Formularios

Tiempo estimado: 30min

La introducción de datos por parte del usuario es una parte indispensable de la mayoría de aplicaciones webs de hoy en día. Para poder recibir datos de los usuarios, se utiliza los formularios HTML. A partir de ellos, se puede recolectar datos como, por ejemplo, el identificador y las credenciales de autenticación, la información de un nuevo empleado o de un nuevo pedido. En React, los formularios HTML requieren una especial atención, de ahí que les dediquemos una lección aparte.

La lección comienza recordando los formularios HTML y describiendo los tipos principales de componentes de entrada de datos. A continuación, se describe cómo definir un formulario HTML en una aplicación React, esto es, los componentes formularios. Después, se presenta los componentes de entrada de datos, tanto los controlados como los no controlados. Seguimos con determinados componentes de entrada de datos especiales como son las casillas de confirmación, los botones de opciones, los desplegables y los botones. Finalmente, se presenta los eventos de foco y el generador de Justo que ayuda a escribir aplicaciones React de manera más segura y fácil.

Al finalizar la lección, el estudiante sabrá:

- Cómo crear formularios HTML en React.
- Cómo definir elementos de entrada de datos en React.
- Cómo definir botones en React.

Introducción

Un formulario (*form*) es un elemento HTML que permite introducir datos al usuario, generalmente, para su envío al servidor web. Son uno de los principales puntos de interacción entre el usuario y la aplicación. Por ejemplo, a través de un formulario se puede proporcionar las credenciales de una cuenta de usuario, los datos de un pedido, de un empleado, de contacto, etc.

En una página web, se representa un formulario mediante un elemento <form>, el cual delimita los elementos a través de los cuales el usuario proporciona datos, conocidos formalmente como campos (*fields*). Además, contiene botones que puede usar el usuario para enviar los datos introducidos al servidor web, para vaciar el formulario, para realizar validaciones y otras muchas cosas.

Elemento form

El elemento <form> representa un formulario y actúa a modo de delimitador de los elementos de entrada de datos. Mediante el atributo method se indica el método HTTP usado por el navegador para remitir los datos al servidor. Los dos posibles valores son post y get.

Atendiendo al método HTTP elegido para enviar los datos, éstos se transmitirán en el cuerpo del mensaje o bien en la cadena de consulta del URL. Cuando se usa el método POST, los datos se envían en el cuerpo de la petición HTTP. En este caso, podemos indicarle al servidor el tipo de contenido mediante el atributo enctype del formulario.

El URL al que remitir el formulario se indica mediante el atributo action. Cuando no se indica, se remitirá al mismo recurso que contiene el formulario.

Veamos un ejemplo ilustrativo:

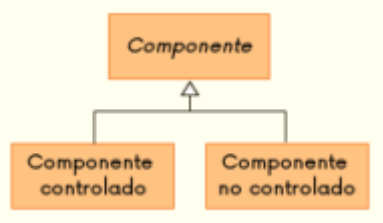
```
<form action="/form/login" method="post">
  <label>Username: <input type="text" name="username"></label>
  <label>Password: <input type="password" name="password"></label>
  <button type="submit">Log in"</button>
</form>
```

Elementos de entrada de datos

Los elementos de entrada de datos son <input>, <textarea> y <select>. Son especiales en las

aplicaciones React. Se encuentran implementados de manera particular, por lo que debemos prestar especial atención.

Se instancian mediante componentes React, los cuales se clasifican formalmente en controlados y no controlados.



Un componente controlado (*controlled component*) es aquel cuyos cambios son supervisados por React internamente. En cambio, un componente no controlado (*uncontrolled component*) no está supervisado.

Componente formulario

Por convenio y buenas prácticas, los formularios HTML se implementan mediante componentes React específicos en la carpeta app/components, al igual que cualquier otro componente (re)utilizable de la aplicación. Se recomienda que su nombre contenga el sufijo Form como, por ejemplo, LoginForm y ContactForm.

Ejemplo:

```
class LoginForm extends React.Component {
  …

  render() {
    return (
      <form>
        …
      </form>
    );
  }
}
```

Eventos de formulario

Básicamente, los formularios tienen dos eventos importantes:

Evento	Descripción
Change	Se genera cuando se produce un cambio en alguno de sus componentes de entrada de datos.
Submit	Se genera cuando el usuario hace clic en el botón de envío del formulario.

