



کتاب آموزش

ReactJS

www.TahlilDadeh.com

مولف:

افشین رفوا

بسمه الله الرحمن الرحيم

آموزشگاه تحلیل داده

تخصصی ترین مرکز برنامه نویسی و دیتابیس در ایران

کتاب آموزش برنامه نویسی ReactJS

مؤلف مهندس افشین رفوا

آموزشگاه تحلیل داده

تقدیم به نائب امام عصر ، آیت الله خامنه ای که عصا زدنش ضرب آهنگ حیدری دارد.



تقدیم به همه جویندگان علم که توان امکان و شرکت در کلاس حضوری ندارند.

فهرست

۱۱	مقدمه
۱۱	توجه
۱۲	آموزش جامع React JS
۱۲	مخاطبان ReactJS
۱۲	پیش نیازها ReactJS
۱۳	مروری بر مطالب ReactJS
۱۳	امکانات ReactJS
۱۴	مزایای ReactJS
۱۴	محدودیت های ReactJS
۱۴	برپا کردن محیط ReactJS
۱۵	نصب ReactJS با استفاده از webpack و babel
۱۵	مرحله ۱: ایجاد پوشه ی اصلی برای ReactJS
۱۷	مرحله ۲: نصب ReactJS و react dom
۱۷	مرحله ۳: نصب وب پک
۱۸	مرحله ۴: نصب Babel

مرحله ۵: ایجاد فایل ها	۱۸
مرحله ۶: تنظیم کامپایلر، سرور و لودرها	۱۹
مرحله ۷: index.html	۲۰
مرحله ۸: App.jsx و main.js	۲۱
App.js	۲۱
main.js	۲۱
مرحله ۹: اجرای سرور	۲۲
مرحله ۱۰: تولید نرم افزار	۲۳
استفاده از دستور create-react-app برای ReactJS	۲۳
مرحله ۱: نصب create-react-app	۲۴
مرحله ۲: حذف تمامی فایل های منبع	۲۴
مرحله ۳: اضافه کردن فایل ها	۲۴
مرحله ۴: اجرای پروژه	۲۵
React JS در JSX	۲۵
استفاده از JSX در ReactJS	۲۵
App.jsx	۲۶
عناصر تو در تو	۲۷

۲۷	App.jsx
۲۸	صفات مربوط ReactJS
۲۸	عبارت های جاوا اسکریپت
۳۰	سیک بندی
۳۱	کامنت
۳۲	قرارداد نام گذاری در ReactJS
۳۳	اجزا (Component) در ReactJs
۳۳	مثال بدون State
۳۳	App.jsx
۳۵	main.js
۳۶	مثال با State
۳۶	App.jsx
۴۰	main.js
۴۰	حالت (State) های ReactJS
۴۰	استفاده از Props
۴۰	App.jsx
۴۲	مروری بر Props مربوط به ReactJS

۴۲	استفاده از props
۴۲	App.jsx
۴۳	main.js
۴۴	Props پیش فرض
۴۴	App.jsx
۴۶	ReactJS مربوط به Props و State
۴۶	App.jsx
۴۸	main.js
۴۹	ارزیابی ویژگی ها (Prop Validation) در ReactJS
۴۹	ارزیابی ویژگی ها
۴۹	App.jsx
۵۱	main.js
۵۲	API جزء (API Component) در ReactJS
۵۲	Set State
۵۴	Force Update
۵۵	Find Dom Node
۵۷	چرخه ی عمر اجزا (Component Life Cycle) در ReactJS

۵۸	متدهای چرخه ی عمر
۵۸	App.jsx
۶۱	main.js
۶۲	فرم ها (Forms) در ReactJS
۶۲	App.jsx
۶۳	main.js
۶۳	مثال پیچیده
۶۳	App.jsx
۶۳	main.js
۶۶	رویدادها (Events) مربوط به ReactJS
۶۶	مثال ساده
۶۶	App.jsx
۶۸	main.js
۶۸	رویدادهای فرزند
۶۸	App.jsx
۷۰	main.js
۷۱	Ref ها مربوط به ReactJS

استفاده از Ref ها	۷۱
App.jsx	۷۱
main.js	۷۳
کلیدها (Keys) در ReactJS	۷۳
استفاده از کلیدها	۷۳
App.jsx	۷۴
روتر (Router) در ReactJS	۷۷
مرحله ۱ - نصب یک ReactJS روتر	۷۷
مرحله ۲ - ایجاد اجزا	۷۷
main.js	۷۷
مرحله ۳ - اضافه کردن روتر	۸۰
main.js	۸۰
مفهوم Flex در ReactJS	۸۱
عناصر فلاکس	۸۱
مزایای فلاکس	۸۱
استفاده از Flux در ReactJS	۸۲
مرحله ۱ - نصب Redux	۸۲

مرحله ۲ - ایجاد فایل ها و فولدرها	۸۲
مرحله ۳ - Action ها	۸۲
مرحله ۴ - Reduser ها	۸۴
مرحله ۵ - Store	۸۶
مرحله ۶ - اجزاء دیگر	۸۹
App.jsx	۸۹
main.js	۸۹
انیمیشن ها (Animation) ها	۹۲
مرحله ۱ : نصب React CSS Transitions Group	۹۲
مرحله ۲ : اضافه کردن یک فایل CSS	۹۲
مرحله ۳ : ظاهر کردن انیمیشن	۹۳
مرحله ۴ : متحرک سازی ورود و خروج	۹۵
اجزای با مرتبه ی بالاتر (Higher Order Components) در ReactJS	۹۹
بهترین شیوه ها در ReactJS	۱۰۲

زکات علم نشر آن است. حضرت علی(ع)

موسسه آموزشی تحلیل داده ، با حضور جمعی از متخصصین مجرب در زمینه برنامه نویسی در نظر دارد،مطالب آموزشی خود را در قالب کتاب های آموزشی و فیلم ، به صورت رایگان در دسترس عموم قرار دهد تا حتی آن دسته از عزیزانی که بنا به دلایل مالی،مسافت جغرافیایی و یا نداشتن وقت کافی ، امکان شرکت در دوره های حضوری برای آنها میسر نیست،از یادگیری بی بهره نمانند.

علاوه بر این علاقه مندان می توانند ، با ثبت نام در انجمن سایت تحلیل داده،سوالات خود را مطرح نموده و مدرسین آموزشگاه و اعضای انجمن در اسرع وقت،پاسخ های خود را، حتی الامکان به صورت فیلم، در دسترس عموم قرار دهند.

لذا از کلیه فعالان در این زمینه دعوت می شود، در این حرکت جمعی در کنار ما باشند و با حضور فعال خود در انجمن،گام موثری در بهبود سطح علمی جوانان کشور عزیزمان،ایران بردارند.

توجه :

برای دانلود سورس کد مثال های کتاب ، در بخش مقالات سایت به آموزش ReactJS در آدرس

www.tahlildadeh.com مراجعه فرمایید.

آموزش جامع React JS

React یک کتابخانه ی فرانت – اند بوده که توسط فیسبوک توسعه یافته است. با کمک آن می توان لایه ی view برنامه های موبایل و وب را مدیریت کرد. با کمک ReactJS می توانیم اجزای رابط کاربری با قابلیت استفاده ی مجدد را ایجاد کنیم. ReactJS در حال حاضر یکی از محبوب ترین کتابخانه های جاوا اسکریپت بوده و بنیان و جامعه ی عظیمی را به همراه دارد.

مخاطبان ReactJS

برنامه نویسان جاوا اسکریپتی که برای اولین بار با ReactJS سروکار دارند، می توانند از این آموزش استفاده کنند. سعی خواهیم کرد که با نمایش نمونه کدهای ساده و قابل فهم به تمامی مفاهیم پردازیم. بعد از تمام کردن این بخش ها شما بر ReactJS تسلط پیدا خواهید کرد. اضافه بر آموزش خود به معرفی عناصر بیشتری خواهیم پرداخت که در کنار ReactJS به خوبی جواب می دهند تا بتوانید با کمک آن ها بهترین شیوه ها را فراگیرید و روندهای امروزی جاوا اسکریپت را دنبال کنید.

پیش نیازها ReactJS

اگر می خواهید با ReactJS کار کنید، نیاز است که دانش خوبی از جاوا اسکریپت، HTML5 و CSS داشته باشید. با وجود این که ReactJS از HTML استفاده نمی کند، JSX شبیه به HTML است، بنابراین دانش HTML شما به کمکتان خواهد آمد. در یکی از بخش ها به این موضوع بیشتر خواهیم پرداخت. همچنین از سینتکس EcmaScript 2015 استفاده خواهیم کرد، بنابراین هر دانشی که در این زمینه دارید، برایتان مفید خواهد بود.

ReactJS یک کتابخانه ی جاوا اسکریپت است که می توان از آن جهت ساختن اجزای رابط کاربری با قابلیت استفاده ی مجدد استفاده کرد. در آموزش رسمی React تعریف زیر برای آن بیان شده است:

React کتابخانه ای است که با کمک آن می توان رابط های کاربری قابل خواندن را ایجاد کرد. همچنین این کتابخانه، ایجاد اجزای رابط کاربری قابل استفاده ی مجدد را تسهیل می کند؛ اجزایی که داده هایی را نشان می دهند که در گذر زمان تغییر می کنند. بسیاری از مردم مانند V در MVC از React استفاده می کنند. انتزاع های React، DOM را از شما دور می کنند و مدل برنامه نویسی ساده تر و عملکرد بهتری را برای شما فراهم می کنند. همچنین React می تواند در سرور با استفاده از Node رندر شود و می تواند با استفاده از React Native برنامه های بومی را تقویت کند. React جریان داده ی واکنشی یک طرفه را اجرا می کند. این کار باعث می شود نیاز به استفاده ی کدهای تکراری کاهش یابد و نسبت به مقیدسازی سنتی داده درک کد آسان تر شود.

امکانات ReactJS

- JSX: افزونه ی سینتکس جاوا اسکریپت است. در برنامه نویسی ReactJS لزومی به استفاده از JSX وجود ندارد، اما توصیه می شود این کار انجام شود.
- اجزا: ReactJS پیرامون اجزا می چرخد. همه چیز را باید به عنوان یک جزء در نظر بگیرید، در این صورت می توانید زمانی که بر روی پروژه های بزرگ تر کار می کنید راحت تر از کد خود نگهداری کنید.
- جریان داده ی یک طرفه و ReactJS : Flux از جریان داده ی یک طرفه بهره می برد و به همین دلیل فهم برنامه را آسان تر می کند. فلاکس الگویی است که با کمک آن می توانید داده های خود را یک طرفه نگه دارید.

- لایسنس: ReactJS توسط شرکت فیسبوک لایسنس شده و آموزش آن تحت CC BY 4.0 لایسنس شده است.

مزایای ReactJS

- استفاده از DOM مجازی که در واقع یک شیء جاوا اسکریپت است. از این طریق می توانید عملکرد برنامه های خود را بهبود بخشید. زیرا DOM مجازی جاوا اسکریپت سریع تر از DOM معمولی است.
- از ReactJS می توانید در هر دو سمت کلاینت و سرور، همچنین در فریمورک های دیگر استفاده کنید.
- الگوهای داده و اجزا خوانایی برنامه ی شما را افزایش می دهند و از این طریق می توانید راحت تر می توانید از برنامه های بزرگ تر خود نگهداری کنید.

محدودیت های ReactJS

- در ReactJS تنها لایه ی view برنامه پوشش داده شده است. به همین دلیل اگر می خواهید امکانات کامل برنامه نویسی را دریافت کنید، باید از فناوری های دیگر استفاده کنید.
- در ReactJS از قالب نویسی و JSX درون خطی استفاده می شود که این امر برای برخی از برنامه نویسان ناخوشایند به نظر می رسد.

برپا کردن محیط ReactJS

در این بخش می خواهیم به چگونگی برپا کردن محیط جهت برنامه نویسی موفق در ReactJS پردازیم. توجه داشته باشید که برای انجام این کار مراحل زیادی نیاز است، اما این کارها در آینده فرآیند برنامه نویسی را سرعت می بخشند. برای انجام این کار به NodeJS نیاز خواهیم داشت. بنابراین اگر آن را نصب نکرده اید، به لینک جدول زیر مراجعه کنید.

