



Buenas Prácticas de Frontend

Especificación Versión 1.0

1. Guía de estilo para Javascript

Tabla de contenido

1. [Reglas básicas](#)
2. [Class vs React.createClass vs stateless](#)
3. [Nomenclatura](#)
4. [Declaraciones](#)
5. [Indentación](#)
6. [Comillas](#)
7. [Espaciado](#)
8. [Props](#)
9. [Refs](#)
10. [Paréntesis](#)
11. [Tags](#)
12. [Metodos](#)
13. [Orden](#)
14. [isMounted](#)

Reglas básicas

- Incluya sólo un componente React por archivo.
- Sin embargo, se admiten múltiples [Stateless, o Pure, Components] (<https://facebook.github.io/react/docs/reusable-components.html#stateless-functions>) por archivo. Eslint: [react / no-multi-comp] (<https://github.com/yannickcr/eslint-plugin-react/blob/master/docs/rules/no-multi-comp.md#ignorestateless>).
- Siempre use la sintaxis de JSX.
- No utilice React.createElement a menos que esté inicializando la aplicación desde un archivo que no sea JSX.
- **Class vs React.createClass vs stateless**
- Si el componente tiene estado y / o refs, elija class extends React.Component sobre React.createClass a menos que tenga una buena razón para usar mixins.
- eslint: [react/prefer-es6-class](#) [react/prefer-stateless-function](#)
- `// mal`
- `const Listing = React.createClass({`
- `// ...`
- `render() {`

- `return <div>{this.state.hello}</div>;`
- `}`
- `});`
- `// bien`
- `class Listing extends React.Component {`
- `// ...`
- `render() {`
- `return <div>{this.state.hello}</div>;`
- `}`
- `}`

Y si no tienes estado o refs, elija funciones normales y no arrow functions en las clases:

```
// mal
class Listing extends React.Component {
  render() {
    return <div>{this.props.hello}</div>;
  }
}
```

```
// mal (no esta bueno confiar en la inferencia del nombre de la funcion)
const Listing = ({ hello }) => (
  <div>{hello}</div>
);
```

```
// bien
function Listing({ hello }) {
  return <div>{hello}</div>;
}
```

Nomenclatura

- **Extensiones:** Use la extensión .jsx para componentes React.
- **Nombre de archivos:** Use PascalCase para el nombre de los archivos. Ejemplo, ReservationCard.jsx.
- **Nomenclatura de referencia:** Use PascalCase para componentes React y camelCase para sus instancias. eslint: [react/jsx-pascal-case](#)
- `// mal`
- `import reservationCard from './ReservationCard';`
- `// bien`
- `import ReservationCard from './ReservationCard';`
- `// mal`
- `const ReservationItem = <ReservationCard />;`
- `// bien`
- `const reservationItem = <ReservationCard />;`

- **Nomenclatura de los componentes:** Utilice el nombre de archivo como el nombre del componente. Por ejemplo, ReservationCard.jsx debe tener una referencia tal como ReservationCard. Sin embargo, para el componente raíz de un directorio, use index.jsx como el nombre del archivo y además use el nombre del directorio como el nombre del componente:
 - // mal
 - import Footer from './Footer/Footer';
 -
 - // mal
 - import Footer from './Footer/index';
 -
 - // bien
 - import Footer from './Footer';
- **Nomenclatura 'Higher-order Components':** Componga el nombre del high order component y el nombre del component como displayName en el nuevo componente generado. Por ejemplo, el componente de orden superior withFoo(), al pasar un componente Bar debería producir un componente con displayName de withFoo(Bar).

¿Por qué? El displayName de un componente puede ser utilizado por herramientas de desarrollo o en mensajes de error, y tener un valor que exprese claramente esta relación ayuda a las personas a entender lo que está sucediendo.

```
// mal
export default function withFoo(WrappedComponent) {
  return function WithFoo(props) {
    return <WrappedComponent {...props} foo />;
  }
}
```

```
// bien
export default function withFoo(WrappedComponent) {
  function WithFoo(props) {
    return <WrappedComponent {...props} foo />;
  }
}
```

```
const wrappedComponentName = WrappedComponent.displayName
  || WrappedComponent.name
  || 'Component';
```

```
WithFoo.displayName = `withFoo(${wrappedComponentName})`;
return WithFoo;
}
```

- **Props Naming:** Evite usar nombres de props de componentes de DOM para diferentes propósitos.

