Community Experience Distilled

Master the art of building modern web applications using React

Mastering React

Adam Horton     Ryan Vice

React stands out in the web framework crowd through its approach to composition. This approach yields blazingly fast rendering capabilities. This book will help you understand what makes React special. It starts with the fundamentals and uses a pragmatic approach, focusing on clear development goals. You’ll learn how to combine many web technologies surrounding React into a complete set for constructing a modern web application.

With this text, you’ll blitz the basics then swiftly move on to advanced topics such as form validation and complete application construction. You’ll also explore several design activities which will help you develop your web applications with a thoughtful plan. Finally, you’ll learn several methods for implementing slick animations using React.

Who this book is written for

This book is ideal for web developers possessing strong core JavaScript fundamentals who are also interested in learning what React brings to the architectural table. Previous experience with React or other web frameworks isn’t required, but may help.

What you will learn from this book

- Understand the React component lifecycle and core concepts such as props and states
- Craft forms and implement form validation patterns using React
- Explore the anatomy of a modern single-page web application
- Develop an approach for choosing and combining web technologies without being paralyzed by the options available
- Create a complete single-page application
- Start coding with a plan using an application design process
- Add to your arsenal of prototyping techniques and tools
- Make your React application feel great using animations

In this package, you will find:

- The author's biography
- A preview chapter from the book, Chapter 1 'Introduction to React'
- A synopsis of the book’s content
- More information on Mastering React
About the Authors

Adam Horton is an avid retro gamer as well as a creator, destroyer, and rebuilder of all things Web, computing, and gaming. He started his career as a firmware developer for the high-end Superdome server division at Hewlett Packard. There, the JavaScript and C he wrought directed the power, cooling, health, and configuration of those behemoth devices. Since then, he has been a web developer for PayPal, utilizing cross-domain JavaScript techniques with an emphasis on user identity. Lately, at ESO Solutions, he's a lead JavaScript developer for next-generation, pre-hospital electronic health record (EHR) field collection applications.

Adam believes in an inclusive, ubiquitous, and open Web. He values pragmatism and practice over dogma in the design and implementation of computing applications and education.

Ryan Vice is the founder and chief architect of Vice Software, which specializes in practical, tailored solutions for clients, whether they are looking to get their MVP to market or modernize existing applications. On top of offering more competitive prices across the board, Vice Software offers skill-based pricing, which means you only pay architect rates when needed and pay much lower production rates for simpler feature work. Ryan has also been awarded Microsoft's MVP award three times, has published one other book on software architecture, and frequently speaks at conferences and events in Texas. Additionally, Ryan is lucky enough to be married to his wife, Heather, and spends most of his free time trying to keep up with their three kids, Grace, Dylan, and Noah.
The book before you is a collaboration between myself—a web-focused, day-to-day code slinger—and Ryan Vice—a veteran .NET expert turned web application Maven and entrepreneur. I met Ryan at my current company while creating very complex web applications for emergency response outfits, such as EMS. At that time, the company was building a revolution of its flagship product. While it was still a very early time for React, it offered something above and beyond all the current MV* frameworks: a fresh approach to blazing-fast rendering. Since speed is paramount in the emergency response industry, we embraced React. When an opportunity came along to write about React, Ryan tapped me for assistance, and this book is the result of that.

It's great timing for a technology such as React. The open web platform has succeeded because of a simple tenet: never break the Web. As a result, the DOM, the in-browser data representation of a rendered web page, has grown to be enormous and somewhat unwieldy. Every small touch on the DOM cascades into a flurry of calculation and reconciliation in order to update pixels on the screen. React treats the DOM as the expensive resource that it is and makes operations on the DOM minimal. The time saved is better spent running your complex application logic.

In this book, you'll learn the fundamentals of React as well as a pragmatic approach to making web applications. You'll also learn how to choose the right tools for your needs. In the first three chapters, we'll start by thoroughly covering basic React topics such as state, props, and JSX, as well as the more complex subject of forms and validation. Chapter 4, Anatomy of a React Application, is something unique to a book like this: a tour of web application anatomy and some design techniques that can help you formulate a clear plan to build your apps. In chapters 5 through 9, we'll build a multiuser blog application using the design techniques and supporting libraries explored in the previous two chapters. Finally, in Chapter 10, Animation in React, we'll have some fun by navigating through the many ways you can create cool animations using React.
What this book covers

Chapter 1, *Introduction to React*, explains the basics of React JS, starting with a simple Hello World example and moving forward thru types of React entities and their definitions.

Chapter 2, *Component Composition and Lifecycle*, explores nesting components and managing their state as they come and go from the DOM.


