Getting Started

Edit on GitHub

JSFiddle

The easiest way to start hacking on React is using the following JSFiddle Hello World examples:

- **React JSFiddle**
- React JSFiddle without JSX

Starter Pack

If you're just getting started, you can download the starter kit. The starter kit includes prebuilt copies of React and React DOM for the browser, as well as a collection of usage examples to help you get started.

Download Starter Kit 15.2.1

In the root directory of the starter kit, create a `helloworld.html` with the following contents.

Code

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Hello React!</title>
    <script src="build/react.js"></script>
    <script src="build/react-dom.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.34/
browser.min.js"></script>
  </head>
  <body>
    <div id="example"></div>
    <script type="text/babel">
      ReactDOM.render(
        <h1>Hello, world!</h1>,
        document.getElementById('example')
      );
    </script>
  </body>
</html>
```

The XML syntax inside of JavaScript is called JSX; check out the JSX syntax to learn more about it. In order to translate it to vanilla JavaScript we use `<script type="text/babel">`

and include Babel to actually perform the transformation in the browser. Open the html from a browser and you should already be able to see the greeting!

Separate File

Your React JSX code can live in a separate file. Create the following `src/helloworld.js`.

Code

```
ReactDOM.render(
  <h1>Hello, world!</h1>,
  document.getElementById('example')
);
```

Then reference it from `helloworld.html`:

Code

```
<script type="text/babel" src="src/helloworld.js"></script>
```

Note that some browsers (Chrome, e.g.) will fail to load the file unless it's served via HTTP.

Using React with npm or Bower

You can also use React with package managers like npm or Bower. You can learn more in our Package Managers section.

Next Steps

Check out the tutorial and the other examples in the starter kit's `examples` directory to learn more.

Good luck, and welcome!

Next →

Tutorial

Edit on GitHub

We'll be building a simple but realistic comments box that you can drop into a blog, a basic version of the realtime comments offered by Disqus, LiveFyre or Facebook comments.

We'll provide:

- A view of all of the comments
- A form to submit a comment
- Hooks for you to provide a custom backend

It'll also have a few neat features:

- **Optimistic commenting:** comments appear in the list before they're saved on the server so it feels fast.
- **Live updates:** other users' comments are popped into the comment view in real time.
- **Markdown formatting:** users can use Markdown to format their text.

Want to skip all this and just see the source?

It's all on GitHub.

Running a server

In order to start this tutorial, we're going to require a running server. This will serve purely as an API endpoint which we'll use for getting and saving data. In order to make this as easy as possible, we've created a simple server in a number of scripting languages that does exactly what we need it to do. **You can view the source or download a zip file containing everything needed to get started.**

For sake of simplicity, the server we will run uses a `JSON` file as a database. You would not run this in production but it makes it easy to simulate what you might do when consuming an API. Once you start the server, it will support our API endpoint and it will also serve the static pages we need.

Getting started

For this tutorial, we're going to make it as easy as possible. Included in the server package discussed above is an HTML file which we'll work in. Open up `public/index.html` in your favorite editor. It should look something like this:

Code

```html
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>React Tutorial</title>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react.js"
></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react-
dom.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.34/
browser.min.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/2.2.0/
jquery.min.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/remarkable/1.6.2/
remarkable.min.js"></script>
  </head>
  <body>
    <div id="content"></div>
    <script type="text/babel" src="scripts/example.js"></script>
    <script type="text/babel">
      // To get started with this tutorial running your own code, simply remove
      // the script tag loading scripts/example.js and start writing code here.
    </script>
  </body>
</html>
```

For the remainder of this tutorial, we'll be writing our JavaScript code in this script tag. We don't have any advanced live-reloading so you'll need to refresh your browser to see updates after saving. Follow your progress by opening `http://localhost:3000` in your browser (after starting the server). When you load this for the first time without any changes, you'll see the finished product of what we're going to build. When you're ready to start working, just delete the preceding `<script>` tag and then you can continue.

Note:

We included jQuery here because we want to simplify the code of our future ajax calls, but it's **NOT** mandatory for React to work.

Your first component

React is all about modular, composable components. For our comment box example, we'll have the following component structure:

Code

```
- CommentBox
  - CommentList
    - Comment
  - CommentForm
```

Let's build the `CommentBox` component, which is just a simple `<div>`:

Code

```
// tutorial1.js
var CommentBox = React.createClass({
  render: function() {
    return (
      <div className="commentBox">
        Hello, world! I am a CommentBox.
      </div>
    );
  }
});
ReactDOM.render(
  <CommentBox />,
  document.getElementById('content')
);
```

Note that native HTML element names start with a lowercase letter, while custom React class names begin with an uppercase letter.

JSX Syntax

The first thing you'll notice is the XML-ish syntax in your JavaScript. We have a simple precompiler that translates the syntactic sugar to this plain JavaScript:

Code

```
// tutorial1-raw.js
var CommentBox = React.createClass({displayName: 'CommentBox',
  render: function() {
    return (
      React.createElement('div', {className: "commentBox"},
        "Hello, world! I am a CommentBox."
      )
    );
  }
});
ReactDOM.render(
  React.createElement(CommentBox, null),
  document.getElementById('content')
);
```

Its use is optional but we've found JSX syntax easier to use than plain JavaScript. Read more on the JSX Syntax article.

What's going on

We pass some methods in a JavaScript object to `React.createClass()` to create a new React component. The most important of these methods is called `render` which returns a tree of React components that will eventually render to HTML.

The `<div>` tags are not actual DOM nodes; they are instantiations of React `div` components. You can think of these as markers or pieces of data that React knows how to handle. React is **safe**. We are not generating HTML strings so XSS protection is the default.

You do not have to return basic HTML. You can return a tree of components that you (or someone else) built. This is what makes React **composable**: a key tenet of maintainable frontends.

`ReactDOM.render()` instantiates the root component, starts the framework, and injects the markup into a raw DOM element, provided as the second argument.

The `ReactDOM` module exposes DOM-specific methods, while `React` has the core tools shared by React on different platforms (e.g., React Native).

It is important that `ReactDOM.render` remain at the bottom of the script for this tutorial. `ReactDOM.render` should only be called after the composite components have been defined.

Composing components

Let's build skeletons for `CommentList` and `CommentForm` which will, again, be simple `<div>`s. Add these two components to your file, keeping the existing `CommentBox` declaration and `ReactDOM.render` call:

Code

```
var CommentList = React.createClass({
  render: function() {
    return (
      <div className="commentList">
        Hello, world! I am a CommentList.
      </div>
    );
  }
});

var CommentForm = React.createClass({
  render: function() {
    return (
      <div className="commentForm">
        Hello, world! I am a CommentForm.
      </div>
    );
  }
});
```

Next, update the `CommentBox` component to use these new components:

Code

```
// tutorial3.js
var CommentBox = React.createClass({
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList />
        <CommentForm />
      </div>
    );
  }
});
```

Notice how we're mixing HTML tags and components we've built. HTML components are regular React components, just like the ones you define, with one difference. The JSX compiler will automatically rewrite HTML tags to `React.createElement(tagName)` expressions and leave everything else alone. This is to prevent the pollution of the global namespace.

Using props

Let's create the `Comment` component, which will depend on data passed in from its parent. Data passed in from a parent component is available as a 'property' on the child component. These 'properties' are accessed through `this.props`. Using props, we will be able to read the data passed to the `Comment` from the `CommentList`, and render some markup:

Code

```
// tutorial4.js
var Comment = React.createClass({
  render: function() {
    return (
      <div className="comment">
        <h2 className="commentAuthor">
          {this.props.author}
        </h2>
        {this.props.children}
      </div>
    );
  }
});
```

By surrounding a JavaScript expression in braces inside JSX (as either an attribute or child), you can drop text or React components into the tree. We access named attributes passed to the component as keys on `this.props` and any nested elements as `this.props.children`.

Component Properties

Now that we have defined the `Comment` component, we will want to pass it the author name and comment text. This allows us to reuse the same code for each unique comment. Now let's add some comments within our `CommentList`:

Code

```
// tutorial5.js
var CommentList = React.createClass({
  render: function() {
    return (
      <div className="commentList">
        <Comment author="Pete Hunt">This is one comment</Comment>
        <Comment author="Jordan Walke">This is *another* comment</Comment>
      </div>
    );
  }
});
```

Note that we have passed some data from the parent `CommentList` component to the child `Comment` components. For example, we passed *Pete Hunt* (via an attribute) and *This is one comment* (via an XML-like child node) to the first `Comment`. As noted above, the `Comment` component will access these 'properties' through `this.props.author`, and `this.props.children`.

Adding Markdown

Markdown is a simple way to format your text inline. For example, surrounding text with asterisks will make it emphasized.

In this tutorial we use a third-party library **remarkable** which takes Markdown text and converts it to raw HTML. We already included this library with the original markup for the page, so we can just start using it. Let's convert the comment text to Markdown and output it:

Code

```
// tutorial6.js
var Comment = React.createClass({
  render: function() {
    var md = new Remarkable();
    return (
      <div className="comment">
        <h2 className="commentAuthor">
          {this.props.author}
        </h2>
        {md.render(this.props.children.toString())}
      </div>
    );
  }
});
```

All we're doing here is calling the remarkable library. We need to convert `this.props.children` from React's wrapped text to a raw string that remarkable will understand so we explicitly call `toString()`.

But there's a problem! Our rendered comments look like this in the browser: "`<p>`This is `<em>`another`</em>` comment`</p>`". We want those tags to actually render as HTML.

That's React protecting you from an XSS attack. There's a way to get around it but the framework warns you not to use it:

Code
```
var Comment = React.createClass({
  rawMarkup: function() {
    var md = new Remarkable();
    var rawMarkup = md.render(this.props.children.toString());
    return { __html: rawMarkup };
  },

  render: function() {
    return (
      <div className="comment">
        <h2 className="commentAuthor">
          {this.props.author}
        </h2>
        <span dangerouslySetInnerHTML={this.rawMarkup()} />
      </div>
    );
  }
});
```

This is a special API that intentionally makes it difficult to insert raw HTML, but for remarkable we'll take advantage of this backdoor.

Remember: by using this feature you're relying on remarkable to be secure. In this case, remarkable automatically strips HTML markup and insecure links from the output.

Hook up the data model

So far we've been inserting the comments directly in the source code. Instead, let's render a blob of JSON data into the comment list. Eventually this will come from the server, but for now, write it in your source:

Code
```
// tutorial8.js
var data = [
  {id: 1, author: "Pete Hunt", text: "This is one comment"},
  {id: 2, author: "Jordan Walke", text: "This is *another* comment"}
];
```

We need to get this data into `CommentList` in a modular way. Modify `CommentBox` and the `ReactDOM.render()` call to pass this data into the `CommentList` via props:

Code

```
var CommentBox = React.createClass({
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList data={this.props.data} />
        <CommentForm />
      </div>
    );
  }
});

ReactDOM.render(
  <CommentBox data={data} />,
  document.getElementById('content')
);
```

Now that the data is available in the `CommentList`, let's render the comments dynamically:

Code

```
// tutorial10.js
var CommentList = React.createClass({
  render: function() {
    var commentNodes = this.props.data.map(function(comment) {
      return (
        <Comment author={comment.author} key={comment.id}>
          {comment.text}
        </Comment>
      );
    });
    return (
      <div className="commentList">
        {commentNodes}
      </div>
    );
  }
});
```

That's it!

Fetching from the server

Let's replace the hard-coded data with some dynamic data from the server. We will remove the data prop and replace it with a URL to fetch:

Code

```
ReactDOM.render(
  <CommentBox url="/api/comments" />,
  document.getElementById('content')
);
```

This component is different from the prior components because it will have to re-render itself. The component won't have any data until the request from the server comes back, at which point the component may need to render some new comments.

Note: the code will not be working at this step.

Reactive state

So far, based on its props, each component has rendered itself once. `props` are immutable: they are passed from the parent and are "owned" by the parent. To implement interactions, we introduce mutable **state** to the component. `this.state` is private to the component and can be changed by calling `this.setState()`. When the state updates, the component re-renders itself.

`render()` methods are written declaratively as functions of `this.props` and `this.state`. The framework guarantees the UI is always consistent with the inputs.

When the server fetches data, we will be changing the comment data we have. Let's add an array of comment data to the `CommentBox` component as its state:

Code

```
// tutorial12.js
var CommentBox = React.createClass({
  getInitialState: function() {
    return {data: []};
  },
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList data={this.state.data} />
        <CommentForm />
      </div>
    );
  }
});
```

`getInitialState()` executes exactly once during the lifecycle of the component and sets up the initial state of the component.

Updating state

When the component is first created, we want to GET some JSON from the server and update the state to reflect the latest data. We're going to use jQuery to make an asynchronous request to the server we started earlier to fetch the data we need. The data is already included in the server you started (based on the `comments.json` file), so once it's fetched, `this.state.data` will look something like this:

Code

```
[
  {"id": "1", "author": "Pete Hunt", "text": "This is one comment"},
  {"id": "2", "author": "Jordan Walke", "text": "This is *another* comment"}
]
```

Code

```
// tutorial13.js
var CommentBox = React.createClass({
  getInitialState: function() {
    return {data: []};
  },
  componentDidMount: function() {
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      cache: false,
      success: function(data) {
        this.setState({data: data});
      }.bind(this),
      error: function(xhr, status, err) {
        console.error(this.props.url, status, err.toString());
      }.bind(this)
    });
  },
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList data={this.state.data} />
        <CommentForm />
      </div>
    );
  }
});
```

Here, `componentDidMount` is a method called automatically by React after a component is rendered for the first time. The key to dynamic updates is the call to `this.setState()`. We replace the old array of comments with the new one from the server and the UI automatically updates itself. Because of this reactivity, it is only a minor change to add live updates. We will use simple polling here but you could easily use WebSockets or other technologies.

Code

```
// tutorial14.js
var CommentBox = React.createClass({
  loadCommentsFromServer: function() {
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      cache: false,
      success: function(data) {
        this.setState({data: data});
      }.bind(this),
      error: function(xhr, status, err) {
        console.error(this.props.url, status, err.toString());
      }.bind(this)
    });
  },
  getInitialState: function() {
    return {data: []};
  },
  componentDidMount: function() {
    this.loadCommentsFromServer();
    setInterval(this.loadCommentsFromServer, this.props.pollInterval);
  },
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList data={this.state.data} />
        <CommentForm />
      </div>
    );
  }
});

ReactDOM.render(
  <CommentBox url="/api/comments" pollInterval={2000} />,
  document.getElementById('content')
);
```

All we have done here is move the AJAX call to a separate method and call it when the component is first loaded and every 2 seconds after that. Try running this in your browser and changing the `comments.json` file (in the same directory as your server); within 2 seconds, the changes will show!

Adding new comments

Now it's time to build the form. Our `CommentForm` component should ask the user for their name and comment text and send a request to the server to save the comment.

Code

```
// tutorial15.js
var CommentForm = React.createClass({
  render: function() {
    return (
      <form className="commentForm">
        <input type="text" placeholder="Your name" />
        <input type="text" placeholder="Say something..." />
        <input type="submit" value="Post" />
      </form>
    );
  }
});
```

Controlled components

With the traditional DOM, `input` elements are rendered and the browser manages the state (its rendered value). As a result, the state of the actual DOM will differ from that of the component. This is not ideal as the state of the view will differ from that of the component. In React, components should always represent the state of the view and not only at the point of initialization.

Hence, we will be using `this.state` to save the user's input as it is entered. We define an initial `state` with two properties `author` and `text` and set them to be empty strings. In our `<input>` elements, we set the `value` prop to reflect the `state` of the component and attach `onChange` handlers to them. These `<input>` elements with a `value` set are called controlled components. Read more about controlled components on the Forms article.

Code

```
var CommentForm = React.createClass({
  getInitialState: function() {
    return {author: '', text: ''};
  },
  handleAuthorChange: function(e) {
    this.setState({author: e.target.value});
  },
  handleTextChange: function(e) {
    this.setState({text: e.target.value});
  },
  render: function() {
    return (
      <form className="commentForm">
        <input
          type="text"
          placeholder="Your name"
          value={this.state.author}
          onChange={this.handleAuthorChange}
        />
        <input
          type="text"
          placeholder="Say something..."
          value={this.state.text}
          onChange={this.handleTextChange}
        />
        <input type="submit" value="Post" />
      </form>
    );
  }
});
```

Events

React attaches event handlers to components using a camelCase naming convention. We attach `onChange` handlers to the two `<input>` elements. Now, as the user enters text into the `<input>` fields, the attached `onChange` callbacks are fired and the `state` of the component is modified. Subsequently, the rendered value of the `input` element will be updated to reflect the current component `state`.

Submitting the form

Let's make the form interactive. When the user submits the form, we should clear it, submit a request to the server, and refresh the list of comments. To start, let's listen for the form's submit event and clear it.

Code

```
var CommentForm = React.createClass({
  getInitialState: function() {
    return {author: '', text: ''};
  },
  handleAuthorChange: function(e) {
    this.setState({author: e.target.value});
  },
  handleTextChange: function(e) {
    this.setState({text: e.target.value});
  },
  handleSubmit: function(e) {
    e.preventDefault();
    var author = this.state.author.trim();
    var text = this.state.text.trim();
    if (!text || !author) {
      return;
    }
    // TODO: send request to the server
    this.setState({author: '', text: ''});
  },
  render: function() {
    return (
      <form className="commentForm" onSubmit={this.handleSubmit}>
        <input
          type="text"
          placeholder="Your name"
          value={this.state.author}
          onChange={this.handleAuthorChange}
        />
        <input
          type="text"
          placeholder="Say something..."
          value={this.state.text}
          onChange={this.handleTextChange}
        />
        <input type="submit" value="Post" />
      </form>
    );
  }
});
```

We attach an `onSubmit` handler to the form that clears the form fields when the form is submitted with valid input.

Call `preventDefault()` on the event to prevent the browser's default action of submitting the form.

Callbacks as props

When a user submits a comment, we will need to refresh the list of comments to include the new one. It makes sense to do all of this logic in `CommentBox` since `CommentBox` owns the state that represents the list of comments.

We need to pass data from the child component back up to its parent. We do this in our parent's `render` method by passing a new callback (`handleCommentSubmit`) into the child, binding it to the child's `onCommentSubmit` event. Whenever the event is triggered, the callback will be invoked:

Code

```
// tutorial18.js
var CommentBox = React.createClass({
  loadCommentsFromServer: function() {
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      cache: false,
      success: function(data) {
        this.setState({data: data});
      }.bind(this),
      error: function(xhr, status, err) {
        console.error(this.props.url, status, err.toString());
      }.bind(this)
    });
  },
  handleCommentSubmit: function(comment) {
    // TODO: submit to the server and refresh the list
  },
  getInitialState: function() {
    return {data: []};
  },
  componentDidMount: function() {
    this.loadCommentsFromServer();
    setInterval(this.loadCommentsFromServer, this.props.pollInterval);
  },
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList data={this.state.data} />
        <CommentForm onCommentSubmit={this.handleCommentSubmit} />
      </div>
    );
  }
});
```

Now that `CommentBox` has made the callback available to `CommentForm` via the `onCommentSubmit` prop, the `CommentForm` can call the callback when the user submits the form:

Code

```
// tutorial19.js
var CommentForm = React.createClass({
  getInitialState: function() {
    return {author: '', text: ''};
  },
  handleAuthorChange: function(e) {
    this.setState({author: e.target.value});
  },
  handleTextChange: function(e) {
    this.setState({text: e.target.value});
  },
  handleSubmit: function(e) {
    e.preventDefault();
    var author = this.state.author.trim();
    var text = this.state.text.trim();
    if (!text || !author) {
      return;
    }
    this.props.onCommentSubmit({author: author, text: text});
    this.setState({author: '', text: ''});
  },
  render: function() {
    return (
      <form className="commentForm" onSubmit={this.handleSubmit}>
        <input
          type="text"
          placeholder="Your name"
          value={this.state.author}
          onChange={this.handleAuthorChange}
        />
        <input
          type="text"
          placeholder="Say something..."
          value={this.state.text}
          onChange={this.handleTextChange}
        />
        <input type="submit" value="Post" />
      </form>
    );
  }
});
```

Now that the callbacks are in place, all we have to do is submit to the server and refresh the list:

Code

```
var CommentBox = React.createClass({
  loadCommentsFromServer: function() {
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      cache: false,
      success: function(data) {
        this.setState({data: data});
      }.bind(this),
      error: function(xhr, status, err) {
        console.error(this.props.url, status, err.toString());
      }.bind(this)
    });
  },
  handleCommentSubmit: function(comment) {
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      type: 'POST',
      data: comment,
      success: function(data) {
        this.setState({data: data});
      }.bind(this),
      error: function(xhr, status, err) {
        console.error(this.props.url, status, err.toString());
      }.bind(this)
    });
  },
  getInitialState: function() {
    return {data: []};
  },
  componentDidMount: function() {
    this.loadCommentsFromServer();
    setInterval(this.loadCommentsFromServer, this.props.pollInterval);
  },
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList data={this.state.data} />
        <CommentForm onCommentSubmit={this.handleCommentSubmit} />
      </div>
    );
  }
});
```

Optimization: optimistic updates

Our application is now feature complete but it feels slow to have to wait for the request to complete before your comment appears in the list. We can optimistically add this comment to the list to make the app feel faster.

Code