¿Por qué? La gente espera props como style y className signifiquen una cosa específica. La variación de esta API para una parte de la aplicación hace que el código sea menos legible y menos mantenible, y podría causar errores.

```
// mal
<MyComponent style="fancy" />
```

```
// bien
```

`<MyComponent variant="fancy" />`

Declaraciones

- No use `displayName` para nombrar componentes. En cambio, nombre los componentes por referencia.
- `// mal`
- `export default React.createClass({`
- `displayName: 'ReservationCard',`
- `// stuff goes here`
- `});`
- `// bien`
- `export default class ReservationCard extends React.Component {`
- `}`

Indentación

- Siga estos estilos de indentación para la sintaxis de JSX.
- eslint: [react/jsx-closing-bracket-location](#)
- `// mal`
- `<Foo superLongParam="bar"`
- `anotherSuperLongParam="baz" />`
- `// bien`
- `<Foo`
- `superLongParam="bar"`
- `anotherSuperLongParam="baz"`
- `/>`
- `// Si existe solo una props, puede mantenerse en una sola línea.`
- `<Foo bar="bar" />`
- `// Al existir más de una props se debe separar por línea`
- `<Foo`
- `superLongParam="bar"`
- `anotherSuperLongParam="baz"`
- `>`
- `<Quux />`
- `</Foo>`

Comillas

- Siempre use comillas dobles (") para los atributos JSX, pero comillas simples (') para otros JS. eslint: [jsx-quotes](#)

¿Por qué? Los atributos HTML suelen usar comillas dobles en lugar de simples, por lo que los atributos JSX reflejan esta convención.

```
// mal
<Foo bar='bar' />
```

```
// bien
<Foo bar="bar" />
```

```
// mal
<Foo style={{ left: "20px" }} />
```

```
// bien
<Foo style={{ left: '20px' }} />
```

Espaciado

- Siempre incluir un espacio en los self-closing tags. eslint: [no-multi-spaces](#), [react/jsx-space-before-closing](#)
- // mal
- <Foo/>
-
- // muy mal
- <Foo />
-
- // mal
- <Foo
- />
-
- // bien
- <Foo />
-
- No use llaves en JSX con espacios en el medio. eslint: [react/jsx-curly-spacing](#)
- // mal
- <Foo bar={ baz } />
-
- // bien
- <Foo bar={baz} />

Props

- Siempre use camelCase para los nombres de las props.
- // mal
- <Foo
- UserName="hello"
- phone_number={12345678}
- />
-
- // bien

- `<Foo`
- `userName="hello"`
- `phoneNumber={12345678}`
- `>`
- Omita el valor de la prop cuando esta sea explícitamente true.
eslint: [react/jsx-boolean-value](#)
- `// mal`
- `<Foo`
- `hidden={true}`
- `>`
- `// bien`
- `<Foo`
- `hidden`
- `>`
- Siempre incluir alt property en `` tags. Si la imagen es de presentación, alt puede ser un string vacío o la `` debe tener `role="presentation"`. eslint: [jsx-a11y/img-has-alt](#)
- `// mal`
- ``
- `// bien`
- ``
- `// bien`
- ``
- `// bien`
- ``
- No utilice palabras como "imagen", "foto" o "imagen" en los soportes `` alteslint: [jsx-a11y/img-redundant-alt](#)

¿Por qué? Los lectores de pantalla ya anuncian elementos `img` como imágenes, por lo que no es necesario incluir esta información en el texto alternativo.

