

# **ReactJS**

## **Web App Development**

### **2nd edition**

**Learn one of the most  
popular javascript libraries**

An abstract graphic consisting of numerous overlapping, curved lines in vibrant colors including blue, magenta, yellow, cyan, and orange. The lines originate from the bottom left and fan out towards the top right, creating a sense of dynamic movement and complexity against a solid black background.

**Daniel Green**

# REACT JS: Web App Development

*Learn one of the most popular Javascript libraries*

---

**2<sup>nd</sup> edition**

Daniel Green

Copyright © 2015 by Daniel Green.

All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the author, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.



## Disclaimer

While all attempts have been made to verify the information provided in this book, the author doesn't assume any responsibility for errors, omissions, or contrary interpretations of the subject matter contained within. **The information provided in this book is for educational and entertainment purposes only. The reader is responsible for his or her own actions and the author does not accept any responsibilities for any liabilities or damages, real or perceived, resulting from the use of this information.**

The trademarks that are used are without any consent, and the publication of the trademark is without permission or backing by the trademark owner. All trademarks and brands within this book are for clarifying purposes only and are owned by the owners themselves, not affiliated with this document.

# Contents

## **Book Description**

## **Introduction**

## **Chapter 1 Definition**

*[Getting started with React Javascript](#)*

## **Chapter 2 Creating Components in React**

*[Adding data to the component](#)*

*[Creating a sub component of React](#)*

*[Adding state](#)*

*[Handling Events](#)*

## **Chapter 3 Comment Box in React**

## **Chapter 4 Breaking the User Interface into an Hierarchy of Components**

## **Chapter 5 Creating Forms in React**

## **Chapter 6 Animations in React**

## **Chapter 7 Two-way Binding Helpers in React**

## **Chapter 8 Transferring Props**

*[Transferring Manually](#)*

*[Transfer with JSX](#)*

*[Consumption and Transfer of the Same Prop](#)*

*[Transfer with an Underscore](#)*

## **Chapter 9 React and Browsers**

*[findDOMNode\(\) and Refs](#)*

*[More on Refs](#)*

*[Ref Callback](#)*

## **Conclusion**



# Book Description

This book is all about React Javascript. It begins by explaining what React is, and where it is used. The book also explains the origin of this language. A guide on how to get started with React is provided, including how to download the library and setting it up to be ready for use. You will also be guided on how to write your first React program.

Sometimes, your program might involve a combination of a HTML, a CSS, and a React code. After reading this book, you will know how to write these separately, and then run them so as to achieve your result. Components, which are a common feature in React, are deeply explored in this book.

The book discusses how to create components with no details left out. You will get to know how to add data to the components that you create in React, as well as creating subcomponents in React. The various ways in which one can add state to the components that have been created in React is also explored. This book will guide you on how to choose the best method for doing this in React. Events, in which you need your component to do something after selection, are discussed in detail.

After reading this book, you will get to know how to create a comment box in React. The user interface is explored in detail including breaking it into a number of components. Animations are also discussed.



The following topics are explored:

- Definition
- Creating Components in React
- Comment Box in React
- Breaking the User Interface into an Hierarchy of Components
- Creating Forms in React
- Animations in React
- Two-way Binding Helpers in React
- Transferring Props
- React and Browsers



# Introduction

Some years ago, developers, and especially web developers, experienced numerous problems when developing single page web applications. This problem persisted until Javascript React was introduced. This is a Javascript library which helps in solving problems experienced by programmers when creating web applications which are worth only a single page.

The good thing about the library is that it can be integrated with other Javascript libraries so as to enhance the functionality of your final web application. Many programmers and non-programmers think that this library cannot be used with other Javascript libraries, which is not the case. This means that this library can also be used to add functionality to the created components, in such a way that once a particular component has been selected, then it will do something that you specify.

The library is very good in supporting animations. The animation can involve either a single word or image, or multiple images or words. This explains why most websites with an image carousel in which image sliders are created for the web pages of the site are developed using this Javascript library. This also shows how good the library is in enhancing the user interface of web applications. With this library, the user interface can be broken into a number of components.





# Chapter 1

## Definition

This is a Javascript Library, and it is used for development of user interfaces. The library is open-source, and most people refer to it as React.js or ReactJS. The library was developed so as to solve the problems that programmers experience when developing applications which are only a single page. It is maintained by Instagram, Facebook, and a group which is made up of developers.

With React, one can develop large applications and use data which can change frequently. This can be done in a very simple manner. Note that it can only be used to develop the interface part of your application, since in terms of MVC (mode-view-controller), it only forms the view.

It is possible to integrate this library with other Javascript libraries such as AngularJS. Although it is used for development of the user interface alone, it is possible to use addons which react to events together with this library, so as to create amazing web applications.

In terms of DOM (Data Object Manipulation), it maintains a DOM which it comes with, meaning that it doesn't rely very much on the DOM of the browser.

This means that it is easy to know the parts of the DOM which have changed by comparing it with the one being stored. It can then update the DOM of the browser.

## ***Getting started with React Javascript***

You need to first download the library. This can be downloaded from GitHub. The downloaded file will always be in a zipped format. You have to extract it so that it can become usable. The files in which you write your code should be saved in the same directory where you have extracted the contents of the download. They should also be saved with an “.html” extension.

Let us demonstrate this by writing our first sample code in React:

Extract the contents of the file in a certain directory. Open your text editor of choice, such as notepadornotepad++ if you are using Windows. For Linux and Mac OS X users, use text editors of choice such as VIM in Linux and Emacs in Mac. Just write the following code in the editor:

```
React.render(  
    <h1>Hello, welcome to React!</h1>,  
    document.getElementById('hello')  
);
```

Once you have written the code, save the file with an “.html” extension and in the same directory where you extracted the contents of the downloaded file. Open the file with your browser, and observe the output. The following should be observed:

**Hello, welcome to React!**



From the figure shown above, it is very clear that the code has produced the words which we specified as the output.

It is also possible to write the React code separately. Write the following “*src/hello.js*” code:

```
React.render(  
  <h1>Hello, welcome to React!</h1>,  
  document.getElementById('hello')  
);
```

Just save the above file with a “.js” extension so as to imply that it is a Javascript file. This should then be referenced from the file saved with the “*.html*” extension as follows:

```
<script type="text/jsx" src="src/hello.js"></script>
```

The HTML file should then be updated as follows:

```
</head><!DOCTYPE html>  
  
<html>  
  
<head>  
  
<title>Hello, welcome to React!</title>  
  
<script src="build/react.js"></script>  
  
<!-- No need for JSXTransformer! -->  
  
<body>
```

```
<div id="hello"></div>
```

```
<script src="build/hello.js"></script>
```

```
</body>
```

```
</html>
```

You can then run your code, and you will observe the following output:

**Hello, welcome to React!**

It is very clear that we have achieved the same result, but by different methods.