Chapter 4, *Anatomy of a React Application*, teaches how to approach web application design and how to choose from the vast menu of web technologies available within the context of React being your primary choice. Practice technical design and learn how to generate artifacts that guide development.

Chapter 5, *Starting a React Application*, begins with a fully featured React multiuser blog application. Prepare a development environment. Install all of the tools. Scaffold the application views.

Chapter 6, *React Blog App Part 1 – Actions and Common Components*, establishes a application communication strategy using ReFlux Actions. Here, we will create some common components.


Chapter 10, *Animation in React*, reveals web animation techniques in React.
Introduction to React

This book is an advanced book on React but we wanted to provide a primer on the basics so that this book could be both comprehensive and accessible. We will not spend a lot of time on all of the subtleties of each technique that we will look at here. We will instead look at concise samples that illustrate the tools and techniques that we are covering. We will also have links to where you can easily access and run the code samples as you follow along.

In this chapter, we will be covering the following concepts:

- About the code samples
- Hello World sample in React
- JSX
- props
- state

Hello React

For the first example we will create a fully working Hello World-style example using React by undertaking the following steps:

1. Create a new HTML file and call it hello-react.html.
2. Paste the following code in hello-react.html:

```html
<!DOCTYPE html>
<html>
<head lang="en">
<meta charset="UTF-8">
<title>Hello React</title>
<script src="https://fb.me/react-with-addons-0.14.0.js">
</head>
<body>
  <div>Hello React</div>
</body>
</html>
```
Introduction to React

3. Open HelloReactJs.html in a browser and you should see something similar to the image that follows:

React is a component-based framework that allows creating composable view components. The first thing that we have to do to be able to create a component in React is to include the React source as shown below:

As of version 0.13.0, React splits its API into two files. Now, there is one file that contains the browser-specific DOM code and another file that contains the rest of React's API. This was done because React is being used in more and more places and currently is used by React Native to build mobile applications. It can also be used to build Windows and Mac desktop applications using platforms such as Electron.
This will bring in version v0.14.0 of React, and then the code in the script block that is shown in the following code creates a `ReactElement` parameter of type `h1` and sets children to be the string `Hello React`:

```javascript
React.createElement('h1', null, 'Hello React');
```

We pass the `ReactElement` parameter that is created by this call as the first argument to the `ReactDOM.render()` method shown as follows:

```javascript
ReactDOM.render(
    React.createElement('h1', null, 'Hello React'),
    document.body);
```

React's render method is taking a `ReactElement` parameter as its first argument and a `document.body` DOM element as its second argument. The render method will then write the HTML generated by the first argument, the `ReactElement`, as a child of the second argument, the `document.body` DOM element. You can see the results in Chrome's Elements tab, as shown in the following screenshot:

![Hello React](image.png)
Introduction to React

As you can see, we now have an h1 element in our DOM that contains the string Hello React as a child.

However, you might be wondering how this is component based and you'd be right in being skeptical as we haven't created a component yet. Components are one of the things that really makes React a powerful and flexible framework so let's see what our example looks like if we update it to create a component. To do this, update the script in hello-react.html as shown in the following code, and refresh the browser to verify that our program still works the same:

```javascript
var HelloReact = React.createClass({
  render: function() {
    return React.DOM.h1(null, 'Hello React');
  }
});

ReactDOM.render(React.createElement(HelloReact), document.body);
```

Now we are creating a HelloReact JavaScript variable and assigning it a newly created React component that is created using the React.createClass() method as shown below:

```javascript
var HelloReact = React.createClass({
  render: function() {
    return React.DOM.h1(null, 'Hello React');
  }
});

ReactDOM.render(React.createElement(HelloReact), document.body);
```

The createClass() method takes an object that must specify a render method. The render method is responsible for returning a single ReactClass to be rendered. Here we are creating a ReactClass that represents an h1 DOM element that contains the string Hello React.
Note that we are now using the React.DOM API to create our h1 ReactElement instance. The React.DOM API provides convenience methods for creating common HTML elements and internally calls the React.createElement() method and passes the needed parameters for us. It may seem odd that the React.DOM part of the API was not moved to ReactDOM like ReactDOM.render. However, it appears that the React team has decided that the HTML semantics and UI widgets are part of their universal approach to building UIs, while the actual rendering is platform specific. Here's an excerpt from the React documentation about the restructuring.

"As we look at packages such as react-native, react-art, react-canvas, and react-three, it has become clear that the beauty and essence of React has nothing to do with browsers or the DOM. To make this more clear and to make it easier to build more environments that React can render, we're splitting the main React package into two: react and react-dom. This paves the way to writing components that can be shared between the Web version of React and React Native. We don't expect all the code in an app to be shared, but we want to be able to share the components that do behave in the same manner across platforms.