```
var CommentBox = React.createClass({
  loadCommentsFromServer: function() {
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      cache: false,
      success: function(data) {
        this.setState({data: data});
      }.bind(this),
      error: function(xhr, status, err) {
        console.error(this.props.url, status, err.toString());
      }.bind(this)
    });
  },
  handleCommentSubmit: function(comment) {
    var comments = this.state.data;
    // Optimistically set an id on the new comment. It will be replaced by an
    // id generated by the server. In a production application you would likely
    // not use Date.now() for this and would have a more robust system in place.
    comment.id = Date.now();
    var newComments = comments.concat([comment]);
    this.setState({data: newComments});
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      type: 'POST',
      data: comment,
      success: function(data) {
        this.setState({data: data});
      }.bind(this),
      error: function(xhr, status, err) {
        this.setState({data: comments});
        console.error(this.props.url, status, err.toString());
      }.bind(this)
    });
  },
  getInitialState: function() {
    return {data: []};
  },
  componentDidMount: function() {
    this.loadCommentsFromServer();
    setInterval(this.loadCommentsFromServer, this.props.pollInterval);
  },
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList data={this.state.data} />
        <CommentForm onCommentSubmit={this.handleCommentSubmit} />
      </div>
    );
  }
});
```

Congrats!

You have just built a comment box in a few simple steps. Learn more about why to use React, or dive into the API reference and start hacking! Good luck!

← Prev   Next →

Thinking in React

Edit on GitHub

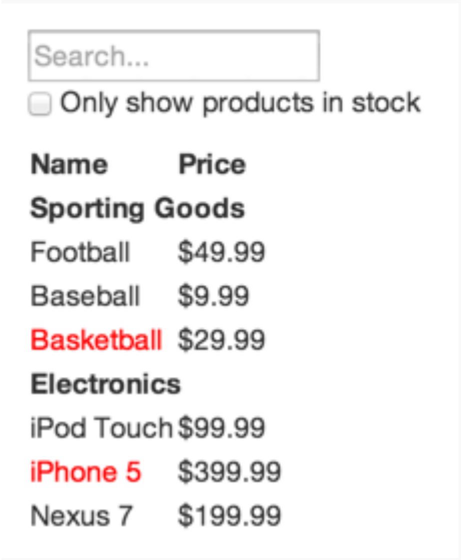
by Pete Hunt

React is, in my opinion, the premier way to build big, fast Web apps with JavaScript. It has scaled very well for us at Facebook and Instagram.

One of the many great parts of React is how it makes you think about apps as you build them. In this post, I'll walk you through the thought process of building a searchable product data table using React.

Start with a mock

Imagine that we already have a JSON API and a mock from our designer. Our designer apparently isn't very good because the mock looks like this:



Name	Price
Sporting Goods	
Football	$49.99
Baseball	$9.99
Basketball	$29.99
Electronics	
iPod Touch	$99.99
iPhone 5	$399.99
Nexus 7	$199.99

Our JSON API returns some data that looks like this:

Code

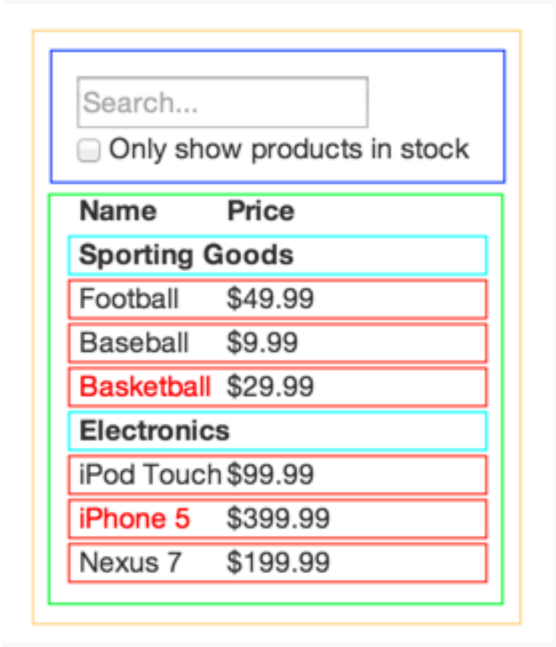
```
  {category: "Sporting Goods", price: "$49.99", stocked: true, name: "Football"},
  {category: "Sporting Goods", price: "$9.99", stocked: true, name: "Baseball"},
  {category: "Sporting Goods", price: "$29.99", stocked: false, name: "Basketball"},
  {category: "Electronics", price: "$99.99", stocked: true, name: "iPod Touch"},
  {category: "Electronics", price: "$399.99", stocked: false, name: "iPhone 5"},
  {category: "Electronics", price: "$199.99", stocked: true, name: "Nexus 7"}
];
```

Step 1: Break the UI into a component hierarchy

The first thing you'll want to do is to draw boxes around every component (and subcomponent) in the mock and give them all names. If you're working with a designer, they may have already done this, so go talk to them! Their Photoshop layer names may end up being the names of your React components!

But how do you know what should be its own component? Just use the same techniques for deciding if you should create a new function or object. One such technique is the single responsibility principle, that is, a component should ideally only do one thing. If it ends up growing, it should be decomposed into smaller subcomponents.

Since you're often displaying a JSON data model to a user, you'll find that if your model was built correctly, your UI (and therefore your component structure) will map nicely. That's because UI and data models tend to adhere to the same *information architecture*, which means the work of separating your UI into components is often trivial. Just break it up into components that represent exactly one piece of your data model.



Name	Price
Sporting Goods	
Football	$49.99
Baseball	$9.99
Basketball	$29.99
Electronics	
iPod Touch	$99.99
iPhone 5	$399.99
Nexus 7	$199.99

You'll see here that we have five components in our simple app. I've italicized the data each component represents.

1. **`FilterableProductTable` (orange):** contains the entirety of the example
2. **`SearchBar` (blue):** receives all *user input*
3. **`ProductTable` (green):** displays and filters the *data collection* based on *user input*
4. **`ProductCategoryRow` (turquoise):** displays a heading for each *category*
5. **`ProductRow` (red):** displays a row for each *product*

If you look at `ProductTable`, you'll see that the table header (containing the "Name" and "Price" labels) isn't its own component. This is a matter of preference, and there's an argument to be made either way. For this example, I left it as part of `ProductTable` because it is part of rendering the *data collection* which is `ProductTable`'s responsibility. However, if this header grows to be complex (i.e. if we were to add affordances for sorting), it would certainly make sense to make this its own `ProductTableHeader` component.

Now that we've identified the components in our mock, let's arrange them into a hierarchy. This is easy. Components that appear within another component in the mock should appear as a child in the hierarchy:

- `FilterableProductTable`
    - `SearchBar`
    - `ProductTable`
        - `ProductCategoryRow`
        - `ProductRow`

Step 2: Build a static version in React

Now that you have your component hierarchy, it's time to implement your app. The easiest way is to build a version that takes your data model and renders the UI but has no interactivity. It's best to decouple these processes because building a static version requires a lot of typing and no thinking, and adding interactivity requires a lot of thinking and not a lot of typing. We'll see why.

To build a static version of your app that renders your data model, you'll want to build components that reuse other components and pass data using *props*. *props* are a way of passing data from parent to child. If you're familiar with the concept of *state*, **don't use state at all** to build this static version. State is reserved only for interactivity, that is, data that changes over time. Since this is a static version of the app, you don't need it.

You can build top-down or bottom-up. That is, you can either start with building the components higher up in the hierarchy (i.e. starting with `FilterableProductTable`) or with the ones lower in it (`ProductRow`). In simpler examples, it's usually easier to go top-down, and on larger projects, it's easier to go bottom-up and write tests as you build.

At the end of this step, you'll have a library of reusable components that render your data model. The components will only have `render()` methods since this is a static version of your app. The component at the top of the hierarchy (`FilterableProductTable`) will take your data model as a prop. If you make a change to your underlying data model and call `ReactDOM.render()` again, the UI will be updated. It's easy to see how your UI is updated and where to make changes since there's nothing complicated going on. React's **one-way data flow** (also called *one-way binding*) keeps everything modular and fast.

Simply refer to the React docs if you need help executing this step.

A brief interlude: props vs state

There are two types of "model" data in React: props and state. It's important to understand the distinction between the two; skim the official React docs if you aren't sure what the difference is.

Step 3: Identify the minimal (but complete) representation of UI state

To make your UI interactive, you need to be able to trigger changes to your underlying data model. React makes this easy with **state**.

To build your app correctly, you first need to think of the minimal set of mutable state that your app needs. The key here is DRY: *Don't Repeat Yourself*. Figure out the absolute minimal representation of the state your application needs and compute everything else you need on-demand. For example, if you're building a TODO list, just keep an array of the TODO items around; don't keep a separate state variable for the count. Instead, when you want to render the TODO count, simply take the length of the TODO items array.

Think of all of the pieces of data in our example application. We have:

- The original list of products
- The search text the user has entered
- The value of the checkbox
- The filtered list of products

Let's go through each one and figure out which one is state. Simply ask three questions about each piece of data:

1. Is it passed in from a parent via props? If so, it probably isn't state.
2. Does it remain unchanged over time? If so, it probably isn't state.
3. Can you compute it based on any other state or props in your component? If so, it isn't state.

The original list of products is passed in as props, so that's not state. The search text and the checkbox seem to be state since they change over time and can't be computed from anything. And finally, the filtered list of products isn't state because it can be computed by combining the original list of products with the search text and value of the checkbox.

So finally, our state is:

- The search text the user has entered
- The value of the checkbox

Step 4: Identify where your state should live

OK, so we've identified what the minimal set of app state is. Next, we need to identify which component mutates, or *owns*, this state.

Remember: React is all about one-way data flow down the component hierarchy. It may not be immediately clear which component should own what state. **This is often the most challenging part for newcomers to understand,** so follow these steps to figure it out:

For each piece of state in your application:

- Identify every component that renders something based on that state.
- Find a common owner component (a single component above all the components that need the state in the hierarchy).
- Either the common owner or another component higher up in the hierarchy should own the state.
- If you can't find a component where it makes sense to own the state, create a new component simply for holding the state and add it somewhere in the hierarchy above the common owner component.

Let's run through this strategy for our application:

- `ProductTable` needs to filter the product list based on state and `SearchBar` needs to display the search text and checked state.
- The common owner component is `FilterableProductTable`.
- It conceptually makes sense for the filter text and checked value to live in `FilterableProductTable`

Cool, so we've decided that our state lives in `FilterableProductTable`. First, add a `getInitialState()` method to `FilterableProductTable` that returns `{filterText: '', inStockOnly: false}` to reflect the initial state of your application. Then, pass `filterText` and `inStockOnly` to `ProductTable` and `SearchBar` as a prop. Finally, use these props to filter the rows in `ProductTable` and set the values of the form fields in `SearchBar`.

You can start seeing how your application will behave: set `filterText` to `"ball"` and refresh your app. You'll see that the data table is updated correctly.

Step 5: Add inverse data flow

So far, we've built an app that renders correctly as a function of props and state flowing down the hierarchy. Now it's time to support data flowing the other way: the form components deep in the hierarchy need to update the state in `FilterableProductTable`.

React makes this data flow explicit to make it easy to understand how your program works, but it does require a little more typing than traditional two-way data binding.

If you try to type or check the box in the current version of the example, you'll see that React ignores your input. This is intentional, as we've set the `value` prop of the `input` to always be equal to the `state` passed in from `FilterableProductTable`.

Let's think about what we want to happen. We want to make sure that whenever the user changes the form, we update the state to reflect the user input. Since components should only update their own state, `FilterableProductTable` will pass a callback to `SearchBar` that will fire whenever the state should be updated. We can use the `onChange` event on the inputs to be notified of it. And the callback passed by `FilterableProductTable` will call `setState()`, and the app will be updated.

Though this sounds complex, it's really just a few lines of code. And it's really explicit how your data is flowing throughout the app.

And that's it

Hopefully, this gives you an idea of how to think about building components and applications with React. While it may be a little more typing than you're used to, remember that code is read far more than it's written, and it's extremely easy to read this modular, explicit code. As you start to build large libraries of components, you'll appreciate this explicitness and modularity, and with code reuse, your lines of code will start to shrink. :)

← Prev Next →

Conferences

Edit on GitHub

React.js Conf 2015

January 28 & 29 in Facebook HQ, CA

Website - Schedule - Videos

ReactEurope 2015

July 2 & 3 in Paris, France

Website - Schedule - Videos

Reactive 2015

November 2-4 in Bratislava, Slovakia

Website - Schedule

React.js Conf 2016

February 22 & 23 in San Francisco, CA

Website - Schedule - Videos

React Amsterdam 2016

April 16 in Amsterdam, The Netherlands

Website - Videos

ReactEurope 2016

June 2 & 3 in Paris, France

Website - Schedule

ReactRally 2016

August 25-26 in Salt Lake City, UT

Website - Schedule

ReactNext 2016

September 15 in Tel Aviv, Israel

Website - Schedule

Reactive 2016

October 26-28 in Bratislava, Slovakia

Website

← Prev

Next →

Videos

Edit on GitHub

Rethinking best practices - JSConf.eu

"At Facebook and Instagram, we’re trying to push the limits of what’s possible on the web with React. My talk will start with a brief introduction to the framework, and then dive into three controversial topics: Throwing out the notion of templates and building views with JavaScript, “re-rendering” your entire application when your data changes, and a lightweight implementation of the DOM and events." -- Pete Hunt

Thinking in react - tagtree.tv

A tagtree.tv video conveying the principles of Thinking in React while building a simple app

Secrets of the Virtual DOM - MtnWest JS

"In this talk I’ll be discussing why we built a virtual DOM, how it compares to other systems, and its relevance to the future of browser technologies." -- Pete Hunt

Going big with React

"On paper, all those JS frameworks look promising: clean implementations, quick code design, flawless execution. But what happens when you stress test Javascript? What happens when you throw 6 megabytes of code at it? In this talk, we'll investigate how React performs in a high stress situation, and how it has helped our team build safe code on a massive scale."

CodeWinds

Pete Hunt talked with Jeff Barczewski about React in CodeWinds Episode 4.

42:17

02:08 - What is React and why use it?
03:08 - The symbiotic relationship of ClojureScript and React
04:54 - The history of React and why it was created
09:43 - Updating web page with React without using data binding
13:11 - Using the virtual DOM to change the browser DOM
13:57 - Programming with React, render targets HTML, canvas, other
16:45 - Working with designers. Contrasted with Ember and AngularJS
21:45 - JSX Compiler bridging HTML and React javascript
23:50 - Autobuilding JSX and in browser tools for React
24:50 - Tips and tricks to working with React, getting started
27:17 - Rendering HTML on the server with Node.js. Rendering backends
29:20 - React evolved through survival of the fittest at Facebook
30:15 - Ideas for having state on server and client, using web sockets.
32:05 - React-multiuser - distributed shared mutable state using Firebase
33:03 - Better debugging with React using the state transitions, replaying events
34:08 - Differences from Web Components
34:25 - Notable companies using React
35:16 - Could a React backend plugin be created to target PDF?
36:30 - Future of React, what's next?
39:38 - Contributing and getting help

Read the episode notes

JavaScript Jabber

Pete Hunt and Jordan Walke talked about React in JavaScript Jabber 73.

01:34 – Pete Hunt Introduction
02:45 – Jordan Walke Introduction
04:15 – React
06:38 – 60 Frames Per Second
09:34 – Data Binding
12:31 – Performance
17:39 – Diffing Algorithm
19:36 – DOM Manipulation
23:06 – Supporting node.js
24:03 – rendr
26:02 – JSX
30:31 – requestAnimationFrame
34:15 – React and Applications
38:12 – React Users Khan Academy
39:53 – Making it work

Read the full transcript

Introduction to React.js - Facebook Seattle

By Tom Occhino and Jordan Walke

Backbone + React + Middleman Screencast

Backbone is a great way in interface a REST API with React. This screencast shows how to integrate the two using Backbone-React-Component. Middleman is the framework used in this example but could easily be replaced with other frameworks. A supported template of this can be found here. -- Open Minded Innovations

Developing User Interfaces With React - Super VanJS

By Steven Luscher

Introduction to React - LAWebSpeed meetup

by Stoyan Stefanov

React, or how to make life simpler - FrontEnd Dev Conf '14

In Russian by Alexander Solovyov

"Functional DOM programming" - Meteor DevShop 11

"Rethinking Web App Development at Facebook" - Facebook F8 Conference 2014

React and Flux: Building Applications with a Unidirectional Data Flow - Forward JS 2014

Facebook engineers Bill Fisher and Jing Chen talk about Flux and React, and how using an application architecture with a unidirectional data flow cleans up a lot of their code.

Server-Side Rendering of Isomorphic Apps at SoundCloud

Walk-through by Andres Suarez on how SoundCloud is using React and Flux for server-side rendering.

Slides and sample code

Introducing React Native (+Playlist) - React.js Conf 2015

Tom Occhino reviews the past and present of React in 2015, and teases where it's going next.

← Prev Next →

Why React?

Edit on GitHub

React is a JavaScript library for creating user interfaces by Facebook and Instagram. Many people choose to think of React as the **V** in **MVC**.

We built React to solve one problem: **building large applications with data that changes over time**.

Simple

Simply express how your app should look at any given point in time, and React will automatically manage all UI updates when your underlying data changes.

Declarative

When the data changes, React conceptually hits the "refresh" button, and knows to only update the changed parts.

Build Composable Components

React is all about building reusable components. In fact, with React the *only* thing you do is build components. Since they're so encapsulated, components make code reuse, testing, and separation of concerns easy.

Give It Five Minutes

React challenges a lot of conventional wisdom, and at first glance some of the ideas may seem crazy. Give it five minutes while reading this guide; those crazy ideas have worked for building thousands of components both inside and outside of Facebook and Instagram.

Learn More

You can learn more about our motivations behind building React in this blog post.

Next →

Displaying Data

Edit on GitHub

The most basic thing you can do with a UI is display some data. React makes it easy to display data and automatically keeps the interface up-to-date when the data changes.

Getting Started

Let's look at a really simple example. Create a `hello-react.html` file with the following code:

Code
```
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Hello React</title>
    <script src="https://fb.me/react-15.2.1.js"></script>
    <script src="https://fb.me/react-dom-15.2.1.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.34/
browser.min.js"></script>
  </head>
  <body>
    <div id="example"></div>
    <script type="text/babel">

      // ** Your code goes here! **

    </script>
  </body>
</html>
```

For the rest of the documentation, we'll just focus on the JavaScript code and assume it's inserted into a template like the one above. Replace the placeholder comment above with the following JSX:

Code
```
var HelloWorld = React.createClass({
  render: function() {
    return (
      <p>
        Hello, <input type="text" placeholder="Your name here" />!
        It is {this.props.date.toTimeString()}
      </p>
    );
  }
});

setInterval(function() {
  ReactDOM.render(
    <HelloWorld date={new Date()} />,
    document.getElementById('example')
  );
}, 500);
```

Reactive Updates

Open `hello-react.html` in a web browser and type your name into the text field. Notice that React is only changing the time string in the UI — any input you put in the text field remains, even though you haven't written any code to manage this behavior. React figures it out for you and does the right thing.

The way we are able to figure this out is that React does not manipulate the DOM unless it needs to. **It uses a fast, internal mock DOM to perform diffs and computes the most efficient DOM mutation for you.**

The inputs to this component are called `props` — short for "properties". They're passed as attributes in JSX syntax. You should think of these as immutable within the component, that is, **never write to `this.props`**.

Components are Just Like Functions

React components are very simple. You can think of them as simple functions that take in `props` and `state` (discussed later) and render HTML. With this in mind, components are easy to reason about.

Note:

One limitation: React components can only render a single root node. If you want to return multiple nodes they *must* be wrapped in a single root.

JSX Syntax

We strongly believe that components are the right way to separate concerns rather than "templates" and "display logic." We think that markup and the code that generates it are intimately tied together. Additionally, display logic is often very complex and using template languages to express it becomes cumbersome.

We've found that the best solution for this problem is to generate HTML and component trees directly from the JavaScript code such that you can use all of the expressive power of a real programming language to build UIs.

In order to make this easier, we've added a very simple, **optional** HTML-like syntax to create these React tree nodes.

JSX lets you create JavaScript objects using HTML syntax. To generate a link in React using pure JavaScript you'd write:

```
React.createElement('a', {href: 'https://facebook.github.io/react/'}, 'Hello!')
```

With JSX this becomes:

```
<a href="https://facebook.github.io/react/">Hello!</a>
```

We've found this has made building React apps easier and designers tend to prefer the syntax, but everyone has their own workflow, so **JSX is not required to use React.**

JSX is very small. To learn more about it, see JSX in depth. Or see the transform in action in the Babel REPL.

JSX is similar to HTML, but not exactly the same. See JSX gotchas for some key differences.

Babel exposes a number of ways to get started using JSX, ranging from command line tools to Ruby on Rails integrations. Choose the tool that works best for you.

React without JSX

JSX is completely optional; you don't have to use JSX with React. You can create React elements in plain JavaScript using `React.createElement`, which takes a tag name or component, a properties object, and variable number of optional child arguments.

Code

```
var child1 = React.createElement('li', null, 'First Text Content');
var child2 = React.createElement('li', null, 'Second Text Content');
var root = React.createElement('ul', { className: 'my-list' }, child1, child2);
ReactDOM.render(root, document.getElementById('example'));
```

For convenience, you can create short-hand factory functions to create elements from custom components.

Code

```
var Factory = React.createFactory(ComponentClass);
...
var root = Factory({ custom: 'prop' });
ReactDOM.render(root, document.getElementById('example'));
```

React already has built-in factories for common HTML tags:

Code

```
var root = React.DOM.ul({ className: 'my-list' },
             React.DOM.li(null, 'Text Content')
           );
```

← Prev

Next →

JSX in Depth

Edit on GitHub

JSX is a JavaScript syntax extension that looks similar to XML. You can use a simple JSX syntactic transform with React.

Why JSX?

You don't have to use JSX with React. You can just use plain JS. However, we recommend using JSX because it is a concise and familiar syntax for defining tree structures with attributes.

It's more familiar for casual developers such as designers.

XML has the benefit of balanced opening and closing tags. This helps make large trees easier to read than function calls or object literals.

It doesn't alter the semantics of JavaScript.

HTML Tags vs. React Components

React can either render HTML tags (strings) or React components (classes).

To render an HTML tag, just use lower-case tag names in JSX:

Code

```
var myDivElement = <div className="foo" />;
ReactDOM.render(myDivElement, document.getElementById('example'));
```

To render a React Component, just create a local variable that starts with an upper-case letter:

Code

```
var MyComponent = React.createClass({/*...*/});
var myElement = <MyComponent someProperty={true} />;
ReactDOM.render(myElement, document.getElementById('example'));
```

React's JSX uses the upper vs. lower case convention to distinguish between local component classes and HTML tags.

Note:

Since JSX is JavaScript, identifiers such as `class` and `for` are discouraged as XML attribute names. Instead, React DOM components expect DOM property names like `className` and `htmlFor`, respectively.

The Transform

React JSX transforms from an XML-like syntax into native JavaScript. XML elements, attributes and children are transformed into arguments that are passed to `React.createElement`.

Code

```
var Nav;
// Input (JSX):
var app = <Nav color="blue" />;
// Output (JS):
var app = React.createElement(Nav, {color:"blue"});
```

Notice that in order to use `<Nav />`, the `Nav` variable must be in scope.

JSX also allows specifying children using XML syntax:

Code

```
var Nav, Profile;
// Input (JSX):
var app = <Nav color="blue"><Profile>click</Profile></Nav>;
// Output (JS):
var app = React.createElement(
  Nav,
  {color:"blue"},
  React.createElement(Profile, null, "click")
);
```

JSX will infer the class's displayName from the variable assignment when the displayName is undefined:

Code

```
// Input (JSX):
var Nav = React.createClass({ });
// Output (JS):
var Nav = React.createClass({displayName: "Nav", });
```

Use the Babel REPL to try out JSX and see how it desugars into native JavaScript, and the HTML to JSX converter to convert your existing HTML to JSX.

If you want to use JSX, the Getting Started guide shows how to set up compilation.

Note:

The JSX expression always evaluates to a ReactElement. The actual implementation details may vary. An optimized mode could inline the ReactElement as an object literal to bypass the validation code in `React.createElement`.

Namespaced Components

If you are building a component that has many children, like a form, you might end up with something with a lot of variable declarations:

Code

```
var Form = MyFormComponent;
var FormRow = Form.Row;
var FormLabel = Form.Label;
var FormInput = Form.Input;

var App = (
  <Form>
    <FormRow>
      <FormLabel />
      <FormInput />
    </FormRow>
  </Form>
);
```

To make it simpler and easier, *namespaced components* let you use one component that has other components as attributes:

Code

```
var Form = MyFormComponent;

var App = (
  <Form>
    <Form.Row>
      <Form.Label />
      <Form.Input />
    </Form.Row>
  </Form>
);
```

To do this, you just need to create your *"sub-components"* as attributes of the main component:

Code

```
var MyFormComponent = React.createClass({ ... });

MyFormComponent.Row = React.createClass({ ... });
MyFormComponent.Label = React.createClass({ ... });
MyFormComponent.Input = React.createClass({ ... });
```

JSX will handle this properly when compiling your code.

Code

```
  React.createElement(Form, null,
    React.createElement(Form.Row, null,
      React.createElement(Form.Label, null),
      React.createElement(Form.Input, null)
    )
  )
);
```

Note:

This feature is available in v0.11 and above.

JavaScript Expressions

Attribute Expressions

To use a JavaScript expression as an attribute value, wrap the expression in a pair of curly braces (`{}`) instead of quotes (`""`).

Code

```
// Input (JSX):
var person = <Person name={window.isLoggedIn ? window.name : ''} />;
// Output (JS):
var person = React.createElement(
  Person,
  {name: window.isLoggedIn ? window.name : ''}
);
```

Boolean Attributes

Omitting the value of an attribute causes JSX to treat it as `true`. To pass `false` an attribute expression must be used. This often comes up when using HTML form elements, with attributes like `disabled`, `required`, `checked` and `readOnly`.

Code

```
// These two are equivalent in JSX for disabling a button
<input type="button" disabled />;
<input type="button" disabled={true} />;

// And these two are equivalent in JSX for not disabling a button
<input type="button" />;
<input type="button" disabled={false} />;
```

Child Expressions

Likewise, JavaScript expressions may be used to express children:

Code

```
// Input (JSX):
var content = <Container>{window.isLoggedIn ? <Nav /> : <Login />}</Container>;
// Output (JS):
var content = React.createElement(
  Container,
  null,
  window.isLoggedIn ? React.createElement(Nav) : React.createElement(Login)
);
```

Comments

It's easy to add comments within your JSX; they're just JS expressions. You just need to be careful to put `{}` around the comments when you are within the children section of a tag.

Code

```
var content = (
  <Nav>
    {/* child comment, put {} around */}
    <Person
      /* multi
         line
         comment */
      name={window.isLoggedIn ? window.name : ''} // end of line comment
    />
  </Nav>
);
```

NOTE:

JSX is similar to HTML, but not exactly the same. See JSX gotchas for some key differences.

← Prev Next →

JSX Spread Attributes

Edit on GitHub

If you know all the properties that you want to place on a component ahead of time, it is easy to use JSX:

Code

Mutating Props is Bad

If you don't know which properties you want to set, you might be tempted to add them onto the object later:

Code

```
var component = <Component />;
component.props.foo = x; // bad
component.props.bar = y; // also bad
```

This is an anti-pattern because it means that we can't help you check the right propTypes until way later. This means that your propTypes errors end up with a cryptic stack trace.

The props should be considered immutable. Mutating the props object somewhere else could cause unexpected consequences so ideally it would be a frozen object at this point.

Spread Attributes

Now you can use a new feature of JSX called spread attributes:

Code

```
var props = {};
props.foo = x;
props.bar = y;
var component = <Component {...props} />;
```

The properties of the object that you pass in are copied onto the component's props.

You can use this multiple times or combine it with other attributes. The specification order is important. Later attributes override previous ones.

Code

```
var props = { foo: 'default' };
var component = <Component {...props} foo={'override'} />;
console.log(component.props.foo); // 'override'
```

What's with the weird `...` notation?

The `...` operator (or spread operator) is already supported for arrays in ES6. There is also an ECMAScript proposal for Object Rest and Spread Properties. We're taking advantage of these supported and developing standards in order to provide a cleaner syntax in JSX.

← Prev Next →

JSX Gotchas

Edit on GitHub

JSX looks like HTML but there are some important differences you may run into.

Note:

For DOM differences, such as the inline `style` attribute, check here.

HTML Entities

You can insert HTML entities within literal text in JSX:

Code

```
<div>First &middot; Second</div>
```

If you want to display an HTML entity within dynamic content, you will run into double escaping issues as React escapes all the strings you are displaying in order to prevent a wide range of XSS attacks by default.

Code

```
// Bad: It displays "First &middot; Second"
<div>{'First &middot; Second'}</div>
```

There are various ways to work-around this issue. The easiest one is to write Unicode characters directly in JavaScript. You need to make sure that the file is saved as UTF-8 and that the proper UTF-8 directives are set so the browser will display it correctly.

Code

```
<div>{'First · Second'}</div>
```

A safer alternative is to find the unicode number corresponding to the entity and use it inside of a JavaScript string.

Code

```
<div>{'First \u00b7 Second'}</div>
<div>{'First ' + String.fromCharCode(183) + ' Second'}</div>
```

You can use mixed arrays with strings and JSX elements. Each JSX element in the array needs a unique key.

Code

```
<div>{['First ', <span key="middot">&middot;</span>, ' Second']}</div>
```

As a last resort, you always have the ability to insert raw HTML.

Code

Custom HTML Attributes

If you pass properties to native HTML elements that do not exist in the HTML specification, React will not render them. If you want to use a custom attribute, you should prefix it with `data-`.

Code

```
<div data-custom-attribute="foo" />
```

However, arbitrary attributes are supported on custom elements (those with a hyphen in the tag name or an `is="..."` attribute).

Code

```
<x-my-component custom-attribute="foo" />
```

Web Accessibility attributes starting with `aria-` will be rendered properly.

Code

```
<div aria-hidden={true} />
```

← Prev Next →

Interactivity and Dynamic UIs

Edit on GitHub

You've already learned how to display data with React. Now let's look at how to make our UIs interactive.

A Simple Example

Code
```
  constructor() {
    super();
    this.state = {
      liked: false
    };
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    this.setState({liked: !this.state.liked});
  }
  render() {
    const text = this.state.liked ? 'liked' : 'haven\'t liked';
    return (
      <div onClick={this.handleClick}>
        You {text} this. Click to toggle.
      </div>
    );
  }
}

