



Capacitación React JS Básico

# Manual de Laboratorios

---



Roque Rueda  
2019

---

# CONTENIDO

---

<b>CONTENIDO</b>	<b>1</b>
<b>LAB 01: Hola Mundo</b>	<b>2</b>
<b>LAB 02: Composición de Componentes</b>	<b>19</b>
<b>LAB 03: Pensando en React</b>	<b>41</b>
<b>LAB 05: Crear una aplicación React</b>	<b>64</b>
<b>LAB 06: Estado del componente</b>	<b>77</b>
<b>LAB 07: Peticiones HTTP</b>	<b>82</b>
<b>LAB 08: Introducción a Redux</b>	<b>92</b>

# LAB 01: Hola Mundo

## Objetivos

- Instalación de las herramientas necesarias para la capacitación
- Crear una página web utilizando HTML y JavaScript
- Desplegar un mensaje “Hola Mundo” utilizando React

## Instalar npm (Node Package Manager)

**Node.js** es un entorno de ejecución para JavaScript orientado a eventos asíncronos y **npm** se distribuye en conjunto con Node.js, para lo cual vamos a instalar Node.js.

Vamos a navegar al sitio: <https://nodejs.org/es/download/> en donde seleccionaremos nuestro sistema operativo y seguiremos las instrucciones para su instalación.

Figura 1.1 Sitio web Node.js



Para verificar nuestra instalación vamos a ejecutar en la terminal el siguiente comando:

### Listado 1.1 Visualizar versión de Node.js

```
npm -v
```

Lo que nos va a mostrar una salida como la siguiente:

### Figura 1.2 Version de Node.js

```
roquerueda ~ $ npm -v  
6.4.1
```

## Instalar Git

**Git** es un sistema de control de versiones que se maneja utilizando **ramas**, para instalar git en caso de contar con el aun, vamos a ingresar al sitio: <https://git-scm.com/downloads> y a descargar la versión de acuerdo a nuestro sistema operativo.

Después de la instalación podemos ejecutar el siguiente comando para validar la versión instalada:

### Listado 1.2 Visualizar versión de git

```
git --version
```

Con lo cual obtendremos una salida en la terminal como la siguiente:

### Figura 1.3 Version de Git

```
roquerueda ~ $ git --version  
git version 2.15.2 (Apple Git-101.1)
```

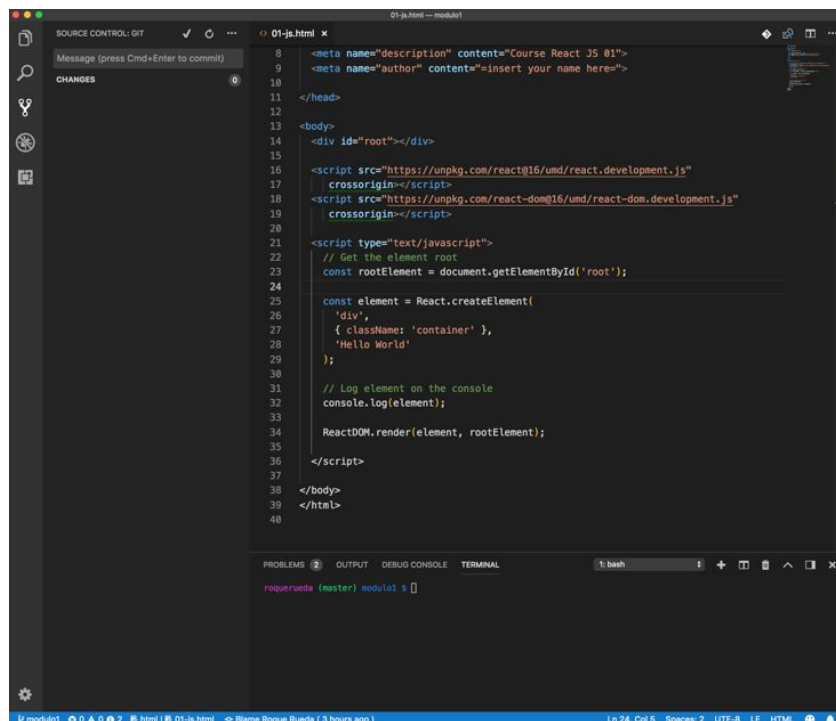
## Instalar Visual Studio Code

Visual Studio Code es un editor de texto ligero, se encuentra disponible para Windows, MacOS y Linux, cuenta con soporte para JavaScript, TypeScript y Node.js.

Para instalar visual studio code debemos ingresar al sitio web de descargas: <https://code.visualstudio.com/download> y vamos a seleccionar nuestro sistema operativo.

Una vez que finaliza la instalación podemos ejecutarlo y ver una pantalla como la siguiente:

### Figura 1.4 Ventana de Visual Studio



```
8 <meta name="description" content="Course React JS 01">
9 <meta name="author" content="Insert your name here">
10
11 </head>
12
13 <body>
14 <div id="root"></div>
15
16 <script src="https://unpkg.com/react@16/umd/react.development.js"
17   crossorigin=</script>
18 <script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"
19   crossorigin=</script>
20
21 <script type="text/javascript">
22   // Get the element root
23   const rootElement = document.getElementById('root');
24
25   const element = React.createElement(
26     'div',
27     { className: 'container' },
28     'Hello World'
29   );
30
31   // Log element on the console
32   console.log(element);
33
34   ReactDOM.render(element, rootElement);
35 </script>
36
37 </body>
38 </html>
39
40
```

## Instalar create-react-app

Es un ambiente de Interfaz de Línea de Comandos **CLI** el cual nos brinda una plantilla para iniciar la construcción de aplicaciones `single page`, las cuales no requieren recargar la página para realizar sus funciones.

Para instalar `react-create-app` vamos a hacer uso de `npm`, primero ejecutaremos el comando en la línea de comandos:

### Listado 1.3 Instalación de create-react-app

```
npm install -g create-react-app
```

Esto nos va a arrojar una salida como la siguiente:

### Figura 1.5 Salida de instalación create-react-app

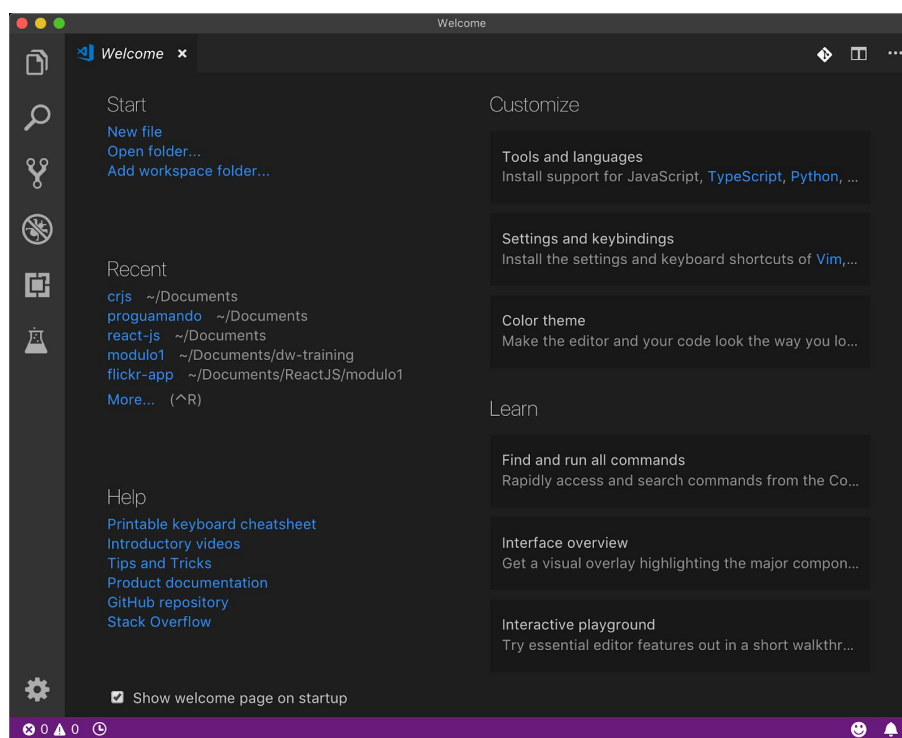
```
roquerueda ~ $ sudo npm install -g create-react-app
Password:
/usr/local/bin/create-react-app -> /usr/local/lib/node_modules/create-react-app/
index.js
+ create-react-app@2.0.3
added 63 packages from 20 contributors in 3.305s
roquerueda ~ $
```

Una vez terminado podemos continuar con la creación de nuestra aplicación

## Crear una página web utilizando HTML y JavaScript

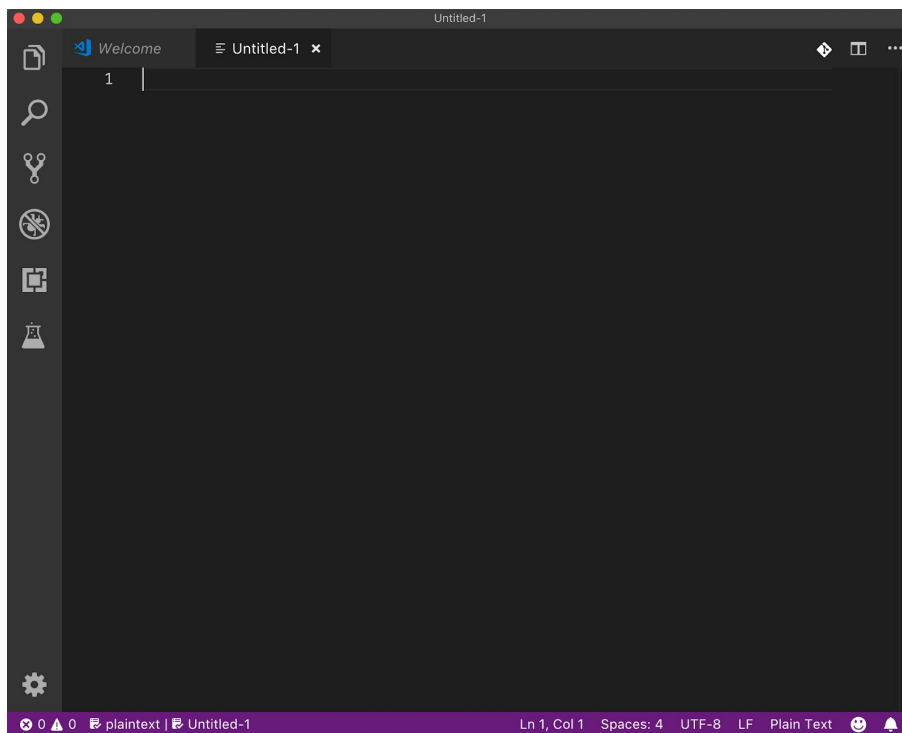
El primer paso para crear una página web es crear un archivo y añadir etiquetas HTML para poder visualizarla mediante un navegador. Para iniciar con la creación de nuestra primera página web, primero vamos a abrir Visual Studio Code.

Figura 1.6 Bienvenida a Visual Studio Code



En el panel izquierdo, seleccionar **New File**. En caso de no ver la opción se puede ir al menú **File -> New File**. Se debe visualizar el editor de texto sin contenido.

Figura 1.7 Nuevo archivo en VSCode



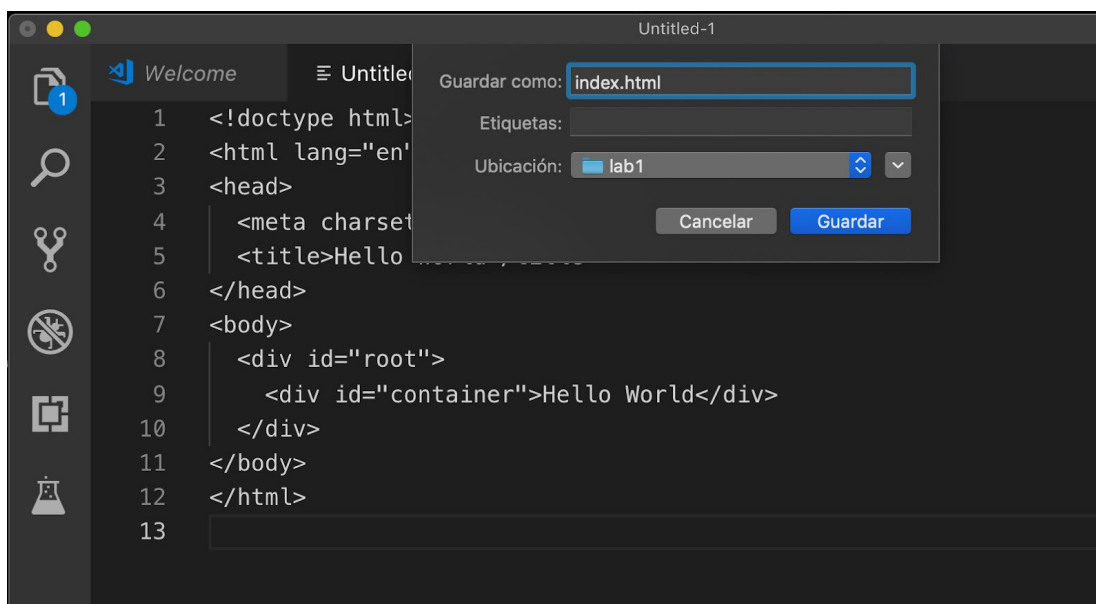
Ahora vamos a teclear el siguiente código en nuestro editor:

#### Listado 1.4 Etiquetas HTML para la página Hola Mundo

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Hello World</title>
</head>
<body>
  <div id="root">
    <div id="container">Hello World</div>
  </div>
</body>
</html>
```

Vamos a guardar el archivo presionando `CMD + S` en mac OS o `Ctrl + S` en Windows y lo llamaremos `index.html`, el archivo lo vamos a colocar en una carpeta llamada `01-hello-world`.

**Figura 1.8** Guarda pagina `index.html`

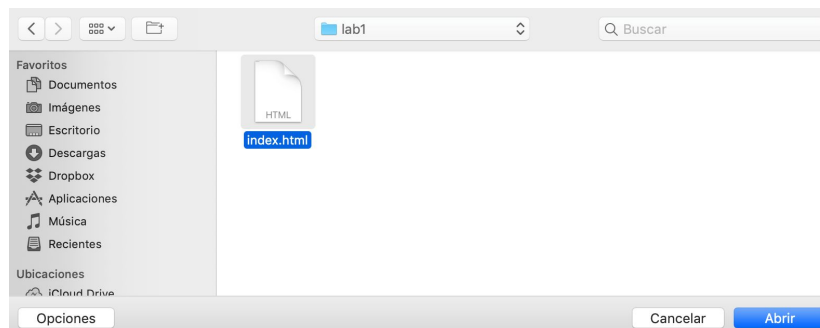


Una vez guardada vamos a notar colores en nuestro editor, esto se debe a que VSCode reconoce el tipo de archivo por su extensión y nos brinda ayuda visual para las palabras reservadas así como los valores de los atributos de las etiquetas HTML.

Ahora para visualizar el contenido de nuestro archivo vamos a abrir una ventana del navegador y abrir nuestra carpeta que contiene el archivo `index.html` que acabamos de crear.

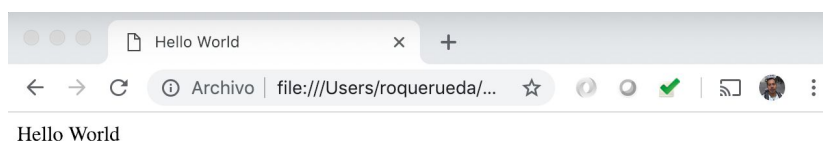
**Figura 1.9** Abrir nuestra página en el navegador





Una vez que nuestro navegador despliega el archivo vamos a visualizar una pantalla como la siguiente (Google Chrome):

**Figura 1.10** Abrir nuestra página en el navegador



## Funcionamiento de una página web

Cuando el navegador accede al archivo `index.html`, se interpretan las etiquetas HTML (Hypertext Markup Language) las cuales brindan la estructura del documento, HTML cuenta con etiquetas estandarizadas para definir las secciones del documento y se representan utilizando etiquetas entre los caracteres: `< >`.

Cada elemento HTML se representa con su etiqueta de inicio y fin como por ejemplo:

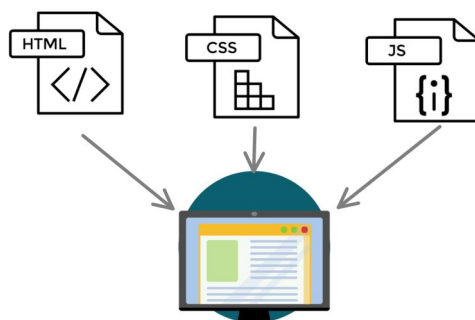
### Listado 1.5 Ejemplo de etiqueta HTML