`// mal`
``

`// bien`
``

- Use solamente válidos y no abstractos [ARIA roles](#). eslint: [jsx-a11y/aria-role](#)
- `// mal - not an ARIA role`
- `<div role="datepicker" />`
- `// mal - abstract ARIA role`
- `<div role="range" />`
- `// bien`
- `<div role="button" />`
- No utilice `accessKey` en elementos. eslint: [jsx-a11y/no-access-key](#)

¿Por qué? Ciertas inconsistencias entre shortcuts de teclado y comandos de teclado utilizados por personas que usan lectores de pantalla y teclados complican la accesibilidad.

```
// mal
<div accessKey="h" />
```

```
// bien
<div />
```

- Evite utilizar key como prop del índice del array, elija un unique ID. ([why?](#))

```
// mal
{todos.map((todo, index) =>
  <Todo
    {...todo}
    key={index}
  />
)}
```

```
// bien
{todos.map(todo => (
  <Todo
    {...todo}
    key={todo.id}
  />
))}
```

- Siempre defina DefaultProps de forma explícita para todos los props no requeridos.

¿Por qué? Los PropTypes son una forma de documentación, y proporcionar DefaultProps significa que el lector de su código no tiene tanto que asumir. Además, puede significar que su código puede omitir ciertas comprobaciones de tipo.

```
// mal
function SFC({ foo, bar, children }) {
  return <div>{foo}{bar}{children}</div>;
}
SFC.propTypes = {
  foo: PropTypes.number.isRequired,
  bar: PropTypes.string,
  children: PropTypes.node,
};
```

```
// bien
function SFC({ foo, bar }) {
  return <div>{foo}{bar}</div>;
}
SFC.propTypes = {
  foo: PropTypes.number.isRequired,
  bar: PropTypes.string,
};
SFC.defaultProps = {
```



```
bar: ",  
children: null,  
};
```

Refs

- Siempre use use ref callbacks. eslint: [react/no-string-refs](#)
- // mal
- <Foo
- ref="myRef"
- />
-
- // bien
- <Foo
- ref={(ref) => { this.myRef = ref; }}
- />

Parentesis

- Cerrar JSX tags con paréntesis cuando se tenga más de una línea a retornar. eslint: [react/wrap-multilines](#)
- // mal
- render() {
- return <MyComponent className="long body" foo="bar">
- <MyChild />
- </MyComponent>;
- }
-
- // bien
- render() {
- return (
- <MyComponent className="long body" foo="bar">
- <MyChild />
- </MyComponent>
-);
- }
-
- // bien, en caso de que solo ocupe una sola línea.
- render() {
- const body = <div>hello</div>;
- return <MyComponent>{body}</MyComponent>;
- }

Tags

- Siempre autocierre tags que no tienen children (o hijos). eslint: [react/self-closing-comp](#)
- // mal
- <Foo className="stuff"></Foo>
-
- // bien
- <Foo className="stuff" />

- Si el componente tiene varias propiedades, cierre su tag en una nueva línea. eslint: [react/jsx-closing-bracket-location](#)
- // mal
- <Foo
- bar="bar"
- baz="baz" />
-
- // bien
- <Foo
- bar="bar"
- baz="baz"
- />

Métodos

- Use arrow functions.
-
- function ItemList(props) {
- return (
-
- {props.items.map((item, index) => (
- <Item
- key={item.key}
- onClick={() => doSomethingWith(item.name, index)}
- }/>
-))}
-
-);
- }
-
- Bindear eventos por cada render method en el constructor. eslint: [react/jsx-no-bind](#)

¿Por qué? Una llamada de bind en render crea una función nueva en cada render.

```
// mal
class extends React.Component {
  onClickDiv() {
    // do stuff
  }

  render() {
    return <div onClick={this.onClickDiv.bind(this)} />
  }
}
```

```
// bien
class extends React.Component {
  constructor(props) {
    super(props);

    this.onClickDiv = this.onClickDiv.bind(this);
  }

  onClickDiv() {
```

```
// do stuff
}  
  
render() {  
  return <div onClick={this.onClickDiv} />  
}  
}
```

- No utilice prefijo de _ (underscore) para los métodos internos de un componente React.

