



# 처음 시작하는 리액트

UI를 위한  
자바스크립트 라이브러리 ReactJS

톰 헬럿, 리차드 펠드만,  
시몬 회벡, 칼 미켈슨,  
존 비비, 프랑키 반야르디 지음 /  
곽현철, 김훈민 옮김



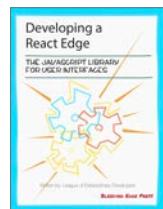


# 처음 시작하는 리액트

UI를 위한  
자바스크립트 라이브러리 ReactJS

톰 헬것, 리처드 펠드만,  
시몬 화벡, 칼 미켈슨,  
존 비비, 프랑키 반야르디 지음 /  
곽현철, 김훈민 옮김

이 도서는  
Developing a React Edge(BLEEDING EDGE PRESS)의  
번역서입니다





표지 사진 김재영

이 책의 표지는 김재영 님이 보내 주신 풍경사진을 담았습니다.

리얼타임은 독자의 시선을 담은 풍경사진을 책 표지로 보여주고자 합니다.

사진 보내기 [ebookwriter@hanbit.co.kr](mailto:ebookwriter@hanbit.co.kr)

## 처음 시작하는 리액트 UI를 위한 자바스크립트 라이브러리 ReactJS

---

초판발행 2016년 7월 25일

지은이 톰 헬럿, 리차드 펠드만, 시몬 회비, 칼 미켈슨, 존 비비, 프랑基 반야르디 / 옮긴이 곽현철, 김훈민 / 펴낸이 김태현  
펴낸곳 한빛미디어(주) / 주소 서울시 마포구 양화로 7길 83 한빛미디어(주) IT출판부

전화 02-325-5544 / 팩스 02-336-7124

등록 1999년 9월 30일 제10-1779호

ISBN 978-89-6848-775-0 15000 / 정가 17,000원

총괄 전태호 / 책임편집 김창수 / 기획·편집 김상민

디자인 표지·내지 여동일, 조판 최승실

마케팅 박상용, 송경석, 변지영 / 영업 김형진, 김진불, 조유미

이 책에 대한 의견이나 오桀자 및 잘못된 내용에 대한 수정 정보는 [한빛미디어\(주\)](http://www.hanbit.co.kr)의 홈페이지나 아래 이메일로 알려주십시오.  
**한빛미디어 홈페이지** [www.hanbit.co.kr](http://www.hanbit.co.kr) / **이메일** [ask@hanbit.co.kr](mailto:ask@hanbit.co.kr)

---

© 2016 HANBIT Media, Inc.

Authorized Korean translation of the English edition of *Developing a React Edge, ISBN 9781939902122*

© 2014 Frankie Bagnardi, Jonathan Beebe, Richard Feldman, Tom Hallett, Simon Højberg, and Karl Mikkelsen.

This translation is published and sold by permission of Bleeding Edge Press, Inc., which owns or controls all rights to publish and sell the same.

이 책의 저작권은 블리딩 엣지 프레스 사와 한빛미디어(주)에 있습니다.

저작권법에 의해 한국 내에서 보호를 받는 저작물이므로 무단 전재와 복제를 금합니다.

---

지금 하지 않으면 할 수 없는 일이 있습니다.

책으로 펴내고 싶은 아이디어나 원고를 메일([ebookwriter@hanbit.co.kr](mailto:ebookwriter@hanbit.co.kr))로 보내주세요.

한빛미디어(주)는 여러분의 소중한 경험과 지식을 기다리고 있습니다.

## 저자 소개

### 지은이\_ 툴 헬럿

샌프란시스코에 있는 실시간 비디오 플랫폼인 Tout.com의 Ruby/JavaScript 시니어 엔지니어이다. Jasmine을 이용한 React 애플리케이션 테스트를 도와주는 Jasmine-react의 제작자이기도 하다. 수중 폴로를 좋아하고, 아내와 아들과 함께 시간을 보낸다.

### 지은이\_ 리차드 펠드만

샌프란시스코에 있는 교육 기술 회사인 NoRedInk에서 리드 프론트엔드 엔지니어로 일하고 있다. 함수형 프로그래밍의 지지자이자, 강연자다. 일반적인 JavaScript 객체와 배열에 하위호환성을 갖는 이뮤티블 데이터 구조를 제공하는 오픈 소스 라이브러리인 *seamless-immutable*의 제작자이기도 하다.

### 지은이\_ 사몬 회美貌

로드아일랜드주 프로비던스에 있는 Swipely에서 시니어 UI 엔지니어로 일하고 있다. 프로비던스 JS 맷업 그룹의 공동주최자이고, 보스턴의 Startup Institute에서 JavaScript를 가르치기도 했다. JavaScript를 이용한 가능한 유저 인터페이스를 만들고, cssarrowplease.com 같은 사이드 프로젝트에 시간을 할애하고 있다.

### **지은이\_ 칼 미켈슨**

LockedOn에서 시니어 PHP/JavaScript 엔지니어로 일하면서 아름답고 강력한 부동산 소프트웨어를 만들고 있다. Karl은 새로운 기술에 대한 열정이 있고, 새로운 방법으로 일하기 위해 공부하는 것을 즐긴다. 자신의 웹사이트인 karlmikko.com에서 그를 찾을 수 없다면, 아내와 함께 암벽등반을 하고 있거나 커피를 즐기고 있을 것이다.

### **지은이\_ 존 비비**

Dave Ramsey의 디지털 개발팀에서 애플리케이션을 개발하고 있다. 웹과 iOS를 위해 사용자를 대하는 기술에 집중하고 있다. Final Cut Pro와 Motion에 사용하는 플러그인과 PHP 웹서비스를 만들기도 했다. Beebe가 예술과 코드에 관한 언어를 섞는 날은 좋은 날이다. 그는 독서광이고, 사진을 좋아하며, 매일 아내의 기대 이상을 달성하기 위해 노력하고 있다.

### **지은이\_ 프랑키 반야르디**

여러 고객을 위해 사용자 경험을 만드는 시니어 프론트엔드 개발자이다. 여가는 StackOverflow(FakeRainBrigand)와 IRC(GreenJello)에 올라오는 질문에 답해주는 한편 작은 프로젝트도 즐긴다.

## 역자 소개

### 옮긴이\_ 곽현철

NHN Technology Services에서 UI 개발자로 일하다가 지금은 티켓몬스터에서 프론트엔드 개발자로 일하고 있다. 좋은 동료가 되겠다는 핑계로 개발 욕심보다 개그 욕심이 많아서 주변에 웃음을 전하느라 바쁜, 조금 재미있는 사람. 아마 지금도 어디선 가 농담을 던지고 있을 것이다.

### 옮긴이\_ 김훈민

NHN Technology Services에서 근무하던 시절 본격적으로 프론트엔드 개발에 입문하여 지금은 네이버에서 스마트에디터3를 개발하고 있다. 잠들어버린 <http://huns.me> 블로그를 운영하고 있으며 테스트 프로세스에 관심이 많다.

요즘 웹 프론트엔드는 그야말로 춘추전국시대다. 수많은 도구, 라이브러리, 프레임워크가 오늘 쏟아지고 내일 사라진다. SPA를 위한 Full Framework로서 한 시대를 풍미했던 Backbone은 벌써 낡은 기술이 되었다. Backbone이 지나간 자리는 구글을 등에 업은 Angular가 차지했다. Angular는 한때 천하를 통일할 가장 유력한 후보였다. 엄청난 기세로 개발자들의 마음을 사로잡으며 높은 점유율을 차지하는 데 성공했지만, 고질적인 성능 이슈와 완전히 새로운 버전인 Angular 2 발표로 인한 역풍 등으로 요즘은 기세가 한풀 꺾인 모습이다.

React는 Angular 이후, 최근 프론트엔드 영역에서 가장 많은 관심을 받는 기술이다. 처음 React가 등장했을 때 커뮤니티의 반응은 호의적이지 않았다. 많은 것이 불편했다. Virtual DOM, JSX 같은 개념은 낯설었고, 페이지 단위로 개발하는 데 익숙해 있던 개발자들은 컴포넌트 단위로 사고하기 힘들어했다. 더군다나 HTML과 JavaScript를 한 곳에 묶어두는 React의 개발 방식은 기능과 구조라는 두 개의 관심사를 분리하라는 오랜 불문율에 어긋났다.

하지만 Facebook은 포기하지 않았고, 전 세계 수억 명이 사용하는 자사 제품의 곳곳에 React를 심었다. 끊임없이 React의 철학을 이야기하며 설득했다. 시간이 지나면서 React에 호감을 보이는 개발자들이 늘기 시작했다. Airbnb, 넷플릭스, Atlassian, BBC 같은 거대 기업이 React를 자사 프로젝트에 사용했다. 분위기가 달라졌다. 때마침 국내에도 React 바람이 불었다. React Native의 등장이 한 몫 했다. "Learn once, Write everywhere"라는 캐치프레이즈에 많은 개발자가 설렜다. 그렇게 React는 프론트엔드를 넘어 모바일 앱까지 조금씩 스며들었다. React 관련 커뮤니티가 문을 열었고, React를 주제로 하는 블로그 포스트가 하나둘 등장했다. 국내외에서 부지런히 영역을 넓힌 React는, 이제 설계부터 개발까지 다양한 플러그인과 라이브러리를 가진 차세대 프론트엔드 개발 도구로 많은 사랑을 받고 있다.

이 책은 React를 소개하지만, React를 처음 접하는 사람을 위한 책은 아니다. 간단한 튜토리얼 정도는 해봤으나 React의 장점이 뭔지 아직 아리송하다는 사람에게 더 적합하다. 6명의 저자가 다양한 주제로 React의 가치와 철학을 이야기한다. 여러 관점에서 바라본 React를 간접 경험해 볼 수 있다. 하지만 저자가 여러 명이다 보니 읽다 보면 전체 내용이 하나의 줄기로 매끄럽게 연결되어 있지 못한 듯한 느낌을 받을 수 있다. 사용자들이 처음부터 끝까지 직접 따라 해 볼 수 있을 만한 예제가 부족하다는 점도 아쉽다. 대신 React를 지탱하는 철학이 무엇인지, 어떻게 활용할 수 있는지를 고민하면서 이 책을 읽을 것을 추천한다. 설치 방법에 대한 설명이나 간단한 튜토리얼은 React 공식 문서(<https://facebook.github.io/react/docs/getting-started.html>)를 추천한다. 영어가 어렵다면 한국어로 번역한 사이트(<http://reactkr.github.io/react/docs/getting-started-ko-KR.html>)도 있다.

원서가 나온 후, 책을 번역하고 출간하기까지 여타의 사정으로 인해 시간이 지연되어 책이 쓰일 당시 0.11이었던 React의 버전이 이제는 0.14 RC까지 올라가 버렸다. 따라서 책에 있는 내용을 최신 버전에 그대로 적용할 수 없는 경우가 있다. 번역서라는 특성상 원문을 수정하기가 쉽지 않기 때문에 논의 끝에 React 0.14 RC를 반영하는 역자 주석을 추가하기로 했다. 또한, 원서에는 없는 React 릴리즈 로그를 부록으로 수록했다. 릴리즈 로그를 보면 지원을 중단하는 API와 React가 그동안 변해온 과정을 알 수 있다. React를 맛은 봤지만, 그 맛이 정확히 뭔지 잘 모르겠다는 분들이 이 책으로 인해 React와 더 친해질 수 있기를 바란다.

끝으로 전문 직업인으로서 배울 점이 많은 존경하는 상훈 님, 개발자로서 성장하는 데 많은 영감과 지식을 준 대선이 형, 방황하던 내 인생에 반전을 만들어준 태훈 님, 지칠 때마다 끊임없이 열정을 불어넣어 주는 우영이, 함께 번역하고 교정하느라 고생한 능력자 현철 님, 한결같이 옆에서 나를 응원해주는 지원이에게 고맙다는 말을 전한다. 이

책을 선택한 모든 독자의 앞날에 축복이 가득하길 바라며 서문을 마친다.

김훈민



최근 몇 년간 Javascript 개발 환경의 눈부신 발전만큼, HTML/CSS를 이용한 UI 개발도 빠르게 변화해왔다. Bootstrap 같은 프레임워크의 사용은 이제 흔한 일이 되었고, BEM, OOCSS 같은 CSS 방법론들이 UI를 바라보는 새로운 시야를 제공하고 있다. 이런 영향으로 UI 개발도 페이지 또는 화면 단위의 개발 방식에서 탈피하여, 컴포넌트라고 부를 만한 재사용성이 높은 UI를 바탕으로 한 스타일 가이드 기반의 개발로 점차 모양새가 바뀌고 있다.

React는 이런 새로운 물결과 잘 어울린다. 컴포넌트와 상태를 중심으로 UI를 개발하면, 화면 구성을 위한 복잡한 논리 구조를 단순하게 풀어낼 수 있다. React에 대한 이해는 UI의 분리를 위해 선행되어야 할 생각의 분리에도 도움을 준다. React가 가져다주는 이런 단편적인 생각의 전환만으로도 자신의 생산성이 높아지는 것을 느끼게 될 것이다.

끝으로 엉성한 번역을 함께 살펴봐 주고 글로 만들어준 훈민 님, 번역의 길에 다리를 놓아준 상훈 님, 개발의 길을 열어준 형국이형, 고락을 함께 하고 있는 표준화개발유닛 동료들, 그리고 늘 나를 웃게 해주는 영원이에게 감사 인사를 전한다.

곽현철

## React는 무엇이고 왜 써야 할까?

---

React는 Facebook에서 내부적으로 개발한 JavaScript 라이브러리로 2013년에 오픈 소스로 공개되었다. 웹에서 상호작용하는 사용자 인터페이스를 만들기 위한 라이브러리이다. React는 브라우저 DOM을 다루는 새로운 방법을 소개했다. 확장성과 새로운 기능의 개발을 위해 수동으로 DOM을 갱신하고 어렵게 각 상태를 추적하는 노력은 이제 옛날이야기가 되었다.

React는 매우 새로운 방법으로 DOM을 다룬다. 아무 때나 선언적으로 사용자 인터페이스를 정의할 수 있다. React는 데이터가 변경되었을 때 어떤 부분의 DOM을 갱신할지 신경 쓰지 않는다. 언제든지 최소한의 DOM 수정 때문에 전체 애플리케이션을 재 렌더링한다.

## 이 책에서 얻을 수 있는 것

---

React는 현재의 방법들에 도전하는 새롭고 흥미로운 개념을 소개한다. 이 책은 이런 모든 개념을 살펴보고, 이런 개념들이 유용한 이유를 설명해준다. 단일 페이지 애플리케이션<sup>SPAs, Single Page Applications</sup>를 만드는 데 특히 도움이 될 것이다.

React는 애플리케이션의 “view”에만 집중한다. 서버 통신이나 코드 조직에는 전혀 관심을 두지 않는다. 이 책에서는 React를 이용해서 완전한 애플리케이션을 만들기 위한 모범 예제와 대체 도구도 설명한다.

## 이 책을 읽기 위해 알아야 할 것

---

이 책의 내용을 이해하기 위해서는 JavaScript와 HTML을 다뤄본 경험이 있어야 한다. 프레임워크의 종류와 상관없이 단일 페이지 애플리케이션을 다뤄본 경험이 있다



면 더욱 도움이 될 것이다. 물론 필수적인 것은 아니다.

### **소스 코드와 예제 애플리케이션**

---

이 책에서는 예제 애플리케이션으로 설문조사 생성기 Survey Builder를 인용한다. 전체 코드는 Github의 저장소 <https://github.com/backstopmedia/bleeding-edge-sample-app>에서 확인할 수 있다.

### **버전 이슈에 대한 역자주**

---

이 책은 React v.0.11.1~v.0.12를 기준으로 집필되었다. Github 저장소에 있는 샘플 예제 코드는 React v.0.11.1을 사용한다. 이 책을 번역하는 현재(2015. 9. 8) 가장 안정화된 최신 버전은 v.0.13.3으로, 버전이 업데이트하면서 인터페이스나 정책이 바뀐 부분이 있어 책에 나와 있는 내용을 최신 버전에 그대로 적용할 수 없는 경우가 있다. 독자의 혼란을 막기 위해 최신 버전에서 호환되지 않는 부분마다 역자주를 삽입했고, 부록에 릴리즈 노트를 수록했으니 참고하기 바란다.

chapter 1 React 소개 —— 019

- 1.1 배경 —— 019
- 1.2 개요 —— 021

chapter 2 JSX —— 027

- 2.1 JSX는 무엇인가? —— 028
- 2.2 JSX의 장점 —— 029
- 2.3 컴포넌트 조합 —— 032
- 2.4 JSX와 HTML의 차이점 —— 035
- 2.5 JSX를 사용하지 않는 경우의 React —— 043
- 2.6 JSX 공식 스펙 —— 046

chapter 3 컴포넌트 라이프사이클 —— 049

- 3.1 라이프사이클 메소드 —— 049
- 3.2 초기화 —— 050
- 3.3 실행시 —— 052
- 3.4 분해와 정리 —— 055
- 3.5 앤티 패턴: 상태에 계산값 사용 —— 055
- 3.6 정리 —— 057



## chapter 4 데이터 흐름 —— 059

4.1	Props	059
4.2	PropTypes	061
4.3	get defaultProps	062
4.4	State	063
4.5	state와 props에는 어떤 값을 저장해야 할까?	064
4.6	정리	065

## chapter 5 이벤트 처리 —— 067

5.1	이벤트 핸들러 연결하기	067
5.2	이벤트와 상태	069
5.3	상태에 따른 렌더링	070
5.4	상태 변경하기	072
5.5	이벤트 객체	074
5.6	정리	075

## chapter 6 컴포넌트 구성 —— 077

6.1	HTML 확장	077
6.2	예제	078
6.3	부모 컴포넌트와 자식 컴포넌트의 관계	084
6.4	정리	086

## chapter 7 믹스인 —— 089

- 7.1 믹스인은 무엇인가? —— 089
- 7.2 정리 —— 093

## chapter 8 DOM 조작 —— 095

- 8.1 React를 통한 DOM 노드 접근 —— 095
- 8.2 React 외의 라이브러리 포함하기 —— 097
- 8.3 부모 엘리먼트에 영향을 주는 플러그인 다루기 —— 100
- 8.4 정리 —— 102

## chapter 9 폼 —— 103

- 9.1 비제어 컴포넌트 —— 104
- 9.2 제어 컴포넌트 —— 107
- 9.3 폼 이벤트 —— 109
- 9.4 레이블 —— 110
- 9.5 textarea와 select —— 110
- 9.6 체크박스와 radio 버튼 —— 112
- 9.7 폼 엘리먼트 이름 —— 113
- 9.8 여러 개의 폼 엘리먼트에 change 핸들러 사용 —— 115
- 9.9 커스텀 폼 컴포넌트 —— 120
- 9.10 포커스 —— 124
- 9.11 사용성 —— 124
- 9.12 정리 —— 129



## chapter 10 애니메이션 —— 131

10.1	CSS 트랜지션 그룹	131
10.2	트랜지션 그룹 사용 시 주의점	134
10.3	인터벌 렌더링	134
10.4	정리	137

## chapter 11 성능 개선 —— 139

11.1	shouldComponentUpdate	139
11.2	Immutability Helpers 애드온	141
11.3	속도 저하 원인 찾기	143
11.4	Key	144
11.5	정리	146

## chapter 12 서버 사이드 렌더링 —— 147

12.1	렌더링 함수	148
12.2	서버 사이드 컴포넌트 라이프사이클	150
12.3	정리	157

## chapter 13 React 패밀리 —— 159

13.1	Jest	159
13.2	Immutable.Map	165
13.3	Flux	166
13.4	정리	167



## chapter 14 개발 도구 —— 169

14.1	빌드 도구	169
14.2	Browserify	170
14.3	Webpack	173
14.4	Webpack과 React	174
14.5	디버깅 도구	177
14.6	정리	179

## chapter 15 테스트 —— 181

15.1	시작하기	181
15.2	첫 번째 명세 : 렌더링	183
15.3	모의 컴포넌트	189
15.4	함수를 스파이 객체로 만들기	196
15.5	이벤트 시뮬레이션	205
15.6	finder 메소드로 컴포넌트 탐색하기	209
15.7	믹스인	212
15.8	<body>에 렌더링 하기	224
15.9	서버 사이드 테스트	229
15.10	브라우저 테스트 자동화	236
15.11	정리	243



## chapter 16 설계 패턴 —— 245

16.1 라우팅	246
16.2 Om(ClojureScript)	251
16.3 Flux	252
16.4 정리	261

## chapter 17 그밖의 사용법 —— 263

17.1 데스크톱	263
17.2 게임	265
17.3 HTML 이메일	271
17.4 차트	276
17.5 정리	278

## chapter 18 부록: 릴리스로그 —— 280

18.1 React v.0.11.2	280
18.2 React v.0.12 RC	281
18.3 React v.0.12	286
18.4 React v.0.12.2	290
18.5 React v.0.13 Beta 1	291
18.6 React v.0.13 RC	296
18.7 React v.0.13 RC2	298
18.8 React v.0.13	300
18.9 React v.0.13.1	303
18.10 React v.0.13.2	304
18.11 React v.0.13.3	305
18.12 React v.0.14 Beta 1	306
18.13 React v.0.14 RC	310



# React 소개

## 1.1 배경

웹 애플리케이션이 막 등장한 시절에는 클라이언트가 페이지 전체를 서버에 요청하는 방식으로만 웹 애플리케이션을 개발할 수 있었다. 이렇게 개발한 애플리케이션은 브라우저에서 일어나는 이벤트를 처리할 필요가 없었기 때문에 구조가 아주 단순했다.

PHP 같은 언어를 사용하면 이런 형태의 애플리케이션을 쉽게 만들 수 있었다. 가능한 PHP로 컴포넌트<sup>functional components</sup>를 만들면, 재사용성과 가독성이 높은 코드를 쉽게 작성할 수 있었고, PHP는 이런 단순함 덕분에 많은 인기를 얻었다.

하지만 이 방식으로는 사용자에게 멋진 경험을 줄 수 없었다. 사용자가 무언가를 할 때마다 매번 서버에 요청을 보내고 응답을 기다려야 했다. 심지어 서버로부터 응답이 오면 지금까지 페이지에서 사용자가 작업한 내용은 모두 날아가 버렸다.

더 나은 사용자 경험을 제공하기 위해서 사람들은 브라우저에서 애플리케이션을 렌더링하는 JavaScript 기반 라이브러리를 만들기 시작했다. 단순한 HTML 템플릿부터 애플리케이션 전체를 제어하는 시스템까지, 다양한 방법으로 DOM을 제어하는 라이브러리가 등장했다. 다양한 JavaScript 라이브러리를 이용하면서 애플리케이션은 점점 더 크고 복잡해졌다. 길게 헝클어진 실처럼 꼬여있는 이벤트로 무장한 애플리케이션의 동작을 설명하기란 쉽지 않다. 그래서 앞에서 이야기한 PHP의

경우에 비해, 요즘의 클라이언트 애플리케이션은 만들기가 매우 어려워졌다.

React는 Facebook이 사용하는 PHP프레임워크인 XHP를 대체하기 위해 시작되었다. PHP 프레임워크인 XHP는 새로운 요청이 들어올 때마다 전체 페이지를 렌더링한다. 이 작업을 클라이언트에서 처리하기 위해 React가 탄생했다.

React는 기본적으로 복잡한 상태 변화를 잘 관리할 수 있게 해주는 상태 시스템 state machine이다. React의 관심사는 오직 두 가지다.

## 1. DOM 업데이트

## 2. 이벤트 응답

React는 Ajax, 라우팅, 저장소, 데이터 구조에 관심이 없으며 MVC<sup>Model-View-Controller</sup> 프레임워크가 아니다. 굳이 MVC를 가지고 이야기하자면, MVC에서 V를 담당한다고 볼 수 있다. 가벼운 React는 다양한 시스템에 자유롭게 어울릴 수 있다. 실제로 몇몇 인기 있는 MVC 프레임워크가 View를 렌더링할 때 React를 사용한 적이 있다.

JavaScript로 DOM을 가져오고 갱신하는 작업은 느리므로 상태가 바뀔 때마다 페이지 전체를 렌더링하는 애플리케이션은 성능이 떨어질 수밖에 없다. React는 가상 DOM<sup>virtual DOM</sup>을 이용해서 DOM을 읽지 않고 갱신할 수 있는 아주 강력한 렌더링 시스템을 가지고 있다.

