Chapter 1: Getting started with React

Version	Release Date
0.3.0	2013-05-29
0.4.0	2013-07-17
0.5.0	2013-10-16
0.8.0	2013-12-19
0.9.0	2014-02-20
0.10.0	2014-03-21
0.11.0	2014-07-17
0.12.0	2014-10-28
0.13.0	2015-03-10
0.14.0	2015-10-07
15.0.0	2016-04-07
15.1.0	2016-05-20
15.2.0	2016-07-01
15.2.1	2016-07-08
15.3.0	2016-07-29
15.3.1	2016-08-19
15.3.2	2016-09-19
15.4.0	2016-11-16
15.4.1	2016-11-23
15.4.2	2017-01-06
15.5.0	2017-04-07
15.6.0	2017-06-13
15.6.1	2017-06-14
15.6.2	2017-09-25
16.0.0	2017-09-26
16.1.0	2017-11-09
16.1.1	2017-11-13
16.3.0	2018-03-29
16.3.1	2018-04-03
16.3.2	2018-04-16

Section 1.1: What is ReactJS?

ReactJS is an open-source, component based front end library responsible only for the **view layer** of the application. It is maintained by Facebook.

ReactJS uses virtual DOM based mechanism to fill in data (views) in HTML DOM. The virtual DOM works fast owning to the fact that it only changes individual DOM elements instead of reloading complete DOM every time

> A React application is made up of multiple **components**, each responsible for outputting a small, reusable piece of HTML. Components can be nested within other components to allow complex applications to be built out of simple building blocks. A component may also maintain internal state - for example, a TabList component may store a variable corresponding to the currently open tab.

React allows us to write components using a domain-specific language called JSX. JSX allows us to write our components using HTML, whilst mixing in JavaScript events. React will internally convert this into a virtual DOM, and will ultimately output our HTML for us.

React "*reacts*" to state changes in your components quickly and automatically to rerender the components in the HTML DOM by utilizing the virtual DOM. The virtual DOM is an in-memory representation of an actual DOM. By doing most of the processing inside the virtual DOM rather than directly in the browser's DOM, React can act quickly and only add, update, and remove components which have changed since the last render cycle occurred.

Section 1.2: Installation or Setup

ReactJS is a JavaScript library contained in a single file `react-<version>.js` that can be included in any HTML page. People also commonly install the React DOM library `react-dom-<version>.js` along with the main React file:

Basic Inclusion

```
<!DOCTYPE html>
<html>
    <head></head>
    <body>
    <script type="text/javascript" src="/path/to/react.js"></script>
    <script type="text/javascript" src="/path/to/react-dom.js"></script>
    <script type="text/javascript">
        // Use react JavaScript code here or in a separate file
    </script>
    </body>
</html>
```

To get the JavaScript files, go to the installation page of the official React documentation.

React also supports JSX syntax. JSX is an extension created by Facebook that adds XML syntax to JavaScript. In order to use JSX you need to include the Babel library and change `<script type="text/javascript">` to `<script type="text/babel">` in order to translate JSX to Javascript code.

```
<!DOCTYPE html>
<html>
    <head></head>
    <body>
    <script type="text/javascript" src="/path/to/react.js"></script>
    <script type="text/javascript" src="/path/to/react-dom.js"></script>
    <script src="https://npmcdn.com/babel-core@5.8.38/browser.min.js"></script>
    <script type="text/babel">
        // Use react JSX code here or in a separate file
    </script>
    </body>
</html>
```

Installing via npm

You can also install React using npm by doing the following:

```
npm install --save react react-dom
```

To use React in your JavaScript project, you can do the following:

```
var React = require('react');
var ReactDOM = require('react-dom');
```

```
ReactDOM.render(<App />, ...);
```

Installing via Yarn

Facebook released its own package manager named Yarn, which can also be used to install React. After installing Yarn you just need to run this command:

yarn add react react-dom

You can then use React in your project in exactly the same way as if you had installed React via npm.

Section 1.3: Hello World with Stateless Functions

Stateless components are getting their philosophy from functional programming. Which implies that: A function returns all time the same thing exactly on what is given to it.

For example:

```
const statelessSum = (a, b) => a + b;

let a = 0;
const statefulSum = () => a++;
```

As you can see from the above example that, statelessSum is always will return the same values given a and b. However, statefulSum function will not return the same values given even no parameters. This type of function's behaviour is also called as a *side-effect*. Since, the component affects somethings beyond.