ردیف	نرم افزار و توضیحات
۱	NodeJS و NPM NodeJS پلتفرمی است که برای توسعه ی ReactJS به آن نیاز است. به برپا کردن محیط NodeJS مراجعه کنید.

پس از نصب موفق NodeJS می توانیم با استفاده از npm React را نصب کنیم. به دو صورت زیر می توانید NodeJS را نصب کنید:

- استفاده از webpack و babel.
- استفاده از دستور create-react-app

نصب ReactJS با استفاده از webpack و babel

وب پک یک مازول مجموعه ای است (ماژول های مستقل را مدیریت کرده و آن ها را بارگیری می کند). وب پک مازول های وابسته را گرفته و آن ها را به یک فایل واحد کامپایل می کند. طی برنامه نویسی با استفاده از خط فرمان یا با پیکربندی آن با استفاده از فایل webpack.config می توانید از وب پک استفاده کنید.

بابل یک کامپایلر و ترنسپایلر جاوا اسکریپت است. Babel در تبدیل یک سورس کد به سورس کد دیگر کاربرد دارد. با استفاده از این قابلیت می توانید در کد خود از امکانات جدید ES6 استفاده کنید، به گونه ای که بابل این امکانات را به ES5 قدیمی و ساده تبدیل می کند تا بتوان از آن ها در تمامی مرورگرها استفاده کرد.

مرحله ۱: ایجاد پوشه ی اصلی برای ReactJS

در دسکتاپ پوشه ای با نام reactApp ایجاد کنید تا بتوانید با استفاده از دستور mkdir فایل های مورد نیاز را نصب کنید.

```
C:\Users\username\Desktop>mkdir reactApp  
C:\Users\username\Desktop>cd reactApp
```

برای این که بتوانید هر مازولی را ایجاد کنید، نیاز است که فایل package.json را تولید کنید. بنابراین پس از ایجاد این پوشه ما باید یک فایل package.json را ایجاد کنیم. برای انجام این کار از دستور npm init را وارد کنید.

```
C:\Users\username\Desktop\reactApp>npm init
```

این دستور اطلاعاتی در رابطه با مازول را درخواست می کند؛ اطلاعاتی مانند اسم بسته، توضیحات، ناشر و با استفاده از گزینه ی -y می توانید از این مرحله عبور کنید.

```
C:\Users\username\Desktop\reactApp>npm init -y  
Wrote to C:\reactApp\package.json:  
{  
  "name": "reactApp",  
  
  "version": "1.0.0",  
  
  "description": "",  
  
  "main": "index.js",  
  
  "scripts": {
```

```
"test": "echo \"Error: no test specified\" && exit 1"
```

```
},
```

```
"keywords": [],
```

```
"author": "",
```

```
"license": "ISC"
```

مرحله ۲: نصب ReactJS و react dom

با توجه به این که کار اصلی ما نصب ReactJS است، جهت نصب آن و بسته های DOM آن به ترتیب از دستورات install react و react-dom متعلق به npm استفاده کنید. می توانید با گزینه ی --save بسته هایی که ما نصب کرده ایم را به فایل package.json اضافه کنید.

```
C:\Users\Tutorialspoint\Desktop\reactApp>npm install react --save  
C:\Users\Tutorialspoint\Desktop\reactApp>npm install react-dom --save
```

یا می توانید تمامی آن ها را مانند زیر در یک دستور واحد نصب کنید.

```
C:\Users\username\Desktop\reactApp>npm install react react-dom --save
```

مرحله سوم : نصب وب پک

با توجه به این که ما جهت تولید این مجموعه در حال استفاده از وب پک هستیم، webpack-dev-server و webpack-cli را نصب کنید.

```
C:\Users\username\Desktop\reactApp>npm install webpack --save
C:\Users\username\Desktop\reactApp>npm install webpack-dev-server --save
C:\Users\username\Desktop\reactApp>npm install webpack-cli --save
```

یا می توانید تمامی آن ها را مانند زیر در یک دستور واحد نصب کنید.

```
C:\Users\username\Desktop\reactApp>npm install webpack webpack-dev-server
webpack-cli --save
```

مرحله چهارم: نصب Babel

babel و پلاگین های babel-core, babel-loader, babel-preset-env, babel-preset-react و html-webpack-plugin را نصب کنید.

```
C:\Users\username\Desktop\reactApp>npm install babel-core --save-dev
C:\Users\username\Desktop\reactApp>npm install babel-loader --save-dev
C:\Users\username\Desktop\reactApp>npm install babel-preset-env --save-dev
C:\Users\username\Desktop\reactApp>npm install babel-preset-react --save-dev
C:\Users\username\Desktop\reactApp>npm install html-webpack-plugin --save-dev
```

یا می توانید مانند زیر تمامی آن ها را در یک دستور واحد نصب کنید.

```
C:\Users\username\Desktop\reactApp>npm install babel-core babel-loader babel-
preset-env
babel-preset-react html-webpack-plugin --save-dev
```

مرحله پنجم: ایجاد فایل ها

جهت تکمیل نصب باید برخی از فایل های خاص را مانند index.html, App.js, main.js

Webpack.config.js و babelrc ایجاد کنیم. می توانید این فایل ها را یا به صورت دستی و یا با استفاده از cmd ایجاد کنید.

```
C:\Users\username\Desktop\reactApp>type nul > index.html
C:\Users\username\Desktop\reactApp>type nul > App.js
C:\Users\username\Desktop\reactApp>type nul > main.js
C:\Users\username\Desktop\reactApp>type nul > webpack.config.js
C:\Users\username\Desktop\reactApp>type nul > .babelrc
```

مرحله ششم: تنظیم کامپایلر، سرور و لودرها

فایل webpack-config.js را باز کنید و کد زیر را به آن اضافه کنید. می خواهیم نقطه ی ورودی وب پک را به گونه ای تنظیم کنیم که این نقطه main.js شود. مسیر خروجی جایی است که سرور به برنامه خدمات می رساند.

همچنین سرور برنامه نویسی را بر روی پورت ۸۰۰۱ تنظیم می کنیم. این پورت را می توانید به صورت دلخواه نیز انتخاب کنید.

webpack.config.js

```
const path = require('path');

const HtmlWebpackPlugin = require('html-webpack-plugin');
module.exports = {

  entry: './main.js',
  output: {
    'path': path.join(__dirname, '/bundle',
  filename: 'index_bundle.js'
  },
  devServer: {
    inline: true,
    port: 8080 },
  module: {
```

```

rules: [
  {
    test: /\.jsx?$/,
    exclude: /node_modules/,
    loader: 'babel-loader',
    query: {
      presets: ['es2015', 'react']
    }
  }
],
plugins: [
  new HtmlWebpackPlugin({
    template: './index.html'
  })
]
}

```

Package.json را باز کنید و "test" "echo \"Error: no test specified\" && exit 1" را از داخل شیء "scripts" پاک کنید. دلیل پاک کردن این خط این است که ما در این آموزش نیازی به آزمایش نخواهیم داشت. بیاید در عوض دستورات start و build را اضافه کنیم.

```

"start": "webpack-dev-server --mode development --open --hot",
"build": "webpack --mode production"

```

مرحله هفتم: index.html

این مرحله صرفاً یک HTML معمولی است. "div id = "app" را به عنوان عنصر اصلی برنامه تنظیم می کنیم و

اسکرپت index_bundle.js که در واقع فایل برنامه ی ما است را اضافه می کنیم.

```

<!DOCTYPE html>
<html lang = "en">
<head>

```



```

<meta charset = "UTF-8">
<title>React App</title>
</head>
<body>
<div id = "app">
</div>
<script src = 'index_bundle.js'>
</script>
</body>
</html>

```

مرحله هشتم: App.jsx و main.js

این مرحله اولین جزء ری اکت را دربر می گیرد. در یکی از بخش های بعدی به صورت مفصل به اجزای ری اکت خواهیم پرداخت. این جزء باعث می شود Hello World رندر شود.

App.js

```

import React, { Component } from 'react';
class App extends Component{
  render(){
    return(
      <div>
        <h1>Hello World</h1>
      </div>
    );
  }
}
export default App;

```

باید این جزء را ایمپورت کنیم و آن را در عنصر اصلی برنامه ی خود رندر کنیم تا بتوانیم آن را در مرورگر ببینیم.

main.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.js';
ReactDOM.render(<App />, document.getElementById('app'));
```

نکته: هر زمان که بخواهید از چیزی استفاده کنید، باید آن را در مرحله ی اول ایمپورت کنید. اگر می خواهید این جزء در بخش های دیگر برنامه قابل استفاده باشد، باید آن را پس از ایجاد export کنید و آن را در فایلی که می خواهید از آن استفاده کنید، import کنید. فایلی را با نام babelrc ایجاد کنید و محتوای زیر را در آن کپی کنید.

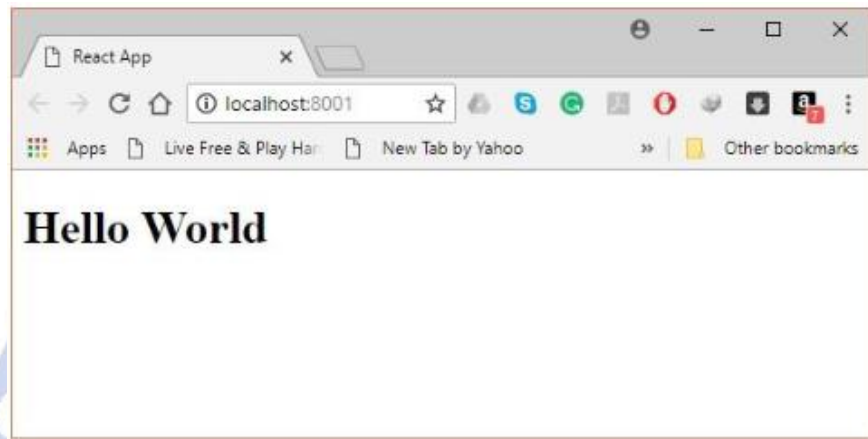
```
{
  "presets":["env", "react"]
}
```

مرحله نهم : اجرای سرور

عملیات برپا کردن محیط تمام شده است و می توانیم با اجرای دستور زیر سرور را راه اندازی کنیم.

```
C:\Users\username\Desktop\reactApp>npm start
```

با اجرای این کار پورتنی که ما می خواهیم در مرورگر باز شود، نمایش داده می شود. این پورت برای ما <http://localhost:8001/> است. بعد از این که این پورت را باز کنیم، نتیجه ی زیر نمایش داده می شود.

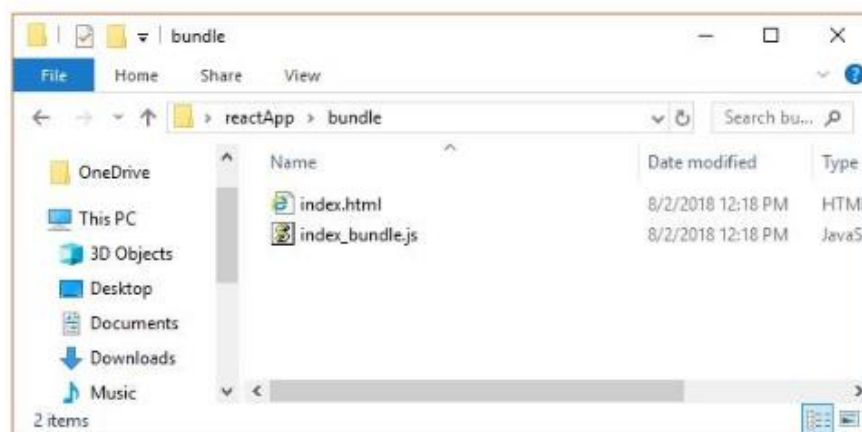


مرحله دهم : تولید نرم افزار

در نهایت برای آن که بتوانید نرم افزار را تولید کنید، باید دستور build را در cmd مانند زیر اجرا کنید.

C:\Users\Tutorialspoint\Desktop\reactApp>npm run build

این کار باعث می شود این نرم افزار مانند زیر در پوشه ی فعلی تولید شود.



استفاده از دستور create-react-app برای ReactJS

به جای استفاده از وب پک و بابل می توانید به صورت ساده تری ReactJS را با نصب create-react-app نصب کنید.

مرحله ۱ : نصب create-react-app

به دسکتاپ بروید و با استفاده از cmd مانند زیر Create React App نصب کنید.