React의 핵심은 렌더링 함수다. 이 함수는 현재의 상태 값을 결과 페이지를 나타내는 가상 표현 객체<sup>virtual representation</sup>으로 변환한다. 마치 고성능 3D 게임 엔진 같다. 상태 변경을 감지한 React는 이 함수를 실행해서 새로운 가상 표현 객체를 만든 다음에 이것을 DOM에 전달해 새로운 상태를 반영한다.

언뜻 보면, JavaScript가 필요할 때 개별 요소를 갱신하는 것보다 느릴 것 같다. 하지만 React는 현재의 가상 표현 객체와 새로운 가상 표현 객체의 차이를 아주

효과적으로 비교하는 알고리즘을 가지고 있다. 이 알고리즘을 이용해서 DOM을 최소한으로 변경한다.

리플로우<sup>reflow</sup>나 불필요한 DOM 조작<sup>mutation</sup> 같이 흔한 성능 저해 요소를 최소화 함으로써 성능을 끌어올린다.

인터페이스가 커질수록, 하나의 상호 작용<sup>interaction</sup>이 다른 상호 작용을 연쇄적으로 부르는 경우가 많아진다. 이런 연쇄 호출을 적절히 처리하지 않으면 애플리케이션의 성능이 급격히 떨어진다. 심지어, 최종 상태에 도달할 때까지 같은 DOM 엘리먼트를 몇 번이고 다시 변경하는 경우도 있다.

React 가상 표현 객체는 단일 패스에서 최소한의 변경을 수행함으로써 이런 문제를 최소화한다. 이를 통해 애플리케이션의 유지 보수성도 높인다. 사용자 입력이나 외부의 변경에 의한 상태 변화가 있다는 사실을 React에게 알려주기만 하면 나머지는 React가 알아서 처리한다. 개발자가 프로세스를 세세히 관리할 필요가 없다.

React는 모든 이벤트를 하나의 이벤트 핸들러에 위임<sup>delegate</sup>함으로써 이벤트 핸들러가 여러 개일 때 발생할 수 있는 성능 저하의 위험을 줄인다.

이 책에서 사용하는 설문조사 생성기<sup>Survey Builder</sup> 예제는 Github 저장소(<https://github.com/backstopmedia/bleeding-edge-sample-app>)에서 자세히 볼 수 있다.

## 1.2 개요

이 책은 React를 이용한 최신 개발 기법을 크게 네 가지 주제로 나눠서 다룬다.

### 컴포넌트 생성 및 구성

1장부터 7장까지는 React 컴포넌트를 생성하고 구성하는 방법을 설명한다. 여기에서는 React 사용방법을 배운다.

## 1) React 소개

1장은 배경과 이 책의 전체 개요를 설명하고 React를 소개한다.

## 2) JSX와 기본 React 구성요소 사용하기

JSX JavaScript XML를 이용하면 JavaScript 코드 안에 XML 스타일의 문법 코드를 작성할 수 있다. JSX를 사용하는 방법과 이를 이용해서 기본적인 React 컴포넌트를 만드는 방법을 학습한다. React 컴포넌트를 개발할 때 반드시 JSX를 같이 사용해야 하는 것은 아니지만, 권장하고 싶은 방법이라는 생각에 이 책에 있는 대부분의 예제는 JSX를 사용해서 작성했다.

## 3) React 구성요소의 라이프 사이클

React는 렌더링 과정 중에 자주 컴포넌트를 생성하고 제거하며, 컴포넌트 라이프 사이클에 접근할 수 있는 다양한 함수를 제공한다. 이 라이프 사이클을 잘 이해하면 애플리케이션 메모리 누수를 방지할 수 있다.

## 4) React의 데이터 흐름

React가 컴포넌트 트리에 데이터를 어떻게 전달하는지, 어떤 데이터를 변경해도 안전한지 잘 알아둬야 한다. React는 속성<sup>props</sup>과 상태<sup>state</sup>를 아주 분명하게 구분한다. 이 장은 속성과 상태가 무엇인지 알아보고, 컴포넌트 개발 시에 이 둘을 제대로 사용하는 방법을 설명한다.

## 5) 이벤트 핸들링

React의 이벤트 처리는 선언적이다. 이벤트 처리는 동적 UI를 구성하는 데 중요하므로 완벽하게 익히는 게 좋다. 다행히 React를 이용하면 이벤트 처리를 아주 간단하게 할 수 있다.

## 6) 컴포넌트의 구성

React를 이용하면 특정 작업을 수행하는 작지만 정교한 컴포넌트를 만들 수 있

다. 이렇게 만든 컴포넌트를 구성해서 오케스트레이션 레이어<sup>orchestration layers<sup>01</sup></sup>를 만든다. 이 장은 한 컴포넌트가 다른 컴포넌트를 사용하는 방법을 설명한다.

## 7) React 믹스인

여러 React 컴포넌트가 사용하는 공통 기능을 공유하는 방법인 믹스인을 사용하면 컴포넌트를 더 쉽게 만들 수 있어서 관리하기 편하다.

## 고급 주제

기본을 배웠으니 이제 더 수준 높은 주제로 넘어간다. 8장부터 13장까지 React 개발 기술을 연마하여 더 훌륭한 React 컴포넌트를 만들어 본다.

## 8) React에서 DOM에 접근하기

기존 JavaScript 라이브러리를 사용하거나 컴포넌트를 더 깊게 제어하기 위해서 때로는 React 가상 DOM이 아닌 진짜 DOM을 이용해야 할 때가 있다. 이 장은 안전하게 DOM에 접근할 수 있는 React 컴포넌트의 라이프 사이클이 어디인지, DOM 제어를 언제 해제하여 메모리 누수를 막아야 하는지를 설명한다.

## 9) React로 폼 요소 다루기

HTML 폼 엘리먼트는 사용자 입력을 받는 대표적인 방법이다. HTML 폼 엘리먼트는 상태를 갖는다. React를 이용하면 놀라운 방법으로 폼의 상태를 React 컴포넌트에 전달할 수 있다.

## 10) 애니메이션

CSS를 이용하면 고성능 애니메이션을 만들 수 있다. React는 CSS 애니메이션 처리를 도와준다. 이 장은 React를 이용해서 CSS 애니메이션을 처리하는 방법을 설명한다.

---

**01** 역자주\_소프트웨어 개발에서 오케스트레이션 레이어(Orchestration Layer, OL)는 모델링한 데이터 요소와 기능을 구체화하는 추상 레이어를 말한다. 작게 나눈 React 컴포넌트를 실제 제품에 적용하기 위해서는 개별 컴포넌트가 서로 조화롭게 동작하여 하나의 콘텐츠를 표현할 수 있게 제어하는 단계가 필요하다. 이 책에서 말하는 오케스트레이션 레이어는 이 단계를 의미한다.

## 11) 성능 개선과 컴포넌트

React 가상 DOM은 뛰어난 성능을 보여주지만, 언제나 그렇듯이 개선할 부분이 있다. React는 변경이 없는 컴포넌트를 브라우저가 다시 렌더링 하는 것을 막음으로써 애플리케이션의 속도를 비약적으로 높인다.

## 12) 서버 사이드 렌더링

많은 애플리케이션이 SEO를 적용한다. React는 Node.js처럼 브라우저가 아닌 환경에서도 문자열로 렌더링할 수 있다. 서버 측 렌더링을 이용하면 애플리케이션의 시작 페이지 로딩 시간을 줄일 수 있다. 서버와 클라이언트 렌더링 방식을 함께 사용하는 것은 어려울 수 있다. 이 장에서는 두 가지 렌더링 방식을 함께 적용하는 전략을 설명하고, 서버 측 렌더링을 처리할 때 발생할 수 있는 복잡한 상황을 자세히 알아본다.

## 13) React 패밀리의 다른 JavaScript 라이브러리 사용하기

Facebook은 React 외에도, React와 함께 사용할 수 있는 여러 오픈 소스 개발 도구를 계속해서 공개하고 있다. 이 장을 학습하면 이런 라이브러리를 React 패밀리에 잘 적용하기 위한 통찰을 얻을 수 있다.

## React를 위한 도구 다루기

React는 뛰어난 개발, 테스트 도구를 제공한다. 이 도구를 사용하면 견고한 애플리케이션을 만들 수 있다. 14, 15장에서 개발자 도구와 테스트 방법을 알아본다.

## 14) 개발자 도구

React 애플리케이션의 규모가 커지면 배포를 위해 코드 패키징 과정을 자동화할 필요가 있으며 코드 디버깅이 점점 어려질 것이다. 이 장에서는 이런 고민을 덜어주는 React 애플리케이션 패키징 도구를 살펴본다. 그리고 더 쉬운 디버깅을 위해 구글 크롬 플러그인으로 React 컴포넌트를 시각화하는 방법도 알아본다.

## 15) React를 위한 테스트 코드 작성하기

애플리케이션의 규모가 점점 커지는 상황에서, 기존 코드에 버그를 추가하지 않으려면 테스트 코드를 작성하는 것이 좋다. 테스트 코드는 더 나은 모듈화 코드를 작성하는 데 도움이 된다. 이 장에서는 React 컴포넌트의 모든 부분을 테스트하는 방법을 알아본다.

## React 활용하기

마지막 장은 React를 활용하는 데 있어서 중요한 부분을 살펴보고, 미처 생각해보지 못한 다른 사용 사례를 설명한다.

## 16) Architectural patterns

React는 “MVC” 중에서 “V”만을 제공한다. 그래서 다른 프레임워크나 시스템에 매우 유연하게 적용할 수 있다. 이 장은 React를 이용해서 규모가 큰 애플리케이션을 설계하는 과정을 설명한다.

## 17) React의 다른 사용 사례

React는 웹 환경에 초점을 맞추고 있지만, 웹이 아니더라도 JavaScript를 지원하는 환경이라면 어디든 사용할 수 있다. 이 장에서는 전통적인 웹 환경이 아닌 곳에서 React를 사용하는 방법을 알아본다.



React는 템플릿이나 표현 로직 대신 컴포넌트 개념을 이용하여 관심사를 분리한다. 기본적으로 React는 마크업과 마크업 생성 코드를 함께 묶어둔다는 사실을 알아야 한다. 이렇게 하면 마크업 코드를 작성할 때 어색하고 무거운 템플릿 언어를 사용할 필요가 없으므로 JavaScript의 표현력을 극대화할 수 있다.

React는 선택적으로 사용할 수 있는, HTML과 유사한 스타일의 마크업 언어를 지원한다. 자세히 알아보기 전에 먼저 봄다면 좋을 게 있다. JavaScript에 마크업 코드를 작성하는 스타일을 좋아하지 않는다면 JSX가 정말 유용한지 잘 모르겠다면, 아래에 나열한 장점을 한 번 읽어보길 바란다.

- 익숙한 마크업을 이용해서 엘리먼트를 트리 구조로 만들 수 있다.
- 더 의미가 잘 드러나고, 더 이해하기 쉬운 마크업을 제공한다.
- 애플리케이션의 구조를 더 쉽게 시각화할 수 있다.
- React 엘리먼트 생성을 추상화한다.
- 마크업과 마크업을 생성하는 코드를 함께 둘 수 있다.
- 순수 JavaScript다.

이번 장에서 JSX의 장점과 사용법을 알아보고 HTML과 다르게 유의해야 할 점을 알아본다. 반드시 JSX를 사용해야 하는 것은 아니라는 사실을 명심하자. JSX를 사용하고 싶지 않다면 이 장의 끝에 있는 JSX 없이 React를 사용하는 방법을 설명하는 부분만 보고 넘어가도 좋다.

## 2.1 JSX는 무엇인가?<sup>01</sup>

JSX는 JavaScript XML의 약자다. React 컴포넌트에서 사용할 마크업을 작성하기 위한 문법체계로 XML과 유사하다. JSX를 사용하지 않아도 상관없지만, JSX를 사용하면 컴포넌트 가독성을 높일 수 있으므로 JSX를 사용할 것을 추천한다.

JSX를 사용하지 않고 헤더를 생성하는 함수를 호출하는 예를 보자.

```
// v0.11  
React.DOM.h1({className: 'question'}, 'Questions');  
  
// v0.12  
React.createElement('h1', {className: 'question'}, 'Questions');
```

JSX를 사용하면 이 코드를 훨씬 더 익숙하고 간결한 마크업으로 만들 수 있다.

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

JavaScript 코드 안에 마크업 코드를 작성하는 이전 방식과 비교해보면 JSX의 특징을 알 수 있다.

1. JSX는 변환되는 구문이다. 각 JSX 노드는 JavaScript 함수에 대응한다.
2. JSX는 런타임 라이브러리를 제공하지 않으며 필요로 하지도 않는다.
3. JSX는 JavaScript의 의미를 변경하거나 새로운 의미를 추가하지 않는다.  
함수를 호출할 뿐이다.

JSX가 HTML과 유사하다는 점이 React의 표현력을 배가한다. 이제 JSX의 이 점, 사용 목적, JSX와 HTML의 핵심 차이점을 알아보자.

<sup>01</sup> 역사주\_ React v.0.12부터는 JSX를 함수 호출로 변환하지 않는다. 대신 React.createElement에 인자를 전달하는 코드로 변환한다. 자세한 내용은 릴리즈 노트의 React v.0.12 RC에 있는 JSX Changes를 참고 하자.

## 2.2 JSX의 장점

다양한 템플릿 언어를 제쳐놓고 왜 굳이 JSX를 사용해야 할까? 결론부터 이야기하자면, JSX가 단순히 마크업을 JavaScript 함수에 매핑하기 때문이다.

JSX의 이점은 코드가 늘어나면서 컴포넌트가 점점 복잡해지는 상황일 때 두드러진다. JSX의 이점을 하나씩 살펴보자.

### 익숙함

개발 조직 안에 개발자만 있는 것은 아니다. HTML에 익숙한 UI/UX 디자이너, 테스트를 전담 QA 조직이 참여하여 개발 관련 업무를 수행한다. JSX를 프로젝트에 사용하면 개발자가 아닌 구성원들도 쉽게 코드를 읽고 업무에 기여할 수 있다. XML에 익숙한 사람이라면 쉽게 JSX를 받아들일 수 있다.

그리고 뒤에서 더 자세히 살펴보겠지만, React 컴포넌트는 DOM으로 표현할 수 있는 모든 것을 처리한다. JSX는 이를 아주 간단명료하게 표현하여 시각화한다.

### 의미론

JSX를 이용하면 JavaScript 코드를 마크업으로 만들어 의미를 더 잘 드러낼 수 있다. 컴포넌트 구조와 정보 흐름을 HTML과 유사한 문법을 이용하여 선언할 수 있고, 이렇게 작성한 코드는 나중에 JavaScript로 변환할 수 있다.

HTML5 태그 명은 물론이고 애플리케이션의 마크업에 사용한 커스텀 컴포넌트 까지 모두 JSX를 작성할 때 사용할 수 있다. 커스텀 컴포넌트를 정의하는 방법은 뒤에서 다루고, 우선 여기에서는 JSX로 JavaScript의 가독성을 높이는 방법을 알아보자.

좌측에 헤더를 표시하고 나머지 오른쪽을 가득 채우는 칸막이<sup>Divider</sup> 역할을 하는 엘리먼트를 생각해보자. HTML로 표현하면 이런 모양일 것이다.

---

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

---

이 마크업 코드를 `Divider`라는 React 컴포넌트로 포장했다. 이제 다른 HTML 엘리먼트처럼 사용할 수 있다. 코드의 의미를 더 이해하기 쉽다.

---

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

---

## 시각화

앞의 예제에서 우리는 JSX로 간단한 예제 코드를 더 명확하고 간결하게 만들었다. 수백 개의 컴포넌트가 복잡한 마크업 트리 형태로 얹혀있는 대형 프로젝트에서 이런 장점은 더욱 두드러진다

앞에서 언급했던 `Divider` 컴포넌트를 다시 보자. JavaScript만 사용했을 때보다 마크업의 의미가 더 분명하게 드러나 더 쉽게 의미를 이해할 수 있다.

우선 순수 JavaScript를 사용한 경우다.

---

```
// v0.11
render: function () {
  return React.DOM.div({className:"divider"},
    "Label Text",
    React.DOM.hr()
  );
}

// v0.12
render: function () {
  return React.createElement('div', {className:"divider"},
    "Label Text",
    React.createElement('hr')
  );
}
```

---

이번에는 JSX 마크업을 사용했다.<sup>02</sup>

---

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

---

JSX 마크업이 이해하기도 쉽고 디버깅하기도 편리하다는 점을 확인할 수 있다.

## 추상화

앞에서 React 0.11과 0.12로 코드를 작성하면서 서로 다른 JavaScript를 사용했다. 두 버전 모두 같은 JSX를 지원한다. JSX 트랜스파일러를 이용하면 마크업을 JavaScript로 변환하는 과정을 추상화할 수 있어 가능한 일이다. React 0.11로 작성한 코드는 추가 변경 작업 없이 0.12로 업데이트 할 수 있다.

추상화가 만병통치약은 아니지만, 프로젝트를 진행해나가면서 코드를 변경하는 일을 줄여준다는 점은 분명한 장점이다.

## 관심사 분리

React로 개발할 때는 마크업과 마크업을 생성하는 JavaScript 코드를 함께 묶어서 작성한다. 또한, 애플리케이션의 관심사나 개별 컴포넌트의 관심사를 분리하지 않는다. 대신 관심사 별로 컴포넌트를 만들어서 모든 관련 로직과 마크업을 하나의 공간에 넣고 캡슐화한다.

JSX를 이용하면 비즈니스 로직에서 마크업 코드를 분리할 수 있어 코드를 간결하게 만들 수 있다. 이렇게 하면 컴포넌트 트리 구조가 선명하게 드러나 개발자가 애플리케이션을 이해하기 쉽다.

---

<sup>02</sup> 역자주 \_ JSX 변환을 위해서 v.0.12 이전에는 @jsx 컴파일 지시문 주석이 필요했지만 v.0.12 이후 버전부터는 더 이상 @jsx 주석을 작성할 필요가 없다.

## 2.3 컴포넌트 조합

지금까지 JSX를 이용해 간결한 마크업 형태로 컴포넌트를 작성하는 방법을 살펴봤다. 이번에는 JSX로 만든 개별 컴포넌트를 하나로 조립하는 과정을 알아보자.

이 절에서는 다음의 내용을 다룬다.

- JSX를 사용하는 JavaScript 파일 작성
- 컴포넌트 조합하는 과정
- 컴포넌트 소유권과 부모/자식 관계

하나씩 차례대로 살펴보자.

### 커스텀 컴포넌트 선언하기

앞에서 설명한 Divider를 다시 보자. 출력하고 싶은 HTML은 다음과 같다.

---

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

---

이 HTML을 React 컴포넌트로 표현하기 위해서, 마크업을 반환하는 render 함수를 가지고 있는 React 컴포넌트를 만들었다.

---

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

---

현재로써는 이 컴포넌트를 이 형태로만 사용할 수 있다. h2 태그 안에 동적으로

텍스트를 넣을 수 있다면 더 유용하지 않을까?

## 동적인 값

JSX를 이용해 동적으로 값을 렌더링하고 싶다면 중괄호를 이용한다. {…} 중괄호는 JavaScript 콘텍스트에 신호를 보낸다. 중괄호 사이에 있는 값을 JavaScript로 평가한 결과를 마크업에 노드로 그린다.

간단한 텍스트나 숫자 같은 단순히 변수를 참조한다. 다음에 나와 있는 방법으로 h2 태그 안에 텍스트를 동적으로 넣을 수 있다.

---

```
var text = 'Questions';
<h2>{text}</h2>
// <h2>Questions</h2>
```

---

로직이 더 복잡하다면 함수를 이용한다. 중괄호를 이용해서 함수를 호출하여 결괏값을 렌더링할 수도 있다.

---

```
function dateToString(d) {
  return [
    d.getFullYear(),
    d.getMonth() + 1,
    d.getDate()
  ].join('-');
};

<h2>{dateToString(new Date())}</h2>
// <h2>2014-10-18</h2>
```

---

중괄호에 배열을 할당하면 React가 중괄호 안에 있는 배열의 각 항목을 자동으로 하나로 묶어준다.

---

```
var text = ['hello', 'world'];
<h2>{text}</h2>

// <h2>helloworld</h2>
```

---

종종 복잡한 값을 표현해야 할 때가 있다. 데이터 배열을 `<li>`로 표현해야하는 경우가 그렇다. 이 문제를 해결하기 위해서 자식 노드를 처리하는 방법을 알아보자.

### 자식 노드

`<h2>Questions</h2>`를 이용해서 헤더를 렌더링하면 “Questions”는 `h2` 엘리먼트의 자식 노드로 들어간다. 그렇다면 `JSX`를 이용해서 `Divider`를 다음과 같이 작성할 수 있을까?

---

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

---

할 수 있다. `React`는 여는 태그와 닫는 태그 사이에 있는 모든 자식 노드를 파악한다. 이때 컴포넌트가 가지고 있는 `this.props.children`이라는 특별한 배열에 모든 자식 노드를 저장한다. 앞 예제가 생성한 컴포넌트의 `this.props.children` 값은 `["Questions"]`다.

이 개념을 이용해서 하드 코딩한 “Questions” 텍스트를 `this.props.children`으로 바꿔보자. 이제 `React`는 사용자가 `<Divder>` 태그 사이에 할당한 값을 동적으로 렌더링할 수 있다.

---

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

---

HTML 엘리먼트처럼 활용할 수 있는 <Divider> 컴포넌트를 만들었다.

---

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

---

이 코드를 JSX 변환기<sup>JSX transformer</sup>를 JavaScript로 변환하면 다음과 같다.

---

```
var Divider = React.createClass({displayName: 'Divider',
  render: function () {
    return (
      React.createElement("div", {className: "divider"}, 
        React.createElement("h2", null, this.props.children),
        React.createElement("hr", null)
      )
    );
  }
});
```

---

결과값 역시 예상한 대로다.

---

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

---

## 2.4 JSX와 HTML의 차이점

JSX는 HTML과 유사하지만, HTML 문법을 완벽하게 따르지는 않는다. 이는 곧 장점이다. JSX 명세에는 다음과 같이 설명하고 있다.

이 명세서는 XML이나 HTML 명세를 따르지 않는다. ECMAScript 기능처럼 JSX를 설계했으며, XML과 유사한 것은 익숙하기 때문이다(<http://facebook.github.io/jsx/>에서 볼춰).

계속해서 JSX와 HTML 문법의 주요 차이점을 알아보자.

## 속성

HTML 엘리먼트의 속성은 다음에 보이는 것처럼 인라인<sup>inline</sup>으로 설정한다.

---

```
<div id="some-id" class="some-class-name">...</div>
```

---

JSX도 이 형식을 지원한다. 여기에 더해 JavaScript 변수를 이용해서 동적으로 속성값을 할당할 수 있다. JavaScript 변수를 속성에 할당할 때는 변수를 중괄호로 감싼다.

---

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

---

할당하려는 값을 얻는 과정이 복잡하다면 함수 호출의 결과를 속성값으로 지정할 수도 있다.

---

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

---

React는 컴포넌트를 렌더링 할 때마다 변수와 함수 호출을 평가하고, 평가 결과로 얻은 새로운 상태 값을 DOM에 전달해 반영한다.

**NOTE**

예제로 사용하는 설문조사 생성기의 전체 소스 코드는 Github 저장소에서 확인할 수 있다.

<https://github.com/backstopmedia/bleeding-edge-sample-app>

## 조건문

React 컴포넌트의 마크업과 마크업 생성 로직은 하나로 묶여있다. 이렇게 함으로써 반복문과 조건문 같은 JavaScript 로직을 이용해 마크업을 생성할 수 있다.

if/else 조건은 마크업으로 표현하기 어렵으므로 컴포넌트에 조건문을 추가하는 일 역시 어렵다. JSX에 바로 조건문을 작성할 수 있을까? 할 수 없다.

---

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

---

하지만 아예 불가능한 것은 아니며, 몇 가지 방법을 사용할 수 있다.

- 삼항 연산자<sup>ternary logic</sup> 사용
- 변수를 선언하고 속성에서 참조
- 함수에 로직을 떠넘기기
- 논리곱(&&) 연산자 사용

각각의 예를 간단히 살펴보자.

## 삼항 연산자 사용

가독성을 위해 공백을 추가했다.

---

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

---

텍스트에는 삼항 연산자가 적합하다. 하지만 JSX 안에 삼항 연산자를 사용한 코드는 거추장스럽고 읽기 어렵다. 이런 경우에는 다음 방법을 고려할 수 있다.