```
<h1>This is a Heading</h1>
```

La intención de HTML no es contener etiquetas para dar formato a los contenidos de un documento, para dar formato se utiliza CSS (Cascading Style Sheets) que nos permite determinar cómo se va a presentar el documento en la pantalla.

Y por último para brindar interacción con el usuario y presentar contenido dinámico se utiliza JavaScript (JS), con JavaScript tenemos acceso a un API (Application Program Interface) conocida como DOM (Document Object Model). Utilizando el DOM podemos desde crear contenido en el documento hasta reaccionar a los eventos del usuario.

**Figura 1.11 Diagrama de elementos que construyen una página web**



## Agregar estilos a nuestro sitio

A continuación vamos a agregar un poco de **CSS** a nuestra página para estilizar el contenido de nuestro texto. Primero vamos a editar nuestro código del archivo `index.html`, en el listado siguiente el código en negritas es el que se agrega al contenido que ya se tenía en el archivo.

**Listado 1.6 Agregando estilo a nuestra página**

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Hello World</title>
  <link href="https://fonts.googleapis.com/css?family=Nunito"
  rel="stylesheet">
  <style>
    body {
      background-color: #0e293c
    }
  </style>
</head>
<body>
```

```
.title {  
  color: #19bbd5;  
  font-family: 'Nunito', sans-serif;  
  font-size: 300%  
}  
</style>  
</head>  
<body>  
  <div id="root">  
    <div id="container" class="title">Hello World</div>  
  </div>  
</body>  
</html>
```

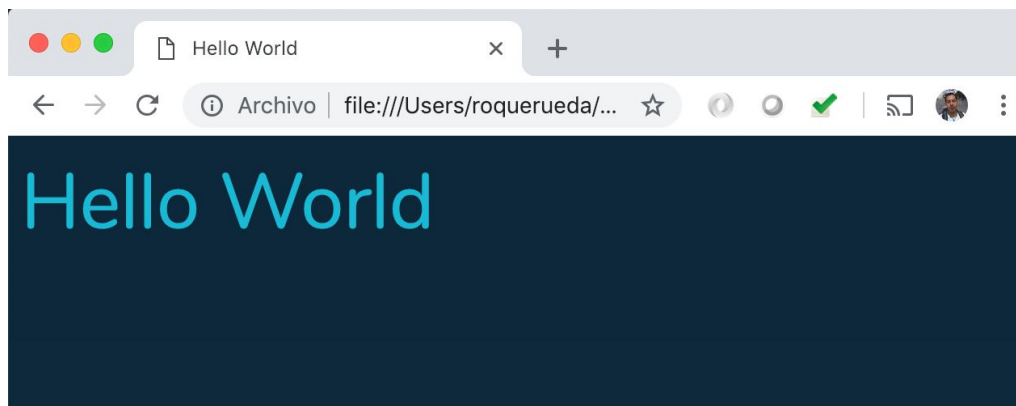
Al abrir nuestra página vamos a ver que se presenta de modo diferente el contenido del documento. La sintaxis de CSS es la siguiente:

Figura 1.12 Sintaxis CSS

Selector	Propiedad	Valor
body	{ background-color:	#0e293c }

Se debe especificar un `selector` el cual indica el tipo de elemento que será afectado, en este caso se indica el elemento `body` por lo que se apunta al cuerpo de la página, seguido entre llaves `{ }` se encuentran declarados las propiedades y los valores que estas tendrán cada propiedad y su valor van separadas por dos puntos `:` y cada nueva propiedad si se desea se separa con punto y coma `;` la última propiedad ya no requiere colocar punto y coma.

El selector que inicia con el punto `.title` indica una clase, es decir que afectara aquellos elementos que su atributo `class` tenga el valor `title`, esto se puede ver en nuestro ejemplo de la etiqueta `<div id="container" class="title">` y al visualizar nuestra página en el navegador nos presenta el texto con un nuevo formato.

**Figura 1.13 Abrir página con CSS**

## Agregar JavaScript a nuestro sitio

Para poder incluir código en de JavaScript existe una etiqueta llamada `<script>` `</script>` la cual indica al navegador que se interprete como JavaScript el contenido. En el siguiente listado el código que se encuentra tachado es código que se debe eliminar del archivo y el código que se encuentra en negritas es código que se debe agregar al archivo.

### Listado 1.7 Agregando JavaScript a nuestra página

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Hello World</title>
  <link href="https://fonts.googleapis.com/css?family=Nunito"
rel="stylesheet">
  <style>
    body {
      background-color: #0e293c
    }
    .title {
      color: #19bbd5;
```

```

        font-family: 'Nunito', sans-serif;
        font-size: 300%
    }
</style>
</head>
<body>
    <div id="root">
        <div id="container" class="title">Hello World</div>
    </div>
    <script>
        // Get the element root
        const rootElement = document.getElementById('root');
        // Create a new element
        const element = document.createElement('div');
        // Add some text
        element.textContent = 'Hello World';
        // Add class name to get the styles
        element.className = 'title';
        // Append the element
        rootElement.appendChild(element);
    </script>
</body>
</html>

```

Se está generando el contenido de la página utilizando JavaScript. La primer función que utilizamos `const rootElement = document.getElementById('root');` está accediendo al DOM mediante la variable `document`, la función `getElementById` recibe por parámetro el identificador que se desea buscar y el resultado se almacena en la variable `rootElement`, cabe señalar que la variable es una constante ya que fue declarada utilizando la palabra `const`, al ser una constante su valor no puede ser asignado nuevamente. La función

`createElement` recibe como parámetro el tipo de elemento que se desea generar, en este caso un elemento `'div'`. Posteriormente a el elemento creado se le asigna el contenido mediante la propiedad `textContent` y la clase CSS mediante la propiedad `className` para que el estilo afecte a este elemento. Finalmente, agregamos dicho elemento al elemento root en la línea `rootElement.appendChild(element);` lo que genera que nuestro elemento generado mediante JavaScript se presente en la página.

## Agregar React a nuestra página Hello World

Hemos creado un elemento utilizando JavaScript sin embargo aún no hacemos uso de las bibliotecas de React, para hacer uso de las mismas vamos a agregar dos etiquetas `<script>` más en nuestro archivo. Las etiquetas `<script>` pueden incluir un script utilizando el atributo `src` y su valor es dirección donde se encuentra el script que se va a incluir, es importante no olvidar la etiqueta de cierre `</script>`. En el listado siguiente las líneas en negritas se agregan a nuestro archivo y se han omitido algunas líneas para ahorro de espacio.

### Listado 1.8 Agregar CDN para React y ReactDOM

```
// ...
<body>
  <div id="root"></div>
  <script src="https://unpkg.com/react@16/umd/react.
development.js" crossorigin></script>
  <script src="https://unpkg.com/react-dom@16/umd/react-dom.
development.js" crossorigin></script>
  <script>
    // Get the element root
    const rootElement = document.getElementById('root');
    // ...
    // Append the element
    rootElement.appendChild(element);
  </script>
</body>
</html>
```

Con la primer línea:

### Listado 1.9 Script tag para incluir React

```
<script src="https://unpkg.com/react@16/umd/react.development.js" crossorigin></script>
```

Este script nos da acceso a la variable global **React** la cual nos permite acceso al API de react, esta API contiene todo lo referente a Componentes, ciclo de vida y estados, la segunda línea:

### Listado 1.10 Script tag para incluir React DOM

```
<script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js" crossorigin></script>
```

Este script brinda acceso a ReactDOM lo que nos permite interactuar con el documento HTML y renderizar los componentes en nuestra página. Para hacer uso de React y ReactDOM vamos a cambiar el contenido de nuestro archivo para que se cree el contenido utilizando React.

### Listado 1.11 Utilizando React.createElement y ReactDOM.render

```
// ...
<body>
  <div id="root"></div>
  <script src="https://unpkg.com/react@16/umd/react.development.js" crossorigin></script>
  <script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js" crossorigin></script>
  <script>
    // Get the element root
    const rootElement = document.getElementById('root');
    // Create a new element
    const element = document.createElement('div');
    // Add some text
  </script>
```

```
element.textContent = 'Hello World';  
// Add class name to get the styles  
element.className = 'title';  
// Append the element  
rootElement.appendChild(element);  
  
// Create an element using React  
const element = React.createElement('div',  
  { className: 'title' }, 'Hello World React');  
  
// Render the element inside the root  
ReactDOM.render(element, rootElement);  
  
</script>  
</body>  
</html>
```

Las nuevas líneas agregadas a nuestro archivo indican a React que cree un nuevo elemento, la función `React.createElement` indica la creación de un nuevo elemento, el valor de retorno es un objeto que contiene toda la información necesaria para representar dicho elemento en la pantalla. Así mismo la función `createElement` recibe 3 parámetros, el primero indica el tipo de elemento, el segundo las propiedades del elemento y el tercero indica los elementos hijos del elemento. Vamos a explicar cada uno a detalle más adelante en nuestra capacitación.

**Figura 1.14** Parametros de `React.createElement`

```
React.createElement(  
  type,  
  [props],  
  [...children]  
)
```



## Crear componente Hello World utilizando JSX

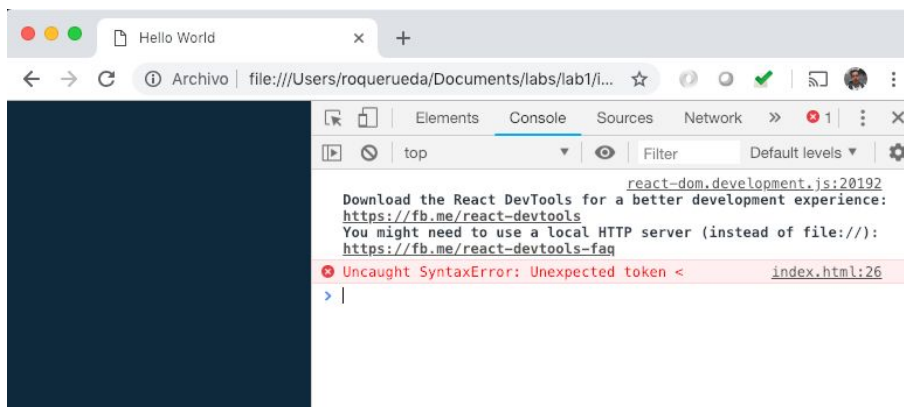
Ahora para crear nuestro elemento vamos a utilizar JSX, es una extensión de JavaScript que nos permite generar elementos en la UI de forma más familiar; React puede ser utilizado sin JSX sin embargo su uso es bastante recomendado por todos los beneficios que trae consigo. Vamos a modificar la creación de nuestro elemento:

### Listado 1.12 Creando un elemento utilizando JSX

```
// ...
<script>
  // Get the element root
  const rootElement = document.getElementById('root');
  // Create an element using React
  const element = React.createElement('div',
  { className:'title' }, 'Hello World React');
  // Create an element using JSX
  const element = <div className="title">Hello World</div>;
  // Render the element inside the root
  ReactDOM.render(element, rootElement);
</script>
</body>
</html>
```

Ahora vamos a abrir nuestro navegador y vamos a abrir las herramientas de desarrollo de nuestro navegador (En Chrome abrir en el menú superior derecho abrir Más Herramientas -> Herramientas del Desarrollador) y vamos a la consola para verificar la salida, veremos una pantalla como la siguiente:

### Figura 1.15 Error al interpretar JSX en la página



Podemos ver que el navegador reporta un error de sintaxis apuntando a la línea que recién modificamos, esto es porque JavaScript no reconoce la etiqueta `<div className="title">Hello World</div>`; debido a que el navegador no cuenta con soporte para interpretar esta sintaxis. Para resolver este error vamos a agregar una etiqueta más a nuestra pagina que nos ayudara a transformar nuestro código para que pueda ser interpretado por el navegador.

### Listado 1.13 Usando Babel

```
// ...
<body>
  <div id="root"></div>
  <script src="https://unpkg.com/react@16/umd/react.
    development.js" crossorigin></script>
  <script src="https://unpkg.com/react-dom@16/umd/react-dom.
    development.js" crossorigin></script>
  <script src="https://unpkg.com/babel-standalone@6/babel.
    min.js"></script>
  <script type="text/babel">
    // Get the element root
    const rootElement = document.getElementById('root');
    // Create an element using JSX
```

```
const element = <div className="title">Hello World</div>;  
// Render the element inside the root  
ReactDOM.render(element, rootElement);  
</script>  
</body>  
// ...
```

Podemos observar que la se incluye un nuevo script que apunta a babel en la línea:

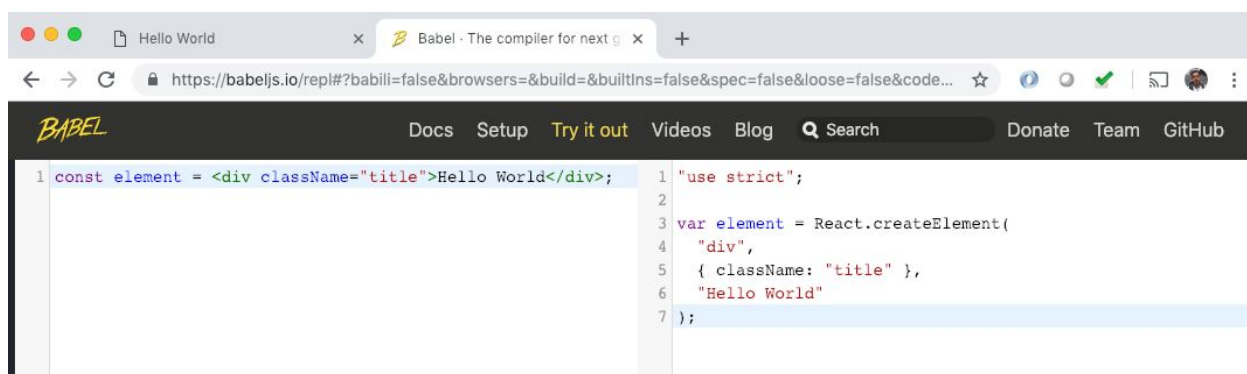
#### Listado 1.14 Script tag para incluir Babel

```
<script src="https://unpkg.com/babel-standalone@6/babel.  
min.js"></script>
```

Y también se debe indicar que nuestro script no se interpretado como javascript sino que sea interpretado por babel para esto se modifica el atributo type quedando `type="text/babel"` lo que indica que el contenido de la etiqueta script sea transformada por babel.

Babel toma JSX y genera llamadas a la función `createElement` de React, para poder visualizar esto vamos a ingresar al sitio: <https://babeljs.io/> una vez ahi vamos a hacer click en la opción **Try it out** y vamos a colocar nuestro código en el editor izquierdo para visualizar cómo es transformado por Babel

#### Figura 1.16 Babel transformando JSX a JavaScript en versiones anteriores



El beneficio de usar JSX es que nos ayuda a genera de forma familiar los componentes de interfaz además de simplificar las llamadas en caso de contar con componentes anidados.

---

# LAB 02: Composición de Componentes

---

## Objetivos

- Crear una aplicación utilizando react-create-app
- Crear functional components
- Crear class components
- Aplicar los conceptos de composición de elementos en una aplicación

## Crear una aplicación utilizando react-create-app

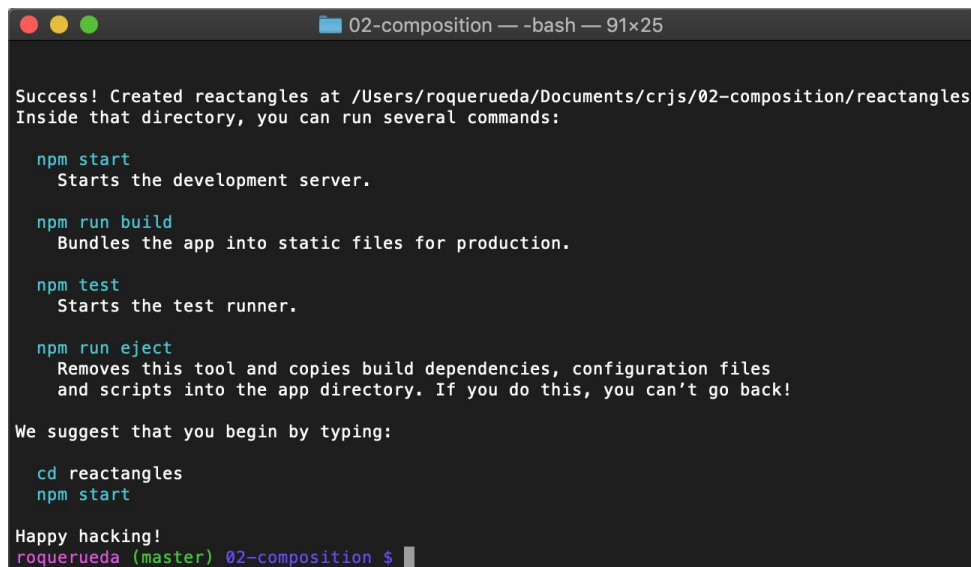
Para iniciar con este laboratorio vamos a crear una nueva carpeta llamada 02-composition y navegar a dicha carpeta, una vez dentro de la carpeta vamos a ejecutar desde la terminal el siguiente comando:

### Listado 2.1 Crear una aplicación utilizando create-react-app