So, it is advised to use stateless components more often, since they are *side-effect free* and will create the same behaviour always. That is what you want to be after in your apps because fluctuating state is the worst case scenario for a maintainable program.

The most basic type of react component is one without state. React components that are pure functions of their props and do not require any internal state management can be written as simple JavaScript functions. These are said to be `Stateless Functional Components` because they are a function only of `props`, without having any `state` to keep track of.

Here is a simple example to illustrate the concept of a `Stateless Functional Component`:

```
// In HTML
<div id="element"></div>

// In React
const MyComponent = props => {
    return <h1>Hello, {props.name}!</h1>;
};

ReactDOM.render(<MyComponent name="Arun" />, element);
// Will render <h1>Hello, Arun!</h1>
```

Note that all that this component does is render an `h1` element containing the `name` prop. This component doesn't keep track of any state. Here's an ES6 example as well:

```
import React from 'react'

const HelloWorld = props => (
    <h1>Hello, {props.name}!</h1>
)
```

```
HelloWorld.propTypes = {
    name: React.PropTypes.string.isRequired
}

export default HelloWorld
```

Since these components do not require a backing instance to manage the state, React has more room for optimizations. The implementation is clean, but as of yet no such optimizations for stateless components have been implemented.

Section 1.4: Absolute Basics of Creating Reusable Components

Components and Props

As React concerns itself only with an application's view, the bulk of development in React will be the creation of components. A component represents a portion of the view of your application. "Props" are simply the attributes used on a JSX node (e.g. `<SomeComponent someProp="some prop's value" />`), and are the primary way our application interacts with our components. In the snippet above, inside of SomeComponent, we would have access to `this.props`, whose value would be the object `{someProp: "some prop's value"}`.

It can be useful to think of React components as simple functions - they take input in the form of "props", and produce output as markup. Many simple components take this a step further, making themselves "Pure Functions", meaning they do not issue side effects, and are idempotent (given a set of inputs, the component will always produce the same output). This goal can be formally enforced by actually creating components as functions, rather than "classes". There are three ways of creating a React component:

- **Functional ("Stateless") Components**

```
const FirstComponent = props => (
    <div>{props.content}</div>
);
```

- **React.createClass()**

```
const SecondComponent = React.createClass({
    render: function () {
        return (
            <div>{this.props.content}</div>
        );
    }
});
```

- **ES2015 Classes**

```
class ThirdComponent extends React.Component {
    render() {
        return (
            <div>{this.props.content}</div>
        );
    }
}
```

These components are used in exactly the same way:

```
const ParentComponent = function (props) {
    const someText = "FooBar";
```

```
    return (
        <FirstComponent content={someText} />
        <SecondComponent content={someText} />
        <ThirdComponent content={someText} />
    );
}
```

The above examples will all produce identical markup.

Functional components cannot have "state" within them. So if your component needs to have a state, then go for class based components. Refer Creating Components for more information.

As a final note, react props are immutable once they have been passed in, meaning they cannot be modified from within a component. If the parent of a component changes the value of a prop, React handles replacing the old props with the new, the component will rerender itself using the new values.

See Thinking In React and Reusable Components for deeper dives into the relationship of props to components.

Section 1.5: Create React App

create-react-app is a React app boilerplate generator created by Facebook. It provides a development environment configured for ease-of-use with minimal setup, including:

- ES6 and JSX transpilation
- Dev server with hot module reloading
- Code linting
- CSS auto-prefixing
- Build script with JS, CSS and image bundling, and sourcemaps
- Jest testing framework

Installation

First, install create-react-app globally with node package manager (npm).

```
npm install -g create-react-app
```

Then run the generator in your chosen directory.

```
create-react-app my-app
```

Navigate to the newly created directory and run the start script.

```
cd my-app/
npm start
```

Configuration

create-react-app is intentionally non-configurable by default. If non-default usage is required, for example, to use a compiled CSS language such as Sass, then the eject command can be used.

```
npm run eject
```

This allows editing of all configuration files. N.B. this is an irreversible process.