ReactDOM.render(
  <LikeButton />,
  document.getElementById('example')
);
```

Event Handling and Synthetic Events

With React you simply pass your event handler as a camelCased prop similar to how you'd do it in normal HTML. React ensures that all events behave similarly in all browsers by implementing a synthetic event system. That is, React knows how to bubble and capture events according to the spec, and the events passed to your event handler are guaranteed to be consistent with the W3C spec, regardless of which browser you're using.

Under the Hood: Autobinding and Event Delegation

Under the hood, React does a few things to keep your code performant and easy to understand.

Autobinding: When creating callbacks in JavaScript, you usually need to explicitly bind a method to its instance such that the value of `this` is correct. With React, every method is automatically bound to its component instance (except when using ES6 class syntax). React caches the bound method such that it's extremely CPU and memory efficient. It's also less typing!

Event delegation: React doesn't actually attach event handlers to the nodes themselves. When React starts up, it starts listening for all events at the top level using a single event listener. When a component is mounted or unmounted, the event handlers are simply added or removed from an internal mapping. When an event occurs, React knows how to dispatch it using this mapping. When there are no event handlers left in the mapping, React's event handlers are simple no-ops. To learn more about why this is fast, see David Walsh's excellent blog post.

Components are Just State Machines

React thinks of UIs as simple state machines. By thinking of a UI as being in various states and rendering those states, it's easy to keep your UI consistent.

In React, you simply update a component's state, and then render a new UI based on this new state. React takes care of updating the DOM for you in the most efficient way.

How State Works

A common way to inform React of a data change is by calling `setState(data, callback)`. This method merges `data` into `this.state` and re-renders the component. When the component finishes re-rendering, the optional `callback` is called. Most of the time you'll never need to provide a `callback` since React will take care of keeping your UI up-to-date for you.

What Components Should Have State?

Most of your components should simply take some data from `props` and render it. However, sometimes you need to respond to user input, a server request or the passage of time. For this you use state.

Try to keep as many of your components as possible stateless. By doing this you'll isolate the state to its most logical place and minimize redundancy, making it easier to reason about your application.

A common pattern is to create several stateless components that just render data, and have a stateful component above them in the hierarchy that passes its state to its children via `props`. The stateful component encapsulates all of the interaction logic, while the stateless components take care of rendering data in a declarative way.

What *Should* Go in State?

State should contain data that a component's event handlers may change to trigger a UI update. In real apps this data tends to be very small and JSON-serializable. When building a stateful component, think about the minimal possible representation of its state, and only store those properties in `this.state`. Inside of `render()` simply compute any other information you need based on this state. You'll find that thinking about and writing applications in this way tends to lead to the most correct application, since adding redundant or computed values to state means that you need to explicitly keep them in sync rather than rely on React computing them for you.

What *Shouldn't* Go in State?

`this.state` should only contain the minimal amount of data needed to represent your UI's state. As such, it should not contain:

- **Computed data:** Don't worry about precomputing values based on state — it's easier to ensure that your UI is consistent if you do all computation within `render()`. For example, if you have an array of list items in state and you want to render the count as a string, simply render `this.state.listItems.length + ' list items'` in your `render()` method rather than storing it on state.
- **React components:** Build them in `render()` based on underlying props and state.
- **Duplicated data from props:** Try to use props as the source of truth where possible. One valid use to store props in state is to be able to know its previous values, because props may change as the result of a parent component re-rendering.

← Prev Next →

Multiple Components

Edit on GitHub

So far, we've looked at how to write a single component to display data and handle user input. Next let's examine one of React's finest features: composability.

Motivation: Separation of Concerns

By building modular components that reuse other components with well-defined interfaces, you get much of the same benefits that you get by using functions or classes. Specifically you can *separate the different concerns* of your app however you please simply by building new components. By building a custom component library for your application, you are expressing your UI in a way that best fits your domain.

Composition Example

Let's create a simple Avatar component which shows a Facebook page picture and name using the Facebook Graph API.

Code

```
  render: function() {
    return (
      <div>
        <PagePic pagename={this.props.pagename} />
        <PageLink pagename={this.props.pagename} />
      </div>
    );
  }
});

var PagePic = React.createClass({
  render: function() {
    return (
      <img src={'https://graph.facebook.com/' + this.props.pagename + '/
picture'} />
    );
  }
});

var PageLink = React.createClass({
  render: function() {
    return (
      <a href={'https://www.facebook.com/' + this.props.pagename}>
        {this.props.pagename}
      </a>
    );
  }
});

ReactDOM.render(
  <Avatar pagename="Engineering" />,
  document.getElementById('example')
);
```

Ownership

In the above example, instances of `Avatar` *own* instances of `PagePic` and `PageLink`. In React, **an owner is the component that sets the `props` of other components**. More formally, if a component `X` is created in component `Y`'s `render()` method, it is said that `X` is *owned by* `Y`. As discussed earlier, a component cannot mutate its `props` — they are always consistent with what its owner sets them to. This fundamental invariant leads to UIs that are guaranteed to be consistent.

It's important to draw a distinction between the owner-ownee relationship and the parent-child relationship. The owner-ownee relationship is specific to React, while the parent-child relationship is simply the one you know and love from the DOM. In the example above, `Avatar` owns the `div`, `PagePic` and `PageLink` instances, and `div` is the **parent** (but not owner) of the `PagePic` and `PageLink` instances.

Children

When you create a React component instance, you can include additional React components or JavaScript expressions between the opening and closing tags like this:

Code

```
<Parent><Child /></Parent>
```

`Parent` can read its children by accessing the special `this.props.children` prop. **`this.props.children` is an opaque data structure:** use the React.Children utilities to manipulate them.

Child Reconciliation

Reconciliation is the process by which React updates the DOM with each new render pass. In general, children are reconciled according to the order in which they are rendered. For example, suppose two render passes generate the following respective markup:

Code

```
// Render Pass 1
<Card>
  <p>Paragraph 1</p>
  <p>Paragraph 2</p>
</Card>
// Render Pass 2
<Card>
  <p>Paragraph 2</p>
</Card>
```

Intuitively, `<p>Paragraph 1</p>` was removed. Instead, React will reconcile the DOM by changing the text content of the first child and destroying the last child. React reconciles according to the *order* of the children.

Stateful Children

For most components, this is not a big deal. However, for stateful components that maintain data in `this.state` across render passes, this can be very problematic.

In most cases, this can be sidestepped by hiding elements instead of destroying them:

Code
```
<Card>
  <p>Paragraph 1</p>
  <p>Paragraph 2</p>
</Card>
// Render Pass 2
<Card>
  <p style={{display: 'none'}}>Paragraph 1</p>
  <p>Paragraph 2</p>
</Card>
```

Dynamic Children

The situation gets more complicated when the children are shuffled around (as in search results) or if new components are added onto the front of the list (as in streams). In these cases where the identity and state of each child must be maintained across render passes, you can uniquely identify each child by assigning it a `key`:

Code
```
  render: function() {
    var results = this.props.results;
    return (
      <ol>
        {results.map(function(result) {
          return <li key={result.id}>{result.text}</li>;
        })}
      </ol>
    );
  }
```

When React reconciles the keyed children, it will ensure that any child with `key` will be reordered (instead of clobbered) or destroyed (instead of reused).

The `key` should *always* be supplied directly to the components in the array, not to the container HTML child of each component in the array:

Code

```
var ListItemWrapper = React.createClass({
  render: function() {
    return <li key={this.props.data.id}>{this.props.data.text}</li>;
  }
});
var MyComponent = React.createClass({
  render: function() {
    return (
      <ul>
        {this.props.results.map(function(result) {
          return <ListItemWrapper data={result}/>;
        })}
      </ul>
    );
  }
});
```

Code

```
// Correct :)
var ListItemWrapper = React.createClass({
  render: function() {
    return <li>{this.props.data.text}</li>;
  }
});
var MyComponent = React.createClass({
  render: function() {
    return (
      <ul>
        {this.props.results.map(function(result) {
           return <ListItemWrapper key={result.id} data={result}/>;
        })}
      </ul>
    );
  }
});
```

You can also key children by passing a ReactFragment object. See Keyed Fragments for more details.

Data Flow

In React, data flows from owner to owned component through `props` as discussed above. This is effectively one-way data binding: owners bind their owned component's props to some value the owner has computed based on its `props` or `state`. Since this process happens recursively, data changes are automatically reflected everywhere they are used.

A Note on Performance

You may be thinking that it's expensive to change data if there are a large number of nodes under an owner. The good news is that JavaScript is fast and `render()` methods tend to be quite simple, so in most applications this is extremely fast. Additionally, the bottleneck is almost always the DOM mutation and not JS execution. React will optimize this for you by using batching and change detection.

However, sometimes you really want to have fine-grained control over your performance. In that case, simply override `shouldComponentUpdate()` to return false when you want React to skip processing of a subtree. See the React reference docs for more information.

Note:

If `shouldComponentUpdate()` returns false when data has actually changed, React can't keep your UI in sync. Be sure you know what you're doing while using it, and only use this function when you have a noticeable performance problem. Don't underestimate how fast JavaScript is relative to the DOM.

← Prev Next →

Reusable Components

Edit on GitHub

When designing interfaces, break down the common design elements (buttons, form fields, layout components, etc.) into reusable components with well-defined interfaces. That way, the next time you need to build some UI, you can write much less code. This means faster development time, fewer bugs, and fewer bytes down the wire.

Prop Validation

As your app grows it's helpful to ensure that your components are used correctly. We do this by allowing you to specify `propTypes`. `React.PropTypes` exports a range of validators that can be used to make sure the data you receive is valid. When an invalid value is provided for a prop, a warning will be shown in the JavaScript console. Note that for performance reasons `propTypes` is only checked in development mode. Here is an example documenting the different validators provided:

Code

```
  propTypes: {
    // You can declare that a prop is a specific JS primitive. By default, these
    // are all optional.
    optionalArray: React.PropTypes.array,
    optionalBool: React.PropTypes.bool,
    optionalFunc: React.PropTypes.func,
    optionalNumber: React.PropTypes.number,
    optionalObject: React.PropTypes.object,
    optionalString: React.PropTypes.string,
    optionalSymbol: React.PropTypes.symbol,

    // Anything that can be rendered: numbers, strings, elements or an array
    // (or fragment) containing these types.
    optionalNode: React.PropTypes.node,

    // A React element.
    optionalElement: React.PropTypes.element,

    // You can also declare that a prop is an instance of a class. This uses
    // JS's instanceof operator.
    optionalMessage: React.PropTypes.instanceOf(Message),

    // You can ensure that your prop is limited to specific values by treating
    // it as an enum.
    optionalEnum: React.PropTypes.oneOf(['News', 'Photos']),

    // An object that could be one of many types
    optionalUnion: React.PropTypes.oneOfType([
      React.PropTypes.string,
      React.PropTypes.number,
      React.PropTypes.instanceOf(Message)
    ]),

    // An array of a certain type
    optionalArrayOf: React.PropTypes.arrayOf(React.PropTypes.number),

    // An object with property values of a certain type
    optionalObjectOf: React.PropTypes.objectOf(React.PropTypes.number),

    // An object taking on a particular shape
    optionalObjectWithShape: React.PropTypes.shape({
      color: React.PropTypes.string,
      fontSize: React.PropTypes.number
    }),

    // You can chain any of the above with `isRequired` to make sure a warning
    // is shown if the prop isn't provided.
    requiredFunc: React.PropTypes.func.isRequired,

    // A value of any data type
    requiredAny: React.PropTypes.any.isRequired,

    // You can also specify a custom validator. It should return an Error
```

Single Child

With `React.PropTypes.element` you can specify that only a single child can be passed to a component as children.

Code

```
var MyComponent = React.createClass({
  propTypes: {
    children: React.PropTypes.element.isRequired
  },

  render: function() {
    return (
      <div>
        {this.props.children} // This must be exactly one element or it will warn.
      </div>
    );
  }

});
```

Default Prop Values

React lets you define default values for your `props` in a very declarative way:

Code

```
var ComponentWithDefaultProps = React.createClass({
  getDefaultProps: function() {
    return {
      value: 'default value'
    };
  }
  /* ... */
});
```

The result of `getDefaultProps()` will be cached and used to ensure that `this.props.value` will have a value if it was not specified by the parent component. This allows you to safely just use your props without having to write repetitive and fragile code to handle that yourself.

Transferring Props: A Shortcut

A common type of React component is one that extends a basic HTML element in a simple way. Often you'll want to copy any HTML attributes passed to your component to the underlying HTML element. To save typing, you can use the JSX *spread* syntax to achieve this:

Code

```
  render: function() {
    // This takes any props passed to CheckLink and copies them to <a>
    return <a {...this.props}>{'√ '}{this.props.children}</a>;
  }
});

ReactDOM.render(
  <CheckLink href="/checked.html">
    Click here!
  </CheckLink>,
  document.getElementById('example')
);
```

Mixins

Components are the best way to reuse code in React, but sometimes very different components may share some common functionality. These are sometimes called cross-cutting concerns. React provides `mixins` to solve this problem.

One common use case is a component wanting to update itself on a time interval. It's easy to use `setInterval()`, but it's important to cancel your interval when you don't need it anymore to save memory. React provides lifecycle methods that let you know when a component is about to be created or destroyed. Let's create a simple mixin that uses these methods to provide an easy `setInterval()` function that will automatically get cleaned up when your component is destroyed.

Code

```
  componentWillMount: function() {
    this.intervals = [];
  },
  setInterval: function() {
    this.intervals.push(setInterval.apply(null, arguments));
  },
  componentWillUnmount: function() {
    this.intervals.forEach(clearInterval);
  }
};

var TickTock = React.createClass({
  mixins: [SetIntervalMixin], // Use the mixin
  getInitialState: function() {
    return {seconds: 0};
  },
  componentDidMount: function() {
    this.setInterval(this.tick, 1000); // Call a method on the mixin
  },
  tick: function() {
    this.setState({seconds: this.state.seconds + 1});
  },
  render: function() {
    return (
      <p>
        React has been running for {this.state.seconds} seconds.
      </p>
    );
  }
});

ReactDOM.render(
  <TickTock />,
  document.getElementById('example')
);
```

A nice feature of mixins is that if a component is using multiple mixins and several mixins define the same lifecycle method (i.e. several mixins want to do some cleanup when the component is destroyed), all of the lifecycle methods are guaranteed to be called. Methods defined on mixins run in the order mixins were listed, followed by a method call on the component.

ES6 Classes

You may also define your React classes as a plain JavaScript class. For example using ES6 class syntax:

Code

```
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}
ReactDOM.render(<HelloMessage name="Sebastian" />, mountNode);
```

The API is similar to `React.createClass` with the exception of `getInitialState`. Instead of providing a separate `getInitialState` method, you set up your own `state` property in the constructor. Just like the return value of `getInitialState`, the value you assign to `this.state` will be used as the initial state for your component.

Another difference is that `propTypes` and `defaultProps` are defined as properties on the constructor instead of in the class body.

Code

```
export class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: props.initialCount};
    this.tick = this.tick.bind(this);
  }
  tick() {
    this.setState({count: this.state.count + 1});
  }
  render() {
    return (
      <div onClick={this.tick}>
        Clicks: {this.state.count}
      </div>
    );
  }
}
Counter.propTypes = { initialCount: React.PropTypes.number };
Counter.defaultProps = { initialCount: 0 };
```

No Autobinding

Methods follow the same semantics as regular ES6 classes, meaning that they don't automatically bind `this` to the instance. You'll have to explicitly use `.bind(this)` or arrow functions `=>`:

Code
```
<div onClick={this.tick.bind(this)}>

// Or you can use arrow functions
<div onClick={() => this.tick()}>
```

We recommend that you bind your event handlers in the constructor so they are only bound once for every instance:

Code
```
constructor(props) {
  super(props);
  this.state = {count: props.initialCount};
  this.tick = this.tick.bind(this);
}
```

Now you can use `this.tick` directly as it was bound once in the constructor:

Code
```
// It is already bound in the constructor
<div onClick={this.tick}>
```

This is better for performance of your application, especially if you implement shouldComponentUpdate() with a shallow comparison in the child components.

No Mixins

Unfortunately ES6 launched without any mixin support. Therefore, there is no support for mixins when you use React with ES6 classes. Instead, we're working on making it easier to support such use cases without resorting to mixins.

Stateless Functions

You may also define your React classes as a plain JavaScript function. For example using the stateless function syntax:

Code
```
function HelloMessage(props) {
  return <div>Hello {props.name}</div>;
}
ReactDOM.render(<HelloMessage name="Sebastian" />, mountNode);
```

Or using the new ES6 arrow syntax:

Code
```
const HelloMessage = (props) => <div>Hello {props.name}</div>;
ReactDOM.render(<HelloMessage name="Sebastian" />, mountNode);
```

This simplified component API is intended for components that are pure functions of their props. These components must not retain internal state, do not have backing instances, and do not have the component lifecycle methods. They are pure functional transforms of their input, with zero boilerplate. However, you may still specify `.propTypes` and `.defaultProps` by setting them as properties on the function, just as you would set them on an ES6 class.

NOTE:

Because stateless functions don't have a backing instance, you can't attach a ref to a stateless function component. Normally this isn't an issue, since stateless functions do not provide an imperative API. Without an imperative API, there isn't much you could do with an instance anyway. However, if a user wants to find the DOM node of a stateless function component, they must wrap the component in a stateful component (eg. ES6 class component) and attach the ref to the stateful wrapper component.

NOTE:

In React v0.14, stateless functional components were not permitted to return `null` or `false` (a workaround is to return a `<noscript />` instead). This was fixed in React v15, and stateless functional components are now permitted to return `null`.

In an ideal world, most of your components would be stateless functions because in the future we’ll also be able to make performance optimizations specific to these components by avoiding unnecessary checks and memory allocations. This is the recommended pattern, when possible.

← Prev Next →

Transferring Props

Edit on GitHub

It's a common pattern in React to wrap a component in an abstraction. The outer component exposes a simple property to do something that might have more complex implementation details.

You can use JSX spread attributes to merge the old props with additional values:

Code

```
<Component {...this.props} more="values" />
```

If you don't use JSX, you can use any object helper such as ES6 `Object.assign` or Underscore `_.extend`:

Code

```
React.createElement(Component, Object.assign({}, this.props, { more: 'values' }
));
```

The rest of this tutorial explains best practices. It uses JSX and experimental ECMAScript syntax.

Manual Transfer

Most of the time you should explicitly pass the properties down. This ensures that you only expose a subset of the inner API, one that you know will work.

Code

```
function FancyCheckbox(props) {
  var fancyClass = props.checked ? 'FancyChecked' : 'FancyUnchecked';
  return (
    <div className={fancyClass} onClick={props.onClick}>
      {props.children}
    </div>
  );
}
ReactDOM.render(
  <FancyCheckbox checked={true} onClick={console.log.bind(console)}>
    Hello world!
  </FancyCheckbox>,
  document.getElementById('example')
);
```

But what about the `name` prop? Or the `title` prop? Or `onMouseOver`?

Transferring with ... in JSX

NOTE:

The `...` syntax is part of the Object Rest Spread proposal. This proposal is on track to become a standard. See the Rest and Spread Properties ... section below for more details.

Sometimes it's fragile and tedious to pass every property along. In that case you can use destructuring assignment with rest properties to extract a set of unknown properties.

List out all the properties that you would like to consume, followed by `...other`.

Code

```
var { checked, ...other } = props;
```

This ensures that you pass down all the props EXCEPT the ones you're consuming yourself.

Code

```
  var { checked, ...other } = props;
  var fancyClass = checked ? 'FancyChecked' : 'FancyUnchecked';
  // `other` contains { onClick: console.log } but not the checked property
  return (
    <div {...other} className={fancyClass} />
  );
}
ReactDOM.render(
  <FancyCheckbox checked={true} onClick={console.log.bind(console)}>
    Hello world!
  </FancyCheckbox>,
  document.getElementById('example')
);
```

NOTE:

In the example above, the `checked` prop is also a valid DOM attribute. If you didn't use destructuring in this way you might inadvertently pass it along.

Always use the destructuring pattern when transferring unknown `other` props.

Code

```
function FancyCheckbox(props) {
  var fancyClass = props.checked ? 'FancyChecked' : 'FancyUnchecked';
  // ANTI-PATTERN: `checked` would be passed down to the inner component
  return (
    <div {...props} className={fancyClass} />
  );
}
```

Consuming and Transferring the Same Prop

If your component wants to consume a property but also wants to pass it along, you can repass it explicitly with `checked={checked}`. This is preferable to passing the full `props` object since it's easier to refactor and lint.

Code

```
  var { checked, title, ...other } = props;
  var fancyClass = checked ? 'FancyChecked' : 'FancyUnchecked';
  var fancyTitle = checked ? 'X ' + title : 'O ' + title;
  return (
    <label>
      <input {...other}
        checked={checked}
        className={fancyClass}
        type="checkbox"
      />
      {fancyTitle}
    </label>
  );
}
```

NOTE:

Order matters. By putting the `{...other}` before your JSX props you ensure that the consumer of your component can't override them. In the example above we have guaranteed that the input will be of type `"checkbox"`.

Rest and Spread Properties ...

Rest properties allow you to extract the remaining properties from an object into a new object. It excludes every other property listed in the destructuring pattern.

This is an experimental implementation of an ECMAScript proposal.

Code

```
var { x, y, ...z } = { x: 1, y: 2, a: 3, b: 4 };
x; // 1
y; // 2
z; // { a: 3, b: 4 }
```

Note:

To transform rest and spread properties using Babel 6, you need to install the `es2015` preset, the `transform-object-rest-spread` plugin and configure them in the `.babelrc` file.

Transferring with Underscore

If you don't use JSX, you can use a library to achieve the same pattern. Underscore supports `_.omit` to filter out properties and `_.extend` to copy properties onto a new object.

Code

```
  var checked = props.checked;
  var other = _.omit(props, 'checked');
  var fancyClass = checked ? 'FancyChecked' : 'FancyUnchecked';
  return (
    React.DOM.div(_.extend({}, other, { className: fancyClass }))
  );
}
```

Forms

Edit on GitHub

Form components such as `<input>`, `<textarea>`, and `<option>` differ from other native components because they can be mutated via user interactions. These components provide interfaces that make it easier to manage forms in response to user interactions.

For information on events on `<form>` see Form Events.

Interactive Props

Form components support a few props that are affected via user interactions:

- `value`, supported by `<input>` and `<textarea>` components.
- `checked`, supported by `<input>` components of type `checkbox` or `radio`.
- `selected`, supported by `<option>` components.

In HTML, the value of `<textarea>` is set via children. In React, you should use `value` instead.

Form components allow listening for changes by setting a callback to the `onChange` prop. The `onChange` prop works across browsers to fire in response to user interactions when:

- The `value` of `<input>` or `<textarea>` changes.
- The `checked` state of `<input>` changes.
- The `selected` state of `<option>` changes.

Like all DOM events, the `onChange` prop is supported on all native components and can be used to listen to bubbled change events.

Note:

For `<input>` and `<textarea>`, `onChange` supersedes — and should generally be used instead of — the DOM's built-in `oninput` event handler.

Controlled Components

A **controlled** `<input>` has a `value` prop. Rendering a controlled `<input>` will reflect the value of the `value` prop.

Code

```
  render: function() {
    return <input type="text" value="Hello!" />;
  }
```

User input will have no effect on the rendered element because React has declared the value to be `Hello!`. To update the value in response to user input, you could use the `onChange` event:

Code

```
  getInitialState: function() {
    return {value: 'Hello!'};
  },
  handleChange: function(event) {
    this.setState({value: event.target.value});
  },
  render: function() {
    return (
      <input
        type="text"
        value={this.state.value}
        onChange={this.handleChange}
      />
    );
  }
```

In this example, we are accepting the value provided by the user and updating the `value` prop of the `<input>` component. This pattern makes it easy to implement interfaces that respond to or validate user interactions. For example:

Code

```
  handleChange: function(event) {
    this.setState({value: event.target.value.substr(0, 140)});
  }
```

This would accept user input and truncate the value to the first 140 characters.

A **Controlled** component does not maintain its own internal state; the component renders purely based on props.

Potential Issues With Checkboxes and Radio Buttons

Be aware that, in an attempt to normalize change handling for checkbox and radio inputs, React uses a `click` event in place of a `change` event. For the most part this behaves as expected, except when calling `preventDefault` in a `change` handler. `preventDefault` stops the browser from visually updating the input, even if `checked` gets toggled. This can be worked around either by removing the call to `preventDefault`, or putting the toggle of `checked` in a `setTimeout`.

Uncontrolled Components

An `<input>` without a `value` property is an *uncontrolled* component:

Code

```
  render: function() {
    return <input type="text" />;
  }
```

This will render an input that starts off with an empty value. Any user input will be immediately reflected by the rendered element. If you wanted to listen to updates to the value, you could use the `onChange` event just like you can with controlled components.

An **uncontrolled** component maintains its own internal state.

Default Value

If you want to initialize the component with a non-empty value, you can supply a `defaultValue` prop. For example:

Code

```
  render: function() {
    return <input type="text" defaultValue="Hello!" />;
  }
```

This example will function much like the **Controlled Components** example above.

Likewise, `<input type="checkbox">` and `<input type="radio">` support `defaultChecked`, and `<select>` supports `defaultValue`.

Note:

The `defaultValue` and `defaultChecked` props are only used during initial render. If you need to update the value in a subsequent render, you will need to use a controlled component.

Advanced Topics

Why Controlled Components?

Using form components such as `<input>` in React presents a challenge that is absent when writing traditional form HTML. For example, in HTML:

Code

```
<input type="text" name="title" value="Untitled" />
```

This renders an input *initialized* with the value, `Untitled`. When the user updates the input, the node's `value` *property* will change. However, `node.getAttribute('value')` will still return the value used at initialization time, `Untitled`.

Unlike HTML, React components must represent the state of the view at any point in time and not only at initialization time. For example, in React:

Code

```
render: function() {
  return <input type="text" name="title" value="Untitled" />;
}
```

Since this method describes the view at any point in time, the value of the text input should *always* be `Untitled`.

Why Textarea Value?

In HTML, the value of `<textarea>` is usually set using its children:

Code