```
C:\Users\Tutorialspoint>cd C:\Users\Tutorialspoint\Desktop\  
C:\Users\Tutorialspoint\Desktop>npx create-react-app my-app
```

این کار باعث می شود، در دسکتاپ پوشه ای به نام my-app ایجاد شود و تمامی فایل های مورد نیاز در آن نصب شود.

مرحله ۲ : حذف تمامی فایل های منبع

به پوشه ی SRC موجود در پوشه ی my-app تولید شده بروید و مانند زیر تمامی فایل ها را حذف کنید.

```
C:\Users\Tutorialspoint\Desktop>cd my-app/src  
C:\Users\Tutorialspoint\Desktop\my-app\src>del *  
C:\Users\Tutorialspoint\Desktop\my-app\src\*, Are you sure (Y/N)? y
```

مرحله ۳ : اضافه کردن فایل ها

با استفاده از نام های index.css و index.js موجود در پوشه ی SRC فایل ها را اضافه کنید.

```
C:\Users\Tutorialspoint\Desktop\my-app\src>type nul > index.css  
C:\Users\Tutorialspoint\Desktop\my-app\src>type nul > index.js
```

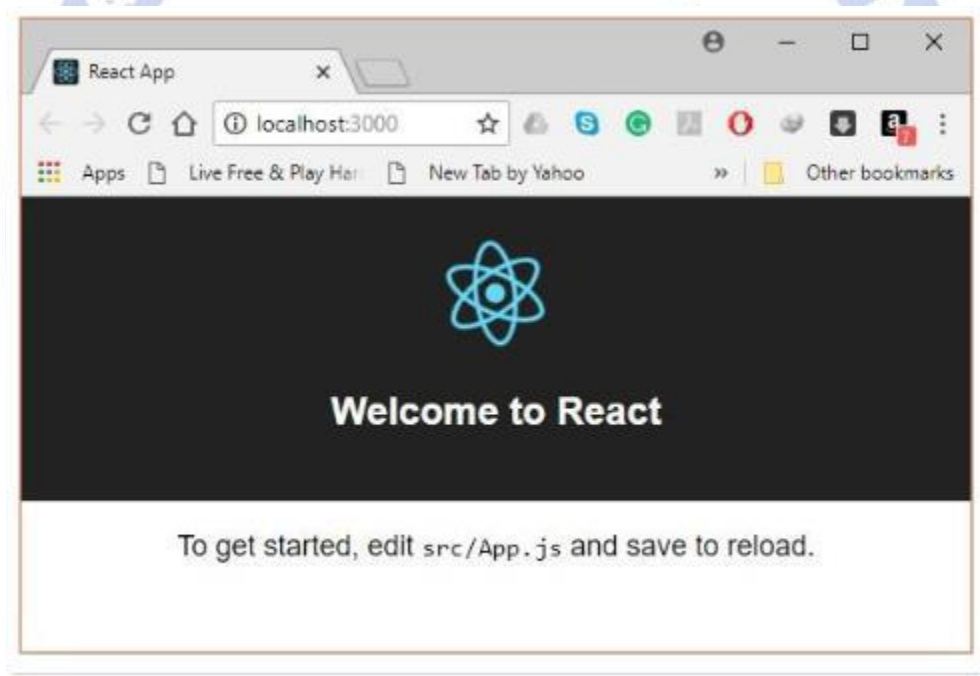
کد زیر را به فایل index.js اضافه کنید.

```
import React from 'react';  
import ReactDOM from 'react-dom';  
import './index.css';
```

مرحله ۴: اجرای پروژه

در نهایت با استفاده از دستور start پروژه را اجرا کنید.

```
npm start
```



React JS در JSX

ReactJS به جای جاوا اسکریپت معمولی جهت قالب بندی از JSX استفاده می کند. اجباری در استفاده از آن وجود ندارد، اما مزایای زیر را به همراه دارد:

- JSX سریع تر است؛ زیرا طی کامپایل کد به جاوا اسکریپت کار بهینه سازی را انجام می دهد.
- اصطلاحا type-safe است و اغلب خطاها را می توان طی کامپایل شناسایی کرد.
- نوشتن قالب ها به شرط آن که با HTML آشنا باشید را آسان تر و سریع تر می کند.

استفاده از JSX در ReactJS

JSX در بسیاری از موارد شبیه به HTML معمولی است و قبلا در بخش برپا کردن محیط از آن استفاده

کرده ایم. به کد App.jsx نگاه کنید، در این کد div برگشت داده می شود

App.jsx

```
import React from 'react';
```

```
class App extends React.Component {
```

```
  render() {
```

```
    return (
```

```
      <div>
```

```
        Hello World!!!
```

```
      </div>
```

```
    );
```

```
  }
```

```
  export default App;
```

با وجود اینکه JSX شبیه به HTML است ، اما این دو تفاوت هایی دارند که در زمان کار با JSX باید آنها را در نظر داشت.

عناصر تو در تو

اگر می خواهید عناصر بیشتری را برگشت دهید، نیاز است که آن را با یک عنصر نگه دارنده بپوشانید. توجه کنید که ما چگونه از div به عنوان یک پوشاننده برای h1، h2 و p استفاده کرده ایم.

App.jsx

```
import React from 'react';

class App extends React.Component {
  render() {
    return (
      <div>
        <h1>Header</h1>
        <h2>Content</h2>
        <p>This is the content!!!</p>
      </div>
    );
  }
}

export default App;
```



صفات مربوط ReactJS

علاوه بر ویژگی ها و صفت های HTML معمولی می توانیم از صفات اختصاصی مخصوص به خودمان نیز استفاده کنیم. زمانی که می خواهیم صفات اختصاصی را اضافه کنیم باید از پیشوند data- استفاده کنیم. در مثال زیر ما data-myattribute را به عنوان صفتی برای عنصر p اضافه کرده ایم.

```
import React from 'react';
class App extends React.Component {
  render() {
    return (
      <div>
        <h1>Header</h1>
        <h2>Content</h2>
        <p data-myattribute = "somevalue">This is the content!!!</p>
      </div>
    );
  }
}
export default App;
```

عبارت های جاوا اسکریپت

از این عبارت ها می توان داخل JSX استفاده کرد. تنها کافی است آن را داخل {} قرار دهیم. مثال زیر ۲ را نمایش می دهد.

```
import React from 'react';
```

```

class App extends React.Component {
  render() {
    return (
      <div>
        <h1>{1+1}</h1>
      </div>
    );
  }
}

export default App;

```



داخل JSX نمی توانیم از دستورات if else استفاده کنیم، در عوض می توانیم از عبارت های شرطی (بر مبنای ۳) استفاده کنیم. در مثال زیر متغیر i برابر با ۱ است. بنابراین مرورگر true را نمایش می دهد. اگر مقدار این متغیر را تغییر دهیم، مرورگر false را نمایش می دهد.

```

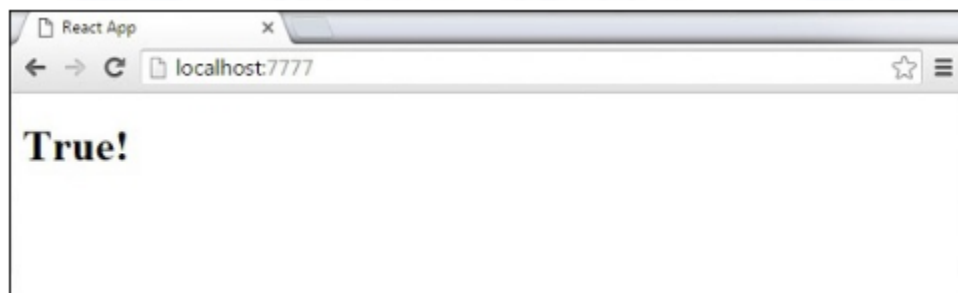
import React from 'react';

class App extends React.Component {

```

```
render() {
  var i = 1;
  return (
    <div>
      <h1>{i == 1 ? 'True!' : 'False'}</h1>
    </div>
  );
}
}
```

export default App;



سبک بندی

ReactJS توصیه می کند از سبک های درون خطی استفاده شود. زمانی که می خواهیم این نوع از سبک ها را تنظیم کنیم، نیاز است از سینتکس camelCase استفاده کنیم. ReactJS نیز به صورت خودکار px را پس از مقدار عددی عناصر مشخص می چسباند. در مثال زیر چگونگی اضافه کردن myStyle درون خطی به عنصر h1 نشان داده شده است.

```
import React from 'react';
```

```
class App extends React.Component {
```

```
  render() {
```

```
    var myStyle = {
```

```
      fontSize: 100,
```

```
      color: '#FF0000'
```

```
    }
```

```
    return (
```

```
      <div>
```

```
        <h1 style={myStyle}>Header</h1>
```

```
      </div>
```

```
    );
```

```
  }
```

```
}
```

```
export default App;
```



کامنت

اگر بخواهیم کامنت نویسی کنیم و بخواهیم این کامنت ها را داخل بخش فرزند یک برچسب بنویسیم، باید از {} استفاده کنیم. بهتر است همیشه در زمان نوشتن کامنت ها از {} استفاده شود، چرا که حفظ ثبات طی برنامه نویسی مهم است.

```
import React from 'react';

class App extends React.Component {
  render() {
    return (
      <div>
        <h1>Header</h1>
        {/**End of the line Comment...*/}
        {/**Multi line comment...*/}
      </div>
    );
  }
}

export default App;
```

قرارداد نام گذاری در ReactJS

همیشه در برچسب های HTML از اسم های برچسب با حروف کوچک استفاده می شود. این در حالی است که اجزای ReactJS با حرف بزرگ آغاز می شوند.

نکته : بهتر است که به جای Class و for به عنوان اسامی صفت XML از ClassName و HtmlFor استفاده شود.

این مطلب در صفحه ی رسمی ReactJS به صورت زیر توضیح داده شده است:

با توجه به این که JSX جاوا اسکریپت است، بهتر است از شناساگرهایی مانند class و for به عنوان اسامی صفات XML استفاده نشود. در عوض بهتر است در اجزای DOM ReactJS به ترتیب از ویژگی هایی مانند className و htmlFor استفاده شود.

اجزا (Component) در RejectJs

در این بخش می خواهیم به چگونگی ترکیب اجزاپردازیم، به گونه ای که نگهداری از برنامه آسان تر شود. با استفاده از این روش می توانیم اجزای خود را بدون تحت تأثیر قرار دادن بخش های باقی مانده ی صفحه به روز رسانی کرده و تغییر دهیم.

مثال بدون State

اولین جزء ما در مثال زیر App است. این جزء مالک Header و Content است. می خواهیم به صورت مجزا Header و Content را ایجاد کنیم و صرفاً آن ها را داخل درخت JSX جزء App خود اضافه کنیم. تنها جزء App باید اکسپورت شود.

App.jsx

```
import React from 'react';

class App extends React.Component {

  render() {

    return (
```

```

<div>

  <Header/>

  <Content/>

</div>

);

}

}

```

```

class Header extends React.Component {
  render() {
    return (
      <div>

        <h1>Header</h1>
      </div>
    );
  }
}

```

```

class Content extends React.Component {

  render() {

    return (

      <div>

        <h2>Content</h2>

        <p>The content text!!!</p>

```

```
</div>
```

```
);
```

```
}
```

```
}
```

```
export default App;
```

برای آن که بتوانیم نتیجه را در صفحه نمایش دهیم، باید این کد را داخل فایل main.js ایمپورت کنیم و ReactDOM.render() را فراخوانی کنیم. قبلا این کار را در بخش برپا کردن محیط انجام داده ایم.

main.js