# Chapter 2

## Creating Components in React

Components make it possible for React to call classes. To achieve this, one should do the following:

```
varMainV = React.createClass({});
```

After the above, the next step should be rendering it onto the screen. This can be achieved as follows:

```
React.renderComponent(newMainV(), document.body);
```

Let us demonstrate this with an example. Consider the following sample code:

```
varMainV = React.createClass({  
  
    getDefaultProps: function(){  
  
        return {  
  
    }  
  
    },  
  
    getInitialState: function(){  
  
        return {
```

```
}  
  
},  
  
render: function(){  
  
    return React.DOM.div(  
  
        className: 'root'  
  
    }, 'My div')  
  
}  
  
})
```

Just write the code, and then run it. Observe the output that you get. It should be as follows:



My div

The code outputs a “*div*” element which we have created together with the text which we have specified. This has been achieved very easily with React. It is important to note the use of the following methods:

- **getDefaultProps:** this method is callable only once, and it must be called before we can render the output on the screen. It has formed the first method to be called.
- **getInitialState:** the method can also be called once. It gives the default state of our component. Notice that it has been called after the method above;

- **render:** this method has the purpose of rendering the output or the result on the screen. It can only be called once.

## ***Adding data to the component***

It is possible to add to the component which we have created above. This can be done with a lot of ease. Consider the following:

Write the following Javascript code:

```
var MainV = React.createClass({  
  
  getDefaultProps: function(){  
  
        return {  
  
            r: null  
  
        }  
  
  },  
  
  getInitialState: function(){  
  
        return {  
  
      }  
  
  },  
  
  render: function(){  
  
        return React.DOM.div({  
  
            className: 'root'  
  
        },
```

```
_.map(this.props.r, function(r){  
  
    return React.DOM.div(  
  
        className: 'box'  
  
    )  
  
    )  
  
    }  
  
    })  
  
var getData = function getData(){  
  
    return [  
  
    {  
  
        color: "green",  
  
        value: "#0f0"  
  
    },  
  
    {  
  
        color: "red",  
  
        value: "#f00"  
  
    },  
  
    {  
  
        color: "cyan",  
  
        value: "#0ff"
```



```
},  
  
{  
  
    color: "blue",  
  
    value: "#00f"  
  
},  
  
{  
  
    color: "magenta",  
  
    value: "#f0f"  
  
},  
  
{  
  
    color: "black",  
  
    value: "#000"  
  
},  
  
{  
  
    color: "yellow",  
  
    value: "#ff0"  
  
}  
  
]  
  
}  
  
var r = getData();  
  
var mainV = new MainV({
```

```
r: r
```

```
});
```

```
React.renderComponent(mainV, document.body);
```

The HTML code for the same should be as follows:

```
<script type="text/javascript" src="http://underscorejs.org/underscore-min.js">  
</script>
```

The CSS code for the same should be as follows:

```
.box {  
  
    width: 45px;  
  
    height: 40px;  
  
    background-color: red;  
  
}
```

Just run the program, and observe the output which should be as follows:



We have created several boxes, but all have the same background color. This means that they have been joined together. To be able to differentiate between the different boxes, we should give each a different color. This can be done by updating the render method as follows:

The Javascript file should be as follows:

```
var MainV = React.createClass({  
  
  getDefaultProps: function(){  
  
    return {  
  
      r: null  
  
    }  
  
  },  
  
  getInitialState: function(){  
  
    return {  
  
    }  
  
  }
```

```
},

render: function(){

    return React.DOM.div({

        className: 'root'

    },

    _.map(this.props.r, function(r){

        return React.DOM.div({

            className: 'box',

            style: {

                backgroundColor: r.value

            }

        })

    })

)

}

})

var getData = function getData(){

    return [
```

```
{  
    color: "green",  
    value: "#0f0"
```

```
},
```

```
{  
    color: "red",  
    value: "#f00"
```

```
},
```

```
{  
    color: "cyan",  
    value: "#0ff"
```

```
},
```

```
{  
    color: "blue",  
    value: "#00f"
```

```
},
```

```
{  
    color: "magenta",  
    value: "#f0f"
```

```
},
```

```
{
```

```
    color: "black",  
    value: "#000"  
  },  
  
  {  
    color: "yellow",  
    value: "#ff0"  
  }  
]  
}  
  
var r = getData();  
  
var mainV = new MainV({  
  r: r  
});  
  
React.renderComponent(mainV, document.body);
```

The CSS file should be as follows:

```
.box {  
  
  width: 45px;  
  
  height: 40px;  
  
  background-color: red;  
  
}
```

The HTML file should be as follows:

```
<script type="text/javascript" src="http://underscorejs.org/underscore-min.js">
</script>
```

Just run the program, and observe the output. It should be as follows:

```
<script type="text/javascript" src="http://underscorejs.org/underscore-min.js">
</script>
```

Just run the program and observe the output. It should be as follows:

```
var MainV = React.createClass({
  getDefaultProps: function(){
    return {
      r: null
    }
  },
  getInitialState: function(){
    return {
    }
  },
  render: function(){
    return React.DOM.div({
```

**'root'**

**},**

**\_.map(this.props.r, function(r){**

**return new ItemRendererV(r);**

**})**

**)**

**}**

**})**

**var ItemRendererV = React.createClass({**

**getDefaultProps: function(){**

**return {**

**color: null,**

**value: null**

**}**

**},**

**render: function(){**

**return React.DOM.div({**

**className: 'box',**

**style: {**

**backgroundColor: this.props.value**

**}**



```
    }, this.props.color)

}

})
```

```
vargetData = function getData(){

return [

{

    color: "green",

    value: "#0f0"

},

{

    color: "red",

    value: "#f00"

},

{

    color: "cyan",

    value: "#0ff"

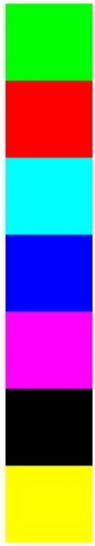
},

{

    color: "blue",

    value: "#00f"
```

```
    },  
  
    {  
  
        color: "magenta",  
  
        value: "#f0f"  
  
    },  
  
    {  
  
        color: "black",  
  
        value: "#000"  
  
    },  
  
    {  
  
        color: "yellow",  
  
        value: "#ff0"  
  
    }  
  
    ]  
  
    }  
  
    var rows = getData();  
  
        var mainV = new MainV({  
  
            rows: rows  
  
        });  
  
    React.renderComponent(mainV, document.body);
```



As shown in the output, each of the boxes is unique in terms of color. It is now easy to differentiate them. It would also be good if we add some text in each of the boxes. This is demonstrated below.