```
<!-- antipattern: DO NOT DO THIS! -->
<textarea name="description">This is the description.</textarea>
```

For HTML, this easily allows developers to supply multiline values. However, since React is JavaScript, we do not have string limitations and can use `\n` if we want newlines. In a world where we have `value` and `defaultValue`, it is ambiguous what role children play. For this reason, you should not use children when setting `<textarea>` values:

Code

```
<textarea name="description" value="This is a description." />
```

If you *do* decide to use children, they will behave like `defaultValue`.

Why Select Value?

The selected `<option>` in an HTML `<select>` is normally specified through that option's `selected` attribute. In React, in order to make components easier to manipulate, the following format is adopted instead:

Code

```
<select value="B">
  <option value="A">Apple</option>
  <option value="B">Banana</option>
  <option value="C">Cranberry</option>
</select>
```

To make an uncontrolled component, `defaultValue` is used instead.

Note:

You can pass an array into the `value` attribute, allowing you to select multiple options in a `select` tag: `<select multiple={true} value={['B', 'C']}>`.

Imperative operations

If you need to imperatively perform an operation, you have to obtain a reference to the DOM node. For instance, if you want to imperatively submit a form, one approach would be to attach a `ref` to the `form` element and manually call `form.submit()`.

← Prev Next →

Working With the Browser

Edit on GitHub

React provides powerful abstractions that free you from touching the DOM directly in most cases, but sometimes you simply need to access the underlying API, perhaps to work with a third-party library or existing code.

The Virtual DOM

React is very fast because it never talks to the DOM directly. React maintains a fast in-memory representation of the DOM. `render()` methods actually return a *description* of the DOM, and React can compare this description with the in-memory representation to compute the fastest way to update the browser.

Additionally, React implements a full synthetic event system such that all event objects are guaranteed to conform to the W3C spec despite browser quirks, and everything bubbles consistently and efficiently across browsers. You can even use some HTML5 events in older browsers that don't ordinarily support them!

Most of the time you should stay within React's "faked browser" world since it's more performant and easier to reason about. However, sometimes you simply need to access the underlying API, perhaps to work with a third-party library like a jQuery plugin. React provides escape hatches for you to use the underlying DOM API directly.

Refs and findDOMNode()

To interact with the browser, you'll need a reference to a DOM node. You can attach a `ref` to any element, which allows you to reference the **backing instance** of the component. This is useful if you need to invoke imperative functions on the component, or want to access the underlying DOM nodes. To learn more about refs, including ways to use them effectively, see our refs to components documentation.

Component Lifecycle

Components have three main parts of their lifecycle:

- **Mounting:** A component is being inserted into the DOM.
- **Updating:** A component is being re-rendered to determine if the DOM should be updated.
- **Unmounting:** A component is being removed from the DOM.

React provides lifecycle methods that you can specify to hook into this process. We provide **will** methods, which are called right before something happens, and **did** methods which are called right after something happens.

Mounting

- `getInitialState(): object` is invoked before a component is mounted. Stateful components should implement this and return the initial state data.
- `componentWillMount()` is invoked immediately before mounting occurs.
- `componentDidMount()` is invoked immediately after mounting occurs. Initialization that requires DOM nodes should go here.

Updating

- `componentWillReceiveProps(object nextProps)` is invoked when a mounted component receives new props. This method should be used to compare `this.props` and `nextProps` to perform state transitions using `this.setState()`.
- `shouldComponentUpdate(object nextProps, object nextState): boolean` is invoked when a component decides whether any changes warrant an update to the DOM. Implement this as an optimization to compare `this.props` with `nextProps` and `this.state` with `nextState` and return `false` if React should skip updating.

- `componentWillUpdate(object nextProps, object nextState)` is invoked immediately before updating occurs. You cannot call `this.setState()` here.
- `componentDidUpdate(object prevProps, object prevState)` is invoked immediately after updating occurs.

Unmounting

- `componentWillUnmount()` is invoked immediately before a component is unmounted and destroyed. Cleanup should go here.

Mounted Methods

Mounted composite components also support the following method:

- `component.forceUpdate()` can be invoked on any mounted component when you know that some deeper aspect of the component's state has changed without using `this.setState()`.

Browser Support

React supports most popular browsers, including Internet Explorer 9 and above.

(We don't support older browsers that don't support ES5 methods, but you may find that your apps do work in older browsers if polyfills such as es5-shim and es5-sham are included in the page. You're on your own if you choose to take this path.)

← Prev

Next →

Language Tooling

Edit on GitHub

ES2015 with JSX

In-browser JSX Transform

If you like using JSX, Babel 5 provided an in-browser ES2015 and JSX transformer for development called browser.js that can be included from CDNJS. Include a `<script type="text/babel">` tag to engage the JSX transformer.

Note:

The in-browser JSX transformer is fairly large and results in extraneous computation client-side that can be avoided. Do not use it in production — see the next section.

Productionizing: Precompiled JSX

If you have npm, you can run `npm install -g babel-cli`. Babel has built-in support for React v0.12+. Tags are automatically transformed to their equivalent `React.createElement(...)`, `displayName` is automatically inferred and added to all `React.createClass` calls.

This tool will translate files that use JSX syntax to plain JavaScript files that can run directly in the browser. It will also watch directories for you and automatically transform files when they are changed; for example: `babel --watch src/ --out-dir lib/`.

Beginning with Babel 6, there are no transforms included by default. This means that options must be specified when running the `babel` command, or a `.babelrc` must specify options. Additional packages must also be installed which bundle together a number of transforms, called presets. The most common use when working with React will be to include the `es2015` and `react` presets. More information about the changes to Babel can be found in their blog post announcing Babel 6.

Here is an example of what you will do if using ES2015 syntax and React:

Code

```
npm install babel-preset-es2015 babel-preset-react
babel --presets es2015,react --watch src/ --out-dir lib/
```

By default JSX files with a `.js` extension are transformed. Run `babel --help` for more information on how to use Babel.

Example output:

Code
```
var HelloMessage = React.createClass({
  render: function() {
    return <div>Hello {this.props.name}</div>;
  }
});

ReactDOM.render(<HelloMessage name="John" />, mountNode);
$ babel test.js
"use strict";

var HelloMessage = React.createClass({
  displayName: "HelloMessage",

  render: function render() {
    return React.createElement(
      "div",
      null,
      "Hello ",
      this.props.name
    );
  }
});

ReactDOM.render(React.createElement(HelloMessage, { name: "John" }
), mountNode);
```

Helpful Open-Source Projects

The open-source community has built tools that integrate JSX with several editors and build systems. See JSX integrations for the full list.

Flow

Flow is a JavaScript type checker released by Facebook, and it supports JSX. For more info, checkout the Flow homepage.

TypeScript

TypeScript is a type-checker and transpiler that supports type-checking React and JSX. For more info, check out their guide on getting started with React and Webpack, or learn more about TypeScript's JSX support.

To learn more about TypeScript in general, visit the TypeScript homepage.

← Prev Next →

Package Management

Edit on GitHub

CDN-hosted React

We provide CDN-hosted versions of React on our download page. These pre-built files use the UMD module format. Dropping them in with a simple `<script>` tag will inject the `React` and `ReactDOM` globals into your environment. It should also work out-of-the-box in CommonJS and AMD environments.

Using React from npm

You can use React with a CommonJS module system like browserify or webpack. Use the `react` and `react-dom` npm packages.

Code

```
// main.js
var React = require('react');
var ReactDOM = require('react-dom');

ReactDOM.render(
  <h1>Hello, world!</h1>,
  document.getElementById('example')
);
```

Configure babel with a `.babelrc` file:

Code

```
 { "presets": ["react"] }
```

Note:

If you are using ES2015, you will want to also use the `babel-preset-es2015` package.

To install React DOM and build your bundle with browserify:

Code

```
$ npm install --save react react-dom babelify babel-preset-react
$ browserify -t [ babelify ] main.js -o bundle.js
```

To install React DOM and build your bundle with webpack:

Code

```
$ npm install --save react react-dom babel-preset-react babel-loader babel-core
$ webpack main.js bundle.js --module-bind 'js=babel-loader'
```

Note:

If you are using ES2015, you will want to also use the `babel-preset-es2015` package.

Note: by default, React will be in development mode, which is slower, and not advised for production. To use React in production mode, set the environment variable `NODE_ENV` to `production` (using envify or webpack's DefinePlugin). For example:

Code

```
new webpack.DefinePlugin({
  "process.env": {
    NODE_ENV: JSON.stringify("production")
  }
});
```

Update your HTML file as below:

Code

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Hello React!</title>
    <script src="build/react.js"></script>
    <script src="build/react-dom.js"></script>
    <!-- No need for Babel! -->
  </head>
  <body>
    <div id="example"></div>
    <script src="build/helloworld.js"></script>
  </body>
</html>
```

Using React from Bower

Bower is a package manager optimized for the front-end development. If multiple packages depend on a package - jQuery for example - Bower will download jQuery just once. This is known as a flat dependency graph and it helps reduce page load. For more info, visit http://bower.io/

If you'd like to use bower, it's as easy as:

Code

```
bower install --save react
```

Code

```
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Hello React!</title>
    <script src="bower_components/react/react.js"></script>
    <script src="bower_components/react/react-dom.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.34/
browser.min.js"></script>
  </head>
  <body>
    <div id="example"></div>
    <script type="text/babel">
      ReactDOM.render(
        <h1>Hello, world!</h1>,
        document.getElementById('example')
      );
    </script>
  </body>
</html>
```

Using master

We have instructions for building from `master` in our GitHub repository.

← Prev   Next →

Server-side Environments

Edit on GitHub

One of the great things about React is that it doesn't require the DOM as a dependency, which means it is possible to render a React application on the server and send the HTML markup down to the client. There are a few things that React expects, so this guide will help you get started in your preferred environment.

Node.js

Node.js is a popular JavaScript runtime that comes with an extensive core library and support for installing packages from npm to expand on the basic functionality. As we've described elsewhere in the documentation, you can install `react` and `react-dom` from npm.

Example:

Code
```
var ReactDOMServer = require('react-dom/server');

var element = React.createElement('div', null, 'Hello World!');
console.log(ReactDOMServer.renderToString(element));
```

If you use JSX, you may want to pre-compile your components. Alternatively you may want to consider using Babel's require hook or `babel-node`.

Note:

Some versions of Node.js have an `Object.assign` implementation that does not preserve key order. This can cause errors when validating the markup, creating a warning that says "React attempted to reuse markup in a container but the checksum was invalid". If you run into this issue, you can override `Object.assign` to use a polyfill that preserves key order. For more details, see Issue #6451.

C#

Support for server-side component rendering and JSX compilation (via Babel) in a .NET Framework / ASP.NET environment is provided through our ReactJS.NET project.

Nashorn

Nashorn is a lightweight high-performance JavaScript runtime that runs within the JVM. React should run out of the box in Java 8+.

Example:

Code

```
import java.io.InputStream;
import java.io.FileReader;

import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;

public class ReactRender
{
  public static void main(String[] args) throws ScriptException, IOException {
    ScriptEngine nashorn = new ScriptEngineManager().getEngineByName("nashorn");

    // These files can be downloaded as a part of the starter kit
    // from https://facebook.github.io/react
    nashorn.eval(new FileReader("path/to/react.js"));
    nashorn.eval(new FileReader("path/to/react-dom-server.js"));

    System.out.println(nashorn.eval(
      "ReactDOMServer.renderToString(" +
        "React.createElement('div', null, 'Hello World!')" +
      ");"
    ));
  }
}
```

If your application uses npm packages, or you want to transform JSX in Nashorn, you will need to do some additional environment setup. The following resources may be helpful in getting you started:

- http://winterbe.com/posts/2015/02/16/isomorphic-react-webapps-on-the-jvm/
- https://github.com/nodyn/jvm-npm
- https://gist.github.com/aesteve/883e0fd33390451cb8eb

Note:

Using Babel within Nashorn will require Java 8u72+, as update 72 fixed JDK-8135190.

← Prev Next →

Add-ons

Edit on GitHub

The React add-ons are a collection of useful utility modules for building React apps. **These should be considered experimental** and tend to change more often than the core.

- `TransitionGroup` and `CSSTransitionGroup`, for dealing with animations and transitions that are usually not simple to implement, such as before a component's removal.
- `LinkedStateMixin`, to simplify the coordination between user's form input data and the component's state.
- `cloneWithProps`, to make shallow copies of React components and change their props.
- `createFragment`, to create a set of externally-keyed children.
- `update`, a helper function that makes dealing with immutable data in JavaScript easier.
- `PureRenderMixin`, a performance booster under certain situations.
- `shallowCompare`, a helper function that performs a shallow comparison for props and state in a component to decide if a component should update.

The add-ons below are in the development (unminified) version of React only:

- `TestUtils`, simple helpers for writing test cases.
- `Perf`, a performance profiling tool for finding optimization opportunities.

To get the add-ons, install them individually from npm (e.g., `npm install react-addons-pure-render-mixin`). We don't support using the addons if you're not using npm.

← Prev Next →

Animation

Edit on GitHub

React provides a `ReactTransitionGroup` add-on component as a low-level API for animation, and a `ReactCSSTransitionGroup` for easily implementing basic CSS animations and transitions.

High-level API: `ReactCSSTransitionGroup`

`ReactCSSTransitionGroup` is based on `ReactTransitionGroup` and is an easy way to perform CSS transitions and animations when a React component enters or leaves the DOM. It's inspired by the excellent ng-animate library.

Getting Started

`ReactCSSTransitionGroup` is the interface to `ReactTransitions`. This is a simple element that wraps all of the components you are interested in animating. Here's an example where we fade list items in and out.

Code

```
var ReactCSSTransitionGroup = require('react-addons-css-transition-group');

var TodoList = React.createClass({
  getInitialState: function() {
    return {items: ['hello', 'world', 'click', 'me']};
  },
  handleAdd: function() {
    var newItems =
      this.state.items.concat([prompt('Enter some text')]);
    this.setState({items: newItems});
  },
  handleRemove: function(i) {
    var newItems = this.state.items.slice();
    newItems.splice(i, 1);
    this.setState({items: newItems});
  },
  render: function() {
    var items = this.state.items.map(function(item, i) {
      return (
        <div key={item} onClick={this.handleRemove.bind(this, i)}>
          {item}
        </div>
      );
    }.bind(this));
    return (
      <div>
        <button onClick={this.handleAdd}>Add Item</button>
        <ReactCSSTransitionGroup transitionName="example" transitionEnterTimeout={500} tra
          {items}
        </ReactCSSTransitionGroup>
      </div>
    );
  }
});
```

Note:

You must provide the `key` attribute for all children of `ReactCSSTransitionGroup`, even when only rendering a single item. This is how React will determine which children have entered, left, or stayed.

In this component, when a new item is added to `ReactCSSTransitionGroup` it will get the `example-enter` CSS class and the `example-enter-active` CSS class added in the next tick. This is a convention based on the `transitionName` prop.

You can use these classes to trigger a CSS animation or transition. For example, try adding this CSS and adding a new list item:

Code

```
.example-enter {
  opacity: 0.01;
}

.example-enter.example-enter-active {
  opacity: 1;
  transition: opacity 500ms ease-in;
}

.example-leave {
  opacity: 1;
}

.example-leave.example-leave-active {
  opacity: 0.01;
  transition: opacity 300ms ease-in;
}
```

You'll notice that animation durations need to be specified in both the CSS and the render method; this tells React when to remove the animation classes from the element and -- if it's leaving -- when to remove the element from the DOM.

Animate Initial Mounting

`ReactCSSTransitionGroup` provides the optional prop `transitionAppear`, to add an extra transition phase at the initial mount of the component. There is generally no transition phase at the initial mount as the default value of `transitionAppear` is `false`. The following is an example which passes the prop `transitionAppear` with the value `true`.

Code

```
  render: function() {
    return (
      <ReactCSSTransitionGroup transitionName="example" transitionAppear={true} transition
        <h1>Fading at Initial Mount</h1>
      </ReactCSSTransitionGroup>
    );
  }
```

During the initial mount `ReactCSSTransitionGroup` will get the `example-appear` CSS class and the `example-appear-active` CSS class added in the next tick.

Code

```
.example-appear {
  opacity: 0.01;
}

.example-appear.example-appear-active {
  opacity: 1;
  transition: opacity .5s ease-in;
}
```

At the initial mount, all children of the `ReactCSSTransitionGroup` will `appear` but not `enter`. However, all children later added to an existing `ReactCSSTransitionGroup` will `enter` but not `appear`.

Note:

The prop `transitionAppear` was added to `ReactCSSTransitionGroup` in version `0.13`. To maintain backwards compatibility, the default value is set to `false`.

Custom Classes

It is also possible to use custom class names for each of the steps in your transitions. Instead of passing a string into transitionName you can pass an object containing either the `enter` and `leave` class names, or an object containing the `enter`, `enter-active`, `leave-active`, and `leave` class names. If only the enter and leave classes are provided, the enter-active and leave-active classes will be determined by appending '-active' to the end of the class name. Here are two examples using custom classes:

Code
```
    <ReactCSSTransitionGroup
      transitionName={ {
        enter: 'enter',
        enterActive: 'enterActive',
        leave: 'leave',
        leaveActive: 'leaveActive',
        appear: 'appear',
        appearActive: 'appearActive'
      } }>
      {item}
    </ReactCSSTransitionGroup>

    <ReactCSSTransitionGroup
      transitionName={ {
        enter: 'enter',
        leave: 'leave',
        appear: 'appear'
      } }>
      {item2}
    </ReactCSSTransitionGroup>
    ...
```

Animation Group Must Be Mounted To Work

In order for it to apply transitions to its children, the `ReactCSSTransitionGroup` must already be mounted in the DOM or the prop `transitionAppear` must be set to `true`. The example below would not work, because the `ReactCSSTransitionGroup` is being mounted along with the new item, instead of the new item being mounted within it. Compare this to the Getting Started section above to see the difference.

Code

```
    var items = this.state.items.map(function(item, i) {
      return (
        <div key={item} onClick={this.handleRemove.bind(this, i)}>
          <ReactCSSTransitionGroup transitionName="example">
            {item}
          </ReactCSSTransitionGroup>
        </div>
      );
    }, this);
    return (
      <div>
        <button onClick={this.handleAdd}>Add Item</button>
        {items}
      </div>
    );
  }
```

Animating One or Zero Items

In the example above, we rendered a list of items into `ReactCSSTransitionGroup`. However, the children of `ReactCSSTransitionGroup` can also be one or zero items. This makes it possible to animate a single element entering or leaving. Similarly, you can animate a new element replacing the current element. For example, we can implement a simple image carousel like this:

Code

```
var ReactCSSTransitionGroup = require('react-addons-css-transition-group');

var ImageCarousel = React.createClass({
  propTypes: {
    imageSrc: React.PropTypes.string.isRequired
  },
  render: function() {
    return (
      <div>
        <ReactCSSTransitionGroup transitionName="carousel" transitionEnterTimeout={300} tra
          <img src={this.props.imageSrc} key={this.props.imageSrc} />
        </ReactCSSTransitionGroup>
      </div>
    );
  }
});
```

Disabling Animations

You can disable animating `enter` or `leave` animations if you want. For example, sometimes you may want an `enter` animation and no `leave` animation, but `ReactCSSTransitionGroup` waits for an animation to complete before removing your DOM node. You can add `transitionEnter={false}` or `transitionLeave={false}` props to `ReactCSSTransitionGroup` to disable these animations.

Note:

When using `ReactCSSTransitionGroup`, there's no way for your components to be notified when a transition has ended or to perform any more complex logic around animation. If you want more fine-grained control, you can use the lower-level `ReactTransitionGroup` API which provides the hooks you need to do custom transitions.

Low-level API: `ReactTransitionGroup`

`ReactTransitionGroup` is the basis for animations. It is accessible from `require('react-addons-transition-group')`. When children are declaratively added or removed from it (as in the example above) special lifecycle hooks are called on them.

`componentWillAppear(callback)`

This is called at the same time as `componentDidMount()` for components that are initially mounted in a `TransitionGroup`. It will block other animations from occurring until `callback` is called. It is only called on the initial render of a `TransitionGroup`.

`componentDidAppear()`

This is called after the `callback` function that was passed to `componentWillAppear` is called.

`componentWillEnter(callback)`

This is called at the same time as `componentDidMount()` for components added to an existing `TransitionGroup`. It will block other animations from occurring until `callback` is called. It will not be called on the initial render of a `TransitionGroup`.

`componentDidEnter()`

This is called after the `callback` function that was passed to `componentWillEnter` is called.

`componentWillLeave(callback)`

This is called when the child has been removed from the `ReactTransitionGroup`. Though the child has been removed, `ReactTransitionGroup` will keep it in the DOM until `callback` is called.

`componentDidLeave()`

This is called when the `willLeave` `callback` is called (at the same time as `componentWillUnmount`).

Rendering a Different Component

By default `ReactTransitionGroup` renders as a `span`. You can change this behavior by providing a `component` prop. For example, here's how you would render a `<ul>`:

Code

```
<ReactTransitionGroup component="ul">
  ...
</ReactTransitionGroup>
```

Any additional, user-defined, properties will become properties of the rendered component. For example, here's how you would render a `<ul>` with CSS class:

Code

```
<ReactTransitionGroup component="ul" className="animated-list">
  ...
</ReactTransitionGroup>
```

Every DOM component that React can render is available for use. However, `component` does not need to be a DOM component. It can be any React component you want; even ones you've written yourself! Just write `component={List}` and your component will receive `this.props.children`.

Rendering a Single Child

People often use `ReactTransitionGroup` to animate mounting and unmounting of a single child such as a collapsible panel. Normally `ReactTransitionGroup` wraps all its children in a `span` (or a custom `component` as described above). This is because any React component has to return a single root element, and `ReactTransitionGroup` is no exception to this rule.

However if you only need to render a single child inside `ReactTransitionGroup`, you can completely avoid wrapping it in a `<span>` or any other DOM component. To do this, create a custom component that renders the first child passed to it directly:

Code
```
  render: function() {
    var children = React.Children.toArray(this.props.children);
    return children[0] || null;
  }
});
```

Now you can specify `FirstChild` as the `component` prop in `<ReactTransitionGroup>` props and avoid any wrappers in the result DOM:

Code
```
<ReactTransitionGroup component={FirstChild}>
  {someCondition ? <MyComponent /> : null}
</ReactTransitionGroup>
```

This only works when you are animating a single child in and out, such as a collapsible panel. This approach wouldn't work when animating multiple children or replacing the single child with another child, such as an image carousel. For an image carousel, while the current image is animating out, another image will animate in, so `<ReactTransitionGroup>` needs to give them a common DOM parent. You can't avoid the wrapper for multiple children, but you can customize the wrapper with the `component` prop as described above.

← Prev Next →

Two-Way Binding Helpers

Edit on GitHub

`ReactLink` is an easy way to express two-way binding with React.

Note:

ReactLink is deprecated as of React v15. The recommendation is to explicitly set the value and change handler, instead of using ReactLink.

In React, data flows one way: from owner to child. This is because data only flows one direction in the Von Neumann model of computing. You can think of it as "one-way data binding."

However, there are lots of applications that require you to read some data and flow it back into your program. For example, when developing forms, you'll often want to update some React `state` when you receive user input. Or perhaps you want to perform layout in JavaScript and react to changes in some DOM element size.

In React, you would implement this by listening to a "change" event, read from your data source (usually the DOM) and call `setState()` on one of your components. "Closing the data flow loop" explicitly leads to more understandable and easier-to-maintain programs. See our forms documentation for more information.

Two-way binding -- implicitly enforcing that some value in the DOM is always consistent with some React `state` -- is concise and supports a wide variety of applications. We've provided `ReactLink`: syntactic sugar for setting up the common data flow loop pattern described above, or "linking" some data source to React `state`.

Note:

`ReactLink` is just a thin wrapper and convention around the `onChange`/`setState()` pattern. It doesn't fundamentally change how data flows in your React application.

ReactLink: Before and After

Here's a simple form example without using `ReactLink`:

Code

```
var NoLink = React.createClass({
  getInitialState: function() {
    return {message: 'Hello!'};
  },
  handleChange: function(event) {
    this.setState({message: event.target.value});
  },
  render: function() {
    var message = this.state.message;
    return <input type="text" value={message} onChange={this.handleChange} />;
  }
});
```

This works really well and it's very clear how data is flowing, however, with a lot of form fields it could get a bit verbose. Let's use `ReactLink` to save us some typing:

Code

```
var LinkedStateMixin = require('react-addons-linked-state-mixin');

var WithLink = React.createClass({
  mixins: [LinkedStateMixin],
  getInitialState: function() {
    return {message: 'Hello!'};
  },
  render: function() {
    return <input type="text" valueLink={this.linkState('message')} />;
  }
});
```

`LinkedStateMixin` adds a method to your React component called `linkState()`. `linkState()` returns a `ReactLink` object which contains the current value of the React state and a callback to change it.

`ReactLink` objects can be passed up and down the tree as props, so it's easy (and explicit) to set up two-way binding between a component deep in the hierarchy and state that lives higher in the hierarchy.

Note that checkboxes have a special behavior regarding their `value` attribute, which is the value that will be sent on form submit if the checkbox is checked (defaults to `on`). The `value` attribute is not updated when the checkbox is checked or unchecked. For checkboxes, you should use `checkedLink` instead of `valueLink`: `<input type="checkbox" checkedLink={this.linkState('booleanValue')} />`

Under the Hood

There are two sides to `ReactLink`: the place where you create the `ReactLink` instance and the place where you use it. To prove how simple `ReactLink` is, let's rewrite each side separately to be more explicit.

ReactLink Without LinkedStateMixin

Code

```
var WithoutMixin = React.createClass({
  getInitialState: function() {
    return {message: 'Hello!'};
  },
  handleChange: function(newValue) {
    this.setState({message: newValue});
  },
  render: function() {
    var valueLink = {
      value: this.state.message,
      requestChange: this.handleChange
    };
    return <input type="text" valueLink={valueLink} />;
  }
});
```

As you can see, `ReactLink` objects are very simple objects that just have a `value` and `requestChange` prop. And `LinkedStateMixin` is similarly simple: it just populates those fields with a value from `this.state` and a callback that calls `this.setState()`.