Por convenio y buenas prácticas, el método controlador del evento Change es handleChange(). Y el de Submit, handleSubmit().

Recordemos que los controladores de evento se clasifican en dos tipos: personalizados y predeterminados. Los personalizados son los que registramos explícitamente mediante el atributo onEvento. En cambio, los predeterminados son aquellos que vienen de fábrica con el *framework* y/o navegador y se ejecutan al finalizar la ejecución de los personalizados.

El comportamiento predeterminado del evento Submit es enviar el formulario al servidor. Si por cualquier circunstancia no deseamos que se envíe, no hay más que invocar el método preventDefault() del objeto evento. Esto es habitual si la aplicación React es la que debe enviar explícitamente los datos al servidor, por ejemplo, mediante los paquetes http o request, la biblioteca jQuery o las APIs WebSockets o Fetch. En estos casos, la plantilla base para el controlador Submit será muy parecida a lo siguiente:

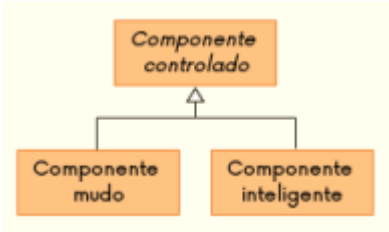
```
handleSubmit(evt) {
  //(1) no enviar el form: cancelación del comportamiento predeterminado
  evt.preventDefault();
```



```
    //(2) procesamiento particular del evento medinte fetch
    fetch(…);
  }
```

Componentes controlados

Un componente controlado (*controlled component*) es aquel elemento HTML de entrada de datos cuyos cambios son supervisados por React. Recordemos que los elementos de entrada de datos son <input>, <textarea> y <select>. Atendiendo a si el usuario puede modificar su valor, se distingue entre componentes mudos e inteligentes.



Un componente mudo (*dumb component*) no puede cambiar su valor inicial, por lo que también se conoce como componente de sólo lectura (*read-only component*). En cambio, un componente inteligente (*smart component*) sí que puede cambiarlo.

Ambos tipos de componente se instancian mediante sus elementos HTML correspondientes, esto es, <input> y <textarea>. Para determinar ante qué tipo nos encontramos, React usa, por un lado, los siguientes atributos especiales del elemento:

- value. El valor actual del componente.
- onChange. Controlador del evento Change, a través del cual se notifica que el usuario ha cambiado el valor.

Por otro lado, se debe tener en cuenta cómo se trabaja con el componente: mediante el objeto propiedades o el objeto estado. Vamos a ver cada uno detenidamente.

Componentes mudos

Un componente mudo (*dumb component*) es un componente controlado cuyo valor no cambia a lo largo de todo su ciclo de vida. El componente no responde a los cambios de valor solicitados por el usuario. Podemos decir que es inmutable, de sólo lectura. Por esta razón, su valor se suele fijar a partir de una propiedad del objeto props del formulario. Hace uso del atributo value, el cual contendrá el valor. Pero en cambio, no registra controlador para el evento Change, pues nunca cambiará.

He aquí un ejemplo ilustrativo:

```
<input type="text" name="fullName" value={this.props.fullName} />
```

Es importante recordar que:

- El valor del componente se asigna mediante el atributo value.
- El valor del componente se suele fijar a partir del valor de un campo del objeto props del formulario.
- No hay que registrar controlador para el evento Change del componente.

Componentes inteligentes

Por su parte, un componente inteligente (*smart component*) es un componente controlado que sí responde a los cambios de valor solicitados por el usuario. A diferencia de los componentes mudos que son de sólo lectura, los inteligentes son de lectura/escritura. Su valor se asigna mediante el atributo value del elemento.

Veamos un ejemplo con el que comenzar:

```
<input type="text"
       name="fullName"
       value={this.state.fullName}
       onChange={(evt) => this.handleChange(evt)} />
```


Estos componentes permiten que el usuario cambie su valor. Pero es muy importante tener claro que este cambio debemos trasladarlo nosotros mismos al estado del formulario. Para ello, nos serviremos del controlador del evento Change del elemento. Cada vez que el usuario cambie el valor, el componente generará el evento y el controlador deberá capturarlo. Para realizar el cambio, no hay más que actualizar el campo del estado del formulario asociado al elemento de entrada. Los datos recolectados por los componentes inteligentes *se almacenan en el estado del formulario*. No se suele acceder al campo value más que en la definición del elemento y en el controlador del evento Change. Es muy importante recordarlo.