Then the Javascript code should be modified to the following:

```
varMainV = React.createClass({  
  
    getDefaultProps: function(){  
  
        return {  
  
            r: null  
  
        }  
  
    },  
  
    getInitialState: function(){  
  
        return {  
  
        }  
  
    },
```

```
render: function(){  
  
    return React.DOM.div(  
  
        className: 'root'  
  
    },  
  
    _.map(this.props.r, function(r){  
  
        return React.DOM.div(  
  
            className: 'box',  
  
            style: {  
  
                backgroundColor: r.value  
  
            }  
  
            }, r.color)  
  
        })  
  
    )  
  
    }  
  
    })  
  
}
```

```
vargetData = function getData(){  
  
    return [  
  
    {  
  
        color: "green",  
  
    }  
  
    ]  
  
}
```

**value: “#0f0”**

**},**

**{**

**color: “red”,**

**value: “#f00”**

**},**

**{**

**color: “cyan”,**

**value: “#0ff”**

**},**

**{**

**color: “blue”,**

**value: “#00f”**

**},**

**{**

**color: “magenta”,**

**value: “#f0f”**

**},**

**{**

**color: “black”,**

**value: “#000”**

```
},  
  
{  
  
    color: “yellow”,  
  
    value: “#ff0”  
  
}  
  
]  
  
}  
  
var r = getData();  
  
    var mainV = new MainV({  
  
        r: r  
  
        });  
  
React.renderComponent(mainV, document.body);
```

The CSS code should be as follows:

```
.box {  
  
    width: 45px;  
  
    height: 40px;  
  
    background-color: red;  
  
}
```

The HTML file should have the following code:

```
<script type="text/javascript" src="http://underscorejs.org/underscore-min.js">  
</script>
```

Just write the program, and run it. Observe the output, which should be as follows:



Text has been added to each of the boxes as shown in the above figure.

## *Creating a subcomponent of React*

This can be done as easily as we have been doing in the previous sections. We just have to create an ItemRenderer with only a single task. This will have the task of rendering the data. Write the following Javascript code:

```
var MainV = React.createClass({  
  
    getDefaultProps: function(){  
  
        return {  
  
            r: null  
  
        }  
  
    },  
  
    getInitialState: function(){  
  
        return {  
  
            }  
  
        },  
  
        render: function(){  
  
    return React.DOM.div({  
  
        className: 'root'  
  
    },  
  
    _.map(this.props.r, function(r){
```



```
        return new ItemRendererV(r);

    })

)

}

})

var ItemRendererV = React.createClass({

    getDefaultProps: function(){

        return {

            color: null,

            value: null

        }

    },

    render: function(){

        return React.DOM.div({

            className: 'box',

            style: {

                backgroundColor: this.props.value

            }

        }, this.props.color)

    }

})
```

```
vargetData = function getData(){  
    return [  
    {  
        color: "green",  
        value: "#0f0"  
    },  
    {  
        color: "red",  
        value: "#f00"  
    },  
    {  
        color: "cyan",  
        value: "#0ff"  
    },  
    {  
        color: "blue",  
        value: "#00f"  
    },  
    {  
        color: "magenta",  
        value: "#f0f"  
    }  
    ]  
}
```

```
},  
  
{  
  
  color: "black",  
  
  value: "#000"  
  
},  
  
{  
  
  color: "yellow",  
  
  value: "#ff0"  
  
}  
  
]  
  
}  
  
var rows = getData();  
  
    var mainV = new MainV({  
  
        rows: rows  
  
    });  
  
React.renderComponent(mainV, document.body);
```

Add the following CSS code:

```
.box {  
  
    width: 45px;  
  
    height: 40px;
```

```
background-color: red;  
  
}
```

Add the following HTML code:

```
<script type="text/javascript" src="http://underscorejs.org/underscore-min.js">  
</script>
```

Just run the program, and observe the output. It should be as follows:



What we have done is that we have just created a new instance of the ItemRenderer, and then we have passed it into our object.

## *Adding state*

To change the state, use the following method:

```
this.setState({key: 'value'})
```

It is good to avoid changing the state using the following method:

```
this.state.key = 'other value'
```

If you need to be able to access the state immediately after it has changed, use the following method:

```
this.setState({key: 'other value'}, function(){alert(this.state.key)})
```

Let us demonstrate this using an example. Write the following code:

```
var MainV = React.createClass({  
  
  getInitialState: function(){  
  
    return {  
  
      rows: null  
  
    }  
  
  },
```

```
getInitialState: function(){  
    return {  
    }  
},  
  
render: function(){  
    return React.DOM.div({  
        className: 'root'  
    },  
    _.map(this.props.r, function(r){  
        return new ItemRendererV(r);  
    })  
    )  
    }  
    })  
  
var ItemRendererV = React.createClass({  
    getDefaultProps: function(){  
        return {  
            color: null,  
            value: null  
        }  
    },  
    },
```

```
getInitialState: function(){  
    return {  
        selected: false  
    }  
},  
  
render: function(){  
    return React.DOM.div({  
        className: 'box ' + (this.state.selected ? 'selected' : 'unselected'),  
        onClick: function(){this.setState({selected:  
            !this.state.selected})}.bind(this),  
        style: {  
            backgroundColor: this.props.value  
        }  
    }, this.props.color)  
}  
})  
  
var getData = function getData(){  
    return [  
    {  
        color: "green",  
        value: "#0f0"
```

```
},  
  
{  
  
    color: "red",  
  
    value: "#f00"  
  
},  
  
{  
  
    color: "cyan",  
  
    value: "#0ff"  
  
},  
  
{  
  
    color: "blue",  
  
    value: "#00f"  
  
},  
  
{  
  
    color: "magenta",  
  
    value: "#f0f"  
  
},  
  
{  
  
    color: "black",  
  
    value: "#000"  
  
},
```



```
{  
  color: "yellow",  
  value: "#ff0"  
}  
  
]  
  
}  
  
var r = getData();  
  
var mainV = new MainV({  
  r: r  
});  
  
React.renderComponent(mainV, document.body);
```

Add the following CSS code:

```
.box {  
  width: 45px;  
  height: 40px;  
  background-color: red;  
}
```

Add the following HTML code:

```
<script type="text/javascript" src="http://underscorejs.org/underscore-min.js">
```

**</script>**

Just write the program, and then run it. The output should be as follows:



## ***Handling Events***

Sometimes, after selection of an item, you might need the parent view to be updated. This can be done during the instantiation of the item renderer, in which a prop value with a function is added. The function will be called by the item render when necessary.

Let us demonstrate this using an example.

Write the following Javascript code:

```
varMainV = React.createClass({  
  
getDefaultProps: function(){  
  
    return {  
  
        r: null  
  
    }  
  
    },  
  
getInitialState: function(){  
  
    return {  
  
    }  
  
    },  
  
    __onItemRendererSelect: function(d){
```

```
alert(d.color)

},

render: function(){

var _this = this;

return React.DOM.div({

    className: 'root'

},

_.map(this.props.r, function(r){

    return new IRendererV({

        d: r,

        onSelect: _this.__onItemRendererSelect

    });

})

)

}

})

var IRendererV = React.createClass({

getDefaultProps: function(){
```

```
    return {  
      d: {  
        color: null,  
        value: null  
      },  
      onSelect: null  
    }  
  },  
  getInitialState: function(){  
    return {  
      selected: false  
    }  
  },  
  __onClick: function(){  
    this.setState({selected: !this.state.selected}, function(){  
      if(this.state.selected&&this.props.onSelect){  
        this.props.onSelect(this.props.d);  
      }  
    })  
  },  
  render: function(){
```

```
return React.DOM.div({
    className: 'box ' + (this.state.selected ? 'selected' : 'unselected'),
    onClick: this.__onClick,
    style: {
        backgroundColor: this.props.d.value
    }, this.props.d.color)
})

var getData = function getData(){
    return [
        {
            color: "green",
            value: "#0f0"
        },
        {
            color: "red",
            value: "#f00"
        },
        {
            color: "cyan",
```

```
        value: "#0ff"
    },
    {
        color: "blue",
        value: "#00f"
    },
    {
        color: "magenta",
        value: "#f0f"
    },
    {
        color: "black",
        value: "#000"
    },
    {
        color: "yellow",
        value: "#ff0"
    }
]

var r= getData();
```

```
varmainV = new MainV({  
  
  r: r  
  
});  
  
React.renderComponent(mainV, document.body);
```