ReactLink Without valueLink

Code

```
var WithoutLink = React.createClass({
  mixins: [LinkedStateMixin],
  getInitialState: function() {
    return {message: 'Hello!'};
  },
  render: function() {
    var valueLink = this.linkState('message');
    var handleChange = function(e) {
      valueLink.requestChange(e.target.value);
    };
    return <input type="text" value={valueLink.value} onChange={handleChange} /
>;
  }
});
```

The `valueLink` prop is also quite simple. It simply handles the `onChange` event and calls `this.props.valueLink.requestChange()` and also uses `this.props.valueLink.value` instead of `this.props.value`. That's it!

← Prev Next →

Test Utilities

Edit on GitHub

`ReactTestUtils` makes it easy to test React components in the testing framework of your choice (we use Jest).

Code

```
var ReactTestUtils = require('react-addons-test-utils');
```

Note:

Airbnb has released a testing utility called Enzyme, which makes it easy to assert, manipulate, and traverse your React Components' output. If you're deciding on a unit testing library, it's worth checking out: http://airbnb.io/enzyme/

Simulate

Code

```
Simulate.{eventName}(
  DOMElement element,
  [object eventData]
)
```

Simulate an event dispatch on a DOM node with optional `eventData` event data. **This is possibly the single most useful utility in `ReactTestUtils`.**

Clicking an element

Code

```
// <button ref="button">...</button>
var node = this.refs.button;
ReactTestUtils.Simulate.click(node);
```

Changing the value of an input field and then pressing ENTER.

Code

```
// <input ref="input" />
var node = this.refs.input;
node.value = 'giraffe';
ReactTestUtils.Simulate.change(node);
ReactTestUtils.Simulate.keyDown(node, {key: "Enter", keyCode: 13, which: 13});
```

Note that you will have to provide any event property that you're using in your component (e.g. keyCode, which, etc...) as React is not creating any of these for you.

`Simulate` has a method for every event that React understands.

renderIntoDocument

Code

```
ReactComponent renderIntoDocument(
  ReactElement instance
)
```

Render a component into a detached DOM node in the document. **This function requires a DOM.**

Note:

You will need to have `window`, `window.document` and `window.document.createElement` globally available **before** you import React. Otherwise React will think it can't access the DOM and methods like `setState` won't work.

mockComponent

Code

```
object mockComponent(
  function componentClass,
  [string mockTagName]
)
```

Pass a mocked component module to this method to augment it with useful methods that allow it to be used as a dummy React component. Instead of rendering as usual, the component will become a simple `<div>` (or other tag if `mockTagName` is provided) containing any provided children.

isElement

Code

```
boolean isElement(
  ReactElement element
)
```

Returns `true` if `element` is any ReactElement.

isElementOfType

Code

```
boolean isElementOfType(
  ReactElement element,
  function componentClass
)
```

Returns `true` if `element` is a ReactElement whose type is of a React `componentClass`.

isDOMComponent

Code

```
boolean isDOMComponent(
  ReactComponent instance
)
```

Returns `true` if `instance` is a DOM component (such as a `<div>` or `<span>`).

isCompositeComponent

Code

```
boolean isCompositeComponent(
  ReactComponent instance
)
```

Returns `true` if `instance` is a composite component (created with `React.createClass()`).

isCompositeComponentWithType

Code
```
  ReactComponent instance,
  function componentClass
)
```

Returns `true` if `instance` is a composite component (created with `React.createClass()`) whose type is of a React `componentClass`.

findAllInRenderedTree

Code
```
array findAllInRenderedTree(
  ReactComponent tree,
  function test
)
```

Traverse all components in `tree` and accumulate all components where `test(component)` is `true`. This is not that useful on its own, but it's used as a primitive for other test utils.

scryRenderedDOMComponentsWithClass

Code
```
array scryRenderedDOMComponentsWithClass(
  ReactComponent tree,
  string className
)
```

Finds all DOM elements of components in the rendered tree that are DOM components with the class name matching `className`.

findRenderedDOMComponentWithClass

Code
```
DOMElement findRenderedDOMComponentWithClass(
  ReactComponent tree,
  string className
)
```

Like `scryRenderedDOMComponentsWithClass()` but expects there to be one result, and returns that one result, or throws exception if there is any other number of matches besides one.

scryRenderedDOMComponentsWithTag

Code
```
  ReactComponent tree,
  string tagName
)
```

Finds all DOM elements of components in the rendered tree that are DOM components with the tag name matching `tagName`.

findRenderedDOMComponentWithTag

Code
```
DOMElement findRenderedDOMComponentWithTag(
  ReactComponent tree,
  string tagName
)
```

Like `scryRenderedDOMComponentsWithTag()` but expects there to be one result, and returns that one result, or throws exception if there is any other number of matches besides one.

scryRenderedComponentsWithType

Code
```
array scryRenderedComponentsWithType(
  ReactComponent tree,
  function componentClass
)
```

Finds all instances of components with type equal to `componentClass`.

findRenderedComponentWithType

Code
```
ReactComponent findRenderedComponentWithType(
  ReactComponent tree,
  function componentClass
)
```

Same as `scryRenderedComponentsWithType()` but expects there to be one result and returns that one result, or throws exception if there is any other number of matches besides one.

Shallow rendering

Shallow rendering is an experimental feature that lets you render a component "one level deep" and assert facts about what its render method returns, without worrying about the behavior of child components, which are not instantiated or rendered. This does not require a DOM.

Code

```
ReactShallowRenderer createRenderer()
```

Call this in your tests to create a shallow renderer. You can think of this as a "place" to render the component you're testing, where it can respond to events and update itself.

Code

```
shallowRenderer.render(
  ReactElement element
)
```

Similar to `ReactDOM.render`.

Code

```
ReactElement shallowRenderer.getRenderOutput()
```

After `render` has been called, returns shallowly rendered output. You can then begin to assert facts about the output. For example, if your component's render method returns:

Code

```
<div>
  <span className="heading">Title</span>
  <Subcomponent foo="bar" />
</div>
```

Then you can assert:

Code

```
var renderer = ReactTestUtils.createRenderer();
result = renderer.getRenderOutput();
expect(result.type).toBe('div');
expect(result.props.children).toEqual([
  <span className="heading">Title</span>,
  <Subcomponent foo="bar" />
]);
```

Shallow testing currently has some limitations, namely not supporting refs. We're releasing this feature early and would appreciate the React community's feedback on how it should evolve.

← Prev   Next →

Cloning ReactElements

Edit on GitHub

Note: `cloneWithProps` is deprecated. Use React.cloneElement instead.

In rare situations, you may want to create a copy of a React element with different props from those of the original element. One example is cloning the elements passed into `this.props.children` and rendering them with different props:

Code

```
var cloneWithProps = require('react-addons-clone-with-props');

var _makeBlue = function(element) {
  return cloneWithProps(element, {style: {color: 'blue'}});
};

var Blue = React.createClass({
  render: function() {
    var blueChildren = React.Children.map(this.props.children, _makeBlue);
    return <div>{blueChildren}</div>;
  }
});

ReactDOM.render(
  <Blue>
    <p>This text is blue.</p>
  </Blue>,
  document.getElementById('container')
);
```

`cloneWithProps` does not transfer `key` or `ref` to the cloned element. `className` and `style` props are automatically merged.

← Prev

Next →

Keyed Fragments

Edit on GitHub

In most cases, you can use the `key` prop to specify keys on the elements you're returning from `render`. However, this breaks down in one situation: if you have two sets of children that you need to reorder, there's no way to put a key on each set without adding a wrapper element.

That is, if you have a component such as:

Code

```
  propTypes: {
    // `leftChildren` and `rightChildren` can be a string, element, array, etc.
    leftChildren: React.PropTypes.node,
    rightChildren: React.PropTypes.node,

    swapped: React.PropTypes.bool
  },
  render: function() {
    var children;
    if (this.props.swapped) {
      children = [this.props.rightChildren, this.props.leftChildren];
    } else {
      children = [this.props.leftChildren, this.props.rightChildren];
    }
    return <div>{children}</div>;
  }
});
```

The children will unmount and remount as you change the `swapped` prop because there aren't any keys marked on the two sets of children.

To solve this problem, you can use the `createFragment` add-on to give keys to the sets of children.

`Array<ReactNode> createFragment(object children)`

Instead of creating arrays, we write:

Code

```
var createFragment = require('react-addons-create-fragment');

if (this.props.swapped) {
  children = createFragment({
    right: this.props.rightChildren,
    left: this.props.leftChildren
  });
} else {
  children = createFragment({
    left: this.props.leftChildren,
    right: this.props.rightChildren
  });
}
```

The keys of the passed object (that is, `left` and `right`) are used as keys for the entire set of children, and the order of the object's keys is used to determine the order of the rendered children. With this change, the two sets of children will be properly reordered in the DOM without unmounting.

The return value of `createFragment` should be treated as an opaque object; you can use the `React.Children` helpers to loop through a fragment but should not access it directly. Note also that we're relying on the JavaScript engine preserving object enumeration order here, which is not guaranteed by the spec but is implemented by all major browsers and VMs for objects with non-numeric keys.

Note:

In the future, `createFragment` may be replaced by an API such as

Code

```
return (
  <div>
    <x:frag key="right">{this.props.rightChildren}</x:frag>,
    <x:frag key="left">{this.props.leftChildren}</x:frag>
  </div>
);
```

allowing you to assign keys directly in JSX without adding wrapper elements.

← Prev Next →

Immutability Helpers

Edit on GitHub

React lets you use whatever style of data management you want, including mutation. However, if you can use immutable data in performance-critical parts of your application it's easy to implement a fast `shouldComponentUpdate()` method to significantly speed up your app.

Dealing with immutable data in JavaScript is more difficult than in languages designed for it, like Clojure. However, we've provided a simple immutability helper, `update()`, that makes dealing with this type of data much easier, *without* fundamentally changing how your data is represented. You can also take a look at Facebook's Immutable-js and the Advanced Performance section for more detail on Immutable-js.

The main idea

If you mutate data like this:

Code

```
myData.x.y.z = 7;
// or...
myData.a.b.push(9);
```

You have no way of determining which data has changed since the previous copy has been overwritten. Instead, you need to create a new copy of `myData` and change only the parts of it that need to be changed. Then you can compare the old copy of `myData` with the new one in `shouldComponentUpdate()` using triple-equals:

Code

```
var newData = deepCopy(myData);
newData.x.y.z = 7;
newData.a.b.push(9);
```

Unfortunately, deep copies are expensive, and sometimes impossible. You can alleviate this by only copying objects that need to be changed and by reusing the objects that haven't changed. Unfortunately, in today's JavaScript this can be cumbersome:

Code

```
var newData = extend(myData, {
  x: extend(myData.x, {
    y: extend(myData.x.y, {z: 7}),
  }),
  a: extend(myData.a, {b: myData.a.b.concat(9)})
});
```

While this is fairly performant (since it only makes a shallow copy of `log n` objects and reuses the rest), it's a big pain to write. Look at all the repetition! This is not only annoying, but also provides a large surface area for bugs.

`update()` provides simple syntactic sugar around this pattern to make writing this code easier. This code becomes:

Code

```
var update = require('react-addons-update');

var newData = update(myData, {
  x: {y: {z: {$set: 7}}},
  a: {b: {$push: [9]}}
});
```

While the syntax takes a little getting used to (though it's inspired by MongoDB's query language) there's no redundancy, it's statically analyzable and it's not much more typing than the mutative version.

The `$`-prefixed keys are called *commands*. The data structure they are "mutating" is called the *target*.

Available commands

- `{$push: array}` `push()` all the items in `array` on the target.

- `{$unshift: array}` `unshift()` all the items in `array` on the target.
- `{$splice: array of arrays}` for each item in `arrays` call `splice()` on the target with the parameters provided by the item.
- `{$set: any}` replace the target entirely.
- `{$merge: object}` merge the keys of `object` with the target.
- `{$apply: function}` passes in the current value to the function and updates it with the new returned value.

Examples

Simple push

Code

```
var initialArray = [1, 2, 3];
var newArray = update(initialArray, {$push: [4]}); // => [1, 2, 3, 4]
```

`initialArray` is still `[1, 2, 3]`.

Nested collections

Code

```
var collection = [1, 2, {a: [12, 17, 15]}];
var newCollection = update(collection, {2: {a: {$splice: [[1, 1, 13, 14]]}}});
// => [1, 2, {a: [12, 13, 14, 15]}]
```

This accesses `collection`'s index `2`, key `a`, and does a splice of one item starting from index `1` (to remove `17`) while inserting `13` and `14`.

Updating a value based on its current one

Code

```
var obj = {a: 5, b: 3};
var newObj = update(obj, {b: {$apply: function(x) {return x * 2;}}});
// => {a: 5, b: 6}
// This is equivalent, but gets verbose for deeply nested collections:
var newObj2 = update(obj, {b: {$set: obj.b * 2}});
```

(Shallow) merge

Code
```
var newObj = update(obj, {$merge: {b: 6, c: 7}}); // => {a: 5, b: 6, c: 7}
```

← Prev Next →

PureRenderMixin

Edit on GitHub

If your React component's render function is "pure" (in other words, it renders the same result given the same props and state), you can use this mixin for a performance boost in some cases.

Example:

Code
```
var PureRenderMixin = require('react-addons-pure-render-mixin');
React.createClass({
  mixins: [PureRenderMixin],

  render: function() {
    return <div className={this.props.className}>foo</div>;
  }
});
```

Example using ES6 class syntax:

Code
```
import PureRenderMixin from 'react-addons-pure-render-mixin';
class FooComponent extends React.Component {
  constructor(props) {
    super(props);
    this.shouldComponentUpdate = PureRenderMixin.shouldComponentUpdate.bind(this);
  }

  render() {
    return <div className={this.props.className}>foo</div>;
  }
}
```

Under the hood, the mixin implements shouldComponentUpdate, in which it compares the current props and state with the next ones and returns `false` if the equalities pass.

Note:

This only shallowly compares the objects. If these contain complex data structures, it may produce false-negatives for deeper differences. Only mix into components which have simple props and state, or use `forceUpdate()` when you know deep data structures have changed. Or, consider using immutable objects to facilitate fast comparisons of nested data.

Furthermore, `shouldComponentUpdate` skips updates for the whole component subtree. Make sure all the children components are also "pure".

← Prev Next →

Performance Tools

Edit on GitHub

React is usually quite fast out of the box. However, in situations where you need to squeeze every ounce of performance out of your app, it provides a shouldComponentUpdate hook where you can add optimization hints to React's diff algorithm.

In addition to giving you an overview of your app's overall performance, ReactPerf is a profiling tool that tells you exactly where you need to put these hooks.

See these two articles by the Benchling Engineering Team for a in-depth introduction to performance tooling: "Performance Engineering with React" and "A Deep Dive into React Perf Debugging"!

Development vs. Production Builds

If you're benchmarking or seeing performance problems in your React apps, make sure you're testing with the minified production build. The development build includes extra warnings that are helpful when building your apps, but it is slower due to the extra bookkeeping it does.

However, the perf tools described on this page only work when using the development build of React. Therefore, the profiler only serves to indicate the *relatively* expensive parts of your app.

General API

The `Perf` object documented here is exposed as `require('react-addons-perf')` and can be used with React in development mode only. You should not include this bundle when building your app for production.

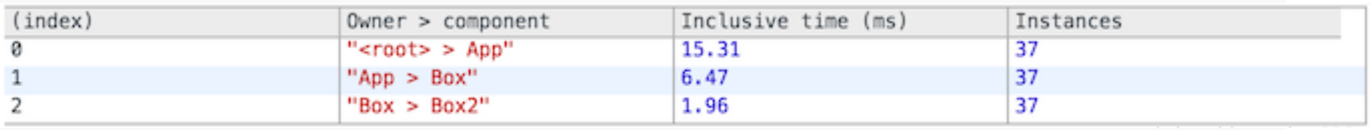
`Perf.start()` and `Perf.stop()`

Start/stop the measurement. The React operations in-between are recorded for analyses below. Operations that took an insignificant amount of time are ignored.

After stopping, you will need `Perf.getLastMeasurements()` (described below) to get the measurements.

`Perf.printInclusive(measurements)`

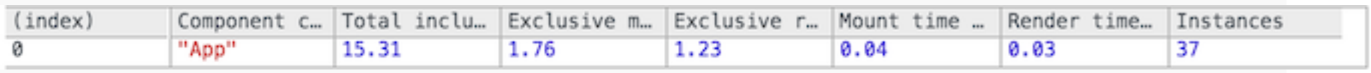
Prints the overall time taken. If no argument's passed, defaults to all the measurements from the last recording. This prints a nicely formatted table in the console, like so:



(index)	Owner > component	Inclusive time (ms)	Instances
0	"<root> > App"	15.31	37
1	"App > Box"	6.47	37
2	"Box > Box2"	1.96	37

`Perf.printExclusive(measurements)`

"Exclusive" times don't include the times taken to mount the components: processing props, `getInitialState`, call `componentWillMount` and `componentDidMount`, etc.

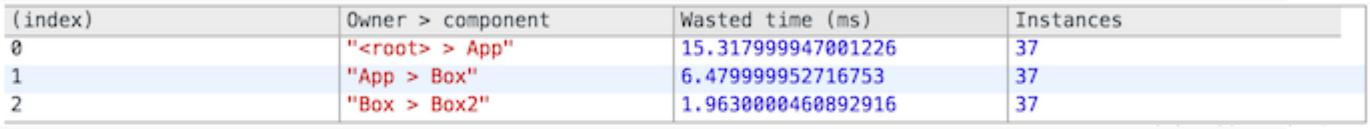


(index)	Component c…	Total inclu…	Exclusive m…	Exclusive r…	Mount time …	Render time…	Instances
0	"App"	15.31	1.76	1.23	0.04	0.03	37

`Perf.printWasted(measurements)`

The most useful part of the profiler.

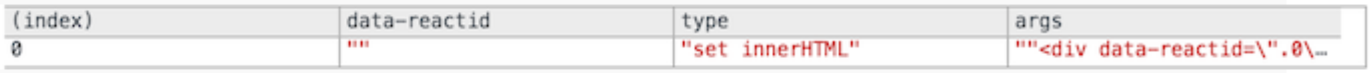
"Wasted" time is spent on components that didn't actually render anything, e.g. the render stayed the same, so the DOM wasn't touched.



(index)	Owner > component	Wasted time (ms)	Instances
0	"<root> > App"	15.317999947001226	37
1	"App > Box"	6.479999952716753	37
2	"Box > Box2"	1.9630000460892916	37

`Perf.printOperations(measurements)`

Prints the underlying DOM manipulations, e.g. "set innerHTML" and "remove".



(index)	data-reactid	type	args
0	""	"set innerHTML"	""<div data-reactid=\".0\…

`Perf.printDOM(measurements)`

This method has been renamed to `printOperations()` which is described in the previous paragraph. Currently `printDOM()` still exists as an alias but it prints a deprecation warning and will eventually be removed.

Advanced API

The above print methods use `Perf.getLastMeasurements()` to pretty-print the result.

`Perf.getLastMeasurements()`

Get the opaque data structure describing measurements from the last start-stop session. You can save it and pass it to the methods above to analyze past measurements.

Don't rely on the exact format of the return value because it may change in minor releases. We will update the documentation if the return value format becomes a supported part of the public API.

← Prev Next →

Shallow Compare

Edit on GitHub

`shallowCompare` is a helper function to achieve the same functionality as `PureRenderMixin` while using ES6 classes with React.

If your React component's render function is "pure" (in other words, it renders the same result given the same props and state), you can use this helper function for a performance boost in some cases.

Example:

Code

```
var shallowCompare = require('react-addons-shallow-compare');
export class SampleComponent extends React.Component {
  shouldComponentUpdate(nextProps, nextState) {
    return shallowCompare(this, nextProps, nextState);
  }

  render() {
    return <div className={this.props.className}>foo</div>;
  }
}
```

`shallowCompare` performs a shallow equality check on the current `props` and `nextProps` objects as well as the current `state` and `nextState` objects.
It does this by iterating on the keys of the objects being compared and returning true when the values of a key in each object are not strictly equal.

`shallowCompare` returns `true` if the shallow comparison for props or state fails and therefore the component should update.
`shallowCompare` returns `false` if the shallow comparison for props and state both pass and therefore the component does not need to update.

← Prev Next →

Advanced Performance

Edit on GitHub

One of the first questions people ask when considering React for a project is whether their application will be as fast and responsive as an equivalent non-React version. The idea of re-rendering an entire subtree of components in response to every state change makes people wonder whether this process negatively impacts performance. React uses several clever techniques to minimize the number of costly DOM operations required to update the UI.

Use the production build

If you're benchmarking or seeing performance problems in your React apps, make sure you're testing with the minified production build. The development build includes extra warnings that are helpful when building your apps, but it is slower due to the extra bookkeeping it does.

Avoiding reconciling the DOM

React makes use of a *virtual DOM*, which is a descriptor of a DOM subtree rendered in the browser. This parallel representation allows React to avoid creating DOM nodes and accessing existing ones, which is slower than operations on JavaScript objects. When a component's props or state change, React decides whether an actual DOM update is necessary by constructing a new virtual DOM and comparing it to the old one. Only in the case they are not equal, will React reconcile the DOM, applying as few mutations as possible.

On top of this, React provides a component lifecycle function, `shouldComponentUpdate`, which is triggered before the re-rendering process starts (virtual DOM comparison and possible eventual DOM reconciliation), giving the developer the ability to short circuit this process. The default implementation of this function returns `true`, leaving React to perform the update:

Code

```
shouldComponentUpdate: function(nextProps, nextState) {
  return true;
}
```

Keep in mind that React will invoke this function pretty often, so the implementation has to be fast.

Say you have a messaging application with several chat threads. Suppose only one of the threads has changed. If we implement `shouldComponentUpdate` on the `ChatThread` component, React can skip the rendering step for the other threads:

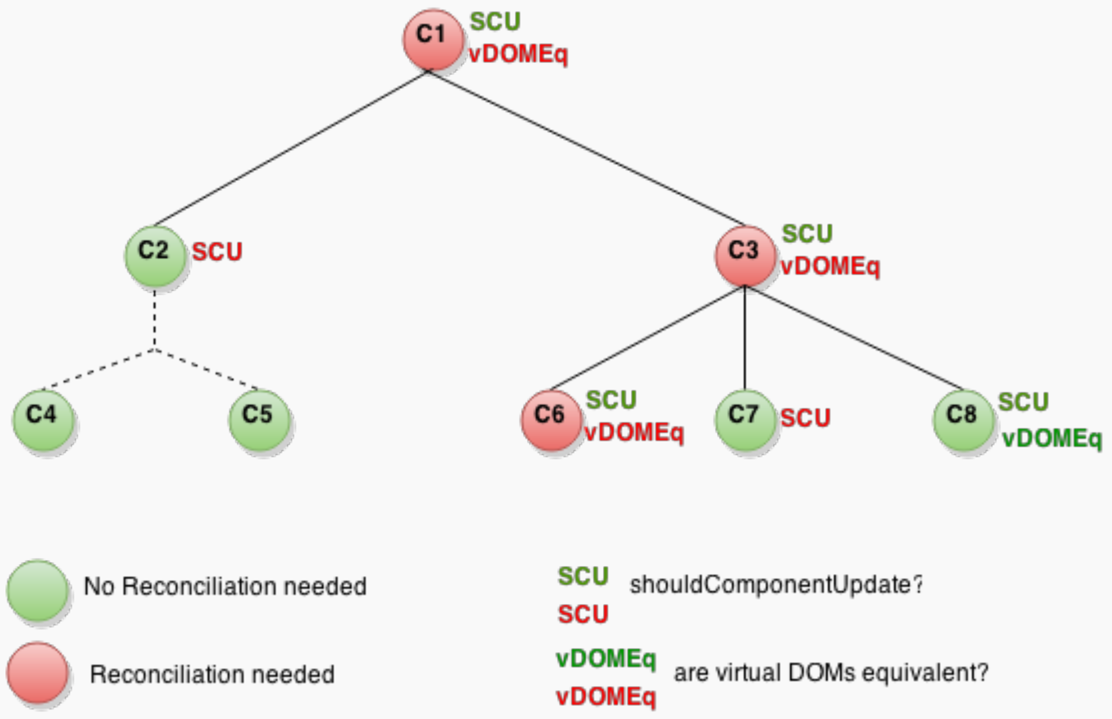
Code

```
shouldComponentUpdate: function(nextProps, nextState) {
  // TODO: return whether or not current chat thread is
  // different to former one.
}
```

So, in summary, React avoids carrying out expensive DOM operations required to reconcile subtrees of the DOM by allowing the user to short circuit the process using `shouldComponentUpdate`, and, for those which should update, by comparing virtual DOMs.

shouldComponentUpdate in action

Here's a subtree of components. For each one is indicated what `shouldComponentUpdate` returned and whether or not the virtual DOMs were equivalent. Finally, the circle's color indicates whether the component had to be reconciled or not.



In the example above, since `shouldComponentUpdate` returned `false` for the subtree rooted at C2, React had no need to generate the new virtual DOM, and therefore, it neither needed to reconcile the DOM. Note that React didn't even have to invoke `shouldComponentUpdate` on C4 and C5.

For C1 and C3 `shouldComponentUpdate` returned `true`, so React had to go down to the leaves and check them. For C6 it returned `true`; since the virtual DOMs weren't equivalent it had to reconcile the DOM. The last interesting case is C8. For this node React had to compute the virtual DOM, but since it was equal to the old one, it didn't have to reconcile it's DOM.

Note that React only had to do DOM mutations for C6, which was inevitable. For C8, it bailed out by comparing the virtual DOMs, and for C2's subtree and C7, it didn't even have to compute the virtual DOM as we bailed out on `shouldComponentUpdate`.

So, how should we implement `shouldComponentUpdate`? Say that you have a component that just renders a string value:

Code
```
  propTypes: {
    value: React.PropTypes.string.isRequired
  },

  render: function() {
    return <div>{this.props.value}</div>;
  }
});
```

We could easily implement `shouldComponentUpdate` as follows:

Code
```
shouldComponentUpdate: function(nextProps, nextState) {
  return this.props.value !== nextProps.value;
}
```

So far so good, dealing with such simple props/state structures is easy. We could even generalize an implementation based on shallow equality and mix it into components. In fact, React already provides such implementation: PureRenderMixin.