## 변수 참조

---

```
...
getIsComplete: function () {
  return this.state.isComplete ? 'is-complete' : '';
},
```

```
render: function () {
  var isComplete = this.getIsComplete();
  return <div className={isComplete}>...</div>;
}
...

```

---

## 함수 호출 사용

---

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

```

---

## 논리곱 연산자(&&) 사용

React는 Null이나 False 값에 특정한 값을 할당하지 않기 때문에 불린 Boolean 값을 사용해서 원하는 문자열을 출력할 수 있다. 다음 예제는 this.state.isComplete가 참이면 'is-complete'를 className으로 사용한다.

---

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

---

## 비DOM 속성

JSX에는 다음과 같은 특별한 속성을 사용할 수 있다.

- key
- ref
- dangerouslySetInnerHTML

자세히 살펴보자.

## 키

키<sup>key</sup>는 선택적으로 사용할 수 있는 고유 식별자다. 런타임 중에는 컴포넌트가 컴포넌트 트리의 위나 아래로 이동할 수 있다. 예를 들어 사용자가 검색하거나 목록에 아이템을 추가 또는 삭제하는 경우를 생각해보자. 이런 경우 컴포넌트를 제거하고 다시 만들 필요가 없다.

컴포넌트에 설정한 고유키를 이용해서 React 컴포넌트 재사용 여부를 결정할 수 있다. 키는 렌더링 과정에서 항상 같다. 이렇게 하면 렌더링 성능을 향상할 수 있다. DOM에 있는 두 아이템의 위치를 서로 변경할 때, 전체를 다시 렌더링하지 않고 키를 비교해서 위치만 변경할 수 있다.

## 참조

참조<sup>ref</sup>를 이용하면 부모 컴포넌트가 렌더링 함수 외부에서도 자식 컴포넌트를 참조할 수 있다.

ref 속성에 원하는 참조 명을 설정해서 참조를 정의한다.

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

컴포넌트 안에서 this.refs.myInput를 이용해서 이 참조 객체에 접근할 수 있다. 이 객체를 지원 인스턴스<sup>backing instance</sup>라고 부른다. 지원 인스턴스는 React 가 DOM을 생성할 때 이용할 뿐, 실제 DOM은 아니다. this.refs.myInput.

`getDOMNode()`<sup>03</sup>를 이용하면 실제 DOM을 가져올 수 있다.

더 자세한 내용은 6장에서 부모/자식 관계와 소유 관계를 비교할 때 다룬다.

## HTML을 문자열로 사용하기

HTML을 문자열로 작성해야 할 때가 있다. DOM 조작에 문자열을 이용하는 외부 라이브러리를 사용하는 경우다.

React는 `dangerouslySetInnerHTML` 속성을 이용해 HTML 문자열을 입력하는 방법을 제공해 호환성을 높였다. 하지만 추천하고 싶은 방법은 아니다. 이 속성을 사용하려면 다음처럼 `_html` 키가 있는 객체를 이용한다.

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

### NOTE DANGEROUSLYSETINNERHTML

이 속성은 곧 바뀔 수도 있다. 다음 이슈를 참고하기 바란다.

<https://github.com/facebook/react/issues/2134>

<https://github.com/facebook/react/pull/1515>

04

**03** 역사주\_ React v.0.13.x에는 최상위 API에 `React.findDOMNode(component)`가 추가되었다. 이 API는 `component.getDOMNode()`를 대신해서 사용할 수 있다. 같은 버전에 추가된 ES6 클래스를 이용해 작성한 컴포넌트는 `getDOMNode` API를 지원하지 않는다. 이때 이 API를 유용하게 사용할 수 있다.

**04** 역사주\_ React v.0.13.3 버전에서는 `deprecated` 되지 않는다.

## 이벤트

모든 브라우저가 같은 이벤트 명을 사용하며, 카멜표기법을 따른다. 예를 들어 change는 onChange, click은 onClick이다. JSX를 이용하면 메소드를 할당하는 것만으로 간단하게 이벤트 리스너를 추가할 수 있다.

---

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

---

React는 자동으로 모든 메소드를 메소드가 속한 컴포넌트 객체에 바인딩한다. 따라서 수동으로 콘텍스트를 바인딩할 필요가 없다.

---

```
...
handleClick: function (event) {...},
render: function () {
  // 안티패턴(anti-pattern) - React를 사용할 때는 컴포넌트 인스턴스에
  // 함수 콘텍스트를 개별적으로 바인딩할 필요가 없다.
  return <div onClick={this.handleClick.bind(this)}>...</div>
}
...
...
```

---

React의 이벤트 처리에 관한 더 자세한 이야기는 9장에서 다룬다.

## 주석

JSX는 JavaScript이기 때문에 JSX 마크업에 JavaScript 주석을 추가할 수 있다. 주석을 작성할 수 있는 위치는 아래 두 곳이다.

1. 엘리먼트의 자식 노드로 작성
2. 노드의 속성에 인라인으로 작성

## 자식 노드로 작성하기

자식 노드 주석은 중괄호로 감싸서 작성한다. 여러 줄로 작성할 수도 있다.

---

```
<div>
  /* 여러 줄에 걸쳐 작성한
     Input에 대한 주석 */
  <input name="email" placeholder="Email Address" />
</div>
```

---

## 속성에 인라인으로 작성

인라인 주석에는 두 가지 방법이 있다. 여러 줄 주석은 이렇게 작성한다.

---

```
<div>
  <input
    /*
      input에 대한 주석
    */
    name="email"
    placeholder="Email Address" />
</div>
```

---

한 줄 주석으로 작성할 수도 있다.

---

```
<div>
  <input
    name="email" // 한 줄 주석
    placeholder="Email Address" />
</div>
```

---

## 특수 속성

JSX 트랜스파일러는 JSX를 JavaScript 함수 호출 구문으로 변환한다. 따라서 변환 시 충돌 우려가 있는 `class`와 `for` 같은 몇 가지 키워드는 JSX에 사용할 수 없다.

<form> 요소의 <label>에 for 속성을 추가하려면 JSX에서는 htmlFor 속성을 이용한다.

---

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

---

CSS 클래스를 적용할 때는 class 속성 대신 className 속성을 사용한다. HTML에 익숙하다면 다소 이상하게 보일 수 있지만, JavaScript 측면에서 볼 때는 일관성이 있다. JavaScript에서는 elem.className으로 클래스 명에 접근할 수 있기 때문이다.

---

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

---

## 스타일

마지막으로 살펴볼 것은 인라인 스타일 속성이다. React는 모든 스타일 명에 카멜표기법을 적용했다. JavaScript의 DOM 스타일 속성과 일관성을 유지하기 위함이다. 인라인 스타일을 선언하려면 CSS 값을 카멜표기법으로 작성해 JavaScript 객체에 전달한다.

---

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

---

## 2.5 JSX를 사용하지 않는 경우의 React

결과적으로 JSX 마크업은 순수 JavaScript 코드로 변환된다. 따라서 반드시 JSX를 사용할 필요가 없지만, JSX를 사용하면 React 컴포넌트를 생성하는 단계의

복잡함을 감출 수 있다. JSX를 사용하지 않을 생각이라면, React 컴포넌트를 생성하는 과정을 알아야 한다. 이 과정은 총 3단계로 이루어진다.

1. 컴포넌트 클래스 정의하기
2. 컴포넌트 클래스의 인스턴스를 만드는 팩토리 생성하기
3. ReactElement 인스턴스를 생성하는 팩토리 사용하기

## React 엘리먼트 만들기

React는 HTML 엘리먼트를 생성해서 돌려주는 팩토리를 `React.DOM.*` 네임스페이스로 제공한다. 이 팩토리는 `React.createElement`를 축약한 것으로 첫 번째 인자로 전달한 엘리먼트를 생성한다. 다음 예제에 있는 두 코드는 모두 같은 결과를 돌려준다.

---

```
React.createElement('div');
React.DOM.div();
```

---

그렇지만 커스텀 컴포넌트를 만들려면 커스텀 컴포넌트 클래스에서 팩토리를 생성해야 한다.

`Divider` 클래스를 만들었던 과정을 기억하는가? 목적을 명확하게 하려고 `DividerClass`라고 이름을 바꿨다.

---

```
var DividerClass = React.createClass({displayName: 'Divider',
  render: function () {
    return (
      React.createElement("div", {className: "divider"}, 
        React.createElement("h2", null, this.props.children),
        React.createElement("hr", null)
      )
    );
  }
});
```

---

JSX를 사용하지 않고 DividerClass를 쓰기 위해서는 두 가지 방법을 생각해볼 수 있다.

1. React.createElement를 바로 호출한다.
2. React.DOM.\* 함수와 비슷한 팩토리를 만든다.

createElement를 호출해서 엘리먼트를 직접 생성할 수 있다.

---

```
var divider = React.createElement(DividerClass, null, 'Questions');
```

---

팩토리를 만들려면 우선 createFactory 함수를 사용한다.

---

```
var Divider = React.createFactory(DividerClass);
```

---

ReactElement를 만들 수 있는 팩토리 함수를 만들었다. 다음처럼 사용할 수 있다.

---

```
var divider = Divider(null, 'Questions');
```

---

## 축약

React.DOM.\* 네임스페이스를 이용하는 방식이 귀찮다면, 다음 예제에 보이는 R과 같이 좀 더 짧은 변수명을 이용하면 좋다.

---

```
var R = React.DOM;
var DividerClass = React.createClass({displayName: 'Divider',
  render: function () {
    return R.div({className: "divider"},
      R.h2(null, "Label Text"),
      R.hr()
    );
  }
});
```

---

각각의 팩토리를 최상위 변수에 할당해놓고 필요할 때 직접 참조할 수도 있다.

---

```
var div = React.DOM.div;
var hr = React.DOM.hr;
var h2 = React.DOM.h2;
var DividerClass = React.createClass({displayName: 'Divider',
  render: function () {
    return div({className: "divider"},  
      h2(null, "Label Text"),  
      hr()
    );
  }
});
```

---

## 참고사항

평소에 JavaScript와 마크업을 함께 사용하는 방식을 별로 좋아하지 않았다면, JSX가 제시하는 해결책에 매력을 느꼈기를 바란다. Facebook은 JSX에 대한 관심이 증가하자 명세를 만들었으며, 기술적으로 깊이 있는 설명도 함께 제공하고 있다. 또한, JSX를 다뤄볼 수 있는 몇 가지 도구도 제공하고 있다. 이러한 도구는 브라우저 환경에서 JSX 사용법을 알고 싶을 때 유용하게 사용할 수 있다.

## 2.6 JSX 공식 스펙

Facebook은 2014년 9월에 JSX 공식 명세를 공개했다. JSX를 만든 이유와 문법에 대한 기술적인 설명을 담고 있다. 자세한 내용은 <http://facebook.github.io/jsx/>에서 확인할 수 있다.

### 브라우저에서 JSX 다뤄보기

JSX를 테스트해 볼 수 있는 여러 가지 도구가 있다. React Getting Started 문서에는 JSFiddle을 이용해서 JSX를 작성하는 방법이 나와 있다.

- <http://facebook.github.io/react/docs/getting-started.html>

그리고 React가 제공하는 JSX 컴파일러 서비스를 이용하면 브라우저 상에서 JSX를 JavaScript로 변환할 수 있다.

- <http://facebook.github.io/react/jsx-compiler.html>



# 컴포넌트 라이프사이클

컴포넌트의 라이프사이클을 거치면서 `props`나 `state` 값이 바뀌면 DOM 표현 객체도 바뀐다. 2장에서 살펴본 것처럼 컴포넌트는 상태 시스템이다. 주어진 입력 값이 같다면 항상 같은 결과를 반환한다. React는 라이프사이클 후킹 hook 메소드를 제공한다. 이 메소드를 이용하면 컴포넌트의 라이프사이클에 따라 원하는 시점에, 원하는 작업을 할 수 있다.

## 3.1 라이프사이클 메소드

React 컴포넌트는 최소한의 라이프사이클 API를 제공하기 때문에 개발자가 쉽게 사용할 수 있다. 컴포넌트 안에서 라이프사이클 메소드를 호출하는 방법을 살펴보자.

### 초기화

React는 인스턴스를 생성할 때 라이프사이클 메소드를 호출한다. 처음 컴포넌트를 생성하면 다음과 같은 순서로 메소드를 호출한다.

- `get defaultProps`
- `getInitialState`
- `componentWillMount`
- `render`
- `componentDidMount`

다음 호출부터는 다음 순서대로 메소드를 호출하며 `get defaultProps`를 호출하지 않는다.

- `getInitialState`
- `componentWillMount`
- `render`
- `componentDidMount`

## 실행

애플리케이션의 상태를 변경하면 컴포넌트가 영향을 받는다. 이 경우 React는 메소드를 다음 순서로 호출한다.

- `componentWillReceiveProps01`
- `shouldComponentUpdate`
- `componentWillUpdate`
- `render`
- `componentDidUpdate`

## 해체와 정리

마지막으로 컴포넌트 사용이 끝나면 `componentWillUnmount`를 호출한다. `componentWillUnmount`는 인스턴스를 종료하고 정리한다. 초기화, 실행, 해체와 정리 과정을 차례로 하나씩 살펴본다.

## 3.2 초기화

컴포넌트가 처음 만들어지면, 컴포넌트를 사용하는 데 필요한 준비나 설정을 할 수 있는 다양한 메소드를 사용할 수 있다. 이 메소드는 각자 특정 역할을 가지고 있다.

---

01 역자주\_ `componentWillReceiveProps`는 `props`를 변경했을 때만 호출한다.

## **getDefaultProps**

컴포넌트 클래스는 이 메소드를 인스턴스를 생성하는 시점에 딱 한 번 호출한다. 새로운 인스턴스를 생성할 때 부모 컴포넌트가 인스턴스의 기본값을 지정하지 않았다면, 이 메소드를 호출한 결과로 얻은 객체를 인스턴스 기본값으로 사용할 수 있다. 이 객체가 참조하는 값이 객체나 배열처럼 원시 값이 아니어도 복사 또는 복제되는 것이 아니며, 하나의 값을 모든 인스턴스가 공유한다는 사실을 명심한다.

## **getInitialState**

컴포넌트 클래스는 컴포넌트 인스턴스를 생성할 때마다 이 메소드를 호출한다. 개별 인스턴스의 상태를 초기화할 수 있다. 인스턴스를 생성할 때마다 호출하는 메소드라는 점이 `getDefaultProps`와 다르다. 이 시점부터 `this.props`에 접근할 수 있다.

## **componentWillMount**

최초 렌더링 직전에 호출하는 메소드다. React가 `render` 메소드를 호출하기 전에 개발자가 컴포넌트의 상태에 영향을 줄 수 있는 마지막 단계다.

## **render**

여기에서 컴포넌트를 나타내는 가상 DOM을 만든다. `render` 메소드는 필수로 작성해야 하는 메소드이며, 이 메소드를 작성할 때는 몇 가지 규칙을 지켜야 한다.

- 접근할 수 있는 데이터는 `this.props`와 `this.state`뿐이다.
- `null`, `false`는 물론, 어떤 React 컴포넌트도 반환할 수 있다.
- 단일 최상위 컴포넌트만 반환할 수 있다. 엘리먼트 배열은 반환할 수 없다.
- 컴포넌트의 상태를 변경하거나 DOM을 수정할 수 없다.

`render` 메소드가 반환하는 것은 실제 DOM이 아닌 가상 표현 객체다. React는 여기서 만든 가상 표현 객체를 실제 DOM과 비교하여 변경할 부분이 있는지 확인한다.

## componentDidMount

render 메소드가 문제 없이 실행되어 실제 DOM이 렌더링 되면 componentDidMount 후킹 메소드 안에서 this.getDOMNode()<sup>02</sup>를 실제 DOM에 접근할 수 있다. 이 후킹 메소드는 렌더링 결과물의 높이 값을 확인하거나, 타이머를 이용해 조작하고 싶을 때, 또는 jQuery 플러그인을 적용하고 싶은 경우에 사용할 수 있다. 예를 들어 jQuery 자동완성 플러그인을 React로 렌더링한 input 엘리먼트에 적용한다고 생각해보자. 다음 예제처럼 플러그인을 연결<sup>attach</sup>할 수 있을 것이다.

---

```
// A list of strings to autocomplete
var datasource = [...];
var MyComponent = React.createClass({
  render: function() {
    return <input... />;
  },
  componentDidMount: function() {
    $(this.getDOMNode()).autocomplete({
      sources: datasource
    });
  }
});
```

---

서버에서 컴포넌트를 렌더링할 때는 React가 componentDidMount 메소드를 호출하지 않는다는 점을 주의한다.

## 3.3 실행시

이제 컴포넌트를 화면에서 볼 수 있으며, 사용자는 컴포넌트를 사용할 수 있다. 보통 사용자는 클릭<sup>click</sup>, 텁<sup>tap</sup>, 키 이벤트<sup>key event</sup>를 발생시켜 컴포넌트와 상호작용한

---

02 역자주\_ React v.0.13 은 같은 역할을 하는 새로운 최상위 API인 React.findDOMNode(component)를 지원한다.

다. 이때 발생한 이벤트가 이벤트 핸들러를 호출하여 컴포넌트나 애플리케이션의 상태를 변경하면, 컴포넌트 트리에 새로운 상태 변경 흐름이 발생한다. 이 흐름에 따라 호출되는 후킹 메소드를 이용해 변화를 제어할 수 있다.

### componentWillReceiveProps

컴포넌트의 `props`는 부모 컴포넌트에서 언제든 변경할 수 있다. 컴포넌트의 `props`가 바뀌면 `componentWillReceiveProps` 메소드가 호출되며, 이때 새로운 `props`나 `state` 객체를 변경할 수 있다. 설문조사 애플리케이션을 예로 들어보자. 사용자가 토글할 수 있는 radio 버튼인 `AnswerRadioInput`이 있다. 부모 컴포넌트는 버튼의 체크 상태를 나타내는 불린값<sup>Boolean</sup>을 조작한다. 이 경우 다음 예제처럼 부모가 전달한 `props` 객체를 참고하여 내부 `state`를 변경할 수 있다.

```
componentWillReceiveProps: function(nextProps) {
  if (nextProps.checked !== undefined) {
    this.setState({
      checked: nextProps.checked
    });
  }
}
```

**NOTE**

설문조사 애플리케이션의 전체 소스 코드는 Github에서 확인할 수 있다.

<https://github.com/backstopmedia/bleeding-edge-sample-app>

### shouldComponentUpdate

React는 이미 빠르지만, 컴포넌트를 렌더링할 때 `shouldComponentUpdate`를 이용해서 최적화하면 더 빠르게 만들 수 있다.

변경한 `props`나 `state`가 컴포넌트 자신이나 자식 컴포넌트를 렌더링할 필요가

없다면 `false`를 반환한다. `false`를 반환하면 React가 `render` 메소드를 호출하지 않는다.

이 메소드는 렌더링 초기화 시 또는 `forceUpdate`를 사용한 후에는 호출되지 않는다. `false`를 반환해서 React가 `render` 메소드 호출을 하지 않도록 할 수 있다.

이 메소드는 최초 렌더링 시와 `forceUpdate`를 사용 이후에는 호출되지 않는다.

대부분의 경우 개발할 때 이 메소드를 사용할 일이 없다. 이 메소드를 성급하게 사용했다간 미묘한 버그를 만들 수도 있다. 최적화를 시도하기 전에 병목구간이 어디인지 잘 살펴보는 것이 좋다.

인스턴스의 `state`를 변경 불가능한 immutable 값으로 관리하고, `render` 메소드에서 `props`와 `state`를 읽기 전용으로 사용하고 있다면, `shouldComponentUpdate`를 오버라이드하여 새로운 `props`와 `state`를 기준 값과 비교할 수 있다.

성능 개선을 위해 React 애드온이 제공하는 `PureRenderMixin`을 사용할 수도 있다. 항상 같은 `props`와 `state`를 가지는 DOM을 렌더링하는 순수한 컴포넌트라면, 이 믹스인은 자동으로 `shouldComponentUpdate`를 이용해서 `state`와 `props`를 얇게 비교한다. 비교 결과 같은 값이라면 `false`를 반환한다.

### `componentWillUpdate`

이 메소드는 `componentWillMount`과 비슷하다. 새로운 `props`와 `state`가 전달되어 React가 컴포넌트를 다시 렌더링하기 직전에 이 메소드를 호출한다. 이 메소드 안에서 `props`나 `state` 값은 변경할 수 없다. 런타임에 컴포넌트 상태를 변경하고 싶다면 `componentWillReceiveProps` 메소드를 사용해야 한다.

### `componentDidUpdate`

이 메소드는 `componentDidMount`과 비슷하다. 이 메소드를 이용해 렌더링이 끝난 DOM을 변경할 수 있다.

## 3.4 분해와 정리

컴포넌트의 수명이 다하면 DOM에서 분리해 제거해야 한다. React는 이 작업을 위한 후킹 메소드를 제공하며, 이 메소드를 이용해서 컴포넌트를 분해하고 정리할 수 있다.

### componentWillUnmount

React가 마지막으로 컴포넌트를 컴포넌트 계층 구조에서 제거하는 단계다. 이 메소드는 컴포넌트가 제거되기 직전에 호출된다. 개발자는 이 시점에 컴포넌트를 정리할 수 있다. componentDidMount에서 타이머를 만들거나 이벤트 리스너를 추가하는 등의 작업을 했다면, 여기에서 해제해야 한다.

## 3.5 안티 패턴: 상태에 계산값 사용

getInitialState 메소드에서 `this.props` 값으로 `state`를 만드는 방식은 안티 패턴이다. React의 철학은 신뢰할 수 있는 단일 출처<sup>single source of truth</sup>를 중요하게 생각한다. 단일 출처를 더 분명하게 만든다는 점은 React가 가지고 있는 중요한 강점이다.

`props`에 가져온 값을 계산한 결과를 `state`에 저장하는 것은 안티 패턴이다. 예를 들어 날짜 객체를 문자열로 변환하거나, 컴포넌트를 렌더링하기 전에 문자열을 모두 대문자로 변경하는 경우를 생각해보자. 이것은 컴포넌트의 상태가 아니다. 단순히 렌더링을 위한 값 변환일 뿐이다.

`state` 값이, 기반으로 하고 있는 `props` 값과 동기화되어 있는지 `render` 함수 안의 코드를 보고 알 수 없다면, 안티 패턴을 사용하고 있음을 의심해 볼 수 있다.

---

```
// 안티 패턴: 계산값은 상태로 저장될 수 없다.  
get defaultProps: function() {
```

```
        return {
          date: new Date()
        };
      },
      getInitialState: function() {
        return {
          day: this.props.date.getDay()
        }
      },
      render: function() {
        return <div> Day: {this.state.day}</div>;
      }
    }
```

---

이보다는 렌더링 시점에 값을 계산하는 것이 좋다. 이렇게 하면 계산 값이, 값의 출처인 props와 동기화되었음을 보장할 수 있다.

```
// 값을 렌더링 시점에 계산하는 것이 바람직하다.
get defaultProps: function() {
  return {
    date: new Date()
  };
},
render: function() {
  var day = this.props.date.getDay();
  return <div> Day: {day} </div>;
}
```

---

목표가 동기화가 아니라 단순히 상태를 초기화하는 것이라면, `getInitialState` 내에서 `state`를 사용하는 것이 적절할 수 있다. 이때는 의도를 명확히 해야 한다. 속성에 `initial` 접두사를 넣어보자.