```
npx create-react-app reactangles
```

Es importante notar que el comando inicia con `npx` y no `npm` además el comando `create-react-app` recibe como parámetro el nombre del proyecto en este caso es "reactangles" al finalizar la ejecución del comando vamos a ver una pantalla similar a la siguiente:

### Figura 2.1 Salida del comando create-react-app

A terminal window titled '02-composition --bash-- 91x25'. It displays a success message: 'Success! Created reactangles at /Users/roquerueda/Documents/crjs/02-composition/reactangles. Inside that directory, you can run several commands:'. It lists four commands: 'npm start' (Starts the development server.), 'npm run build' (Bundles the app into static files for production.), 'npm test' (Starts the test runner.), and 'npm run eject' (Removes this tool and copies build dependencies, configuration files and scripts into the app directory. If you do this, you can't go back!). It suggests starting with 'cd reactangles' followed by 'npm start'. It ends with 'Happy hacking!' and a prompt 'roquerueda (master) 02-composition \$'.

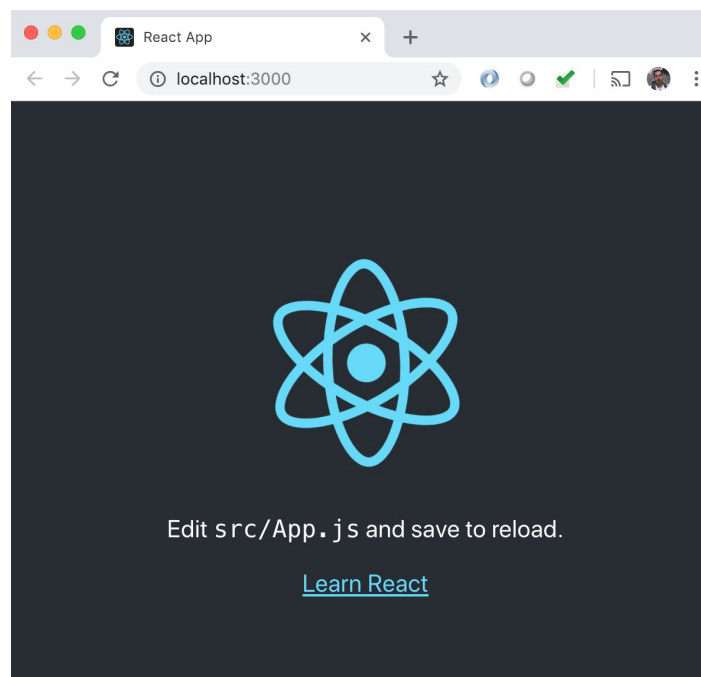
Podemos ver al final que se nos sugiere navegar al folder e iniciar la aplicación, vamos a navegar al folder de nuestra aplicación y posteriormente vamos a iniciar nuestra aplicación. Para esto vamos a ejecutar en la terminal:

### Listado 2.2 Iniciar aplicación reactangles

```
npm start
```

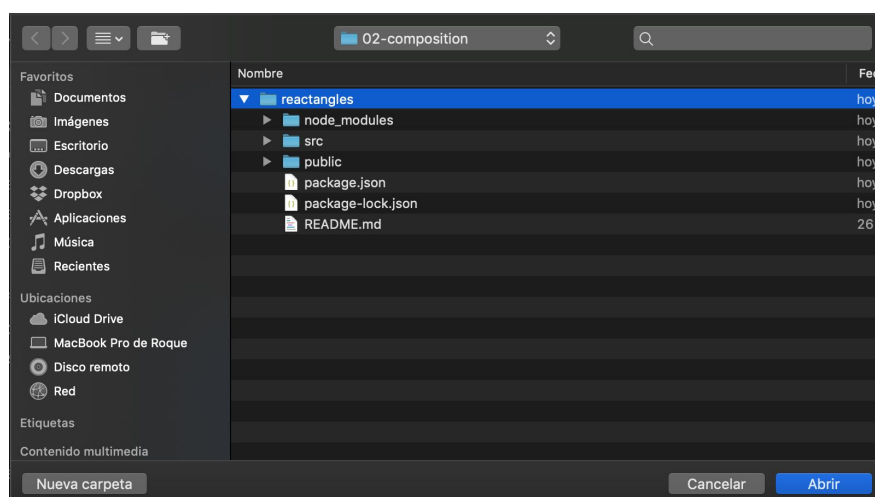
Se abre una nueva ventana de nuestro navegador la cual va a la dirección `http://localhost:3000/` podemos ver que está ingresando a localhost en el puerto 3000 es donde nuestra aplicación responderá a las peticiones del navegador.

**Figura 2.2** Abrir la aplicación en el navegador



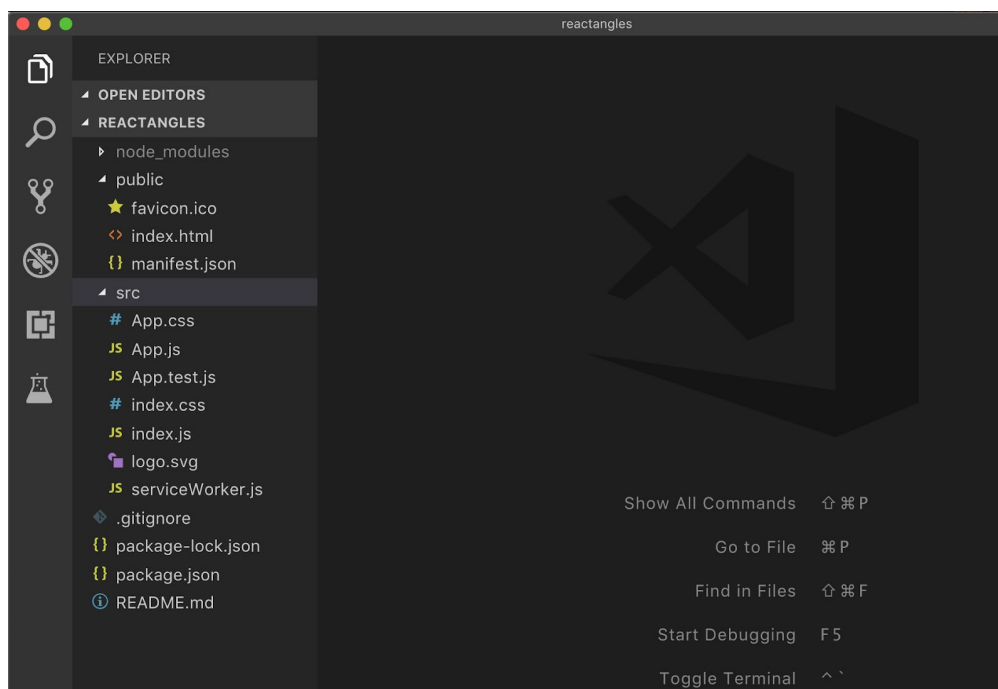
Ahora vamos a abrir nuestra aplicación en nuestro editor, para eso vamos a abrir VSCode y vamos a abrir la carpeta donde se encuentra nuestro proyecto la cual creamos previamente 02-composition/**reactangles**

Figura 2.3 Abrir carpeta en VSCode



Y cuando se abre la carpeta vamos a ver en nuestro lado izquierdo de la ventana de VSCode el explorador de archivos nos muestra diferentes archivos, los cuales en su conjunto crean nuestra aplicación react, vamos a explicar la estructura de nuestro proyecto:

Figura 2.4 Explorador de VSCode



Los elementos principales de nuestra aplicación son los siguientes:

Tabla 2.1 Archivos del proyecto reactangles

Carpeta contenedora del proyecto	node_modules	Carpeta que contiene las dependencias de nuestra aplicación
	public	Carpeta que contiene los archivos estáticos del proyecto como imágenes
	src	Carpeta donde se coloca todo el código fuente que escribimos
	.gitignore	Especifica los archivos que no se les dará seguimiento por parte de Git
	package-lock.json	Mantiene una lista de la versión exacta que fue instalada
	package.json	Mantiene una lista de las dependencias del proyecto así como configuración del mismo
	README.md	Instrucciones de nuestro proyecto

El punto de entrada de nuestra aplicación se encuentra en: `reactangles/public/index.html` en este archivo podemos visualizar que es un archivo simple de HTML el cual es el contenedor de nuestra aplicación React. Usualmente no vamos a editar este archivo de forma directa sin embargo es importante saber su localización.

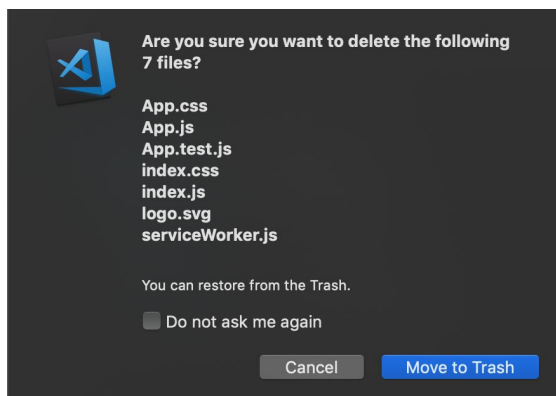
Dentro del archivo `reactangles/src/index.js` podemos ver como se invoca la función `ReactDOM.render` la cual es la encargada de presentar el componente de React raíz de nuestra aplicación, en este caso llamado `App` por convención.

El archivo `reactangles/src/App.js` es el módulo inicial de la aplicación, este módulo es una clase ES6 que extiende de `React.Component`, podemos ver que la primera línea del archivo está importando `React` y a su vez `React.Component` utilizando un destructuring `{ Component }` también se importa el logo y la hoja de estilos CSS, podemos también ver que el método `render` está generando JSX el cual es el resultado que observamos en la página.

## Crear functional components

Para iniciar vamos a eliminar los archivos contenidos en el folder `reactangles/src/` para poder iniciar con el desarrollo de la aplicación de este laboratorio.

**Figura 2.5 Eliminar archivos generados automáticamente de la carpeta src**



Una vez que la carpeta está vacía vamos a generar un nuevo archivo llamado `index.js` es importante que el nombre sea el adecuado. Vamos a generar nuestra aplicación siguiendo los siguientes pasos:

1. Importar React y ReactDOM
2. Crear un componente React
3. Mostrar el componente en pantalla.



Vamos a escribir las siguientes líneas en nuestro archivo `reactangles/src/index.js`:

### Listado 2.2 Importar módulos React y ReactDOM

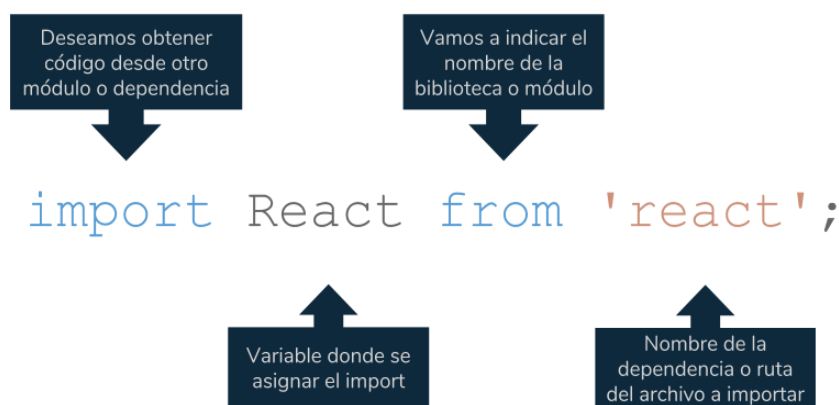
```
// 1. Import React and ReactDOM
import React from 'react';
import ReactDOM from 'react-dom';

// 2. Create component

// 3. Display component
```

Como podemos notar se está utilizando la sintaxis de `import` para que nuestro módulo pueda utilizar la funcionalidad de `React` y `ReactDOM` cada módulo es independiente de los demás es por ello que debemos explícitamente indicar que requerimos hacer uso de ellos.

Figura 2.6 Sintaxis de la sentencia `import`



Ahora vamos a generar un componente, un componente puede ser una función o una clase que genera HTML para presentar al usuario y maneja los eventos del usuario. Para iniciar vamos a crear un componente simple utilizando una función que despliegue un mensaje al usuario para

verificar que nuestra aplicación funcione correctamente, vamos a modificar nuestro archivo `reactangles/src/index.js`.

### Listado 2.3 Crear un componente utilizando una función

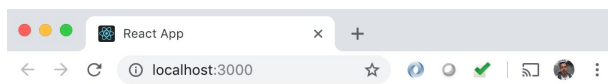
```
// 1. Import React and ReactDOM
import React from 'react';
import ReactDOM from 'react-dom';

// 2. Create component
const App = function() {
  return (
    <h1>Reactanlges</h1>
  );
};

// 3. Display component
ReactDOM.render(
  <App />,
  document.querySelector('#root')
);
```

Si abrimos nuestro navegador vamos a visualizar que se muestra únicamente el texto Reactangles en nuestra página:

### Figura 2.7 Visualizar Reactangles en nuestra página



## Reactanlges

El componente que definimos utilizando una función también pudo ser definido utilizando un `arrow function` que es una sintaxis más corta de una función, además de que no cuenta con `this`, `arguments`, `super`, o `new.target` y siempre son anonimas. Vamos a modificar nuestro archivo `reactangles/src/index.js` una vez más.

### Listado 2.4 Utilizar arrow function

```
// ...  
const App = function() => {  
  return (  
    <h1>Reactanlges</h1>  
  );  
};  
// ...
```

Este cambio básicamente nos genera el mismo resultado ya que el valor de `App` sigue siendo una función que genera JSX y el código es equivalente. Vamos a ahora a agregar una nueva carpeta dentro de `src` la cual llamaremos `components` quedando del siguiente modo: `reactangles/src/components` y dentro de la carpeta `components` vamos a crear un archivo llamado `Rectangle.js` dentro de la carpeta quedando la ruta del archivo `reactangles/src/components/Rectangle.js` en dicho archivo vamos a agregar el siguiente código:

### Listado 2.5 Componente Rectangulo

```
import React from 'react';
```

```
const Rectangle = (props) => {  
  return (  
    <div>  
      <p>Rectangle</p>  
      <br />  
      <p>height={props.height}</p>  
      <p>width={props.width}</p>  
      <p>area={props.height*props.width}</p>  
    </div>  
  );  
};  
  
export default Rectangle;
```

Este nuevo archivo es un componente que se define mediante una función flecha, la cual recibe como parámetro un objeto llamado `props`, este objeto contiene la información que se envía a nuestro componente, por convención se denomina `props` ya que hace alusión a las propiedades con las que se genera el componente.

Es importante notar que se está incluyendo valores de JavaScript dentro de JSX utilizando llaves `{ }` para denotar que el ese valor proviene desde JavaScript a esto se le llama interpolación. La interpolación acepta cualquier valor que pueda ser evaluado como una expresión de JavaScript es decir se pueden poner valores `{ 'Hola' }` **cadena**, `{ 123 }` **numéricos**, `{ true }` **booleanos** hasta `{ ( ) => return true }` **funciones** o inclusive otros **objetos** de JavaScript `{ { one: 1, two: 2 } }` siempre y cuando el valor del atributo lo permita, esta es una característica poderosa que nos permite incrustar lógica dentro de nuestro JSX.

La última línea está indicando que nuestro módulo puede ser utilizado por otros módulos `export default Rectangle`. Para poder utilizar ese componente que generamos recientemente vamos a modificar nuestro archivo `reactangles/src/index.js` y vamos a indicar que se importe el módulo y crear una instancia de dicho componente.

## Listado 2.6 Utilizando un componente definido en otro módulo

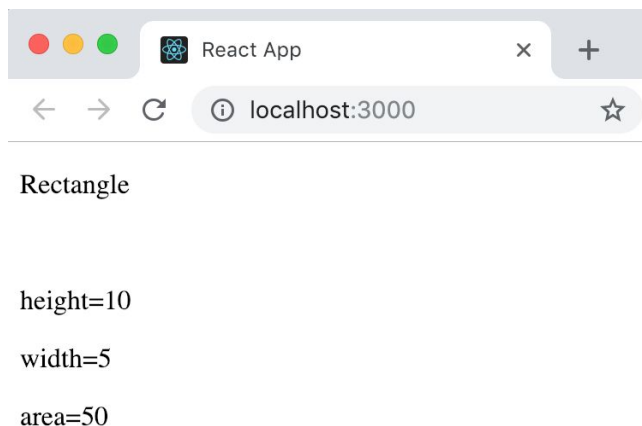
```
// 1. Import React and ReactDOM
import React from 'react';
import ReactDOM from 'react-dom';
import Rectangle from '../components/Rectangle';

// 2. Create component
const App = function() {
  return (
    <h1>Reactanlges</h1>
    <Rectangle height="10" width="5" />
  );
};

// 3. Display component
ReactDOM.render(
  <App />,
  document.querySelector('#root')
);
```

Primero vamos a importar el componente que fue definido por nosotros en el archivo `reactangles/src/components/Rectangle.js` para eso la sentencia `import` hace alusión a la ruta del archivo no solo al nombre de la dependencia como las sentencias `import` anteriores que solo indican el nombre `react` o `react-dom`, también podemos observar que cuando creamos nuestro componente `<Rectangle height="10" width="5" />` se están agregando dos atributos `height` y `width` los cuales conforman las llaves del objeto `props` que se recibe como parámetro en nuestro componente. Al declarar `const Rectangle = (props) => { //..` Se indica que se recibirá un objeto `props`, React toma todos los atributos definidos en JSX y genera un objeto JavaScript el cual contiene los valores y es por ello que al ver nuestra página en el navegador podemos observar los valores del rectángulo.

**Figura 2.8 Ver nuestro componente con propiedades**



Ahora vamos a editar nuestro componente rectángulo para que los valores de las propiedades afecten de una forma más visual nuestro elemento, vamos a nuestro archivo `rectangles/src/components/Rectangle.js` y escribimos el siguiente código:

#### Listado 2.7 Actualizar estilo de rectángulo

