PREV
1. Introduction to React.js

NEXT
3. Component Lifecycle

Chapter 2. JSX

React is a rather opinionated library with strong beliefs on how you should structure and manage the data in your app. JSX is the result of one of those strongly held beliefs. Specifically React embraces the idea that markup, and the code which generates it, are inherently tied together. This is expressed in React components by rendering the markup directly within Javascript, using the full expressive power of the language.

To that end, React introduces an optional markup language remarkably similar to HTML. For example, a function call in plain React to create a header might look like this.

```
React.DOM.h1({className: 'question'}, 'Questions');
```

But with JSX it becomes much more familiar and terse looking markup.

```
<h1 className="question">Questions</h1>
```

In this chapter we'll walk you through the benefits of JSX and how to use it, along with some of the gotchas that separate it from HTML. Remember, JSX is optional. If you choose not to employ JSX you can skip to the end of the chapter for tips on using React without it.

What is JSX

JSX stands for JavaScript XML — an XML-like syntax for constructing markup within React components. React works without JSX, however it's use can make your components more readable and it is recommended.

Compared to past attempts at embedding markup in Javascript there are a few distinguishing characteristics that set JSX apart.

1. JSX is a syntactic transform — each JSX node maps to a javascript function.

2. JSX neither provides nor requires a runtime library.
3. JSX doesn't alter or add to the semantics of JavaScript — it's just simple function calls.

The similarity of JSX to HTML is what gives it so much expressive power within React. But in the end it maps to plain-old-javascript functions. Here we'll discuss the benefits of JSX & it's purpose within your app, as well as the key differences between JSX and HTML.

Benefits of JSX

A question many ask when considering JSX is why? Why use it at all when there are plenty of existing templating languages? Why not use plain javascript instead? After all, JSX simply maps to javascript functions.

Here are a few benefits or JSX which we'll discuss in turn.

- JSX is easier to visualize than javascript functions
- The markup is more familiar to designer and the rest of your team
- Your markup becomes more semantic, more meaningful.

Easy to Visualize

It's easiest to demonstrate JSX by example. Below you'll see the render function for a simple page-divider component used within the sample-app that accompanies this book. Notice that its intent is more clear and readable in JSX format.

Plain Javascript

```
render: function () {
  return React.DOM.div({className:"divider"},
    "Label Text",
    React.DOM.hr()
  );
}
```

JSX Markup

```
render: function () {
  return <div className="divider">
    Label Text<hr />
  </div>;
}
```

While it's certainly arguable, many agree the JSX markup is easier to understand and easier to debug. Which brings us to another key benefit of JSX: it's easier for non-programmers already familiar with html to work with it.

Familiarity

Many development teams include non-developers, from UI & UX designers who are familiar with html to quality-assurance teams responsible for thoroughly testing the product. These team members can more easily read and contribute code to a JSX project. Anyone with a familiarity with XML based languages can easily adopt JSX.

In addition, because React components capture all possible representations of your DOM (more about this in the section or Components,) JSX makes a great way to represent and visualize this structure in a compact and succinct way.

Semantics

Along with familiarity, JSX transforms your javascript code into more semantic, meaningful markup. This allows you the benefit of declaring your component structure and information flow using an HTML-like syntax, knowing it will transform into simple javascript functions.

React defines all the HTML elements you'd expect in the `React.DOM` namespace. It also allows you to use your own, custom components within the markup. We teach you all about custom components [link to chapter], so here we'll simply show how JSX can make your javascript more readable.

Let's refer back to the divider component mentioned above. It renders some header text on the left and has a horizontal-divider that stretches to fill the right. The HTML for this diver looks like this:

```
<div className="divider">
  <h2>Questions</h2><hr />
</div>
```

However, after wrapping this in a `Divider` React component you can use it you would any other html element, with the added benefit of markup with richer semantics.

```
<Divider>Questions</Divider>
```

Composite Components

Now that you've discovered some of the benefits of JSX and seen how it can be used to express a component in a compact markup format, let's look at how it helps us assemble multiple components.

This section will show you:

- How to set up a JavaScript file that contains JSX
- Walk you through how to assemble a component
- Discuss component ownership and the parent/child relationship

Let's look at each in turn.

Setup

While JSX allows you to express your components using markup, eventually it must be transformed into Javascript. In order for the JSX transform to know it should operate on a file you must insert the `/** @jsx React.DOM */` pragma at the top of each file containing JSX. Without this JSX will not know it should process the file for React.

The approach has a few benefits such as:

- The transformer can ignore all files without this pragma
- It allows JSX to live beside non-JSX JavaScript files

As mentioned earlier all element tag names in JSX map to plain JavaScript functions. React predefines all of the HTML tags you'd expect in the `React.DOM` namespace. The JSX transform conveniently pulls these in as top-level variables, which allows you to use `<div>...</div>` without needing to tediously specify `var div = React.DOM.div` in each JSX file.