```
get defaultProps: function() {
  return {
    initialValue: 'some-default-value'
  };
},
```

```
getInitialState: function() {
  return {
    value: this.props.initialValue
  };
},
render: function() {
  return <div> {this.props.value} </div>
}
}
```

---

### 3.6 정리

React의 라이프사이클 메소드는 컴포넌트 사용단계에 맞게 잘 설계된 흐름 방법을 제공한다. 상태 시스템으로서 각 컴포넌트는 자신의 생명주기에 따라 안정적이고 예측할 수 있는 마크업을 출력할 수 있게 만들어졌다.

모든 컴포넌트는 다른 컴포넌트와 관계를 맺는다. 부모 컴포넌트가 자식 컴포넌트에 `props`를 전달하고 자식 컴포넌트는 자신의 자식 컴포넌트를 렌더링한다. 이 경우 애플리케이션의 데이터 흐름을 어떻게 관리할지 고민해야만 한다. 자식 컴포넌트는 서로를 어느 정도 알아야 할까? 애플리케이션의 상태는 누가 관리하지? 다음 장에서 데이터 흐름을 주제로 이야기하면서 이 문제의 답을 함께 찾아보자.



# 데이터 흐름

React는 데이터가 부모 컴포넌트에서 자식 컴포넌트로 흐르는 단방향 데이터 흐름을 지향한다. 이렇게 함으로써 컴포넌트를 단순하고 예측할 수 있게 만들었다. 부모는 자식에게 props 값을 전달하고 렌더링한다. 최상위 컴포넌트의 속성이 바뀌면 React는 변경 사실을 컴포넌트 트리에 전달하고, 해당 속성을 사용한 모든 컴포넌트를 다시 렌더링한다.

또한, 컴포넌트는 내부 상태 값을 가질 수 있으며, 이 값은 컴포넌트 내부에서만 수정할 수 있다. React 컴포넌트는 본질적으로 단순하다. props와 state, 두 값을 이용해서 가상 표현 객체를 생성하는 함수라고 볼 수 있다.

이 장에서는 다음의 내용을 살펴본다.

- props는 무엇인가?
- state는 무엇인가?
- props와 state는 언제 사용하는가?

## 4.1 Props

props는 “properties”의 줄임말로, 사용자가 컴포넌트에 전달해 보관하기 원하는 데이터를 의미한다.

이 같은 컴포넌트를 초기화 할 때 다음과 같이 설정할 수 있다.

---

```
var surveys = [{ title: 'Superheroes' }];
<ListSurveys surveys={surveys}/>
```

---

또는 컴포넌트 인스턴스의 `setProps` 메소드<sup>01</sup>를 이용할 수도 있다. 하지만 사용 할 일은 별로 없다.

---

```
var surveys = [{ title: 'Superheroes' }];
var listSurveys = React.render(
  <ListSurveys/>,
  document.querySelector('body')
);
listSurveys.setProps({ surveys: surveys });
```

---

`setProps` 메소드는 자식 컴포넌트에서 사용하거나 앞의 예제처럼 컴포넌트 트리의 밖에서 사용해야 한다. 컴포넌트 내부에서 `this.setProps`를 호출하거나, `this.props` 값을 직접 수정하지 않도록 주의하자. 값을 변경하고 싶다면 `props`가 아닌 `state`를 사용하는 게 좋다. 이 내용에 대해서는 이 장의 뒷부분에서 더 자세히 설명한다.

**NOTE**

앞의 예제로 쓰인 설문조사 애플리케이션의 전체 코드는 Github 저장소에서 볼 수 있다.

<https://github.com/backstopmedia/bleeding-edge-sample-app>

`this.props`에 접근은 할 수 있지만, 이 값을 변경해서는 안 된다. 이 말은 컴포넌트가 자신의 `props` 값을 변경해서는 안 된다는 의미다.

JSX를 사용한다면 문자열을 이용해 컴포넌트의 `props`를 설정할 수 있다.

---

```
<a href='/surveys/add'>Add survey</a>
```

---

<sup>01</sup> 역사주\_ React v.0.13 버전부터는 ES6 클래스 기반 컴포넌트가 `setProps`를 제공하지 않는다.

{ } 문법을 사용할 수도 있다. 이 방식을 이용하면 JSX 안에 JavaScript 코드를 삽입할 수 있으며, 모든 종류의 변수를 전달할 수 있다.

---

```
<a href={'/surveys/' + survey.id}>{survey.title}</a>
```

---

JSX 스프레드 문법으로 JavaScript 객체를 props 값으로 할당할 수 있다.

---

```
var ListSurveys = React.createClass({
  render: function() {
    var props = {
      one: 'foo',
      two: 'bar'
    };

    return <SurveyTable {...props} />;      // 스프레드
  }
})
```

props는 이벤트 핸들러로도 유용하게 사용할 수 있다.

```
var SaveButton = React.createClass({
  render: function() {
    return ( <a className = 'button save' onClick = {this.handleClick} >
Save </a> );
  },
  handleClick: function () {
    // ...
  }
});
```

---

앞의 예제에서는 앵커 태그의 onClick 속성에 handleClick 메소드를 전달하고 있다. 사용자가 앵커를 클릭하면 컴포넌트는 handleClick 메소드를 호출한다.

## 4.2 PropTypes

컴포넌트에 설정 객체를 정의해서 props의 유효성을 검사할 수 있다.

---

```
var SurveyTableRow = React.createClass({
  propTypes: {
    survey: React.PropTypes.shape({
      id: React.PropTypes.number.isRequired
    }).isRequired,
    onClick: React.PropTypes.func
  },
  // ...
});
```

---

React는 컴포넌트를 초기화하면서 인스턴스의 props가 propTypes에 설정한 조건에 충족하는지 확인한다. props가 이 조건에 어긋나면 console.warn에 로그를 출력한다.

.isRequired는 필수 속성에만 사용한다.

propTypes를 반드시 사용할 필요는 없지만, 잘 사용하면 컴포넌트의 API를 설명하는 데 도움이 된다.

### 4.3 get defaultProps

컴포넌트에 defaultProps 메소드를 작성하면 기본 props 값을 제공할 수 있다. 이 방식은 컴포넌트를 생성할 때 필수로 설정해야 하는 속성 값이 아닌 경우에만 사용해야 한다.

---

```
var SurveyTable = React.createClass({
  getDefaultProps: function () {
    return {
      surveys: []
    };
  }
  // ...
});
```

---

`get defaultProps` 메소드는 컴포넌트 초기화 중에는 호출되지 않으며, React가 `React.createClass` 메소드를 호출해서 컴포넌트 클래스를 메모리에 적재하는 시점에 호출된다. 이는 `get defaultProps` 메소드를 이용해서 어떤 인스턴스의 특정 데이터에 접근할 수 없다는 것을 의미한다.

## 4.4 State

React는 컴포넌트의 상태를 저장할 수 있다. React 컴포넌트의 `state`는 컴포넌트 내부에 존재한다는 점에서 `props`와 다르다. `state`는 엘리먼트가 사용자에게 보여지는 상태를 결정할 때 유용하다.

```
var CountryDropdown = React.createClass({
  getInitialState: function () {
    return {
      showOptions: false
    };
  },
  render: function () {
    var options;

    if (this.state.showOptions) {
      options = <countryoptions>/countryoptions>;
    }
    return (
      <div className="dropdown"> onClick={this.handleClick};
        <label> Choose a country </label>
      </div>
    );
  },
  handleClick: function () {
    this.setState({ showOptions: true });
  }
});
```

앞의 예제에서 드롭다운의 옵션을 노출할지 결정하는 데 state의 프로퍼티를 사용했다.

state는 `setState` 메소드로 값을 변경하거나, 위의 경우처럼 `getInitialState` 메소드를 이용해 기본값을 설정할 수 있다. React는 사용자가 `setState` 메소드를 호출할 때마다 `render` 메소드를 호출한다. 이때 메소드 호출해서 얻은 결괏값에 변경이 있으면, 가상 표현 객체와 DOM을 갱신하여 사용자에게 변경사항을 보여준다.

따라서 `this.state`에 직접 접근하는 것보다, `this.setState` 메소드를 이용하는 것이 좋다.

state는 컴포넌트를 더욱 복잡하게 만든다. state를 특정 컴포넌트 안에 고립시키면 애플리케이션을 더 쉽게 디버깅할 수 있다.

## 4.5 state와 props에는 어떤 값을 저장해야 할까?

계산한 값이나 컴포넌트는 state에 저장하지 않는다. 대신 컴포넌트가 기능하는 데 직접 필요한 단순 데이터를 저장한다. 앞에서 언급했던 체크 상태 값이 한 예다.

체크 상태 값은 체크 박스를 체크하거나 해제하는 데 필수다. 드롭다운 옵션이나 input 폼의 불린 값을 보여주는 데에도 state를 사용할 수 있다.

props의 데이터를 state에 복사하도록 주의한다. props에 가능한 신뢰할 수 있는 단일 출처 원칙을 적용하는 것이 좋다.<sup>02</sup>

---

02 역사주\_ React는 `getInitialState`에 prop로 값을 전달하는 것을 앤티패턴(Anti-Pattern)으로 본다 props로 전달한 값에 대한 조작은 `render` 메소드에서 처리하는 것을 권장한다(<https://facebook.github.io/react/tips/props-in-getInitialState-as-anti-pattern.html>).

## 4.6 정리

이 장에서 학습한 내용을 요약하면 다음과 같다.

1. props를 이용해서 데이터와 설정을 컴포넌트 트리에 전달할 수 있다.
2. this.props를 이용해서 props 값을 직접 변경하거나 컴포넌트 내부에서 this.setProps를 호출하지 않아야 하며, props는 변경할 수 없는 값으로 간주하는 것이 좋다.
3. props를 이벤트 핸들러로 사용하여 자식 컴포넌트와 상호작용할 수 있다.
4. 드롭다운 옵션의 노출 여부와 같이 state에는 컴포넌트의 단순한 표현 상태를 저장한다.
5. this.state를 직접 변경해서는 안 되며, this.setState를 사용한다.

다음 장에서는 앞에서 잠깐 살펴봤던 이벤트 처리를 보다 자세하게 알아본다.



# 이벤트 처리

사용자 인터페이스에서 표현 로직을 처리하는 코드는 전체 코드 중 절반 정도다. 나머지는 사용자 입력을 처리하고 응답하는 로직으로 JavaScript로 사용자가 입력한 이벤트를 처리하는 것을 의미한다.

React는 이벤트를 처리하기 위해 컴포넌트에 이벤트 핸들러를 등록한다. 그리고 이벤트가 발생하면 컴포넌트의 내부 상태를 변경한다. 컴포넌트의 내부 상태가 바뀌면 컴포넌트를 다시 렌더링한다. 즉, UI를 조작해서 이벤트가 발생하면 render 메소드에서 컴포넌트의 내부 상태를 확인해야 한다.

보통 발생한 이벤트의 종류에 따라 상태를 변경한다. 때로는 상태를 변경하는 방식을 결정하기 위해서 이벤트가 가지고 있는 추가적인 정보를 활용할 수도 있다. 이 경우는 핸들러는 인자로 전달받은 이벤트 객체가 가지고 있는 추가적인 정보를 이용해 컴포넌트의 내부 상태를 변경한다.

앞장에서 설명한 기술과 React의 효율적인 렌더링 방식을 이용하면 사용자의 입력에 응답하고 UI를 변경하는 작업을 더욱 쉽게 처리할 수 있다.

## 5.1 이벤트 핸들러 연결하기

기본적으로 React가 처리하는 이벤트는 JavaScript와 같다. 클릭은 MouseEvent, form 엘리먼트 변경은 Change 이벤트를 이용하는 식이다. 이러한 이벤트는 JavaScript에서 사용하는 것과 같은 이름을 가지고 있고, 발생 조건도 같다.

React에서 이벤트 핸들러를 등록하는 문법은 HTML 문법과 거의 유사하다. 예를 들어 예제로 사용 중인 설문조사 생성기에서 다음 코드는 onClick 핸들러를 Save 버튼에 등록한다.

```
<button className="btn btn-save" onClick={this.handleSaveClicked}>Save</button>
```

사용자가 버튼을 클릭하면 React는 버튼의 handleSaveClicked 메소드를 실행한다. 이 메소드는 저장 작업을 처리할 것이다.

**NOTE**

예제로 사용하는 설문조사 생성기의 전체 소스 코드는 Github 저장소에서 확인할 수 있다.

<https://github.com/backstopmedia/bleeding-edge-sample-app>

HTML을 코딩할 때 이런 방식으로 onclick 이벤트를 등록하는 것을 권장하지 않지만, React 컴포넌트의 이벤트 핸들러는 HTML의 onclick과 다르다. React는 단순히 이벤트 핸들러를 지정하기 위해 이 문법을 사용할 뿐, 실제 내부에서는 효율적인 방법으로 이벤트 핸들러를 관리한다.

JSX를 사용하지 않는 경우, 다음 예제처럼 설정 객체를 이용해 이벤트 핸들러를 지정할 수 있다.

```
React.DOM.button({className: "btn btn-save", onClick: this.handleSaveClicked},  
"Save");
```

React는 여러 종류의 이벤트 처리를 지원한다. 지원하는 이벤트 목록은 React 공식 문서의 Event System<sup>01</sup>에 나와 있다.

01 역자주\_ <https://facebook.github.io/react/docs/events.html>

대부분의 경우 추가로 해야 할 작업이 없다. 다만 터치 이벤트는 다음과 같이 활성화해야 한다.

---

```
React.initializeTouchEvents(true);
```

---

## 5.2 이벤트와 상태

사용자 입력에 따라 상태가 바뀌는 컴포넌트를 만들고 싶을 때가 있다. 설문조사 편집기를 예로 들자면, 질문 종류 메뉴에서 질문 항목을 드래그할 수 있게 만들려는 경우다.

먼저 render 함수에서 시작하는데, HTML5 드래그 앤 드롭<sup>Drag and Drop API</sup>를 이용해서 이벤트를 등록한다.

---

```
var SurveyEditor = React.createClass({
  render: function () {
    return (
      <div className='survey-editor'>
        <div className='row'>
          <aside className='sidebar col-md-3'>
            <h2>Modules</h2>
            <DraggableQuestions />
          </aside>
          <div className='survey-canvas col-md-9'>
            <div
              className={'drop-zone well well-drop-zone'}
              onDragOver={this.handleDragOver}
              onDragEnter={this.handleDragEnter}
              onDragLeave={this.handleDragLeave}
              onDrop={this.handleDrop}
            >
              Drag and drop a module from the left
            </div>
          </div>
        </div>
      </div>
    );
  }
});
```

```
        </div>
    </div>
);
});
```

---

DraggableQuestions 컴포넌트가 질문 종류 목록을 렌더링한다. 핸들러 메소드가 드래그 앤 드롭 동작을 담당한다.

### 5.3 상태에 따른 렌더링

핸들러 메소드는 사용자가 추가한 질문을 열거해야 한다. 이를 위해 모든 React 컴포넌트가 가진 내부의 state 객체를 이용한다. state 객체의 기본값은 null 이지만, getInitialState() 메소드를 이용해서 state 객체에 의미 있는 값을 부여하여 초기화할 수 있다.

---

```
getInitialState: function () {
  return {
    dropZoneEntered: false,
    title: '',
    introduction: '',
    questions: []
  };
}
```

---

이 메소드는 상태의 기본값을 만든다. 비어있는 title과 questions 배열, dropZoneEntered의 false 값은 사용자가 드롭 영역에 드래그한 것이 없다는 것을 의미한다.

사용자에게 전체 질문 항목을 보여주기 위해서 render 메소드에서 this.state 를 읽어온다.

---

```
render: function () {
    var questions = this.state.questions;
    var dropZoneEntered = '';
    if (this.state.dropZoneEntered) {
        dropZoneEntered = 'drag-enter';
    }

    return (
        <div className='survey-editor'>
            <div className='row'>
                <aside className='sidebar col-md-3'>
                    <h2>Modules</h2>
                    <DraggableQuestions />
                </aside>
                <div className='survey-canvas col-md-9'>
                    <SurveyForm
                        title={this.state.title}
                        introduction={this.state.introduction}
                        onChange={this.handleFormChange}>
                    </SurveyForm>
                    <Divider>Questions</Divider>
                    <ReactCSSTransitionGroup transitionName='question'>
                        {questions}
                    </ReactCSSTransitionGroup>
                    <div
                        className={'drop-zone well well-drop-zone ' +
                        dropZoneEntered}
                        onDragOver={this.handleDragOver}
                        onDragEnter={this.handleDragEnter}
                        onDragLeave={this.handleDragLeave}
                        onDrop={this.handleDrop}>
                        >
                        Drag and drop a module from the left
                    </div>
                    <div className='actions'>
                        <button className="btn btn-save" onClick={this.
                            handleSaveClicked}>Save</button>
                        </div>
                    </div>
                </div>
            </div>
        </div>
    )
}
```

```
    );
}
```

---

this.props와 마찬가지로 render 메소드는 this.state에 따라 렌더링을 처리한다. 같은 엘리먼트를 속성만 약간 다르게 렌더링하거나, 서로 완전히 다른 엘리먼트를 렌더링할 수도 있다. 어떤 경우든 잘 동작한다.

## 5.4 상태 변경하기

이제 드래그 핸들러 메소드의 상태를 변경해서 컴포넌트를 다시 렌더링하게 하자. 그러면 React는 render 메소드를 호출한다. 그리고 this.state의 현재 값을 읽어서 title, introduction, questions를 화면에 출력할 것이다. 사용자는 모든 것이 잘 수정되었음을 확인할 수 있다.

컴포넌트의 상태를 변경할 때는 컴포넌트의 setState와 replaceState 메소드를 이용한다. replaceState는 기존 state 객체를 새로운 state 객체로 덮어쓴다. 상태를 관리하기 위해 불변<sup>immutable</sup> 자료 구조를 사용할 때 유용하다. 그렇지만 이 메소드를 사용하는 경우는 별로 없을 것이다. 이보다는 setState 메소드를 보다 더 많이 사용하게 될 텐데, 이 메소드는 기존 state 객체에 새로운 state 객체를 병합한다.

다음과 같은 state를 가지고 있다고 생각해보자.

---

```
getInitialState: function () {
  return {
    dropZoneEntered: false,
    title: 'Fantastic Survey',
    introduction: 'This survey is fantastic!',
    questions: []
  };
}
```

---

앞의 경우에 호출하는 `this.setState({title: "Fantastic Survey 2.0"})`는 `this.state.title`에만 영향을 주며, `this.state.dropZoneEntered`, `this.state.introduction`, `this.state.questions`는 영향을 받지 않는다.

`this.replaceState({title: "Fantastic Survey 2.0"})`를 호출하면 `state` 객체를 새로운 `state` 객체인 `{title: "Fantastic Survey 2.0"}`로 덮어쓰면서 `this.state.dropZoneEntered`, `this.state.introduction`, `this.state.questions`를 삭제한다. 이후에 `render` 메소드를 호출하면 제대로 동작하지 않을 수 있으며, `this.state.questions`는 배열이 아닌 `undefined` 값을 갖는다.

`this.setState`를 사용하면 앞에서 살펴본 핸들러 메소드를 구현할 수 있다.

---

```
handleFormChange: function (formData) {
  this.setState(formData);
},
handleDragOver: function (ev) {
  // handleDropZoneDrop 호출을 허용한다
  // https://code.google.com/p/chromium/issues/detail?
  id=168387
  ev.preventDefault();
},
handleDragEnter: function () {
  this.setState({dropZoneEntered: true});
},
handleDragLeave: function () {
  this.setState({dropZoneEntered: false});
},
handleDrop: function (ev) {
  var questionType = ev.dataTransfer.getData('questionType');
  var questions = this.state.questions;
  questions = questions.concat({ type: questionType });
  this.setState({
    questions: questions,
  });
},
```

```
        dropZoneEntered: false
    });
}

```

---

state 객체를 수정할 때 반드시 `setState`나 `replaceState` 메소드를 사용한다. 일반적으로 볼 때 `this.state.saveInProgress = true`와 같이 직접 state 객체에 접근해서 값을 수정하는 것은 좋은 방법이 아니다. React가 렌더링 필요 여부를 확인할 수 없기 때문이다. 그리고 이후에 `setState`를 호출했을 때 당황스러운 결과를 볼 수도 있다.

## 5.5 이벤트 객체

대부분의 이벤트 핸들러는 호출하는 것으로 충분하지만, 때로는 사용자 입력에서 더 많은 정보를 얻어내야 할 때가 있다.

설문조사 생성기 애플리케이션의 `AnswerEssayQuestion` 클래스를 살펴보자.

```
var AnswerEssayQuestion = React.createClass({
    handleComplete: function(event) {
        this.callMethodOnProps('onCompleted', event.target.value);
    },
    render: function() {
        return (
            <div className="form-group">
                <label className="survey-item-label">{this.props.label}</label>
                <div className="survey-item-content">
                    <textarea className="form-control" rows="3" onBlur={this.
handleComplete} />
                </div>
            </div>
        );
    }
});
```

---

React의 이벤트 핸들러 함수는 이벤트 객체를 전달받는데, 일반적인 JavaScript의 이벤트 리스너와 매우 유사하다. 여기서는 `handleComplete` 메소드가 이벤트 객체를 받아서 `event.target.value`에 접근해 `textarea`의 현재 값을 가져온다. 이런 식으로 이벤트 핸들러에서 `event.target.value`를 사용하는 것은 `input form`에서 값을 가져오는 흔한 방법이다. 특히 `onChange` 핸들러에서 많이 사용한다(`callMethodOnProps`은 `PropsMethodMixin`라는 믹스인이 제공하는 메소드라는 점을 유의하자. 이 믹스인은 부모와 자식 컴포넌트 간의 통신을 간편하게 해주는 메소드를 제공한다. 믹스인은 7장에서 더 자세하게 다룬다).

React는 원래의 이벤트 객체를 제공할 때 브라우저에서 받은 그대로를 전달하지 않고, `SyntheticEvent` 인스턴스로 한 번 맵핑한다. `SyntheticEvent`는 브라우저가 생성한 이벤트와 같은 형태와 작동 방식을 제공하는 동시에 크로스 브라우저 이슈에도 대응한다. `SyntheticEvent`는 보통의 이벤트 객체처럼 사용할 수 있으며, `SyntheticEvent`의 `nativeEvent` 프로퍼티를 이용해서 브라우저가 생성한 원래의 이벤트 객체에 접근할 수 있다.

## 5.6 정리

React를 이용하면 간단하게 UI를 이용한 사용자 입력에 따라 컴포넌트에 변화를 줄 수 있다.

1. React 컴포넌트에 이벤트 핸들러를 등록한다.
2. 이벤트 핸들러를 이용해서 컴포넌트의 내부 상태를 변경하면 React가 컴포넌트를 다시 렌더링한다.
3. 컴포넌트의 `render` 함수를 수정해서 렌더링 과정에 `this.state`를 적절하게 이용하게 한다.

지금까지는 사용자 입력에 응답하는 데 하나의 컴포넌트만 사용했다. 다음 장에서

는 여러 개의 컴포넌트를 조합하여 단순한 부분의 합 이상의 인터페이스를 만드는 방법을 살펴본다.

# 컴포넌트 구성

전통적인 HTML 개발 방식은 각 페이지를 기본 블록 엘리먼트로 구성한다. React는 페이지를 컴포넌트를 조합하는 페이지 구성 방식을 제안한다. React 컴포넌트는 JavaScript가 섞여 있는 HTML 엘리먼트로 생각할 수 있다. 엘리먼트를 가지고 HTML 문서를 만들 듯이 React를 사용할 때는 컴포넌트를 이용해서 페이지를 만든다.

React 애플리케이션은 전부 컴포넌트로 만들기 때문에, 이 책 역시 React 컴포넌트에 관한 책이라고 이야기할 수 있다. 이 장에서 컴포넌트에 대한 모든 것을 다루지는 않는다. 대신 컴포넌트 구성 방법을 소개한다.

컴포넌트는 기본적으로 JavaScript 함수다. `state`와 `props`를 인자로 받아 HTML을 렌더링한다. 컴포넌트는 애플리케이션의 데이터를 표현하도록 만들어졌다. 이런 점에서 React 컴포넌트를 HTML의 확장으로 볼 수 있다.

## 6.1 HTML 확장

React와 JSX 조합은 강력하고 표현력 있는 도구다. HTML과 유사한 문법으로 커스텀 엘리먼트를 생성할 수 있다. 라이프사이클을 이용해 커스텀 엘리먼트의 작동을 제어할 수 있다는 점은 순수 HTML로는 할 수 없는, React만의 장점이다.

`React.createClass` 메소드가 모든 출발점이다.

React는 상속보다 구성을 선호한다. 이는 작고 간단한 컴포넌트와 데이터 객체

를 결합해서 더 크고 복잡한 컴포넌트로 만들 수 있다는 의미다. 다른 MVC 프레임워크나 객체지향 도구를 사용해보았다면, React도 상속에 사용하는 `React.extendClass` 메소드를 가지고 있을 거로 생각했을 것이다. 하지만 React 컴포넌트는 상속할 수 없다. 대신 구성한다. 웹 페이지를 작성할 때 HTML DOM 노드를 상속하지 않는 것과 같다.

React는 결합성을 가지고 있다. 사용자는 다양한 자식 컴포넌트를 결합해서 복잡하면서도 강력한 새로운 컴포넌트를 만들 수 있다. 이 개념을 좀 더 명확하게 하도록 설문조사 생성기에서 사용자가 설문에 응답하는 방식을 생각해보자. 많은 컴포넌트 중에 특히 `AnswerMultipleChoiceQuestion` 컴포넌트는 여러 개의 질문을 렌더링하며, 사용자의 응답을 수집하는 역할을 한다.

#### NOTE

예제로 사용하는 설문조사 생성기의 전체 소스 코드는 Github 저장소에서 확인할 수 있다.

<https://github.com/backstopmedia/bleeding-edge-sample-app>

분명 HTML 폼 엘리먼트를 이용해서 설문을 만들 것이다. 그리고 설문 응답 컴포넌트를 더 정교하게 만들기 위해 기본 HTML `input` 엘리먼트를 포장하는 코드를 작성해서 작동방식을 원하는 대로 변경할 것이다.

## 6.2 예제

여러 질문을 표현하는 컴포넌트를 생각해보자. 몇 가지 요구사항이 있다.

- 사용자에게 선택 항목을 입력받는다.
- 선택 항목을 사용자에게 보여준다.
- 사용자는 한 가지 항목만 선택할 수 있다.

HTML의 기본 엘리먼트인 `radio` 버튼과 `input group`을 사용할 수 있다. 컴포

넌트 계층을 위에서 아래로 그려보면 다음과 같다.

---

MultipleChoice → RadioInput → Input (type="radio")

---

여기에서 화살표는 가지고 있다는 의미로 생각할 수 있다. `MultipleChoice` 컴포넌트는 `RadioInput` 컴포넌트를 가지고 있다. `RadioInput`은 `input` 엘리먼트를 갖는다. 이를 통해 컴포넌트 구성 패턴을 확인할 수 있다.

## HTML 조합

아래에서 위로 올라가면서 컴포넌트를 하나씩 조합하자. React는 `input` 컴포넌트를 `React.DOM.input` 네임스페이스에 이미 정의해놓다. 우선 이 컴포넌트를 `RadioInput` 컴포넌트로 포장해야 한다. 이 컴포넌트는 기본 `input`을 변경하는 역할을 하는데, `input`의 스코프를 좁게 만들어 `radio` 버튼처럼 작동하게 한다. 예제 앱의 `AnswerRadioInput` 컴포넌트가 이에 해당한다.

먼저 기본구조를 만든다. `render` 메소드를 추가하고, 원하는 출력 형태에 맞게 마크업을 작성한다. 다음 예제에서 컴포넌트를 특정 타입의 `input`으로 만드는 구성 패턴을 볼 수 있다.

---