The CSS code should be written as follows:

```
.box {  
  
  width: 40px;  
  
  height: 40px;  
  
  background-color: red;  
  
}
```

```
.selected {  
  
  opacity: 1;  
  
}
```

```
.unselected {  
  
  opacity: 0.5;  
  
}
```

The following is the HTML code for the same:



```
<script type="text/javascript" src="http://underscorejs.org/underscore-min.js">  
</script>
```

Once you have written all the above, just run the program. Observe the output. It should be as follows:



Note that we have passed both data and function so as to get the above output. The item render has the task of calling the “*onSelect*” method.

To make our colors a bit nicer, write the following Javascript code:

```
var MainV = React.createClass({  
  getDefaultProps: function(){  
    return {  
      r: null  
    }  
  },  
  getInitialState: function(){
```

```
return {  
  
}  
  
},  
  
__onItemRendererSelect: function(d){  
  
  // alert(d.color)  
  
},  
  
render: function(){  
  
  var _this = this;  
  
  return React.DOM.div({  
  
    className: 'root'  
  
  },  
  
    _.map(this.props.r, function(r){  
  
      return new IRendererV({  
  
        d: r,  
  
        onSelect: _this.__onItemRendererSelect  
  
      });  
  
    })  
  
  )  
  
}  
  
})  
  
var IRendererV = React.createClass({
```

```
getDefaultProps: function(){  
  
  return {  
  
    d: {  
  
      color: null,  
  
      value: null  
  
    },  
  
    onSelect: null  
  
  }  
  
},  
  
getInitialState: function(){  
  
  return {  
  
    selected: false  
  
  }  
  
},  
  
__onClick: function(){  
  
  this.setState({selected: !this.state.selected}, function(){  
  
    if(this.state.selected&&this.props.onSelect){  
  
      this.props.onSelect(this.props.d);  
  
    }  
  
  })  
  
},
```

```
render: function(){  
  
  return React.DOM.div({  
  
    className: 'box ' + (this.state.selected ? 'selected' : 'unselected'),  
  
    onClick: this.__onClick,  
  
    style: {  
  
      backgroundColor: this.props.d.value  
  
    }  
  
    }, this.props.d.color)  
  
  }  
  
  })  
  
  var getData = function getData(){  
  
    return [  
  
      {  
  
        color: "First color",  
  
        value: "#3498db"  
  
      },  
  
      {  
  
        color: "second color",  
  
        value: "#34495e"  
  
      },  
  
      {
```

**color: “third color”,**

**value: “#2ecc71”**

**},**

**{**

**color: “fourth color”,**

**value: “#e74c3c”**

**},**

**{**

**color: “fifth color”,**

**value: “#f1c40f”**

**},**

**{**

**color: “sixth color”,**

**value: “#1abc9c”**

**},**

**{**

**color: “seventh color”,**

**value: “#ecf0f1”}**

**]**

```
}  
  
var rows = getData();  
  
var mainV = new MainV({  
  
    r: r  
  
});  
  
React.renderComponent(mainV, document.body);
```

The CSS code should be as follows:

```
.box {  
  
    width: 60px;  
  
    height: 60px;  
  
    background-color: black;  
  
    cursor: pointer;  
  
    -webkit-transition: opacity 0.4s;  
  
    -moz-transition: opacity 0.4s;  
  
    transition: opacity 0.4s;  
  
}  
  
    .selected {  
  
        opacity: 0.85;  
  
    }  
  
    .selected:hover {
```

```
        opacity: 1;

    }

    .unselected {

        opacity: 0.5;

    }

    .unselected:hover {

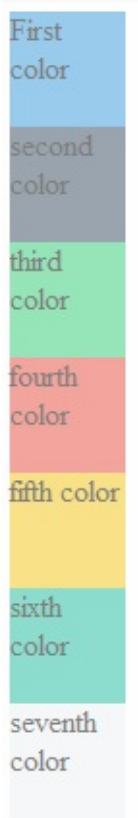
        opacity: 0.65;

    }
```

The HTML code should be as follows:

```
<script type="text/javascript" src="http://underscorejs.org/underscore-min.js">
</script>
```

Once you have written the program, just run it. The following output will be observed:



Notice that the look of the colors has been enhanced. We have also added a kind of transition to our code.





# Chapter 3

## Comment Box in React

Just open your editor and write the following code:

```
<!-- index.html -->

<html>

<head>

<title>React component</title>

<scriptsrc="https://github/react-0.13.3.js"></script>

<scriptsrc="https://github/JSXTransformer-0.13.3.js"></script>

<scriptsrc="https://code.jquery.com/jquery-2.1.3.min.js"></script>

</head>

<body>

<div id="content"></div>

<script type="text/jsx">

varCommentBox = React.createClass({

render: function() {

return (

<div className="commentBox">
```

**Hello there, this is a comment box.**

```
</div>
```

```
);
```

```
}
```

```
});
```

```
React.render(  
  <CommentBox />,  
  document.getElementById('content')
```

```
);
```

```
</script>
```

```
</body>
```

```
</html>
```

The code will give you the following result:

Hello there, this is a comment box.

The output shows a comment box. What we have done is that we have used the “React.createClass()” method to create a new React component.



## Chapter 4

# Breaking the User Interface into an Hierarchy of Components

This is possible with React Javascript. Write the following Javascript code:

```
var AttributeCategoryRow = React.createClass({  
  render: function() {  
    return (<tr><th colspan="2">{this.props.category}</th></tr>);  
  }  
});
```

```
var AttributeRow = React.createClass({  
  render: function() {  
    var name = this.props.attribute.stocked ?  
      this.props.attribute.name :  
      <span style={{color: 'red'}}>  
        {this.props.attribute.name}  
      </span>;
```

```

        return (

<tr>

<td>{name}</td>

<td>{this.props.attribute.value}</td>

</tr>

);

}

});

var AttributeTable = React.createClass({

    render: function() {

        var rs = [];

        var lCategory = null;

        this.props.attributes.forEach(function(product) {

            if (attribute.category !== lCategory) {

                rs.push(<AttributeCategoryRow category={attribute.category} key=
{attribute.category} />);

            }

            rs.push(<AttributeRow attribute={attribute} key={attribute.name} />);

            lCategory = attribute.category;

        });

        return (