The React package contains React.createElement, .createClass, .Component, .PropTypes, .Children, and the other helpers related to elements and component classes. We think of these as isomorphic or universal helpers that you need to build components."

As shown in the following code, we then call the React.render() method and pass the results of calling React.createElement(HelloReact):

```javascript
ReactDOM.render(React.createElement(HelloReact), document.body);
```

Now we've updated our code to create a React component and, as we will see, these components will offer a lot of power and flexibility to our web applications.

JSX

In order to make its component API easier to use, React has its own syntax called JSX, which combines JavaScript and HTML. Let’s take a look at updating our sample code to use JSX by copying the following code into hello-react.html and refreshing our browser:

```html
<!DOCTYPE html>
<html>
<head lang="en">
  <meta charset="UTF-8">
  <title>Hello React</title>
  <script src="https://fb.me/react-with-addons-0.14.0.js"></script>
  <script src="https://fb.me/react-dom-0.14.0.js"></script>
  <script src="http://fb.me/JSXTransformer-0.13.1.js"></script>
</head>
<body>
  <script>
    var HelloReact = React.createClass({
      render: function() {
        return React.DOM.h1(null, 'Hello React');
      }
    });
    ReactDOM.render(React.createElement(HelloReact), document.body);
  </script>
</body>
</html>
```

Note that we are assigning a type of "text/jsx" to our script tag in the preceding sample.

How it works

The first thing we had to do to make this work was to add a reference to the JSX Transformer file shown in the following code:

```html
<script src="http://fb.me/JSXTransformer-0.13.1.js"></script>
```
Note that using an in browser transformer is only recommended for testing. We will look into more appropriate ways to transform the JSX code to JavaScript in later chapters. Also note that as of 0.14 React recommends using Babel for doing JSX transforms and has deprecated it's JSX Transformer.

Next we updated our component creation code as shown below:

```javascript
var HelloReact = React.createClass({
    render: function() {
        return <h1>Hello React</h1>;
    }
});
```

Now instead of calling into the React API to define our components DOM structure, we just write the desired DOM structure inline within our return statement.

As we will see later, we can even reference other react components as HTML elements!

And now we can just pass the HTML tag version of our component into the `React.render()` method to have it rendered to the DOM:

```javascript
ReactDOM.render(<HelloReact />, document.body);
```

Notice how we no longer need to call `React.createElement()`. This is because the JSX compiler will make the call for us.

Source code: [http://j.mp/MasteringReact-1-2-Gist](http://j.mp/MasteringReact-1-2-Gist)

Now that we have seen how to set up a page to host the React application, we will start using JsFiddle to look at the examples. The example we just looked at is in the fiddle ([http://j.mp/MasteringReact-1-2-1-Fiddle](http://j.mp/MasteringReact-1-2-1-Fiddle)).
Follow that link and click the **Run** button to run the code. You should see the following output:

![Fiddle screenshot](image)

---

**Decompiling JSX**

Babel has created a tool that allows seeing the decompiled JavaScript that would be created during JSX transformation. The Babel tool is shown in the following screenshot, and you can find the tool at [https://babeljs.io/repl/](https://babeljs.io/repl/):
The JSX Compiler has two code windows. The code window on the left is where you can put JSX, and the results of compiling it to React's API is shown in the code window on the right.

Structure of render result

There are a few things we should take note of about using JSX to create the ReactElement parameter that you return from your component's `render()` method. Let's say that we want to write Hello React code in two lines. You might be tempted to structure your code like the code that follows:

```javascript
var HelloMessage = React.createClass({
    render: function() {
        return <div>Hello React</div> // error
        <div>How are you?</div>
    }
});
```

However, if we paste this code into the JSX Compiler we will see the following error:

```
Error: Parse Error: Line 1: Adjacent JSX elements must be wrapped in an enclosing tag
```

[9]
The error here is indicating that we need to wrap our JSX in a single root element shown in the following code:

```javascript
var HelloMessage = React.createClass({
    render: function() {
        return <div> // works
            <div>Hello React</div>
            <div>Hello React</div>
            <div>Hello React</div>
        </div>;
    }
});
```

If we paste this in the JSX compiler we won't see any errors and, instead, we will see the compiled React API code as shown below:

```javascript
var HelloMessage = React.createClass({displayName: "HelloMessage",
    render: function() {
        return React.createElement("div", null, " // works",
            React.createElement("div", null, "Hello React"),
            React.createElement("div", null, "How are you?")
        );
    }
});
```

Looking at the code generated by the compiler we can see why our previous code structure wouldn't work as our JavaScript return statement can only return a single ReactElement. Because of this we have to return a single root node and can't return multiple adjacent sibling nodes as the error pointed out. However, now that we have added a root node, everything works and we can see in the compiled output that we are creating a div that contains two <div> tags and returning a single ReactElement.

