Curso de React.js

# ¿Qué es React.js?

**React** cumple su función como biblioteca ya que para utilizar su código se debe importar. También es un Framework aunque las convenciones de cómo debe ser organizado todo no son estrictas.  
En este curso aprenderás las prácticas que la comunidad ha decidido que son buenas.

**React es declarativo**, lo que quiere decir que se le indica qué debe hacer pero no cómo debe hacerse, ahorrando de esta manera muchos pasos.

**JSX** es HTML dentro de Javascript, esto se verá más adelante en detalle.

React está estructurado por **componentes** que son como pequeños bloques de lego que al ser unidos forman aplicaciones de React. Estos componentes pueden tener estilos, ser enlazados a eventos y sus estados pueden ser modificados.

Con React también se tiene la ventaja de que será escrito una sola vez y podrá ser utilizado en aplicaciones web, móviles, entre otras.

# Pre-requisitos

Estos son los conocimientos que deberás tener antes de comenzar con este curso:

* **Desarrollo web online:** Esto implica tener familiaridad y fortaleza en el uso de HTML y CSS.
* **Javascript:** React es Javascript. Es importante saber usar Javascript en el navegador. Es deseable conocer JQuery y saber sobre promesas, clases y tener conocimientos sobre asincronía.
* **Terminal:** La línea de comandos es indispensable para instalar herramientas, correr servidores y hacer diversas tareas.

# Herramientas que usaremos

Estas son las herramientas que usaremos en el curso:

* **Navegador:** Especialmente Chrome ya que cuenta con óptimas herramientas de desarrollo.
* **React Developer Tools:** Es una herramienta Open Source creada por Facebook y tiene instalación para Chrome o Firefox. Nos dejará ver el código de React inspeccionando elementos.
* **Editor de texto:** Puedes usar cualquiera, en este curso sugerimos **Visual Studio Code**. Tiene muchos plugins útiles para el desarrollo.
* **Prettier:** Es un plugin que hace que el código se vea bien sin importar cómo está escrito.

# Create-react-app

Para todos los que estan siguiendo el tutorial al pie de la letra. Les cuento que el comando npm install -g create-react-app ya fue sustituido por npx. Asi que lo que tiene que hacer es lo siguiente. En caso de haber instalado previamente create-react-app se debe desinstalar:  
npm uninstall -g create-react-app.  
Y luego usar el comando npx create-react-app hello-react  
En caso de no tener instalado npx. Sencillamente instalarlo con:  
npm install -g npx

Hola a todos; para aquellos que intentan correr Create-react-app y obtienen vulnerabilidades al finalizar de descarga. **No lo intenten hacer por medio del código del profesor** debido a que es **Obsoleto.🐛** Intenten borrar la carpeta que les creo, y vuélvanlo hacer con este comando en su terminal.

npx create-react-**app** my-**app**

**cd** my-**app**

npm start

Esto generara la carpeta del archivo sin vulnerabilidades **“Por mucho una vulnerabilidad baja”**

**Borren la instalación que hicieron globalmente que el profesor les recomendó, por medio del siguiente comando** ya que el comando que les recomendé anteriormente lo hace de forma local y no global.

npm **uninstall** -g **create**-react-app

Para mayor información pueden consultar en el siguiente link <https://create-react-app.dev/docs/getting-started>

Fundamentos

# Clonar el código de GitHub

En esta clase vamos a comenzar clonando el código del proyecto del repositorio en GitHub.

Es importante que tú y yo tengamos un punto de partida en común. Así vamos a poder asegurarnos que cada cambio que yo haga en el código, tú también lo recibas.

Para hacer esto, en la terminal ve a una carpeta donde quieras que exista el proyecto. Entonces escribes lo siguiente:

$ git **clone** **https**://github.com/Sparragus/platzi-badges.git

Eso va a clonar el repositorio del curso a una carpeta llamada platzi-badges.

Ahora es necesario que te muevas a esa carpeta.

$ cd platzi-**badges**

Ahora necesitamos instalar todas las dependencias necesarias para poder correr el proyecto. Lo haremos utilizando npm.

$ npm **install**

Este proceso puede tardar un poco. Lo que esta haciendo es descargando todas las bibliotecas de código que el proyecto necesita.

Una vez haya concluido, estamos listos para echar a correr el servidor. Lo hacemos con el comando

$ npm run start

Cuando el servidor comience, automáticamente va a abrir una pantalla en el navegador con la aplicación.

Si todo salió bien, vas a ver una pantalla que dice “Hello, Platzi Badges”.

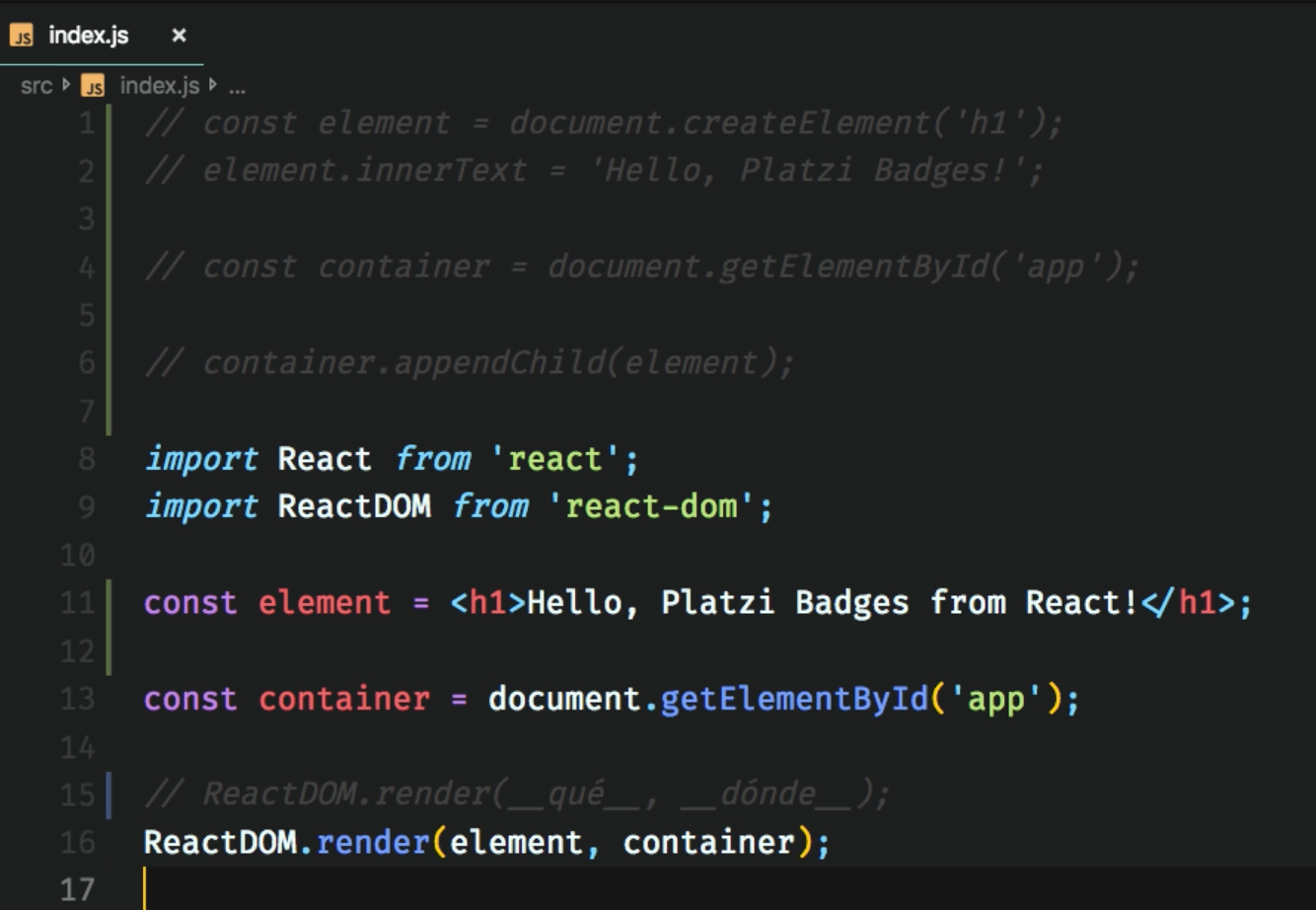
En la próxima clase vas a aprender como fue que “Hello, Platzi Badges” llegó desde el código hasta la pantalla de tu navegador.

\*[Aquí](https://github.com/sparragus/platzi-badges) encuentras el repositorio.

Lo que hice y me resultó, fué ir a la solapa de **"Archivos y enlaces"** y bajar el archivo **platzi-badges-1.ReactDOM.render.zip**. Lo descomprimí, modifiqué el nombre de la carpeta descomprimida a “**platzi-badges**”, luego ingresé a dicha carpeta y corrí **npm install**. Una vez terminado, corrí el comando **npm run start** y listo!  
Eso fue todo amigos!

# ReactDOM.render

* **React** y **ReactDOM** trabajarán en conjunto.
  + React como análogo a **createElement**
  + ReactDOM a **appendChild**
* **ReactDOM.render()** toma dos argumentos: Qué queremos renderizar y dónde lo queremos renderizar.
* Siempre que escribas **JSX** es requisito importar **React.**





# JSX

JSX es una extensión de JavaScript creada por Facebook para el uso con la biblioteca React. Sirve de preprocesador (como Sass o Stylus a CSS) y transforma el código generado con React a JavaScript.

JSX tiene su alternativa que es **React.createElement** pero es preferible JSX porque es mucho más legible y expresivo. Ambos tienen el mismo poder y la misma capacidad.