```
import React from 'react';
```

```
import ReactDOM from 'react-dom';
```

```
import App from './App.jsx';
```

```
ReactDOM.render(<App />, document.getElementById('app'));
```

کد بالا نتیجه ی زیر را ایجاد می کند.



مثال با State

در این مثال می خواهیم state یا حالت جزء مالک (App) را تنظیم کنیم. با توجه به اینکه جزء Header هیچ حالتی ندارد، آن را مانند مثال قبل اضافه می کنیم. به جای برچسب content عناصر table و tbody را ایجاد می کنیم و از طریق آن ها به ازای هر شیء از آرایه ی data، TableRow را به صورت پویا درج می کنیم.

همان طور که مشخص است، ما از سینتکس EcmaScript 2015 arrow (=) استفاده می کنیم. چرا که این سینتکس تمیزتر از سینتکس قدیمی جاوا اسکریپت به نظر می رسد. از این طریق می توانیم عناصر خود را با کدهای کمتری ایجاد کنیم. این سینتکس به ویژه در مواقعی کاربرد دارد که بخواهیم لیستی را با آیتم های بسیاری ایجاد کنیم.

App.jsx

```
import React from 'react';

class App extends React.Component {
  constructor() {
    super();

    this.state = { data:
      [
        {
          "id":1,
          "name":"Foo",
          "age":"20"
```

```

    },
    {
      "id":2,
      "name":"Bar",
      "age":"30"
    },
    {
      "id":3,
      "name":"Baz",
      "age":"40"
    }
  ]
}

```

```

render() {
  return (
    <div>
      <Header/>
      <table>

```



```

<tbody>

  {this.state.data.map((person, i) => <TableRow key = {i}
data = {person} />)}

</tbody>

</table>

</div>

);

}

}

class Header extends React.Component {

  render() {

    return (

      <div>

        <h1>Header</h1>

      </div>

    );

  }

}

class TableRow extends React.Component {

```



```

render() {
  return (
    <tr>
      <td>{this.props.data.id}</td>
      <td>{this.props.data.name}</td>
      <td>{this.props.data.age}</td>
    </tr>
  );
}
}

export default App;

```

main.js

```

import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.jsx';
ReactDOM.render(<App/>, document.getElementById('app'));

```

نکته : توجه داشته باشید که ما از تابع `map` `inside {i} = key` () استفاده می کنیم. از این طریق ReactJS راحت تر می تواند صرفا عناصر ضروری را به روز رسانی کند. به جای آن که زمانی که چیزی

تغییر می کند، کل لیست را مجددا رندر کند. در صورتی که با تعداد زیادی از عناصری سروکار داشته باشیم که به صورت پویا ایجاد شده اند، این کار باعث می شود عملکرد تا حد چشمگیری افزایش یابد.



حالت (State) های ReactJS

State یا حالت مکانی است که داده ها از آن سرچشمه می گیرند. همواره بهتر است تا حد امکان حالت خود را ساده در نظر بگیریم و تعداد اجزای حالت دار را به حداقل برسانیم. مثلا اگر ده جزء داشته باشیم که داده هایی را از حالتی نیاز دارند، بهتر است یک جزء نگهدارنده را ایجاد کنیم که این حالت را برای تمامی این اجزا نگهداری کند.

استفاده از Props

در نمونه کد زیر چگونگی ایجاد یک جزء حالت دار با استفاده از سینتکس EcmaScript2016 نشان داده شده است.

App.jsx

```
import React from 'react';  
  
class App extends React.Component
```



```

{ constructor(props) {
  super(props);

  this.state = {
    header: "Header from state...",
    content: "Content from state..."
  }
}

render() {
  return (
    <div>
      <h1>{this.state.header}</h1>
      <h2>{this.state.content}</h2>
    </div>
  );
}
}

export default App;

```

main.js

```

import React from 'react';

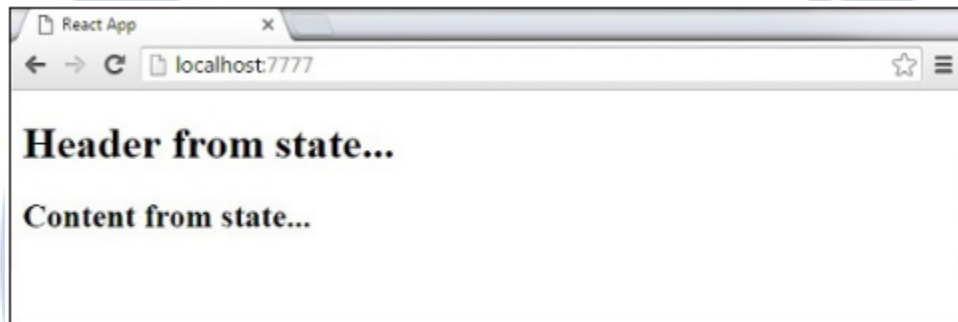
import ReactDOM from 'react-dom';

import App from './App.jsx';

```

```
ReactDOM.render(<App />, document.getElementById('app'));
```

این کار باعث می شود نتیجه ی زیر نمایش داده شود



مروری بر Props مربوط به ReactJS

تفاوت اصلی بین state و props این است که prop ها تغییر ناپذیر هستند. دلیل این امر این است که جزء نگهدارنده باید حالتی را تعریف کند که بتوان آن را به روز رسانی کرده و تغییر داد. این در حالی است که اجزای فرزند تنها باید داده ها را از حالت با استفاده از props عبور دهند.

استفاده از props

اگر در جزء خود به داده های تغییرناپذیری نیاز داریم، می توانیم صرفا props را به تابع ReactDOM.render() موجود در main.js اضافه کنیم و از آن داخل جزء خود استفاده کنیم.

App.jsx

```
import React from 'react';  
  
class App extends React.Component {
```

```

render() {

  return (

    <div>

      <h1>{this.props.headerProp}</h1>

      <h2>{this.props.contentProp}</h2>

    </div>
  );

}

}

export default App;

```

main.js

```

import React from 'react';

import ReactDOM from 'react-dom';

import App from './App.jsx';
ReactDOM.render(<App headerProp = "Header from props..."
contentProp = "Content from props..." />,
document.getElementById('app'));

export default App;

```

این کار باعث می شود نتیجه ی زیر نمایش داده شود.



Props پیش فرض

می توانید مقادیر مشخصه ی پیش فرض را مستقیما در سازنده ی جزء تنظیم کنید، به جای آن که آن را به عنصر `ReactDOM.render()` اضافه کنید.

App.jsx

```
import React from 'react';

class App extends React.Component {
  render() {

    return (

      <div>

        <h1>{this.props.headerProp}</h1>

        <h2>{this.props.contentProp}</h2>

      </div>

    );

  }

}

App.defaultProps = {
```



```
headerProp: "Header from props...",  
contentProp: "Content from props..."  
}  
  
export default App;
```

main.js

```
import React from 'react';  
import ReactDOM from 'react-dom';  
import App from './App.jsx';  
  
ReactDOM.render(<App/>, document.getElementById('app'));
```

خروجی مانند قبل است.



ReactJS State و Props مربوط به

در مثال زیر چگونگی ترکیب state و props در برنامه نشان داده شده است. ما state را در جزء مادر تنظیم کرده ایم و با استفاده از props آن را به درخت جزء عبور داده ایم. headerProp و contentProp استفاده شده در اجزای فرزند را داخل تابع render تنظیم کرده ایم.

App.jsx

```
import React from 'react';

class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      header: "Header from props...",
      content: "Content from props..."
    }
  }

  render() {
    return (
      <div>
        <Header headerProp = {this.state.header}/>
        <Content contentProp = {this.state.content}/>
      </div>
    );
  }
}
```

```

);
}
}

class Header extends React.Component {
  render() {
    return (
      <div>
        <h1>{this.props.headerProp}</h1>
      </div>
    );
  }
}

```

```

class Content extends React.Component {
  render() {
    return
    ( <div>
      <h2>{this.props.contentProp}</h2>
    </div>
    );
  }
}

```

```
}
```

```
}
```

```
export default App;
```

main.js

```
import React from 'react';
```

```
import ReactDOM from 'react-dom';
```

```
import App from './App.jsx';
```

```
ReactDOM.render(<App/>, document.getElementById('app'));
```

نتیجه باز هم تغییری نخواهد کرد و تنها چیزی که تغییر می کند، منبع داده های ما است که در حال حاضر از state سرچشمه می گیرند. اگر بخواهیم این برنامه را به روز رسانی کنیم، باید state را به روز رسانی کنیم و به دنبال آن تمامی اجزای فرزند به روز رسانی خواهند شد. در بخش رویدادها بیشتر به این مطلب پرداخته شده است.



ارزیابی ویژگی ها (Prop Validation) در ReactJS

ارزیابی ویژگی ها روشی مفید جهت استفاده ی صحیح از اجزا به شمار می رود. از این طریق می توانیم طی برنامه نویسی از باگ ها و مشکلات آینده جلوگیری کنیم، مخصوصا زمانی که برنامه به اندازه ی کافی بزرگ می شود. همچنین این قابلیت خوانایی کد را افزایش می دهد، زیرا از این طریق می توانیم ببینیم که چگونه از هر یک از اجزا باید استفاده شود.

ارزیابی ویژگی ها

در این مثال می خواهیم جزء App را به همراه تمام prop هایی که نیاز داریم، ایجاد کنیم. App.propTypes جهت ارزیابی ویژگی ها کاربرد دارد. اگر برخی از ویژگی ها از نوع صحیحی که ما تعیین کرده ایم استفاده نکنند، با هشدار کنسول مواجه می شویم. بعد از مشخص کردن الگوهای ارزیابی App.defaultProps را تنظیم می کنیم.

App.jsx

```
import React from 'react';

class App extends React.Component {

  render() {

  return (

    <div>

      <h3>Array: {this.props.propArray}</h3>

      <h3>Bool: {this.props.propBool ? "True..." : "False..."}</h3>

      <h3>Func: {this.props.propFunc(3)}</h3>

      <h3>Number: {this.props.propNumber}</h3>
```

```

<h3>String: {this.props.propString}</h3>
<h3>Object: {this.props.propObject.objectName1}</h3>
<h3>Object: {this.props.propObject.objectName2}</h3>
<h3>Object: {this.props.propObject.objectName3}</h3>
</div>
);
}
}

App.propTypes = {
  propArray: React.PropTypes.array.isRequired,
  propBool: React.PropTypes.bool.isRequired,
  propFunc: React.PropTypes.func,
  propNumber: React.PropTypes.number,
  propString: React.PropTypes.string,
  propObject: React.PropTypes.object
}

App.defaultProps = {
  propArray: [1,2,3,4,5],
  propBool: true,

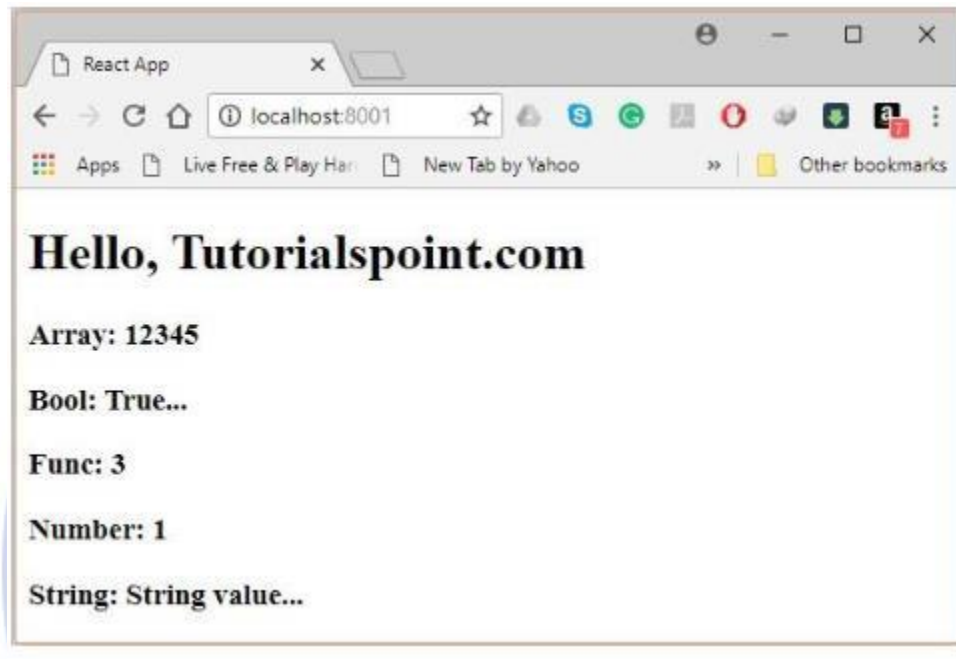
```



```
propFunc: function(e){return e},  
propNumber: 1,  
propString: "String value...",  
propObject: {  
  objectName1:"objectValue1",  
  objectName2: "objectValue2",  
  objectName3: "objectValue3"  
}  
}  
export default App;
```

main.js