```

```
<table>
```

```
<thead>
```

```
<tr>
```

```
<th>Attribute</th>
```

```
<th>Value</th>
```

```
</tr>
```

```
</thead>
```

```
<tbody>{rs}</tbody>
```

```
</table>
```

```
);
```

```
}
```

```
});
```

```
var SBar = React.createClass({
```

```
  render: function() {
```

```
    return (
```

```
<form>
```

```
<input type="text" placeholder="Search..." />
```

```
<p>
```

```
<input type="checkbox" />
```

```
{ ' }
```

```
Show the available students only
```

</p>

</form>

);

}

});

var FAttributeTable = React.createClass({

render: function() {

return (

<div>

<SearchBar />

<AttributeTable attributes={this.props.attributes} />

</div>

);

}

});

var ATTRIBUTES = [

{category: 'Form 1', price: '12', stocked: true, name: 'Male'},

{category: 'Form 1', price: '14', stocked: true, name: 'Female'},

{category: 'Form 1', price: '15', stocked: false, name: 'footballers'},

{category: 'Form 2', price: '17', stocked: true, name: 'Male students'},

{category: 'Form 2', price: '14', stocked: false, name: 'Female students'},



```
{category: 'Form 2', price: '20', stocked: true, name: 'football students'}
```

```
];
```

```
React.render(<FAttributeTable attributes={ATTRIBUTES} />, document.body);
```

Add the following CSS code to it:

```
body {
```

```
    padding: 20px
```

```
}
```

The following should be the HTML code for the same:

```
<script src="https://facebook.github.io/react/js/jsfiddle-integration.js"></script>
```

Just write the above program, and then run it. You will then observe the following output:

<input type="text" value="Search..."/>	
<input type="checkbox"/> Show the available students only	
<b>Attribute</b>	<b>Value</b>
<b>Form 1</b>	
Male	12
Female	14
footballers	15
<b>Form 2</b>	
Male students	17
Female students	14
football students	20

What we have done is that a box has been drawn around each of the components. A name has then been given to each of the boxes. Notice that each of the above components only

has a single task. If the component grows, then divide it into a number of subcomponents.

The header of the table which contains the “*Attribute*” and the “*Value*” labels is a component on its own. However, it is also possible to handle each separately if you need to do it that way, so do not be limited to the way we have done it above. A good example is when it grows, meaning that you might have to separate its components into a number of subcomponents. Notice that the components have been arranged into an hierarchy in which components to be placed inside other components should be treated as children.

It is possible to implement this statically with React. In this case, since we already have our hierarchy, we can come up with an app that will render our data into a user interface, but there will be no interactivity. With this, much typing will be needed compared to thinking. In the latter version, much thinking will be needed compared to typing.

For you to implement this, you must make it in such a way that components will use other components for the purpose of passing data. Props, which are normally used for the passage of data from the parent to child, are normally used in this case. The app can be developed using the top-down approach, in which you start with components higher up in the hierarchy or bottom-up, in which you start with the components down in the hierarchy.

However, in the case of simple apps, the top-down approach is highly recommended while for large and complex apps, the bottom-up approach is used. In this case, we will come up with a set of reusable components. Since we are trying to make the app static, we will only use the “*render()*” method. The components located at the top of the hierarchy will have

the task of taking the data model as a prop. In case changes are made to the underlying data model, then it will be easy to update the user interface after calling the “*React.render()*” method.

Just write the following Javascript code:

```
varAttributeCategoryRow = React.createClass({  
  
render: function() {  
  
    return (<tr><thcolSpan=“2”>{this.props.category}</th></tr>);  
  
    }  
  
});  
  
varAttributeRow = React.createClass({  
  
    render: function() {  
  
        var name = this.props.attribute.stocked ?  
  
        this.props.attribute.name :  
  
        <span style={{color: ‘red’}}>  
  
            {this.props.attribute.name}  
  
        </span>;  
  
        return (  
  
            <tr>  
  
                <td>{name}</td>  
  
                <td>{this.props.attribute.value}</td>  
  
            </tr>
```

```

    );

    }

});

var AttributeTable = React.createClass({

    render: function() {

        var rs = [];

        var lCategory = null;

        this.props.attributes.forEach(function(attribute) {

            if (attribute.category !== lCategory) {

                rs.push(<AttributeCategoryRow category={attribute.category} key=
                {attribute.category} />);

            }

            rs.push(<AttributeRow attribute={attribute} key={attribute.name} />);

            lCategory = attribute.category;

        });

        return (

            <table>

                <thead>

                    <tr>

                        <th>Attribute</th>

                        <th>Value</th>

```

```
</tr>
```

```
</thead>
```

```
<tbody>{rs}</tbody>
```

```
</table>
```

```
);
```

```
}
```

```
});
```

```
var SBar = React.createClass({
```

```
  render: function() {
```

```
    return (
```

```
      <form>
```

```
        <input type="text" placeholder="Search..." />
```

```
        <p>
```

```
          <input type="checkbox" />
```

```
            { ' }
```

```
            Show the available students only
```

```
        </p>
```

```
      </form>
```

```
    );
```

```
  }
```

```
});
```

```
var FAttributeTable = React.createClass({

    render: function() {

        return (

            <div>

                <SearchBar />

                <AttributeTable attributes={this.props.attributes} />

            </div>

        );

    }

});

var ATTRIBUTES = [

    {category: 'Form 1', price: '12', stocked: true, name: 'Male'},

    {category: 'Form 1', price: '14', stocked: true, name: 'Female'},

    {category: 'Form 1', price: '15', stocked: false, name: 'footballers'},

    {category: 'Form 2', price: '17', stocked: true, name: 'Male students'},

    {category: 'Form 2', price: '14', stocked: false, name: 'Female students'},

    {category: 'Form 2', price: '20', stocked: true, name: 'football students'}

];

React.render(<FAttributeTable attributes={ATTRIBUTES} />, document.body);
```

The CSS code should be as follows:

```
body {  
  
    padding: 20px  
  
}
```

Add the following HTML code:

```
<script src="https://facebook.github.io/react/js/jsfiddle-integration.js"></script>
```

Once you have written the above program, just run it and observe the resulting output. It will be as follows:

☐ Only show products in stock

Name	Price
<b>Form 1</b>	
Male	12
Female	14
footballers	15
<b>Form 2</b>	
Male students	17
Female students	14
football students	20





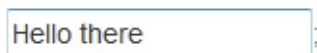
# Chapter 5

## Creating Forms in React

It is possible to create forms in React which allow for user interaction. Both controlled and uncontrolled form components can be created. Write the following Javascript code and run it:

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

After running it, observe the output. It should be as follows:



The result shows a text box with the text which we specified. This should be the default text for the text box. However, you might need to change this default text. Consider the following Javascript code:

```
getInitialState: function() {  
  
    return {value: 'Hello there'};  
  
},
```

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

After running the above code, you will notice that it will be possible for a user to provide other input into the text box rather than the one displayed by default. Suppose that you would like to do a truncation on what the user provides as input into the text box. This is a good feature in websites. Let's say that you want to truncate this input into the first 150 characters. This can be done as follows:

```
handleChange: function(event) {  
  
    this.setState({value: event.target.value.substr(0, 150)});  
  
}
```

The above function will truncate what the user enters as input into the first 150 characters. You now know how to create controlled components in React. It is also possible to create uncontrolled components in this language. Suppose that you want the text box to contain nothing in it, so that the user can provide the input that he or she wishes. This can be done as follows:

```
render: function() {
```

```
return<input type="text" />;  
}
```

Just write the code, and then run it. The following output will be observed:

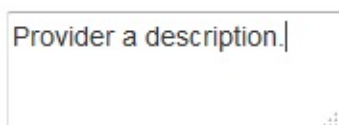


The text box contains nothing in it. The user can then provide the input of his or her choice. This is an example of an uncontrolled element in React. Notice the use of the *“render”* function.