```
var AnswerRadioInput = React.createClass({
  render: function() {
    return (
      <div className = "radio">
        <label>
          <input type = "radio" />
          Label Text
        </label>
      </div>
    );
  }
});
```

---

## 동적인 속성 추가하기

Input을 동적으로 만들기 위해 부모 컴포넌트가 radio input에 전달해야 하는 속성을 정의해야 한다.

- 어떤 값이나 선택 항목을 input이 보여줘야 할 것인가?(필수)
- 어떤 텍스트로 input을 설명할 것인가?(필수)
- input의 이름은 무엇인가?(필수)
- id를 수정할 수 있다.
- 기본값을 덮어쓸 수 있다.

앞의 목록을 바탕으로 속성의 타입을 정의할 수 있다. 클래스 작성할 때 propTypes 해시에 추가한다.

---

```
var AnswerRadioInput = React.createClass({  
  propTypes: {  
    id: React.PropTypes.string,  
    name: React.PropTypes.string.isRequired,  
    label: React.PropTypes.string.isRequired,  
    value: React.PropTypes.string.isRequired,  
    checked: React.PropTypes.bool  
  },  
  ...  
});
```

---

선택적으로 입력 가능한 속성은 기본값은 getDefaultProps 메소드에 작성한다. 부모 컴포넌트가 값을 제공하지 않으면 이 값이 각각의 새로운 인스턴스에 적용된다.

React는 클래스를 생성할 때 이 메소드를 단 한 번 호출하고, 개별 인스턴스를 만들 때는 호출하지 않는다. 따라서 인스턴스마다 자신의 고유값을 가지고 있어야 하는 id는 여기에서 지정할 수 없다. 이 문제는 state를 이용해서 해결한다.

---

```
var AnswerRadioInput = React.createClass({  
  propTypes: {...},  
  getDefaultProps: function() {  
    return {  
      id: null,  
      checked: false  
    };  
  },  
  ...  
});
```

---

## 상태 추적

컴포넌트는 시간에 따라 변하는 데이터를 추적해야 한다. 개별 인스턴스는 고유의 id를 가지고 있고, 사용자는 checked 값을 아무 때나 변경할 수 있다. 따라서 최초 상태를 정의해야 한다.

---

```
var AnswerRadioInput = React.createClass({  
  propTypes: {...},  
  getDefaultProps: function() {...},  
  getInitialState: function() {  
    var id = this.props.id ? this.props.id : uniqueId('radio-');  
    return {  
      checked: !!this.props.checked,  
      id: id,  
      name: id  
    };  
  },  
  ...  
});
```

---

이제 동적인 state와 props에 접근하여 마크업의 렌더링 상태를 변경할 수 있다.

---

```
var AnswerRadioInput = React.createClass({  
  propTypes: {...},  
  getDefaultProps: function() {...},
```

```
getInitialState: function() {...},
render: function() {
    return (
        <div className = "radio">
            <label htmlFor = {this.props.id}>
                <input type = "radio"
                    name = {this.props.name}
                    id = {this.props.id}
                    value = {this.props.value}
                    checked = {this.state.checked} />
                {this.props.label}
            </label>
        </div>
    );
}
});
```

---

## 부모 컴포넌트로 통합하기

이제 부모 컴포넌트로 통합할 준비가 끝났다. 다음 레이어인 AnswerMultipleChoiceQuestion 컴포넌트를 만들어 보자. 여기에서 할 일은 사용자가 고를 수 있는 선택 목록을 그리는 것이다. 앞에서 이야기한 패턴에 따라 이번에도 기본적인 HTML과 기본 props를 먼저 작성한다.

---

```
var AnswerMultipleChoiceQuestion = React.createClass({
    propTypes: {
        value: React.PropTypes.string,
        choices: React.PropTypes.array.isRequired,
        onCompleted: React.PropTypes.func.isRequired
    },
    getInitialState: function() {
        return {
            id: uniqueId('multiple-choice-'),
            value: this.props.value
        };
    },
    render: function() {
```

```
        return (
            <div className = "form-group">
                <label className = "survey-item-label"
                    htmlFor = {this.state.id}{this.props.label}</label>
                <div className="survey-item-content">
                    <AnswerRadioInput ... />
                    ...
                    <AnswerRadioInput... />
                </div>
            </div>
        );
    }
});
```

---

자식 radio 컴포넌트 목록을 생성하려면 선택항목 배열을 배열에 매핑된 컴포넌트로 전환해야 한다. 도우미 함수를 이용하면 간단히 해결할 수 있다. `renderChoices`를 확인한다.

```
var AnswerMultipleChoiceQuestion = React.createClass({
    ...
    renderChoices: function() {
        return this.props.choices.map(function(choice, i) {
            return AnswerRadioInput({
                id: "choice-" + i,
                name: this.state.id,
                label: choice,
                value: choice,
                checked: this.state.value === choice
            });
        }.bind(this));
    },
    render: function() {
        return (
            <div className = "form-group">
                <label className = "survey-item-label"
                    htmlFor = {this.state.id}{this.props.label}</label>
                <div className="survey-item-content">
                    {this.renderChoices()}
                </div>
            </div>
        );
    }
});
```

```
        </div>
    </div>
);
}
});
```

---

이제 React가 이야기하는 결합성의 의미가 더 명확해졌다. 기본 `input`을 이용해서 `radio input`으로 만든다. 그다음에 여러 개의 `radio input`을 포장하는 다음 선택 컴포넌트를 만든다. 이것은 특정 목적을 위해 만든 폼 컨트롤인 셈이다. 이제 여러 개의 선택사항을 아주 간단하게 렌더링할 수 있다.

```
<AnswerMultipleChoiceQuestion choices={arrayOfChoices} ... />
```

---

눈치가 빠른 독자라면 여기서 빠진 부분을 발견했을 것이다. 현재로써는 `radio input`의 상태 변화를 부모 컴포넌트에 전달할 수 없다. `AnswerRadioInput` 컴포넌트와 부모 컴포넌트를 연결해서 부모 컴포넌트가 변화를 파악할 수 있게 만들어야 한다. 그래야 부모 컴포넌트가 설문 결과 데이터를 적절하게 처리할 수 있다. 이제 부모 컴포넌트와 자식 컴포넌트의 관계를 구성하는 방법을 알아본다.

## 6.3 부모 컴포넌트와 자식 컴포넌트의 관계

이제 폼을 화면에 렌더링할 수 있다. 하지만 아직 사용자의 조작을 컴포넌트끼리 공유하지 못한다. `AnswerRadioInput` 컴포넌트는 아직 부모 컴포넌트와 통신할 수 없다.

부모 컴포넌트와 자식 컴포넌트가 통신하는 가장 간단한 방법은 `props`를 이용하는 것이다. 부모 컴포넌트는 속성을 통해 콜백을 전달하면, 자식 컴포넌트가 필요할 때 콜백을 호출하는 방식이다.

먼저 자식 컴포넌트에 변경이 있을 때 AnswerMultipleChoiceQuestion 컴포넌트의 동작을 정의한다. handleChanged 메소드를 만들어서 AnswerRadioInput 컴포넌트에 넘겨준다.

---

```
var AnswerMultipleChoiceQuestion = React.createClass({
  ...
  handleChanged: function(value) {
    this.setState({
      value: value
    });
    this.props.onCompleted(value);
  },
  renderChoices: function() {
    return this.props.choices.map(function(choice, i) {
      return AnswerRadioInput({
        ...
        onChange: this.handleChanged
      });
    }.bind(this));
  },
  ...
});
```

---

이제 각 radio input은 사용자 조작을 확인해서 부모 컴포넌트에 값을 전달할 수 있다. 이렇게 하려면 이벤트 핸들러를 input의 onChange 이벤트에 연결해야 한다.

---

```
var AnswerRadioInput = React.createClass({
  propTypes: {
    ...
    onChange: React.PropTypes.func.isRequired
  },
  handleChanged: function(e) {
    var checked = e.target.checked;
    this.setState({
      checked: checked
    });
  }
});
```

```
});  
    if (checked) {  
        this.props.onChanged(this.props.value);  
    }  
},  
render: function() {  
    return (  
        <div className="radio">  
            <label htmlFor={this.state.id}>  
                <input type="radio"  
                    ...  
                    onChange={this.handleChange} />  
                {this.props.label}  
            </label>  
        </div>  
    );  
}  
};
```

---

## 6.4 정리

여기까지 React를 이용한 컴포넌트 구성 패턴을 살펴봤다. HTML 엘리먼트나 커스텀 컴포넌트를 포장해서 원하는 대로 동작하게 만들 수 있다. 컴포넌트를 구성하는 방식을 이용하면 컴포넌트의 목적이 뚜렷해지고 의미가 더 잘 드러난다.

React는 먼저 기본 input 엘리먼트를 받는다.

---

```
<input type="radio" ... />
```

---

이것을 조금 더 의미있는 형태로 변환한다.

---

```
<AnswerRadioInput ... />
```

---

마지막으로 배열 데이터를 이용해 사용자를 위한 UI를 만들 수 있는 컴포넌트를 완성했다.

---

```
<AnswerMultipleChoiceQuestion choices={arrayOfChoices} ... />
```

---

구성은 React 컴포넌트를 목적에 맞게 만들 수 있는 기능 중의 하나일 뿐이다. 믹스인<sup>Mixins</sup>은 또 다른 방법을 보여준다. 믹스인을 이용하면 여러 컴포넌트에 공통으로 사용할 수 있는 메소드를 정의할 수 있다. 다음 장에서는 믹스인을 정의하는 방법과 믹스인을 이용해 공통 코드를 공유하는 방법을 설명한다.



# 믹스인

지난 장에서 설명한 대로 믹스인을 이용하면 다른 컴포넌트 간에 공유할 수 있는 메소드를 정의할 수 있다. 이 장에서 믹스인에 대해서 더 자세히 살펴본다.

## 7.1 믹스인은 무엇인가?

React 홈페이지에 있는 타이머 컴포넌트 예제를 보자.

```
var Timer = React.createClass({
  getInitialState: function() {
    return {
      secondsElapsed: 0
    };
  },
  tick: function() {
    this.setState({
      secondsElapsed: this.state.secondsElapsed + 1
    });
  },
  componentDidMount: function() {
    this.interval = setInterval(this.tick, 1000);
  },
  componentWillUnmount: function() {
    clearInterval(this.interval);
  },
  render: function() {
    return (
      <div>Seconds Elapsed: {this.state.secondsElapsed}</div>
    );
  }
});
```

```
    }
});
```

---

이런 방법도 좋다. 하지만 여러 컴포넌트가 같은 코드를 이용해서 타이머를 사용하고 싶은 경우가 있다. 이때 믹스인을 사용할 수 있다. 타이머 컴포넌트를 믹스인으로 바꿔보자.

```
var Timer = React.createClass({
  mixins: [ IntervalMixin(1000) ],
  getInitialState: function() {
    return {
      secondsElapsed: 0
    };
  },
  onTick: function() {
    this.setState({
      secondsElapsed: this.state.secondsElapsed + 1
    });
  },
  render: function() {
    return (
      <div>Seconds Elapsed: {this.state.secondsElapsed}</div>
    );
  }
});
```

---

믹스인은 쉽게 만들 수 있다. 객체를 컴포넌트 클래스에 첨가한다. React는 여기에서 더 나아가 함수를 덮어쓰지 않고, 여러 믹스인을 적용할 수 있는 방법을 제공한다. 이 부분은 React가 아닌 다른 시스템에서 충돌을 일으킬 수 있다. 다음 예제를 보자.

```
React.createClass({
  mixins: [
    getInitialState: function() {
```

---

```
        return {a: 1}
    }
},
getInitialState: function() {
    return {b: 2}
}
});
});
```

---

getInitialState 메소드가 컴포넌트 클래스와 믹스인, 양쪽에 존재한다. 초기화 상태는 {a: 1, b: 2}이다. 양쪽에 같은 키가 있다면, 에러가 발생한다.

믹스인 역시 컴포넌트가 시작할 때 호출되는 componentDidMount 같은 라이프사이클 메소드를 가질 수 있다. React는 믹스인 배열을 순차적으로 처리하면서 componentDidMount를 호출한다. 만약 컴포넌트 클래스가 componentDidMount 메소드를 가지고 있다면 이 메소드 역시 마찬가지로 호출된다.

최초의 예제로 돌아가서 IntervalMixin을 구현하자. 간단한 객체로 충분할 때도 있지만, 객체를 반환하는 함수를 만들어야 할 때가 더 많다.

다음 예제에서 interval을 어떻게 사용하는지 살펴보자.

```
var IntervalMixin = function(interval) {
    return {
        componentDidMount: function() {
            this._interval = setInterval(this.onTick, interval);
        },
        componentWillUnmount: function() {
            clearInterval(this._interval);
        }
    };
};
```

---

앞의 예제는 아주 훌륭하지만, 몇 가지 제약을 가지고 있다. 여러 개의 interval을 쓸 수 없고, 사용자가 interval을 제어하는 함수를 선택할 수 없다. 또한

interval을 수동으로 제거할 때에는 내부의 `__interval` 속성을 사용해야만 한다. 이 문제를 해결하기 위해 믹스인에 공개 API를 추가할 수 있다.

다음 예제는 2014년 1월 1일부터 현재까지 몇 초가 지났는지 보여준다. 믹스인에 약간의 코드를 추가해서 훨씬 유연하고 강력하게 만들었다.

---

```
var IntervalMixin = {
  setInterval: function(callback, interval) {
    var token = setInterval(callback, interval);
    this.__intervals.push(token);
    return token;
  },
  componentDidMount: function() {
    this.__intervals = [];
  },
  componentWillUnmount: function() {
    this.__intervals.map(clearInterval);
  }
};

var Since2014 = React.createClass({
  mixins: [IntervalMixin],
  componentDidMount: function() {
    this.setInterval(this.forceUpdate.bind(this), 1000);
  },
  render: function() {
    var from = Number(new Date(2014, 0, 1));
    var to = Date.now();
    return (
      <div>{Math.round((to-from) / 1000)}</div>
    );
  }
});
```

---

믹스인의 예는 여기에서 다 보여줄 수 없을 정도로 다양하다. 몇 가지 대표적인 사례를 들어보자면,

- 이벤트를 리스닝하다가 state에 적용하는 믹스인(예: flux store 믹스인)
- XHR 업로드를 제어하고 업로드 진행과 상황을 state에 적용하는 믹스인
- 자식을 <body>의 끝에 렌더링할 수 있게 도와주는 레이어 믹스인(예: 레이어 팝업)

## 7.2 정리

믹스인은 코드 반복을 제거하고 컴포넌트가 필요한 부분에만 집중할 수 있게 도와주는 강력한 도구다. 믹스인을 이용하면 강력한 추상화를 구현할 수 있으며 복잡한 문제를 우아하게 해결할 수 있다.

믹스인을 반드시 여러 개의 컴포넌트에서 사용할 목적으로 작성할 필요는 없다. 믹스인을 사용하면 어떤 동작이나 역할을 쉽게 설명하고, 컴포넌트에 적용할 수 있다. 또한, 컴포넌트의 코드를 줄일 수 있다. 덕분에 가독성도 높아진다. 앞에서 살펴본 `_intervals`처럼 보기 좋지 않은 코드를 작성해야 할 때, 이런 코드를 믹스인으로 만들어 감출 수 있다.

다음 장에서는 DOM을 소개하면서 컴포넌트에서 분리할 수 있는 동작과 역할에 대해서 생각해보고 직접 수정해본다.



# DOM 조작

React 가상 표현 객체를 이용하면 대체로 DOM을 직접 조작하지 않고도 원하는 사용자 경험을 만들 수 있다. 컴포넌트를 결합해 복잡한 인터랙션을 응집력 있게 하나로 엮어 사용자한테 제공할 수 있다. 하지만 불가피하게 직접 DOM을 다뤄야 하는 경우도 있다. React를 사용하지 않는 외부 라이브러리를 이용하거나 React가 지원하지 않는 기능을 수행하기 위해 직접 DOM을 조작해야 할 때가 있기 때문이다. 이 문제를 해결하기 위해 React는 DOM 노드를 이용할 수 있는 시스템을 제공한다. 컴포넌트 라이프사이클의 특정 시점에만 접근할 수 있지만, 이런 문제를 해결하는 데 매우 유용하다.

## 8.1 React를 통한 DOM 노드 접근

React를 이용해 DOM 노드에 접근하려면, 먼저 DOM을 관리하는 컴포넌트에 접근해야 한다. 다음 예제처럼 자식 컴포넌트에 ref 속성을 추가한다.

```
var DoodleArea = React.createClass({
  render: function() {
    return <canvas ref="mainCanvas" />;
  }
});
```

이제 <canvas> 컴포넌트에 this.refs.mainCanvas로 접근할 수 있다. 자식 컴

포넌트에 추가하는 ref는 유일성을 가져야 하며, 같은 값을 다른 자식 컴포넌트에 사용할 수 없다. 다른 자식 컴포넌트의 ref로 mainCanvas를 사용하면 컴포넌트가 제대로 작동하지 않는다. 일단 자식 컴포넌트에 접근해서 getDOMNode() 메소드<sup>01</sup>를 호출하면 DOM 노드를 가져올 수 있다. 그렇지만 render 메소드에서는 이렇게 할 수 없다. 렌더링이 끝나고 React가 업데이트를 수행하기 전까지 DOM 노드가 최신 상태가 아니거나, 때에 따라서는 아예 DOM 노드가 생성도 채 되지 않았을 수 있기 때문이다. 따라서 getDOMNode() 메소드는 컴포넌트 렌더링이 끝난 후에야 사용할 수 있다. componentDidMount 핸들러가 실행되는 시점이다.

---

```
var DoodleArea = React.createClass({
  render: function() {
    // render 메소드 내부에서는 렌더링이 끝나지 않았기 때문에 예외가 발생한다.
    this.getDOMNode();

    return <canvas ref="mainCanvas" />
  },
  componentDidMount: function() {
    var canvasNode = this.refs.mainCanvas.getDOMNode();
    // 이렇게 렌더링 이후에 사용하면 정상 작동한다.
    // 이제 HTML5 Canvas 노드에 접근해서 페인팅 메소드를 사용할 수 있다.
  }
});
```

---

componentDidMount에서만 getDOMNode를 호출할 수 있는 것은 아니다. React는 렌더링이 끝나면 이벤트 핸들러도 실행한다. 따라서 componentDidMount에서 사용한 것처럼 이벤트 핸들러에서도 getDOMNode를 사용할 수 있다.

---

```
var RichText = React.createClass({
  render: function() {
    return <div ref="editableDiv" contentEditable="true" onKeyDown={this.
```

<sup>01</sup> 역사주\_ ES6 클래스를 이용하여 생성한 컴포넌트는 getDOMNode API를 제공하지 않는다. 대신 React.findDOMNode(component)를 이용해야 한다.

```
handleKeyDown}>
},
handleKeyDown: function() {
  var editor = this.refs.editableDiv.getDOMNode()
  var html = editor.innerHTML;
  // 이제 사용자가 입력한 HTML 콘텐츠를 확인할 수 있다.
}
});
```

---

앞의 예제는 `contentEditable`를 적용한 `div`를 만들어 사용자에게 리치 텍스트를 입력기능을 제공하는 코드를 보여준다.

React는 컴포넌트의 HTML 콘텐츠에 접근하는 방법을 제공하지 않는다. 대신에 `keyDown` 핸들러를 이용해 `div`의 DOM 노드에 접근하는 방식으로 HTML 콘텐츠에 접근할 수 있다. 이렇게 해서 사용자가 입력한 내용을 저장할 수 있고, 작성한 글자 수를 표시하는 등의 기능을 만들 수 있다.

`refs`와 `getDOMNode`는 DOM에 접근할 때 유용하게 사용할 수 있지만, 원하는 기능을 만드는 데 다른 방법이 없을 때만 제한적으로 사용하는 것이 좋다. 이 방법은 React의 성능 최적화를 방해하고, 애플리케이션의 복잡도를 높인다. 따라서 보통의 방법으로는 해결할 수 없는 경우에만 사용한다.

## 8.2 React 외의 라이브러리 포함하기

유용한 많은 라이브러리 중에는 React를 적용하기 까다로운 것들이 있다. DOM에 접근할 필요가 없는 날짜/시간 조작 라이브러리 같은 경우에는 문제가 없다. 하지만 DOM에 접근해야 하는 라이브러리는, 라이브러리의 상태를 React 컴포넌트와 동기화하는 것이 결합을 가로막는 걸림돌이 될 수 있다.

다음 예제 코드처럼 자동완성<sup>autocomplete</sup> 라이브러리를 사용하는 경우를 생각해보자.

---

```
autocomplete({
  target: document.getElementById("cities"),
  data: [
    "San Francisco",
    "St. Louis",
    "Amsterdam",
    "Los Angeles"
  ],
  events: {
    select: function(city) {
      alert("You have selected the city of " + city);
    }
  }
});
```

---

autocomplete 함수를 사용하기 위해서는 적용할 DOM 노드, 데이터로 사용할 문자열 배열, 이벤트 핸들러가 필요하다. React와 라이브러리의 장점을 모두 끌어내려면, React 컴포넌트가 autocomplete() 필요로 하는 조건을 제공하게 해야 한다.

---

```
var AutocompleteCities = React.createClass({
  render: function() {
    return <div id = "cities"
      ref = "autocompleteTarget" / >
  },
  getDefaultProps: function() {
    return {
      data: [
        "San Francisco",
        "St. Louis",
        "Amsterdam",
        "Los Angeles"
      ]
    };
  },
  handleSelect: function(city) {
    alert("You have selected the city of " + city);
  }
});
```

---

React로 라이브러리를 포장하기 위해서 componentDidMount 핸들러를 추가한다. componentDidMount 핸들러는 자식 컴포넌트인 autocompleteTarget의 DOM을 이용해 React와 라이브러리를 연결한다.

---

```
var AutocompleteCities = React.createClass({
  render: function() {
    return <div id="cities" ref="autocompleteTarget" />
  },
  getDefaultProps: function() {
    return {
      data: [
        "San Francisco",
        "St. Louis",
        "Amsterdam",
        "Los Angeles"
      ]
    };
  },
  handleSelect: function(city) {
    alert("You have selected the city of " + city);
  },
  componentDidMount: function() {
    autocomplete({
      target: this.refs.autocompleteTarget.getDOMNode(),
      data: this.props.data,
      events: {
        select: this.handleSelect
      }
    });
  }
});
```

---

React가 componentDidMount를 DOM 노드 당 한 번씩만 호출한다는 점을 알아두자. 따라서 autocomplete를 같은 노드에 두 번 적용해서 원하지 않는 결과를 만들지 않을까 걱정하지 않아도 된다.

이 컴포넌트를 제거한 후 다른 DOM 노드에서 다시 렌더링할 수 있다는 점을 기억한다. DOM 노드를 제거했는데 `componentDidMount`의 영향이 계속 된다면 메모리 누수나 다른 문제로 이어질 수 있다. `componentWillUnmount` 핸들러를 이용해서 컴포넌트를 정리함으로써 이 문제를 해결할 수 있다.

### 8.3 부모 엘리먼트에 영향을 주는 플러그인 다루기

앞서 살펴본 자동완성 예제는 자동완성이 자식만 수정하는 플러그인이라고 가정했다. 그러나 실제로 이런 경우는 보기 어렵다.

이런 플러그인을 사용할 때는 React에서 플러그인을 숨겨야 한다. 그렇지 않으면 예상치 못한 DOM 변경으로 오류가 발생한다. 게다가 추가로 제거 작업을 해야 할 수도 있다.

다음 예제는 가상의 jQuery 플러그인을 다룬다. 이 플러그인은 커스텀 이벤트를 일으켜서, 이벤트를 받는 엘리먼트를 변경한다. 부모 엘리먼트를 수정하는 플러그인은 React와 함께 사용하기에 적절하지 않다. 이때는 다른 플러그인을 찾거나 소스를 수정하는 것이 가장 좋다.

이런 플러그인은 방어적으로 사용하기 위해 플러그인의 DOM 노드를 직접 관리한다. 자식이나 속성이 없는 `div`를 생성하는 컴포넌트를 만든다.

---

```
var SuperSelect = React.createClass({
  render: function(){ return
  }
});
```

`componentDidMount`에서 플러그인을 사용하기 위해 지저분한 코드를 추가했다.