```
import React from 'react';  
import ReactDOM from 'react-dom';  
import App from './App.jsx';  
ReactDOM.render(<App/>, document.getElementById('app'));
```



API جزء (API Component) در ReactJS

در این بخش می خواهیم به بررسی API جزء در ری اکت پردازیم. برای این منظور به سه متد `setState()`, `forceUpdate()` و `ReactDOM.findDOMNode()` می پردازیم. در کلاس های جدید ES6 باید این api را به صورت دستی مقید کنیم. در این مثال از `this.method.bind(this)` استفاده خواهیم کرد.

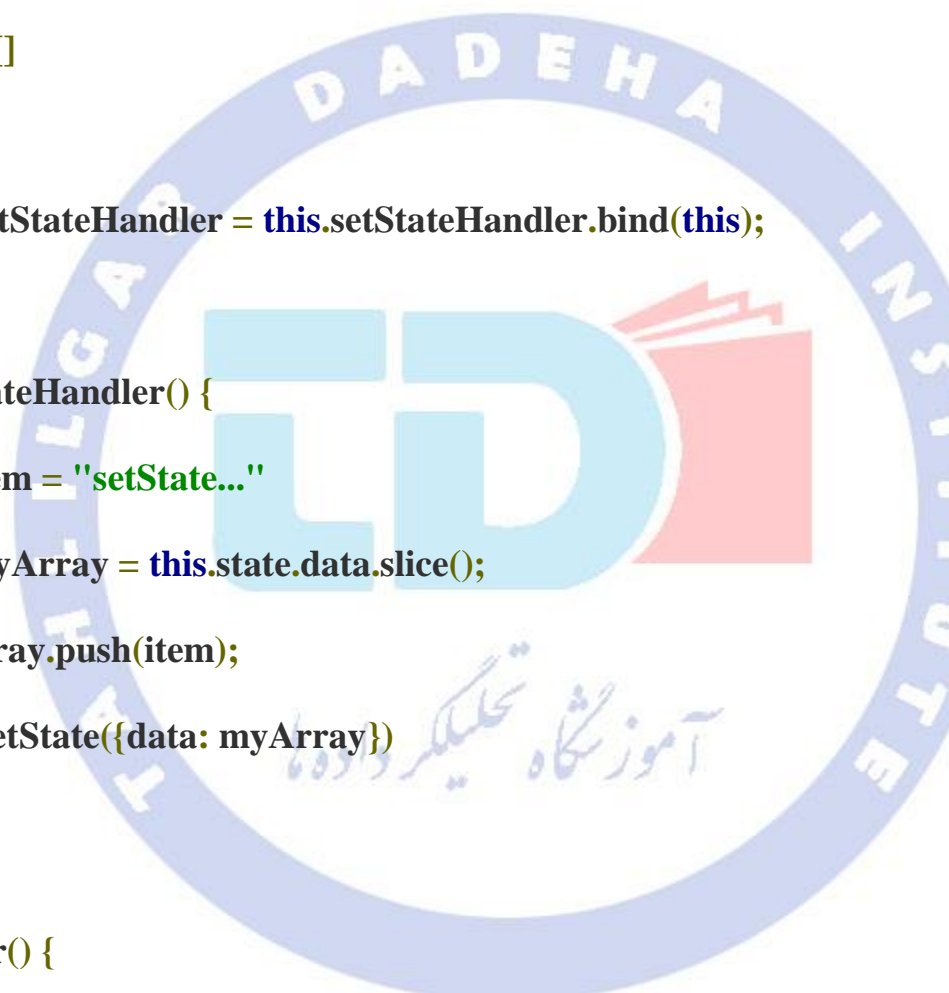
Set State

متد `setState()` متدی است که در بروز رسانی حالت جزء کاربرد دارد. این متد جای حالت را عوض نمی کند بلکه صرفا تغییراتی را بر حالت اصلی اعمال می کند.

```
import React from 'react';

class App extends React.Component {

  constructor() {
    super();
  }
}
```



```

this.state = {
  data: []
}

this.setStateHandler = this.setStateHandler.bind(this);
};

setStateHandler() {
  var item = "setState..."
  var myArray = this.state.data.slice();
  myArray.push(item);
  this.setState({data: myArray})
};

render() {
  return (
    <div>
      <button onClick = {this.setStateHandler}>SET STATE</button>
      <h4>State Array: {this.state.data}</h4>
    </div>
  );
}

```

```
}  
export default App;
```

ما کار خود را با یک آرایه ی خالی آغاز کرده ایم. هر بار که بر روی دکمه کلیک کنیم، حالت مورد نظر ما به روز رسانی می شود. اگر ۵ بار بر روی دکمه کلیک کنیم در این صورت نتیجه زیر را شاهد خواهیم بود.



Force Update

گاهی اوقات نیاز است که جزء خود را به صورت دستی به روز رسانی کنیم. برای انجام این کار می توانیم از متد `forceUpdate` استفاده کنیم.

```
import React from 'react';  
  
class App extends React.Component {  
  constructor() {  
    super();  
  
    this.forceUpdateHandler = this.forceUpdateHandler.bind(this);  
  };  
  
  forceUpdateHandler() {  
    this.forceUpdate();  
  }  
}
```

```

    };

    render() {
      return (
        <div>
          <button onClick = {this.forceUpdateHandler}>FORCE UPDATE</button>
          <h4>Random number: {Math.random()}</h4>
        </div>
      );
    }
  }
  export default App;

```

می خواهیم عددی تصادفی را تنظیم کنیم که هر بار که بر روی دکمه کلیک می کنیم به روز شود.



Find Dom Node

جهت دستکاری DOM میتوانیم از متد `ReactDOM.findDOMNode()` استفاده کنیم. ابتدا باید `react-dom` را ایمپورت کنیم.



```

import React from 'react';

import ReactDOM from 'react-dom';

class App extends React.Component {

  constructor() {

    super();

    this.findDOMNodeHandler = this.findDOMNodeHandler.bind(this);

  };

  findDOMNodeHandler() {

    var myDiv = document.getElementById('myDiv');

    ReactDOM.findDOMNode(myDiv).style.color = 'green';

  }

  render() {

    return (

      <div>

        <button onClick = {this.findDOMNodeHandler}>FIND DOME
        NODE</button>

        <div id = "myDiv">NODE</div>

      </div>

    );

  }

```



```
}
```

```
export default App;
```

بعد اینکه بر روی دکمه کلیک می شود , عنصر myDiv به رنگ سبز در می آید.



از به روز رسانی 0.14 , اغلب متدهای API اجزای قدیمی منسوخ یا حذف شده اند تا با ES6 سازگار شوند.

چرخه ی عمر اجزا (Component Life Cycle) در ReactJS

در این بخش می خواهیم به متدهای چرخه ی عمر اجزا بپردازیم.

متدهای چرخه ی عمر

- ComponentWillMount قبل از رندر شدن در هر دو سمت کلاینت و سرور اجرا می شود.
- ComponentDidMount پس از اولین رندر و تنها در سمت کلاینت اجرا می شود. در همین مکان است که درخواست های AJAX و DOM یا به روز رسانی حالات رخ می دهد. این متد در یکپارچه شدن با فریمورک های جاوا اسکریپت دیگر و هر تابعی که اجرای آن با تأخیر همراه است، مثل setTimeout و setInterval کاربرد دارد. ما برای به روز کردن حالت از این متد استفاده می کنیم تا بتوانیم متدهای چرخه ی عمر دیگر را فعال کنیم.

- `ComponentWillReceiveProps` به محض به روز شدن ویژگی ها قبل از فراخوانی رندر دیگر احضار می شود. ما این متد را از `setNewNumber` و زمانی که حالت را به روز کردیم، فعال کردیم.
 - `ShouldComponentUpdate` مقدار `true` یا `false` را برگشت می دهد. این متد به روز بودن یا نبودن جزء را مشخص می کند و به صورت پیش فرض بر روی `true` قرار دارد. اگر مطمئن هستید که جزء مورد نظر شما پس از به روز شدن حالت یا ویژگی ها نیازی به رندر ندارد، می توانید مقدار `false` را برگشت دهید.
 - `ComponentWillUpdate` درست قبل از رندر شدن فراخوانی می شود.
 - `ComponentDidUpdate` درست بعد از رندر شدن فراخوانی می شود.
 - `ComponentWillUnmount` پس از جدا شدن جزء از DOM فراخوانی می شود. ما جزء خود را در `main.js` از DOM جدا می کنیم.
- در مثال زیر حالت اولیه ی تابع سازنده را مشخص کرده ایم. جهت به روز رسانی حالت از `setNewnumber` استفاده شده است. تمامی متدهای چرخه ی عمر داخل جزء `Content` قرار دارند.

App.jsx

```
import React from 'react';

class App extends React.Component {

  constructor(props) {

    super(props);

    this.state = {

      data: 0

    }

  }

}
```

```

    this.setNewNumber = this.setNewNumber.bind(this)
  };
  setNewNumber() {
    this.setState({data: this.state.data + 1})
  }
  render() {
    return (
      <div>
        <button onClick = {this.setNewNumber}>INCREMENT</button>
        <Content myNumber = {this.state.data}></Content>
      </div>
    );
  }
}

class Content extends React.Component {
  componentWillMount() {

    console.log('Component WILL MOUNT!')
  }

  componentDidMount() {

```

```
console.log('Component DID MOUNT!')
}
componentWillReceiveProps(newProps) {
console.log('Component WILL RECIEVE PROPS!')
}
shouldComponentUpdate(newProps, newState) {
return true;
}
componentWillUpdate(nextProps, nextState) {
console.log('Component WILL UPDATE!');
}
componentDidUpdate(prevProps, prevState) {
console.log('Component DID UPDATE!')
}
componentWillUnmount() {
console.log('Component WILL UNMOUNT!')
}
render() {
return (
<div>
```

```
<h3>{this.props.myNumber}</h3>
```

```
</div>
```

```
);
```

```
}
```

```
}
```

```
export default App;
```

main.js

```
import React from 'react';
```

```
import ReactDOM from 'react-dom';
```

```
import App from './App.jsx';
```

```
ReactDOM.render(<App/>, document.getElementById('app'));
```

```
setTimeout(() => {
```

```
ReactDOM.unmountComponentAtNode(document.getElementById('app'));  
}, 10000);
```

پس از رندر اولیه نتیجه ی زیر نمایش داده می شود.



فرم ها (Forms) در ReactJS

در این بخش می خواهیم به چگونگی استفاده از فرم ها در ReactJS بپردازیم.

مثال ساده

در مثال زیر می خواهیم یک فرم ورودی را به کمک `value = {this.state.data}` تنظیم کنیم. از این طریق می توانیم هر زمان که مقدار ورودی تغییر می کند، حالت را به روز رسانی کنیم. ما در اینجا از رویداد `onChange` استفاده کرده ایم. این رویداد تغییرات ورودی را زیر نظر می گیرد و حالت را بر اساس آن به روز رسانی می کند.

App.jsx

```
import React from 'react';

class App extends React.Component {

  constructor(props) {

    super(props);

    this.state = {

      data: 'Initial data...'