Another component of a form in React is the Textarea. This will allow you to provide much input to it, especially when describing something. Consider the following code:

```
render: function() {  
  
    return<textarea name="textarea">Provider a description.</textarea>  
  
/>;  
}
```

You can write and run the above program. It will provide the following output:



The figure shows the text area which we have just created. With this, the user can provide multiple lines when describing something. We have also provided a default text for the text area, but this is not a must. You might not be interested in this.

Sometimes, you might need to select your own choice from a set of provided options. This can be accomplished by use of a “*select*” in React. This is how a select is created in React:

```
<select value=“School”>  
  
<option value=“university”>University</option>  
  
<option value=“college”>College</option>  
  
<option value=“secondary”>Secondary</option>  
  
<option value=“primary”>Primary</option>  
  
<option value=“nursery”>Nursery</option>  
  
</select>
```

The above program should give the following output:



On clicking the arrow facing downwards, you will see the options which we specified. You can select the choice that you want from the ones being provided. The above component is controlled. However, you may want to make it uncontrolled. In this case, you will have to provide a default value for the same.





# Chapter 6

## Animations in React

React has an add-on component called “*ReactTransitionGroup*” which is a low-level API responsible for the creation of animations in the language. “*ReactCSSTransitionGroup*” is also used for enhancement of the CSS part of the transition. It is a high level API, and programmers use it to perform an animation whenever a component of React leaves or enters the DOM.

The “*ReactCSSTransitionGroup*” is responsible for implementation of the interface in which all the elements to be used in the animation are wrapped. Consider the following example in which the elements of a list fade in and out:

```
varRCSSTransitionGroup = React.addons.CSSTransitionGroup;
```

```
    varTdoList = React.createClass({
```

```
        getInitialState: function() {
```

```
            return {items: ['hello', 'there', 'click', 'here']};
```

```
    },
```

```
    handleAdd: function() {
```

```
        var nItems =
```

```
        this.state.items.concat([prompt('Key in some text')]);
```

```

    this.setState({items: nItems});

},

handleRemove: function(j) {

    varnItems = this.state.items;

    nItems.splice(j, 1);

    this.setState({items: nItems});

},

render: function() {

    varitms = this.state.items.map(function(item, j) {

return (

    <div key={itm} onClick={this.handleRemove.bind(this, j)}>

{itm}

</div>

);

    }.bind(this));

return (

<div>

<button onClick={this.handleAdd}>Create Item</button>

<ReactCSSTransitionGrouptransitionName="sample">

{itms}

</RCSSTransitionGroup>

```



**</div>**

**);**

**}**

**});**

Just write the above program, and then run it. The first output of the program would appear as follows:

hello

This is a text which we specified within our program. The program should animate the text which we specified. The next word to be animated is “*there*.” It will be as shown below:

there

This is the second word which we specified. This will continue until the last word which we had specified has been animated. The animation happens by fading in and out of the elements of the lists which we specify.

In case you need to use CSS so as to trigger the transition, add the following CSS code and a new item in the list to be translated:

**.sample-enter {**

**opacity: 0.01;**

```
    transition: opacity .5s ease-in;
}

.sample-enter.sample-enter-active {

    opacity: 1;

}
```

What will happen is that after removing an element from the list, the function will keep it in the DOM. If you animate, you will be notified that an animation which was expected has not happened. The reason is because the function “*ReactCSSTransitionGroup*” will keep the elements that you specify in the page until the transition is complete. Add this CSS to your program:

```
.sample-leave {

    opacity: 1;

    transition: opacity .5s ease-in;

}

.sample-leave.sample-leave-active {

    opacity: 0.01;

}
```

Note that the above program will only show a single transition. It is possible for you to add an extra transition to the animation after the initial transition. This can be done using

the function “*transitionAppear*”. Consider the following program, which uses the above function after setting its value to “*true*.”

```
render: function() {  
  
    return (  
  
    <ReactCSSTransitionGrouptransitionName=“sample” transitionAppear=“true”>  
  
        <h1>Fade the initial mount</h1>  
  
    </ReactCSSTransitionGroup>  
  
    );  
  
}
```

The following should be the CSS code for the above:

```
.sample-appear {  
  
    opacity: 0.01;  
  
    transition: opacity .5s ease-in;  
  
}  
  
.sample-appear.sample-appear-active {  
  
    opacity: 1;  
  
}
```

In order for the animation to be applied to the children, the prop “*transitionAppear*” should be set to “*true*.” The “*ReactCSSTransitionGroup*” can also be mounted to the DOM.

Consider the example shown below:

```

render: function() {

    var itms = this.state.itms.map(function(item, j) {

        return (

<div key={itm} onClick={this.handleRemove.bind(this, j)}>

<ReactCSSTransitionGroup transitionName="sample">

    {itm}

</ReactCSSTransitionGroup>

</div>

);

    }, this);

    return (

<div>

<button onClick={this.handleAdd}>Create Item</button>

{itms}

</div>

);

}

```

Just try to run the program, and observe what will happen. There will be no observed result. The reason is because the “*ReactCSSTransitionGroup*” was mounted along the new item. To solve this problem, mount the new item within the “*ReactCSSTransitionGroup*”.

Note that it is possible for the children of “*ReactCSSTransitionGroup*” to be zero or one. This means that it is possible for a single element which enters or leaves to be animated. Consider the following program:

```
var ReactCSSTransitionGroup = React.addons.CSSTransitionGroup;
```

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

The program shows how to implement a simple image carousel in React Javascript. You need to change the “*imgSc*” to the name of your image for the image to be animated. Notice that we have used only a single image in the above example. However, it is

possible to animate multiple images in an image carousel created in React.

You might also want to disable an animation. This can be done by setting the “*transitionLeave*” to “*false*” or “*transitionEnter*” to “*false*.” The effect will be the same in both cases. This feature can be applied if you want an enter animation but no leave animation. Also, note that with the “*ReactTransitionGroup*” there is no way on how you can be notified in case a transition ends. You also can not perform anything else which is complex on your animation with this.



# Chapter 7

## Two-way Binding Helpers in React

React allows the data to flow only in one way, that is, from the owner to the child. This is due to the use of “*ReactLink*.” It employs the use of the mechanism of the Von Neumann Architecture of computers.