But what if your components' props or state are mutable data structures? Say the prop the component receives, instead of being a string like `'bar'`, is a JavaScript object that contains a string such as, `{ foo: 'bar' }`:

Code
```
React.createClass({
  propTypes: {
    value: React.PropTypes.object.isRequired
  },

  render: function() {
    return <div>{this.props.value.foo}</div>;
  }
});
```

The implementation of `shouldComponentUpdate` we had before wouldn't always work as expected:

Code
```
// assume this.props.value is { foo: 'bar' }
// assume nextProps.value is { foo: 'bar' },
// but this reference is different to this.props.value
this.props.value !== nextProps.value; // true
```

The problem is `shouldComponentUpdate` will return `true` when the prop actually didn't change. To fix this, we could come up with this alternative implementation:

Code

```
  return this.props.value.foo !== nextProps.value.foo;
}
```

Basically, we ended up doing a deep comparison to make sure we properly track changes. In terms of performance, this approach is pretty expensive. It doesn't scale as we would have to write different deep equality code for each model. On top of that, it might not even work if we don't carefully manage object references. Say this component is used by a parent:

Code

```
React.createClass({
  getInitialState: function() {
    return { value: { foo: 'bar' } };
  },

  onClick: function() {
    var value = this.state.value;
    value.foo += 'bar'; // ANTI-PATTERN!
    this.setState({ value: value });
  },

  render: function() {
    return (
      <div>
        <InnerComponent value={this.state.value} />
        <a onClick={this.onClick}>Click me</a>
      </div>
    );
  }
});
```

The first time the inner component gets rendered, it will have `{ foo: 'bar' }` as the value prop. If the user clicks on the anchor, the parent component's state will get updated to `{ value: { foo: 'barbar' } }`, triggering the re-rendering process of the inner component, which will receive `{ foo: 'barbar' }` as the new value for the prop.

The problem is that since the parent and inner components share a reference to the same object, when the object gets mutated on line 2 of the `onClick` function, the prop the inner component had will change. So, when the re-rendering process starts, and `shouldComponentUpdate` gets invoked, `this.props.value.foo` will be equal to `nextProps.value.foo`, because in fact, `this.props.value` references the same object as `nextProps.value`.

Consequently, since we'll miss the change on the prop and short circuit the re-rendering process, the UI won't get updated from `'bar'` to `'barbar'`.

Immutable-js to the rescue

Immutable-js is a JavaScript collections library written by Lee Byron, which Facebook recently open-sourced. It provides *immutable persistent* collections via *structural sharing*. Let's see what these properties mean:

- *Immutable*: once created, a collection cannot be altered at another point in time.
- *Persistent*: new collections can be created from a previous collection and a mutation such as set. The original collection is still valid after the new collection is created.
- *Structural Sharing*: new collections are created using as much of the same structure as the original collection as possible, reducing copying to a minimum to achieve space efficiency and acceptable performance. If the new collection is equal to the original, the original is often returned.

Immutability makes tracking changes cheap; a change will always result in a new object so we only need to check if the reference to the object has changed. For example, in this regular JavaScript code:

Code

```
var x = { foo: "bar" };
var y = x;
y.foo = "baz";
x === y; // true
```

Although `y` was edited, since it's a reference to the same object as `x`, this comparison returns `true`. However, this code could be written using immutable-js as follows:

Code

```
var SomeRecord = Immutable.Record({ foo: null });
var x = new SomeRecord({ foo: 'bar'  });
var y = x.set('foo', 'baz');
x === y; // false
```

In this case, since a new reference is returned when mutating `x`, we can safely assume that `x` has changed.

Another possible way to track changes could be doing dirty checking by having a flag set by setters. A problem with this approach is that it forces you to use setters and, either write a lot of additional code, or somehow instrument your classes. Alternatively, you could deep copy the object just before the mutations and deep compare to determine whether there was a change or not. A problem with this approach is both deepCopy and deepCompare are expensive operations.

So, Immutable data structures provides you a cheap and less verbose way to track changes on objects, which is all we need to implement `shouldComponentUpdate`. Therefore, if we model props and state attributes using the abstractions provided by immutable-js we'll be able to use `PureRenderMixin` and get a nice boost in perf.

Immutable-js and Flux

If you're using Flux, you should start writing your stores using immutable-js. Take a look at the full API.

Let's see one possible way to model the thread example using Immutable data structures. First, we need to define a `Record` for each of the entities we're trying to model. Records are just immutable containers that hold values for a specific set of fields:

Code

```
var User = Immutable.Record({
  id: undefined,
  name: undefined,
  email: undefined
});

var Message = Immutable.Record({
  timestamp: new Date(),
  sender: undefined,
  text: ''
});
```

The `Record` function receives an object that defines the fields the object has and its default values.

The messages *store* could keep track of the users and messages using two lists:

Code

```
this.users = Immutable.List();
this.messages = Immutable.List();
```

It should be pretty straightforward to implement functions to process each *payload* type. For instance, when the store sees a payload representing a new message, we can just create a new record and append it to the messages list:

Code

```
this.messages = this.messages.push(new Message({
  timestamp: payload.timestamp,
  sender: payload.sender,
  text: payload.text
});
```

Note that since the data structures are immutable, we need to assign the result of the push function to `this.messages`.

On the React side, if we also use immutable-js data structures to hold the components' state, we could mix `PureRenderMixin` into all our components and short circuit the re-rendering process.

← Prev Next →

Context

Edit on GitHub

One of React's biggest strengths is that it's easy to track the flow of data through your React components. When you look at a component, you can easily see exactly which props are being passed in which makes your apps easy to reason about.

Occasionally, you want to pass data through the component tree without having to pass the props down manually at every level. React's "context" feature lets you do this.

Note:

Context is an advanced and experimental feature. The API is likely to change in future releases.

Most applications will never need to use context. Especially if you are just getting started with React, you likely do not want to use context. Using context will make your code harder to understand because it makes the data flow less clear. It is similar to using global variables to pass state through your application.

If you have to use context, use it sparingly.

Regardless of whether you're building an application or a library, try to isolate your use of context to a small area and avoid using the context API directly when possible so that it's easier to upgrade when the API changes.

Passing info automatically through a tree

Suppose you have a structure like:

Code
```
  render() {
    return (
      <button style={{background: this.props.color}}>
        {this.props.children}
      </button>
    );
  }
}

class Message extends React.Component {
  render() {
    return (
      <div>
        {this.props.text} <Button color={this.props.color}>Delete</Button>
      </div>
    );
  }
}

class MessageList extends React.Component {
  render() {
    const color = "purple";
    const children = this.props.messages.map((message) =>
      <Message text={message.text} color={color} />
    );
    return <div>{children}</div>;
  }
}
```

In this example, we manually thread through a `color` prop in order to style the `Button` and `Message` components appropriately. Theming is a good example of when you might want an entire subtree to have access to some piece of information (a color). Using context, we can pass this through the tree automatically:

Code

```
  render() {
    return (
      <button style={{background: this.context.color}}>
        {this.props.children}
      </button>
    );
  }
}

Button.contextTypes = {
  color: React.PropTypes.string
};

class Message extends React.Component {
  render() {
    return (
      <div>
        {this.props.text} <Button>Delete</Button>
      </div>
    );
  }
}

class MessageList extends React.Component {
  getChildContext() {
    return {color: "purple"};
  }

  render() {
    const children = this.props.messages.map((message) =>
      <Message text={message.text} />
    );
    return <div>{children}</div>;
  }
}

MessageList.childContextTypes = {
  color: React.PropTypes.string
};
```

By adding `childContextTypes` and `getChildContext` to `MessageList` (the context provider), React passes the information down automatically and any component in the subtree (in this case, `Button`) can access it by defining `contextTypes`.

If `contextTypes` is not defined, then `context` will be an empty object.

Parent-child coupling

Context can also let you build an API such as:

Code

```
<Menu>
  <MenuItem>aubergine</MenuItem>
  <MenuItem>butternut squash</MenuItem>
  <MenuItem>clementine</MenuItem>
</Menu>
```

By passing down the relevant info in the `Menu` component, each `MenuItem` can communicate back to the containing `Menu` component.

Before you build components with this API, consider if there are cleaner alternatives. We're fond of simply passing the items as an array in cases like this:

Code

```
<Menu items={['aubergine', 'butternut squash', 'clementine']} />
```

Recall that you can also pass entire React components in props if you'd like to.

Referencing context in lifecycle methods

If `contextTypes` is defined within a component, the following lifecycle methods will receive an additional parameter, the `context` object:

Code

```
void componentWillReceiveProps(
  object nextProps, object nextContext
)

boolean shouldComponentUpdate(
  object nextProps, object nextState, object nextContext
)

void componentWillUpdate(
  object nextProps, object nextState, object nextContext
)

void componentDidUpdate(
  object prevProps, object prevState, object prevContext
)
```

Referencing context in stateless functional components

Stateless functional components are also able to reference `context` if `contextTypes` is defined as a property of the function. The following code shows the `Button` component above written as a stateless functional component.

Code

```
const Button = ({children}, context) =>
  <button style={{background: context.color}}>
    {children}
  </button>;

Button.contextTypes = {color: React.PropTypes.string};
```

Updating context

The `getChildContext` function will be called when the state or props changes. In order to update data in the context, trigger a local state update with `this.setState`. This will trigger a new context and changes will be received by the children.

Code

```
  constructor(props) {
    super(props);
    this.state = {type:'desktop'};
  }

  getChildContext() {
    return {type: this.state.type};
  }

  componentDidMount() {
    const checkMediaQuery = () => {
      const type = window.matchMedia("(min-width: 1025px)").matches ? 'desktop'
 : 'mobile';
      if (type !== this.state.type) {
        this.setState({type});
      }
    };

    window.addEventListener('resize', checkMediaQuery);
    checkMediaQuery();
  }

  render() {
    return this.props.children;
  }
}

MediaQuery.childContextTypes = {
  type: React.PropTypes.string
};
```

When not to use context

Just as global variables are best avoided when writing clear code, you should avoid using context in most cases. In particular, think twice before using it to "save typing" and using it instead of passing explicit props.

The best use cases for context are for implicitly passing down the logged-in user, the current language, or theme information. All of these might otherwise be true globals, but context lets you scope them to a single React subtree.

Do not use context to pass your model data through components. Threading your data through the tree explicitly is much easier to understand. Using context makes your components more coupled and less reusable, because they behave differently depending on where they're rendered.

Known limitations

If a context value provided by a component changes, descendants that use that value won't update if an intermediate parent returns `false` from `shouldComponentUpdate`. See issue #2517 for more details.

← Prev

Top-Level API

Edit on GitHub

React

`React` is the entry point to the React library. If you're using one of the prebuilt packages it's available as a global; if you're using CommonJS modules you can `require()` it.

React.Component

Code

```
class Component
```

This is the base class for React Components when they're defined using ES6 classes. See Reusable Components for how to use ES6 classes with React. For what methods are actually provided by the base class, see the Component API.

React.createClass

Code

```
ReactClass createClass(object specification)
```

Create a component class, given a specification. A component implements a `render` method which returns **one single** child. That child may have an arbitrarily deep child structure. One thing that makes components different than standard prototypal classes is that you don't need to call new on them. They are convenience wrappers that construct backing instances (via new) for you.

For more information about the specification object, see Component Specs and Lifecycle.

React.createElement

Code

```
    string/ReactClass type,
    [object props],
    [children ...]
  )
```

Create and return a new `ReactElement` of the given type. The type argument can be either an html tag name string (eg. 'div', 'span', etc), or a `ReactClass` (created via `React.createClass`).

React.cloneElement

Code

```
ReactElement cloneElement(
  ReactElement element,
  [object props],
  [children ...]
)
```

Clone and return a new `ReactElement` using `element` as the starting point. The resulting element will have the original element's props with the new props merged in shallowly. New children will replace existing children. Unlike `React.addons.cloneWithProps`, `key` and `ref` from the original element will be preserved. There is no special behavior for merging any props (unlike `cloneWithProps`). See the v0.13 RC2 blog post for additional details.

React.createFactory

Code

```
factoryFunction createFactory(
  string/ReactClass type
)
```

Return a function that produces ReactElements of a given type. Like `React.createElement`, the type argument can be either an html tag name string (eg. 'div', 'span', etc), or a `ReactClass`.

React.isValidElement

Code

```
boolean isValidElement(* object)
```

Verifies the object is a ReactElement.

React.DOM

`React.DOM` provides convenience wrappers around `React.createElement` for DOM components. These should only be used when not using JSX. For example, `React.DOM.div(null, 'Hello World!')`

React.PropTypes

`React.PropTypes` includes types that can be used with a component's `propTypes` object to validate props being passed to your components. For more information about `propTypes`, see Reusable Components.

React.Children

`React.Children` provides utilities for dealing with the `this.props.children` opaque data structure.

React.Children.map

Code

```
array React.Children.map(object children, function fn [, object thisArg])
```

Invoke `fn` on every immediate child contained within `children` with `this` set to `thisArg`. If `children` is a keyed fragment or array it will be traversed: `fn` will never be passed the container objects. If children is `null` or `undefined` returns `null` or `undefined` rather than an array.

React.Children.forEach

Code

```
React.Children.forEach(object children, function fn [, object thisArg])
```

Like `React.Children.map()` but does not return an array.

React.Children.count

Code

```
number React.Children.count(object children)
```

Return the total number of components in `children`, equal to the number of times that a callback passed to `map` or `forEach` would be invoked.

React.Children.only

Code

```
object React.Children.only(object children)
```

Return the only child in `children`. Throws otherwise.

React.Children.toArray

Code

```
array React.Children.toArray(object children)
```

Return the `children` opaque data structure as a flat array with keys assigned to each child. Useful if you want to manipulate collections of children in your render methods, especially if you want to reorder or slice `this.props.children` before passing it down.

ReactDOM

The `react-dom` package provides DOM-specific methods that can be used at the top level of your app and as an escape hatch to get outside of the React model if you need to. Most of your components should not need to use this module.

ReactDOM.render

Code

```
render(
  ReactElement element,
  DOMElement container,
  [function callback]
)
```

Render a ReactElement into the DOM in the supplied `container` and return a reference to the component (or returns `null` for stateless components).

If the ReactElement was previously rendered into `container`, this will perform an update on it and only mutate the DOM as necessary to reflect the latest React component.

If the optional callback is provided, it will be executed after the component is rendered or updated.

Note:

`ReactDOM.render()` controls the contents of the container node you pass in. Any existing DOM elements inside are replaced when first called. Later calls use React's DOM diffing algorithm for efficient updates.

`ReactDOM.render()` does not modify the container node (only modifies the children of the container). In the future, it may be possible to insert a component to an existing DOM node without overwriting the existing children.

`ReactDOM.render()` currently returns a reference to the root `ReactComponent` instance. However, using this return value is legacy and should be avoided because future versions of React may render components asynchronously in some cases. If you need a reference to the root `ReactComponent` instance, the preferred solution is to attach a callback ref to the root element.

ReactDOM.unmountComponentAtNode

Code

```
boolean unmountComponentAtNode(DOMElement container)
```

Remove a mounted React component from the DOM and clean up its event handlers and state. If no component was mounted in the container, calling this function does nothing. Returns `true` if a component was unmounted and `false` if there was no component to unmount.

ReactDOM.findDOMNode

Code

```
DOMElement findDOMNode(ReactComponent component)
```

If this component has been mounted into the DOM, this returns the corresponding native browser DOM element. This method is useful for reading values out of the DOM, such as form field values and performing DOM measurements. **In most cases, you can attach a ref to the DOM node and avoid using `findDOMNode` at all.** When `render` returns `null` or `false`, `findDOMNode` returns `null`.

Note:

`findDOMNode()` is an escape hatch used to access the underlying DOM node. In most cases, use of this escape hatch is discouraged because it pierces the component abstraction.

`findDOMNode()` only works on mounted components (that is, components that have been placed in the DOM). If you try to call this on a component that has not been mounted yet (like calling `findDOMNode()` in `render()` on a component that has yet to be created) an exception will be thrown.

`findDOMNode()` cannot be used on stateless components.

ReactDOMServer

The `react-dom/server` package allows you to render your components on the server.

ReactDOMServer.renderToString

Code

```
string renderToString(ReactElement element)
```

Render a ReactElement to its initial HTML. This should only be used on the server. React will return an HTML string. You can use this method to generate HTML on the server and send the markup down on the initial request for faster page loads and to allow search engines to crawl your pages for SEO purposes.

If you call `ReactDOM.render()` on a node that already has this server-rendered markup, React will preserve it and only attach event handlers, allowing you to have a very performant first-load experience.

ReactDOMServer.renderToStaticMarkup

Code

```
string renderToStaticMarkup(ReactElement element)
```

Similar to `renderToString`, except this doesn't create extra DOM attributes such as `data-react-id`, that React uses internally. This is useful if you want to use React as a simple static page generator, as stripping away the extra attributes can save lots of bytes.

Next →

Component API

Edit on GitHub

React.Component

Instances of a React Component are created internally in React when rendering. These instances are reused in subsequent renders, and can be accessed in your component methods as `this`. The only way to get a handle to a React Component instance outside of React is by storing the return value of `ReactDOM.render`. Inside other Components, you may use refs to achieve the same result.

setState

Code

```
void setState(
  function|object nextState,
  [function callback]
)
```

Performs a shallow merge of nextState into current state. This is the primary method you use to trigger UI updates from event handlers and server request callbacks.

The first argument can be an object (containing zero or more keys to update) or a function (of state and props) that returns an object containing keys to update.

Here is the simple object usage:

Code

```
setState({mykey: 'my new value'});
```

It's also possible to pass a function with the signature `function(state, props)`. This can be useful in some cases when you want to enqueue an atomic update that consults the previous value of state+props before setting any values. For instance, suppose we wanted to increment a value in state:

Code

```
setState(function(previousState, currentProps) {
  return {myInteger: previousState.myInteger + 1};
});
```

The second (optional) parameter is a callback function that will be executed once `setState` is completed and the component is re-rendered.

Notes:

NEVER mutate `this.state` directly, as calling `setState()` afterwards may replace the mutation you made. Treat `this.state` as if it were immutable.

`setState()` does not immediately mutate `this.state` but creates a pending state transition. Accessing `this.state` after calling this method can potentially return the existing value.

There is no guarantee of synchronous operation of calls to `setState` and calls may be batched for performance gains.

`setState()` will always trigger a re-render unless conditional rendering logic is implemented in `shouldComponentUpdate()`. If mutable objects are being used and the logic cannot be implemented in `shouldComponentUpdate()`, calling `setState()` only when the new state differs from the previous state will avoid unnecessary re-renders.

replaceState

Code

```
void replaceState(
  object nextState,
  [function callback]
)
```

Like `setState()` but deletes any pre-existing state keys that are not in nextState.

Note:

This method is not available on ES6 `class` components that extend `React.Component`. It may be removed entirely in a future version of React.

forceUpdate

Code

```
void forceUpdate(
  [function callback]
)
```

By default, when your component's state or props change, your component will re-render. However, if these change implicitly (eg: data deep within an object changes without changing the object itself) or if your `render()` method depends on some other data, you can tell React that it needs to re-run `render()` by calling `forceUpdate()`.

Calling `forceUpdate()` will cause `render()` to be called on the component, skipping `shouldComponentUpdate()`. This will trigger the normal lifecycle methods for child components, including the `shouldComponentUpdate()` method of each child. React will still only update the DOM if the markup changes.

Normally you should try to avoid all uses of `forceUpdate()` and only read from `this.props` and `this.state` in `render()`. This makes your component "pure" and your application much simpler and more efficient.

isMounted

Code

```
boolean isMounted()
```

`isMounted()` returns `true` if the component is rendered into the DOM, `false` otherwise. You can use this method to guard asynchronous calls to `setState()` or `forceUpdate()`.

Note:

This method is not available on ES6 `class` components that extend `React.Component`. It will likely be removed entirely in a future version of React, so you might as well start migrating away from isMounted() now.

← Prev   Next →

Component Specs and Lifecycle

Edit on GitHub

Component Specifications

When creating a component class by invoking `React.createClass()`, you should provide a specification object that contains a `render` method and can optionally contain other lifecycle methods described here.

Note:

It is also possible to use plain JavaScript classes as component classes. These classes can implement most of the same methods, though there are some differences. For more information about these differences, please read our documentation about ES6 classes.

render

Code

```
ReactElement render()
```

The `render()` method is required.

When called, it should examine `this.props` and `this.state` and return a single child element. This child element can be either a virtual representation of a native DOM component (such as `<div />` or `React.DOM.div()`) or another composite component that you've defined yourself.

You can also return `null` or `false` to indicate that you don't want anything rendered. Behind the scenes, React renders a `<noscript>` tag to work with our current diffing algorithm. When returning `null` or `false`, `ReactDOM.findDOMNode(this)` will return `null`.

The `render()` function should be *pure*, meaning that it does not modify component state, it returns the same result each time it's invoked, and it does not read from or write to the DOM or otherwise interact with the browser (e.g., by using `setTimeout`). If you need to interact with the browser, perform your work in `componentDidMount()` or the other lifecycle methods instead. Keeping `render()` pure makes server rendering more practical and makes components easier to think about.

getInitialState

Code

```
object getInitialState()
```

Invoked once before the component is mounted. The return value will be used as the initial value of `this.state`.

getDefaultProps

Code

Invoked once and cached when the class is created. Values in the mapping will be set on `this.props` if that prop is not specified by the parent component (i.e. using an `in` check).

This method is invoked before any instances are created and thus cannot rely on `this.props`. In addition, be aware that any complex objects returned by `getDefaultProps()` will be shared across instances, not copied.

propTypes

Code

```
object propTypes
```

The `propTypes` object allows you to validate props being passed to your components. For more information about `propTypes`, see Reusable Components.

mixins

Code

```
array mixins
```

The `mixins` array allows you to use mixins to share behavior among multiple components. For more information about mixins, see Reusable Components.

statics

Code

```
object statics
```

The `statics` object allows you to define static methods that can be called on the component class. For example:

Code

```
var MyComponent = React.createClass({
  statics: {
    customMethod: function(foo) {
      return foo === 'bar';
    }
  },
  render: function() {
  }
});

MyComponent.customMethod('bar');  // true
```

Methods defined within this block are *static*, meaning that you can run them before any component instances are created, and the methods do not have access to the props or state of your components. If you want to check the value of props in a static method, have the caller pass in the props as an argument to the static method.

displayName

Code

```
string displayName
```

The `displayName` string is used in debugging messages. JSX sets this value automatically; see JSX in Depth.

Lifecycle Methods

Various methods are executed at specific points in a component's lifecycle.

Mounting: componentWillMount

Code

```
void componentWillMount()
```

Invoked once, both on the client and server, immediately before the initial rendering occurs. If you call `setState` within this method, `render()` will see the updated state and will be executed only once despite the state change.

Mounting: componentDidMount

Code

```
void componentDidMount()
```

Invoked once, only on the client (not on the server), immediately after the initial rendering occurs. At this point in the lifecycle, you can access any refs to your children (e.g., to access the underlying DOM representation). The `componentDidMount()` method of child components is invoked before that of parent components.

If you want to integrate with other JavaScript frameworks, set timers using `setTimeout` or `setInterval`, or send AJAX requests, perform those operations in this method.

Updating: componentWillReceiveProps

Code

```
  object nextProps
)
```

Invoked when a component is receiving new props. This method is not called for the initial render.

Use this as an opportunity to react to a prop transition before `render()` is called by updating the state using `this.setState()`. The old props can be accessed via `this.props`. Calling `this.setState()` within this function will not trigger an additional render.

Code

```
componentWillReceiveProps: function(nextProps) {
  this.setState({
    likesIncreasing: nextProps.likeCount > this.props.likeCount
  });
}
```

Note:

One common mistake is for code executed during this lifecycle method to assume that props have changed. To understand why this is invalid, read A implies B does not imply B implies A

There is no analogous method `componentWillReceiveState`. An incoming prop transition may cause a state change, but the opposite is not true. If you need to perform operations in response to a state change, use `componentWillUpdate`.

Updating: shouldComponentUpdate

Code

```
boolean shouldComponentUpdate(
  object nextProps, object nextState
)
```

Invoked before rendering when new props or state are being received. This method is not called for the initial render or when `forceUpdate` is used.

Use this as an opportunity to `return false` when you're certain that the transition to the new props and state will not require a component update.

Code

```
shouldComponentUpdate: function(nextProps, nextState) {
  return nextProps.id !== this.props.id;
}
```

If `shouldComponentUpdate` returns false, then `render()` will be completely skipped until the next state change. In addition, `componentWillUpdate` and `componentDidUpdate` will not be called.

By default, `shouldComponentUpdate` always returns `true` to prevent subtle bugs when `state` is mutated in place, but if you are careful to always treat `state` as immutable and to read only from `props` and `state` in `render()` then you can override `shouldComponentUpdate` with an implementation that compares the old props and state to their replacements.

If performance is a bottleneck, especially with dozens or hundreds of components, use `shouldComponentUpdate` to speed up your app.

Updating: componentWillUpdate

Code

```
void componentWillUpdate(
  object nextProps, object nextState
)
```

Invoked immediately before rendering when new props or state are being received. This method is not called for the initial render.

Use this as an opportunity to perform preparation before an update occurs.

Note:

You *cannot* use `this.setState()` in this method. If you need to update state in response to a prop change, use `componentWillReceiveProps` instead.

Updating: componentDidUpdate

Code

```
void componentDidUpdate(
  object prevProps, object prevState
)
```

Invoked immediately after the component's updates are flushed to the DOM. This method is not called for the initial render.

Use this as an opportunity to operate on the DOM when the component has been updated.

Unmounting: componentWillUnmount

Code

```
void componentWillUnmount()
```

Invoked immediately before a component is unmounted from the DOM.

Perform any necessary cleanup in this method, such as invalidating timers or cleaning up any DOM elements that were created in `componentDidMount`.