```
import React from 'react';

const Rectangle = (props) => {
  return (
    <div>
      <p>Rectangle</p>
      <br />
      <p>height={props.height}</p>
      <p>width={props.width}</p>
      <p>area={props.height*props.width}</p>
    </div>
    <div style={{ backgroundColor: '#0e293c',
      color: '#c6daec', height:props.height,
      width:props.width }} />
  )
}
```

```
);  
};  
  
export default Rectangle;
```

Podemos ver que estamos ahora estamos afectado el atributo `style` utilizando interpolación, el atributo `style` recibe un objeto de JavaScript donde cada llave o nombre de propiedad será el nombre del estilo a afectar. Por ejemplo si se desea modificar el color de fondo de HTML existe el atributo `background-color` pero en React se llama `backgroundColor`, esta notación se conoce como **camelCase** dentro de JavaScript.

Debido a que los atributos `height` y `width` aceptan solo valores numéricos vamos a modificar también nuestro archivo `rectangles/src/index.js` escribiendo el siguiente código:

#### Listado 2.8 Utilizando un componente definido en otro módulo

```
// ...  
// 2. Create component  
const App = function() {  
  return (  
    <Rectangle height="10" height={150} width="5" width={300} />  
  );  
};  
// ...
```

Al visualizar nuestra página en el navegador vamos a encontrar que se muestra un rectángulo en la página.

## Crear class components

Los componentes que se generan mediante una clase tienen características adicionales. El **estado** es similar a las **propiedades** pero es privado y totalmente controlado por el componente. Para definir un componente utilizando una clase se debe:

- Debe extender de `React.Component`
- Debe contener por lo menos un método `render`
- Utilizamos la palabra reservada `this` para acceder a las propiedades del componente.
- Si se crea un constructor se debe recibir un parámetro `props` y hacer el llamado a `super` en la primer sentencia del constructor

Para definir un `class` component vamos a crear un nuevo archivo en nuestra carpeta `components` al cual vamos a llamar `Pyramid.js` quedando del siguiente modo la ruta del archivo `reactangles/src/components/Pyramid.js` y vamos a escribir el siguiente código:

### Listado 2.9 Creando el componente Pyramid

```
import React, { Component } from "react";
import Rectangle from './Rectangle'

class Pyramid extends Component {

  constructor(props) {
    super(props);
    const { levels } = props
    const generatedFloors = [];
    for(let i = 0; i < levels; i ++) {
      generatedFloors.push(
        <Rectangle
          height={20}
          width={20 * (i + 1)} />
      );
    }

    this.state = {
```



```
    floors: generatedFloors
  }
}

render() {
  return (
    <div>{this.state.floors}</div>
  );
}
}

export default Pyramid;
```

Podemos observar que en la primera línea donde estamos importando `React` y estamos también usando `destructuring` para importar `Component`, `destructuring` se puede definir como descomponer estructuras complejas en partes más simples. Al utilizar `destructuring` podemos descomponer una estructura compleja como lo es un objeto o un array y extraer partes más pequeñas. A continuación se presenta un breve listado para explicar `destructuring` de un objeto, cabe mencionar que `destructuring` puede ser utilizado para **declaración de variables** o para **asignación de variables**.

#### Listado 2.10 Explicación destructuring

```
const person = { name: 'Roque', age: 30 };
// Obtener las propiedades name y age
const { name, age } = person;
// equivalente
const name = person.name; const age = person.age;
```

En el listado anterior podemos visualizar que se cuenta con un objeto el cual tiene 2 propiedades, posteriormente se crean dos variables las cuales van a tomar los valores de las propiedades `name` y `age`. El código debajo es equivalente al `destructuring`.

Cuando se define un componente utilizando una clase ES6 es importante que se extienda de `React.Component`, debido a que se utilizó destructuring podemos solo indicar únicamente `Component` para que nuestro código sea más corto. También podemos ver que en nuestra clase se define un constructor, en una clase ES6 solo puede existir un método denominado constructor y podemos ver que la primer sentencia es invocar a `super` enviando las propiedades como parámetro. En caso de no invocar a `super` obtendremos un error.

Posteriormente se usa `destructuring` nuevamente para asignar el valor de `levels` extrayendolo del objeto `props` y se generan los niveles de la pirámide, los niveles de la pirámide se representan mediante un arreglo que se almacena en el **estado del componente**, dicho arreglo contiene los elementos JSX que se van a presentar en pantalla. Y podemos observar que en el método `render` de nuestra pirámide está usando interpolación enviando el el arreglo con los componentes `Rectángulo`. `React` es lo suficientemente inteligente para entender qué queremos presentar una secuencia de elementos que se encuentran en el arreglo.

Es muy importante entender que el **estado** del componente sólo debe ser afectado directamente en el `constructor`, en cualquier otro caso se debe utilizar la función `setState` de la cual se hablará más a detalle adelante.

Para hacer uso del **estado** es decir **lectura** de los valores del estado si podemos hacer uso directo de sus valores. Vamos a hablar a detalle del estado en el capítulo siguiente pero es importante tener en cuenta como utilizar el **estado**.

Ahora vamos a editar nuestro archivo `reactangles/src/index.js` para hacer el uso de la Pirámide en lugar del componente `Rectángulo`, vamos a escribir el siguiente código:

### Listado 2.11 Utilizando componente Pyramid

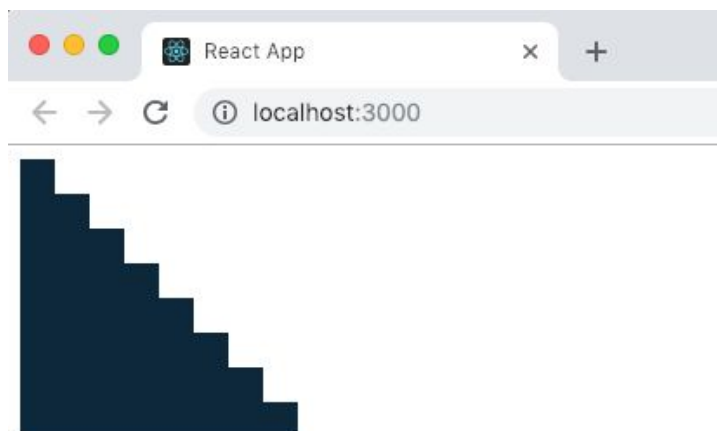
```
// 1. Import React and ReactDOM
import React from 'react';
import ReactDOM from 'react-dom';
import Rectangle from './components/Rectangle';
import Pyramid from './components/Pyramid';

// 2. Create component
const App = function() {
```

```
return (  
  <Rectangle height={150} width={300} />  
  <Pyramid levels={8} />  
);  
};  
  
// 3. Display component  
ReactDOM.render(  
  <App />,  
  document.querySelector('#root')  
)
```

Y si visualizamos nuestra página veremos algo similar a lo siguiente:

**Figura 2.9** Mostrar component Pyramid



## Aplicar los conceptos de composición de elementos en una aplicación

Cuando nuestros componentes reciben el objeto `props` el cual contiene una llave llamada `children`, esta propiedad contendrá todos los elementos hijos que sean agregados dentro de

nuestra etiqueta JSX o bien asignados mediante atributos. Vamos a permitir ahora que nuestro elemento Piramide despliegue contenido el cual podría ser inclusive otro elemento de JSX.

Vamos a permitir que se puede agregar un elemento debajo de nuestra Pirámide, brindando la flexibilidad de que pueda ser cualquier elemento. Para esto vamos a modificar nuestro elemento `reactangles/src/components/Pyramid.js` agregando el siguiente código:

#### Listado 2.12 Utilizar la propiedad children

```
//...  
render() {  
  return (  
    <div>{this.state.floors}</div>  
    <div>  
      {this.state.floors}  
      <br />  
      {this.props.children}  
    </div>  
  );  
}  
}  
  
export default Pyramid;
```

Podemos ver que estamos utilizando interpolación para que en nuestro método render se presenten los elementos hijos en caso de existir, ahora vamos a editar nuestro archivo `reactangles/src/index.js` para enviar algunos elementos utilizando JSX. Vamos a escribir el siguiente código:

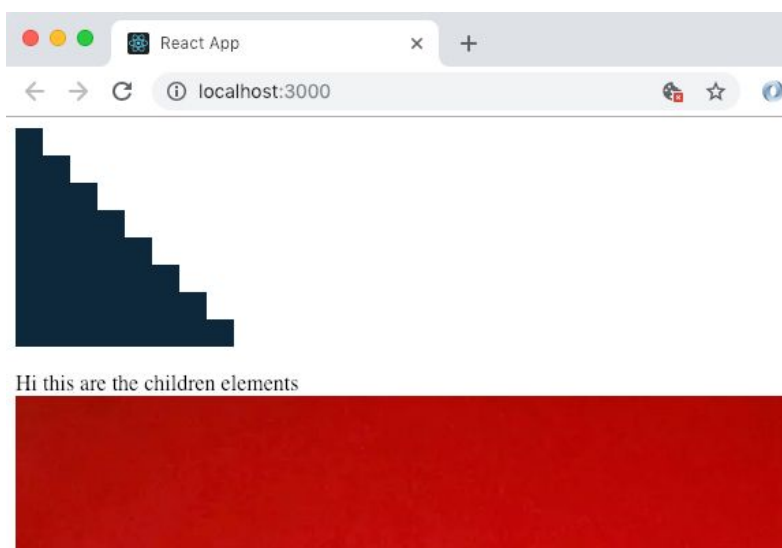
#### Listado 2.13 Enviando elementos hijos utilizando JSX

```
//...  
// 2. Create component  
const App = () => {
```

```
return (  
  <Pyramid levels={8} />  
  <Pyramid levels={8}>  
    Hi this are the children elements  
    <br/>  
      
  </Pyramid>  
);  
};  
  
//...
```

Podemos ver como nuestra etiqueta JSX ahora contiene 3 elementos internos `children`, una cadena, un salto de línea y una imagen, al visualizar la página vamos a ver nuestra pirámide y los elementos que están contenidos dentro de él debajo.

**Figura 2.10** Componente Pyramid con elementos children



Ahora vamos a crear un nuevo elemento el cual sea una Pirámide que use una imagen como fondo de nuestros rectángulos. Para eso vamos a editar nuestros componentes vamos a iniciar

con el componente `reactangles/src/components/Rectangle.js` al cual le vamos a agregar el siguiente código:

#### Listado 2.14 Actualizar estilo de rectángulo

```
import React from 'react';

const Rectangle = (props) => {
  return (
    

Como podemos apreciar le estamos agregando propiedades al estilo de nuestro componente Rectángulo para que pueda utilizar una imagen de fondo y que el fondo esté fijo al elemento en este caso div.



Para poder recibir la ruta de la imagen que deseamos mostrar como fondo vamos a modificar nuestro elemento reactangles/src/components/Pyramid.js para que envíe a los componentes Rectangle un atributo más en JSX



© 2019 DW Training S.C. Todos los derechos reservados.



37


```

### Listado 2.15 Enviar atributo de imagen mediante JSX a Rectangle

```
//...
class Pyramid extends Component {
  constructor(props) {
    super(props);
    const { levels } = props
    const { levels, bgImg } = props;
    const generatedFloors = [];
    for(let i = 0; i < levels; i ++) {
      generatedFloors.push(
        <Rectangle
          height={20}
          width={20 * (i + 1)} />
        bgImg={bgImg} />
      );
    }
  }
//...
```

Como podemos observar estamos extrayendo la propiedad `bgImg` del objeto `props` y después estamos enviando el valor utilizando un atributo JSX. Ahora vamos a generar un nuevo componente en la carpeta `components` el cual llamaremos `BackgroundPyramid.js` quedando la ruta del archivo de la siguiente manera: `reactangles/src/components/BackgroundPyramid.js` y vamos a escribir el siguiente código:

### Listado 2.16 Componente BackgroundPyramid.js

```
import React from 'react'
import Pyramid from './Pyramid'
```

```
const BackgroundPyramid = (props) => {
  return (
    <Pyramid levels={30} bgImg={'https://hmp.me/cf1j'}>
      This pyramid has background
    </Pyramid>
  );
}

export default BackgroundPyramid;
```

Como podemos observar BackgroundPyramid hace uso del componente Pyramid y define los atributos levels y bgImg con los cuales se desplegará una imagen de fondo en nuestra pirámide. Ahora solo debemos actualizar nuestro archivo reactangles/src/index.js para que haga uso de BackgroundPyramid vamos a escribir el siguiente código:

#### Listado 2.16 Componente BackgroundPyramid.js

```
// 1. Import React and ReactDOM
import React from 'react';
import ReactDOM from 'react-dom';
import Pyramid from '../components/Pyramid';
import BackgroundPyramid from '../components/BackgroundPyramid';

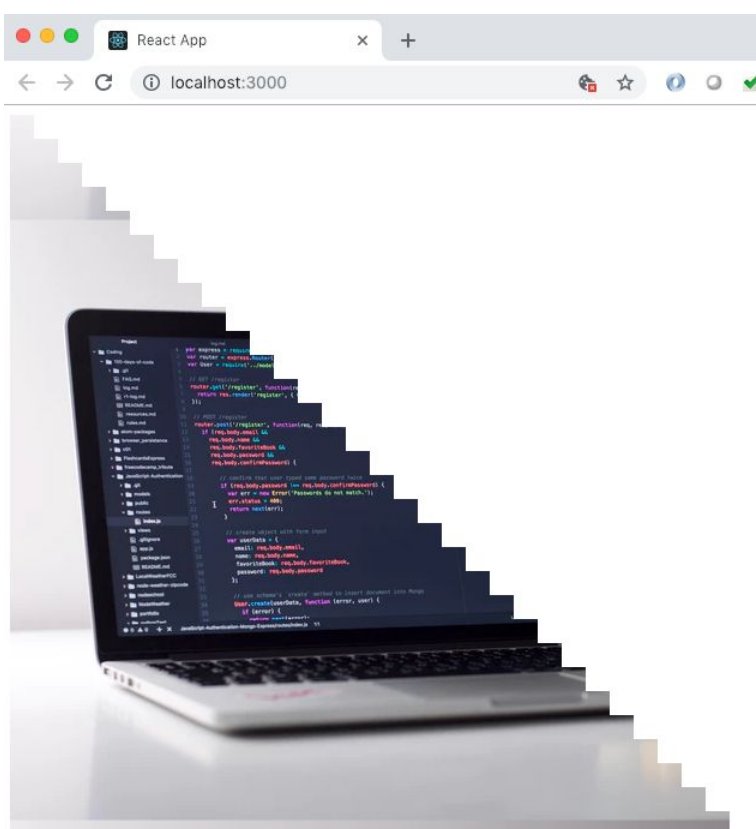
// 2. Create component
const App = () => {
  return (
    <Pyramid levels={30}>
    Hi this are the children elements
    <br/>
    <BackgroundPyramid />
  );
}
```



```
<BackgroundPyramid />  
  
);  
};  
//...
```

Si abrimos nuestra página podemos ver un resultado como el siguiente:

**Figura 2.11** Piramide con fondo y caption



This pyramid has background

## LAB 03: Pensando en React

### Objetivos

- Crear una aplicación siguiendo el flujo de pensando en React
- Crear una jerarquía de componentes
- Utilizar propiedades y estado

### Crear una aplicación utilizando el flujo de pensando en React

Para iniciar con este laboratorio vamos a crear una nueva carpeta llamada 03-thinking y navegar a dicha carpeta, una vez dentro de la carpeta vamos a ejecutar desde la terminal el siguiente comando:

#### Listado 3.1 Crear aplicación users

```
npx create-react-app users
```

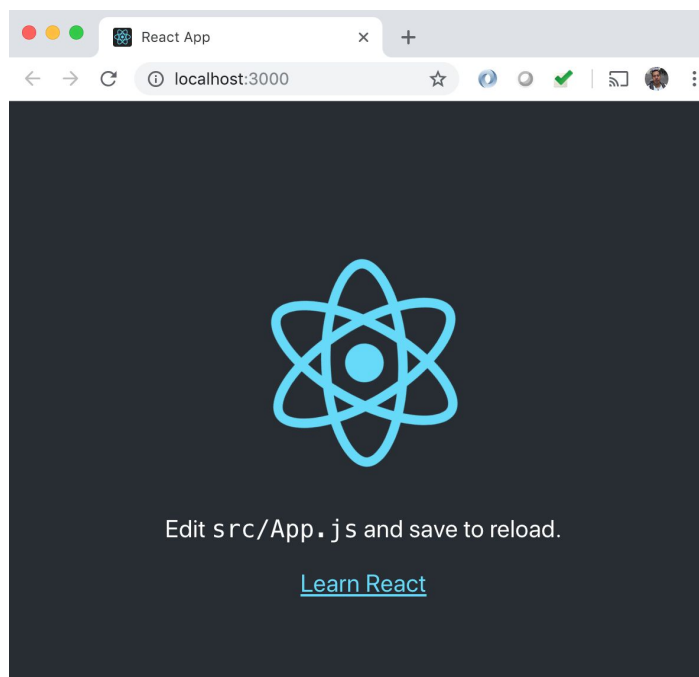
Vamos a navegar a nuestra carpeta e iniciar la aplicación para lo cual vamos a escribir el siguiente comando dentro de la carpeta `users`:

#### Listado 3.2 Iniciar aplicación users