However, you might need to read some data and then flow it back to the program in your application. A good example is when you are creating forms. In this case, you might need to update the state of React once you have received some input from the user. This can also be applicable when the layout has been created in Javascript, and then you want to react to any changes in the size of a DOM element.

In React, this can be easily implemented, in which you will listen to a “*change*” event, read from the source of your data, and then finally call the “*setState()*” on any of your components. If you need to create programs which are easy to understand, then you can close the loop for data flow.

Consider the sample program given below:

```
var NLink = React.createClass({  
  
  getInitialState: function() {
```



```

        return {message: 'Hello there'};

    },

    handleChange: function(event) {

        this.setState({message: event.target.value});

    },

    render: function() {

        var mssge = this.state.mssge;

        return <input type="text" value={mssge} onChange={this.handleChange}
/>;

    }

});

```

We have just created a simple form without making use of “*ReactLink*.” Just run the program, and observe the output. It will be as follows:

The form works just in the right way. However, in case the form has several fields, we might need to be more verbose. With “*ReactLink*,” we can save some typing time. Consider the following code:

```

var WLink = React.createClass({

    mixins: [React.addons.LinkedStateMixin],

    getInitialState: function() {

```

```

    return {message: 'Hello there'};

},

render: function() {

    return<input type="text" valueLink={this.linkState('message')} />;

}

});

```

Notice the use of “*ReactSateMixin*” which has the task of adding the method “*linkSate()*” to the React component. This will, in turn, give a “*ReactLink*” object which provides the current state of the React, and a callback to it so as to change this state. Once you have run the program, a text field will be observed as shown below:

Objects which belong to the “*ReactLink*” can be passed either up or down the tree as props. This means that two-way binding can easily be set between states higher in the hierarchy and those deep in the hierarchy. The first phase of “*ReactLink*” is where you create it, and then the second phase involves using it.

If you need to create a *ReactLink* without “*LinkedStateMixin*,”do the following:

```

VarWMixin = React.createClass({

    getInitialState: function() {

        return {message: 'Hello there'};

    },

```

```
handleChange: function(newValue) {  
    this.setState({message: newValue});  
},  
  
render: function() {  
    var vLink = {  
        value: this.state.message,  
        requestChange: this.handleChange  
    };  
  
    return <input type="text" valueLink={vLink} />;  
}  
});
```

It is very clear that “*ReactLink*” objects contain only a value and a “*RequestChange*” prop.

If you need to create a “*ReactLink*” without a valueLink, do the following:

```
var WLink = React.createClass({  
    mixins: [React.addons.LinkedStateMixin],  
    getInitialState: function() {  
        return {message: 'Hello there'};  
    },  
  
    render: function() {  
        var vLink = this.linkState('message');
```

```
    varhandleChange = function(f) {  
        valueLink.requestChange(f.target.value);  
    };  
    return<input type="text" value={vLink.value} onChange={handleChange} />;  
}  
});
```

This will give you a text field, but without a valueLink.



# Chapter 8

## Transferring Props

Programmers want to create abstraction while programming in React. The outer part will have a component that can be used for performing something which is complex.

## *Transferring Manually*

You need to pass the properties down most of the time. This is to make sure that only a subset of the inner API has been exposed, and the one that is intended to work.

Consider the following program code:

```
var FCheckbox = React.createClass({  
  
  render: function() {  
  
    var fClass = this.props.checked ? 'FancyChecked' : 'FancyUnchecked';  
  
    return (  
  
      <div className={fClass} onClick={this.props.onClick}>  
  
        {this.props.children}  
  
      </div>  
  
    );  
  
  }  
  
  });  
  
  React.render(  
  
    document.getElementById('sample')  
  
  );
```

Just write the program, and then run it. You will get the following output:

Hello there!

We have created FancyCheckBox in React. The transfer has been done manually.



## ***Transfer with JSX***

Passing every property along can be tedious and fragile most of the time. To avoid this, it is recommended that you use destructuring assignment together with rest properties for the purpose of extracting unknown properties. All the properties which need to be consumed should be listed out and then followed by “... *other*”. This is demonstrated below:

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

With this, all the props will be passed, with the exception of the ones being consumed. Consider the program code shown below:

```
var FCheckbox = React.createClass({  
  
  render: function() {  
  
    var { checked, ...other } = this.props;  
  
    var fClass = checked ? 'FancyChecked' : 'FancyUnchecked';  
  
    // `other` contains { onClick: console.log } but has no checked property  
  
    return (  
  
      <div {...other} className={fClass} />  
  
    );  
  
  }  
  
  });
```

```

React.render(

  <FCheckbox checked={true} onClick={console.log.bind(console)}>

    Hello there!

  </FancyCheckbox>,

  document.getElementById('sample')

);

```

Just write the above program, and then run it. The output will be as follows:

```

Hello there!

```

This shows that we have successfully created our fancy checkbox. The prop “*checked*” is a DOM attribute. In case you fail to pass the property in the way shown above, then you might end up passing it along. If you are passing unknown “*other*” props then use the property as shown below:

```

var FCheckbox = React.createClass({

  render: function() {

    var fClass = this.props.checked ? 'FancyChecked' :

      'FancyUnchecked';

    // ANTI-PATTERN: `checked` which will be passed down the inner component

    return (

      <div {...this.props} className={fClass} />

    );

```

}

});

## ***Consumption and transfer of the same prop***

It is possible that your component might need to consume and then transfer the same prop. In this case, you can explicitly pass it using `checked = {checked}`. This method is better compared to passing “*this.prop*” in full. The reason is because it will be easy to lint and refactor. Consider the program code given below:

```
var FCheckbox = React.createClass({  
  
  render: function() {  
  
    var { checked, title, ...other } = this.props;  
  
    var fClass = checked ? 'FancyChecked' : 'FancyUnchecked';  
  
    var fTitle = checked ? 'X ' + title : 'O ' + title;  
  
    return (  
  
      <label>  
  
        <input {...other}  
  
          checked={checked}  
  
          className={fClass}  
  
          type="checkbox"  
  
        >  
  
        {fTitle}  
  
      </label>  
  
    );
```

```
}
```

```
});
```

Just write and run the above program. You will observe the following output:



The output shows a checkbox which has been checked by default. You need to note that the order used matters a lot. In case “... *other*” comes before JSX props, then it will be hard for the consumer of the props to do overriding. In the above program, we are very sure that we will get an input of type checkbox once we run the program.

With the Rest properties, it is possible for you to do an extraction of the properties of an object which are remaining into another and new object. Every other property listed in the pattern is not included. Consider the following:

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

```
  a; // 1
```

```
  b; // 2
```

```
  c; // { x: 3, y: 4 }
```

The program shows how to effectively implement an ES7 proposal experimentally.