    }

    this.updateState = this.updateState.bind(this);

  };

  updateState(e) {

    this.setState({data: e.target.value});
```



```

}

render() {
  return (
    <div>

    <input type = "text" value = {this.state.data}
    onChange = {this.updateState} />

    <h4>{this.state.data}</h4>

    </div>

  );
}
}

export default App;

```

main.js

```

import React from 'react';

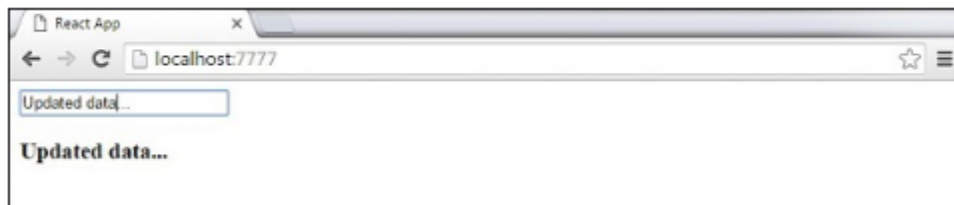
import ReactDOM from 'react-dom';

import App from './App.jsx';

ReactDOM.render(<App/>, document.getElementById('app'));

```

زمانی که مقدار متن ورودی تغییر می کند، حالت به روز رسانی می شود.



مثال پیچیده

در مثال زیر چگونگی استفاده از فرم ها از جزء فرزند نشان داده شده است. متد onChange به روز رسانی حالت را فعال می کند. سپس این به روز رسانی به مقدار ورودی فرزند داده می شود و در صفحه نمایش داده می شود. مثال مشابه در بخش رویدادها آورده شده است. هر زمان که نیاز باشد حالت را از جزء فرزند به روز رسانی کنیم، باید تابعی که به روز رسانی (updateState) را به صورت یک ویژگی (updateStateProp) مدیریت می کند را عبور دهیم.

App.jsx

```
import React from 'react';

class App extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      data: 'Initial data...'
    }

    this.updateState = this.updateState.bind(this);
  };

  updateState(e) {
```

```

    this.setState({data: e.target.value});
  }
  render() {
    return (
      <div>
        <Content myDataProp = {this.state.data}
          updateStateProp = {this.updateState}></Content>
      </div>
    );
  }
}

class Content extends React.Component {
  render() {
    return (
      <div>
        <input type = "text" value = {this.props.myDataProp}
          onChange = {this.props.updateStateProp} />
        <h3>{this.props.myDataProp}</h3>
      </div>
    );
  }
}

```

```
}  
  
}
```

```
export default App;
```

main.js

```
import React from 'react';  
import ReactDOM from 'react-dom';  
import App from './App.jsx';  
ReactDOM.render(<App/>, document.getElementById('app'));
```

این کار باعث می شود نتیجه ی زیر حاصل شود.



رویدادها (Events) مربوط به ReactJS

در این بخش می خواهیم به چگونگی استفاده از رویدادها بپردازیم.

مثال ساده

این مثال تا حدی ساده بوده و در آن می خواهیم تنها از یک جزء استفاده کنیم. در این مثال از رویداد onClick استفاده شده است. این رویداد پس از فشردن دکمه تابع updateState را فعال می کند.

App.jsx



```

import React from 'react';

class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      data: 'Initial data...'
    }
    this.updateState = this.updateState.bind(this);
  };
  updateState() {
    this.setState({data: 'Data updated...'})
  }
  render() {
    return (
      <div>
        <button onClick = {this.updateState}>CLICK</button>

        <h4>{this.state.data}</h4>
      </div>
    );
  }
}

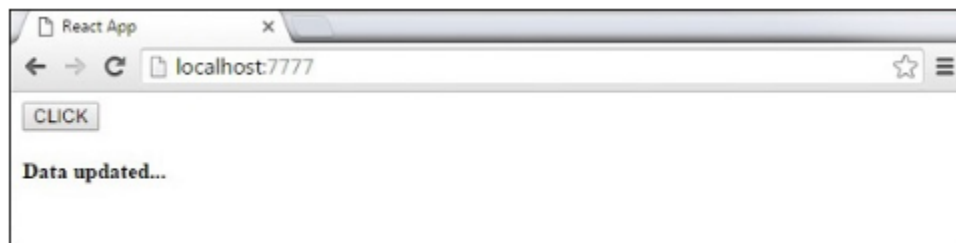
```

```
}  
  
}  
export default App;
```

main.js

```
import React from 'react';  
import ReactDOM from 'react-dom';  
import App from './App.jsx';  
ReactDOM.render(<App/>, document.getElementById('app'));
```

این کد نتیجه ی زیر را به همراه دارد.

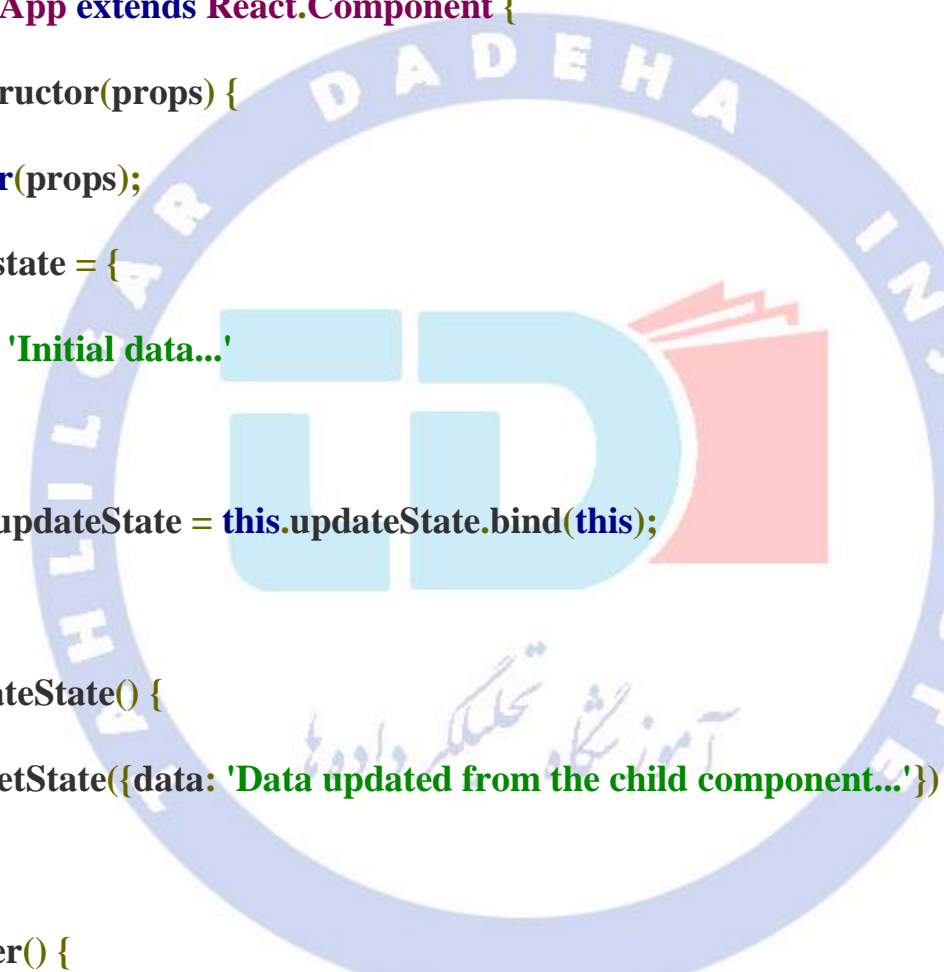


رویدادهای فرزند

ما باید حالت جزء مادر را از فرزند آن به روز رسانی کنیم. می توانیم مدیریت کننده ی رویدادی را (updateState) در جزء مادر ایجاد کنیم و آن را به عنوان یک ویژگی (updateStateProp) به جزء فرزند عبور دهیم و در آن جا صرفا آن را فراخوانی کنیم.

App.jsx

```
import React from 'react';
```



```

class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      data: 'Initial data...'
    }
    this.updateState = this.updateState.bind(this);
  };
  updateState() {
    this.setState({data: 'Data updated from the child component...'})
  }
  render() {
    return (
      <div>
        <Content myDataProp = {this.state.data}
          updateStateProp = {this.updateState}></Content>
      </div>
    );
  }
}

```



```

}

class Content extends React.Component {

  render() {

  return (

    <div>

    <button onClick = {this.props.updateStateProp}>CLICK</button>

    <h3>{this.props.myDataProp}</h3>

    </div>

  );

  }

}

export default App;

```

main.js

```

import React from 'react';

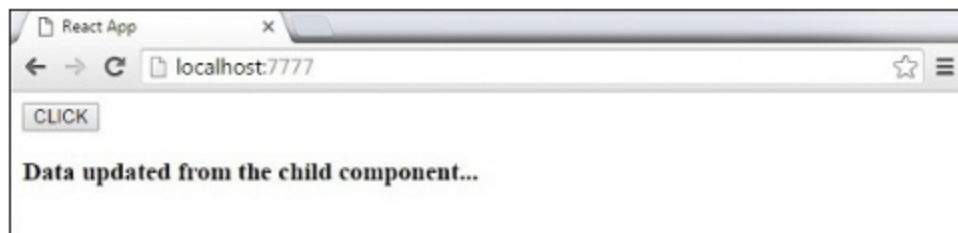
import ReactDOM from 'react-dom';

import App from './App.jsx';

ReactDOM.render(<App/>, document.getElementById('app'));

```

این کار نتیجه ی زیر را به همراه دارد.



Ref ها مربوط به ReactJS

Ref ها در برگشت دادن مرجعی به عنصر کاربرد دارند. در اغلب مواقع بهتر است از ref ها پرهیز کرد، اما اگر به اندازه گیری DOM و یا به افزودن متدها به اجزا نیاز داریم، ref ها می توانند به ما کمک کنند.

استفاده از Ref ها

در مثال زیر چگونگی استفاده از ref ها جهت پاک کردن کادر ورودی نشان داده شده است. تابع ClearInput به کمک مقدار `ref = "myInput"` به دنبال عنصر می گردد، حالت را ریست می کند و پس از فشرده شدن دکمه تمرکز را به آن وارد می کند.

App.jsx

```
import React from 'react';
import ReactDOM from 'react-dom';

class App extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      data: ''
    }
  }
}
```

```

    this.updateState = this.updateState.bind(this);
    this.clearInput = this.clearInput.bind(this);
  };
  updateState(e) {
    this.setState({data: e.target.value});
  }
  clearInput() {
    this.setState({data: ""});
    ReactDOM.findDOMNode(this.refs.myInput).focus();
  }
  render() {
    return (
      <div>
        <input value = {this.state.data} onChange = {this.updateState}
          ref = "myInput"></input>
        <button onClick = {this.clearInput}>CLEAR</button>
        <h4>{this.state.data}</h4>
      </div>
    );
  }

```

```
}
```

```
}
```

```
export default App;
```

main.js

```
import React from 'react';
```

```
import ReactDOM from 'react-dom';
```

```
import App from './App.jsx';
```

```
ReactDOM.render(<App/>, document.getElementById('app'));
```

بعد از فشردن شدن دکمه ورودی پاک شده و تمرکز بر روی آن قرار می گیرد.



کلیدها (Keys) در ReactJS

کلیدهای ReactJS، زمان کار با اجزایی که به صورت پویا ایجاد شده اند و یا در مواقعی که لیست های شما توسط کاربران تغییر داده شده اند، مفید هستند. با تنظیم مقدار key می توانید بعد از تغییرات منحصر به فرد بودن اجزای خود را حفظ کنید.

استفاده از کلیدها

بیا باید به صورت پویا عناصر Content را به همراه ایندکس منحصر به فرد (i) ایجاد کنیم. تابع map سه عنصر را از آرایه ی data ما ایجاد خواهد کرد. با توجه به اینکه مقدار key برای تمامی عناصر باید منحصر به فرد باشد، برای هر یک از عناصر ایجاد شده i را به عنوان یک کلید تعیین می کنیم.

App.jsx

```
import React from 'react';
```

```
class App extends React.Component {
```

```
  constructor() {
```

```
    super();
```

```
    this.state = {
```

```
      data:[
```

```
        {
```

```
          component: 'First...',
```

```
          id: 1
```

```
        },
```

```
        {
```

```
          component: 'Second...',
```

```
          id: 2
```

```
        },
```

```
        {
```

```
          component: 'Third...',
```

id: 3

```
}  
]  
}  
}  
  
render() {  
  return (  
    <div>  
      <div>  
        {this.state.data.map((dynamicComponent, i) => <Content  
          key = {i} componentData = {dynamicComponent}/>))  
      </div>  
    </div>  
  );  
}  
}  
  
class Content extends React.Component {  
  render() {  
    return (  
      <div>
```

```

<div>{this.props.componentData.component}
</div>
<div>{this.props.componentData.id}</div>
</div>
);
}
}
export default App;

```

main.js

```

import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.jsx';
ReactDOM.render(<App/>, document.getElementById('app'));

```

نتیجه ی زیر برای مقادیر Key هر یک از عناصر نمایش داده می شود .



اگر در آینده برخی از عناصر را حذف کنیم، آن ها را اضافه کنیم یا ترتیب عناصری که به صورت پویا ایجاد شده اند را تغییر دهیم، ReactJS برای حفظ ردگیری هر یک از عناصر از مقادیر key استفاده خواهد کرد.

روتر (Router) در ReactJS

در این بخش می خواهیم به چگونگی تنظیم مسیریابی در برنامه پردازیم.

مرحله - 1 نصب یک ReactJS روتر

ساده ترین راه برای نصب ReactJS روتر اجرای تکه کد زیر در پنجره ی cmd است.