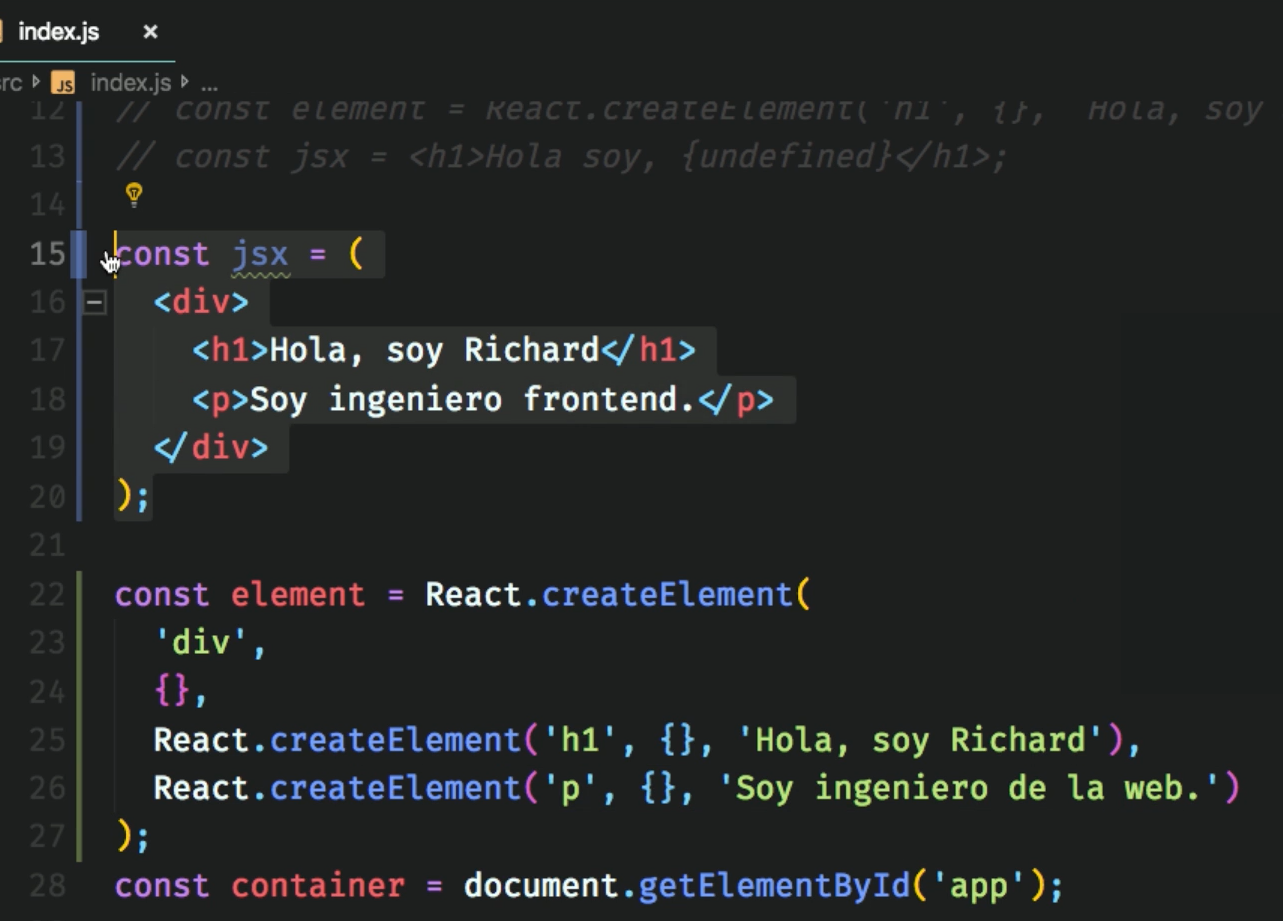
**React.createElement** recibe 3 argumentos:

* El tipo de elemento que estamos creando
* sus atributos o *props*
* y el *children* que es el contenido.

Ejemplo:  
React.createElement(‘a’, { href: ‘https://platzi.com’ }, ‘Ir a Platzi’);

En JSX se utilizan las llaves para introducir variables o expresiones de Javascript. Lo que sea que esté adentro se va a evaluar y su resultado se mostrará en pantalla.

Las expresiones pueden ser llamadas a otras funciones, cálculos matemáticos, etc. Si las expresiones son false, 0, null, undefined, entre otros, no se verán.



Creación y diseño de componentes

# ¿Qué es un componente?

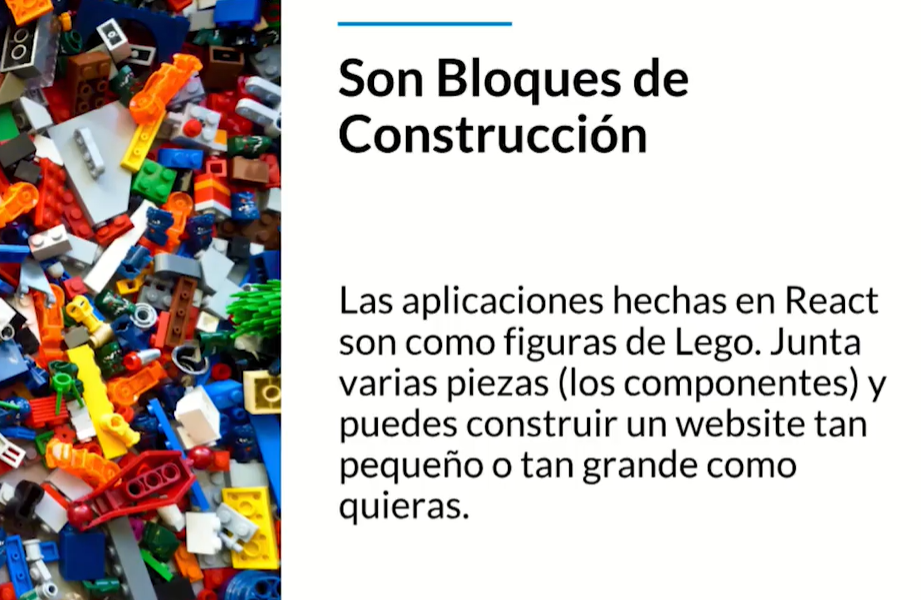
Los componentes en React son **bloques de construcción**.  
Las aplicaciones hechas con React son como figuras de Lego. Junta varias piezas (componentes) y puedes construir un website tan pequeño o tan grande como quieras.  
Los componentes serán barras de búsquedas, enlaces, encabezados, el *header*, etc.

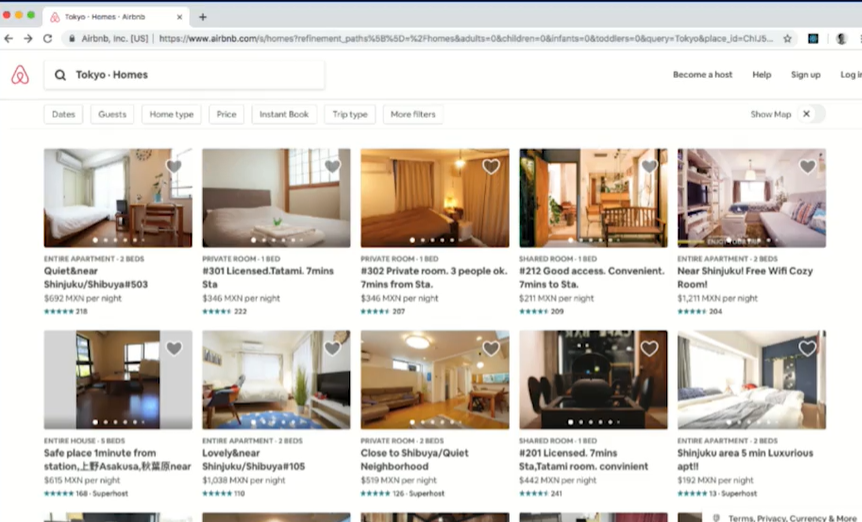
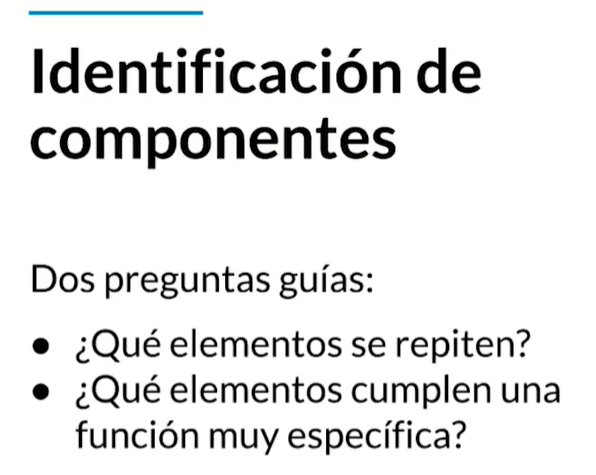
**”Componente” vs “elemento**  
Un elemento es a un objeto como un componente es a una clase. Si el elemento fuera una casa, el componente serían los planos para hacer esa casa.

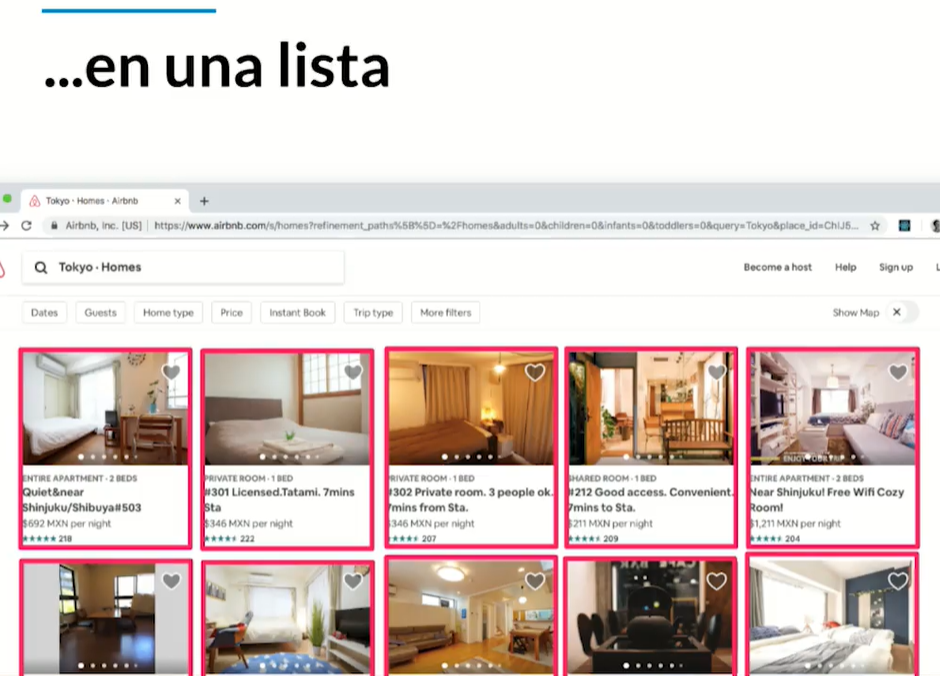
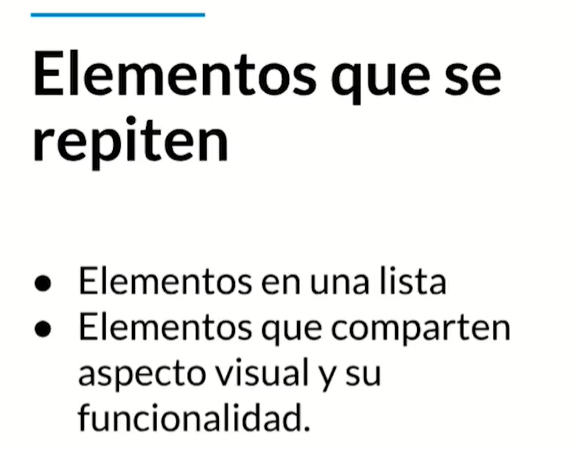
**Identificación de componentes**  
Para identificarlos debes hacerte las siguientes preguntas:

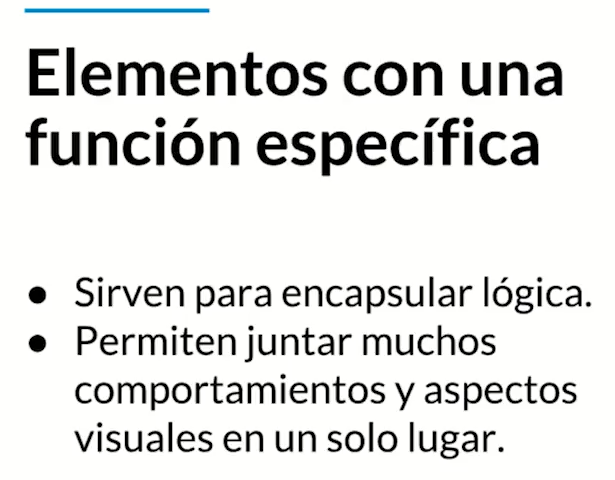
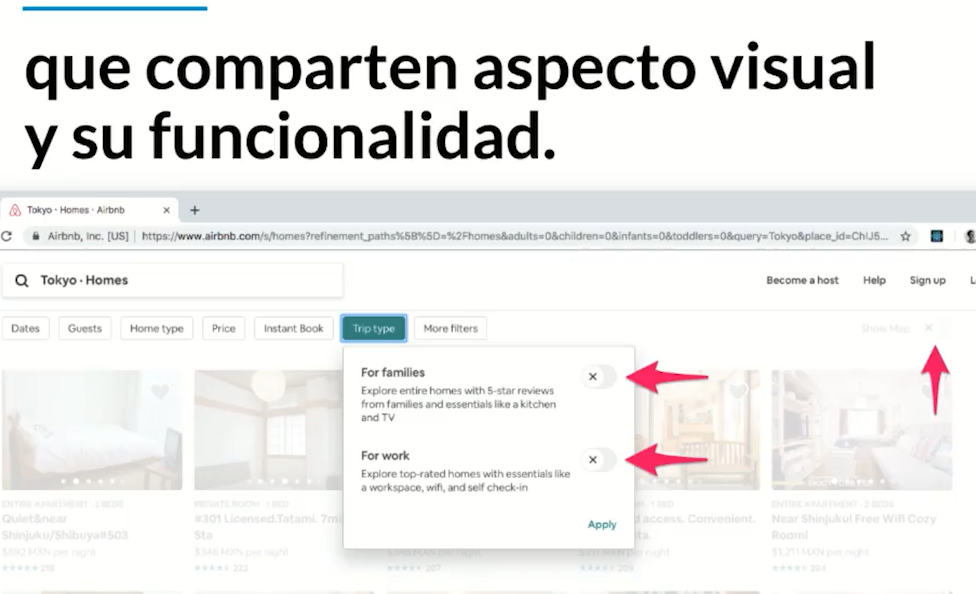
* ¿Qué elementos se repiten? Estos son los elementos en una lista o los que comparten aspecto visual y su funcionalidad
* ¿Qué elementos cumplen una función muy específica? Estos sirven para encapsular la lógica y permiten juntar muchos comportamientos y aspectos visuales en un solo lugar.

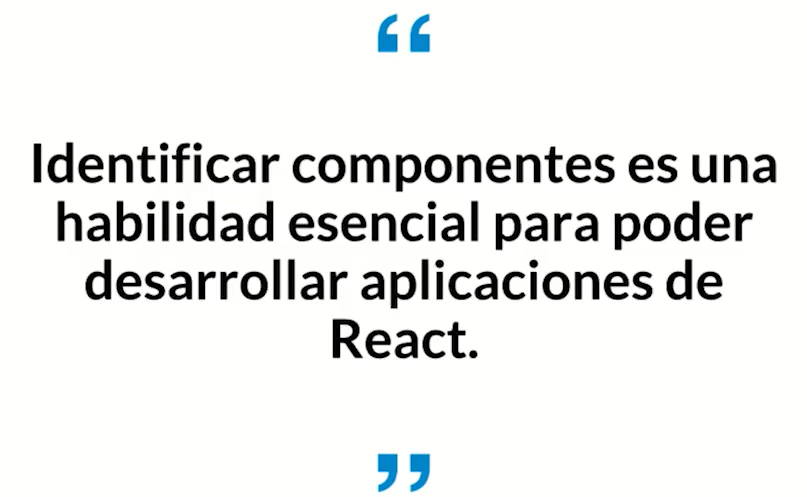
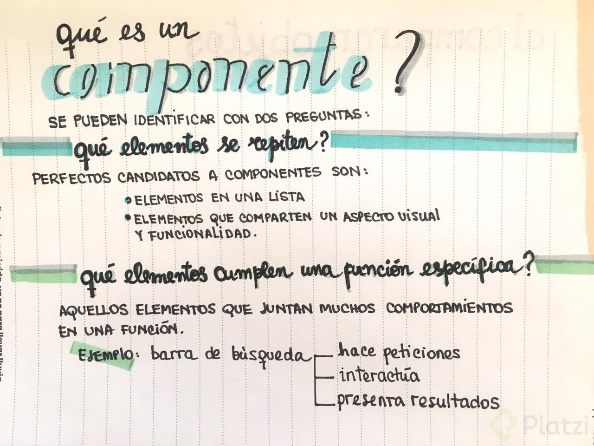
**Identificar componentes es una habilidad esencial para poder desarrollar aplicaciones de React.**



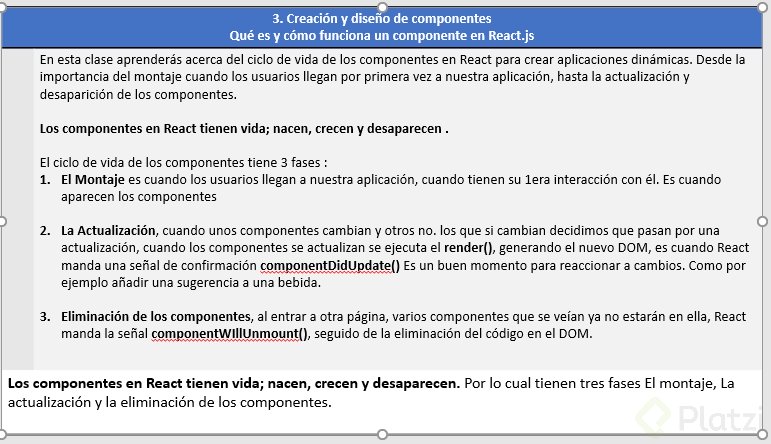


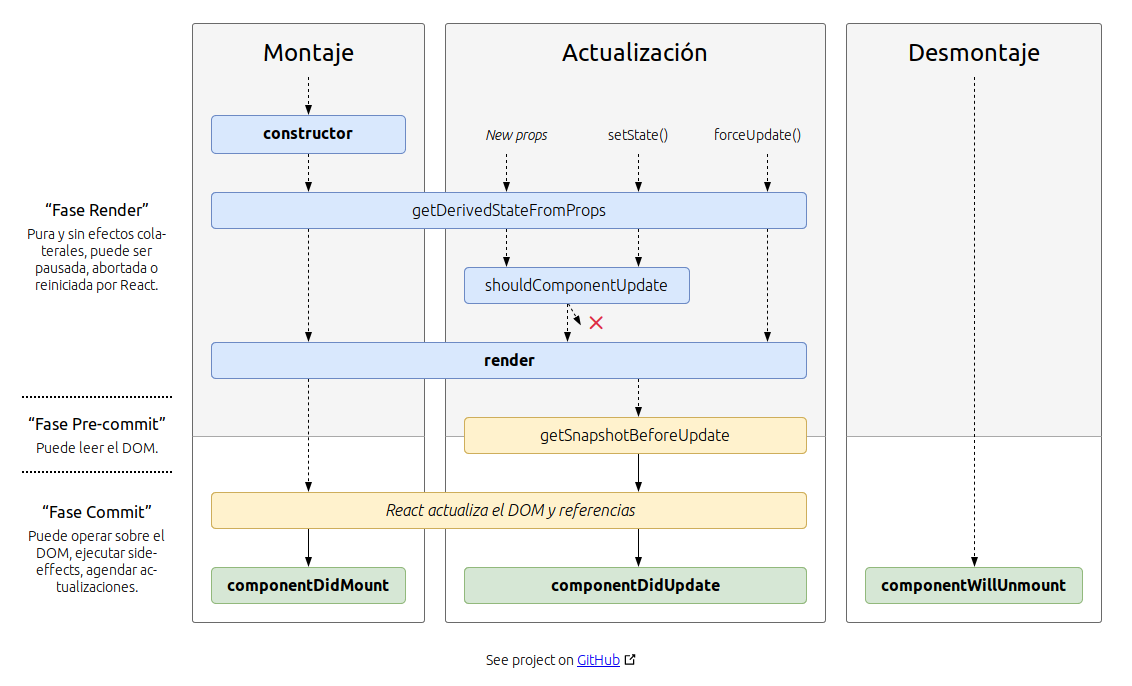




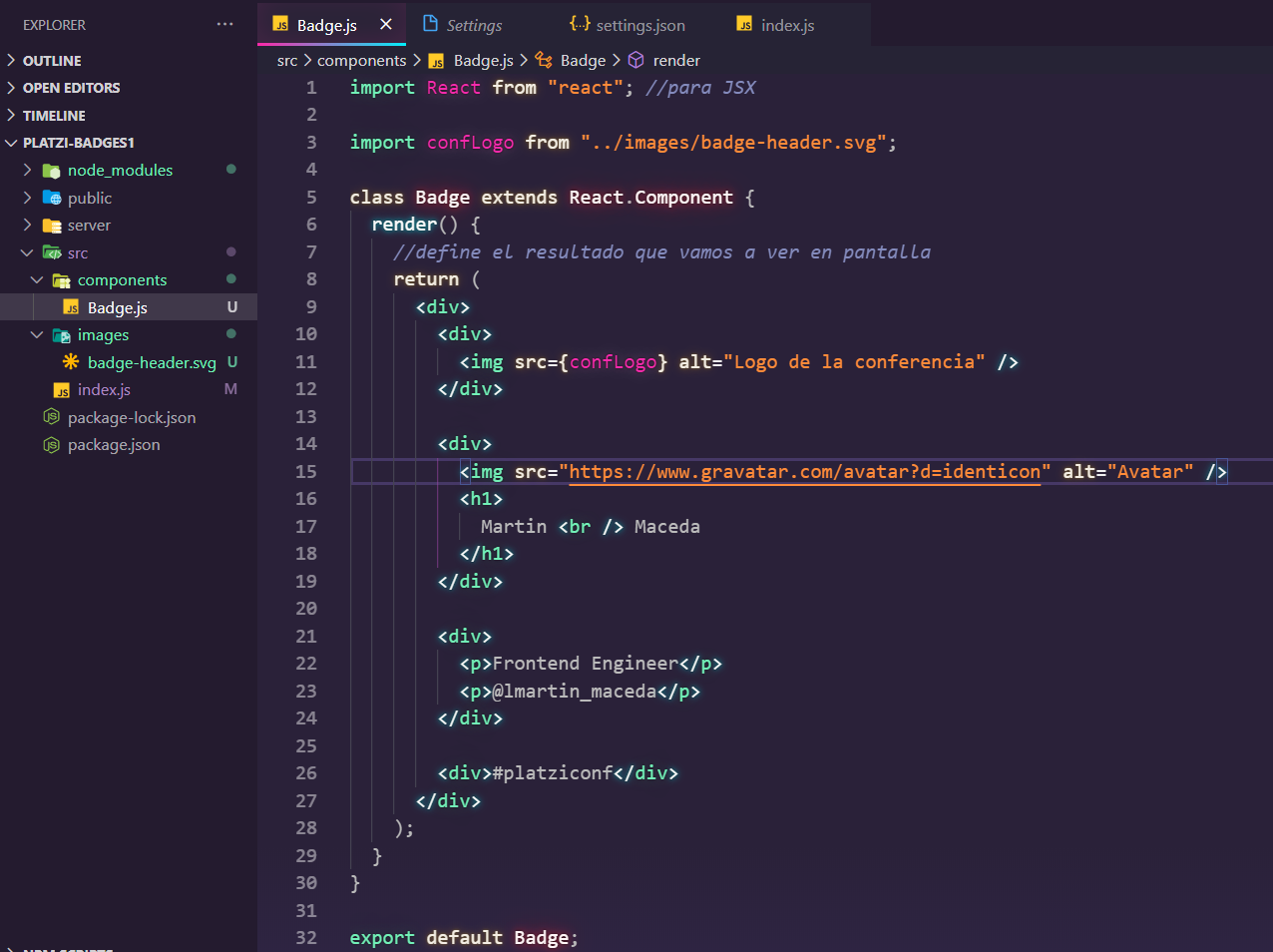
# Qué es y cómo funciona un componente en React.js