← Prev

Next →

Tags and Attributes

Edit on GitHub

Supported Elements

React attempts to support all common elements in both HTML and SVG. Any lower case tag in JSX will be rendered to an element with that tag. SVG elements must be contained within an `<svg>` element to work properly.

Using `React.DOM` Factory methods

If you aren't using JSX and are using the `React.DOM.*` API to create elements, then you are slightly more limited and there is list of supported elements that will be available on that API.

HTML Elements

The following HTML elements are supported in `React.DOM.*`:

Code

```
a abbr address area article aside audio b base bdi bdo big blockquote body br
button canvas caption cite code col colgroup data datalist dd del details dfn
dialog div dl dt em embed fieldset figcaption figure footer form h1 h2 h3 h4 h5
h6 head header hgroup hr html i iframe img input ins kbd keygen label legend li
link main map mark menu menuitem meta meter nav noscript object ol optgroup
option output p param picture pre progress q rp rt ruby s samp script section
select small source span strong style sub summary sup table tbody td textarea
tfoot th thead time title tr track u ul var video wbr
```

SVG elements

The following SVG elements are supported in `React.DOM.*`:

Code

```
circle clipPath defs ellipse g image line linearGradient mask path pattern
polygon polyline radialGradient rect stop svg text tspan
```

You may also be interested in react-art, a cross-browser drawing library for React.

Supported Attributes

React supports all `data-*` and `aria-*` attributes as well as every attribute in the following lists.

Note:

All attributes are camel-cased and the attributes `class` and `for` are `className` and `htmlFor`, respectively, to match the DOM API specification.

For a list of events, see Supported Events.

HTML Attributes

These standard attributes are supported:

Code

```
accept acceptCharset accessKey action allowFullScreen allowTransparency alt
async autoComplete autoFocus autoPlay capture cellPadding cellSpacing challenge
charSet checked cite classID className colSpan cols content contentEditable
contextMenu controls coords crossOrigin data dateTime default defer dir
disabled download draggable encType form formAction formEncType formMethod
formNoValidate formTarget frameBorder headers height hidden high href hrefLang
htmlFor httpEquiv icon id inputMode integrity is keyParams keyType kind label
lang list loop low manifest marginHeight marginWidth max maxLength media
mediaGroup method min minLength multiple muted name noValidate nonce open
optimum pattern placeholder poster preload profile radioGroup readOnly rel
required reversed role rowSpan rows sandbox scope scoped scrolling seamless
selected shape size sizes span spellCheck src srcDoc srcLang srcSet start step
style summary tabIndex target title type useMap value width wmode wrap
```

These RDFa attributes are supported (several RDFa attributes overlap with standard HTML attributes and thus are excluded from this list):

Code

```
about datatype inlist prefix property resource typeof vocab
```

In addition, the following non-standard attributes are supported:

- `autoCapitalize autoCorrect` for Mobile Safari.
- `color` for `<link rel="mask-icon" />` in Safari.
- `itemProp itemScope itemType itemRef itemID` for HTML5 microdata.
- `security` for older versions of Internet Explorer.
- `unselectable` for Internet Explorer.
- `results autoSave` for WebKit/Blink input fields of type `search`.

There is also 2 React-specific attributes: - `dangerouslySetInnerHTML` (more here), used for directly inserting HTML strings into a component. - `suppressContentEditableWarning`, used to suppress the warning when using `contentEditable` and `children`.

SVG Attributes

Code

```
accentHeight accumulate additive alignmentBaseline allowReorder alphabetic
amplitude arabicForm ascent attributeName attributeType autoReverse azimuth
baseFrequency baseProfile baselineShift bbox begin bias by calcMode capHeight
clip clipPath clipPathUnits clipRule colorInterpolation
colorInterpolationFilters colorProfile colorRendering contentScriptType
contentStyleType cursor cx cy d decelerate descent diffuseConstant direction
display divisor dominantBaseline dur dx dy edgeMode elevation enableBackground
end exponent externalResourcesRequired fill fillOpacity fillRule filter
filterRes filterUnits floodColor floodOpacity focusable fontFamily fontSize
fontSizeAdjust fontStretch fontStyle fontVariant fontWeight format from fx fy
g1 g2 glyphName glyphOrientationHorizontal glyphOrientationVertical glyphRef
gradientTransform gradientUnits hanging horizAdvX horizOriginX ideographic
imageRendering in in2 intercept k k1 k2 k3 k4 kernelMatrix kernelUnitLength
kerning keyPoints keySplines keyTimes lengthAdjust letterSpacing lightingColor
limitingConeAngle local markerEnd markerHeight markerMid markerStart
markerUnits markerWidth mask maskContentUnits maskUnits mathematical mode
numOctaves offset opacity operator order orient orientation origin overflow
overlinePosition overlineThickness paintOrder panose1 pathLength
patternContentUnits patternTransform patternUnits pointerEvents points
pointsAtX pointsAtY pointsAtZ preserveAlpha preserveAspectRatio primitiveUnits
r radius refX refY renderingIntent repeatCount repeatDur requiredExtensions
requiredFeatures restart result rotate rx ry scale seed shapeRendering slope
spacing specularConstant specularExponent speed spreadMethod startOffset
stdDeviation stemh stemv stitchTiles stopColor stopOpacity
strikethroughPosition strikethroughThickness string stroke strokeDasharray
strokeDashoffset strokeLinecap strokeLinejoin strokeMiterlimit strokeOpacity
strokeWidth surfaceScale systemLanguage tableValues targetX targetY textAnchor
textDecoration textLength textRendering to transform u1 u2 underlinePosition
underlineThickness unicode unicodeBidi unicodeRange unitsPerEm vAlphabetic
vHanging vIdeographic vMathematical values vectorEffect version vertAdvY
vertOriginX vertOriginY viewBox viewTarget visibility widths wordSpacing
writingMode x x1 x2 xChannelSelector xHeight xlinkActuate xlinkArcrole
xlinkHref xlinkRole xlinkShow xlinkTitle xlinkType xmlBase xmlLang xmlSpace
y y1 y2 yChannelSelector z zoomAndPan
```

← Prev

Next →

Edit on GitHub

Event System

SyntheticEvent

Your event handlers will be passed instances of `SyntheticEvent`, a cross-browser wrapper around the browser's native event. It has the same interface as the browser's native event, including `stopPropagation()` and `preventDefault()`, except the events work identically across all browsers.

If you find that you need the underlying browser event for some reason, simply use the `nativeEvent` attribute to get it. Every `SyntheticEvent` object has the following attributes:

Code

```
boolean bubbles
boolean cancelable
DOMEventTarget currentTarget
boolean defaultPrevented
number eventPhase
boolean isTrusted
DOMEvent nativeEvent
void preventDefault()
boolean isDefaultPrevented()
void stopPropagation()
boolean isPropagationStopped()
DOMEventTarget target
number timeStamp
string type
```

Note:

As of v0.14, returning `false` from an event handler will no longer stop event propagation. Instead, `e.stopPropagation()` or `e.preventDefault()` should be triggered manually, as appropriate.

Event pooling

The `SyntheticEvent` is pooled. This means that the `SyntheticEvent` object will be reused and all properties will be nullified after the event callback has been invoked. This is for performance reasons. As such, you cannot access the event in an asynchronous way.

Code

```
  console.log(event); // => nullified object.
  console.log(event.type); // => "click"
  var eventType = event.type; // => "click"

  setTimeout(function() {
    console.log(event.type); // => null
    console.log(eventType); // => "click"
  }, 0);

  this.setState({clickEvent: event});
// Won't work. this.state.clickEvent will only contain null values.
  this.setState({eventType: event.type});
// You can still export event properties.
}
```

Note:

If you want to access the event properties in an asynchronous way, you should call `event.persist()` on the event, which will remove the synthetic event from the pool and allow references to the event to be retained by user code.

Supported Events

React normalizes events so that they have consistent properties across different browsers.

The event handlers below are triggered by an event in the bubbling phase. To register an event handler for the capture phase, append `Capture` to the event name; for example, instead of using `onClick`, you would use `onClickCapture` to handle the click event in the capture phase.

Clipboard Events

Event names:

Code

```
onCopy onCut onPaste
```

Properties:

Code

```
DOMDataTransfer clipboardData
```

Composition Events

Event names:

Code

```
```

Properties:

Code

```
string data
```

Keyboard Events

Event names:

Code

```
onKeyDown onKeyPress onKeyUp
```

Properties:

Code

```
boolean altKey
number charCode
boolean ctrlKey
boolean getModifierState(key)
string key
number keyCode
string locale
number location
boolean metaKey
boolean repeat
boolean shiftKey
number which
```

Focus Events

Event names:

Code

```
onFocus onBlur
```

Properties:

Code

```
DOMEventTarget relatedTarget
```

These focus events work on all elements in the React DOM, not just form elements.

Form Events

Event names:

```
onChange onInput onSubmit
```

For more information about the onChange event, see Forms.

Mouse Events

Event names:

```
onClick onContextMenu onDoubleClick onDrag onDragEnd onDragEnter onDragExit
onDragLeave onDragOver onDragStart onDrop onMouseDown onMouseEnter onMouseLeave
onMouseMove onMouseOut onMouseOver onMouseUp
```

The `onMouseEnter` and `onMouseLeave` events propagate from the element being left to the one being entered instead of ordinary bubbling and do not have a capture phase.

Properties:

```
boolean altKey
number button
number buttons
number clientX
number clientY
boolean ctrlKey
boolean getModifierState(key)
boolean metaKey
number pageX
number pageY
DOMEventTarget relatedTarget
number screenX
number screenY
boolean shiftKey
```

Selection Events

Event names:

```
onSelect
```

Touch Events

Event names:

Code

```
onTouchCancel onTouchEnd onTouchMove onTouchStart
```

Properties:

Code

```
boolean altKey
DOMTouchList changedTouches
boolean ctrlKey
boolean getModifierState(key)
boolean metaKey
boolean shiftKey
DOMTouchList targetTouches
DOMTouchList touches
```

UI Events

Event names:

Code

```
onScroll
```

Properties:

Code

```
number detail
DOMAbstractView view
```

Wheel Events

Event names:

Code

```
onWheel
```

Properties:

Code
```
number deltaX
number deltaY
number deltaZ
```

Media Events

Event names:

Code
```
onAbort onCanPlay onCanPlayThrough onDurationChange onEmptied onEncrypted
onEnded onError onLoadedData onLoadedMetadata onLoadStart onPause onPlay
onPlaying onProgress onRateChange onSeeked onSeeking onStalled onSuspend
onTimeUpdate onVolumeChange onWaiting
```

Image Events

Event names:

Code
```
onLoad onError
```

Animation Events

Event names:

Code
```
onAnimationStart onAnimationEnd onAnimationIteration
```

Properties:

Code
```
string animationName
string pseudoElement
float elapsedTime
```

Transition Events

Event names:

Code
```
onTransitionEnd
```

Properties:

```
Code
string pseudoElement
float elapsedTime
```

← Prev   Next →

DOM Differences

Edit on GitHub

React has implemented a browser-independent events and DOM system for performance and cross-browser compatibility reasons. We took the opportunity to clean up a few rough edges in browser DOM implementations.

- All DOM properties and attributes (including event handlers) should be camelCased to be consistent with standard JavaScript style. We intentionally break with the spec here since the spec is inconsistent. **However**, `data-*` and `aria-*` attributes conform to the specs and should be lower-cased only.
- The `style` attribute accepts a JavaScript object with camelCased properties rather than a CSS string. This is consistent with the DOM `style` JavaScript property, is more efficient, and prevents XSS security holes.
- Since `class` and `for` are reserved words in JavaScript, the JSX elements for built-in DOM nodes should use the attribute names `className` and `htmlFor` respectively, (eg. `<div className="foo" />`). Custom elements should use `class` and `for` directly (eg. `<my-tag class="foo" />`).
- All event objects conform to the W3C spec, and all events (including submit) bubble correctly per the W3C spec. See Event System for more details.
- The `onChange` event behaves as you would expect it to: whenever a form field is changed this event is fired rather than inconsistently on blur. We intentionally break from existing browser behavior because `onChange` is a misnomer for its behavior and React relies on this event to react to user input in real time. See Forms for more details.
- Form input attributes such as `value` and `checked`, as well as `textarea`. More here.

← Prev   Next →

Special Non-DOM Attributes

Edit on GitHub

Beside DOM differences, React offers some attributes that simply don't exist in DOM.

- `key`: an optional, unique identifier. When your component shuffles around during `render` passes, it might be destroyed and recreated due to the diff algorithm. Assigning it a key that persists makes sure the component stays. See more here.
- `ref`: see here.
- `dangerouslySetInnerHTML`: Provides the ability to insert raw HTML, mainly for cooperating with DOM string manipulation libraries. See more here.

← Prev Next →

Reconciliation

Edit on GitHub

React's key design decision is to make the API seem like it re-renders the whole app on every update. This makes writing applications a lot easier but is also an incredible challenge to make it tractable. This article explains how with powerful heuristics we managed to turn a O(n^3) problem into a O(n) one.

Motivation

Generating the minimum number of operations to transform one tree into another is a complex and well-studied problem. The state of the art algorithms have a complexity in the order of O(n^3) where n is the number of nodes in the tree.

This means that displaying 1000 nodes would require in the order of one billion comparisons. This is far too expensive for our use case. To put this number in perspective, CPUs nowadays execute roughly 3 billion instructions per second. So even with the most performant implementation, we wouldn't be able to compute that diff in less than a second.

Since an optimal algorithm is not tractable, we implement a non-optimal O(n) algorithm using heuristics based on two assumptions:

1. Two components of the same class will generate similar trees and two components of different classes will generate different trees.
2. It is possible to provide a unique key for elements that is stable across different renders.

In practice, these assumptions are ridiculously fast for almost all practical use cases.

Pair-wise diff

In order to do a tree diff, we first need to be able to diff two nodes. There are three different cases being handled.

Different Node Types

If the node type is different, React is going to treat them as two different sub-trees, throw away the first one and build/insert the second one.

Code

```
renderA: <div />
renderB: <span />
=> [removeNode <div />], [insertNode <span />]
```

The same logic is used for custom components. If they are not of the same type, React is not going to even try at matching what they render. It is just going to remove the first one from the DOM and insert the second one.

Code

```
renderA: <Header />
renderB: <Content />
=> [removeNode <Header />], [insertNode <Content />]
```

Having this high level knowledge is a very important aspect of why React's diff algorithm is both fast and precise. It provides a good heuristic to quickly prune big parts of the tree and focus on parts likely to be similar.

It is very unlikely that a `<Header>` element is going to generate a DOM that is going to look like what a `<Content>` would generate. Instead of spending time trying to match those two structures, React just re-builds the tree from scratch.

As a corollary, if there is a `<Header>` element at the same position in two consecutive renders, you would expect to see a very similar structure and it is worth exploring it.

DOM Nodes

When comparing two DOM nodes, we look at the attributes of both and can decide which of them changed in linear time.

Code

```
renderA: <div id="before" />
renderB: <div id="after" />
=> [replaceAttribute id "after"]
```

Instead of treating style as an opaque string, a key-value object is used instead. This lets us update only the properties that changed.

Code

```
renderA: <div style={{color: 'red'}} />
renderB: <div style={{fontWeight: 'bold'}} />
=> [removeStyle color], [addStyle font-weight 'bold']
```

After the attributes have been updated, we recurse on all the children.

Custom Components

We decided that the two custom components are the same. Since components are stateful, we cannot just use the new component and call it a day. React takes all the attributes from the new component and calls `componentWillReceiveProps()` and `componentWillUpdate()` on the previous one.

The previous component is now operational. Its `render()` method is called and the diff algorithm restarts with the new result and the previous result.

List-wise diff

Problematic Case

In order to do children reconciliation, React adopts a very naive approach. It goes over both lists of children at the same time and generates a mutation whenever there's a difference.

For example if you add an element at the end:

Code

```
renderA: <div><span>first</span></div>
renderB: <div><span>first</span><span>second</span></div>
=> [insertNode <span>second</span>]
```

Inserting an element at the beginning is problematic. React is going to see that both nodes are spans and therefore run into a mutation mode.

Code

```
renderA: <div><span>first</span></div>
renderB: <div><span>second</span><span>first</span></div>
=> [replaceAttribute textContent 'second'], [insertNode <span>first</span>]
```

There are many algorithms that attempt to find the minimum sets of operations to transform a list of elements. Levenshtein distance can find the minimum using single element insertion, deletion and substitution in O(n^2). Even if we were to use Levenshtein, this doesn't find when a node has moved into another position and algorithms to do that have much worse complexity.

Keys

In order to solve this seemingly intractable issue, an optional attribute has been introduced. You can provide for each child a key that is going to be used to do the matching. If you specify a key, React is now able to find insertion, deletion, substitution and moves in O(n) using a hash table.

Code

```
renderB: <div><span key="second">second</span><span key="first">first</span></
div>
=> [insertNode <span>second</span>]
```

In practice, finding a key is not really hard. Most of the time, the element you are going to display already has a unique id. When that's not the case, you can add a new ID property to your model or hash some parts of the content to generate a key. Remember that the key only has to be unique among its siblings, not globally unique.

Trade-offs

It is important to remember that the reconciliation algorithm is an implementation detail. React could re-render the whole app on every action; the end result would be the same. We are regularly refining the heuristics in order to make common use cases faster.

In the current implementation, you can express the fact that a sub-tree has been moved amongst its siblings, but you cannot tell that it has moved somewhere else. The algorithm will re-render that full sub-tree.

Because we rely on two heuristics, if the assumptions behind them are not met, performance will suffer.

1. The algorithm will not try to match sub-trees of different components classes. If you see yourself alternating between two components classes with very similar output, you may want to make it the same class. In practice, we haven't found this to be an issue.
2. Keys should be stable, predictable, and unique. Unstable keys (like those produced by Math.random()) will cause many nodes to be unnecessarily re-created, which can cause performance degradation and lost state in child components.

← Prev Next →

Web Components

Edit on GitHub

Trying to compare and contrast React with WebComponents inevitably results in specious conclusions, because the two libraries are built to solve different problems. WebComponents provide strong encapsulation for reusable components, while React provides a declarative library that keeps the DOM in sync with your data. The two goals are complementary; engineers can mix-and-match the technologies. As a developer, you are free to use React in your WebComponents, or to use WebComponents in React, or both.

Using Web Components in React

Code

```
  render() {
    return <div>Hello <x-search>{this.props.name}</x-search>!</div>;
  }
}
```

Note:

The programming models of the two component systems (web components vs. react components) differ in that web components often expose an imperative API (for instance, a `video` web component might expose `play()` and `pause()` functions). To the extent that web components are declarative functions of their attributes, they should work, but to access the imperative APIs of a web component, you will need to attach a ref to the component and interact with the DOM node directly. If you are using third-party web components, the recommended solution is to write a React component that behaves as a wrapper for your web component.

At this time, events emitted by a web component may not properly propagate through a React render tree. You will need to manually attach event handlers to handle these events within your React components.

Using React in your Web Components

Code

```
var proto = Object.create(HTMLElement.prototype, {
  attachedCallback: {
    value: function() {
      var mountPoint = document.createElement('span');
      this.createShadowRoot().appendChild(mountPoint);

      var name = this.getAttribute('name');
      var url = 'https://www.google.com/search?q=' + encodeURIComponent(name);
      ReactDOM.render(<a href={url}>{name}</a>, mountPoint);
    }
  }
});
document.registerElement('x-search', {prototype: proto});
```

Complete Example

Check out the `webcomponents` example in the starter kit for a complete example.

← Prev

Next →

Edit on GitHub

React (Virtual) DOM Terminology

In React's terminology, there are five core types that are important to distinguish:

- ReactElement / ReactElement Factory
- ReactNode
- ReactComponent / ReactComponent Class

React Elements

The primary type in React is the `ReactElement`. It has four properties: `type`, `props`, `key` and `ref`. It has no methods and nothing on the prototype.

You can create one of these objects through `React.createElement`.

Code

```
var root = React.createElement('div');
```

To render a new tree into the DOM, you create `ReactElement`s and pass them to `ReactDOM.render` along with a regular DOM `Element` (`HTMLElement` or `SVGElement`). `ReactElement`s are not to be confused with DOM `Element`s. A `ReactElement` is a light, stateless, immutable, virtual representation of a DOM `Element`. It is a virtual DOM.

Code

```
ReactDOM.render(root, document.getElementById('example'));
```

To add properties to a DOM element, pass a properties object as the second argument and children to the third argument.

Code

```
var child = React.createElement('li', null, 'Text Content');
var root = React.createElement('ul', { className: 'my-list' }, child);
ReactDOM.render(root, document.getElementById('example'));
```

If you use React JSX, then these `ReactElement`s are created for you. So this is equivalent:

Code

```
var root = <ul className="my-list">
             <li>Text Content</li>
           </ul>;
ReactDOM.render(root, document.getElementById('example'));
```

Factories

A `ReactElement`-factory is simply a function that generates a `ReactElement` with a particular `type` property. React has a built-in helper for you to create factories. It's effectively just:

Code

```
function createFactory(type) {
  return React.createElement.bind(null, type);
}
```

It allows you to create a convenient short-hand instead of typing out `React.createElement('div')` all the time.

Code

```
var div = React.createFactory('div');
var root = div({ className: 'my-div' });
ReactDOM.render(root, document.getElementById('example'));
```

React already has built-in factories for common HTML tags:

Code

```
var root = React.DOM.ul({ className: 'my-list' },
             React.DOM.li(null, 'Text Content')
           );
```

If you are using JSX you have no need for factories. JSX already provides a convenient short-hand for creating `ReactElement`s.

React Nodes

A `ReactNode` can be either:

- `ReactElement`
- `string` (aka `ReactText`)
- `number` (aka `ReactText`)
- Array of `ReactNode`s (aka `ReactFragment`)

These are used as properties of other `ReactElement`s to represent children. Effectively they create a tree of `ReactElement`s.

React Components

You can use React using only `ReactElement`s but to really take advantage of React, you'll want to use `ReactComponent`s to create encapsulations with embedded state.

A `ReactComponent` Class is simply just a JavaScript class (or "constructor function").

Code

```
var MyComponent = React.createClass({
  render: function() {
    ...
  }
});
```

When this constructor is invoked it is expected to return an object with at least a `render` method on it. This object is referred to as a `ReactComponent`.

Code

```
var component = new MyComponent(props); // never do this
```

Other than for testing, you would normally *never* call this constructor yourself. React calls it for you.

Instead, you pass the `ReactComponent` Class to `createElement` you get a `ReactElement`.

Code

```
var element = React.createElement(MyComponent);
```

OR using JSX:

Code

```
var element = <MyComponent />;
```

When this is passed to `ReactDOM.render`, React will call the constructor for you and create a `ReactComponent`, which is returned.

Code

```
var component = ReactDOM.render(element, document.getElementById('example'));
```

If you keep calling `ReactDOM.render` with the same type of `ReactElement` and the same container DOM `Element` it always returns the same instance. This instance is stateful.

Code

```
var componentA = ReactDOM.render(<MyComponent />, document.getElementById(
'example'));
var componentB = ReactDOM.render(<MyComponent />, document.getElementById(
'example'));
componentA === componentB; // true
```

This is why you shouldn't construct your own instance. Instead, `ReactElement` is a virtual `ReactComponent` before it gets constructed. An old and new `ReactElement` can be compared to see if a new `ReactComponent` instance should be created or if the existing one should be reused.

The `render` method of a `ReactComponent` is expected to return another `ReactElement`. This allows these components to be composed. Ultimately the render resolves into `ReactElement` with a `string` tag which instantiates a DOM `Element` instance and inserts it into the document.

React 0.14 introduced stateless functional components as an alternative way of defining components. Instead of being a class, it is a simple function that accepts props and is expected to return a `ReactElement`.

Formal Type Definitions

Entry Point

Code

```
ReactDOM.render = (ReactElement, HTMLElement | SVGElement) => ReactComponent;
```

Nodes and Elements

Code

```
type ReactElement = ReactComponentElement | ReactDOMElement;

type ReactDOMElement = {
  type : string,
  props : {
    children : ReactNodeList,
    className : string,
    etc.
  },
  key : string | boolean | number | null,
  ref : string | null
};

type ReactComponentElement<TProps> = {
  type : ReactClass<TProps> | ReactFunctionalComponent<TProps>,
  props : TProps,
  key : string | boolean | number | null,
  ref : string | null
};

type ReactFragment = Array<ReactNode | ReactEmpty>;

type ReactNodeList = ReactNode | ReactEmpty;

type ReactText = string | number;

type ReactEmpty = null | undefined | boolean;
```

Classes and Components

Code

```
type ReactClass<TProps> = (TProps) => ReactComponent<TProps>;

type ReactComponent<TProps> = {
  props : TProps,
  render : () => ReactElement
};

type ReactFunctionalComponent<TProps> = (TProps) => ReactElement;
```

← Prev

Introduction

The React tips section provides bite-sized information that can answer lots of questions you might have and warn you against common pitfalls.

Contributing

Submit a pull request to the React repository following the current tips entries' style. If you have a recipe that needs review prior to submitting a PR you can find help in the #reactjs channel on freenode or the discuss.reactjs.org forum.

Next →

Inline Styles

In React, inline styles are not specified as a string. Instead they are specified with an object whose key is the camelCased version of the style name, and whose value is the style's value, usually a string (more on that later):

Code

```
var divStyle = {
  color: 'white',
  backgroundImage: 'url(' + imgUrl + ')',
  WebkitTransition: 'all', // note the capital 'W' here
  msTransition: 'all' // 'ms' is the only lowercase vendor prefix
};

ReactDOM.render(<div style={divStyle}>Hello World!</div>, mountNode);
```

Style keys are camelCased in order to be consistent with accessing the properties on DOM nodes from JS (e.g. `node.style.backgroundImage`). Vendor prefixes other than `ms` should begin with a capital letter. This is why `WebkitTransition` has an uppercase "W".