Por convenio y buenas prácticas, se recomienda controlar los cambios mediante el método handleChange() del formulario, usando la propiedad target del objeto evento para conocer el elemento modificado por el usuario. He aquí un ejemplo ilustrativo:

```
handleChange(evt) {
  const comp = evt.target;

  if (comp.name == "fullName") this.setState({fullName: comp.value});
  else if (comp.name == "birthDate") this.setState({birthDate: comp.value});
}
```

Recordemos que cuando se realiza un cambio en el estado, automáticamente se dispara una reproducción por actualización. Entonces, cuando se regenere el formulario, los componentes detectarán automáticamente si su valor ha cambiado. Si no lo han hecho, no se reproducirán de nuevo, recordemos que así se reduce la carga y se mejora el rendimiento de la aplicación. En cambio, si el valor es distinto, el elemento <input>, <textarea> o <select> se reproducirá de nuevo y, entonces, su atributo value se fijará al valor indicado por su expresión asociada. Que casualidades de la vida, procede del estado. Más concretamente del campo del estado asociado al elemento.

Observemos la diferencia entre los elementos mudos y los inteligentes. Los mudos no cambian y *suelen* recibir su valor del objeto propiedades del formulario. Mientras que los inteligentes son de L/E y reciben su valor del objeto estado del formulario que contiene el valor actual de los datos recolectados. Por otra parte, los mudos al no cambiar su valor, nunca generarán el evento Change y, por lo tanto, no deben definirle controlador. En cambio, los inteligentes cada vez que el usuario cambia su valor generarán el evento, el cual debe ser capturado para actualizar explícitamente el valor. Esta actualización se realiza mediante una reproducción de actualización, la cual recordemos por *enésima* vez se dispara mediante el cambio del estado, esto es, el método setState() del formulario.

Veamos un ejemplo. Supongamos un formulario de *login*. Este formulario se implementa mediante un componente React específico, el cual contiene dos instancias del componente <input> y un botón. Los datos de *login* se almacenan en el estado del formulario e inicialmente estará vacío. Para controlar los cambios, es decir, para cambiar el estado cuando el usuario introduzca su nombre y contraseña, usaremos el método controlador handleChange() del formulario. Veamos el código:

```
/**
 * Formulario de login.
 */
class LoginForm extends React.Component {
  /**
   * Constructor.
   */
  constructor(props) {
    super(props);

    this.state = {
      username: "",
      password: ""
    };
  }

  /**
   * Controlador del evento Change de los elementos de entrada de datos
   * del formulario.
   */
  handleChange(evt) {
    const target = evt.target;

    this.state({
```



```
      [target.name]: target.value
    });
  }

  /**
   * Controlador del evento Submit del formulario.
   */
  handleSubmit(evt) {
    //...
  }

  /**
   * Método reproductor del componente.
   */
  render() {
    <form onSubmit={(evt) => this.handleSubmit(evt)}>
      Usuario: <input type="text" name="username"
                      value={this.state.username}
                      onChange={(evt) => this.handleChange(evt)} />
      <br />
      Contraseña: <input type="password" name="password"
                         value={this.state.password}
                         onChange={(evt) => this.handleChange(evt)} />
      <br />
      <input type="submit" />
    </form>
  }
}
```

Ahora, vamos a comprender mejor el funcionamiento. Cuando el formulario se reproduce en el DOM del documento, se llevará a cabo la fase de montaje, en la cual se invoca el método render() para obtener la representación inicial del formulario. En este caso, como el estado del componente es {username: "", password: ""}, el formulario no mostrará nada en los cuadros de texto.

Cuando el usuario escriba su nombre de usuario, React generará el evento Change sobre el componente username. Entonces, el formulario lo captura mediante su método controlador handleChange(). El cual modifica el estado del formulario mediante el método setState(). Entonces, se dispara una reproducción de actualización, mediante la cual se reproducirá de nuevo el componente username. Y como ahora el estado contiene un valor distinto para el campo username, entonces el <input> se representará mediante otro <input> cuyo valor es el nuevo valor introducido por el usuario. Así se consigue que el cambio se traslade al estado y al propio componente.