```
var SuperSelect = React.createClass({
  render: function(){ return
  },
  componentDidMount: function(){
```

```
var el = this.el = document.createElement('div');
this.getDOMNode().appendChild(el);
$(el).superSelect(this.props);
$(el).on('superSelect', this.handleSuperSelectChange);
},
handleSuperSelectChange: function()
{ ... },
});
```

---

직접 제어할 수 있는 컴포넌트 div 내부에 새로운 div를 추가했다. 나중에 이 div를 책임지고 정리해야 한다는 의미이기도 하다.

```
componentWillUnmount: function() {
  // DOM에서 노드를 제거한다.
  this.getDOMNode().removeChild(this.el);

  // superSelect와 React의 리스너를 제거한다.
  $(this.el).off();
}
```

---

추가로 해당 플러그인의 문서를 확인하여 노드를 제거할 때 추가 작업을 해야 하는지도 확인한다. 전역 이벤트 리스너, 타이머, AJAX 요청도 제거해야 한다.

뿐만 아니라 업데이트도 신경 써야 한다. 크게 두 가지 방법을 생각해 볼 수 있다. componentWillMount와 componentDidMount를 사용하거나, 플러그인의 업데이트 API를 사용할 수 있다. 첫 번째 방법이 더 확실하지만, 두 번째 방법은 깔끔하고 성능상 이점이 있다.

componentWillUnmount와 componentDidMount를 직접 호출하는 방법이다.

```
componentDidUpdate: function(){
  this.componentWillUnmount();
  this.componentDidMount();
}
```

---

그리고 플러그인을 업데이트하는 방법은 이런 식이다.

---

```
componentWillReceiveProps: function(nextProps){  
    $(this.el).superSelect('update', nextProps);  
}  
});
```

---

다양한 변수를 가지고 있는 라이브러리나 플러그인은 포장하기 어렵다. 사용하고 싶은 대상이 단순한 jQuery 플러그인일 수도 있고, 리치 텍스트 편집기처럼 자체 플러그인을 가지고 있는 복잡한 경우일 수도 있다. 단순한 플러그인은 쉽게 React 컴포넌트로 포장할 수 있다. 하지만 복잡한 플러그인을 React 컴포넌트로 만든다는 것은 불가능하다.

## 8.4 정리

React 컴포넌트를 개발하다 보면 가상 표현 객체를 사용하는 것만으로 충분하지 않을 때가 있다. 이런 경우에는 componentDidMount를 실행한 후에 ref 속성을 이용해 특정 엘리먼트에 접근하면 getDOMNode를 이용해 DOM 노드를 수정할 수 있다.

이 방법을 이용하면 React가 지원하지 않는 기능을 사용할 수 있으며, React를 지원하지 않는 외부 라이브러리를 React 컴포넌트에 통합할 수 있다.

다음 장에서는 React를 이용해 폼을 생성하고 관리하는 법을 살펴본다.

폼 Form<sup>01</sup>은 사용자가 애플리케이션에 입력할 때 중요한 역할을 한다. 그렇지만 기존의 단일 페이지 애플리케이션은 폼을 올바르게 사용하기가 어려웠다. 사용자가 폼의 상태를 마음대로 바꿀 수 있었기 때문이다. 폼의 상태를 처리하는 일은 복잡하며 자주 버그를 만든다. React를 이용하면 애플리케이션의 상태를 처리하는 것처럼 폼을 처리할 수 있다.

예측 가능성과 테스트 가능성은 React 컴포넌트의 핵심이다. 같은 props와 state를 가지고 있는 React 컴포넌트는 항상 같은 형태로 렌더링 된다. 폼도 예외가 아니다.

React는 두 종류의 폼 컴포넌트를 제공하는데, value 값의 유무에 따라 제어 Controlled 폼 컴포넌트와 비제어 Uncontrolled 폼 컴포넌트라고 부른다. 이번 장에서는 이 두 컴포넌트의 차이점을 알아보고, 어떤 상황에서 사용하는 것이 좋은지 생각해본다.

또한, 다음 내용도 이번 장에서 소개한다.

- React로 폼 이벤트 처리 방법
- 조작 폼 컴포넌트로 데이터 입력 제어하기
- React로 폼 컴포넌트의 인터페이스를 변경하는 방법
- 폼 컴포넌트 네이밍의 중요성

<sup>01</sup> 역사주\_ 여기에서 언급하는 form은 HTML의 form 엘리먼트가 아닌 HTML이 제공하는 input, checkbox, radio, select 엘리먼트 같은 모든 사용자 입력 엘리먼트를 가르킨다.

- 여러 개의 조작 폼 컴포넌트 다루기
- 재사용할 수 있는 커스텀 폼 컴포넌트 만들기
- React로 AutoFocus 이용하기
- 사용할 수 있는 애플리케이션 만드는 팁

이전 장에서 React 컴포넌트의 DOM에 접근하는 방법을 배웠다. React를 이용하면 DOM의 상태를 컴포넌트로 옮길 수 있지만, 복잡한 폼 컴포넌트를 다룰 때는 직접 DOM에 접근해야 한다.

예제 애플리케이션인 설문조사 생성기는 폼을 비표준 방식으로 사용한다. 설문 기준에 따라 폼을 동적으로 생성하기 위해서다. 이 장의 예제도 마찬가지로, 설문 조사 생성기와 같은 방식으로 개념을 전달한다. 예제를 통해서 React를 이용해 폼을 다루는 과정을 더 자세히 살펴본다.

NOTE

예제로 사용하는 설문조사 생성기의 전체 소스 코드는 Github 저장소에서 확인할 수 있다.

<https://github.com/backstopmedia/bleeding-edge-sample-app>

## 9.1 비제어 컴포넌트

대체로 중요한 폼에는 비제어 컴포넌트를 사용하고 싶지 않을 것이다. 그런데 제어 컴포넌트를 이해하는 데에 비제어 컴포넌트는 아주 유용하다. 비제어 컴포넌트는 대부분의 React 컴포넌트가 만들어진 방식, 즉 부모 컴포넌트에서 값을 설정하는 것과 정반대로 만들어졌다.

HTML의 폼은 React 컴포넌트와 다르게 동작한다. HTML <input/>에 value가 있으면 <input/>은 value 값을 변경할 수 있다. 비제어 컴포넌트라는 이름은 React 컴포넌트가 폼 컴포넌트의 value를 제어하지 않는다는 것을 뜻한다.

React를 이용할 때는 defaultValue를 이용해 <input/>에 기본값을 설정한다.

---

```
//http://jsfiddle.net/pmsy5y2u/
var MyForm = React.createClass({
  render: function () {
    return <input
      type="text"
      defaultValue="Hello World!" />;
  }
});
```

---

앞의 예제가 바로 비제어 컴포넌트를 보여주고 있다. 부모 컴포넌트가 value를 설정하는 것이 아니라, <input/>이 자체적으로 value를 조작한다.

비제어 컴포넌트의 value에 접근할 수 없다면 별로 쓸모가 없을 것이다. ref를 <input/>에 추가하면 DOMNode를 통해서 value에 접근할 수 있다.

ref는 DOM 속성이 아닌 특별 속성으로, this 안에서 컴포넌트를 식별하는 데 사용한다. 모든 컴포넌트의 refs는 this.refs를 이용해서 쉽게 접근할 수 있다.

다음 예제는 <input/>을 폼에 추가하고, value 값이 전달되면 알림으로 보여준다.

---

```
//http://jsfiddle.net/opfkts4/
var MyForm = React.createClass({
  submitHandler: function (event) {
    event.preventDefault();
    //ref로 input에 접근한다
    var helloTo = this.refs.helloTo.getDOMNode().value;
    alert(helloTo);
  },
  render: function () {
    return <form onSubmit={this.submitHandler}>
      <input
        ref="helloTo"
        type="text"
        defaultValue="Hello World!" />
    </form>
  }
});
```

```
<br />
<button type="submit">Speak</button>
</form>;
}
});
```

---

비제어 컴포넌트는 입력값을 검사하거나 제어할 필요가 없는 기본 품에 가장 적합하다.

## 9.2 제어 컴포넌트

제어 컴포넌트는 다른 React 컴포넌트와 같은 방식을 따른다. React 컴포넌트가 폼 컴포넌트의 상태를 제어하며, React 컴포넌트의 상태를 `value`에 저장한다. 폼 컴포넌트를 더 많이 제어하고 싶을 때는 제어 컴포넌트를 사용한다. 제어 컴포넌트는 부모 컴포넌트가 `input`의 `value` 값을 설정한다.

앞에서 봤던 예제를 제어 컴포넌트로 변경했다.

---

```
//http://jsfiddle.net/1a8xr2z6/
var MyForm = React.createClass({
  getInitialState: function () {
    return {
      helloTo: "Hello World!"
    };
  },
  handleChange: function (event) {
    this.setState({
      helloTo: event.target.value
    });
  },
  submitHandler: function (event) {
    event.preventDefault();
    alert(this.state.helloTo);
  }
});
```

```
    },  
  
    render: function () {  
      return <form onSubmit={this.handleSubmit}>  
        <input  
          type="text"  
          value={this.state.helloTo}  
          onChange={this.handleChange} />  
        <br />  
        <button type="submit">Speak</button>  
      </form>;  
    }  
  );  
};
```

---

<input/>의 value를 부모 컴포넌트의 상태에 저장한다는 점이 가장 중요하게 바뀐 부분이다. 결과적으로 데이터 흐름이 명확해졌다.

- getInitialState가 defaultValue를 설정한다.
- 렌더링하면서 <input/> value를 설정한다.
- <input/>의 value가 바뀌면 onChange에 등록한 change 핸들러를 호출한다.
- change 핸들러가 state를 변경한다.
- 렌더링하면서 <input/> value를 변경한다.

비제어 컴포넌트 때보다 코드의 라인 수가 훨씬 많아졌다. 하지만 이제 데이터 흐름을 제어하고 사용자가 입력한 값으로 컴포넌트의 상태를 변경할 수 있다.

사용자가 입력한 값을 대문자로 변환하는 예제다.

---

```
handleChange: function (event) {  
  this.setState({  
    helloTo: event.target.value.toUpperCase()  
  });  
}
```

---

이 예제를 작성해서 브라우저에서 실행해보면 데이터를 입력할 때 소문자가 대문

자로 전환되어도 깜빡거림이 발생하지 않는 것을 알 수 있다. 이는 React가 브라우저의 네이티브 change 이벤트를 가로채기 때문이다. 컴포넌트는 setState 호출 후에 input을 다시 렌더링한다. React는 DOM의 diff를 만들어서 비교한 다음에 input 엘리먼트의 value를 변경한다.

이 방법을 응용하면 입력 문자를 제한하거나, 이메일 주소를 검증할 수 있다.

또한, 입력받은 데이터를 다른 컴포넌트의 value 값으로 사용할 수도 있다.

- 입력 글자 수 제한이 걸린 경우, 입력 가능한 남은 글자 수 표시
- HEX 값으로 입력한 색상 표시
- 자동완성 표시
- 입력 값을 이용해 다른 UI 엘리먼트 변경

### 9.3 폼 이벤트

폼을 다양하게 제어하는 데에 폼 이벤트 접근은 중요한 부분이다.

React는 HTML이 제공하는 모든 이벤트를 지원한다. React의 이벤트 명은 카멜표기법을 따르며, React는 기본 이벤트를 가공해서 인조 synthetic 이벤트로 변환 한다. 이 이벤트는 크로스 브라우징에 적합하게 표준화되어 있다.

React의 인조 이벤트를 이용하면 DOMNode에 접근할 수 있다. event.target을 이용한다.

---

```
handleEvent: function (syntheticEvent) {
  var DOMNode = syntheticEvent.target;
  var newValue = DOMNode.value;
}
```

---

조작 컴포넌트 value에 접근하는 가장 간단한 방법을 살펴봤다.

## 9.4 레이블

폼 엘리먼트의 레이블은 사용자 요구사항에 명확하게 대응하고, radio 버튼과 체크박스의 접근성을 높이는 데 중요한 역할을 한다.

React에서 레이블 사용할 때 for 속성이 문제가 된다. JSX를 사용할 때 속성은 JavaScript 객체로 변환되어 컴포넌트 생성자의 첫 번째 인자로 전달된다. JavaScript에서 for는 예약어이므로 객체의 속성 이름으로 사용할 수 없다.

이 때문에 React로 개발할 때 for를 사용할 수 없으며 대신 htmlFor를 써야 한다. class를 className으로 써야 하는 것도 같은 이유에서다.

---

```
//JSX
<label htmlFor="name">Name:</label>

//JavaScript
React.DOM.label({htmlFor:"name"}, "Name:");

//렌더링 후
<label for="name">Name:</label>
```

---

## 9.5 textarea와 select

React는 <textarea/>와 <select/> 사용 방식의 일관성을 높이고 더 쉬운 조작법을 사용자에게 제공하기 위해서 인터페이스를 약간 변경했다.

<textarea/>는 <input/>처럼 value와 defaultValue를 지정할 수 있다.

---

```
//비제어 컴포넌트
<textarea defaultValue="Hello World" />

//제어 컴포넌트
<textarea
  value={this.state.helloTo}
  onChange={this.handleChange} />
```

---

<select/>는 value와 defaultValue로 선택된 option을 지정할 수 있다. 덕분에 value를 더욱 쉽게 조작할 수 있다.

---

```
//제어 컴포넌트
<select defaultValue="B">
  <option value="A">First Option</option>
  <option value="B">Second Option</option>
  <option value="C">Third Option</option>
</select>

//비제어 컴포넌트
<select value={this.state.helloTo} onChange={this.handleChange}>
  <option value="A">First Option</option>
  <option value="B">Second Option</option>
  <option value="C">Third Option</option>
</select>
```

---

React는 다중 선택을 지원한다. value와 defaultValue 값으로 배열을 전달한다.

---

```
//비제어 컴포넌트
<select multiple="true" defaultValue={['A', 'B']}>
  <option value="A">First Option</option>
  <option value="B">Second Option</option>
  <option value="C">Third Option</option>
</select>
```

---

다중 선택을 이용하면 옵션을 선택했을 때 select 컴포넌트의 값이 변경되지 않는다. 옵션의 selected 속성이 바뀌었을 때 비로소 값이 바뀐다. 옵션 선택 여부를 확인하려면 ref나 syntheticEvent.target을 이용해서 옵션에 접근한다.

다음 예제에서는 handleChange가 DOM을 순회하면서 사용자가 선택한 옵션을 찾는다.

---

```
//http://jsfiddle.net/yddy2ep0/
var MyForm = React.createClass({
```

```
getInitialState: function () {
  return {
    options: ["B"]
  };
},

handleChange: function (event) {
  var checked = [];
  var sel = event.target;
  for(var i=0; i < sel.length; i++){
    var option = sel.options[i];
    if (option.selected){
      checked.push(option.value);
    }
  }
  this.setState({
    options: checked
  });
},

submitHandler: function (event) {
  event.preventDefault();
  alert(this.state.options);
},

render: function () {
  return <form onSubmit={this.submitHandler}>
    <select multiple="true"
      value={this.state.options}
      onChange={this.handleChange}>
      <option value="A">First Option</option>
      <option value="B">Second Option</option>
      <option value="C">Third Option</option>
    </select>
    <br />
    <button type="submit">Speak</button>
  </form>;
}
});
```

---

## 9.6 체크박스와 radio 버튼

checkbox와 radio 버튼은 조작 방법이 서로 다르다.

HTML <input/>의 type에 체크박스를 입력했을 때와 radio를 입력했을 때 다르게 동작한다. 보통 체크박스나 radio의 value 값은 잘 바뀌지 않는다. checked 속성이 바뀔 뿐이다. 체크박스나 radio 버튼은 checked 속성을 이용해서 제어한다. 또한, 비제어 체크박스나 radio 버튼은 defaultChecked를 가질 수 있다.

---

```
// 비제어 컴포넌트 - http://jsfiddle.net/es83ydmn/
var MyForm = React.createClass({
    submitHandler: function (event) {
        event.preventDefault();
        alert(this.refs.checked.getDOMNode().checked);
    },
    render: function () {
        return <form onSubmit={this.submitHandler}>
            <input
                ref="checked"
                type="checkbox"
                value="A"
                defaultChecked="true" />
            <br />
            <button type="submit">Speak</button>
        </form>;
    }
});

// 제어 컴포넌트 - http://jsfiddle.net/L8brrj25/
var MyForm = React.createClass({
    getInitialState: function () {
        return {
            checked: true
        };
    },
});
```

```
handleChange: function (event) {
  this.setState({
    checked: event.target.checked
  });
},

submitHandler: function (event) {
  event.preventDefault();
  alert(this.state.checked);
},

render: function () {
  return <form onSubmit={this.submitHandler}>
    <input
      type="checkbox"
      value="A"
      checked={this.state.checked}
      onChange={this.handleChange} />
    <br />
    <button type="submit">Speak</button>
  </form>;
}

});
```

---

앞 예제에서 <input/>의 value 값은 계속 A다. 바뀌는 것은 checked 상태뿐이다.

## 9.7 폼 엘리먼트 이터

제어 폼 엘리먼트가 state에 value 값을 저장하고 있고, 폼 submit 이벤트를 가로채고 있다면 폼의 name 속성은 별로 중요하지 않다. name을 몰라도 value 값에 접근할 수 있다. 비제어 폼 엘리먼트라면 ref를 이용해서 직접 폼 엘리먼트에 접근할 수도 있다.

그렇다하더라도 name은 폼 컴포넌트에 중요하다.

- name 속성을 이용하면 폼을 문자열로 만드는 외부 라이브러리를 React와 함께 사용할

수 있다.

- 브라우저의 기본 폼 전송 방식을 사용하는 경우 name 속성이 필요하다.
- 브라우저는 name 속성을 기준으로 사용자 주소 등을 입력할 때 자동 완성을 제공하기도 한다.
- 같은 name을 갖는 radio 버튼은 같은 시점에 오직 하나만 체크할 수 있게 그룹화할 수 있어야 한다. 이런 점에서 name 속성은 비제어 radio 버튼을 사용할 때 중요하다. 제어 radio 버튼은 name 속성을 주어서 그룹화하지 않아도 이 기능을 제공한다.

다음 예제는 MyForm 컴포넌트에 상태를 보관함으로써 비제어 radio 그룹을 복제 한다. name 속성을 사용하지 않고 있다.

---

```
//http://jsfiddle.net/8qzu1eos/
var MyForm = React.createClass({
  getInitialState: function () {
    return {
      radio: "B"
    };
  },
  handleChange: function (event) {
    this.setState({
      radio: event.target.value
    });
  },
  submitHandler: function (event) {
    event.preventDefault();
    alert(this.state.radio);
  },
  render: function () {
    return <form onSubmit={this.submitHandler}>
      <input
        type="radio"
        value="A"
        checked={this.state.radio == "A"}
        onChange={this.handleChange} /> A
      <br />
    </form>
  }
});
```

```
<input  
    type="radio"  
    value="B"  
    checked={this.state.radio == "B"}  
    onChange={this.handleChange} /> B  
<br />  
<input  
    type="radio"  
    value="C"  
    checked={this.state.radio == "C"}  
    onChange={this.handleChange} /> C  
<br />  
    <button type="submit">Speak</button>  
</form>;  
}  
});
```

---

## 9.8 여러 개의 폼 엘리먼트에 change 핸들러 사용

제어 폼 엘리먼트를 사용할 때 모든 폼 엘리먼트에 change 핸들러를 작성하고 싶지 않을 수 있다. 다행히 change 핸들러를 재사용하는 몇 가지 방법이 있다.

첫 번째 방법은 .bind에 추가 인자를 전달한다.

---

```
//http://jsfiddle.net/wyzvLhkb/  
var MyForm = React.createClass({  
  getInitialState: function () {  
    return {  
      given_name: "",  
      family_name: ""  
    };  
  },  
  
  handleChange: function (name, event) {  
    var newState = {};  
    newState[name] = event.target.value;  
    this.setState(newState);  
  },
```

```

    },
    submitHandler: function (event) {
        event.preventDefault();
        var words = [
            "Hi",
            this.state.given_name,
            this.state.family_name
        ];
        alert(words.join(" "));
    },
    render: function () {
        return <form onSubmit={this.submitHandler}>
            <label htmlFor="given_name">Given Name:</label>
            <br />
            <input
                type="text"
                name="given_name"
                value={this.state.given_name}
                onChange={this.handleChange.bind(this,'given_name')} />
            <br />
            <label htmlFor="family_name">Family Name:</label>
            <br />
            <input
                type="text"
                name="family_name"
                value={this.state.family_name}
                onChange={this.handleChange.bind(this,'family_name')} />
            <br />
            <button type="submit">Speak</button>
        </form>;
    }
);

```

---

두 번째 방법은 DOMNode의 name 값을 이용해서 변경할 상태를 확인한다.

---

```
//http://jsfiddle.net/q3g0sk84/
var MyForm = React.createClass({
```

```

getInitialState: function () {
  return {
    given_name: "",
    family_name: ""
  };
},

handleChange: function (event) {
  var newState = {};
  newState[event.target.name] = event.target.value;
  this.setState(newState);
},

submitHandler: function (event) {
  event.preventDefault();
  var words = [
    "Hi",
    this.state.given_name,
    this.state.family_name
  ];
  alert(words.join(" "));
},

render: function () {
  return <form onSubmit={this.submitHandler}>
    <label htmlFor="given_name">Given Name:</label>
    <br />
    <input
      type="text"
      name="given_name"
      value={this.state.given_name}
      onChange={this.handleChange}/>
    <br />
    <label htmlFor="family_name">Family Name:</label>
    <br />
    <input
      type="text"
      name="family_name"
      value={this.state.family_name}
      onChange={this.handleChange}/>

```

여러 개의 폼 요소와 change 핸들러

```
<br />
<button type="submit">Speak</button>
</form>;
}
});
```

---

앞의 두 예제는 매우 비슷하지만, 같은 문제를 다른 방법으로 해결한다. React가 제공하는 `React.addons.LinkedStateMixin`을 사용할 수도 있다. 이 믹스인은 또 다른 방법으로 문제를 해결한다.

`React.addons.LinkedStateMixin`은 `linkState` 메소드를 컴포넌트에 추가한다. `linkState` 메소드는 `value`와 `requestChange` 두 가지 프로퍼티를 가진 객체를 반환한다.

`value`는 `state`가 가지고 있는 `name`에 해당하는 값을 갖는다. `requestChange`는 새로운 `value` 값에 따라 상태를 갱신하는 함수다.

---