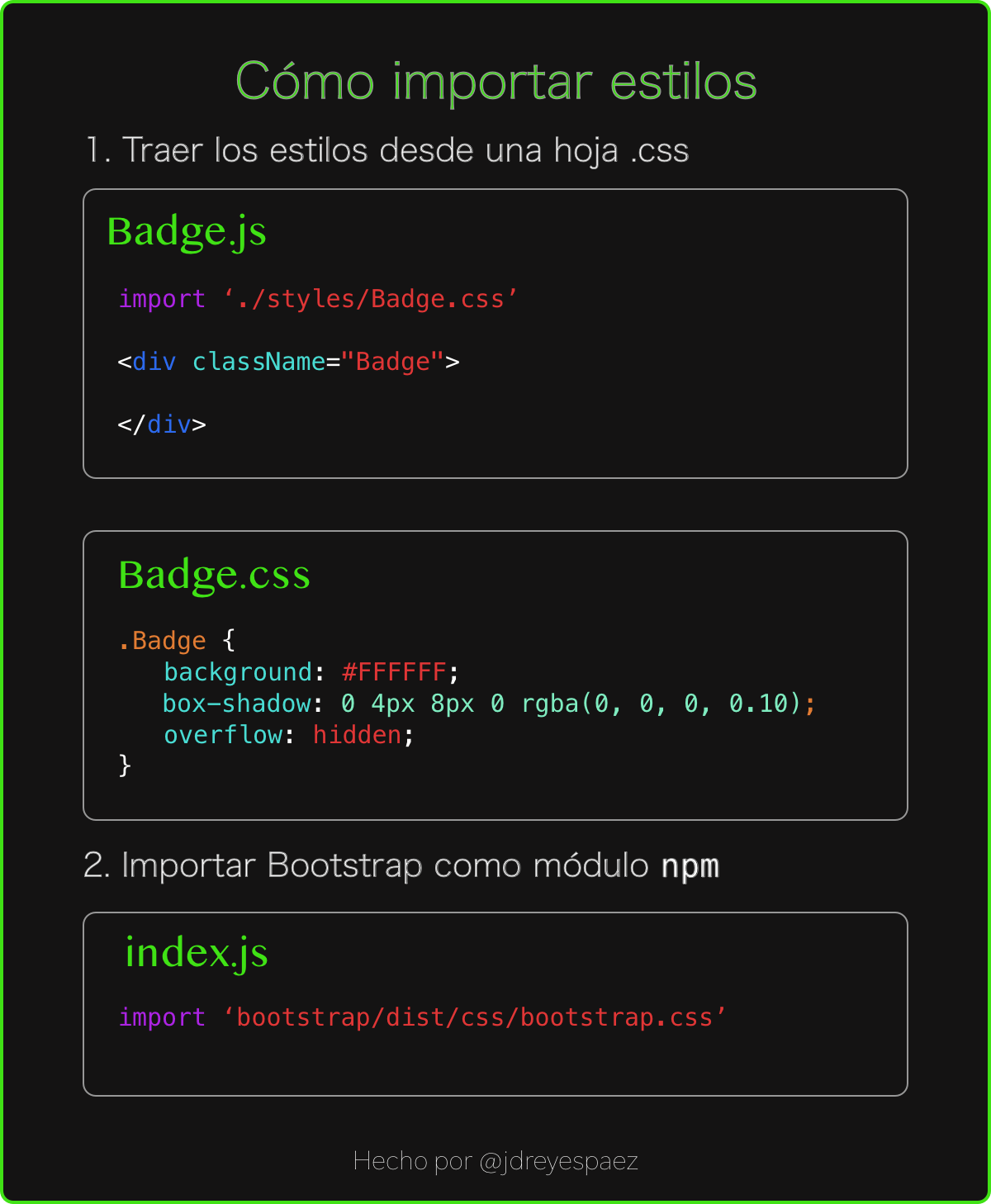
# Nuestro primer componente

* Es una buena práctica que los componentes vivan en su propio archivo y para ello se les crea una carpeta.
* Todos los componentes requieren por lo menos el método **render** que define cuál será el resultado que aparecerá en pantalla.
* El *source* de las imágenes en React puede contener direcciones en la web o se le puede hacer una referencia directa importándola. Si se importa deben usarse llaves para que sea evaluado.



# Cómo aplicar estilos

* Para los estilos crearemos una carpeta llamada **Styles** y allí vivirán todos los archivos de estilos que tienen que ver con los componentes.
* Para usar los estilos es necesario importarlos con *import*
* React funciona ligeramente diferente y para los atributos de clases no se utiliza *class* sino *className*
* Es posible utilizar **Bootstrap** con React, sólo debe ser instalado con npm install bootstrap y debe ser importado en el **index.js**
* Existen estilos que son usados de manera global o en varios componentes, así que deben ser importados en el index.js



En create-react-app tenemos una configuración de Webpack que nos permite importar estilos directamente en un archivo de javascript.  
.  
Para eso podemos importar directamente los estilos del componente con un importado del archivo

**import** './estilos.css'

.  
Ésto nos va a importar los estilos a nuestro archivo, pero los estilos no se aislan completamente (se importan en el momento de crear el componente, pero siguen interactuando con toda la aplicación)  
.  
Para evitar estos problemas de especificidad tenemos que generar convenciones de nombrado de estilos (clases), una de las más populares es BEM (Bloque\_\_elemento—modificador), ésto va a hacer que no se sobreescriban nuestros estilos.  
.  
Para aplicar los estilos sobre clases de CSS, la convención dice que envés de utilizar el atributo (prop) class, utilicemos className:

**const** Componente = ()=>(

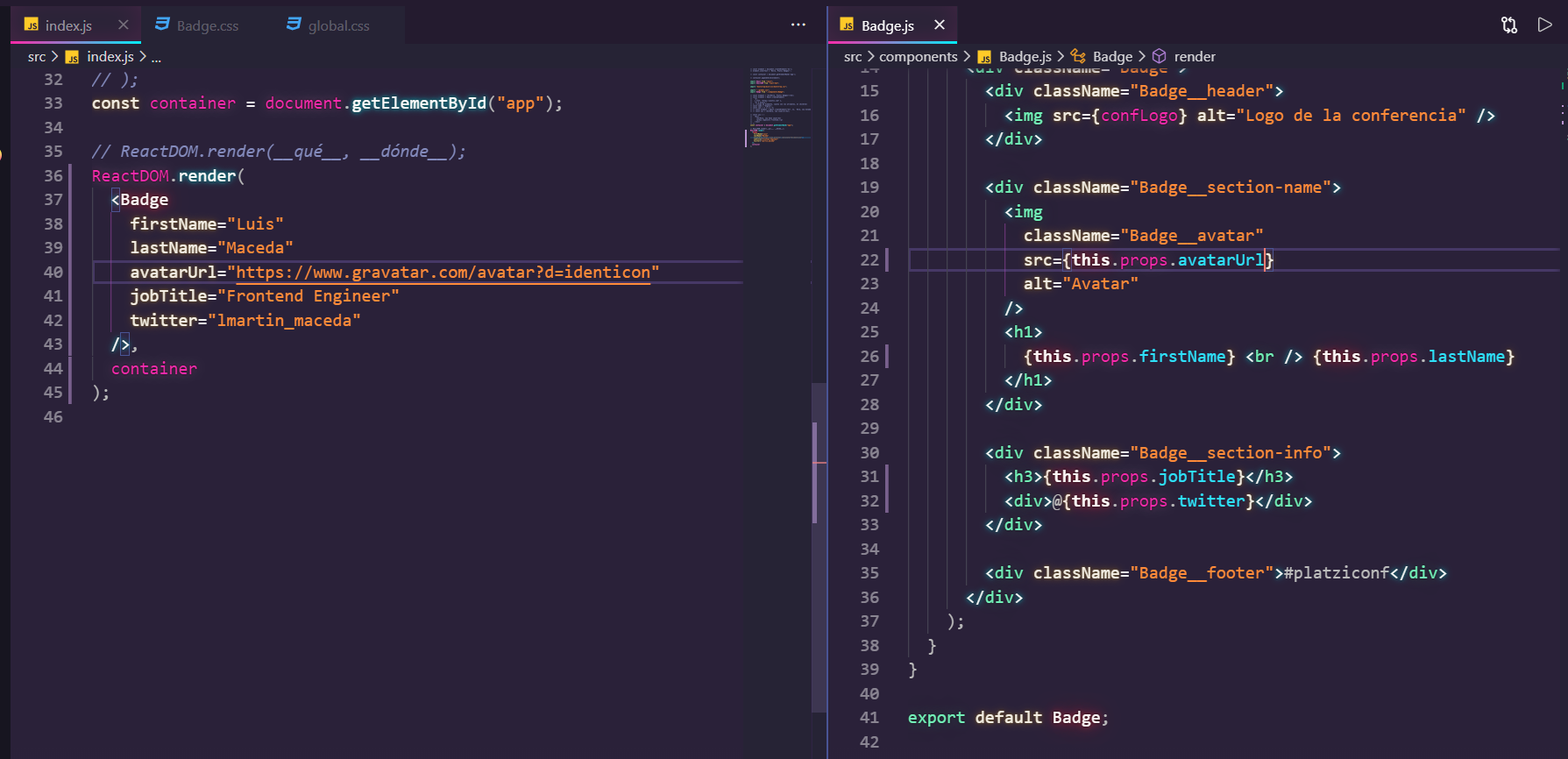
<div className="clase">Childrens del componente</div>

);

# Props

Los **props** que es la forma corta de properties son argumentos de una función y en este caso serán los atributos de nuestro componente como **class**, **src**, etc.