A continuación, cuando el usuario inserte su contraseña, se producirá el mismo proceso. Pero ahora, el cambio de estado afectará sólo a la contraseña y a la nueva representación del <input name="password">.

Estado de formulario

Generalmente, los datos del formulario y de sus componentes de entrada se almacenan en su estado o bien en su objeto props. Es muy importante que cuando usemos los componentes inteligentes, definamos los campos del estado inicial a sus valores vacíos. Por convenio y buenas prácticas, se destina un campo a cada elemento de entrada de datos, haciendo que sus nombres y los atributos name sean homónimos.

Así por ejemplo, en un formulario de *login*, tendremos básicamente dos datos: username y password. El constructor del componente formulario debe de definir los campos username y password, por ejemplo como sigue:

```
constructor(props) {
  super(props);

  this.state = {
    username: "",
    password: ""
  };
}
```

Y por otra parte, el formulario debe de definir dos elementos <input>, uno con name a username y otro a password:

```
render() {
```



```
    return (
      <form …>
        <input type="text" name="username" … />
        <input type="password" name="password" … />
        …
      </form>
    );
  }
```

Evento Change

El evento Change se genera cada vez que se produce un cambio de valor en un componente de entrada de datos. Hemos visto que es de vital importancia para detectar si un componente controlado es mudo (*dumb*) o inteligente (*smart*). Si se indica, es inteligente; en otro caso, mudo.

Para facilitar la explicación, hemos sido un poco abstractos. Es importante tener claro que el evento Change, cuando se usa un cuadro de texto, se genera para cada carácter introducido por el usuario. No se genera una única vez cuando el usuario ha introducido todo el contenido y sale del cuadro de texto. Sino que lo hace para cada carácter introducido.

Por suerte, el evento Change se puede indicar a nivel de formulario. Hacerlo así evita tener que definirlo en cada uno de los componentes de entrada de datos que contenga. Y a su vez, hace el código más elegante. Pero lleva implícito una cosa: *todos* los componentes de entrada de datos que indiquen el atributo value serán inteligentes.

Y en estos casos, ¿cómo se puede definir un componente mudo? Sencillo, basta con añadirle el atributo readOnly. De esta manera, nunca propagará el evento Change y React lo considerará mudo.

He aquí un ejemplo de formulario con todos sus componentes como inteligentes, gestionándose sus cambios mediante el controlador handleChange() definido a nivel de formulario:

```
/**
 * Un formulario de Login.
 */
class LoginForm extends React.Component {
  /**
   * Constructor.
   *
   * @param props:object  The element-passed properties.
   */
  constructor(props) {
    super(props);

    this.state = {
      username: "",
      password: ""
    };
  }

  /**
   * Display name.
   *
   * @type string
   */
  static get displayName() {
    return "LoginForm";
  }

  /**
   * Handle the Submit event.
   *
   * @param evt:SyntheticEvent  The event object.
   */
  handleSubmit(evt) {
    evt.preventDefault();
    alert(`User: ${this.state.username}\nPassword: ${this.state.password}`);
  }

  /**
```



```
     * Handle the Change event of the input components.
     *
     * @param evt:SyntheticEvent  The event object.
     */
    handleChange(evt) {
      const target = evt.target;

      this.setState({
        [target.name]: (target.type == "checkbox" ? target.checked : target.value)
      });
    }

    /**
     * Return the component render
     *.
     * @override
     */
    render() {
      return (
        <form onSubmit={(evt) => this.handleSubmit(evt)}
              onChange={(evt) => this.handleChange(evt)}>
          <input type="text"
                 name="username"
                 value={this.state.username}
                 placeholder="Your username"
                 required />
          <input type="password"
                 name="password"
                 value={this.state.password}
                 placeholder="Your password"
                 required />
          <input type="submit" name="submit" value="Sign in" />
        </form>
      );
    }
  }
```

Componentes no controlados

Un componente no controlado (*uncontrolled component*) es el otro tipo de elemento HTML de entrada de datos. Este tipo se asemeja más a lo que conocemos de HTML. Reaccionan activamente a los cambios del usuario, modificando el atributo value del elemento automáticamente. Sin tanta parafernalia como los controlados inteligentes.