Now we might want to format our code better for readability and we could be tempted to do something, as shown in the following code:

```javascript
var HelloReact = React.createClass({
    render: function() {
        return {
            <div>
                <div>Hello React</div>
                <div>Hello React</div>
            </div>
        };
    }
});
```

Source code: http://j.mp/MasteringReact-1-3-Gist
This will not throw an error in the JSX Compiler but it will throw the runtime error shown in the following code:

```
Uncaught Error: Invariant Violation: HelloReact.render(): A valid ReactComponent must be returned. You may have returned undefined, an array or some other invalid object.
```

Source code: [http://j.mp/MasteringReact-1-4-Gist](http://j.mp/MasteringReact-1-4-Gist)
Fiddle: [http://j.mp/MasteringReact-1-4-Fiddle](http://j.mp/MasteringReact-1-4-Fiddle)

You will need to open the developer tools in your browser and look in the console output to see the error.

All is not lost though because simply wrapping our JSX in parenthesis will allow us more flexibility when formatting our code. The following code sample uses this approach and we will no longer get an error when we run it. I like using this approach as it lets me keep my code nice and neatly formatted:

```
var HelloReact = React.createClass({
    render: function() {
        return (
            <div>
                <div>Hello React</div>
                <div>How are you?</div>
            </div>
        );
    }
});
```

Source code: [http://j.mp/MasteringReact-1-5-Gist](http://j.mp/MasteringReact-1-5-Gist)
Fiddle: [http://j.mp/MasteringReact-1-5-Fiddle](http://j.mp/MasteringReact-1-5-Fiddle)

### props

So far our components have not been configurable. Clearly we would want to be able to define a component that can take arguments via the component's HTML element attributes. The following code shows how we can do this:

```
var HelloReact = React.createClass({
    render: function() {
        return (
            <div>
                <div>Hello React</div>
            </div>
        );
    }
});
```
Introduction to React

```javascript
<div>{this.props.message}</div>
</div>

ReactDOM.render(
  <HelloReact message='Message from props'/>,
  document.getElementById('view'))
```

Note that props are immutable and not dynamic. To change the value of a prop requires rerendering the component.

How it works

React components have props collections that will be populated from the component's HTML attributes that the component is declared with. So, in the following code, we are setting the message attribute of the prop collection to `Message from props`:

```javascript
React.render(
  <HelloReact message='Message from props'/>,
  document.getElementById('view'))
```

Now, in our component's definition, we can access the value that message was set to by using `this.props.message`.

Note that if we want to access this in JSX markup, we need to surround the code with brackets, shown in the following code:

```javascript
var HelloReact = React.createClass({
  render: function() {
    return {
      <div>
        <div>Hello React</div>
        <div>{this.props.message}</div>
      </div>
    }
  }
});
```
We can also copy prop values to local variables as shown in the following code, and then reference the local variables in our JSX markup shown in the following code:

```javascript
var HelloReact = React.createClass({
  render: function() {
    var localMessage = this.props.message;

    return {
      <div>
        <div>Hello React</div>
        <div>{localMessage}</div>
      </div>
    };
  }
});
```

We can use any valid JavaScript expression between the brackets in our JSX so we could update the previous example shown in the following code:

```
{localMessage + ' and from JSX'}
```

This will concatenate the value contained in the `localMessage` variable with the string 'and from JSX', resulting in the output shown in the following screenshot:

![Hello React](image)

**propTypes**

We would also like to be able to validate our props and we can do some basic validations using `propTypes` as shown in the following code:

```javascript
var HelloReact = React.createClass({
  propTypes: {
    message: React.PropTypes.string,
  }
});
```
number: React.PropTypes.number,
requiredString: React.PropTypes.string.isRequired
},
render: function() {
return {
<div>
<div>Hello React</div>
<div>{this.props.message}</div>
</div>
};
});

ReactDOM.render(<HelloReact message='How are you' number='not a number'/>,
document.getElementById('view'));

Now, if we run this code and look at the warnings in the console of our browser's debug tools we will see the warnings shown in the following code. These warnings indicate that we have two invalid props:

Warning: Failed propType: Invalid prop `number` of type `string` supplied to `HelloReact`, expected `number`.

Warning: Failed propType: Required prop `requiredString` was not specified in `HelloReact`.