```
npm start
```

Una vez que nuestra aplicación inicia vamos a verificar en nuestro navegador que podemos visualizarla. Abrimos la ruta: <http://localhost:3000/> y debemos poder ver una pantalla como la siguiente:

#### Figura 3.1 Abrir la aplicación users en el navegador

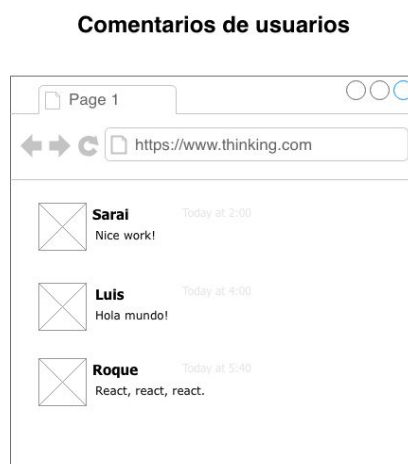


Ahora que tenemos todo listo vamos a iniciar con la construcción de nuestra aplicación para lo cual vamos seguiremos el flujo recomendado para crear una aplicación en react.

## Iniciar con un Mock

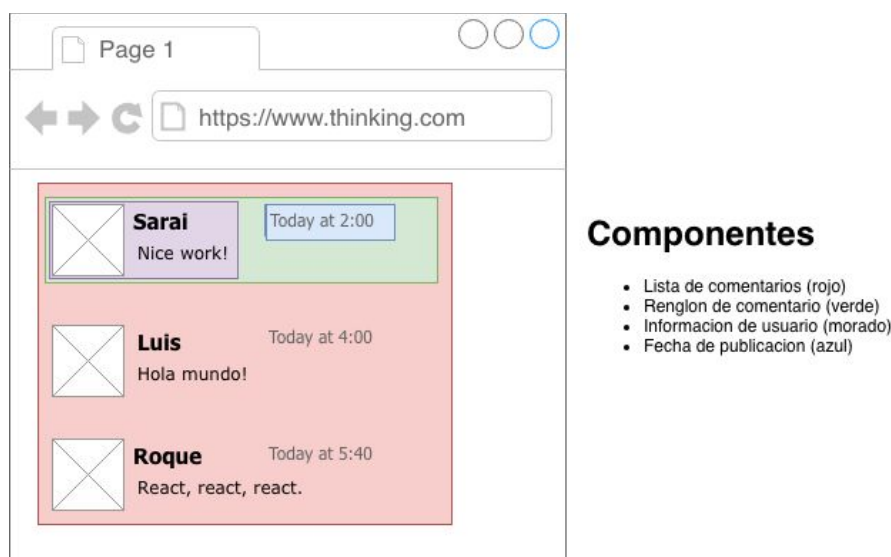
Para iniciar el desarrollo de nuestra aplicación se recomienda iniciar con una pantalla modelo la cual será nuestra guía durante el proceso, para nuestro caso se utiliza el siguiente modelo:

**Figura 3.2 Pantalla modelo de comentarios de usuario**



Como podemos observar queremos crear una lista de comentarios de usuarios y queremos que dichos elementos sean generados utilizando React. El primer paso es descomponer la interfaz en una jerarquía de componentes, siguiendo ese proceso podemos identificar los siguientes componentes:

**Figura 3.3 Jerarquía de componentes**



## Crear versión estática del componente

Ahora vamos a generar el código necesario para desplegar la información de forma estática dentro de los componentes, vamos a eliminar los archivos que fueron generados por create-react-app dentro de la carpeta `users/src` y vamos a crear un nuevo archivo `index.js` la ruta de nuestro archivo será `users/src/index.js` dentro del archivo vamos a escribir el siguiente código:

**Listado 3.3 Archivo index de la aplicación users**

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

const App = () => {
  return (
```

```

    <div>Hi, users!</div>
  );
}

ReactDOM.render(<App />, document.querySelector("#root"));

```

Ahora vamos a visualizar en nuestra página el mensaje, para dar un poco de estilo a nuestra página vamos a acceder al archivo `users/public/index.html` que es el archivo que se accede al iniciar nuestra aplicación y vamos a agregar el siguiente código:

### Listado 3.4 Agregar estilos de semantic-ui

```

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="shortcut icon" href="%PUBLIC_URL%/favicon.ico" />
    <meta
      name="viewport"
      content="width=device-width, initial-scale=1,
shrink-to-fit=no"
    />
    <meta name="theme-color" content="#000000" />
    <link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/semantic-ui/2.4.1/
semantic.min.css" />
    //...

```

Con esta línea estamos agregando estilos de `semantic-ui`, el cual es un framework que nos ayuda a generar diseños bien estilizados y responsivos. Para continuar con nuestro flujo vamos

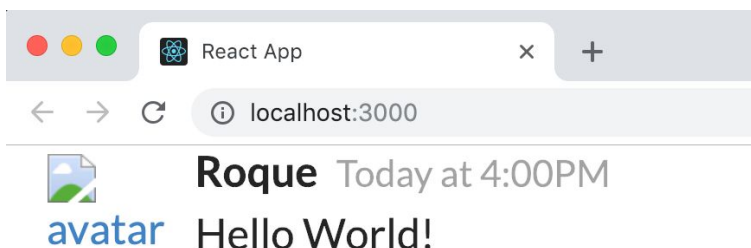
a modificar nuestro archivo `users/src/index.js` para generar JSX que represente algo similar a nuestra interfaz de usuario (UI) deseada. Vamos a escribir el siguiente código:

### Listado 3.5 Archivo index con un comentario

```
//..  
const App = () => {  
  return (  
    <div>Hi, users!</div>  
    <div className="ui container comments">  
      <div className="comment">  
        <a href="/" className="avatar">  
          <img alt="avatar" />  
        </a>  
        <div className="content">  
          <a href="/" className="author">  
            Roque  
          </a>  
          <div className="metadata">  
            <span className="date">Today at 4:00PM</span>  
          </div>  
          <div className="text">Hello World!</div>  
        </div>  
      </div>  
    </div>  
  );  
}  
  
ReactDOM.render(<App />, document.querySelector("#root"));
```

Si abrimos nuestra página podemos visualizar una imagen como la siguiente:

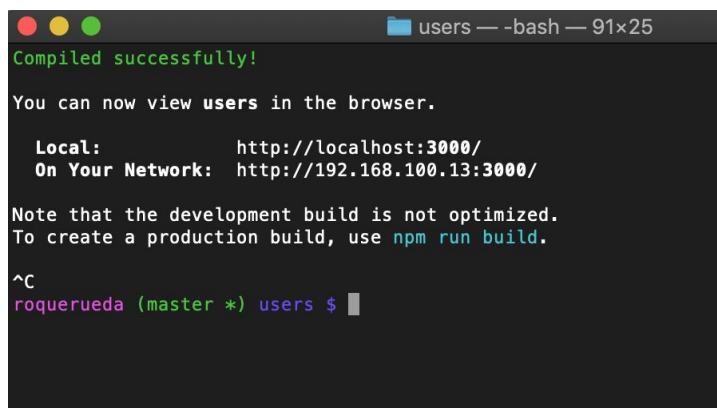
**Figura 3.4 Versión estática de comentarios**



Para tener una imagen de usuario vamos a hacer uso de otra biblioteca de JavaScript, el uso de una biblioteca nueva agrega una nueva dependencia dentro de nuestro archivo `package.json` sin embargo muy pocas veces tendremos que realizar la edición del archivo de forma manual, para agregar una nueva dependencia utilizaremos `npm`.

Vamos a detener nuestra aplicación para lo cual presiona las teclas `Ctrl + C` en la terminal que estaba ejecutando nuestra aplicación

**Figura 3.5 Detener aplicación users**



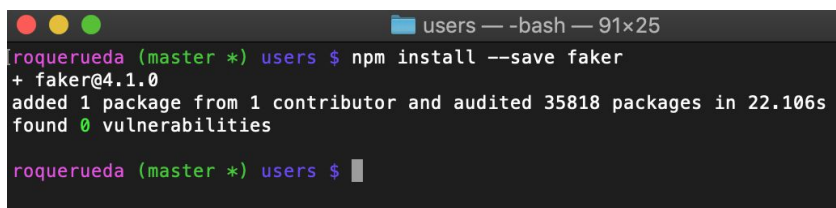
Y vamos a escribir el siguiente comando:

### Listado 3.6 Instalar faker

```
npm install --save faker
```

Se va a iniciar el proceso de instalación y al terminar tendremos una salida como la siguiente:

**Figura 3.6 Instalar faker**

A terminal window with a dark background and light text. The title bar shows 'users — -bash — 91x25'. The prompt is 'roquerueda (master \*) users \$'. The command 'npm install --save faker' has been entered. The output shows '+ faker@4.1.0', 'added 1 package from 1 contributor and audited 35818 packages in 22.106s', and 'found 0 vulnerabilities'. The prompt is now 'roquerueda (master \*) users \$' with a cursor.

```
roquerueda (master *) users $ npm install --save faker
+ faker@4.1.0
added 1 package from 1 contributor and audited 35818 packages in 22.106s
found 0 vulnerabilities
roquerueda (master *) users $
```

Vamos a volver a iniciar nuestra aplicación escribiendo el comando:

**Listado 3.7 Iniciar aplicacion usuarios**

```
npm start
```

Una vez que se inicie nuestra aplicación vamos a editar el archivo `users/src/index.js` para hacer uso de `faker`, la dependencia `faker` ayuda a generar datos falsos realistas en nuestra aplicación lo que nos ayuda a tener una UI con datos durante el desarrollo. Vamos a escribir el siguiente código:

**Listado 3.8 Utilizando faker para imagen avatar**

```
//...
import ReactDOM from 'react-dom';
import faker from 'faker';

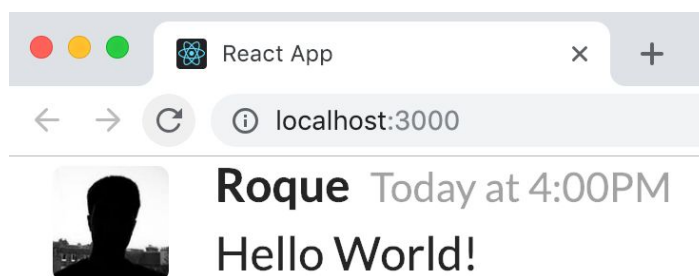
const App = () => {
  return (
    <div className="ui container comments">
      <div className="comment">
        <a href="/" className="avatar">
          <img alt="avatar" src={faker.image.avatar()} />
        </a>
      </div>
    </div>
  );
}
```



```
//...
```

Para poder hacer uso de la dependencia `faker` debemos importarla por lo que se agregará una sentencia `import` y vamos a agregar el atributo `src={faker.image.avatar()}` al visualizar nuestra página ahora contamos con una imagen que representa el avatar del usuario en el comentario.

**Figura 3.7** Obtener imagen de `faker`



## Evitar la repeticion

Aunque nuestra pantalla, ya se asemeja mucho a lo que queremos obtener sólo tenemos un comentario para poder generar más comentarios podríamos copiar y pegar el JSX dentro de nuestro archivo `users/src/index.js` sin embargo esto generaría que la navegación dentro de dicho archivo sea compleja además de que repetiríamos código lo cual no es recomendable dentro de nuestros componentes, para generar más de un renglón en la lista de comentarios vamos a crear un nuevo componente, primero vamos a crear una carpeta llamada `components` y dentro de dicha carpeta creamos un nuevo archivo llamado `CommentRow.js` quedando la ruta del archivo del siguiente modo: `users/src/components/CommentRow.js` y vamos a escribir el siguiente código el cual extraemos del archivo `users/src/index.js`:

**Listado 3.9** Crear `CommentRow` para evitar duplicar código

```
import React from 'react';
import faker from 'faker';

const CommentRow = (props) => {
```

```
return (  
  <div className="comment">  
    <a href="/" className="avatar">  
      <img alt="avatar" src={faker.image.avatar()} />  
    </a>  
    <div className="content">  
      <a href="/" className="author">  
        Roque  
      </a>  
      <div className="metadata">  
        <span className="date">Today at 4:00PM</span>  
      </div>  
      <div className="text">Hello World!</div>  
    </div>  
  </div>  
</div>  
);  
}  
  
export default CommentRow;
```

Ahora para hacer uso del componente `CommentRow` vamos a editar nuestro archivo `users/src/index.js` para importar y tener más de un solo comentario en nuestra lista de comentarios.

### Listado 3.10 Utilizar `CommentRow` en archivo `index`

```
import React from 'react';  
import ReactDOM from 'react-dom';  
import faker from 'faker';  
import CommentRow from '../components/CommentRow';
```

```

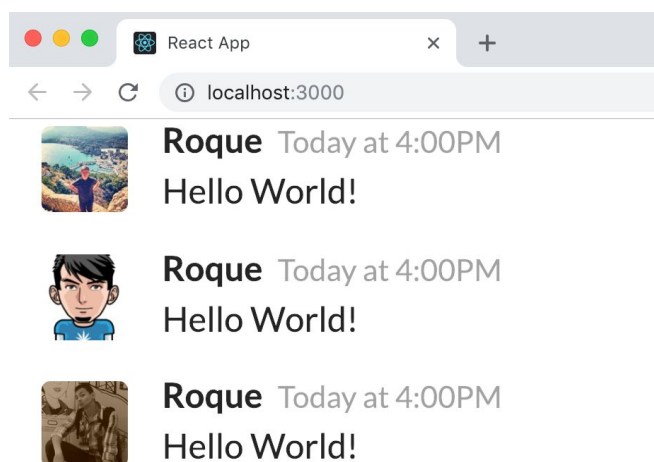
const App = () => {
  return (
    <div className="ui container comments">
      <div className="comment">
        <a href="/" className="avatar">
          <img alt="avatar" src={faker.image.avatar()} />
        </a>
        <div className="content">
          <a href="/" className="author">
            Roque
          </a>
          <div className="metadata">
            <span className="date">Today at 4:00PM</span>
          </div>
          <div className="text">Hello World!</div>
        </div>
      </div>
      <CommentRow />
      <CommentRow />
      <CommentRow />
    </div>
  );
}

ReactDOM.render(<App />, document.querySelector("#root"));

```

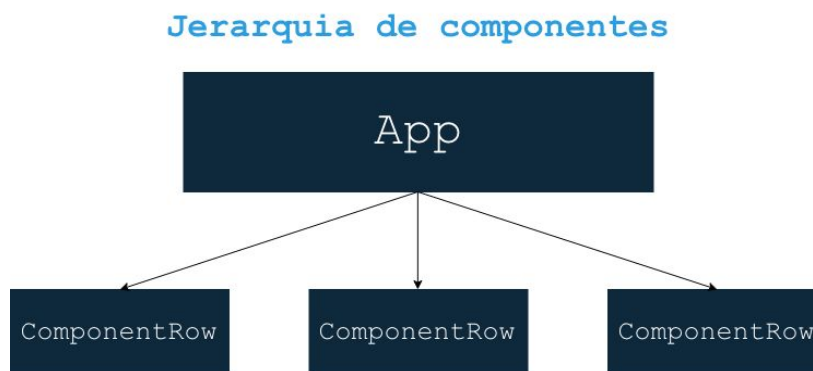
Podemos ver una pantalla ahora mucho más similar a lo que deseamos obtener sin embargo los datos de la pantalla se repiten:

**Figura 3.8 Lista reutilizando CommentRow**



Podemos ver entonces que hemos creado lo que es una jerarquía de componentes: La cual tiene 2 niveles únicamente pero que representa nuestra interfaz:

**Figura 3.9 Jerarquía de componentes de lista de comentarios.**



Ahora vamos a determinar los datos de la aplicación utilizando `props` y `state` para que nuestro componente muestre los datos correctamente. Este es uno de los pasos que requiere más atención debido a que vamos a determinar cuáles componentes requieren mantener estado y cuales solo deben ser configurados mediante propiedades.

## Utilizar propiedades

Las propiedades de un componente son un sistema por el cual podemos enviar datos desde un componente padre hacía componentes hijos y su función principal es la configuración del componente hijo.