← Prev   Next →

If-Else in JSX

`if-else` statements don't work inside JSX. This is because JSX is just syntactic sugar for function calls and object construction. Take this basic example:

Code
```
ReactDOM.render(<div id="msg">Hello World!</div>, mountNode);

// Is transformed to this JS:
ReactDOM.render(React.createElement("div", {id:"msg"}, "Hello World!"), mountNode);
```

This means that `if` statements don't fit in. Take this example:

Code
```
// This JSX:
<div id={if (condition) { 'msg' }}>Hello World!</div>

// Is transformed to this JS:
React.createElement("div", {id: if (condition) { 'msg' }}, "Hello World!");
```

That's not valid JS. You probably want to make use of a ternary expression:

Code
```
ReactDOM.render(<div id={condition ? 'msg' : null}>Hello World!</
div>, mountNode);
```

If a ternary expression isn't robust enough, you can use `if` statements outside of your JSX to determine which components should be used:

Code
```
var loginButton;
if (loggedIn) {
  loginButton = <LogoutButton />;
} else {
  loginButton = <LoginButton />;
}

return (
  <nav>
    <Home />
    {loginButton}
  </nav>
);
```

Or if you prefer a more "inline" aesthetic, define immediately-invoked function expressions *inside* your JSX:

Code

```
  <section>
    <h1>Color</h1>
    <h3>Name</h3>
    <p>{this.state.color || "white"}</p>
    <h3>Hex</h3>
    <p>
      {(() => {
        switch (this.state.color) {
          case "red":   return "#FF0000";
          case "green": return "#00FF00";
          case "blue":  return "#0000FF";
          default:      return "#FFFFFF";
        }
      })()}
    </p>
  </section>
);
```

Note:

In the example above, an ES6 arrow function is utilized to lexically bind the value of `this`.

Try using it today with the Babel REPL.

← Prev Next →

Self-Closing Tag

In JSX, `<MyComponent />` alone is valid while `<MyComponent>` isn't. All tags must be closed, either with the self-closing format or with a corresponding closing tag (`</MyComponent>`).

Note:

Every React component can be self-closing: `<div />`. `<div></div>` is also an equivalent.

← Prev Next →

Maximum Number of JSX Root Nodes

Currently, in a component's `render`, you can only return one node; if you have, say, a list of `div`s to return, you must wrap your components within a `div`, `span` or any other component.

Don't forget that JSX compiles into regular JS; returning two functions doesn't really make syntactic sense. Likewise, don't put more than one child in a ternary.

← Prev   Next →

Shorthand for Specifying Pixel Values in style props

When specifying a pixel value for your inline `style` prop, React automatically appends the string "px" for you after your number value, so this works:

Code

```
var divStyle = {height: 10}; // rendered as "height:10px"
ReactDOM.render(<div style={divStyle}>Hello World!</div>, mountNode);
```

See Inline Styles for more info.

Sometimes you *do* want to keep the CSS properties unitless. Here's a list of properties that won't get the automatic "px" suffix:

- `animationIterationCount`
- `boxFlex`
- `boxFlexGroup`
- `boxOrdinalGroup`
- `columnCount`
- `fillOpacity`
- `flex`
- `flexGrow`
- `flexPositive`
- `flexShrink`
- `flexNegative`
- `flexOrder`
- `fontWeight`
- `lineClamp`
- `lineHeight`
- `opacity`

- `order`
- `orphans`
- `stopOpacity`
- `strokeDashoffset`
- `strokeOpacity`
- `strokeWidth`
- `tabSize`
- `widows`
- `zIndex`
- `zoom`

← Prev Next →

Type of the Children props

Usually, a component's children (`this.props.children`) is an array of components:

Code

```
var GenericWrapper = React.createClass({
  componentDidMount: function() {
    console.log(Array.isArray(this.props.children)); // => true
  },

  render: function() {
    return <div />;
  }
});

ReactDOM.render(
  <GenericWrapper><span/><span/><span/></GenericWrapper>,
  mountNode
);
```

However, when there is only a single child, `this.props.children` will be the single child component itself *without the array wrapper*. This saves an array allocation.

Code

```
  componentDidMount: function() {
    console.log(Array.isArray(this.props.children)); // => false

    // warning: yields 5 for length of the string 'hello', not 1 for the
    // length of the non-existent array wrapper!
    console.log(this.props.children.length);
  },

  render: function() {
    return <div />;
  }
});

ReactDOM.render(<GenericWrapper>hello</GenericWrapper>, mountNode);
```

To make `this.props.children` easy to deal with, we've provided the React.Children utilities
.

← Prev   Next →

Value of null for Controlled Input

Specifying the `value` prop on a controlled component prevents the user from changing the input unless you desire so.

You might have run into a problem where `value` is specified, but the input can still be changed without consent. In this case, you might have accidentally set `value` to `undefined` or `null`.

The snippet below shows this phenomenon; after a second, the text becomes editable.

Code

```
ReactDOM.render(<input value="hi" />, mountNode);

setTimeout(function() {
  ReactDOM.render(<input value={null} />, mountNode);
}, 1000);
```

← Prev   Next →

componentWillReceiveProps Not Triggered After Mounting

`componentWillReceiveProps` isn't triggered after the node is put on scene. This is by design. Check out other lifecycle methods for the one that suits your needs.

The reason for that is because `componentWillReceiveProps` often handles the logic of comparing with the old props and acting upon changes; not triggering it at mounting (where there are no old props) helps in defining what the method does.

← Prev Next →

Props in getInitialState Is an Anti-Pattern

Note:

This isn't really a React-specific tip, as such anti-patterns often occur in code in general; in this case, React simply points them out more clearly.

Using props to generate state in `getInitialState` often leads to duplication of "source of truth", i.e. where the real data is. This is because `getInitialState` is only invoked when the component is first created.

Whenever possible, compute values on-the-fly to ensure that they don't get out of sync later on and cause maintenance trouble.

Bad example:

Code

```
var MessageBox = React.createClass({
  getInitialState: function() {
    return {nameWithQualifier: 'Mr. ' + this.props.name};
  },

  render: function() {
    return <div>{this.state.nameWithQualifier}</div>;
  }
});

ReactDOM.render(<MessageBox name="Rogers"/>, mountNode);
```

Better:

Code

```
  render: function() {
    return <div>{'Mr. ' + this.props.name}</div>;
  }
});

ReactDOM.render(<MessageBox name="Rogers"/>, mountNode);
```

(For more complex logic, simply isolate the computation in a method.)

However, it's **not** an anti-pattern if you make it clear that the prop is only seed data for the component's internally-controlled state:

Code

```
var Counter = React.createClass({
  getInitialState: function() {
    // naming it initialX clearly indicates that the only purpose
    // of the passed down prop is to initialize something internally
    return {count: this.props.initialCount};
  },

  handleClick: function() {
    this.setState({count: this.state.count + 1});
  },

  render: function() {
    return <div onClick={this.handleClick}>{this.state.count}</div>;
  }
});

ReactDOM.render(<Counter initialCount={7}/>, mountNode);
```

← Prev   Next →

DOM Event Listeners in a Component

Note:

This entry shows how to attach DOM events not provided by React (check here for more info). This is good for integrations with other libraries such as jQuery.

Try to resize the window:

Code

```
  getInitialState: function() {
    return {windowWidth: window.innerWidth};
  },

  handleResize: function(e) {
    this.setState({windowWidth: window.innerWidth});
  },

  componentDidMount: function() {
    window.addEventListener('resize', this.handleResize);
  },

  componentWillUnmount: function() {
    window.removeEventListener('resize', this.handleResize);
  },

  render: function() {
    return <div>Current window width: {this.state.windowWidth}</div>;
  }
});

ReactDOM.render(<Box />, mountNode);
```

`componentDidMount` is called after the component is mounted and has a DOM representation. This is often a place where you would attach generic DOM events.

Notice that the event callback is bound to the react component and not the original element. React automatically binds methods to the current component instance for you through a process of autobinding.

← Prev Next →

Load Initial Data via AJAX

Fetch data in `componentDidMount`. When the response arrives, store the data in state, triggering a render to update your UI.

When fetching data asynchronously, use `componentWillUnmount` to cancel any outstanding requests before the component is unmounted.

This example fetches the desired Github user's latest gist:

Code

```
  getInitialState: function() {
    return {
      username: '',
      lastGistUrl: ''
    };
  },

  componentDidMount: function() {
    this.serverRequest = $.get(this.props.source, function (result) {
      var lastGist = result[0];
      this.setState({
        username: lastGist.owner.login,
        lastGistUrl: lastGist.html_url
      });
    }.bind(this));
  },

  componentWillUnmount: function() {
    this.serverRequest.abort();
  },

  render: function() {
    return (
      <div>
        {this.state.username}'s last gist is
        <a href={this.state.lastGistUrl}>here</a>.
      </div>
    );
  }
});

ReactDOM.render(
  <UserGist source="https://api.github.com/users/octocat/gists" />,
  mountNode
);
```

← Prev Next →

False in JSX

Here's how `false` renders in different situations:

Renders as `id="false"`:

Code

String `"false"` as input value:

Code

```
ReactDOM.render(<input value={false} />, mountNode);
```

No child:

Code

```
ReactDOM.render(<div>{false}</div>, mountNode);
```

The reason why this one doesn't render as the string `"false"` as a `div` child is to allow the more common use-case: `<div>{x > 1 && 'You have more than one item'}</div>`.

← Prev Next →

Communicate Between Components

For parent-child communication, simply pass props.

For child-parent communication: Say your `GroceryList` component has a list of items generated through an array. When a list item is clicked, you want to display its name:

Code

```
    console.log('You clicked: ' + props.items[i]);
  }

  function GroceryList(props) {
    return (
      <div>
        {props.items.map(function(item, i) {
          return (
            <div onClick={handleClick.bind(this, i, props)} key={i}>{item}</div>
          );
        })}
      </div>
    );
  }

  ReactDOM.render(
    <GroceryList items={['Apple', 'Banana', 'Cranberry']} />, mountNode
  );
```

Notice the use of `bind(this, arg1, arg2, ...)`: we're simply passing more arguments to `handleClick`. This is not a new React concept; it's just JavaScript.

For communication between two components that don't have a parent-child relationship, you can set up your own global event system. Subscribe to events in `componentDidMount()`, unsubscribe in `componentWillUnmount()`, and call `setState()` when you receive an event. Flux pattern is one of the possible ways to arrange this.

← Prev   Next →

Expose Component Functions

There's another (uncommon) way of communicating between components: simply expose a method on the child component for the parent to call.

Say a list of todos, which upon clicking get removed. If there's only one unfinished todo left, animate it:

Code

```
  render: function() {
    return <div onClick={this.props.onClick}>{this.props.title}</div>;
  },

  //this component will be accessed by the parent through the `ref` attribute
  animate: function() {
    console.log('Pretend %s is animating', this.props.title);
  }
});

var Todos = React.createClass({
  getInitialState: function() {
    return {items: ['Apple', 'Banana', 'Cranberry']};
  },

  handleClick: function(index) {
    var items = this.state.items.filter(function(item, i) {
      return index !== i;
    });
    this.setState({items: items}, function() {
      if (items.length === 1) {
        this.refs.item0.animate();
      }
    }.bind(this));
  },

  render: function() {
    return (
      <div>
        {this.state.items.map(function(item, i) {
          var boundClick = this.handleClick.bind(this, i);
          return (
            <Todo onClick={boundClick} key={i} title={item} ref={'item' + i} />
          );
        }, this)}
      </div>
    );
  }
});

ReactDOM.render(<Todos />, mountNode);
```

Alternatively, you could have achieved this by passing the `todo` an `isLastUnfinishedItem` prop, let it check this prop in `componentDidUpdate`, then animate itself; however, this quickly gets messy if you pass around different props to control animations.

← Prev   Next →

this.props.children undefined

You can't access the children of your component through `this.props.children`. `this.props.children` designates the children being **passed onto you** by the owner:

Code

```
var App = React.createClass({
  componentDidMount: function() {
    // This doesn't refer to the `span`s! It refers to the children between
    // last line's `<App></App>`, which are undefined.
    console.log(this.props.children);
  },

  render: function() {
    return <div><span/><span/></div>;
  }
});

ReactDOM.render(<App></App>, mountNode);
```

To access your own subcomponents (the `span`s), place refs on them.

← Prev   Next →

Use React with Other Libraries

You don't have to go full React. The component lifecycle events, especially `componentDidMount` and `componentDidUpdate`, are good places to put your other libraries' logic.

Code

```
  getInitialState: function() {
    return {myModel: new myBackboneModel({items: [1, 2, 3]})};
  },

  componentDidMount: function() {
    $(this.refs.placeholder).append($('<span />'));
  },

  componentWillUnmount: function() {
    // Clean up work here.
  },

  shouldComponentUpdate: function() {
    // Let's just never update this component again.
    return false;
  },

  render: function() {
    return <div ref="placeholder"/>;
  }
});

ReactDOM.render(<App />, mountNode);
```

You can attach your own event listeners and even event streams this way.

← Prev   Next →

Dangerously Set innerHTML

Improper use of the `innerHTML` can open you up to a cross-site scripting (XSS) attack. Sanitizing user input for display is notoriously error-prone, and failure to properly sanitize is one of the leading causes of web vulnerabilities on the internet.

Our design philosophy is that it should be “easy” to make things safe, and developers should explicitly state their intent when performing “unsafe” operations. The prop name `dangerouslySetInnerHTML` is intentionally chosen to be frightening, and the prop value (an object instead of a string) can be used to indicate sanitized data.

After fully understanding the security ramifications and properly sanitizing the data, create a new object containing only the key `__html` and your sanitized data as the value. Here is an example using the JSX syntax:

Code
```
<div dangerouslySetInnerHTML={createMarkup()} />
```

The point being that if you unintentionally do say `<div dangerouslySetInnerHTML={getUsername()} />` it will not be rendered because `getUsername()` would return a plain `string` and not a `{__html: ''}` object. The intent behind the `{__html:...}` syntax is that it be considered a "type/taint" of sorts. Sanitized data can be returned from a function using this wrapper object, and this marked data can subsequently be passed into `dangerouslySetInnerHTML`. For this reason, we recommend against writing code of the form `<div dangerouslySetInnerHTML={{__html: getMarkup()}} />`.

This functionality is mainly provided for cooperation with DOM string manipulation libraries, so the HTML provided must be well-formed (ie., pass XML validation).

For a more complete usage example, refer to the last example on the front page.

← Prev

Design Principles

After using React in a couple of applications, you might be interested in contributing to React. Before diving into specifics, we think it's important to establish a few design principles guiding our decisions about changes in React.

We wrote this document so that you have a better idea of how we decide what React does and what React doesn't do, and what our development philosophy is like. While we are excited to see community contributions, we are not likely to choose a path that violates one or more of these principles.

Note:

This document assumes a strong understanding of React. It describes the design principles of *React itself*, not React components or applications.

For an introduction to React, check out Thinking in React instead.

Composition

The key feature of React is composition of components. Components written by different people should work well together. It is important to us that you can add functionality to a component without causing rippling changes throughout the codebase.

For example, it should be possible to introduce some local state into a component without changing any of the components using it. Similarly, it should be possible to add some initialization and teardown code to any component when necessary.

There is nothing "bad" about using state or lifecycle hooks in components. Like any powerful features, they should be used in moderation, but we have no intention to remove them. On the contrary, we think they are integral parts of what makes React useful. We might enable more functional patterns in the future, but both local state and lifecycle hooks will be a part of that model.

Components are often described as "just functions" but in our view they need to be more than that to be useful. In React, components describe any composable behavior, and this includes rendering, lifecycle, and state. Some external libraries like Relay augment components with other responsibilities such as describing data dependencies. It is possible that those ideas might make it back into React too in some form.

Common Abstraction

In general we resist adding features that can be implemented in userland. We don't want to bloat your apps with useless library code. However, there are exceptions to this.

For example, if React didn't provide support for local state or lifecycle hooks, people would create custom abstractions for them. When there are multiple abstractions competing, React can't enforce or take advantage of the properties of either of them. It has to work with the lowest common denominator.

This is why sometimes we add features to React itself. If we notice that many components implement a certain feature in incompatible or inefficient ways, we might prefer to bake it into React. We don't do it lightly. When we do it, it's because we are confident that raising the abstraction level benefits the whole ecosystem. State, lifecycle hooks, cross-browser event normalization are good examples of this.

We always discuss such improvement proposals with the community. You can find some of those discussions by the “big picture” label on the React issue tracker.

Escape Hatches

React is pragmatic. It is driven by the needs of the products written at Facebook. While it is influenced by some paradigms that are not yet fully mainstream such as functional programming, staying accessible to a wide range of developers with different skills and experience levels is an explicit goal of the project.

If we want to deprecate a pattern that we don't like, it is our responsibility to consider all existing use cases for it and educate the community about the alternatives before we deprecate it. If some pattern that is useful for building apps is hard to express in a declarative way, we will provide an imperative API for it. If we can't figure out a perfect API for something that we found necessary in many apps, we will provide a temporary subpar working API as long as it is possible to get rid of it later and it leaves the door open for future improvements.

Stability

We value API stability. At Facebook, we have more than 20 thousand components using React. Many other companies, including Twitter and Airbnb, are also heavy users of React. This is why we are usually reluctant to change public APIs or behavior.

However we think stability in the sense of "nothing changes" is overrated. It quickly turns into stagnation. Instead, we prefer the stability in the sense of "It is heavily used in production, and when something changes, there is a clear (and preferably automated) migration path."

When we deprecate a pattern, we study its internal usage at Facebook and add deprecation warnings. They let us assess the impact of the change. Sometimes we back out if we see that it is too early, and we need to think more strategically about getting the codebases to the point where they are ready for this change.

If we are confident that the change is not too disruptive and the migration strategy is viable for all use cases, we release the deprecation warning to the open source community. We are closely in touch with many users of React outside of Facebook, and we monitor popular open source projects and guide them in fixing those deprecations.

Given the sheer size of the Facebook React codebase, successful internal migration is often a good indicator that other companies won't have problems either. Nevertheless sometimes people point out additional use cases we haven't thought of, and we add escape hatches for them or rethink our approach.

We don't deprecate anything without a good reason. We recognize that sometimes deprecations warnings cause frustration but we add them because deprecations clean up the road for the improvements and new features that we and many people in the community consider valuable.

For example, we added a warning about unknown DOM props in React 15.2.0. Many projects were affected by this. However fixing this warning is important so that we can introduce the support for custom attributes to React. There is a reason like this behind every deprecation that we add.

When we add a deprecation warning, we keep it for the rest of the current major version, and change the behavior in the next major version. If there is a lot of repetitive manual work involved, we release a codemod script that automates most of the change. Codemods enable us to move forward without stagnation in a massive codebase, and we encourage you to use them as well.

You can find the codemods that we released in the react-codemod repository.

Interoperability

We place high value in interoperability with existing systems and gradual adoption. Facebook has a massive non-React codebase. Its website uses a mix of a server-side component system called XHP, internal UI libraries that came before React, and React itself. It is important to us that any product team can start using React for a small feature rather than rewrite their code to bet on it.

This is why React provides escape hatches to work with mutable models, and tries to work well together with other UI libraries. You can wrap an existing imperative UI into a declarative component, and vice versa. This is crucial for gradual adoption.

Scheduling

Even when your components are described as functions, when you use React you don't call them directly. Every component returns a description of what needs to be rendered, and that description may include both user-written components like `<LikeButton>` and platform-specific components like `<div>`. It is up to React to "unroll" `<LikeButton>` at some point in the future and actually apply changes to the UI tree according to the render results of the components recursively.

This is a subtle distinction but a powerful one. Since you don't call that component function but let React call it, it means React has the power to delay calling it if necessary. In its current implementation React walks the tree recursively and calls render functions of the whole updated tree during a single tick. However in the future it might start delaying some updates to avoid dropping frames.

This is a common theme in React design. Some popular libraries implement the "push" approach where computations are performed when the new data is available. React, however, sticks to the "pull" approach where computations can be delayed until necessary.

React is not a generic data processing library. It is a library for building user interfaces. We think that it is uniquely positioned in an app to know which computations are relevant right now and which are not.

If something is offscreen, we can delay any logic related to it. If data is arriving faster than the frame rate, we can coalesce and batch updates. We can prioritize work coming from user interactions (such as an animation caused by a button click) over less important background work (such as rendering new content just loaded from the network) to avoid dropping frames.

To be clear, we are not taking advantage of this right now. However the freedom to do something like this is why we prefer to have control over scheduling, and why `setState()` is asynchronous. Conceptually, we think of it as "scheduling an update".

The control over scheduling would be harder for us to gain if we let the user directly compose views with a "push" based paradigm common in some variations of Functional Reactive Programming. We want to own the "glue" code.

It is a key goal for React that the amount of the user code that executes before yielding back into React is minimal. This ensures that React retains the capability to schedule and split work in chunks according to what it knows about the UI.

There is an internal joke in the team that React should have been called "Schedule" because React does not want to be fully "reactive".

Developer Experience

Providing a good developer experience is important to us.

For example, we maintain React DevTools which let you inspect the React component tree in Chrome and Firefox. We have heard that it brings a big productivity boost both to the Facebook engineers and to the community.

We also try to go an extra mile to provide helpful developer warnings. For example, React warns you in development if you nest tags in a way that the browser doesn't understand, or if you make a common typo in the API. Developer warnings and the related checks are the main reason why the development version of React is slower than the production version.

The usage patterns that we see internally at Facebook help us understand what the common mistakes are, and how to prevent them early. When we add new features, we try to anticipate the common mistakes and warn about them.

We are always looking out for ways to improve the developer experience. We love to hear your suggestions and accept your contributions to make it even better.

Debugging

When something goes wrong, it is important that you have breadcrumbs to trace the mistake to its source in the codebase. In React, props and state are those breadcrumbs.

If you see something wrong on the screen, you can open React DevTools, find the component responsible for rendering, and then see if the props and state are correct. If they are, you know that the problem is in the component’s `render()` function, or some function that is called by `render()`. The problem is isolated.

If the state is wrong, you know that the problem is caused by one of the `setState()` calls in this file. This, too, is relatively simple to locate and fix because usually there are only a few `setState()` calls in a single file.

If the props are wrong, you can traverse the tree up in the inspector, looking for the component that first "poisoned the well" by passing bad props down.

This ability to trace any UI to the data that produced it in the form of current props and state is very important to React. It is an explicit design goal that state is not "trapped" in closures and combinators, and is available to React directly.

While the UI is dynamic, we believe that synchronous `render()` functions of props and state turn debugging from guesswork into a boring but finite procedure. We would like to preserve this constraint in React even though it makes some use cases, like complex animations, harder.

Configuration

We find global runtime configuration options to be problematic.

For example, it is occasionally requested that we implement a function like `React.configure(options)` or `React.register(component)`. However this poses multiple problems, and we are not aware of good solutions to them.

What if somebody calls such a function from a third-party component library? What if one React app embeds another React app, and their desired configurations are incompatible? How can a third-party component specify that it requires a particular configuration? We think that global configuration doesn't work well with composition. Since composition is central to React, we don't provide global configuration in code.

We do, however, provide some global configuration on the build level. For example, we provide separate development and production builds. We may also add a profiling build in the future, and we are open to considering other build flags.

Beyond the DOM

We see the value of React in the way it allows us to write components that have less bugs and compose together well. DOM is the original rendering target for React but React Native is just as important both to Facebook and the community.

Being renderer-agnostic is an important design constraint of React. It adds some overhead in the internal representations. On the other hand, any improvements to the core translate across platforms.

Having a single programming model lets us form engineering teams around products instead of platforms. So far the tradeoff has been worth it for us.

Implementation

We try to provide elegant APIs where possible. We are much less concerned with the implementation being elegant. The real world is far from perfect, and to a reasonable extent we prefer to put the ugly code into the library if it means the user does not have to write it. When we evaluate new code, we are looking for an implementation that is correct, performant and affords a good developer experience. Elegance is secondary.

We prefer boring code to clever code. Code is disposable and often changes. So it is important that it doesn't introduce new internal abstractions unless absolutely necessary. Verbose code that is easy to move around, change and remove is preferred to elegant code that is prematurely abstracted and hard to change.

Optimized for Tooling

Some commonly used APIs have verbose names. For example, we use `componentDidMount()` instead of `didMount()` or `onMount()`. This is intentional. The goal is to make the points of interaction with the library highly visible.

In a massive codebase like Facebook, being able to search for uses of specific APIs is very important. We value distinct verbose names, and especially for the features that should be used sparingly. For example, `dangerouslySetInnerHTML` is hard to miss in a code review.

Optimizing for search is also important because of our reliance on codemods to make breaking changes. We want it to be easy and safe to apply vast automated changes across the codebase, and unique verbose names help us achieve this. Similarly, distinctive names make it easy to write custom lint rules about using React without worrying about potential false positives.

JSX plays a similar role. While it is not required with React, we use it extensively at Facebook both for aesthetic and pragmatic reasons.

In our codebase, JSX provides an unambigious hint to the tools that they are dealing with a React element tree. This makes it possible to add build-time optimizations such as hoisting constant elements, safely lint and codemod internal component usage, and include JSX source location into the warnings.

Dogfooding

We try our best to address the problems raised by the community. However we are likely to prioritize the issues that people are *also* experiencing internally at Facebook. Perhaps counter-intuitively, we think this is the main reason why the community can bet on React.

Heavy internal usage gives us the confidence that React won't disappear tomorrow. React was created at Facebook to solve its problems. It brings tangible business value to the company and is used in many of its products. Dogfooding it means that our vision stays sharp and we have a focused direction going forward.

This doesn't mean that we ignore the issues raised by the community. For example, we added support for web components and SVG to React even though we don't rely on either of them internally. We are actively listening to your pain points and address them to the best of our ability. The community is what makes React special to us, and we are honored to contribute back.

After releasing many open source projects at Facebook, we have learned that trying to make everyone happy at the same time produced projects with poor focus that didn't grow well. Instead, we found that picking a small audience and focusing on making them happy brings a positive net effect. That's exactly what we did with React, and so far solving the problems encountered by Facebook product teams has translated well to the open source community.

The downside of this approach is that sometimes we fail to give enough focus to the things that Facebook teams don't have to deal with, such as the "getting started" experience. We are acutely aware of this, and we are thinking of how to improve in a way that would benefit everyone in the community without making the same mistakes we did with open source projects before.