By defining the propTypes property of the component's class we were able to configure validations to enforce that our message prop is a string, our number is a number, and our requiredString is a string that is required. Then, when we define our component as shown in the following code, we violate some of our validation rules and get appropriate warning messages:

React.render(<HelloReact message='How are you' number='not a number'/>,
document.getElementById('view'));

Source code: http://j.mp/Mastering-React-1-7-Gist
Fiddle: http://j.mp/Mastering-React-1-7-Fiddle
**getDefaultProps**

We can also provide default property values that will get used if an attribute isn’t specified in the HTML markup declaring our component. The following code shows how we can create a default value for the message prop by defining a `getDefaultProps` method:

```javascript
var HelloReact = React.createClass({
    getDefaultProps: function() {
        return {
            message: 'I am from default'
        },
    },
    render: function() {
        return (
            <div>
                <div>Hello React</div>
                <div>{this.props.message}</div>
            </div>
        );
    }
});

ReactDOM.render(<HelloReact />, document.getElementById('view'));
```

Now when we run the code we will see the output as shown below:

```
Hello React
I am from default
```

Source code: http://j.mp/Mastering-React-1-8-Gist
Fiddle: http://j.mp/Mastering-React-1-8-Fiddle
state
So far our components are static and don't allow for dynamic behavior, which isn't very interesting and we would need to be able to make our components dynamic for them to be useful. React components have the concept of state to allow for dynamic behavior. The following code shows how we can use state to create some simple dynamic behavior:

```javascript
var HelloReact = React.createClass({
  getInitialState: function() {
    return {
      message: 'I am from default state'
    },
  },
  updateMessage: function(e) {
    this.setState({message: e.target.value});
  },
  render: function() {
    return (
      <div>
        <input type='text' onChange={this.updateMessage}/>
        <div>Hello React</div>
        <div>{this.state.message}</div>
      </div>
    );
  }
});

ReactDOM.render(<HelloReact />, document.getElementById('view'));
```

Go ahead and run the code and you will see the output as shown below:

Hello React
I am from default state
Now type something into the text box and you will see that the **I am from default state** will change to something dynamically as you type, as shown in the following screenshot:

![Screen shot](http://j.mp/Mastering-React-1-9-Gist)

**Source code: [http://j.mp/Mastering-React-1-9-Gist](http://j.mp/Mastering-React-1-9-Gist)**
**Fiddle: [http://j.mp/Mastering-React-1-9-Fiddle](http://j.mp/Mastering-React-1-9-Fiddle)**

**How it works**

We are seeing a few new concepts here. First, we are wiring up our dynamic property from the state collection using the following code:

```jsx
<div>{this.state.message}</div>
```

Now, anytime `this.state.message` changes we will see that change reflected in the browser because React will rerender our component. Next, we need to wire up our UI to allow us to update `this.state.message` in response to user input. To do this we will take advantage of the synthetic events' capabilities of React's virtual DOM.

The virtual DOM is described on React's website, as shown below:

"React abstracts away the DOM from you, giving a simpler programming model and better performance."

The virtual DOM exposes synthetic events and you can learn more about React's synthetic events here:

[https://facebook.github.io/react/docs/events.html](https://facebook.github.io/react/docs/events.html)

We use the following code to subscribe the `this.updateMessage` method to the `onChange` synthetic event:

```jsx
<input type='text' onChange={this.updateMessage} />
```
Now, anytime we change the text in our textbox, `this.updateMessage` will be called, which is shown in the following code:

```javascript
updateMessage: function(e) {
    this.setState({message: e.target.value});
},
```

Here we are capturing the synthetic event's argument, `e`, and then calling `this.setState` and passing in a JavaScript object with a message property that is set to the value of `e.target` (our text box). The `this.setState()` method is added to our React component by React and it will update the component's state with the properties that are defined in the JSON object that is passed in, and then rerender the component using the new state. Components in React are meant to be state machines and changing the state transitions the UI from one visual state to another.

Note that `this.setState()` method will merge the existing `this.state` with the object that is passed in. This means that you only need to specify the properties that you want to update as it will not delete any properties that are not defined in the JSON object, which are currently defined on `this.state`.

The only remaining detail in the code sample is how we are able to declare a default state by defining a `getInitialState()` method:

```javascript
getInitialState: function() {
    return {
        message: 'I am from default state'
    };
},
```

The object we return from the `getInitialState()` method will be used to initialize our component's state.

**Summary**

In this chapter we looked at the most basic and fundamental concepts in React. We saw how to create components, how to pass in data to them using props, and how to make them dynamic using state.

In the next chapter we will look at component composition and component lifecycle.
Where to buy this book

You can buy Mastering React from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.