Para nuestra aplicación podemos identificar que tenemos las siguientes piezas de información:

- **Autor**
- **Fecha**
- **Comentario**
- **Imagen**

Vamos a iniciar enviando las propiedades a cada renglón de comentario iniciando por el autor del comentario enviando un atributo JSX para lo cual vamos a editar nuestro archivo `users/src/index.js` esta vez enviando los valores del autor a cada renglón, vamos a escribir el siguiente código:

### Listado 3.10 Utilizar `CommentRow` en archivo `index`

```
import React from 'react';
import ReactDOM from 'react-dom';
import CommentRow from '../components/CommentRow';

const App = () => {
  return (
    <div className="ui container comments">
      <CommentRow author="Sarai" />
      <CommentRow author="Luis" />
      <CommentRow author="Roque" />
    </div>
  );
}
```

```
ReactDOM.render(<App />, document.querySelector("#root"));
```

Ahora debemos utilizar dicho valor en nuestro componente `users/src/components/CommentRow.js` vamos a escribir el siguiente código:

### Listado 3.11 Usar propiedades en CommentRow

```
//...
const CommentRow = (props) => {
  return (
    <div className="comment">
      <a href="/" className="avatar">
        <img alt="avatar" src={faker.image.avatar()} />
      </a>
      <div className="content">
        <a href="/" className="author">
          Request {props.author}
        </a>
        <div className="metadata">
          <span className="date">Today at 4:00PM</span>
        </div>
        <div className="text">Hello World!</div>
      </div>
    </div>
  );
}
//...
```

Y veremos que los valores de nuestros renglones de comentario muestran los datos que fueron enviados por atributos desde el componente padre. Vamos a realizar el mismo proceso con la

fecha, el comentario y la imagen avatar del usuario, para esto vamos a editar nuevamente el archivo: `users/src/components/CommentRow.js` y escribir el siguiente código:

### Listado 3.12 Remover valores `hardcode` de `CommentRow`

```
import React from 'react';
import faker from 'faker';
//...
return (
  <div className="comment">
    <a href="/" className="avatar">
      <img alt="avatar" src={faker.image.avatar()props.avatar} />
    </a>
    <div className="content">
      <a href="/" className="author">
        {props.author}
      </a>
      <div className="metadata">
        <span className="date">Today at 4:00PM{props.time}</span>
      </div>
      <div className="text">Hello World!{props.comment}</div>
    </div>
  </div>
);
//...
```

Y vamos a editar también nuestro archivo `users/src/index.js` escribiendo el siguiente código:

### Listado 3.13 Enviar atributos a componente `CommentRow`

```
import React from 'react';
```

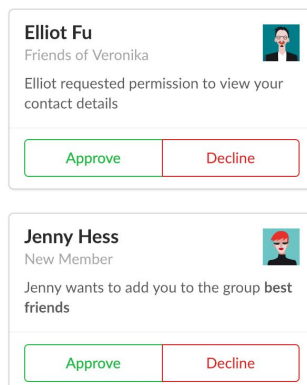
```
import ReactDOM from 'react-dom';
import CommentRow from '../components/CommentRow';
import faker from 'faker';
//...
<div className="ui container comments">
  <CommentRow author="Sarai" />
  <CommentRow author="Luis" />
  <CommentRow author="Roque" />
  <CommentRow
    author="Sarai"
    time="Today at 5:00PM"
    comment="Hi, nice post!"
    avatar={faker.image.avatar()} />
  <CommentRow
    author="Luis"
    time="Yesterday at 4:00PM"
    comment="Hello World"
    avatar={faker.image.avatar()} />
  <CommentRow
    author="Roque"
    time="Yesterday at 2:00PM"
    comment="Cool post!"
    avatar={faker.image.avatar()} />
</div>
//...
```

Ahora al ver nuestra página veremos que todos los valores que nuestra lista de comentarios de usuario provienen desde el componente padre. Ahora que tenemos nuestra vista vamos a agregar un nuevo componente el cual sea una tarjeta con botones para aceptar o rechazar un



comentario. Para esto vamos a regresar a la pagina <http://semantic-ui.com> y buscar en el menu superior izquierdo la palabra Card vamos a encontrar una vista como la siguiente:

**Figura 3.10 Ejemplo de Card utilizando semantic ui**



Si presionamos en el botón de example vamos a ver los nombres de clases con los que podemos generar los elementos similares como a los que tenemos en pantalla, como podemos observar cada elemento Card contiene un elemento content el cual vamos a utilizar para colocar nuestro contenido. Para iniciar vamos a crear un nuevo componente el cual llamaremos ApprovalCard.js dentro de nuestra carpeta components quedando la ruta del siguiente modo: users/src/components/ApprovalCard.js y vamos a escribir el siguiente código:

**Listado 3.14 Crear componente ApprovalCard.js**

```
import React from 'react'

const ApprovalCard = props => {
  return (
    <div className="ui card">
      <div className="content">{props.children}</div>
      <div className="extra content">
        <div className="ui two buttons">
          <div className="ui basic green button">Approve</div>
          <div className="ui basic red button">Reject</div>
        </div>
      </div>
    </div>
  )
}
```

```

    </div>
  </div>
);
}

export default ApprovalCard;

```

Con esto ahora tenemos un componente que puede recibir un contenido dinámicamente y presentar un elemento borde y botones alrededor de dicho elemento. Para hacer uso de dicho componente vamos a modificar nuestro archivo `users/src/index.js` y vamos a escribir el siguiente código:

### Listado 3.15 Crear componente ApprovalCard.js

```

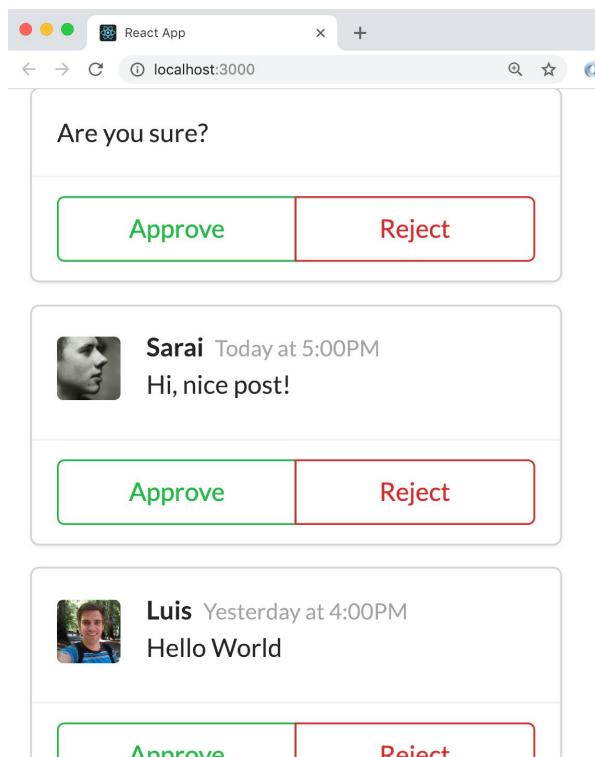
//...
const App = () => {
  return (
    <div className="ui container comments">
      <ApprovalCard>
        Are you sure?
      </ApprovalCard>
      <ApprovalCard>
        <CommentRow
          author="Sarai"
          time="Today at 5:00PM"
          comment="Hi, nice post!"
          avatar={faker.image.avatar()} />
        </ApprovalCard>
      <ApprovalCard>
        <CommentRow

```

```
        author="Luis"  
        time="Yesterday at 4:00PM"  
        comment="Hello World"  
        avatar={faker.image.avatar()} />  
      </ApprovalCard>  
      <ApprovalCard>  
        <CommentRow  
          author="Rogue"  
          time="Yesterday at 2:00PM"  
          comment="Cool post!"  
          avatar={faker.image.avatar()} />  
        </ApprovalCard>  
    </div>  
  );  
}
```

Y si visualizamos nuestra página vamos a poder ver algo como lo siguiente:

**Figura 3.11** Componente ApprovalCard



Podemos ver que nuestro nuevo componente agregar elementos y utiliza la propiedad `children` para mostrar el contenido, es por eso que podemos incrustar inclusive el componente `CommentRow`, el cual es un componente definido por nosotros y que nos ayuda a elaborar nuestra interfaz de usuario. El primer elemento contiene un texto simple sin embargo podemos agregar cualquier elemento no solo texto o componentes definidos por nosotros. Gracias a la flexibilidad que nos brinda la composición en React es que podemos conformar jerarquías de componentes como en este ejemplo.

## Utilizar estado del componente

El estado son los datos que un componente administra de forma privada y que afectan el comportamiento del componente así como la manera como se presenta en pantalla. Para poder aprobar o rechazar un contenido es necesario contar con un estado de aprobado, esto debido a que cada elemento puede ser bien aprobado (es decir que se muestre en pantalla) o rechazado (que no se muestre en pantalla). Si este valor no ha sido definido por el usuario entonces mostraremos la tarjeta completa para que el usuario pueda aprobar o rechazar el contenido, una vez que aprueba o rechaza el contenido, vamos solo a mostrar el contenido en caso de ser aprobado o nada en caso de ser rechazado.

Para poder hacer uso del estado debemos cambiar nuestro componente de una función a una clase. Vamos a cambiar `users/src/components/ApprovalCard.js` colocando el siguiente código y vamos a discutir la razón de dichos cambios más adelante.

### Listado 3.16 Crear componente ApprovalCard.js

```
import React, { Component } from 'react';

const ApprovalCard = props => {
  return (
    <div className="ui card">
      <div className="content">{props.children}</div>
      <div className="extra content">
        <div className="ui two buttons">
          <div className="ui basic green button">Approve</div>
          <div className="ui basic red button">Reject</div>
        </div>
      </div>
    </div>
  )
}

class ApprovalCard extends Component {
  constructor(props) {
    super(props);
    this.state = {
      isApprove: undefined
    }
  }

  userHasNotApproveOrReject = () => {
    return this.state.isApprove === undefined
  }
}
```

```

approveContent = () => {
  this.setState({
    isApprove: true
  });
}

rejectContent = () => {
  this.setState({
    isApprove: false
  });
}

renderCard = () => {
  return (
    <div className="ui card">
      <div className="content">{this.props.children}</div>
      <div className="extra content">
        <div className="ui two buttons">
          <button
            onClick={() => { this.approveContent() }}
            className="ui basic green button">Approve</button>
          <button
            onClick={() => { this.rejectContent() }}
            className="ui basic red button">Reject</button>
          </div>
        </div>
      </div>
    </div>
  );
}

renderOnlyContent = () => {
  if (this.state.isApprove) {
    return this.props.children
  }
}

```

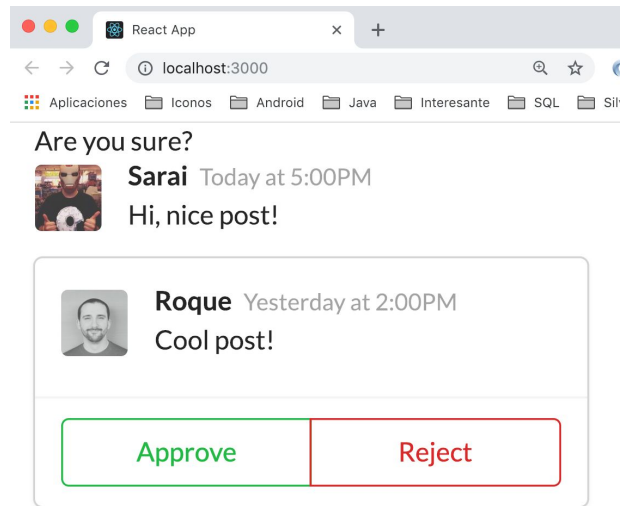
```
    } else {  
      return null  
    }  
  }  
}  
  
render() {  
  return (  
    this.userHasNotApproveOrReject() ?  
    this.renderCard() : this.renderOnlyContent()  
  );  
}  
}  
  
export default ApprovalCard;
```

Ahora con estos cambios nuestra aplicación ya puede responder a los eventos del usuario para aprobar o rechazar el contenido mostrado en la tarjeta. Este cambio aplicará funcionalidad para todos los componentes que se encuentren dentro de la tarjeta.

Podemos ver que el primer cambio a nuestro componente `ApprovalCard` es en su definición la cual ahora se realiza mediante una clase que extiende de `Component`. Dentro del constructor se define un estado inicial. Estamos definiendo 3 propiedades como `arrow function` las cuales se encargan de trabajar con el estado del componente bien sea para determinar si el estado ya fue asignado por el usuario, aprobar o rechazar el contenido. Vemos que el estado se afecta utilizando la función `setState`, esta función es la forma en que React espera que el estado del componente sea actualizado y posteriormente a su ejecución el componente es renderizado nuevamente.

Es por eso que cuando el usuario aprueba o rechaza el contenido estamos actualizando el estado con sus valores correspondientes, se dispara el renderizado del componente nuevamente, el método `render` cuenta con las condiciones para mostrar toda la tarjeta o solo el contenido validando si el estado ya fue asignado por el usuario.

**Figura 3.12 Reaccionando a eventos del usuario**





# LAB 05: Crear una aplicación React

## Objetivos

- Crear una aplicación React que tenga elementos de entrada
- Actualizar el estado del componente de forma adecuada
- Dar estilo a nuestro componente

## Crear una aplicación React que tenga elementos de entrada

Esta será la aplicación más grande de nuestro curso por lo que es importante tratar de terminar cada laboratorio debido a que los siguientes laboratorios dependen de que se generen los componentes dentro de este capítulo.

Para iniciar con este laboratorio vamos a crear una nueva carpeta llamada 05-application y navegar a dicha carpeta, una vez dentro de la carpeta vamos a ejecutar desde la terminal el siguiente comando:

### Listado 5.1 Crear aplicación users

```
npx create-react-app flickr
```

Una vez que se genere nuestra aplicación vamos a abrir la carpeta que lo contiene, iniciar la aplicación y abrir la carpeta con VSCode. Eliminaremos los archivos autogenerados de la carpeta `src` y vamos a crear un nuevo archivo `index.js`. La ruta del archivo sera `flickr/src/index.js` en donde vamos a escribir el siguiente código:

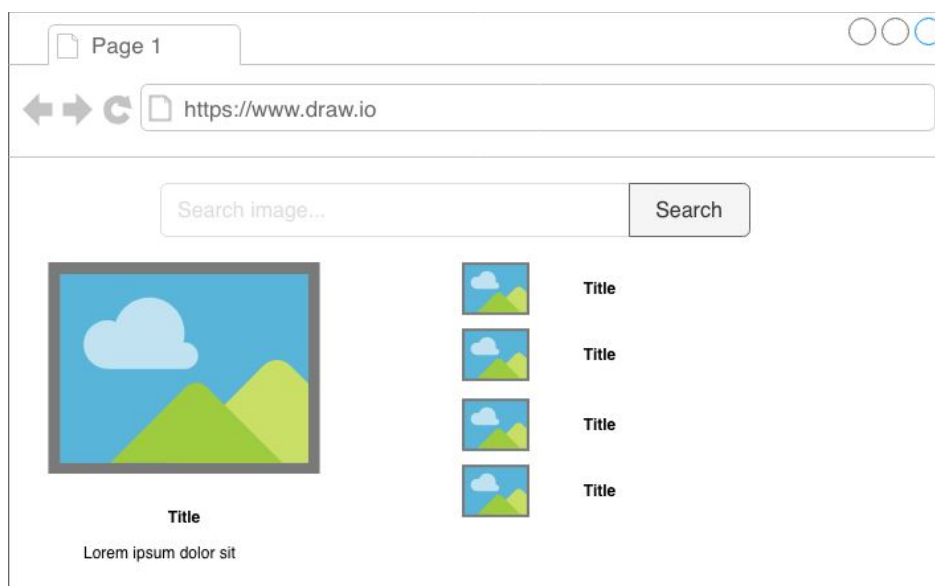
### Listado 5.2 Flickr inicio

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

```
const App = () => {  
  return (  
    <div>Hello World</div>  
  );  
}  
  
ReactDOM.render(<App />, document.querySelector('#root'));
```

Ahora podremos ver un simple mensaje en nuestra pantalla, siguiendo el flujo de pensando en React debemos iniciar con una pantalla modelo la cual nos determinará los componentes a construir.

**Figura 5.1 Mockup de la aplicación Flickr**



Como podemos ver nuestra aplicación será un buscador de imágenes el cual contará con una barra de búsqueda una lista de resultados y un área de visualización de los detalles de la imagen.