¿Por qué? Los prefijos _ (underscore) a veces se usan como una convención en otros idiomas para denotar privacidad. Pero, a diferencia de esos lenguajes, no hay soporte nativo para el ámbito en JavaScript ya que todo es público. Independientemente de sus intenciones, agregar prefijos a sus propiedades en realidad no los hacen privados, y cualquier property debe ser tratado como público. Ver errores en [#1024](#), y [#490](#).

```
// mal  
React.createClass({  
  _onClickSubmit() {  
    // do stuff  
  },  
  
  // other stuff  
});  
  
// bien  
class extends React.Component {  
  onClickSubmit() {  
    // do stuff  
  }  
  
  // other stuff  
}
```

- Asegúrese de devolver un valor en sus métodos render.
eslint: [react/require-render-return](#)
- // mal
- render() {
- (<div />);
- }
-
- // bien
- render() {
- return (<div />);
- }

Orden

- Orden de class extends React.Component:
 1. opcional static metodos

2. constructor
3. getChildContext
4. componentWillMount
5. componentDidMount
6. componentWillReceiveProps
7. shouldComponentUpdate
8. componentWillUpdate
9. componentDidUpdate
10. componentWillUnmount
11. *clickHandlers or eventHandlers* like onClickSubmit() or onChangeDescription()
12. *getter methods for render* like getSelectReason() or getFooterContent()
13. *optional render* like renderNavigation() or renderProfilePicture()
14. render

- Como definir propTypes, defaultProps, contextTypes, etc...
- import React, { PropTypes } from 'react';
-
- const propTypes = {
- id: PropTypes.number.isRequired,
- url: PropTypes.string.isRequired,
- text: PropTypes.string,
- };
-
- const defaultProps = {
- text: 'Hello World',
- };
-
- class Link extends React.Component {
- static methodsAreOk() {
- return true;
- }
-
- render() {
- return {this.props.text}
- }
- }
-
- Link.propTypes = propTypes;
- Link.defaultProps = defaultProps;
-
- export default Link;
-
- Orden de metodos en React.createClass: eslint: [react/sort-comp](#)

1. displayName
2. propTypes
3. contextTypes
4. childContextTypes
5. mixins
6. statics
7. defaultProps
8. getDefaultProps
9. getInitialState

10. getChildContext
11. componentWillMount
12. componentDidMount
13. componentWillReceiveProps
14. shouldComponentUpdate
15. componentWillUpdate
16. componentDidUpdate
17. componentWillUnmount
18. *clickHandlers or eventHandlers* like onClickSubmit() or onChangeDescription()
19. *getter methods for render* like getSelectReason() or getFooterContent()
20. *optional render methods* like renderNavigation() or renderProfilePicture()
21. render

isMounted

- text: 'Hello World',
- };
-
- class Link extends React.Component {
- static methodsAreOk() {
- return true;
- }
-
- render() {
- return {this.props.text}
- }
- }
-
- Link.propTypes = propTypes;
- Link.defaultProps = defaultProps;
-
- export default Link;
- Orden de metodos en React.createClass: eslint: [react/sort-comp](#)

1. displayName
2. propTypes
3. contextTypes
4. childContextTypes
5. mixins
6. statics
7. defaultProps
8. getDefaultProps
9. getInitialState
10. getChildContext
11. componentWillMount
12. componentDidMount
13. componentWillReceiveProps
14. shouldComponentUpdate
15. componentWillUpdate
16. componentDidUpdate
17. componentWillUnmount
18. *clickHandlers or eventHandlers* like onClickSubmit() or onChangeDescription()
19. *getter methods for render* like getSelectReason() or getFooterContent()

- 20. *optional render methods* like `renderNavigation()` or `renderProfilePicture()`
- 21. `render`

isMounted

- No use `isMounted`. eslint: [react/no-is-mounted](#)

¿Por qué? [isMounted is an anti-pattern](#), no está disponible cuando se usan clases ES6, y está en camino de ser oficialmente obsoleto.