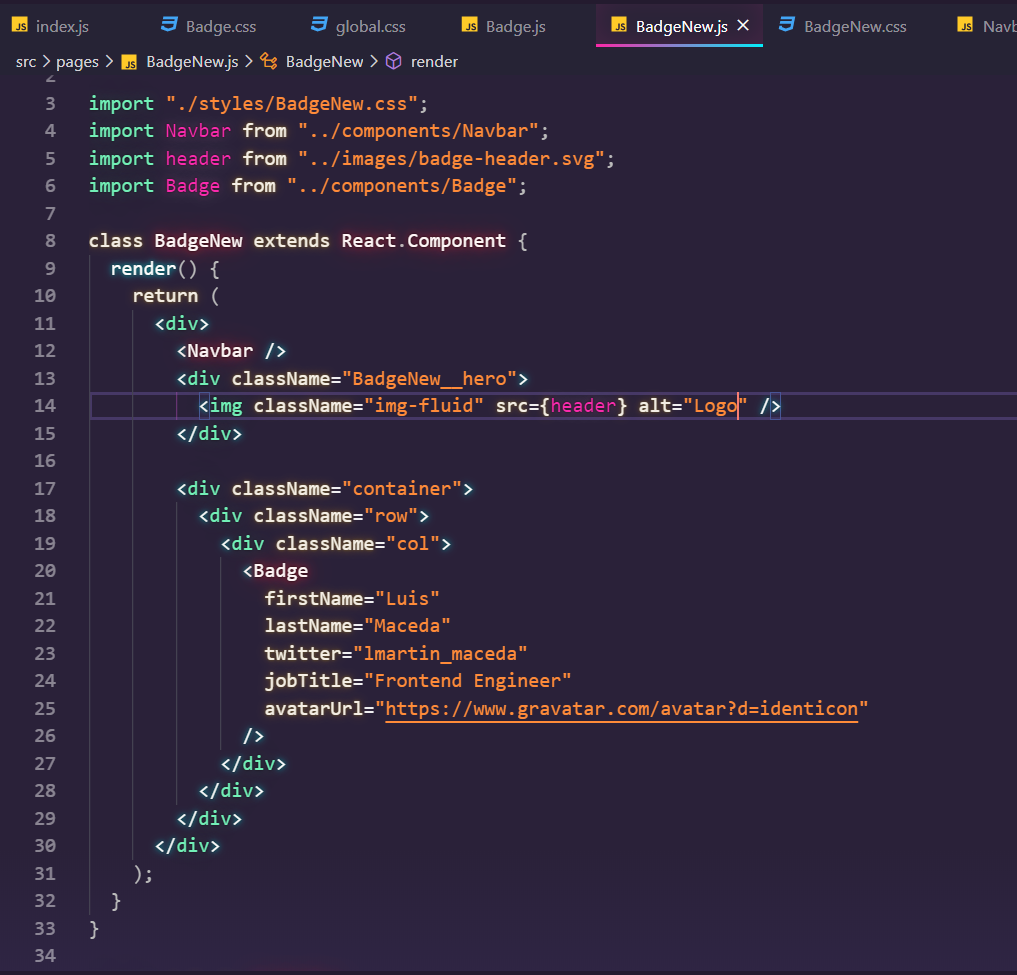
Estos props salen de una variable de la clase que se llama this.props y los valores son asignados directamente en el **ReactDOM.render()**.



# Nuestra primera página

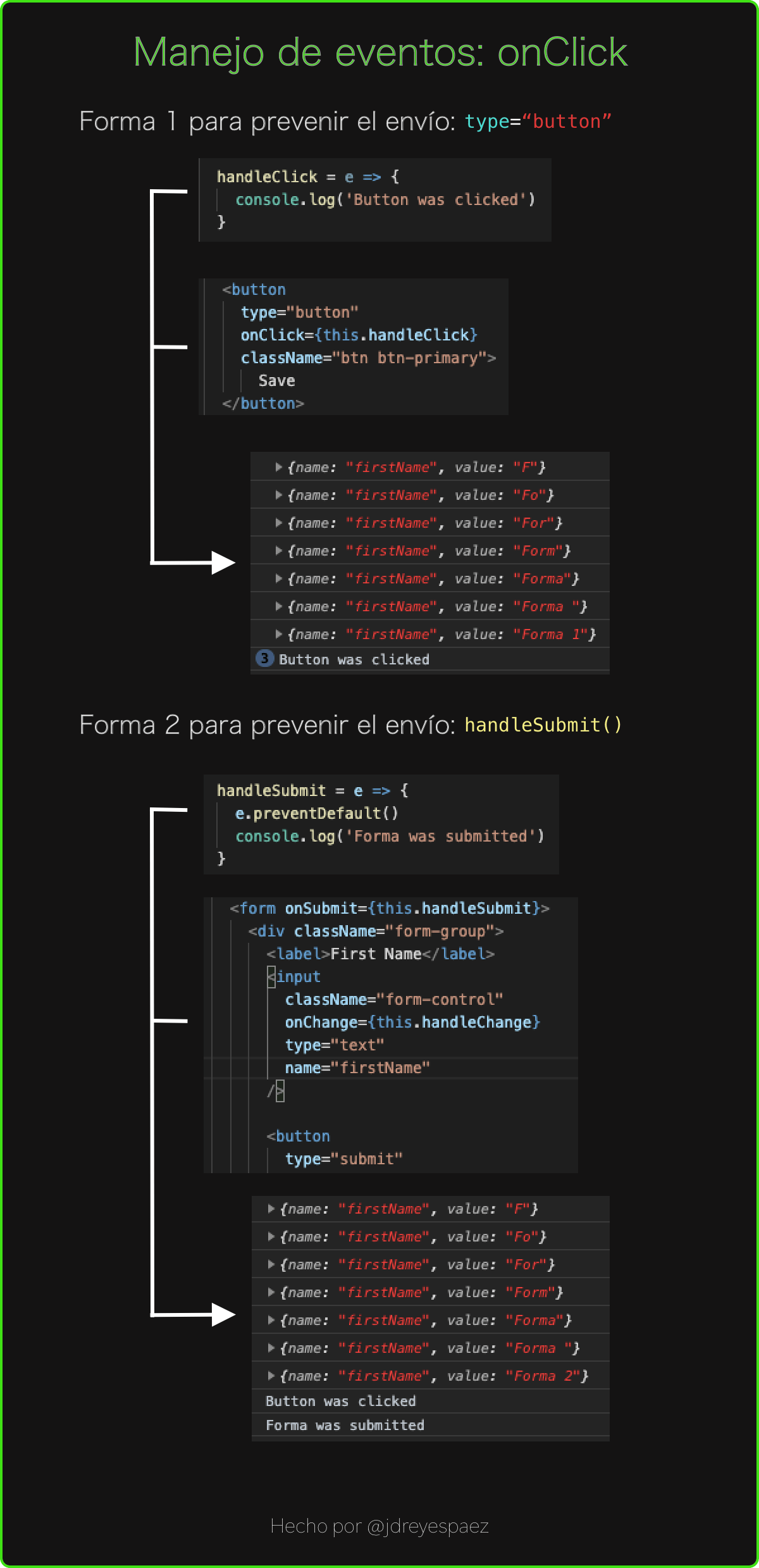
Las páginas en React son componentes y conseguir distinguirlas nos servirá para saber que es un componente que adentro lleva otros componentes.

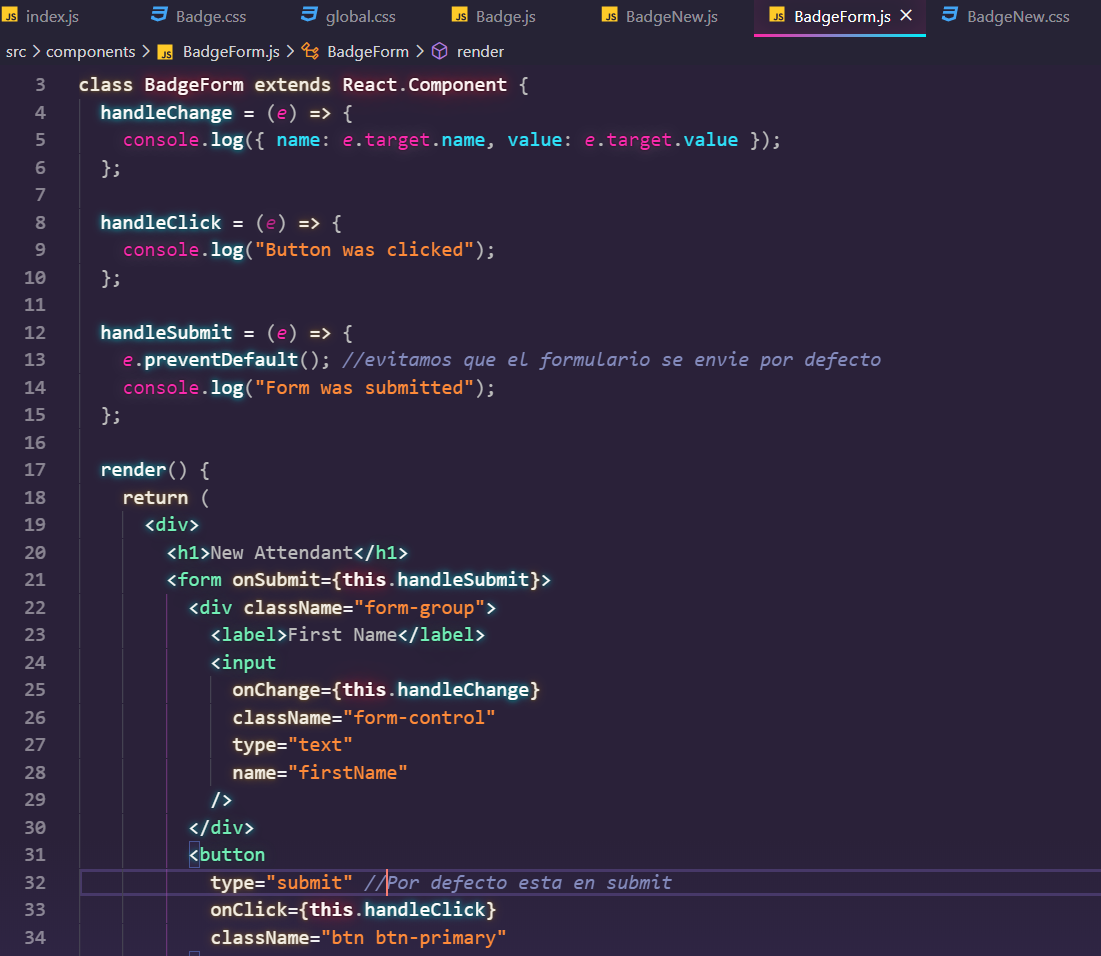
* Al escribir los *props* no importa el orden en el que lo hagas, únicamente importa el nombre.