```
this.linkState('given_name');
//다음 객체를 반환한다.
{
  value: this.state.given_name,
  requestChange: function (newValue) {
    this.setState( { given_name: newValue } );
  }
}
```

---

React의 특수한 속성인 `valueLink`에 이 객체를 전달해야 한다. `valueLink`는 전달받은 객체의 `value` 값으로 `input` 엘리먼트의 `value` 값을 변경하며, 새로운 `DOMNode` 값을 가진 `onChange` 핸들러를 제공한다. `input`에 변경이 생기면 `onChange` 핸들러가 `requestChange` 메소드를 호출해서 `value`를 갱신한다.<sup>02</sup>

---

<sup>02</sup> 역사주\_ 쉽게 표현하면 `React.addons.LinkedStateMixin`은 품 엘리먼트의 `value`와 React 컴포넌트의 `state` 값을 바인딩 하여 동기화한다.

---

```
//http://jsfiddle.net/be5e5oqt/
var MyForm = React.createClass({
  mixins: [React.addons.LinkedStateMixin],
  getInitialState: function () {
    return {
      given_name: "",
      family_name: ""
    };
  },
  submitHandler: function (event) {
    event.preventDefault();
    var words = [
      "Hi",
      this.state.given_name,
      this.state.family_name
    ];
    alert(words.join(" "));
  },
  render: function () {
    return <form onSubmit={this.submitHandler}>
      <label htmlFor="given_name">Given Name:</label>
      <br />
      <input
        type="text"
        name="given_name"
        valueLink={this.linkState('given_name')} />
      <br />
      <label htmlFor="family_name">Family Name:</label>
      <br />
      <input
        type="text"
        name="family_name"
        valueLink={this.linkState('family_name')} />
      <br />
      <button type="submit">Speak</button>
    </form>;
  }
});
```

---

이렇게 하면 input 조작이 훨씬 쉬워지고, value 값을 부모 컴포넌트의 state에 저장할 수 있다. 데이터 흐름도 제어 폼 엘리먼트와 같은 방식으로 유지할 수 있다.

하지만 이 방법을 사용하면 기존 데이터 흐름에 새로운 기능을 추가하기가 어렵다. 이 믹스인은 제한적으로 사용하는 것이 좋다. 믹스인을 사용하는 것보다 제어 폼 엘리먼트를 사용하는 방법이 더욱 유연하다.

## 9.9 커스텀 폼 컴포넌트

커스텀 폼 컴포넌트를 만들면 애플리케이션에서 자주 사용하는 기능을 재사용할 수 있다. 체크박스나 radio 버튼처럼 복잡한 폼 컴포넌트의 인터페이스를 개선하는 좋은 방법이다.

커스텀 폼 컴포넌트를 만들 때는 다른 폼 컴포넌트와 같은 인터페이스를 갖추어야 한다. 이렇게 하면 코드의 예측 가능성을 높일 수 있으며, 구현 내용을 몰라도 컴포넌트의 동작 과정을 쉽게 이해할 수 있다.

React 셀렉트 컴포넌트와 인터페이스가 같은 커스텀 radio 버튼 컴포넌트를 만들어보자. radio 버튼이기 때문에 다중 선택 기능은 제공하지 않는다.

---

```
var Radio = React.createClass({
  propTypes: {
    onChange: React.PropTypes.func
  },
  getInitialState: function () {
    return {
      value: this.props.defaultValue
    };
  },
  handleChange: function (event) {
    if (this.props.onChange) {
```

```

        this.props.onChange(event);
    }
    this.setState({
        value: event.target.value
    });
},
render: function () {
    var children = {};
    var value = this.props.value || this.state.value;

    React.Children.forEach(this.props.children, function
        (child, i) {
        var label = <label>
            <input
                type="radio"
                name={this.props.name}
                value={child.props.value}
                checked={child.props.value == value}
                onChange={this.handleChange} />
            {child.props.children}
        <br/>
        </label>;
        children['label' + i] = label;
    }.bind(this));
}

return this.transferPropsTo(<span>{children}</span>);
}
});

```

---

기본적으로 이 컴포넌트는 제어 컴포넌트지만, 제어와 비제어 컴포넌트의 인터페이스를 모두 지원한다.

먼저 onChange가 함수를 전달받는지 확인한다. 그리고 기본값을 상태에 저장한다.

React는 컴포넌트를 렌더링할 때마다 자식 컴포넌트에게 전달한 옵션을 가지고 새로운 label과 radio 버튼을 만든다. 또한, 동적으로 만든 자식 컴포넌트를 렌

더링할 때는 일관성 있는 키값을 부여해야 한다. 그래야 키보드 컨트롤을 사용할 때 React가 <input/>을 DOM에 둔 채 포커스를 유지할 수 있다.

다음에는 value, name, checked 상태를 설정한다. 이어서 onChange 핸들러를 연결하고, 새로운 자식 컴포넌트를 렌더링하면 끝이다.

---

```
// 비제어 컴포넌트
// http://jsfiddle.net/moyfLkfV/
var MyForm = React.createClass({
  submitHandler: function (event) {
    event.preventDefault();
    alert(this.refs.radio.state.value);
  },
  render: function () {
    return <form onSubmit={this.submitHandler}>
      <Radio ref="radio" name="my_radio" defaultValue="B">
        <option value="A">First Option</option>
        <option value="B">Second Option</option>
        <option value="C">Third Option</option>
      </Radio>
      <button type="submit">Speak</button>
    </form>;
  }
});
```

---

비제어 컴포넌트를 사용할 때는 인터페이스를 약간 수정해야 한다. this.refs.radio에서 getDOMNode를 사용하면 <input/>이 아닌 실제 DOM 노드를 가져올 수 있다. getDOMNode()의 기능을 변경해서 이 동작을 덮어쓸 수는 없다.

value 값을 컴포넌트의 state 객체에 저장하면, DOM 노드에 접근하지 않고 직접 state 객체에 접근해서 현재 value 값을 확인할 수 있다.

---

```
// 제어 컴포넌트
// http://jsfiddle.net/cwabLksg/
var MyForm = React.createClass({
```

```
getInitialState: function () {
  return {my_radio: "B"};
},
handleChange: function (event) {
  this.setState({
    my_radio: event.target.value
  );
},
submitHandler: function (event) {
  event.preventDefault();
  alert(this.state.my_radio);
},
render: function () {
  return <form onSubmit={this.submitHandler}>
    <Radio name="my_radio"
      value={this.state.my_radio}
      onChange={this.handleChange}>
      <option value="A">First Option</option>
      <option value="B">Second Option</option>
      <option value="C">Third Option</option>
    </Radio>
    <button type="submit">Speak</button>
  </form>;
}
});
```

---

앞 예제는 셀렉트 박스와 같이 동작한다. onChange가 전달받는 이벤트는 실제 <input/>에서 가져온 이벤트로 현재 value 값을 확인하는 데 사용할 수 있다.

조금 더 연습해보고 싶다면, React.addons.LinkedStateMixin을 사용해서 컴포넌트가 valueLink 속성을 지원하게 하여 보자.

## 9.10 포커스

폼 컴포넌트의 포커스를 조작하면 사용자가 폼을 논리적으로 작성하게 유도할 수 있다. 또 인터렉션을 줄여 사용성을 높일 수 있다. 이 방법의 장점은 다음 절에서 더 자세히 다룬다.

React 폼은 항상 브라우저 로딩 시점에 렌더링 되지 않기 때문에 `input` 폼 자동 포커스를 약간 다르게 처리해야 한다. React는 `autoFocus`를 구현했다. 컴포넌트를 처음 렌더링할 때 다른 `input`에 포커스가 없으면 해당 `input`으로 포커스를 이동한다. 이렇게 하면 간단하게 HTML 폼에 자동 포커스를 적용할 수 있다.

---

```
//JSX
<input type="text" name="given_name" autoFocus="true" />
```

---

DOM 노드에 `focus()`를 호출하여 수동으로 포커스를 이동할 수도 있다.

## 9.11 사용성

React는 높은 생산성을 제공하지만, 단점도 가지고 있다.

아주 쉽게 컴포넌트의 사용성을 떨어뜨릴 수 있다. 키보드 조작을 제한해서 링크의 `onClick`만으로 전송할 수 있는 폼이 있다고 생각해보자. 기본적으로 폼은 엔터 키를 눌러서 전송할 수 있기 때문에, 이렇게 만든 폼은 일반적인 폼에 비해 사용성이 떨어진다.

반대로 아주 쉽게 사용성을 높일 수도 있다. 컴포넌트를 만들 때는 시간을 들여서 고민해야 한다. 사용성이 좋은 컴포넌트를 만드는 과정은 사소한 작업의 연속이다.

사용성이 좋은 폼을 만드는 데 참고할 몇 가지 실천 사례가 있다. React가 아닌 다른 경우에도 적용할 수 있다.

## 요구사항을 명확하게 제시하라

좋은 커뮤니케이션은 애플리케이션의 모든 측면에서 중요하지만 폼의 경우에는 특히 더 중요하다.

HTML 레이블은 사용자가 폼 엘리먼트를 사용할 때 우리가 기대하는 걸 전달하는데 사용할 수 있는 최고의 커뮤니케이션 수단이다. 또한, 레이블은 사용자가 라디오 버튼이나 체크박스 같은 폼 엘리먼트와 상호작용 하는 방법을 추가로 제공한다.

플레이스홀더<sup>Placeholder</sup>는 입력 예시나 기본으로 사용할 값을 보여줄 때 사용한다. 폼에 입력할 수 있는 적절한 값을 가이드하기 위해 플레이스홀더를 사용하는 게 유행하기도 했다. 입력을 시작하면 이 내용은 사라져버리기 때문에 다소 문제가 될 수 있다. 유효성 검사 등과 관련 있는 부가적인 설명은 `input`의 주변에 노출하거나 팝오버<sup>popover</sup>를 이용하는 것이 더 나은 방법이다.

## 지속해서 피드백을 제공하라

이는 입력 명확한 입력 요구사항 전달과 함께 고려해야 한다. 피드백은 사용자에게 가능한 한 빠르게 전달해야 한다.

유효성 검사 후 에러를 보여주는 것은 피드백 제공의 좋은 예이다. 유효성 에러를 보여줌으로써 폼의 사용성을 높일 수 있다. 예전에는 사용자가 폼에 내용을 모두 입력하고 나서야 에러를 확인할 수 있었다. 브라우저에서 행하는 유효성 검사는 사용성을 비약적으로 높였다. `Input`이 포커스를 잃는 `blur` 시점이 유효성 에러를 사용자에게 알려주기 가장 좋은 때다.

또한, 사용자의 요청에 응답하고 있다는 사실을 알려주는 것도 중요하다. 특히 완료하기까지 어느 정도 시간이 걸리는지 예측할 수 있는 동작일 때가 그렇다. 로딩, 진행 그래프, 알림 메시지 등을 이용하면 애플리케이션이 동작을 멈추지 않았다는 사실을 효과적으로 전달할 수 있다. 사용자는 동작이 바로 수행되지 않으면 쉽게 짜증을 내기도 하는데, 적절한 피드백을 줘서 사용자의 인내심을 끌어낼 수 있다.

트랜지션과 애니메이션을 사용하는 것도 좋은 방법이다. 이런 시각적인 도구를 이용하면 애플리케이션이 변경 중이라는 사실을 사용자에게 알려줄 수 있다. React를 이용해서 애니메이션과 트랜지션<sup>transition</sup>을 처리하는 방법에 대해서는 10장에서 이야기한다.

### 속도에 신경을 써라

React는 매우 강력한 렌더링 엔진을 가지고 있다. React를 이용해서 애플리케이션의 속도를 아주 빠르게 만들 수 있다. 하지만 DOM 갱신이 아닌 다른 요인이 애플리케이션의 속도를 떨어뜨리는 경우도 흔하다.

트랜지션이 대표적인 사례이다. 애플리케이션 사용을 위해 기나긴 트랜지션이 끝날 때까지 기다려야 한다면 좋아할 사용자는 아마 별로 없을 것이다.

애플리케이션 외부의 요소들이 문제를 일으킬 수도 있다. AJAX 호출이 길게 지속되거나 네트워크 성능이 좋지 못하다면, 애플리케이션의 뛰어난 성능은 의미가 없다. 이런 이슈를 해결하기 위해서는 해당 애플리케이션에만 해당하는 특정한 방법을 이용하거나, 때로는 외부 서비스를 이용하기도 한다. 해결이 어려울수록 사용자에게 적절한 피드백을 제공해서 사용자 요청의 진행상태를 보여주어야 한다.

속도는 상대적이라는 사실을 기억하라. 속도는 사용자의 인식에 달려있다. 빠른 것보다 빠르게 보이는 것이 더 중요하다. 예를 들어 사용자가 애플리케이션에서 “좋아요”를 눌렀을 때, 서버에 AJAX 호출을 보내기 전에 숫자를 증가시킬 수 있다. 이렇게 하면 AJAX 호출에 긴 시간이 소요되더라도 사용자는 이런 사실을 알 수 없을 것이다. 그렇지만 이 방법은 다른 문제를 만들기도 한다.

### 예측할 수 있게 만들어라

사용자가 애플리케이션의 작동 방식을 파악할 때 가장 큰 영향을 미치는 것은 경험이다. 대부분 이런 경험은 당신이 만든 애플리케이션을 사용하면서 얻은 것이

아니다.

애플리케이션이 사용자가 이용하는 플랫폼과 유사하다면 사용자는 플랫폼이 제공하는 기본 동작이 애플리케이션에도 그대로 적용될 것이라고 기대한다.

이런 점으로 생각해 보면 선택할 방법은 크게 두 가지가 있다. 하나는 플랫폼이 제공하는 방식을 그대로 따르는 것이고, 다른 하나는 사용자 인터페이스를 근본적으로 변경해서 플랫폼의 방식을 전혀 따르지 않는 것이다.

일관성은 또 다른 형태의 예측 가능성이다. 애플리케이션의 다른 부분에도 같은 사용자 인터랙션을 제공한다면 사용자는 사용법을 쉽게 습득할 수 있을 것이다. 플랫폼이 제공하는 방식을 따를 때의 장점이기도 하다.

### 접근성을 높여라

개발자나 디자이너가 사용자 인터페이스를 만들 때 접근성을 간과하는 경우가 자주 있다. 사용자 인터페이스의 어떤 면이든 사용자를 중심으로 생각해야 한다. 앞에서도 살펴본 것처럼 사용자는 기존의 경험을 바탕으로 동작 방식을 예측한다.

여러 가지 입력 형태에 대한 선호도 역시 과거의 경험이 기준이 된다. 키보드나 마우스 같은 입력 장치를 사용하는 데 사용자가 신체적 어려움을 겪고 있을 수 있다. 또는 디스플레이이나 스피커 같은 출력장치를 다루는 데 어려움을 겪는 경우도 있다.

여러 입출력 장치에 적합한 기능을 애플리케이션의 모든 측면에서 지원하기란 쉽지 않은 일이다. 사용자 요구사항과 우선순위를 이해하고 이를 기준으로 접근성을 개선해 나가자.

키보드, 마우스, 터치스크린 중 하나의 입력장치만을 이용해서 애플리케이션을 사용해보면 좋다. 장치가 가진 사용성 문제를 더 잘 알 수 있다.

시각장애가 있는 사용자가 애플리케이션을 사용하는 과정도 고려해야 한다. 이런 경우에는 스크린 리더가 두 눈이 된다.

HTML5의 리치 인터넷 애플리케이션 접근성<sup>ARIA, Accessible Rich Internet Applications</sup> 명세는 스크린 리더 같은 보조 기술을 위해 의미론적으로 부족한 부분을 보완할 수 있는 수단을 제공한다. 이 명세를 이용하면 스크린 리더를 이용할 때 UI 컴포넌트의 역할을 알려주고, 엘리먼트를 감추거나 보여줄 수 있다.

Google Chrome의 접근성 개발자 도구는 애플리케이션의 접근성 개선에 활용할 수 있는 좋은 도구이다.

### 사용자 입력은 적을수록 좋다

사용자가 입력해야 하는 양을 줄이면 사용성도 높일 수 있다. 사용자가 입력해야 하는 정보가 적을수록 사용자가 실수할 가능성은 작아지며 생각하는 시간 역시 짧아진다.

앞에서도 설명했지만, 사용자의 지각은 중요하다. 사용자는 입력할 항목이 많은 큰 양식에 쉽게 적응하지 못한다. 항목을 줄이고 관리하기 편한 둑음으로 분리하여 입력할 양이 적다는 것을 사용자가 느끼게 해야 한다. 이렇게 하면 사용자는 데이터 입력에 좀 더 집중할 수 있다.

자동입력은 데이터 입력을 줄이는 좋은 방법이다. 브라우저의 자동입력은 사용자가 자신의 주소나 결제정보 같은 일상적인 정보를 다시 작성하는 수고를 덜어준다. 따라서 `input`에 표준 `name` 값을 사용하는 것이 좋다.

자동완성은 사용자가 정보를 입력할 때 가이드가 되기 때문에 지속적인 피드백과 같은 맥락으로 생각해볼 수 있다. 예를 들어 영화를 검색할 때 자동완성 덕분에 더욱 쉽게 영화 이름을 작성할 수 있다.

사용자 입력을 줄이는 또 다른 방법은 기준에 입력한 데이터를 활용하는 것이다. 예를 들어 사용자가 신용카드 정보를 입력한다면 첫 4자리를 기준으로 어떤 카드인지 판단하고 적절한 카드 종류를 선택할 수 있다. 입력할 양을 줄이는 동시에 사

용자가 작성한 카드번호의 유효성 검사 결과를 알려줄 수 있다.

자동포커스는 간단하지만, 입력 양식의 사용성을 높이는 데 아주 효과적이다. 자동포커스는 데이터 작성의 시작점을 알려준다. 사용자가 마음대로 작성하는 것을 막아준다. 이 방법은 간단하게 보이지만 사용자가 데이터 입력을 빠르게 시작할 수 있게 도와준다.

## 9.12 정리

React는 DOM이 아닌 컴포넌트에서 상태를 관리한다. 폼 요소를 좀 더 세밀하게 조작하고, 애플리케이션에 사용할 복잡한 컴포넌트를 만들 수 있다.

폼은 애플리케이션에서 사용자와의 인터랙션이 가장 복잡한 부분이다. 폼 컴포넌트를 만들고 결합할 때는 폼의 사용성에 항상 주의를 기울여야 한다.

다음 장에서는 React 컴포넌트에 애니메이션을 더해서 애플리케이션을 더욱 매력 있게 만든다.



# 애니메이션

이제 복잡한 React 컴포넌트를 조합하는 방법을 알았으니 반짝반짝 광을 낼 차례다. 애니메이션은 더욱 유연하고 자연스러운 사용자 경험을 만든다. React의 트랜지션 그룹 `Transition Groups` addon과 CSS3를 조합하면 React 프로젝트에 쉽게 애니메이션을 적용할 수 있다.

그동안 접해 온 브라우저 애니메이션은 명령형<sup>imperative</sup> API다. 엘리먼트를 이동시키거나 스타일을 변경해서 애니메이션을 만드는 방식이다. 이 방법은 React의 렌더링 방식에 어울리지 않는다. React는 대신 서술형<sup>declarative</sup> 방식을 취한다.

CSS 트랜지션 그룹을 이용하면 CSS 애니메이션을 이용해서 컴포넌트에 전환 효과를 줄 수 있다. 렌더링과 재 렌더링 진행 중 적절한 시점에 클래스를 추가/제거하는 전략을 사용한다. 따라서 각 클래스에 사용할 CSS만 정의하면 끝이다.

인터벌 렌더링<sup>Interval Rendering</sup>을 이용하면 성능을 희생해야 하지만 더욱 유연하고 세밀하게 애니메이션을 제어할 수 있다. 이 방법을 이용하면 컴포넌트에 훨씬 많은 렌더링이 일어나지만, 단순한 CSS 변경을 넘어 스크롤 위치나 Canvas 그리기 같은 더 많은 애니메이션 효과를 만들 수 있다.

## 10.1 CSS 트랜지션 그룹

설문조사 생성기 예제 중 설문 편집기가 질문 목록을 렌더링하는 과정을 살펴보자.

---

```
<ReactCSSTransitionGroup transitionName='question'>
  {questions}
</ReactCSSTransitionGroup>
```

---

ReactCSSTransitionGroup은 애드온으로 파일의 최상단에 있는 var ReactCSSTransitionGroup = React.addons.CSSTransitionGroup에서 선언하고 있다.

이 애드온은 컴포넌트를 적절한 시점에 다시 렌더링하고, 트랜지션 그룹의 클래스를 변경해서 전환 효과에 맞는 스타일을 적용한다.

**NOTE**

예제로 사용하는 설문조사 생성기의 전체 소스 코드는 Github 저장소에서 확인할 수 있다.

<https://github.com/backstopmedia/bleeding-edge-sample-app>

### 트랜지션 클래스 스타일 적용하기

transitionName='question' 속성은 요소를 네 개의 CSS 클래스에 연결한다. 컨벤션에 따라 question-enter, question-enter-active, question-leave, question-leave-active, 모두 네 개의 클래스 명을 작성한다. CSSTransitionGroup 애드온은 자식 컴포넌트가 ReactCSSTransitionGroup 들어가고 나옴에 따라서 자동으로 클래스를 추가하거나 삭제한다.

설문 편집기가 사용하는 트랜지션 스타일은 다음과 같다.

---

```
.survey-editor .question-enter {
  transform: scale(1.2);
  transition: transform 0.2s cubic-bezier(.97,.84,.5,1.21);
}
.survey-editor .question-enter-active {
  transform: scale(1);
```

```
}

.survey-editor .question-leave {
    transform: translateY(0);
    opacity: 0;
    transition: opacity 1.2s, transform 1s cubic-bezier(.52,-0.25,.52,.95);
}

.survey-editor .question-leave-active {
    opacity: 0;
    transform: translateY(-100%);
}
```

---

.survey-editor 선택자는 CSSTransitionGroup 사용을 위해 꼭 필요한 것은 아니며, CSS를 작성한 목적이 설문 편집기에 사용하기 위한 것임을 명확하게 밝히는 용도로 사용했다.

## 트랜지션 라이프사이클

question-enter 클래스는 컴포넌트가 트랜지션 그룹에 추가되자마자 적용되고, question-enter-active 클래스는 다음 차례에 적용된다. 이 덕분에 전환 애니메이션의 시작과 끝, 그리고 중간 과정을 쉽게 정의할 수 있다.

예를 들어 설문 편집기에서 목록에 질문을 추가하면, 질문은 1.2배로 커졌다가 0.2초에 걸쳐 원래의 크기로 돌아온다.

기본적으로 트랜지션 그룹은 시작과 마지막 애니메이션을 모두 사용한다. 이 기능을 해제하고 싶다면 컴포넌트를 선언할 때 transitionEnter={false}나 transitionLeave={false}를 전달한다. 또한, 다음 예제처럼 설정값을 이용해서 상황에 따라 애니메이션을 사용 여부를 지정할 수도 있다.

---

```
<ReactCSSTransitionGroup transitionName='question'
    transitionEnter={this.props.enableAnimations}
    transitionLeave={this.props.enableAnimations}>
    {questions}
</ReactCSSTransitionGroup>
```

---

## 10.2 트랜지션 그룹 사용 시 주의점

트랜지션 그룹을 사용할 때는 다음 두 가지를 유의한다.

첫째, 트랜지션 그룹은 애니메이션이 끝나야만 자식 컴포넌트를 제거한다. 컴포넌트 여러 개를 트랜지션 그룹의 자식으로 추가해놓고서 `transitionName` 클래스에 CSS를 명시하지 않으면 자식 컴포넌트는 삭제되지 않는다. 심지어 렌더링이다 끝나도 자식 컴포넌트는 그대로 존재한다.

둘째, 트랜지션 그룹의 자식 컴포넌트는 고유의 `key`를 가져야 한다. 트랜지션 그룹은 `key` 속성을 이용해 컴포넌트의 애니메이션 적용 시점을 결정하기 때문에 `key` 속성이 없으면 애니메이션을 적용하지 할 수 없으며 컴포넌트를 제거하지 못할 수도 있다.

트랜지션 그룹의 자식 컴포넌트가 하나뿐이라도 반드시 `key` 속성을 가져야 한다.

## 10.3 인터벌 렌더링

CSS3 애니메이션은 성능이 좋고 코드가 간결하지만, 사용할 수 없는 경우가 있다. CSS3를 지원하지 않는 구형 브라우저 사용을 지원해야 할 때 그렇다. 이 외에 CSS 속성이 아닌 스크롤 위치나 캔버스에 애니메이션을 처리해야 할 때도 있다. 이런 경우에는 인터벌 렌더링 방식을 사용하는 것을 생각해 볼 수 있는데, CSS3 애니메이션에 비해 성능은 좋지 않다.

인터벌 렌더링은 애니메이션의 진행 정도에 따라 컴포넌트의 상태를 변경하는 것이 기본이다. 상태 값을 컴포넌트의 렌더링 메소드에 전달하면 상태 변화에 따라 컴포넌트를 다시 렌더링하면서 적절한 애니메이션을 만든다.

이 방법을 이용하면 렌더링이 여러 번 발생한다. 불필요한 렌더링을 막고 싶다면 `requestAnimationFrame`을 함께 사용하는 것이 좋다. 만약

`requestAnimationFrame`를 사용할 수 없는 경우에는 `setTimeout`을 사용하는 것이 유일한 대안이다. `setTimeout`은 `requireAnimationFrame`에 비해 정확한 작동 시점을 예측하기 어려운 것이 단점이다.

### `requestAnimationFrame`을 사용한 인터벌 렌더링

인터벌 렌더링을 이용해 화면을 가로질러 이동하는 div를 만들어보자. div에 `position: absolute` 속성을 주고 시간에 따라 `left` 또는 `top` 속성을 변경해서 이동하는 것처럼 보이게 만든다. `requestAnimationFrame`을 이용해서 경과 시간에 따라 div의 위치를 옮기면 애니메이션을 부드럽게 처리할 수 있다.

다음 코드는 구현 예를 보여준다.

---

