para eso). Podemos pensar en las *props*

como una *incialización* o una *configuración* de un Componente antes de agregarlo. Para datos que *cambian* veremos más adelante los **estados** de un Componente.

Eventos de Usuarios y Callbacks

Veamos como podemos capturar algunos eventos disparados por el usuario en React. Vamos a usar un ejemplo, en donde el usuario pueda escribir su nombre en un `form` y luego vamos a mostrar un `alert` con lo que escribió. Primero comenzamos agregando el `form`:

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

class HelloWorld extends React.Component {
  render(){
    return (
      <div>
        <form>
          <input type='text' ref='name'>
          <button>Poner Nombre</button>
        </form>
        Hola, {this.props.name}!!
      </div>
    )
  }
};
ReactDOM.render(<HelloWorld name='Soy Henry!' />, document.getElementById('app'));
```

Hemos creado el `form` y dentro un `input` donde el usuario va a escribir su nombre, como pueden ver hemos agregado el atributo `ref`, este es especial de React, y lo usa para poder hacer referencia luego a ese elemento. Por ahora este `form` no hace nada, le agreguemos una acción cuando es Submiteado. Para eso usamos un Evento nativo de React llamado `onSubmit`, que tambien pasamos como un atributo, y dentro suyo la función que queremos que se ejecute como callback. Generalmente estas funciones son partes del Componente, por lo tanto la vamos a definir dentro del mismo:

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

class HelloWorld extends React.Component {
  constructor(props) {
    super(props);
    // Necesitamos el binding para hacer funcionar el this en el evento
    this.onButtonClick = this.onButtonClick.bind(this);
  }
  onButtonClick(e){
    e.preventDefault();
    const name = this.refs.name.value;
    alert(name);
  }
  render(){
```

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

Como vemos, pudimos acceder al elemento `input` usando `this.refs` gracias a que agregamos el bind del **this** en nuestro constructor. En el constructos vamos a poder setear props y estados por default para nuestro componente. Cabe notar que lo que se guarda dentro de `refs` es una referencia al **elemento HTML**, por lo tanto si queremos lo que escribió el usuario, usamos `this.refs.name.value`.

Viendo el ejemplo con una funcion:

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

function HelloWorld(props) {

  let textInput = useRef(null);

  const onButtonClick = (e) => {
    e.preventDefault();
    const name = textInput.name.value;
    alert(name);
  }

  return (
    <div>
      <form onSubmit={onButtonClick}>
        <input type='text' ref={textInput} />
        <button>Poner Nombre</button>
      </form>
      Hola, {props.name}!!
    </div>
  )
}

ReactDOM.render(<HelloWorld name='Soy Henry!' />, document.getElementById('app'));
```

En el caso de un componente de funcion, no podemos usar el atributo `ref` ya que esta no puede ser instanciada, y no tenemos el uso de la palabra `this` para hacer referencia a 'refs' dentro de la clase. Aca estamos usando una nueva caracteristica de React que son los Hooks. En este caso estamos importando el Hook `useRef`, que lo iniciamos con un argumento, en este caso null, cuya propiedad `.current` se inicializa

sobre el argumento pasado. En este caso pasamos un objeto de referencia a React con `ref`, React configurará su propiedad `.current` al nodo del DOM correspondiente cuando sea que el nodo cambie.

Como vemos, logramos tener el `alert` con el nombre que escribió el usuario. Ahora, ya sabemos que los `alert`s no sirven. Intenemos hacer que cuando el usuario haga click en el botón se cambie el nombre en el saludo. Para esto vamos a introducir el concepto de **Estado** de un componente.

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 = ['Santi', 'Guille', 'Facu', '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`.

this.props.children

Digamos que tenemos un Componente cualquiera (de React o bien HTML simple), y queremos acceder a la data que está entre el opening tag y el closing tag. Por ejemplo, quiero acceder al nombre que está dentro de `<Nombre>`:

```
<Nombre>
  SoyHenry
</Nombre>
```

React nos da una forma simple de acceder a ellos, y es con la propiedad `children` de `this.props`. En este ejemplo en particular, la propiedad `this.props.children` del Componente *Nombre* va a tomar el valor 'Soy Henry'.

Qué Pasa si lo que está adentro es un poco más complejo? Por Ejemplo:

```
<Nombre>
  <Foto />
  <Link />
</Nombre>
```

Ahora dentro de `Nombre` tenemos dos Componentes (`Foto` y `Link`). Bien, como se podrían imaginar, `this.props.children` va a evaluar a un **arreglo** de Componentes.

PropTypes

Algunas veces va a ser necesario controlar el tipo de datos de las *props* que estamos enviado a un Componente. Por suerte, React nos provee de una funcionalidad para hacer de forma *nativa*. Esta forma son las `PropTypes`. Básicamente consiste en definir un objeto de configuración en donde vamos a declarar el tipo de datos de cada propiedad, si no coinciden cuando los pasemos, React nos generará un error. Por ejemplo:

```
import React from 'react';

const PropTypes = React.PropTypes;

class Icono extends React.createClass {
  propTypes: {
    nombre: PropTypes.string.isRequired,
    tamanio: PropTypes.number.isRequired,
    color: PropTypes.string.isRequired,
    style: PropTypes.object
  }
  render: ...
};
```

En este ejemplo estamos declarando que las *props* que le lleguen al Componente `Icono`, deberán ser:

- *nombre*: Tiene que ser de tipo *String* y es obligatorio.
- *tamanio*: Tiene que ser de tipo *Number* y es obligatorio.
- *color* : Tambien un *String* y obligatorio.
- *style* : Debe ser un *Objeto*, no es obligatorio.

Como vemos, toda esta funcionalidad está contenida en el objeto `React.PropTypes` que viene nativamente en React.

> Para más información sobre `React.PropTypes` y las cosas que podemos controlar con ella vamos a la documentación oficial aquí.

Homework

Completa la tarea descrita en el archivo README