Recordemos que un componente controlado mudo se define cuando se define la propiedad value, pero no así onChange. Y los componentes controlados inteligentes requieren tanto de value como de onChange.

Por su parte, un componente no controlado *no* debe definir la propiedad value. Pero ojo, existe, contiene el valor actual del elemento. Y ¿cómo se indica su valor inicial? Fácil, mediante la propiedad defaultValue. Si lo deseamos, podemos definir controlador para el evento Change, mediante el atributo onChange. Generalmente, se suele hacer cuando necesitamos conocer que el usuario ha cambiado el valor. Pero recordemos que el valor lo cambia implícitamente el componente, no debemos hacerlo nosotros mismos explícitamente como ocurre con los componentes inteligentes.

He aquí unos ejemplos ilustrativos de componentes no controlados:

```
<input type="text" name="username" />
<input type="password" name="password" />
```

Ahora, es muy importante tener claro dónde se almacena los datos introducidos por el usuario. No se hace en el estado, como ocurre con los componentes inteligentes. Ahora, sencillamente se dejan en el propio componente. Cuando deseemos acceder al valor actual del elemento username, deberemos acceder al componente y, entonces, acceder a su atributo value.

Referencias

Si es necesario acceder a un elemento HTML de un componente, podemos hacerlo en tiempo de

ejecución mediante el elemento en cuestión del DOM. Para facilitar el acceso a los elementos presentados por el componente en el método render(), React proporciona el atributo ref, un identificador único asociado a uno de los elementos del componente, para facilitar su acceso.

Veamos un ejemplo ilustrativo:

```
<input type="text" name="username" ref="username" />
```

Una vez tenemos definido el elemento con este atributo, lo siguiente es accederlo mediante la propiedad this.refs del componente. Todo elemento definido con un ref, tendrá una propiedad homónima en this.refs. Así pues, si deseamos acceder al elemento del formulario cuyo ref es username, bastará con:

```
this.refs.username
```

Pero ojo, se accede al componente. Si deseamos acceder al valor almacenado, por ejemplo, en un cuadro de texto, tendremos que usar:

```
this.refs.username.value
```

Vamos a hacer un pequeño resumen de situación.

Los componentes controlados se clasifican en mudos o inteligentes. Los mudos nunca cambian su valor, el cual generalmente se almacena en el objeto props del formulario. En cambio, los inteligentes son de L/E y mantienen los datos recolectados del usuario en el estado, por lo que se fijan a partir del objeto state. Es muy raro tener la necesidad de acceder al atributo value de los elementos controlados. Se suele utilizar props o state, según el caso.

Los componentes no controlados no tienen un soporte especial. Se implementan como los propios elementos HTML que representan. Y por esta razón, si tenemos que conocer el valor actual de uno de ellos, tendremos que consultar su atributo value. Para ayudar a acceder a un componente no controlado, se le suele asignar una referencia, un identificador único en el atributo ref. Lo que hará que React le cree una propiedad con el mismo nombre en la propiedad refs del componente padre que, generalmente, es el componente formulario.

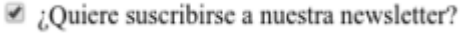
Componentes de entrada de datos

Algunos componentes de entrada requieren una atención especial, además de lo presentado hasta ahora.

Casillas de confirmación

Recordemos de HTML que una casilla de confirmación (*checkbox*) es un cuadro que puede seleccionarse, dando a entender que se asegura o corrobora lo indicado. Ejemplo:

```
<input type="checkbox" name="newsletter"> ¿Quiere suscribirse a nuestra newsletter?
```



Como sabemos de HTML, estos componentes se crean mediante el elemento <input type="checkbox">. Pero ahora, no se usa los atributos value o defaultValue, sino checked y defaultChecked.

Cuando el elemento lo definimos como controlado, se usa la propiedad checked para indicar si la casilla debe aparecer marcada. Ejemplo:

```
<input type="checkbox"
       name="important"
       checked={this.state.newsletter}
       onChange={(evt) => this.handleChange(evt)} />
```

En cambio, cuando se usa un componente no controlado, se usará defaultChecked para indicar la marca de la casilla durante la reproducción inicial del componente.