## *Transfer with an Underscore*

A library can be used instead of the JSX, and the same pattern can be achieved. With the underscore, properties can be filtered out using the “*\_.omit*” and properties can be copied into a new object using “*\_.extend*”. The following program code demonstrates how this can be done:

```
Var FCheckbox = React.createClass({  
  
render: function() {  
  
    varchckd = this.props.checked;  
  
    varothr = _.omit(this.props, 'chckd');  
  
    varfClass = checked ? 'FancyChecked' : 'FancyUnchecked';  
  
return (  
  
    React.DOM.div(_.extend({}, other, { className: fClass })))  
  
);  
  
}  
  
});
```

Just write the program above, and then run it. You will observe the following output:



From the figure above, it is very clear that we have achieved the same result, but using a

different method.





# Chapter 9

## React and Browsers

With React, too much abstraction is provided in such a way that you do not even have to access the DOM in a direct manner. In most cases, accessing the underlying API is enough for you to be able to work with existing code or a third-party library.

React Javascript is a very fast programming language. It keeps a representation of the DOM in memory, and the “*render()*” method is responsible for returning the description of this DOM if needed. To update the browser, React must compute the difference between the in-memory description and this description.

The Reacts’ “*faked browser*” is easier to use and performs well, so you should consider using it most of the times. However, sometimes, you might need to access the underlying API so as to work with a third-party library such as the JQuery plugin.

## ***findDOMNode() and Refs***

You have to get a reference to a DOM node so as to work with a browser in React. The function “*React.findDOMNode(component)*” in React serves this purpose. You have to call it so as to get a reference to the DOM node of your component. However, this function will only work on components which have been placed on DOM, that is, mounted components, otherwise, an exception will be thrown.

Consider the following program code:

```
var MComponent = React.createClass({  
  
  handleClick: function() {  
  
    // focusing on the text input explicitly using raw DOM API.  
  
      React.findDOMNode(this.refs.mTxtInput).focus();  
  
  },  
  
  render: function() {  
  
    // The attribute ref will add a reference to our component to  
  
    // this.refs once the component has been mounted.  
  
    return (  
  
      <div>  
  
      <input type=“text” ref=“mTxtInput” />  
  
      <input
```

```
type="button"

value="Focussing on the text input"

onClick={this.handleClick}

/>

</div>

);

}

});
```

```
React.render(

  <MComponent />,

  document.getElementById('sample')

);
```

Just write the above program, and then run it. Observe the output which you get. It should be as follows:



The image shows a simple web interface with two elements: a text input field on the left and a button on the right. The text input field is empty and has a vertical cursor on the left. The button is labeled "Focussing on the text input".

Notice that the program has displayed the components that we needed. The program has also shown that we can use either “*refs*” so as to refer to the component that you own or “*this*” so as to get the current component of React.



## ***More on Refs***

Once you have used the “*render()*” method so as to display the user interface of your application, you might need to invoke components which are on the instances of components which have been returned by the render method.

However, this idea is not for making data flow through your application since the Reactive data flow is responsible for this either directly or indirectly. However, it is good to know how to implement this, since you might desire to use it somewhere someday. Consider the program given below:

```
var Application = React.createClass({  
  
getInitialState: function() {  
  
    return {userInput: ”};  
  
},  
  
handleChange: function(ev) {  
  
    this.setState({userInput: ev.target.value});  
  
},  
  
clearAndFocusInput: function() {  
  
    this.setState({userInput: ”}); // Clearing the input  
  
// focusing on the <input /> now!  
  
},
```

```
render: function() {  
    return (  
  
    <div>  
  
    <div onClick={this.clearAndFocusInput}>  
  
    Click for focusing and Resetting  
  
    </div>  
  
    <input  
  
    value={this.state.userInput}  
  
    onChange={this.handleChange}  
  
    />  
  
    </div>  
  
    );  
    }  
    });
```

Just write the above program, and then run it. You will be presented with the following output:

Click for focusing and Resetting

In the program, we want to tell the input component to focus, once its value has been changed to an empty string. This component cannot infer this from its props over time. We

need to inform it that it should become focused. However, the “*render()*” method returns a description of the children at a particular time, rather than the actual composition of the components of the child. This means that you should not entirely rely on anything returned from the “*render()*” method, since it might not be meaningful at a particular time.

Consider the following counterexample being given:

**// This is a counterexample. Avoid this.**

```
render: function() {  
  
    varmInput = <input />;// let us try to call a number of methods on this  
  
    this.rememberThisInput = mInput; // this will serve as an input in  
    futureNOW!  
  
    return (  
  
    <div>  
  
    <div>...</div>  
  
    {mInput}  
  
    </div>  
  
    );  
  
    }
```

In the above example, we have created a real backing instance for the out `<input />` element.

## ***Ref Callback***

The “*ref*” attribute can be used as a callback function rather than as a name. Once the component has been mounted, then execution of this element will be done immediately. The component being referenced needs to be passed as a parameter, meaning that the parameter can be used immediately by the callback function. The reference can also be saved for future use.

Consider the following example:

```
var App = React.createClass({  
  
  getInitialState: function() {  
  
    return {userInput: ”};  
  
},  
  
  handleChange: function(ev) {  
  
    this.setState({userInput: ev.target.value});  
  
},  
  
  clearAndFocusInput: function() {  
  
    // Clearing the input  
  
    this.setState({userInput: ”}, function() {  
  
      // After re-rendering of the component, this code will be executed.  
  
      React.findDOMNode(this.refs.theInput).focus(); // Already focussed
```



```

});

},

render: function() {

    return (

<div>

<div onClick={this.clearAndFocusInput}>

Click for Focusing and Resetting

</div>

<input

ref="theInput"

value={this.state.userInput}

onChange={this.handleChange}

/>

</div>

);

}

});

```

In the above example, the returned result will be a description of an instance of `<input/>`.

The program should present to you the following interface:

Click for Focusing and Resetting

However, the true instance is accessible via “*this.refs.theInput.*” This will always return the correct instance, provided render returns the child component having ref=”input.” This is also applicable on higher-level components. Notice that if a child is destroyed, the same thing happens to its ref. You should avoid accessing refs inside the render method of any component.



# Conclusion

It can be concluded that React Javascript, which people commonly refer to as ReactJS or React.js is a Javascript library. It is open-source, meaning that you can download it for free. Programmers use it to develop user interfaces for web applications. The library was developed to help programmers solve problems that they experience when developing simple page applications.

It is also possible for developers to use this library for development of large web applications which are able to access and manage data which change most frequently. The language has the view of the web application, meaning that it can only be used for the development of the interface part of the application. It is possible to integrate it with other libraries belonging to Javascript so that the developed applications can be amazing to the users.

For you to use the library, you need to begin by downloading it. The downloaded file will always be in a zipped format, meaning that it must be extracted after the download. The extracted components should be kept in the same directory where your React program files will be kept. You may also choose to correctly specify the path of the location of the extracted components. As long as you have the library, you are set to start programming.

The writing of the code can be done in any of the editors, provided you are comfortable with it. There also are numerous editors available online for the purpose of writing React

programs. A good example of these is the “*jsfiddle*.” With these, you will have not to download the Javascript library, but just to write your React code and then run it. React supports the use of components which one can use to call classes.

One can also add data to React components. The language makes use of elements called “*props*,” which are responsible for the passage of data from the child to the parent. The user interface of an application developed in React can be broken into a hierarchy of components.