All of your custom components, however, must be declared within the scope of your file, otherwise the React will not know how to resolve the function. As an example, throughout the sample app that accompanies this book you'll see code like this, specifying the custom components to be used within the scope of the file.

```
/** @jsx React.DOM */
var React = require("react");
```



```
// Use any `React.DOM.*` element or `Divider` below
<div>
  <Divider>...</Divider>
  <p>...</p>
</div>
```

Now that you've seen how to set up a JSX file you are ready to define a custom component using JSX.

Defining a Custom Component

Continuing with our previous example of the page divider, here again is the HTML we desire as output.

```
<div className="divider">
  <h2>Questions</h2><hr />
</div>
```

To express this HTML as a React component *all you have to do* is wrap it like this so the render function returns the markup.

```
var Divider = React.createClass({
  render: function () {
    return (
      <div className="divider">
        <h2>Questions</h2><hr />
      </div>
    );
  }
});
```

Of course this is currently a single-use component. To be truly useful you need the ability to express the text within the `h2` tag dynamically. But before we discuss making the text dynamic we need to discuss children.

Child Nodes

In HTML you render a header using `<h2>Questions</h2>` where the text "Questions" became a child text-node of the `h2` element. So the goal here is to express the divider in JSX similarly, like this:

```
<Divider>Questions</Divider>
```

React captures all child nodes between the open and close tags in an array on the component's props, `this.props.children`. In this example `this.props.children === ["Questions"]`.

`this.props.children`. React will now render anything you place inside the `<Divider>` tags.

```
var Divider = React.createClass({
  render: function () {
    return (
      <div className="divider">
        <h2>{this.props.children}</h2><hr />
      </div>
    );
  }
});
```

Now you can use the `<Divider>` component like you would any html element.

```
<Divider>Questions</Divider>
```

When run through the JSX transformer the above declaration will transform into this JavaScript.

```
var Divider = React.createClass({
  render: function () {
    return React.DOM.div(
      {className="divider"},
```

```
        React.DOM.h2(null, this.props.children),
        React.DOM.hr(null)
      );
    }
});
```

The output will be exactly what you expect.

```
<div className="divider">
  <h2>Questions</h2><hr />
</div>
```

How is JSX Different than HTML?

JSX is html-like, but it is not a perfect replication of the html syntax (for good reason.) In fact the JSX spec states:

> This specification does not attempt to comply with any XML or HTML specification. JSX is designed as an ECMAScript feature and the similarity to XML is only for familiarity. [1]

Here we explore some of the key differences between JSX and the HTML syntax.

Attributes

In HTML we set the attributes of each node inline, like this:

JSX implements attributes in the same manner, with the huge advantage that you can set attributes to dynamic javascript variables. You do this by wrapping a javascript variable in curly-brackets instead of quotes.

```
var surveyQuestionId = this.props.id;
var classes = 'some-class-name';
...
<div id={surveyQuestionId} className={classes}>...</div>
```

For more complex situations you can set an attribute to the result of a function call.

```
<div id={this.getSurveyId()} >...</div>
```

Now each time React chooses to render a component the variables and function calls will be evaluated and the resulting DOM will reflect this new state.

Conditionals

React embraces the idea that a component's markup and the logic that generate it are inherently tied together. This means you have the full logical power of javascript at your fingertips, such as

loops and conditionals.

It can be tricky to add conditional logic to your components since if/else logic is hard to express as markup. Adding if statements directly to JSX will render invalid JavaScript:

```
<div className={if(isComplete) { 'is-complete' }}>...</div>
```

So the solution is to use one of the following:

- Use ternary logic
- Set a variable and refer to it in the attribute
- Offload the switching logic to a function

Here is a quick example showing what each may look like.

Using the ternary operator — *whitespace added for clarity*

```
...
render: function () {
  return <div className={
    this.state.isComplete ? 'is-complete' : ''
  }>...</div>;
}
...
```


want to use a React component in either case. For these situations it is better to use the following methods.

Using a variable

```
...
getIsComplete: function () {
  return this.state.isComplete ? 'is-complete' : '';
},
render: function () {
  var isComplete = this.getIsComplete();
  return <div className={isComplete}>...</div>;
}
...
```

Using a function call

```
...
getIsComplete: function () {
  return this.state.isComplete ? 'is-complete' : '';
},
render: function () {
  return <div className={this.getIsComplete()}>...</div>;
```

```
}
...
```

Non-DOM Attributes

The following attributes are reserved words in JSX.

- `key`
- `ref`
- `dangerouslySetInnerHTML`

Keys

`key` is an optional unique identifier. During runtime a component might move up or down the component tree, for example as the user performs a search or items are added or removed from a list. When this happens you component might be needlessly destroyed and recreated.

By setting a unique key on a component that remains consistent throughout render passes you inform React so it more intelligently decides when to reuse or destroy a component, improving rendering performance. Then when two items already in the DOM switch positions React can match the keys and move them without completely re-rendering their DOM.