# Enlazando eventos

* React dispone de **eventos**. Cada vez que se recibe información en un input se obtiene un evento **onChange** y se maneja con un método de la clase *this.handleChange*
* Los elementos *button* también tienen un evento que es **onClick**.
* Cuando hay un botón dentro de un formulario, este automáticamente será de tipo **submit**. Si no queremos que pase así hay dos maneras de evitarlo: especificando que su valor es de tipo *button* o manejándolo desde el formulario cuando ocurre el evento **onSubmit**.





# Enlazando eventos

Hasta esta clase todos los componentes han obtenido su información a través de *props* que vienen desde afuera (otros componentes) pero hay otra manera en la que los componentes pueden producir su propia información y guardarla para ser consumida o pasada a otros componentes a través de sus *props*. La clave está en que la información del **state** a otros componentes pasará en una sola dirección y podrá ser consumida pero no modificada.

* Para guardar la información en el estado se usa una función de la clase *component* llamada **setState** a la cual se le debe pasar un objeto con la información que se quiere guardar.
* Aunque no se ve, la información está siendo guardada en dos sitios. Cada *input* guarda su propio valor y al tiempo la está guardando en *setState*, lo cual no es ideal. Para solucionarlo hay que modificar los *inputs* de un estado de no controlados a controlados.

**Este es el resumen de esta clase**

El estado es por así decirlo un almacén que permite guardar los datos que produce un componente para ser utilizado para asimismo o ser pasado a otros componentes.

Para guardar o actualizar la información de un componente en su estado se usa una función llamada **setState()** la cual debe pasarse un objeto como parámetro para que sea guardada la información.

setState() puede ser utilizado cuando queramos guardar los datos (puede ser un formulario) después de escuchar un evento de un elemento (input, boton, etc).

**state**={}

handleChange = e => {

this.setState({

[e.target.name] : e.target.value,

})

}

Aunque no se ve, la información que se obtiene en el componente se guarda en dos sitios, en el state y en el elemento de un formulario (inputs, textarea, select, etc). Lo ideal es tener una sola fuente de información.

Para ello debemos modificar los elementos de un formulario de un estado **no controlados** (Normalmente los elementos de un formulario mantienen sus propios estados y los actualiza de acuerdo a la interacción del usuario) a **controlados** (*que react sea la única fuente de verdad,* es decir, que los valores obtenidos por los elementos del formulario sean controlados por react [*para mayor información*](https://es.reactjs.org/docs/forms.html#controlled-components) ).

Esto se logra agregando un props llamado **value** al elemento del formulario para que obtenga la información que contiene el estado del componte.

<**input** value={this.props.firstName}>

</input>

Cuando se recarga la pagina , el props llamado value del elemento del formulario (el input en este caso) intentara leer los datos del estado, por ello es necesario inicializar state (ya sea vació o no vació) en el componente.

**class** **BadgeForm** **extends** **React**.**Component**{

state={}

handleChange = e => {

[e.target.name] : e.target.value

}

}

# Levantamiento del estado

**Levantar el estado** es una técnica de React que pone el estado en una localización donde se le pueda pasar como *props* a los componentes. Lo ideal es poner el estado en el lugar más cercano a todos los componentes que quieren compartir esa información.

Algo interesante que le da el nombre a React es su parte de “reactivo” ya que cada vez que hay un cambio en el estado o en los *props* que recibe un componente se vuelve a renderizar todo el componente y todos sus descendientes.

Ahora guardaremos las variables en el BadgesNew.js que contiene el formulario y los badges, por lo que:

**En BadgeNew.js**

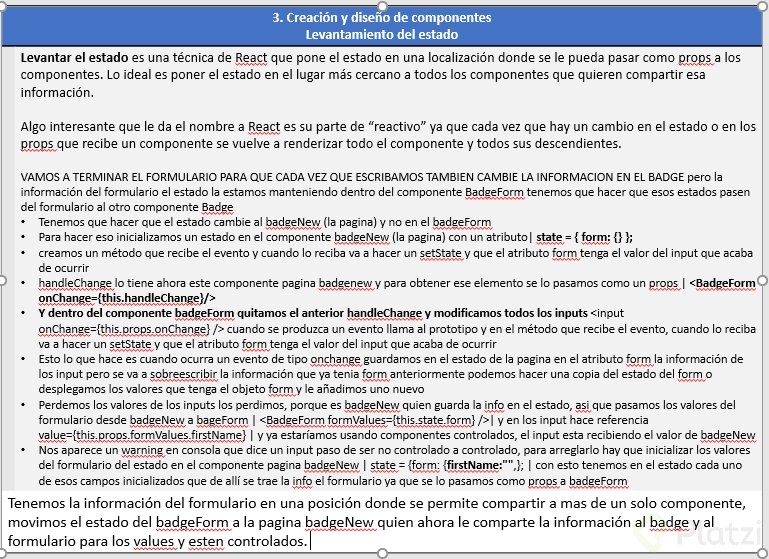
* Inicializamos el state y creamos un objeto form, con todos sus atributos inicializados vacíos tmbn
* Creamos el método handleChange para que cree una copia, la edite y la guarde en el state.form cada que suceda el evento onchange  
  Dentro del render
* Mandamos como props cada uno de los states del form a Badge
* Mandamos como props el arreglo del state.form a BadgeForm para que refleje el valor(?no estoy seguro)
* Declaramos el atributo onchange en BadgeForm para que llame al metodo handleChange

**En Badge.js**

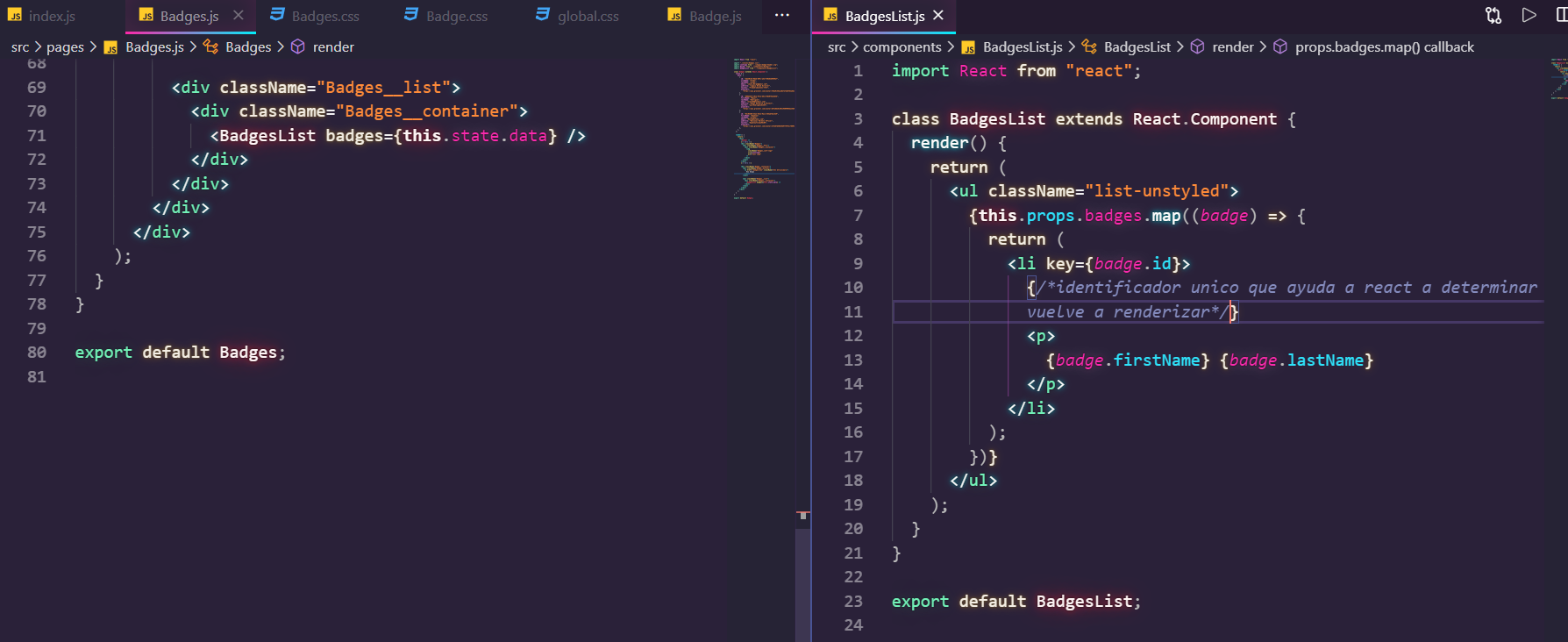
* Mostramos cada uno de los valores recibidos como [this.props.name](http://this.props.name/) dentro del render

**En el BadgeForm**

* Mostramos cada posicion en el atributo value con this.props.formValues.value
* Declaramos que al hacer un cambio se vera reflejado en el change padre, mediante el onchange hijo



# Listas de componentes



React Router

# Introducción a React Router

Las aplicaciones que se trabajan en React son llamadas **single page apps**. Esto es posible gracias a **React Router** que es una librería *Open Source*.

**Multi Page Apps:** Cada página implica una petición al servidor. La respuesta usualmente tiene todo el contenido de la página.

**Single Page Apps (SPA):** Aplicaciones que cargan una sola página de HTML y cualquier actualización la hacen re-escribiendo el HTML que ya tenían.

**React Router (v4):** Nos da las herramientas para poder hacer SPA fácilmente. Usaremos 4 componentes:

* BrowserRouter: es un componente que debe estar siempre lo más arriba de la aplicación. Todo lo que esté adentro funcionará como una SPA.
* Route: Cuando hay un match con el path, se hace render del component. El component va a recibir tres props: match, history, location.
* Switch: Dentro de Switch solamente van elementos de Route. Switch se asegura que solamente un Route se renderize.
* Link: Toma el lugar del elemento <a>, evita que se recargue la página completamente y actualiza la URL.

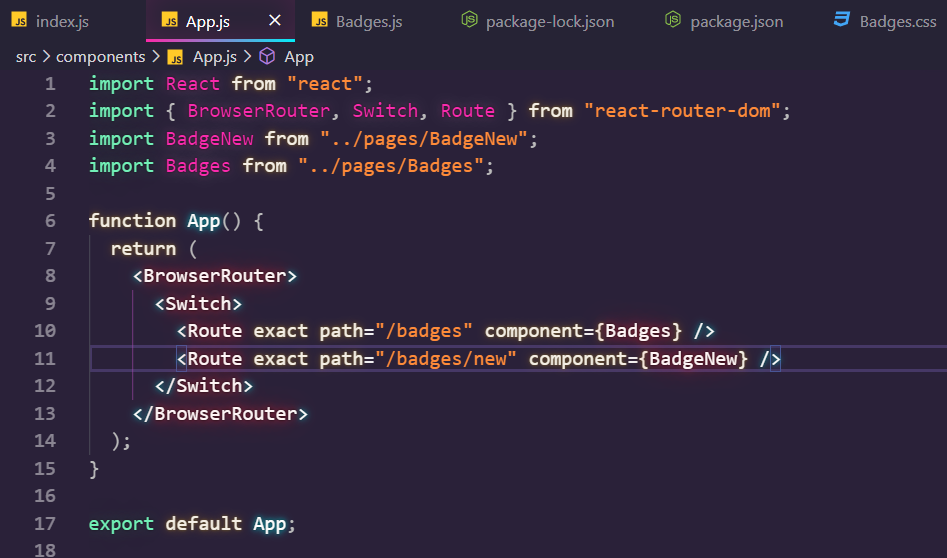
**React Router** tiene 4 componentes principales:

* Browser Router. Es necesario colocar el componente en la parte superior de la aplicación, lo que esté dentro de este va a servir como una **Single Page App** y les va a permitir al resto de las herramientas funcionar. Ya que fuera de él NO Funciona.
* Route. Representa una dirección de Internet, el **path** es donde va a renderizar el componente y el **componente** es el componente que va a renderizar, usualmente es una página.
* Switch. Componente que nos sirve para representar 1 sola ruta de varias que podemos poner dentro del componente.
* Link. Toma el lugar del elemento ancla, evita que se recargue completamente la página, actualiza la URL sin recargar la página completa

# División de la aplicación en rutas

Para instalar **React Router** lo hacemos desde la terminal con *npm install react-router-dom*. Como es importante usar exactamente la misma versión, del package.json en “dependencies” se quita lo que está delante del 4.