Botones de opción

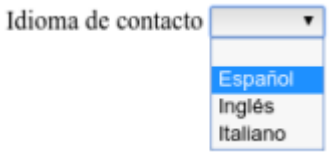
Un botón de opción (*radio button*) es similar a una caja de confirmación, pero pertenece a un grupo. De tal manera que si uno del grupo es seleccionado, los demás se deseleccionan. Siendo sólo posible que haya como máximo uno seleccionado. Ejemplo:

En HTML, se definen mediante elementos <input type="radio">. Por cada opción, se define su propio <input>. La agrupación se consigue mediante el atributo name, todos ellos deben tener el mismo valor. Por otra parte, para distinguir cada una de las opciones posibles, se indica un valor identificativo en el atributo value. Y cuál de ellas debe estar seleccionada, mediante el atributo checked. He aquí un ejemplo ilustrativo:

```
Idioma de contacto:
<input type="radio"
       name="lang"
       value="es"
       checked={this.state.lang == "es"}
       onChange={(evt) => this.handleChange(evt)} /> Español
<input type="radio"
       name="lang"
       value="en"
       checked={this.state.lang == "en"}
       onChange={(evt) => this.handleChange(evt)} /> Inglés
<input type="radio"
       name="lang"
       value="it"
       checked={this.state.lang == "it"}
       onChange={(evt) => this.handleChange(evt)} /> Italiano
```

Desplegables

Otro elemento que requiere especial atención en React es <select>. Consiste en un desplegable con una lista de opciones. Ejemplo:



Cada opción se indica mediante un elemento <option> donde su atributo <value> indica el valor a remitir cuando la opción se encuentra seleccionada. <select> indica el valor de la opción seleccionada por el usuario mediante el atributo selected. Para indicar si se puede seleccionar múltiples opciones se usa el atributo booleano multiple. Veamos un ejemplo ilustrativo:

```
Idioma de contacto
<select name="lang" selected={this.state.lang} onChange={(evt) => this.handleChange(evt)}>
  <option value=""></option>
  <option value="es">Español</option>
  <option value="en">Inglés</option>
  <option value="it">Italiano</option>
</select>
```

Botones

Los botones también se representan mediante elementos <input>, pero ahora con los tipos button, reset o submit. Mediante submit, se define un botón que generará el envío de los datos del formulario al servidor. Cada vez que el usuario lo pulsa, genera el evento Submit del formulario. Mientras que button se utiliza para cualquier otro botón. Y con reset, se suprimen los datos introducidos por el usuario en el formulario.

Cuando se pulsa, genera el evento Click.

Ejemplo:

```
<input type="submit" onSubmit={(evt) => this.handleSubmit(evt)} value="Enviar" />
<input type="button" onClick={(evt) => alert("Check!!!")} value="Comprobar" />
```

El atributo value contiene el texto que se mostrará al usuario en el botón.

Eventos de foco

Un evento de foco (*focus event*) es aquel que está relacionado con el foco de los elementos de entrada de datos de un formulario.

Evento	Descripción
Focus	El usuario ha entrado en un control de entrada.
Blur	El usuario ha abandonado un control de entrada.

El objeto de evento dispone de la siguiente propiedad: relatedTarget (DOMEventTarget).

Generador de Justo

El generador de Justo proporciona comandos específicos para trabajar con formularios. Con ellos, podemos crear componentes formularios y ubicarlos en la carpeta app/components más rápida y fácilmente, así como generar componentes <input>, <textarea> y <select>.

Comando form

Para crear un componente formulario, se puede utilizar el comando form del generador de Justo:

```
justo -g react form
```

Comando snippet input

Para obtener un elemento <input>, usar el comando snippet input:

```
justo -g react snippet input
```

Hay que tener claro:

- El tipo de <input>.
- El tipo de componente: controlado o no controlado.
- Si controlado, el subtipo: mudo o inteligente.

Una vez generado el código, éste se muestra en la consola. Y es posible que también se encuentre en el portapapeles listo para pegarlo en el componente formulario. Para que se copie en el portapapeles, es necesario que se encuentre instalado: en Windows, el comando clip; y en Linux, xclip.

Comando snippet textarea

Para generar un <textarea>, podemos usar el comando snippet textarea:

```
justo -g react snippet textarea
```

Comando snippet select

Para generar el código de un elemento <select>, se puede usar el comando snippet select:

```
justo -g react snippet select
```

Comando snippet option

Para generar un <option> de un <select>, usar snippet option:

```
justo -g react snippet option
```