References

`ref` allows parent components to keep a reference to child components available outside of the

You define a ref in JSX by setting the attribute to the desired reference name.

```
...
render: function () {
  return <div>
    <input ref="myInput" ... />
  </div>;
}
...
```

And later you can access this ref by using `this.refs.myInput` anywhere in your component. The object you access through this ref is called a **backing instance**. It's not the actual DOM, but a description of the component React uses to create the DOM when necessary. To access the actual DOM node you use `this.refs.myInput.getDOMNode()`.

For more detail see the discussion on parent/child relationships vs ownership in chapter 4.

Setting Raw HTML

`dangerouslySetInnerHTML` — sometimes you need to set html content as a string, especially when working with third party libraries that manipulate DOM via strings. To improve React's

interoperability this attribute allows you to use html strings, but it's not recommended if you can avoid it. To use this property set it to an object with key `__html` set, like this:

```
...
render: function () {
  var htmlString = {
    __html: "<span>an html string</span>"
  };
  return <div dangerouslySetInnerHTML={htmlString} ></div>;
}
...
```

Events

Event names are normalized across all browsers and are represented camelCased. For example `change` becomes `onChange`, and `click` becomes `onClick`. When capturing an event in JSX it's as simple as assigning the property to a method on the component.

```
...
handleClick: function (event) {...},
render: function () {
  return <div onClick={this.handleClick}>...</div>
}
...
```

Note that React automatically binds all of a component's methods, so you never need to manually


```
...
handleClick: function (event) {...},
render: function () {
  // anti-pattern - manually binding the function context
  // to a component instance is unnecessary in React.
  return <div onClick={this.handleClick.bind(this)}>...</div>
}
...
```

For more details on the event system in React reference the chapter on Forms.

Special Attributes

Because JSX transforms to plain javascript function calls there are a few keywords we cannot use — `class` and `for`.

To create a form label with the `for` attribute use `htmlFor`.

```
<label htmlFor="for-text" ... >
```

To render a custom `class` use `className`. This might seem odd if you're used to HTML, but it is more consistent with vanilla javascript, where we can access the class of an element using

`elem.className`.

```
<div className={classes} ... >
```

Styles

Lastly we come to the inline style attribute. React normalizes all styles to camelCased names, consistent with the DOM style JavaScript property.

To define a custom style attribute simply pass a JavaScript object with camelCase property names and the desired css values.

```
var styles = {
  borderColor: "#999",
  borderThickness: "1px"
};
React.renderComponent(<div style={styles}>...</div>, node);
```

React Without JSX

All JSX markup is eventually transformed into simple JavaScript function calls. So JSX is not *necessary* for using React.

It Functions All The Way Down

defined in the `React.DOM` namespace. So creating a div is as simple as calling the function.

```
React.DOM.div();

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

The first argument to a component function is the props object. So customizing the attributes of a div would look like the following code. Each key in the props object maps to the attribute name on the rendered div element.

```
React.DOM.div({className: "divider"});

// <div class="divider"></div>
```

Child nodes are passed in as optional arguments after the props object. Using our divider example from above, which as two child nodes, we can render it as raw JavaScript with the text node and `hr` component passed in as the second and third arguments, respectively.

```
React.DOM.div(
  {className: "divider"},
```

```
  "Label Text",
  React.DOM.hr()
)

// <div class="divider">Label Text<hr/></div>
```

Shorthand

Writing `React.DOM.div` can be tedious after a while. It can be convenient to save a reference using a shorter variable name, e.g. `R`. Thus we can express the above div a bit more tersely.

```
var R = React.DOM;
//...
R.div(
  {className: "divider"},
  "Label Text",
  R.hr()
);
```

Or, if you prefer to access the component as top-level variables you can reference them directly.

```
var div = React.DOM.div;
var hr = React.DOM.hr;
//...
div(
  {className: "divider"},
  "Label Text",
```


Further Reading & References

Even if you don't like the idea of markup in your Javascript, hopefully you can now appreciate how JSX offers a solution to the intimate relationship between your Javascript and the markup it renders. Its growing popularity has earned JSX its own spec offering a deep technical definition, and there are a few tools to help you experiment with it if you're still uncertain or confused about how it works.

The Official JSX Spec

In September of 2014 Facebook released an official spec for JSX (http://facebook.github.io/jsx/), stating its rationale for creating JSX, along with technical details of the syntax.

You can read more at http://facebook.github.io/jsx/ (http://facebook.github.io/jsx/).

In-Browser Experimenting

There are also a number of tools for experimenting with JSX. The React docs Getting Started (http://facebook.github.io/react/docs/getting-started.html) page links to JSFiddle playgrounds with and without JSX.

http://facebook.github.io/react/docs/getting-started.html (http://facebook.github.io/react/docs/getting-started.html)

And React comes with a JSX Compiler Service (http://facebook.github.io/react/jsx-compiler.html) that transforms JSX into plain Javascript in-browser.

http://facebook.github.io/react/jsx-compiler.html (http://facebook.github.io/react/jsx-compiler.html)

[1] Retrieved from http://facebook.github.io/jsx/

PREV
1. Introduction to React.js

NEXT
3. Component Lifecycle