# Contributing

Thank you for your interest in contributing to the React Docs!

## Code of Conduct

Facebook has adopted a Code of Conduct that we expect project

participants to adhere to. Please [read the full text](https://code.facebook.com/codeofconduct)

so that you can understand what actions will and will not be tolerated.

## Technical Writing Tips

This is a [good summary](https://medium.com/@kvosswinkel/coding-like-a-journalist-ee52360a16bc) for things to keep in mind when writing technical docs.

## Guidelines for Text

\*\*Different sections intentionally have different styles.\*\*

The documentation is divided into sections to cater to different learning styles and use cases. When editing an article, try to match the surrounding text in tone and style. When creating a new article, try to match the tone of the other articles in the same section. Learn about the motivation behind each section below.

\*\*[Installation](https://reactjs.org/docs/getting-started.html)\*\* gives an overview of the docs, and demonstrates two different ways to use it: either as a simple `<script>` tag on the page, or as part of a complex JavaScript toolchain.

\*\*[Tutorial](https://reactjs.org/tutorial/tutorial.html)\*\* is relatively informal, and is designed for people who prefer a hands-on approach to learning, and can tolerate skipping theory in favor of practice. Its goal is to give the reader a feel for a typical React workflow rather than to explain fundamentals in detail. Here we focus on \*what\* to do and spend less time on \*how\* or \*why\* we did it. It is extremely important to do a lot of hand-holding and unambiguously describe every single change. It should be possible for a beginner to mechanically follow every instruction, and still get to a working tic-tac-toe game.

\*\*[Main Concepts](https://reactjs.org/docs/hello-world.html)\*\* is designed to introduce fundamental concepts in a step-by-step way. Each individual article in Main Concepts builds on the knowledge from the previous ones, so make sure not to add any "cyclical dependencies" between them. It is important that the reader can start with the first article and work their way to the last Main Concepts article without ever having to "look ahead" for a definition. This explains some ordering choices (e.g. that state is explained before events, or that "thinking in React" doesn't use refs). Main Concepts also serves as a reference manual for React concepts, so it is important to be very strict about their definitions and relationships between them. This is, for example, why we introduce elements before components. Resist adding too much detail to Main Concepts articles. They intentionally don't cover all corner cases, and focus on establishing firm foundations.

\*\*[Advanced Guides](https://reactjs.org/docs/jsx-in-depth.html)\*\* are deep dives into topics that aren't essential for a beginner developer but that everyone bumps into sooner or later. They don't have a specific order, and target more experienced developers. If you have a set of recipes fitting a particular use case, and those recipes aren't opinionated (most React users would agree on them), this is the place to add them.

\*\*[Reference](https://reactjs.org/docs/react-api.html)\*\* is organized by APIs rather than concepts. It is intended to be exhaustive. Any corner cases or recommendations that were skipped for brevity in Main Concepts or Advanced Guides should be mentioned in the reference documentation for the corresponding APIs.

\*\*[Contributing](https://reactjs.org/docs/how-to-contribute.html)\*\* should stay up-to-date and be friendly to relatively experienced developers.

\*\*[FAQ](https://reactjs.org/docs/faq-ajax.html)\*\* has a more conversational tone than the other sections. Here, it's fine to include some content that's not primarily concerned with React, as long as React users are overwhelmingly interested in it (e.g. recommendations on directory structure). It's also okay to show more than a single way to do something in the FAQ, and briefly discuss the tradeoffs. The FAQ prioritizes unblocking people with a working solution over explaining concepts in detail, and can be more opinionated than the rest of the docs, even providing popular library recommendations.

\*\*Try to follow your own instructions.\*\*

When writing step-by-step instructions (e.g. how to install something), try to forget everything you know about the topic, and actually follow the instructions you wrote, a single step at time. Often you will discover that there is implicit knowledge that you forgot to mention, or that there are missing or out-of-order steps in the instructions. Bonus points for getting \*somebody else\* to follow the steps and watching what they struggle with. Often it would be something very simple that you have not anticipated.

## Guidelines for Code Examples

### Syntax

#### Prefer JSX to `createElement`.

Ignore this if you're specifically describing `createElement`.

#### Use `const` where possible, otherwise `let`. Don't use `var`.

Ignore this if you're specifically writing about ES5.

#### Don't use ES6 features when equivalent ES5 features have no downsides.

Remember that ES6 is still new to a lot of people. While we use it in many places (`const` / `let`, classes, arrow functions), if the equivalent ES5 code is just as straightforward and readable, consider using it.

In particular, you should prefer named `function` declarations over `const myFunction = () => ...` arrows for top-level functions. However, you \*should\* use arrow functions where they provide a tangible improvement (such as preserving `this` context inside a component). Consider both sides of the tradeoff when deciding whether to use a new feature.

#### Don't use features that aren't standardized yet.

For example, \*\*don't\*\* write this:

```js

class MyComponent extends React.Component {

state = {value: ''};

handleChange = (e) => {

this.setState({value: e.target.value});

};

}

```

Instead, \*\*do\*\* write this:

```js

class MyComponent extends React.Component {

constructor(props) {

super(props);

this.handleChange = this.handleChange.bind(this);

this.state = {value: ''};

}

handleChange(e) {

this.setState({value: e.target.value});

}

}

```

Ignore this rule if you're specifically describing an experimental proposal. Make sure to mention its experimental nature in the code and in the surrounding text.

### Style

- Use semicolons.

- No space between function names and parens (`method() {}` not `method () {}`).

- When in doubt, use the default style favored by [Prettier](https://prettier.io/playground/).

### Highlighting

Use `js` as the highlighting language in Markdown code blocks:

````

```js

// code

```

````

Sometimes you'll see blocks with numbers.

They tell the website to highlight specific lines.

You can highlight a single line:

````

```js{2}

function hello() {

// this line will get highlighted

}

```

````

A range of lines:

````

```js{2-4}

function hello() {

// these lines

// will get

// highlighted

}

```

````

Or even multiple ranges:

````

```js{2-4,6}

function hello() {

// these lines

// will get

// highlighted

console.log('hello');

// also this one

console.log('there');

}

```

````

Be mindful that if you move some code in an example with highlighting, you also need to update the highlighting.

Don't be afraid to often use highlighting! It is very valuable when you need to focus the reader's attention on a particular detail that's easy to miss.