Vamos a iniciar con la creación del componente de barra de búsqueda, para dar estilos a nuestra aplicación volveremos a utilizar semantics por lo que vamos a abrir nuestro archivo `flickr/public/index.html` y vamos a agregar el siguiente código:

### Listado 5.3 Agregar estilos de semantic-ui

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="shortcut icon" href="%PUBLIC_URL%/favicon.ico" />
    <meta
      name="viewport"
      content="width=device-width, initial-scale=1,
shrink-to-fit=no"
    />
    <meta name="theme-color" content="#000000" />
    <link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/semantic-ui/2.4.1/
semantic.min.css" />
//...
```

Una vez hecho esto vamos a acceder a la página: <https://semantic-ui.com/elements/input.html>, en esta pagina vamos a encontrar en la sección de action un elemento que es muy parecido a lo que buscamos generar. Tomaremos como ejemplo dicho elemento y los nombres de clase CSS.

Vamos a crear una carpeta llamada `components` y dentro de dicha carpeta crear un archivo `SearchBar.js`, la ruta al archivo será: `flickr/src/components/SearchBar.js` al cual le vamos a agregar el siguiente código:

### Listado 5.4 Agregar estilos de semantic-ui

```
import React from 'react'

const SearchBar = props => {
  return (
```

```

    <center>
      <div className="ui action input" style={{ marginTop: 16
    }}>
        <input type="text" placeholder="Search images..." />
        <button className="ui button">Search</button>
      </div>
    </center>
  );
}

export default SearchBar;

```

Y para utilizarlos vamos a importar el componente en nuestro archivo flickr/src/index.js quedando del siguiente modo:

#### Listado 5.5 Utilizando el componente SearchBar

```

import React from 'react';
import ReactDOM from 'react-dom';
import SearchBar from './components/SearchBar';

const App = () => {
  return (
    <div>Hello World</div>
    <SearchBar />
  );
}

ReactDOM.render(<App />, document.querySelector('#root'));

```

Con lo que obtendremos una barra de búsqueda, sin embargo este componente se está generando utilizando una función, debido a que deseamos que el componente reaccione a los eventos del usuario así como que almacene el texto de búsqueda resulta mejor modificar el componente para poder utilizar estado. Vamos a modificar nuestro componente `flickr/src/components/SearchBar.js` para convertirlo en una clase ES6:

#### Listado 5.6 Convertir SearchBar en una clase

```
import React, { Component } from 'react'

const SearchBar = props => {
  return (
    <center>
      <div className="ui action input" style={{ marginTop: 16 }}>
        <input type="text" placeholder="Search images..." />
        <button className="ui button">Search</button>
      </div>
    </center>
  )
}

class SearchBar extends Component {
  constructor(props) {
    super(props);
    this.state = { term: '' }
  }

  render() {
    return (
      <center>
        <div className="ui action input" style={{ marginTop: 16 }}>
```

```

        <input type="text" placeholder="Search images..." />
        <button className="ui button">Search</button>
      </div>
    </center>
  );
}
}

export default SearchBar;

```

Una vez convertido el componente en una clase podemos ver que nuestra aplicación sigue mostrando la barra de búsqueda, podemos notar que en el constructor estamos declarando un valor denominado term, el cual será el encargado de almacenar el término de búsqueda tecleado por el usuario. Vamos ahora a utilizar el estado para asignar el valor del componente input de nuestra barra de búsqueda.

#### Listado 5.7 Convertir SearchBar en una clase

```

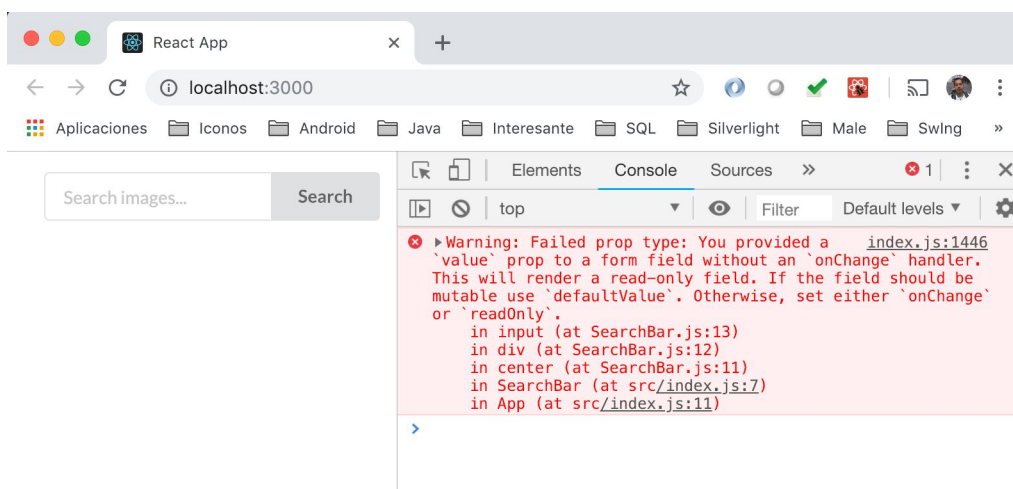
//...
render() {
  return (
    <center>
      <div className="ui action input" style={{ marginTop: 16 }}>
        <input type="text" placeholder="Search images..." />
        <input
          type="text"
          placeholder="Search images..."
          value={this.state.term} />
        <button className="ui button">Search</button>
      </div>
    </center>
  );
}

```

```
}  
//...
```

Al visualizar nuestra página y abrir la consola del navegador podemos ver un mensaje de error:

**Figura 5.2 Mensaje de error en un campo de formulario**



Como podemos observar React espera que brindemos una función que se encargue de responder a los cambios en el componente debido a que solo se envió el atributo value el componente es de solo lectura, esto lo podemos validar al intentar escribir y veremos que nuestro componente no reacciona. Para resolver este error vamos a volver a editar nuestro archivo flickr/src/components/SearchBar.js y vamos a agregar el siguiente código:

**Listado 5.8 Responder al evento onChange de SearchBar.js**

```
//...  
render() {  
  return (  
    <center>  
      <div className="ui action input" style={{ marginTop: 16 }}>  
        <input type="text" placeholder="Search images..." />  
        <input  
          type="text"  
          placeholder="Search images..."
```

```

        value={this.state.term}
        onChange={event=>
            this.onInputChange(event.target.value) } />
        <button className="ui button">Search</button>
    </div>
</center>
);
}
//...

```

Al guardar estos cambios y cargar nuestra página vamos a ver que nuestro componente funciona adecuadamente y el error se ha ido. El error se presenta debido a que estábamos asignado el valor del `input` y React detecta que no hay ningún evento asociado al evento `onChange` por lo cual el valor del `input` nunca cambiara. Al contar con un el evento `onChange` el estado se ve alterado lo que provoca que el componente sea renderizado de forma correcta.

Vamos ahora a continuar con la creación de la lista de imágenes, vamos a repetir el proceso que con la barra de búsqueda; primero vamos a buscar en la página <https://semantic-ui.com/elements/list.html> podemos encontrar un componente similar a lo que buscamos crear vamos a crear un elemento llamado `ListItem.js` en nuestra carpeta `components` quedando la ruta del archivo de la siguiente manera: `flickr/src/components/ListItem.js` y vamos a agregar el siguiente código:

### Listado 5.9 Componente ListItem.js

```

import React from 'react';

const ListItem = props => {
    return (
        <div className="item">
            <img className="ui avatar image" alt="alt" />
            <div className="content">
                <a href="/" className="header">Roque</a>
            </div>
        </div>
    );
}

```



```
    <div className="description">An item of the list</div>
  </div>
</div>
);
}

export default ListItem;
```

Ahora vamos a agregar otro componente que representa la lista de los elementos y vamos a mostrar dicha lista en nuestra página para tener una versión estática de la página. El componente lo crearemos en la carpeta `components` y lo nombraremos `ImageList.js` la ruta del archivo será: `flickr/src/components/ImageList.js` y vamos a escribir el siguiente código:

#### Listado 5.10 Componente `ImageList.js`

```
import React from 'react';
import ListItem from './ListItem';

const ImageList = props => {
  return (
    <div className="ui relaxed divided selection list">
      <ListItem />
      <ListItem />
      <ListItem />
    </div>
  );
}

export default ImageList;
```

Y finalmente vamos a editar nuestro archivo: `flickr/src/index.js` para poder mostrar mas de un componente debemos añadirlos en un solo componente padre:

### Listado 5.11 Mostrar lista de imágenes y barra de búsqueda

```
import React from 'react';
import ReactDOM from 'react-dom';
import SearchBar from '../components/SearchBar';
import ImageList from '../components/ImageList';

const App = () => {
  return (
    <SearchBar />
    <div className="ui grid">
      <div className="row">
        <div className="wide column">
          <SearchBar />
        </div>
      </div>
      <div className="row">
        <div className="nine wide column"></div>
        <div className="seven wide column">
          <ImageList />
        </div>
      </div>
    </div>
  );
}

ReactDOM.render(<App />, document.querySelector('#root'));
```

Ahora vamos a poder ver nuestra página algo similar a la aplicación que deseamos crear, sin embargo aún nos falta generar el componente que presente los detalles de la imagen, vamos a crear un nuevo componente dentro de la carpeta `components` el cual llamaremos

ImageDetail.js y la ruta del archivo será: flickr/src/components/ImageDetail.js y vamos a escribir el siguiente código:

#### Listado 5.12 Componente image detail

```
import React from 'react';

const ImageDetail = props => {
  return (
    <div className="ui card">
      <div className="image">
        <img alt="alt" />
      </div>
      <div className="content">
        <a href="/" className="header">Roque</a>
        <div className="meta">
          <span className="date">05/02/2019</span>
        </div>
        <div className="description">
          Image description
        </div>
      </div>
    </div>
  );
}

export default ImageDetail;
```

Y vamos a editar nuestro componente flickr/src/index.js para hacer uso del componente, vamos a agregar el siguiente código:

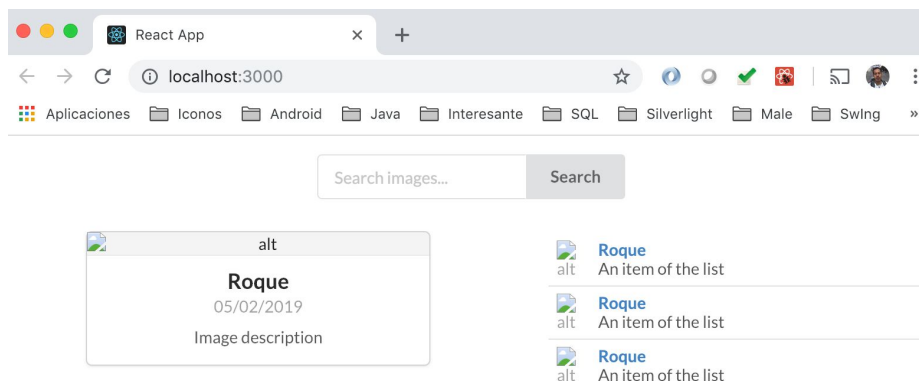
#### Listado 5.13 Utilizando el componente ImageDetail

```
//...
```

```
const App = () => {  
  return (  
    <div className="ui grid">  
      <div className="row">  
        <div className="wide column">  
          <SearchBar />  
        </div>  
      </div>  
      <div className="row">  
        <div className="nine wide column">  
          <center>  
            <ImageDetail />  
          </center>  
        </div>  
        <div className="seven wide column">  
          <ImageList />  
        </div>  
      </div>  
    </div>  
  );  
}
```

Ahora podemos ver una página que tiene mayor parecido a lo que estábamos buscando, aún sin imágenes.

**Figura 5.3 Versión estática de la aplicación flickr**



# LAB 06: Estado del componente

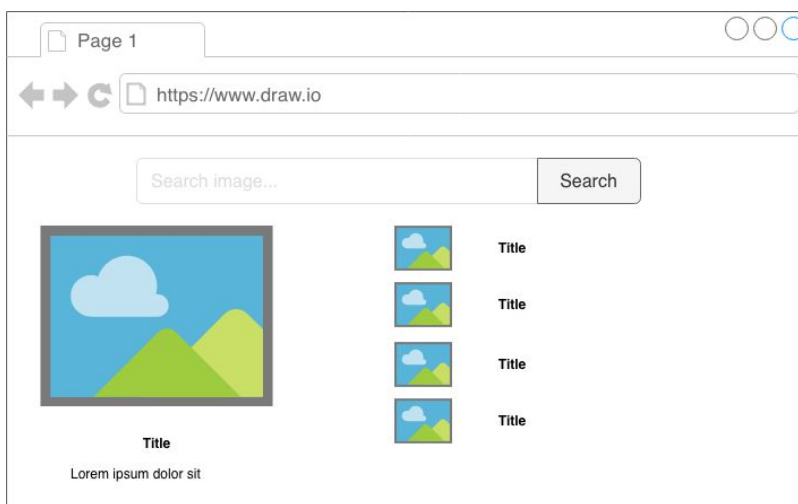
## Objetivos

- Determinar los lugares donde se requiere estado en la aplicación
- Renderizar componentes basados en su estado

## Determinar los lugares donde se requiere estado en la aplicación

Tomando como punto de partida el laboratorio número 5 vamos a determinar los lugares de nuestra aplicación donde se requiere estado o únicamente propiedades.

Figura 6.1 Mockup de la aplicación Flickr



Podemos ver que parte del trabajo ya fue realizado para el componente de búsqueda ya que se cuenta con el término de búsqueda y este término va a ser actualizado cuando el usuario teclee valores dentro de la caja de texto. Podemos decir que nuestra aplicación requiere por lo menos los siguientes datos

## Datos identificados de la aplicación Flickr

- **Termino de la busqueda**
- **Lista de imágenes resultado de la búsqueda**
- **Imagen seleccionada**

El término de búsqueda debe vivir en el componente de barra de búsqueda, sin embargo debe poder comunicarse con otros componentes para indicar que el usuario desea buscar nuevas imagenes.

La lista de imágenes aunque podría vivir en la lista de imágenes, también puede ser únicamente un valor de entrada en la lista de imágenes por lo que se enviará la lista de imágenes como una propiedad del componente.

La imagen seleccionada también será una propiedad enviada al componente de detalles de imagen, esta propiedad tiene un caso especial donde el usuario no haya seleccionado una imagen lo cual debería mostrar un componente que indique al usuario que seleccione una imagen.

Ya que la lista de imágenes así como la imagen seleccionada son propiedades enviadas estos valores residirán en el componente padre de ambos para poder realizar el paso de información a los mismos.

Vamos a modificar nuestros componentes para que reciban propiedades, iniciaremos con el componente `flickr/src/components/ImageDetail.js` y vamos a escribir el siguiente código:

### Listado 6.1 Recibir propiedades en componente ImageDetail

```
import React from 'react';

const ImageDetail = props => {
  const { img } = props;
  return (
    <div className="ui card">
      <div className="image">
        <img alt="alt"/>
      </div>
    </div>
  );
}
```

```

    {img === undefined ?
      <div className="ui placeholder">
        <div className="square image"></div>
      </div> :
      <img alt="alt" src={img.url_s} />
    }
  </div>
<div className="content">
  <a href="/" className="header">Rogue</a>
  {img === undefined ?
    <div className="ui placeholder">
      <div className="header">
        <div className="very short line"></div>
        <div className="medium line"></div>
      </div>
    </div> :
    <a href="/" className="header">{img.id}</a>
  }
  <div className="meta">
    <span className="date">05/02/2019</span>
    {img === undefined ?
      <div className="ui placeholder">
        <div className="very short line"></div>
      </div> :
      <span className="date">{img.date}</span>
    }
  </div>
  <div className="description">
    {img === undefined ? '' : img.title}
  </div>

```



```
        </div>
      </div>
    );
  }

  export default ImageDetail;
```

Al ver nuestra página vamos a ver que el detalle de la imagen presenta un elemento de relleno esto es debido a las clases de `semantics ui` además de que se está validando la propiedad `img`, si dicha propiedad no está definida la aplicación mostrará el componente de relleno, esto es renderizado condicional.

Vamos a realizar el mismo proceso con el componente `ListItem`, vamos a abrir el archivo `flickr/src/components/ListItem.js` y vamos a agregar el siguiente código:

#### Listado 5.9 Componente `ListItem.js`

```
import React from 'react';

const ListItem = props => {
  const { img } = props;

  if (img === undefined) {
    return (
      <div className="ui placeholder">
        <div className="image header">
          <div className="line"></div>
          <div className="line"></div>
        </div>
      </div>
    );
  }
}
```

```

return (
  <div className="item">
    <img className="ui avatar image" alt="alt" />
    <img
      className="ui avatar image"
      alt={img.id}
      src={img.url_s} />
    <div className="content">
      <a href="/" className="header">Rogue{img.id}</a>
      <div className="description">An item of the
list{img.title}</div>
    </div>
  </div>
);
}

export default ListItem;

```

Si vemos nuestra página podemos ver que en ambos casos la página muestra elementos de relleno debido a que no se están enviando las propiedades a los componentes por el momento.

# LAB 07: Peticiones HTTP

## Objetivos

- Utilizar axios para obtener imágenes desde flickr
- Enviar propiedades a los componentes
- Reaccionar a eventos de usuario

## Utilizar axios para obtener imágenes desde flickr

Para iniciar vamos a tomar como punto de partida el laboratorio 6, vamos a agregar la dependencia `axios` a nuestro proyecto para esto vamos a detener nuestra aplicación en la terminal y vamos a escribir el siguiente comando en la terminal dentro de la carpeta de nuestro proyecto:

### Listado 7.1 Instalación de axios

```
npm install --save axios
```

Una vez terminada la instalación vamos a modificar nuestro archivo `flickr/src/index.js` para realizar una petición que obtenga imágenes desde flickr agregando el siguiente código:

### Listado 7.2 Obtener Imágenes con axios