```
C:\Users\username\Desktop\reactApp>npm install react-router
```

مرحله - 2 ایجاد اجزا

در این مرحله ۴ جزء را ایجاد خواهیم کرد. از جزء App به عنوان یک منوی تب استفاده خواهیم کرد. ۳ جزء دیگر (Home, About) و (Contact) بعد از تغییر مسیر رندر خواهند شد.

main.js

```
import React from 'react';

import ReactDOM from 'react-dom';
import { Router, Route, Link, browserHistory, IndexRoute } from 'react-router'

class App extends React.Component {

  render() {

    return (
```

```

<div>
  <ul>
    <li>Home</li>
    <li>About</li>
    <li>Contact</li>
  </ul>
  {this.props.children}
</div>
)
}
}

```

```
export default App;
```

```

class Home extends React.Component {
  render() {
    return (
      <div>
        <h1>Home...</h1>
      </div>
    )
  }
}

```

```

    }
  }
  export default Home;

  class About extends React.Component {
    render() {
    return (
      <div>
      <h1>About...</h1>
      </div>
    )
  }
}
export default About;

class Contact extends React.Component {
  render() {
    return (
      <div>
      <h1>Contact...</h1>
      </div>
    )
  }
}

```

```

}

}

export default Contact;

```

مرحله - 3 اضافه کردن روتر

حالا مسیرهایی را به برنامه اضافه می کنیم. به جای رندر کردن عنصر App مانند مثال قبل، این بار روتر رندر خواهد شد. همچنین برای هر یک از مسیرها اجزایی را تنظیم می کنیم.

main.js

```

ReactDOM.render((
  <Router history = {browserHistory}>
    <Route path = "/" component = {App}>
      <IndexRoute component = {Home} />
      <Route path = "home" component = {Home} />
      <Route path = "about" component = {About} />
      <Route path = "contact" component = {Contact} />
    </Route>
  </Router>
), document.getElementById('app'))

```

پس از باز شدن برنامه ۳ لینک قابل کلیک ظاهر می شوند که از طریق آن ها می توانیم تغییر مسیر دهیم.



مفهوم Flex در ReactJS

Flux یک مفهوم برنامه نویسی بوده که طی آن داده در یک مسیر حرکت می کند. این داده به برنامه وارد شده و تا زمانی که در صفحه نمایش داده شود، در آن به جریان می افتد.

عناصر فلاکس

در ادامه به توضیح ساده ای از مفهوم فلاکس پرداخته شده است. در بخش بعد به چگونگی اجرای این مفهوم در برنامه خواهیم پرداخت.

- Actions: اکشن ها جهت فعال کردن جریان داده به dispatcher ارسال می شوند.
- Dispatcher: این بخش مرکز مهم برنامه است. تمامی داده ها به store ها ارسال می شوند.
- Store: Store مکانی است که در آن حالت و منطق برنامه نگهداری می شوند. هر یک از Store ها حالت خاصی را نگهداری می کند و هر زمان که نیاز باشد آن را به روز می کند.
- View: view داده را از Store دریافت کرده و برنامه را مجددا رندر می کند.

جریان داده در تصویر زیر نمایش داده شده است.



مزایای فلاکس

- درک جریان یک طرفه ی داده آسان است.
- نگهداری از برنامه آسان تر است.
- اجزای برنامه به یکدیگر کوپل نیستند.

استفاده از Flux در ReactJS

در این بخش می خواهیم به چگونگی پیاده سازی الگوی فلاکس در برنامه های ReactJS بپردازیم. برای انجام این کار از فریمورک Redux استفاده می کنیم. هدف این فصل ارائه ی ساده ترین مثال از تمامی بخش های مورد نیاز جهت اتصال Redux و ReactJS است.

مرحله 1 – نصب Redux

Redux را از طریق پنجره ی cmd نصب می کنیم.

```
C:\Users\username\Desktop\reactApp>npm install --save react-redux
```

مرحله 2 – ایجاد فایل ها و فولدرها

در این بخش می خواهیم پوشه ها و فولدرهای مربوط به action ها، reducer ها و component های خود را ایجاد کنیم. بعد از انجام این کار ساختار پوشه ها به شکل زیر درخواهد آمد.

```
C:\Users\Tutorialspoint\Desktop\reactApp>mkdir actions
C:\Users\Tutorialspoint\Desktop\reactApp>mkdir components
C:\Users\Tutorialspoint\Desktop\reactApp>mkdir reducers
C:\Users\Tutorialspoint\Desktop\reactApp>type nul > actions/actions.js
C:\Users\Tutorialspoint\Desktop\reactApp>type nul > reducers/reducers.js
C:\Users\Tutorialspoint\Desktop\reactApp>type nul > components/AddTodo.js
C:\Users\Tutorialspoint\Desktop\reactApp>type nul > components/Todo.js
C:\Users\Tutorialspoint\Desktop\reactApp>type nul > components/TodoList.js
```



مرحله ۳ – Action ها

اکشن ها اشیاء جاوا اسکریپتی هستند که جهت اطلاع دادن به داده برای ارسال شدن به store از ویژگی type استفاده می کنند. ما می خواهیم عمل ADD_TODO را تعریف کنیم. این عمل در اضافه کردن آیتم جدید به لیست کاربرد دارد. تابع addTodo ایجاد کننده ی عملی است که عمل ما را برگشت می دهد و برای هر یک از آیتم های ایجاد شده شناسه ای را تنظیم می کند.

Actions/actions.js


```
export const ADD_TODO = 'ADD_TODO'
```

```
let nextTodoId = 0;
```

```
export function addTodo(text) {
```

```
  return {
```

```
    type: ADD_TODO,
```

```
    id: nextTodoId++,
```

```
    text
```

```
  };
```

```
}
```

مرحله ۴ - Reduser ها

در حالی که عمل ها تنها تغییرات را در برنامه فعال می کنند، reducer ها این تغییرات را مشخص می کنند. ما برای جستجوی عمل ADD_TODO از دستور switch استفاده می کنیم. reducer تابعی است که جهت محاسبه و برگشت یک حالت به روز شده دو پارامتر (action و state) را می گیرد.

تابع اول در ایجاد آیتم جدید و تابع دوم در ارسال این آیتم به لیست کاربرد دارد. در همین راستا ما از تابع کمکی combineReducers استفاده می کنیم. در این تابع می توانیم هر reducer جدیدی را برای روز مبدا اضافه کنیم.

Reducers/reducers.js

```
import { combineReducers } from 'redux'
```

```
import { ADD_TODO } from '../actions/actions'
```

```
function todo(state, action) {
```

```
switch (action.type) {
```

```
  case ADD_TODO:
```

```
    return {
```

```
      id: action.id,
```

```
      text: action.text,
```

```
    }
```

```
  default:
```

```
    return state
```

```
  }
```

```
}
```

```
function todos(state = [], action) {
```

```
  switch (action.type) {
```

```
    case ADD_TODO:
```

```
      return [
```

```
        ...state,
```

```
        todo(undefined, action)
```

```
      ]
```

```
    default:
```

```

return state
}
}

const todoApp = combineReducers({
  todos
}) export default todoApp

```

مرحله ۵ - Store

Store مکانی است که در آن حالت برنامه نگهداری می شود. بعد از در اختیار داشتن reducer ها، ایجاد یک Store کار آسانی است. ما ویژگی Store را به عنصر provider عبور می دهیم که این عنصر جزء route ما را پوشش می دهد.

main.js

```

import React from 'react'

import { render } from 'react-dom'

import { createStore } from 'redux'

import { Provider } from 'react-redux'

import App from './App.jsx'

import todoApp from './reducers/reducers'

let store = createStore(todoApp)

```

```

let rootElement = document.getElementById('app')

render(
  <Provider store = {store}>
    <App />
  </Provider>,
  rootElement
)

```

مرحله ۶ - جزء اصلی

جزء App ، جزء اصلی برنامه است. تنها جزء اصلی باید نسبت به Redux آگاه باشد. بخش مهمی که باید به آن توجه کرد، تابع connect است. این تابع جهت اتصال جزء اصلی App به store کاربرد دارد. این تابع، تابع select را به عنوان یک آرگومان می گیرد. تابع select حالت را از store گرفته و ویژگی هایی (visibleTodos) را برگشت می دهد که از آن ها می توانیم در اجزای خود استفاده کنیم.

App.jsx

```

import React, { Component } from 'react'


import { connect } from 'react-redux'

import { addTodo } from './actions/actions'

import AddTodo from './components/AddTodo.js'

import TodoList from './components/TodoList.js'

```



```

class App extends Component {
  render() {
    const { dispatch, visibleTodos } = this.props
    return (
      <div>
        <AddTodo onClick = {text => dispatch(addTodo(text))} />
        <TodoList todos = {visibleTodos}/>
      </div>
    )
  }
}

function select(state) {
  return {
    visibleTodos: state.todos
  }
}

export default connect(select)(App);

```

این اجزا نباید نسبت به Redux آگاه باشند.

Components/AddTodo.js

```
import React, { Component, PropTypes } from 'react'

export default class AddTodo extends Component {

  render() {

  return (

    <div>

      <input type = 'text' ref = 'input' />

      <button onClick = {(e) => this.handleClick(e)}>

Add

      </button>

    </div>

  )

}

  handleClick(e) {

const node = this.refs.input

const text = node.value.trim()

this.props.onAddClick(text)
```

```
node.value = ''
```

```
}
```

```
}
```

Components/ToDo.js

```
import React, { Component, PropTypes } from 'react'
```

```
export default class Todo extends Component {
```

```
  render() {
```

```
    return (
```

```
      <li>
```

```
        {this.props.text}
```

```
      </li>
```

```
    )
```

```
  }
```

```
}
```

Components/ToDoList.js

```
import React, { Component, PropTypes } from 'react'
```

```
import Todo from './Todo.js'
```

```
export default class ToDoList extends Component {
```



```

render() {
  return (
    <ul>
      {this.props.todos.map(todo =>
        <Todo
          key = {todo.id}
          {...todo}
        />
      )}
    </ul>
  )
}
}

```

بعد از اجرای برنامه می توانیم آیتم هایی را به لیست خود اضافه کنیم.



انیمیشن ها (Animation) در ReactJS

در این بخش می خواهیم به چگونگی متحرک کردن عناصر با استفاده از ReactJS بپردازیم.

مرحله ۱ : نصب React CSS Transitions Group

این برنامه افزونه ای برای ReactJS بوده که با کمک آن می توان انیمیشن ها و گذارهای CSS اولیه را ایجاد کرد. برای نصب آن از پنجره ی cmd کمک می گیریم.

```
C:\Users\username\Desktop\reactApp>npm install react-addons-css-transition-group
```

مرحله ۲ : اضافه کردن یک فایل CSS

بیا یک فایل جدید style.css را ایجاد کنیم.