Alternatives

Alternative React boilerplates include:

- enclave
- nwb
- motion
- rackt-cli
- budō
- rwb
- quik
- sagui
- roc

Build React App

To build your app for production ready, run following command

```
npm run build
```

Section 1.6: Hello World

Without JSX

Here's a basic example that uses React's main API to create a React element and the React DOM API to render the React element in the browser.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Hello React!</title>

    <!-- Include the React and ReactDOM libraries -->
    <script src="https://fb.me/react-15.2.1.js"></script>
    <script src="https://fb.me/react-dom-15.2.1.js"></script>

  </head>
  <body>
    <div id="example"></div>

    <script type="text/javascript">

      // create a React element rElement
      var rElement = React.createElement('h1', null, 'Hello, world!');

      // dElement is a DOM container
      var dElement = document.getElementById('example');

      // render the React element in the DOM container
      ReactDOM.render(rElement, dElement);

    </script>

  </body>
</html>
```

With JSX

Instead of creating a React element from strings one can use JSX (a Javascript extension created by Facebook for adding XML syntax to JavaScript), which allows to write

```
var rElement = React.createElement('h1', null, 'Hello, world!');
```

as the equivalent (and easier to read for someone familiar with HTML)

```
var rElement = <h1>Hello, world!</h1>;
```

The code containing JSX needs to be enclosed in a `<script type="text/babel">` tag. Everything within this tag will be transformed to plain Javascript using the Babel library (that needs to be included in addition to the React libraries).

So finally the above example becomes:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Hello React!</title>

    <!-- Include the React and ReactDOM libraries -->
    <script src="https://fb.me/react-15.2.1.js"></script>
    <script src="https://fb.me/react-dom-15.2.1.js"></script>
    <!-- Include the Babel library -->
    <script src="https://npmcdn.com/babel-core@5.8.38/browser.min.js"></script>

  </head>
  <body>
   <div id="example"></div>

    <script type="text/babel">

      // create a React element rElement using JSX
      var rElement = <h1>Hello, world!</h1>;

      // dElement is a DOM container
      var dElement = document.getElementById('example');

      // render the React element in the DOM container
      ReactDOM.render(rElement, dElement);

    </script>

  </body>
</html>
```

Section 1.7: Hello World Component

A React component can be defined as an ES6 class that extends the base `React.Component` class. In its minimal form, a component *must* define a `render` method that specifies how the component renders to the DOM. The `render` method returns React nodes, which can be defined using JSX syntax as HTML-like tags. The following example shows how to define a minimal Component:

```
import React from 'react'

class HelloWorld extends React.Component {
    render() {
```

```
        return <h1>Hello, World!</h1>
    }
}

export default HelloWorld
```

A Component can also receive `props`. These are properties passed by its parent in order to specify some values the component cannot know by itself; a property can also contain a function that can be called by the component after certain events occur - for example, a button could receive a function for its `onClick` property and call it whenever it is clicked. When writing a component, its `props` can be accessed through the `props` object on the Component itself:

```
import React from 'react'

class Hello extends React.Component {
    render() {
        return <h1>Hello, {this.props.name}!</h1>
    }
}

export default Hello
```

The example above shows how the component can render an arbitrary string passed into the `name` prop by its parent. Note that a component cannot modify the props it receives.

A component can be rendered within any other component, or directly into the DOM if it's the topmost component, using `ReactDOM.render` and providing it with both the component and the DOM Node where you want the React tree to be rendered:

```
import React from 'react'
import ReactDOM from 'react-dom'
import Hello from './Hello'

ReactDOM.render(<Hello name="Billy James" />, document.getElementById('main'))
```

By now you know how to make a basic component and accept `props`. Lets take this a step further and introduce `state`.

For demo sake, let's make our Hello World app, display only the first name if a full name is given.

```
import React from 'react'

class Hello extends React.Component {

    constructor(props){

        //Since we are extending the default constructor,
        //handle default activities first.
        super(props);

        //Extract the first-name from the prop
        let firstName = this.props.name.split(" ")[0];

        //In the constructor, feel free to modify the
        //state property on the current context.
        this.state = {
            name: firstName
        }
```

```
    } //Look maa, no comma required in JSX based class defs!

    render() {
        return <h1>Hello, {this.state.name}!</h1>
    }
}

export default Hello
```

Note: Each component can have it's own state or accept it's parent's state as a prop.

Codepen Link to Example.