* **Link** internamente tiene un elemento <a> pero va a interceptar el clic para navegar de manera interna sin refrescar toda la página.



# Mejorando la User Interface con un Layout

[**Aquí encuentras el archivo para hacer el reto que propone el profesor al final de la clase**](https://drive.google.com/file/d/11C4nqBCTqF6dJMrVYMIs06sgwI5Y1KsF/view?usp=sharing)

* React.Fragment es la herramienta que te ayudará a renderizar varios componentes y/o elementos sin necesidad de colocar un div o cualquier otro elemento de **HTML** para renderizar sus hijos. Al usar esta característica de React podremos renderizar un código más limpio y legible, ya que ``React.Fragment` no se renderiza en el navegador.
* El 404 es la ruta que se renderizará cuando ninguna otra coincida con la dirección ingresada.

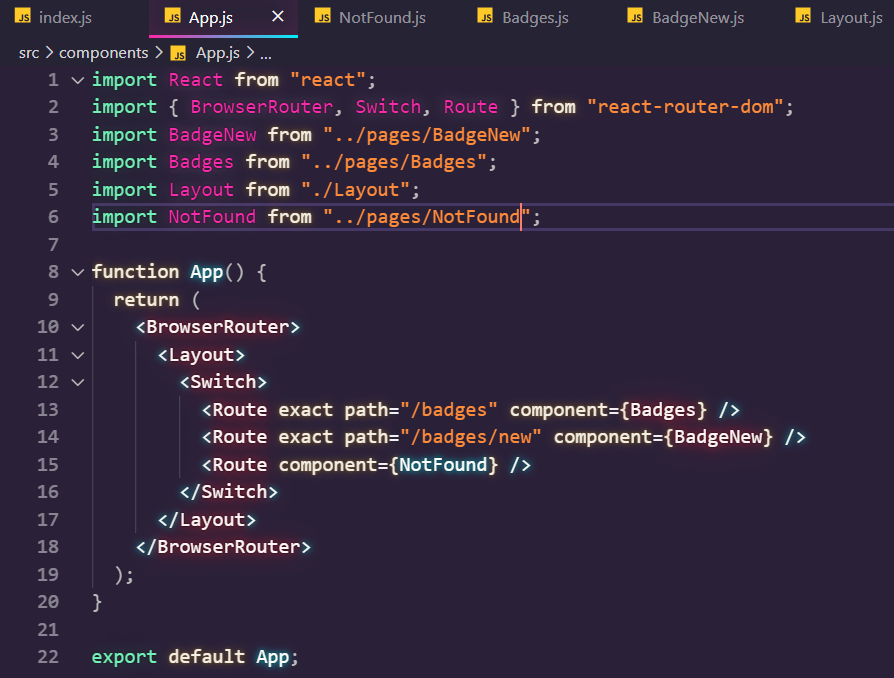
Otra forma de hacer que todas tus URL’s que no existan sean redirigidas a tu componente de 404 sería de la siguiente forma:

**import** { Redirect, Route } **from** "react-router-dom";

<Route path="/404" component={MiComponente404} />

<Redirect from="\*" to="/404" />

Como podemos observar llamamos a nuestro componente 404 y luego utilizamos Redirect, el cual es un componente de **React Router** para hacer redirecciones; en este caso hacemos que todas las URL’s que no correspondan a alguna que hayamos declarado, sean redirigidas a MiComponente404.



React Router

# Introducción del ciclo de vida de un componente

Cuando React renderiza los componentes decimos que entran en escena, cuando su estado cambia o recibe unos *props* diferentes se actualizan y cuando cambiamos de página se dice que se desmontan.

**Montaje:**

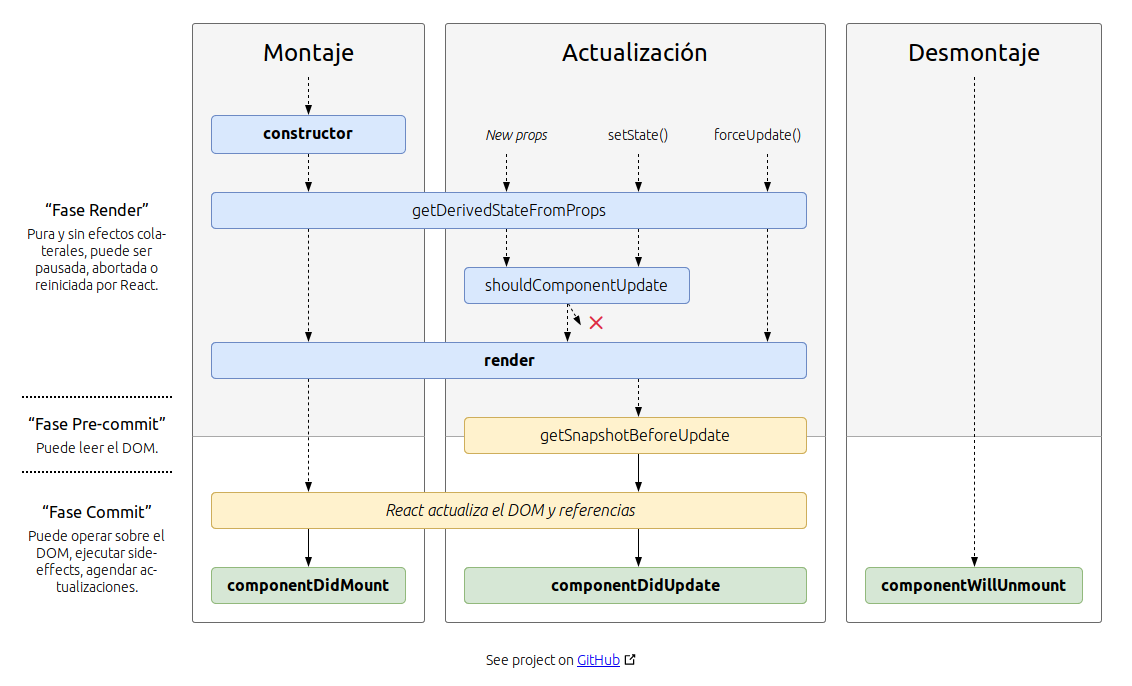
* Representa el momento donde se inserta el código del componente en el DOM.
* Se llaman tres métodos: *constructor*, *render*, *componentDidMount*.

**Actualización:**

* Ocurre cuando los *props* o el estado del componente cambian.
* Se llaman dos métodos: *render*, *componentDidUpdate*.

**Desmontaje:**

* Nos da la oportunidad de hacer limpieza de nuestro componente.
* Se llama un método: *componentWillUnmount*.



# Práctica del ciclo de vida

**Constructor**(props){ /\*Este método se ejecuta cuando se instancia un componente. Nos permite definir el estado inicial del componente, hacer bind de métodos y definir propiedades internas en las que podemos guardar muchos datos diferente\*/ }

**componentWillMonunt**(){ /\*Este método se ejecuta cuando el componente se está por renderizar. En este punto es posible modificar el estado del componente sin causar una actualización (y por lo tanto no renderizar dos veces el componente).\*/ }

**render**(){ /\*En este momento de la fase de montado se van a tomar las propiedades, el estado y el contexto y se va a generar la UI inicial de este componente\*/ }

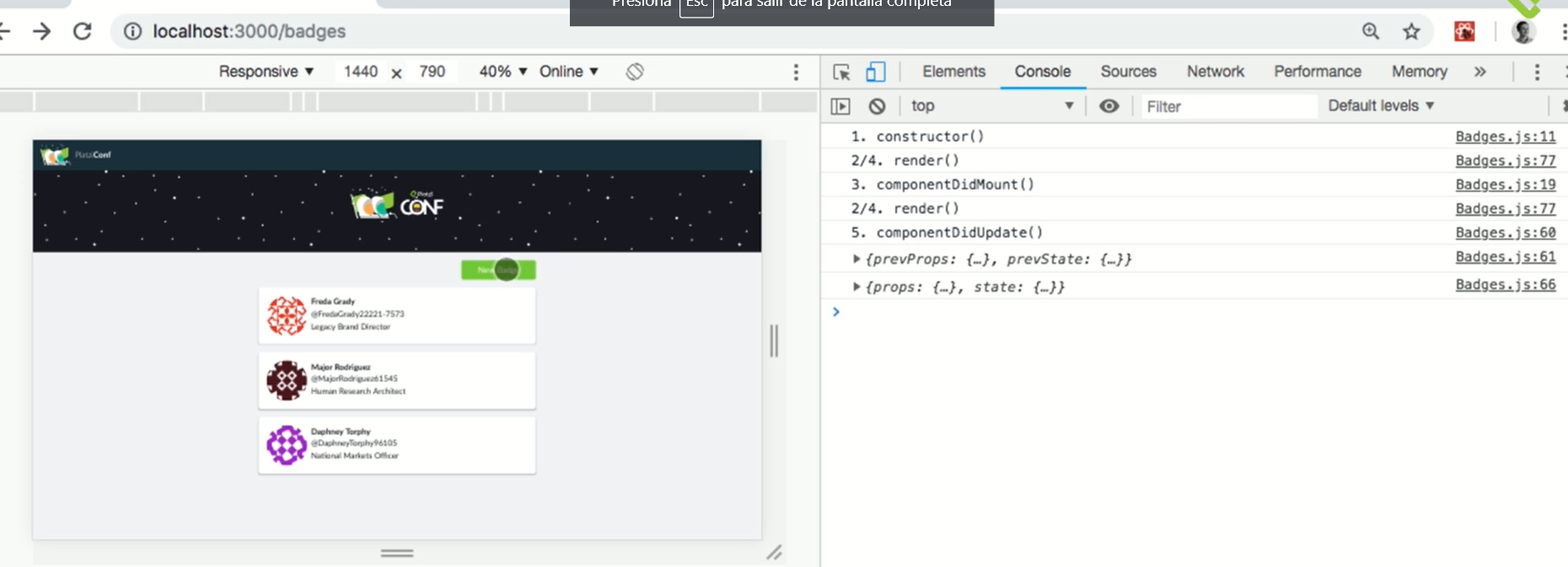
**componentDidMount**(){ /\*Este último método de la fase de montado se ejecuta una vez el componente se renderizó en el navegador y nos permite interactuar con el DOM o las otras APIs del navegador (geolocation, navigator, notificaciones, etc.).\*/ }