```
C:\Users\Tutorialspoint\Desktop\reactApp>type nul > css/style.css
```

برای اینکه بتوانیم در برنامه خود از این فایل استفاده کنیم , باید آن را به عنصر هد index.html پیوند دهیم.

```
<!DOCTYPE html>
<html lang = "en">
<head>
  <link rel = "stylesheet" type = "text/css" href = "./style.css">
  <meta charset = "UTF-8">
  <title>React App</title>
</head>
<body>
  <div id = "app"></div>
  <script src = 'index_bundle.js'></script>
</body>
```

</html>

مرحله ۳: ظاهر کردن انیمیشن

حالا که می خواهیم یک جزء ابتدایی ReactJS را ایجاد کنیم ، از عنصر **ReactCSSTransitionGroup** به عنوان پوشاننده ی جزئی که ما می خواهیم آن را متحرک کنیم استفاده می شود. در اینجا از **transitionAppear** و **transitionAppearTimeout** استفاده خواهیم کرد در حالی که **transitionEnter** و **transitionLeave** false هستند.

App.jsx

```
import React from 'react';

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

class App extends React.Component {

  render() {

    return (

      <div>

        <ReactCSSTransitionGroup transitionName = "example"

          transitionAppear = {true} transitionAppearTimeout = {500}

          transitionEnter = {false} transitionLeave = {false}>

          <h1>My Element...</h1>

        </ReactCSSTransitionGroup>

      </div>
```

```
);
```

```
}
```

```
}
```

```
export default App;
```

main.js

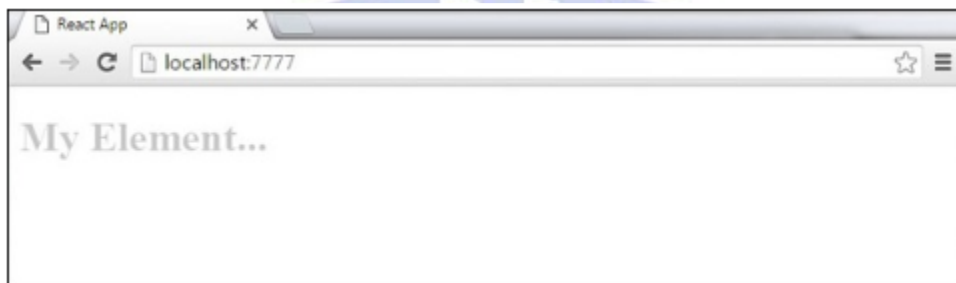
```
import React from 'react'  
import ReactDOM from 'react-dom';  
import App from './App.jsx';  
ReactDOM.render(<App />, document.getElementById('app'));
```

متحرک سازی CSS خیلی ساده است.

Css/style.css

```
.example-appear {  
  
  opacity: 0.04;  
  
}  
  
.example-appear.example-appear-active {  
  
  opacity: 2;  
  
  transition: opacity 50s ease-in;  
}
```

بعد از باز کردن برنامه عنصر مورد نظر ما محو می شود.



مرحله ۴: متحرک سازی ورود و خروج

از این متحرک سازی ها می توانیم در زمانی استفاده کنیم که نیاز است عناصری را از لیست حذف و یا به آن اضافه کنیم.

App.jsx

```
import React from 'react';

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

class App extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      items: ['Item 1...', 'Item 2...', 'Item 3...', 'Item 4...']
    }

    this.handleAdd = this.handleAdd.bind(this);
  }
};
```

```

handleAdd() {
  var newItems = this.state.items.concat([prompt('Create New Item')]);
  this.setState({items: newItems}); }

handleRemove(i) {
  var newItems = this.state.items.slice();
  newItems.splice(i, 1);
  this.setState({items: newItems});
}

render() {
  var items = this.state.items.map(function(item, i) {
    return (
      <div key = {item} onClick = {this.handleRemove.bind(this, i)}>
        {item}
      </div>
    );
  }).bind(this));

  return (
    <div>
      <button onClick = {this.handleAdd}>Add Item</button>

```

```

<ReactCSSTransitionGroup transitionName = "example"
transitionEnterTimeout = {500} transitionLeaveTimeout = {500}>
  {items}
</ReactCSSTransitionGroup>
</div>
);
}
}
export default App;

```

main.js

```

import React from 'react'
import ReactDOM from 'react-dom';
import App from './App.jsx';

ReactDOM.render(<App />, document.getElementById('app'));

```

Css/style.css

```

.example-enter {

opacity: 0.04;

}

```



```
.example-enter.example-enter-active {
```

```
opacity: 5;
```

```
transition: opacity 50s ease-in;
```

```
}
```

```
.example-leave {
```

```
opacity: 1;
```

```
}
```

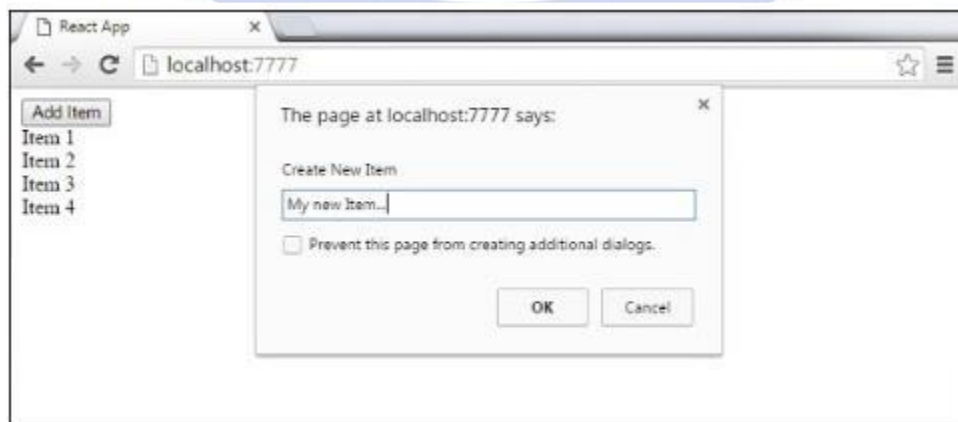
```
.example-leave.example-leave-active {
```

```
opacity: 0.04;
```

```
transition: opacity 50s ease-in;
```

```
}
```

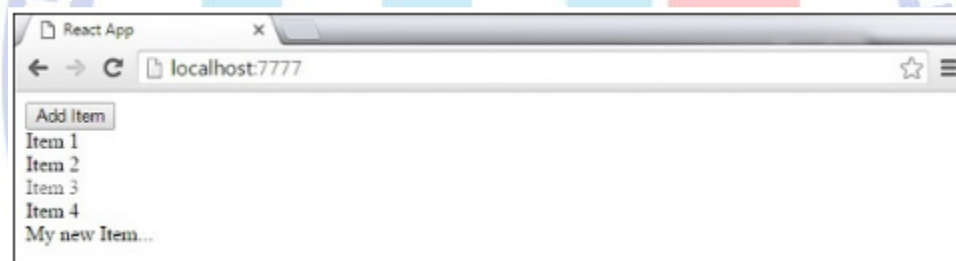
پس از باز کردن برنامه و کلیک کردن بر روی دکمه Add Item پیام زیر نمایش داده میشود.



بعد از وارد کردن اسم و OK کردن، عنصر جدید در حالت وارد شدن محو می شود.



حالا می توانیم با کلیک کردن بر روی آیتم ها، برخی از آن ها (Item 3...) را حذف کنیم. این کار باعث می شود این آیتم در حال خروج از لیست محو شود.



اجزای با مرتبه ی بالاتر (Higher Order Components) در ReactJS

این اجزا توابع جاوا اسکریپتی هستند که جهت افزودن قابلیت های بیشتر به اجزای موجود کاربرد دارند. این توابع خالص هستند، به این معنی که داده ها را دریافت می کنند و بر اساس این داده مقادیر را برگشت می دهند. اگر این داده تغییر کند، توابع با مرتبه ی بالاتر با ورودی داده ی مختلف مجددا اجرا می شوند. اگر بخواهیم جزء برگشتی خود را به روز رسانی کنیم، نیازی نیست که این اجزا را تغییر دهیم. تنها کاری که باید انجام دهیم این است که داده ای که تابع ما در حال استفاده از آن است را تغییر دهیم.

جزء با مرتبه ی بالاتر (HOC) جزء معمولی را پوشانده است و داده های ورودی بیشتری را فراهم می کند. این جزء در واقع تابعی است که یک جزء را می گیرد و جزء دیگری که جزء اصلی را می پوشاند را برگشت می دهد.

بیاید برای درک بهتر این مفهوم به مثال ساده ای نگاه کنیم. MyHOC تابع با مرتبه ی بالاتری است که تنها جهت عبور دادن داده به MyComponent کاربرد دارد. این تابع MyComponent را دریافت می کند، آن را به کمک newData ارتقا می دهد و جزء ارتقا یافته که در صفحه نمایش داده می شود را برگشت می دهد.

```
import React from 'react';

var newData = {
  data: 'Data from HOC...',
}

var MyHOC = ComposedComponent => class extends React.Component {
  componentDidMount() {
    this.setState({
      data: newData.data
    });
  }
  render() {
    return <ComposedComponent {...this.props} {...this.state} />;
  }
}
```

```

}
};
class MyComponent extends React.Component {
  render() {
    return (
      <div>
        <h1>{this.props.data}</h1>
      </div>
    )
  }
}
export default MyHOC(MyComponent);

```

اگر برنامه را اجرا کنیم، خواهیم دید که این داده به MyComponent عبور داده شده است.



نکته:

از اجزای با مرتبه ی بالاتر می توان برای کارایی های مختلفی استفاده کرد. این توابع خالص اساس برنامه نویسی تابعی را تشکیل می دهند. بعد از آن که به آن عادت کردید، خواهید دید که چقدر نگهداری و ارتقاء برنامه برایتان راحت می شود.

بهترین شیوه ها در ReactJS

در این بخش می خواهیم لیستی از بهترین شیوه ها، روش ها و تکنیک هایی که در برنامه نویسی به ما کمک می کنند را ارائه کنیم.

- State: تا حد امکان باید از State یا حالت دوری کرد. جهت متمرکز کردن حالت و عبور دادن آن به درخت اجزا به عنوان ویژگی این روش، روش مناسبی است. هر زمان که با گروهی از اجزا سروکار داشته باشیم، به گونه ای که این اجزا به داده های یکسانی نیاز داشته باشند، باید حول آن ها یک عنصر نگهدارنده را تنظیم کنیم تا حالت را در خود نگهداری کند. الگوی فلاکس برای مدیریت حالت در برنامه های ReactJS روش خوبی است.
- PropTypes: همیشه باید PropTypes ها را تعریف کرد. از این طریق می توانیم تمامی ویژگی های برنامه را ردیابی کنیم و راحت تر بر روی پروژۀ ی یکسان کار کنیم.
- Render: منطق اغلب برنامه ها باید داخل متد render در حرکت باشد. تا حد امکان باید سعی کنیم منطق متدهای چرخه ی عمر اجزا را به حداقل برسانیم و این منطق را به متد رندر انتقال دهیم. هرچه کمتر از حالت و ویژگی ها استفاده کنیم، کدمان تمیزتر می شود. همیشه باید تا حد امکان حالت را ساده کنیم. اگر می خواهیم چیزی را از حالت یا ویژگی ها محاسبه کنیم، می توانیم این کار را داخل متد رندر انجام دهیم.

- Composition: گروه ReactJS پیشنهاد می کند که از اصل مسئولیت واحد استفاده شود. به این معنی که تنها یک جزء باید مسئول یک کار باشد. اگر برخی از اجزا بیش از یک کار داشته باشند، در این صورت باید آن ها را تجزیه کنیم و برای هر کار یک جزء جدید ایجاد کنیم.
- اجزای با مرتبه ی بالاتر (HOC): در نسخه های قبلی ReactJS برای مدیریت کارهای با قابلیت استفاده ی مجدد میکسین هایی پیش بینی شده بود. با توجه به این که این میکسین ها در حال حاضر منسوخ شده اند، یکی از راه حل های جایگزین استفاده از HOC است.