```
//...  
import ImageDetail from '../components/ImageDetail';  
import axios from 'axios';  
  
const API_KEY_FLICKR = "b5b00d16f3e70b5aa0157d79edaab8b5";  
  
const executeSearch = searchText => {  
  axios.get(`https://api.flickr.com/services/rest/?  
    method=flickr.photos.search&
```

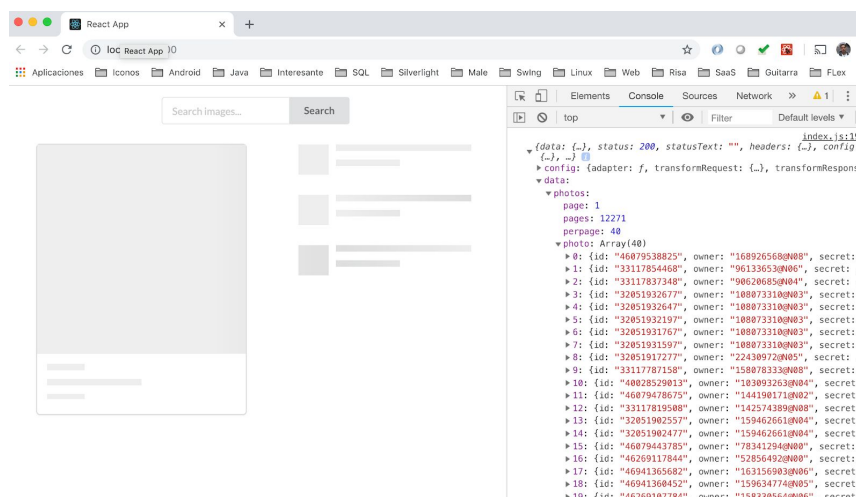
```
api_key=${API_KEY_FLICKR}&
format=json
text=${searchText}&
nojsoncallback=true&
per_page=40&
extras=url_s`).then(response => {
  console.log(response);
}).catch(error => {
  console.log(error);
})
}

const App = () => {
  executeSearch('dog');

  return (
    <div className="ui grid">
    //...
```

Al finalizar la edición de nuestro archivo podemos abrir la consola de desarrollador y veremos que nuestra aplicación está trayendo datos desde flickr.

**Figura 7.1** Obteniendo datos e imprimirlos en consola



Ahora para poder hacer uso de los datos vamos a cambiar nuestro componente App de una función a una clase esto con el objetivo de tener acceso al estado y poder mantener los datos obtenidos desde flickr, vamos a editar nuestro archivo flickr/src/index.js nuevamente.

### Listado 7.3 Cambiar componente App a una clase

```
import React, { Component } from 'react';
import ReactDOM from 'react-dom';
import SearchBar from './components/SearchBar';
import ImageList from './components/ImageList';
import ImageDetail from './components/ImageDetail';
import axios from 'axios';

const API_KEY_FLICKR = "b5b00d16f3e70b5aa0157d79edaab8b5";

const class App => extends Component {
  constructor(props) {
    super(props);
    this.state = {
      images: [],
      selectedImage: undefined
    }
  }
```

```

    this.executeSearch('penguin');
  }

  executeSearch = searchText => {
    axios.get(`https://api.flickr.com/services/rest/?
      method=flickr.photos.search&
      api_key=${API_KEY_FLICKR}&
      format=json&
      text=${searchText}&
      nojsoncallback=true&
      per_page=40&
      extras=url_s`) .then(response => {
      this.setState({
        images: response.data.photos.photo,
        selectedImage: response.data.photos.photo[0]
      })
    }) .catch(error => {
      console.log(error);
    })
  }

  render() {
    return (
      <div className="ui grid">
        <div className="row">
          <div className="wide column">
            <SearchBar />
          </div>
        </div>
        <div className="row">
          <div className="nine wide column">
            <center>
              <ImageDetail img={this.state.selectedImage} />
            </center>
          </div>
        </div>
      </div>
    )
  }
}

```

```
        </center>
      </div>
      <div className="seven wide column">
        <ImageList />
      </div>
    </div>
  </div>
);
}
}

ReactDOM.render(<App />, document.querySelector('#root'));
```

Como podemos ver en nuestra aplicación el componente `image detail` ya despliega información desde `flickr` en este caso la primer foto que se encuentra en la respuesta que se obtiene desde `flickr`, esto debido a que se está ejecutando la búsqueda desde el constructor, sin embargo, no es hasta que `axios` obtiene la respuesta que se ejecuta el método `setState` lo que hace que el componente se renderice de nuevo y presente los datos.

Ahora vamos a pasar la lista de fotos al componente lista de imágenes para generar los renglones de la lista de imágenes, volveremos a editar el archivo `flickr/src/index.js`

#### Listado 7.4 Enviar imágenes como propiedad

```
//...
<div className="seven wide column">
  <ImageList images={this.state.images} />
</div>
//...
```

Ahora debemos hacer uso de estas nuevas propiedades en nuestro componente de lista de imágenes por lo que vamos a editar el archivo: `flickr/src/components/ImageList.js` y agregar el siguiente código:

#### Listado 7.5 Utilizar lista de imágenes

```
import React from 'react';
```

```
import ListItem from './ListItem';

const ImageList = props => {
  const { images } = props;
  if (images === undefined) {
    return (
      <div className="ui relaxed divided selection list">
        <ListItem />
        <ListItem />
        <ListItem />
      </div>
    );
  }
  const rows = images.map(img => {
    return (
      <ListItem key={img.id} img={img} />
    )
  })
  return (
    <div className="ui relaxed divided selection list">
      {rows}
    </div>
  );
}

//...
```

Al guardar nuestra página podemos ver que ya se están mostrando la lista de imágenes con sus respectivos datos desde flickr, ahora debemos agregar interacción cuando el usuario hace click en un elemento de la lista, deseamos que el detalle se actualice para mostrar el elemento seleccionado. Vamos a editar nuestro elemento `flickr/src/components/ListItem.js` para que reaccione al evento click, y que ejecute una función para el manejo del mismo.



**Listado 7.6 Agregar click al elemento de ListItem**

```
//...
return (
  <div className="item" onClick={event => props.onClick(img)}>
    <img
      className="ui avatar image"
      alt={img.id}
      src={img.url_s} />
    <div className="content">
      <a href="/" className="header">{img.id}</a>
      <div className="description">{img.title}</div>
    </div>
  </div>
);
}
//...
```

La función se recibe como propiedad por lo que debe ser enviada por el componente padre en este caso `ImageList`, vamos ahora a modificar el archivo `flickr/src/components/ImageList.js` y agregar el siguiente código:

**Listado 7.7 Agregar click al elemento de ListItem**

```
//...
const rows = images.map(img => {
  return (
    <ListItem key={img.id} img={img} onClick={props.onItemClick} />
  );
})
//...
```

Y asu vez la función se recibe del componente padre por lo que vamos a editar el archivo `flickr/src/index.js` y escribir el siguiente código:

### Listado 7.8 Manejo de click en componente App

```
//...

onImageSelected = img => {
  this.setState({
    selectedImage: img
  });
}

render() {
  return (
    <div className="ui grid">
      <div className="row">
        <div className="wide column">
          <SearchBar />
        </div>
      </div>
      <div className="row">
        <div className="nine wide column">
          <center>
            <ImageDetail img={this.state.selectedImage} />
          </center>
        </div>
        <div className="seven wide column">
          <ImageList images={this.state.images} />
          <ImageList
            images={this.state.images}
            onItemClick={this.onImageSelected} />
        </div>
      </div>
    </div>
  );
}
```

```
    </div>  
    //...
```

Y si guardamos nuestra aplicación, al hacer click un elemento de la lista el detalle cambia para mostrar la imagen adecuada tanto como su detalle. Finalmente vamos a agregar una función más para invocar cuando se presione el botón de búsqueda y el usuario teclee un valor en la barra de búsqueda editaremos de nuevo nuestro archivo `flickr/src/index.js`

#### Listado 7.9 Manejo de click en componente App

```
//...  
  
onSearch = searchText => {  
    this.executeSearch(searchText);  
}  
  
render() {  
    return (  
        <div className="ui grid">  
            <div className="row">  
                <div className="wide column">  
                    <SearchBar onSearch={this.onSearch} />  
                </div>  
            </div>  
        </div>  
    //...
```

Y ahora vamos a hacer uso de esta propiedad dentro de nuestro componente de barra de búsqueda, para eso vamos a nuestro archivo `flickr/src/components/SearchBar.js` y agregaremos el siguiente código:

#### Listado 7.10 Utilizando propiedad para búsqueda

```
//...  
  
<center>  
    <div className="ui action input" style={{ marginTop: 16 }}>  
        <input
```

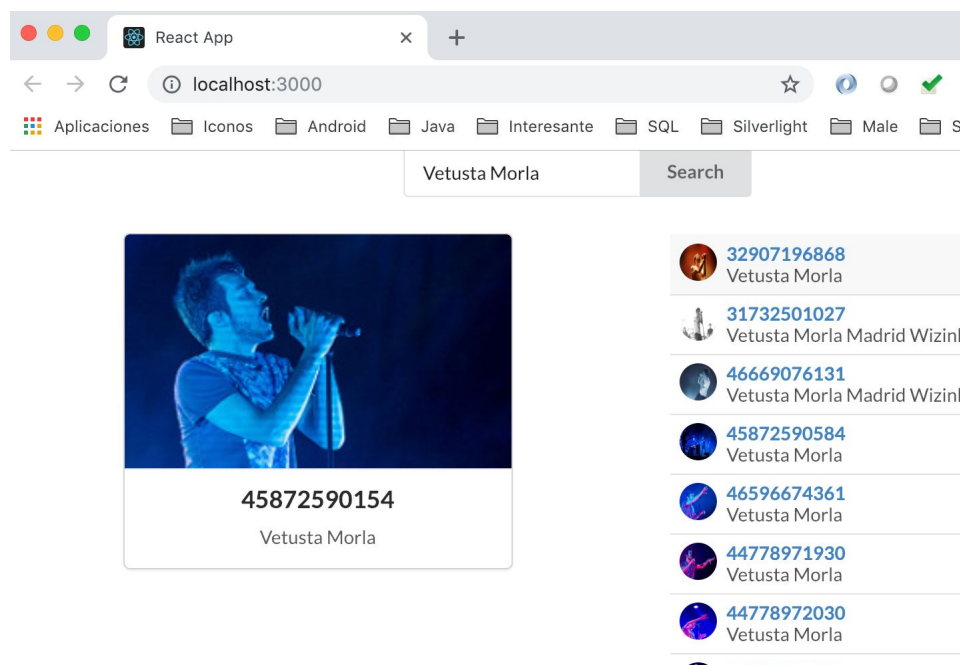
```

type="text"
placeholder="Search images..."
value={this.state.term}
onChange={event=>
  this.onChange(event.target.value)} />
<button
  className="ui button"
  onClick={event=>
    this.props.onSearch(this.state.term)}>Search</button>
</div>
</center>
//...

```

Ahora podemos escribir un texto y al presionar el boton de busqueda nuestras imágenes se van a cargar desde flickr y mostrar los resultados de la búsqueda.

Figura 7.2 Aplicación Flickr buscando imagenes



# LAB 08: Introducción a Redux

## Objetivos

- Crear ActionCreators
- Crear Reducers
- Conectar React y Redux
- Actualizar el estado global de la aplicación

## Crear ActionCreators

Tomando como punto de partida el laboratorio anterior vamos a agregar dos nuevas carpetas dentro de nuestra carpeta `src`, las cuales contendrán todo lo relacionado a Actions en `actions` y todo lo relacionado a Reducers en `reducers` vamos a iniciar con la creación de nuestros ActionCreators para manejar el elemento seleccionado por el usuario. Primero vamos a instalar `redux` y `react-redux` en nuestra aplicación vamos a ejecutar en la terminal el siguiente comando:

### Listado 8.1 Instalación de `redux` y `react-redux`

```
npm install --save redux react-redux
```

Una vez finalizado vamos a ejecutar nuestra aplicación nuevamente y vamos a verificar que no exista ningún error. Vamos a crear un archivo llamado `index.js` dentro de la carpeta `actions` quedando la ruta del archivo de la siguiente manera: `flickr/src/actions/index.js` al llamar a nuestro archivo de ese modo webpack nos permite importar el folder sin especificar un archivo en concreto. Ahora bien en el archivo vamos a escribir el siguiente código:

### Listado 8.2 Creando el ActionCreator para seleccionar Imagen

```
// Action creator
export const selectImage = (image) => {
  return {
```

```
    type: 'SELECTED_IMAGE',  
    payload: image  
  };  
}
```

Como podemos ver nuestro action creator está generando un objeto con dos propiedades `type` y `payload` las cuales serán utilizadas por nuestros reducers para afectar o no el estado.

Ahora vamos a crear un nuevo archivo `index.js` pero esta ocasión dentro de la carpeta `reducers` quedando la ruta al archivo del siguiente modo: `flickr/src/reducers/index.js` y vamos a escribir el siguiente código:

#### Listado 8.3 Creando el Reducer para seleccionar image

```
import { combineReducers } from 'redux';  
  
// Reducer  
const selectImageReducer =  
  (selectedImage = null, action) => {  
    if(action.type === 'SELECTED_IMAGE') {  
      return action.payload;  
    }  
  
    return selectedImage;  
  }  
  
export default combineReducers({  
  selectedImage: selectImageReducer  
})
```

Con esto tenemos la infraestructura requerida por redux y vamos a hacer uso de ella dentro de nuestros componente JSX.

Para esto vamos a editar nuestro archivo `flickr/src/index.js` al cual vamos a editar para hacer uso de redux en nuestra aplicación:

#### Listado 8.4 Utilizando provider y creando el store

```
import React, { Component } from 'react';
import ReactDOM from 'react-dom';
import axios from 'axios';
import { Provider } from 'react-redux';
import { createStore } from 'redux';

import SearchBar from './components/SearchBar';
import ImageList from './components/ImageList';
import ImageDetail from './components/ImageDetail';
import reducers from './reducers';
//...

ReactDOM.render(<App />, document.querySelector('#root'));
ReactDOM.render(
  <Provider store={createStore(reducers)}>
    <App />
  </Provider>,
  document.querySelector('#root'));
```

Al crear un componente `Provider` el cual es encargado de brindar acceso a la store de `redux` a los componentes de nuestra aplicación. Ahora vamos a editar nuestro componente `flickr/src/components/ImageDetails.js` y vamos a hacer uso de la función `connect` para obtener acceso al estado global de `redux` en nuestro componente.

#### Listado 8.5 Utilizando función connect para obtener acceso al estado global de redux

```
import React from 'react';
import { connect } from 'react-redux';
```

```
const ImageDetail = props ({ img }) => {
const { img } = props;
  return (
    <div className="ui card">
      <div className="image">
        {img === undefined null ?
          <div className="ui placeholder">
            <div className="square image"></div>
          </div> :
          <img alt="alt" src={img.url_s} />
        }
      </div>
      <div className="content">
        {img === undefined null ?
          <div className="ui placeholder">
            <div className="header">
              <div className="very short line"></div>
              <div className="medium line"></div>
            </div>
          </div> :
          <a href="/" className="header">{img.id}</a>
        }
        <div className="meta">
          {img === undefined null ?
            <div className="ui placeholder">
              <div className="very short line"></div>
            </div> :
            <span className="date">{img.date}</span>
          }
        </div>
      </div>
    </div>
  )
}
```



```
    </div>
    <div className="description">
      {img === undefined null ? '' : img.title}
    </div>
  </div>
</div>
);
}

export default ImageDetail;
const mapStateToProps = (state) => {
  return { img: state.selectedImage }
}

export default connect(mapStateToProps, ImageDetail);
```

Con este cambio nuestro componente va a recibir la información del estado de redux mediante las propiedades por lo cual podemos quitar el envío de atributos desde el componente App, vamos a editar nuevamente el archivo `flickr/src/index.js` y vamos a eliminar el siguiente código:

#### Listado 8.6 Quitar atributos JSX de componente ImageDetail

```
// ...
<ImageDetail img={this.state.selectedImage} />
// ...
```

Ahora debemos actualizar el estado global de redux cuando el usuario interactúe con la lista para esto vamos a modificar nuestra lista de imágenes del archivo `flickr/src/components/ListItem.js` y vamos a agregar el siguiente código:

#### Listado 8.7 Quitar atributos JSX de componente ImageDetail

```
import React from 'react';
```

```
import { connect } from 'react-redux';
import { selectImage } from '../actions';

const ListItem = props => {
  // ...

  return (
    <div className="item" onClick={event => props.onClick selectImage
    (img)}>
      // ...
    </div>
  );
};

export default ListItem;
const mapStateToProps = (state) => {
  return { }
};

export default connect(mapStateToProps, { selectImage })(ListItem);
```

Ahora podemos observar que nuestra aplicación sigue funcionando normalmente sin embargo podemos remover los atributos enviados al componente `ListItem` para el click del elemento debido a que el componente ahora notifica a `redux` el cual actualiza el estado global y envía nuevas propiedades al componente `ImageDetail` ocasionando que se renderize de nuevo. Vamos a realizar la limpieza de los atributos del componente para verificar que nuestro `ActionCreator` éste modificando el estado global de `Redux`. Para iniciar vamos al componente `flickr/src/components/ImageList.js` para remover el envío del atributo `onClick`:

#### Listado 8.8 Remover atributo `onClick` enviado del componente `ImageList` al componente `ListItem`

```
import React from 'react';
```

```
// ...
const rows = images.map(img => {
  return (
    <ListItem key={img.id} img={img} onClick={props.onItemClick} />
  )
})
// ...
```

Y vamos también a remover el envío de propiedades en el archivo `flickr/src/index.js` en donde vamos a eliminar el siguiente código:

#### Listado 8.8 Remover atributos enviados de App a ImageList

```
// ...
onImageSelected = img => {
  this.setState({
    selectedImage: img
  })
}
+
// ...
<ImageList
  images={this.state.images}
  onClick={this.onImageSelected} />
//...
```

Podemos ver que nuestra aplicación se sigue ejecutando de forma normal, esto es debido a que el estado está siendo manejado ahora por `Redux` y por el estado del componente `App`.

Así mismo es importante resaltar que aunque se utiliza `Redux` para manejar el elemento seleccionado también se pueden seguir recibiendo propiedades como lo venimos haciendo en `ListItem` ambas propiedades pueden llegar al componente.