**componentWillReceiveProps**(nextProps){ /\*Este método se ejecuta inmediatamente después que el componente reciba nuevas propiedades. En este punto es posible actualizar el estado para que refleje el cambio de propiedades, ya sea reiniciando su valor inicial o cambiándolo por uno nuevo.\*/ }

**shouldComponentUpdate**(nextProps, nextState){ /\*Este método (el cual debe ser puro) se ejecuta antes de empezar a actualizar un componente, cuando llegan las nuevas propiedades (nextProps) y el nuevo estado (nextState). Acá es posible validar que estos datos sean diferentes de los anteriores (this.props y this.state) y devolver true o false dependiendo de si queremos volver a renderizar o no el componente.\*/ }

**componentWillUpdate**(nextProps, nextState){ /\*Una vez el método anterior devolvió true se ejecuta este método, acá es posible realizar cualquier tipo de preparación antes de que se actualice de la UI\*/ }

Todos estos son las fases del ciclo de vida de un componente en su montaje y actualización, no esta el desmontado, pueden ver la información completa [haciendo click aquí](https://platzi.com/blog/ciclo-de-vida-de-un-componente-de-reactjs/)



Llamadas a un API

# Introducción llamadas a un API

Las llamadas a una API siguen un patrón similar siempre que las hacemos, cada llamada consta de tres estados:

* **Loading**: cuando la petición se envía y estamos esperando.
* **Error**: se debe dejar un mensaje para el usuario para arreglar el error o volver a intentarlo.
* **Data**: los datos nos pueden llegar de dos formas, o en error o con los datos requeridos.

Una llamada a una API tiene tres estados

*Una promesa*

1. Loading  
   =>
   1. Error  
      ||
   2. Data
      1. Without data {}  
         ||
      2. With data {…}

* Es vital indicar que se está cargando para que el usuario tenga paz
* Si no hay data es vital hacer un CTA para introducir data



# React.js: Cómo traer datos de un API en React

Una llamada a una **API** es un proceso asíncrono, es decir que lo comenzamos pero no sabemos cuándo acabará. Por lo mismo la función a escribir debe ser asíncrona.  
La llamada se hará usando **fetch** que es una función de JavaScript que al pasarle una dirección de internet, hará una petición **GET** y lo que sea que exista ahí será devuelto.

# Solicitando datos (GET)

Para esta clase es necesario hacer ciertas cosas antes o no funcionará nada correctamente y no sabréis por qué. Falta un trozo de vídeo (creo) en la edición, y no se explican unos pasos intermedios necesarios. Estos pasos se entienden si se hacen y luego se mira el vídeo.

Lo primero, tenéis que descargaros 2 cosas, la carpeta ‘server’ junto con el archivo ‘api.js’. La carpeta server entera, a la altura de la carpeta src. El archivo api.js debe estar dentro de src. Lo tenéis todo en ‘Archivos y Enlaces’ arriba a la derecha.

Seguidamente es necesario abrir la terminal e instalar las siguientes dependencias:

faker (Crea un servicio ‘falso’ de datos para nuestra petición a la api creada por Richard)  
json-server (Crea nuestra falso puerto 3001 para la petición a la api)  
npm-run-all (cuidado, no es run-all que también existe)

Se pueden hacer al mismo tiempo todas:

**npm install faker json-server npm-run-all --save-dev**

Finalmente entramos en el archivo ‘package.json’ y reescribimos los ‘scripts’ para que quede de la siguiente forma:

“scripts”: {  
“start”: “npm-run-all -p client server”,  
“client”: “react-scripts start”,  
“server”: “json-server --port 3001 --watch server/db.json”,  
“seed”: “node server/seed.js”,  
“build”: “react-scripts build”,  
“test”: “react-scripts test”,  
“eject”: “react-scripts eject”  
}

Explicados:

**“start”: “npm-run-all -p client server”,**  
Ejecuta todos los scripts a una

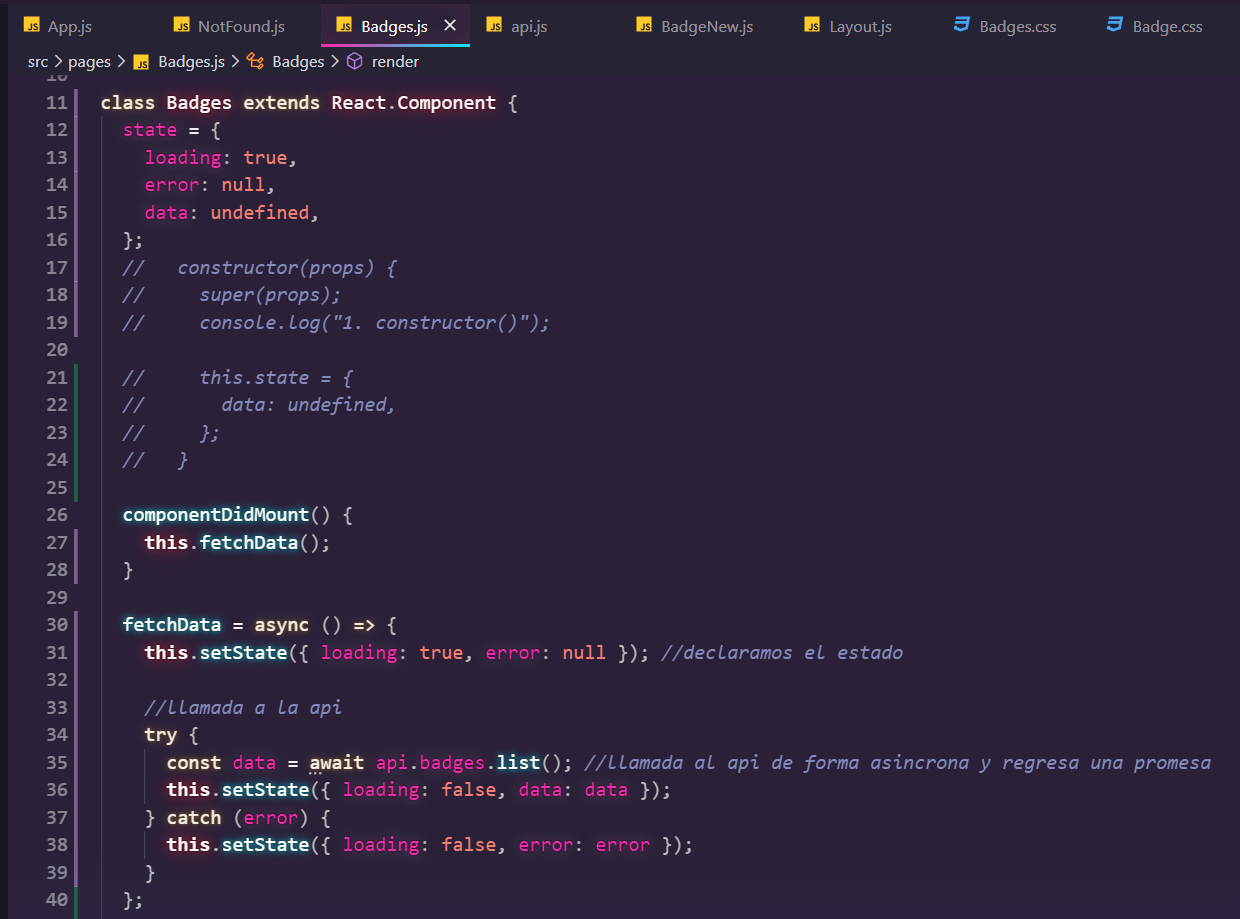
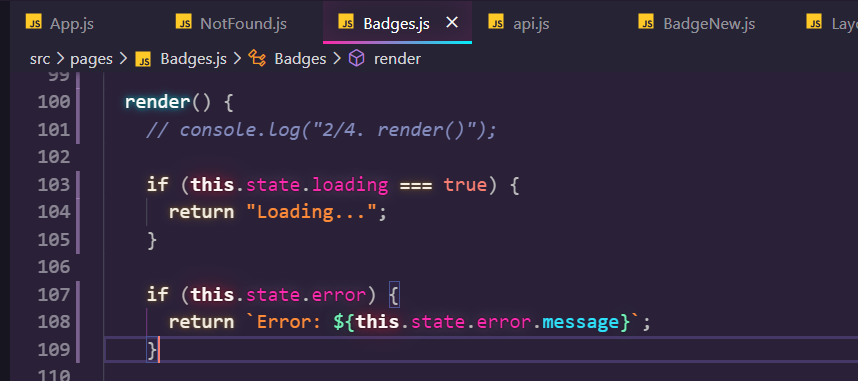
**“server”: “json-server --port 3001 --watch server/db.json”,**  
Crea un servidor falso con un puerto 3001 y unos datos que busca la api en el archivo especificado.

**“seed”: “node server/seed.js”,**  
Crea los datos falsos

Volvemos a la terminal y tecleamos:

npm run start

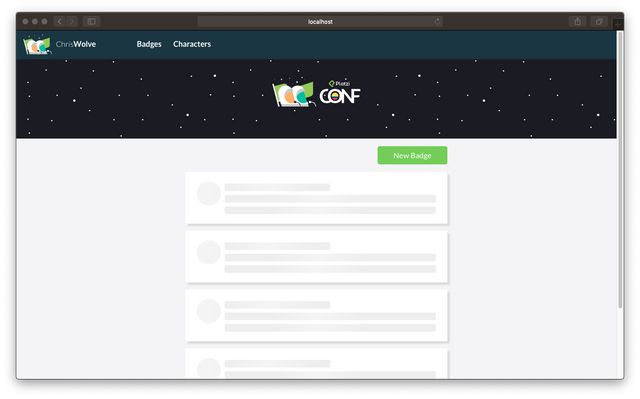
¡¡¡Listo!!!

# Mejorando la Experiencia de Usuario durante una petición



Nota:

Recomendaria usar **Skeleton** en lugar de un *Loader* genera mejor UX: [Skeleton for react](https://www.npmjs.com/package/react-loading-skeleton" \t "_blank)  


Generalmente las apis devuelven un **code error** dependiendo de la situación.

Con este codigo puedes personalizar el Error y manejar la excepción de la mejor forma, como notificaciones con el stack detallado, etc.

**class** **MyError** {

**constructor**(message,code) {

**this**.name = 'MyError';

**this**.code = code|| '500';

**this**.message = message || 'Default Message';

**this**.stack = (**new** Error()).stack;

}

}

MyError.prototype = Object.create(Error.prototype);

MyError.prototype.constructor = MyError;

**export** **default** MyError

...

throw **new** MyError(error.message,error.code);