```
var Positioner = React.createClass({
  getInitialState: function() {
    return { position: 0 };
  },
  resolveAnimationFrame: function() {
    var timestamp = new Date();
    var timeRemaining = Math.max(0, this.props.animationCompleteTimestamp
      - timestamp);
    if (timeRemaining > 0) {
      this.setState({ position: timeRemaining });
    }
  },
  componentWillMount: function() {
    if (this.props.animationCompleteTimestamp) {
      requestAnimationFrame(this.resolveAnimationFrame);
    }
  },
  render: function() {
    var divStyle = { left: this.state.position };

    return <div style={divStyle}>This will animate!</div>
  }
});
```

---

이 예제는 컴포넌트의 props에 animationCompleteTimestamp 값을 보관한다. 이 값과 requestAnimationFrame의 콜백이 반환하는 타임스탬프 값을 이용해서 남은 이동거리를 계산한다. 계산 결과는 this.state.position에 저장해서 div의 위치로 이용한다.

컴포넌트의 속성(animationCompleteTimestamp 같은)에 변화가 있을 때마다 componentWillUpdate 핸들러는 requestAnimationFrame를 실행한다. resolveAnimationFrame가 this.setState를 호출하는 경우도 마찬가지다. 즉, 애니메이션 종료 시점인 animationCompleteTimestamp를 설정하면 자동으로 이 시간만큼 requestAnimationFrame을 반복 실행한다.

앞의 코드는 타임스탬프를 이용하는 경우만 보여주고 있을 뿐이다. animationCompleteTimestamp를 설정하면 애니메이션이 시작되며, 현재 시점의 타임스탬프와 animationCompleteTimestamp의 차이를 계산해서 this.state.position 값을 결정한다. 이와 같은 방법으로 렌더링 메소드 안에서 스크롤 위치를 조작하거나 캔버스에 뭔가를 그리는 등의 애니메이션 처리도 원한다면 얼마든지 할 수 있다.

### setTimeout을 사용한 인터벌 렌더링

requestAnimationFrame을 이용하면 성능을 최소한으로 희생하여 부드러운 애니메이션을 만들 수 있지만, 단점도 있다. 구형 브라우저에서 사용할 수 없을 때가 있으며, 원하는 것보다 더 자주 함수가 호출되는(또는 호출 간격을 예측하기 어려운) 경우도 있다. 이럴 때는 setTimeout을 사용한다.

---

```
var Positioner = React.createClass({
  getInitialState: function() { return {position: 0}; },

  resolveSetTimeout: function() {
    var timestamp = new Date();
    var timeRemaining = Math.max(0, this.props.animationCompleteTimestamp - timestamp);
    this.setState({position: this.state.position + timeRemaining});
  }
});
```

```
- timestamp);
    if (timeRemaining > 0) {
        this.setState({position: timeRemaining});
    }
},
componentWillUpdate: function() {
    if (this.props.animationCompleteTimestamp) {
        setTimeout(this.resolveSetTimeout, this.props.timeoutMs);
    }
},
render: function() {
    var divStyle = {left: this.state.position};
    return <div style={divStyle}>This will animate!</div>
}
});
```

---

`requestAnimationFrame` 간격을 알아서 설정하는 것과 달리 `setTimeout`은 명확한 호출 간격을 지정해야 한다. 따라서 이 간격을 설정하는 `this.props.timeoutMs`를 추가로 이용한다.

오픈 소스인 React Tween State 라이브러리는 이러한 애니메이션을 구현할 수 있는 편리한 추상 레이어를 제공한다.

## 10.4 정리

지금까지 살펴본 애니메이션 기술로 다음 작업을 할 수 있다.

1. CSS3와 트랜지션 그룹을 이용해 상태에 맞는 효율적인 애니메이션을 구현 할 수 있다.
2. `requestAnimationFrame`을 이용하여 CSS가 아닌 스크롤 위치 이동이나 캔버스 애니메이션을 구현할 수 있다.

3. requestAnimationFrame을 이용할 수 없는 상황에는 setTimeout을 사용할 수 있다.

다음 장에서는 React의 성능을 개선하는 방법을 알아본다.

# 성능 개선

React는 DOM의 diff를 비교해서 변경 작업을 수행한다. 이 덕분에 전체 UI를 제거할 때 DOM에 주는 영향을 최소화할 수 있다. 하지만 애플리케이션의 속도를 높이기 위해서 컴포넌트가 새로운 가상 표현 객체를 만드는 과정을 섭세하게 개선해야 할 때가 있다. 특히 깊은 중첩관계가 있는 컴포넌트 트리를 다뤄야 할 때가 그렇다. 이번 장에서는 컴포넌트에 간단한 설정을 적용해 애플리케이션의 속도를 높이는 방법을 살펴본다.

## 11.1 shouldComponentUpdate

새로운 props, setState, forceUpdate를 호출해서 컴포넌트를 변경하면 React는 해당 컴포넌트의 자식 컴포넌트가 가지고 있는 render 메소드를 실행한다. 대부분 이 과정은 성능에 영향을 미치지 않는다. 하지만 컴포넌트 트리의 중첩이 깊거나 복잡한 렌더링 과정을 거치는 페이지라면 이 과정이 성능에 영향을 미칠 수도 있다.

가끔 컴포넌트의 render 메소드를 불필요하게 실행하는 경우가 있다. 예를 들어 컴포넌트가 state나 props를 사용하지 않거나 부모 컴포넌트를 다시 렌더링하면서 props나 state를 변경하지 않는 경우가 있다. 이 경우에는 이미 존재하는 이전과 같은 가상 DOM 표현 객체를 컴포넌트가 굳이 다시 렌더링할 필요가 없다.

`shouldComponentUpdate`는 컴포넌트 라이프사이클 메소드를 이용하면 특정 컴포넌트의 `render` 메소드 실행 여부를 적절하게 결정할 수 있다.

`shouldComponentUpdate`는 불린 값을 반환한다. 이 메소드가 `false` 값을 반환하면 컴포넌트의 `render` 메소드를 실행하지 않고, 기존에 렌더링한 가상 표현 객체를 사용한다. `true` 값을 반환하면 컴포넌트의 `render` 메소드를 실행하고 새로운 가상 표현 객체를 만든다. 기본적으로 `shouldComponentUpdate`는 `true`를 반환하기 때문에 React는 컴포넌트를 항상 다시 렌더링한다.

컴포넌트를 처음 렌더링할 때는 `shouldComponentUpdate`를 호출하지 않는다는 점을 주의한다. `shouldComponentUpdate`는 새로운 `props`와 `state`를 인자로 받는다. 이 값을 이용해서 렌더링 여부를 결정할 수 있다.

---

```
var SurveyEditor = React.createClass({  
  shouldComponentUpdate: function(nextProps, nextState) {  
    return nextProps.id !== this.props.id;  
  }  
});
```

---

`props`와 `state`가 같을 경우 항상 같은 렌더링을 하는 순수한 컴포넌트라면 `React.addons.PureRenderMixin`을 이용할 수 있다.

**NOTE**

예제로 사용하는 설문조사 생성기의 전체 소스 코드는 Github 저장소에서 확인할 수 있다.

<https://github.com/backstopmedia/bleeding-edge-sample-app>

이 믹스인은 새로운 `props`와 `state`를 이전 값과 비교해서 값이 같다면 `false`를 반환하도록 `shouldComponentUpdate`를 덮어쓴다.

다음 예제의 `EditEssayQuestion`처럼 여기서 다루는 몇몇 컴포넌트는 간단하므

로 `React.addons.PureRenderMixin`를 사용할 수 있다.

---

```
var EditEssayQuestion = React.createClass({
  mixin: [React.addons.PureRenderMixin],

  render: function () {
    var description = this.props.question.description || "";
    return (
      <EditQuestion type='essay' onRemove={this.handleRemove}>
        <label>Description</label>
        <input type='text' className='description' value={description} onChange={this.handleChange} />
      </EditQuestion>
    );
  },
  // ...
});
```

---

`props`와 `state`가 깊고 복잡한 구조로 되어 있다면 비교하는 과정이 오래 걸릴 수도 있다. `Immutable.js` 같은 불변 데이터 구조를 사용하면 이 점을 보완할 수 있다. 이 부분은 React 패밀리나 Immutability Helpers 애드온을 다루면서 더 살펴본다.

## 11.2 Immutability Helpers 애드온

불변 데이터 구조를 사용하면 `shouldComponentUpdate`가 변경을 확인하기 위해 객체를 비교하는 과정을 더 간단하게 만들 수 있다.

다음 예제처럼 `<SurveyEditor/>` 컴포넌트의 `change` 핸들러 함수를 수정해서 `React.addons.update01`를 사용하면 컴포넌트의 불변성을 보장할 수 있다.

---

<sup>01</sup> 역사주\_ React v.0.14 버전부터는 `addons`가 별도의 패키지로 분리되었다. `React.addons.Update`는 `react-addons-update`를 이용한다.

---

```
var update = React.addons.update;
var SurveyEditor = React.createClass({
  // ...
  handleDrop: function (ev) {
    var questionType = ev.dataTransfer.getData('questionType');
    var questions = update(this.state.questions, {
      $push: [{ type: questionType }]
    });

    this.setState({
      questions: questions,
      dropZoneEntered: false
    });
  },
  handleQuestionChange: function (key, newQuestion) {
    var questions = update(this.state.questions, {
      $splice: [[key, 1, newQuestion]]
    });

    this.setState({ questions: questions });
  },
  handleQuestionRemove: function (key) {
    var questions = update(this.state.questions, {
      $splice: [[key, 1]]
    });

    this.setState({ questions: questions });
  }
});
```

---

React.addons.update는 데이터 객체와 옵션 해쉬 객체를 인자로 받는다.  
\$slice, \$push, \$unshift, \$set, \$merge, \$apply를 사용할 수 있다.

### 11.3 속도저하 원인 찾기

앞에서 언급한 것처럼 `shouldComponentUpdate`를 수정해서 애플리케이션을 최적화할 수 있다. `React.addons.Perf`를 사용하면 `shouldComponentUpdate`를 추가할 최적의 위치를 찾을 수 있다. 설문조사 생성기의 `<SurveyEditor/>` 페이지에서 `React.addons.Perf`를 사용해서 병목이 발생하는 구간을 찾아보자.

먼저 크롬 개발자 도구의 콘솔을 열고 `React.addons.Perf.start();`<sup>02</sup>를 입력한다. 이렇게 하면 스냅 샷을 시작한다. 질문을 몇 개 드래그하고 `React.addons.Perf.stop();`를 실행한다. 이어서 `React.addons.Perf.printWasted();`를 입력하면 다음과 같은 결과를 확인할 수 있다.

[그림 11-1]

> React.addons.Perf.printWasted()				
(index)	Owner > component	Wasted time (ms)	Instances	
0	"ReactTransitionGroup > SurveyEditor > DraggableQuestions > SurveyForm > ReactDOM"	51.088000007439405	86	ReactDefaultPerf.js:99
1	"SurveyEditor > DraggableQuestions > SurveyForm > ReactDOM"	19.280999898910522	28	ReactDefaultPerf.js:100
2	"SurveyEditor > SurveyForm > ReactDOM"	10.835000139195472	28	ReactDefaultPerf.js:101
3	"DraggableQuestions > SurveyForm > ReactDOM"	8.496999624185264	84	ReactDefaultPerf.js:102
4	"SurveyEditor > ReactCS > SurveyForm > ReactDOM"	7.958000060170889	6	ReactDefaultPerf.js:103
5	"AddSurvey > SurveyEditor > SurveyForm > ReactDOM"	3.349000005982816	1	ReactDefaultPerf.js:104
6	"SurveyEditor > Divider" > SurveyForm > ReactDOM	2.7780000236816704	28	ReactDefaultPerf.js:105
7	"EditMultipleChoiceQuestion > SurveyEditor > SurveyForm > ReactDOM"	2.5900001055561006	34	ReactDefaultPerf.js:106
8	"SurveyForm > ReactDOM" > SurveyEditor > SurveyForm > ReactDOM	2.0920000970363617	28	ReactDefaultPerf.js:107
9	"SurveyForm > ReactDOM" > SurveyEditor > SurveyForm > ReactDOM	1.808999918486923	28	ReactDefaultPerf.js:108
10	"SurveyEditor > SurveyForm > ReactDOM" > SurveyEditor > SurveyForm > ReactDOM	1.7349999397993088	28	ReactDefaultPerf.js:109

<ReactTransitionGroup/> 컴포넌트는 제어할 수 없으므로 할 수 있는 게 별로 없다. 그렇지만 <DraggableQuestions/>은 shouldComponentUpdate를 수정하면 큰 효과를 볼 수 있다. <DraggableQuestion/>은 실제로 아주 간단한 컴포넌트다. 변경할 가능성이 전혀 없는 구조로 되어 있다. shouldComponentUpdate 함수를 수정해서 항상 false를 반환하게 하자.

```
var DraggableQuestions = React.createClass({
  render: function () {
    return (
      <ul className="modules list-unstyled">
```

02 역자주 <https://facebook.github.io/react/docs/perf.html>

```

        <li>ModuleButton text='Yes / No' questionType='yes_no' /></li>
        <li>ModuleButton text='Multiple choice'
questionType='multiple_choice' /></li>
    </ul>
);
},
};

shouldComponentUpdate: function () {
    return false;
}
});

```

---

이제 목록에서 이 컴포넌트가 사라졌다.

[그림 11-2]

Perf.printWasted()			
(index)	Owner > component	Wasted time (ms)	Instances
0	"ReactTransitionGroup > ReactCS..."	53.026000037789345	57
1	"SurveyEditor > SurveyForm"	14.3220000090832816	19
2	"SurveyEditor > ReactCSSTransit..."	8.422000042628497	4
3	"SurveyEditor > Divider"	3.816000127699226	19
4	"SurveyForm > ReactDOMInput"	3.74899996774136	19
5	"SurveyForm > ReactDOMTextarea"	2.119999827885497	19
6	"EditMultipleChoiceQuestion > R..."	2.111000183504075	28

Total time: 117.07 ms      ReactDefaultPerf.js:99  
ReactDefaultPerf.js:106

## 11.4 Key

key 속성은 목록에서 자주 사용한다. key를 사용하면 React가 컴포넌트 클래스 보다 더 명확하게 컴포넌트를 확인할 수 있다. 예를 들어 key 값이 foo인 div를 나중에는 bar로 변경한다면 React는 변경 사항을 비교해서 다시 렌더링하지 않고 아예 컴포넌트를 처음부터 렌더링한다.

비교할 필요 없는 거대한 서브트리를 렌더링할 때 이 방법을 사용할 수 있다. 노드에서 제외하는 시점을 지정하는 것 외에 엘리먼트의 순서를 변경할 때 사용할 수도 있다. 다음 예제처럼 정렬 함수를 이용해서 아이템을 렌더링하는 경우를 생각해보자.

---

```
var items = sortBy(this.state.sortingAlgorithm, this.props.items);
return items.map(function (item) {
  return <img src={item.src} />;
});
```

---

엘리먼트의 순서가 바뀌면 React는 `diff`를 비교할 것이고, 그 결과 `img` 엘리먼트의 `src` 속성을 변경하는 것이 가장 효율적인 방법이라고 판단할 것이다. 하지만 이는 아주 비효율적인 방법이며 브라우저가 캐쉬를 확인해서 새로운 네트워크 요청을 일으킬 수 있다. 각 아이템에 고유의 문자열이나 숫자를 넣어 `key`로 사용하면 이 문제를 해결할 수 있다.

---

```
return <img src={item.src} key={item.id} />;
```

---

React는 이제 `src` 속성을 바꾸는 대신 `key`를 찾아서 최소한의 `insertBefore` 동작을 수행한다. 이것이 DOM 노드를 이동하는 최선의 방법이다.

**NOTE 단일 단계 제약**

`key` 속성은 해당 부모 컴포넌트를 기준으로 유일한 값이어야 한다. 다른 부모로의 이동은 고려하지 않는다.

이 방법은 순서 변경뿐만 아니라, 끝이 아닌 곳에 추가해도 사용할 수 있다. `key` 속성이 없는 경우에는 이미지를 추가하면 이후의 모든 `img` 태그의 `src` 속성에 영향을 준다.

한 가지 더 알아두어야 할 것이 있는데 `key`를 `props`로 넘기는 것처럼 보이지만 실제 컴포넌트 내부에서는 이 값에 접근할 수 없다.

## 11.5 정리

이번 장에서 살펴본 내용은 다음과 같다.

1. 성능 개선을 위해 `shouldComponentUpdate`를 수정하여 `true` 또는 `false`를 반환하게 하기
2. `React.addons.Perf`를 이용해 느려지는 원인과 불필요한 렌더링 피악하기
3. 다수의 자식 컴포넌트에 `key`를 적용해서 변경을 최소화하기

지금까지 React를 브라우저에서 사용하는 방법을 집중적으로 살펴봤다. 다음 장에서는 React를 서버에서 사용하는 방법을 살펴본다.

# 서버 사이드 렌더링

검색 엔진에 사이트를 노출하고 싶다면 서버 사이드 렌더링은 필수다. 또한, JavaScript를 로딩 중일 때 웹사이트를 먼저 보여줄 수 있어 성능상 이점이 있다.

React의 가상 DOM은 React를 서버 사이드 렌더링에 사용할 수 있는 이유를 설명하는 데 가장 중요하다. React 컴포넌트는 처음에는 가상 DOM을 먼저 렌더링한 다음에 변경이 발생하면 DOM을 갱신한다. 가상 DOM은 메모리에 존재하기 때문에 Node.js처럼 브라우저가 없는 환경에서도 사용할 수 있다. 실제 DOM을 변경하는 대신에 가상 DOM에서 문자열을 생성할 수 있다. 이러한 이유로 React 컴포넌트는 서버와 클라이언트 양쪽에서 사용할 수 있다. 컴포넌트를 서버 사이드에서 렌더링하려면 `React.renderToString`와 `React.renderToStaticMarkup` 함수를 사용한다. 컴포넌트를 서버 사이드에서 렌더링 할 때는 다음 사항을 잘 고려해서 설계해야 한다.<sup>01</sup>

- 어떤 렌더링 함수를 사용할 것인가
- 컴포넌트의 비동기 상태를 어떻게 지원할 것인가
- 애플리케이션의 초기 상태를 클라이언트에 어떻게 전달할 것인가
- 서버 사이드에서 사용할 수 있는 라이프사이클 함수는 무엇인가
- 애플리케이션에 동형 라우팅(isomorphic routing)을 어떻게 지원할 것인가
- 싱글턴, 인스턴스, 콘텍스트는 어떻게 사용하는가

01 역사주\_ React v.0.14 버전부터는 DOM과 관련된 부분이 `react-dom`으로 분리되었다. 특히 서버 쪽 렌더링에는 `react-dom/server` 패키지를 사용하고, 이에 따라 `ReactDOMServer.renderToString`과 `ReactDOMServer.renderToStaticMarkup`으로 API가 변경되었다.

## 12.1 렌더링 함수

서버 사이드에서 React 컴포넌트를 렌더링할 때는 `React.render` 메소드를 사용할 수 없다. 서버 사이드에는 DOM이 없기 때문이다. React가 서버 사이드에서 지원하는 렌더링 함수는 두 가지가 있다. 이 렌더링 함수는 React 컴포넌트의 라이프 사이클 메소드 중 일부를 지원한다.

### `React.renderToString`

먼저 설명할 `React.renderToString`은 가장 흔하게 사용하는 렌더링 함수다.

`React.render`과 차이점은 렌더링할 위치를 인자로 받지 않으며 메소드 호출 결과를 문자열로 반환한다는 점이다. 이 함수 동기(차단) 방식으로 동작하며 매우 빠르다.

---

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

var world = React.renderToString(<MyComponent/>);

//결과 마크업
<div data-reactid=".fgvrzhhg2yo" data-react-checksum="-1663559667">Hello
World!</div>
```

---

React가 `<div>`에 `data` 속성 두 개를 추가했다. `data-reactid`는 React가 브라우저 환경에서 DOM 노드를 확인할 때 사용한다. `state`나 `props` 값이 바뀌면 이 속성을 이용해서 변경할 노드를 찾는다.

`data-react-checksum`은 서버 사이드 렌더링인 경우에만 추가한다. 이름 그대로 생성하는 DOM의 체크섬<sup>checksum</sup>이다. 이렇게 하면 클라이언트에서 같은 컴포

넌트를 렌더링할 때 서버에서 사용한 DOM을 재사용할 수 있다. 이 속성은 루트 엘리먼트만 가진다.

### React.renderToString

React.renderToString은 서버 사이드에서 사용할 수 있는 또 다른 렌더링 함수다. data 속성을 추가하지 않는다는 점만 React.renderToString과 다르다.

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

var world = React.renderToString(<MyComponent/>);

// 결과 마크업
<div>Hello World!</div>
```

### 적절한 렌더링 함수 선택

각각의 렌더링 함수는 사용 목적이 다르므로 요구사항에 따라 어떤 렌더링 함수를 사용할지 결정해야 한다. React.renderToString은 클라이언트에서 같은 컴포넌트를 렌더링하지 않는 경우에만 사용한다.

예를 들면 다음과 같은 경우다.

- HTML 이메일 생성
- HTML - PDF 변환을 통한 PDF 생성
- 컴포넌트 테스트

대부분은 React.renderToString을 선택할 것이다. 이 함수는 data-react-checksum을 이용해서 같은 컴포넌트를 클라이언트 사이드에서 매우 빠르게 렌

더링한다. 서버에서 제공한 DOM을 재사용할 수 있으므로 DOM 노드를 생성해서 HTML 도큐먼트<sup>document</sup>에 추가하는 비용을 절약할 수 있다. 복잡한 사이트라면 로딩 시간을 획기적으로 줄일 수 있어 사용자가 더욱 빠르게 사이트를 사용할 수 있다.

React 컴포넌트는 서버와 클라이언트에서 같이 렌더링 되어야 한다. `data-react-checksum`이 같지 않으면, 서버에서 제공한 DOM을 제거하고 새로운 DOM 노드를 만들어서 문서에 추가한다. 이렇게 하면 서버 사이드 렌더링으로 얻을 수 있는 성능상의 이점을 모두 잃어버린다.

## 12.2 서버 사이드 컴포넌트 라이프사이클

문자열로 렌더링할 때는 렌더링 이전에 호출할 수 있는 라이프 사이클 메소드만 사용할 수 있다. `componentDidMount`와 `componentWillUnmount`는 렌더링 중에 호출하지 않으며 `componentWillMount`는 렌더링 전과 렌더링 도중에 호출한다.

서버와 클라이언트에서 렌더링하는 컴포넌트를 만들 때 이 점을 유의해야 한다. 이벤트 리스너를 만들 때 특히 중요한데, React 컴포넌트의 종료 시점을 알려주는 라이프사이클 메소드가 없기 때문이다.

서버 사이드에서는 `componentWillMount`에서 생성한 이벤트 리스너나 타이머가 메모리 누수를 일으킬 수 있다. 가장 좋은 해결책은 이벤트 리스너와 타이머를 `componentDidMount`에서만 생성하고 `componentWillUnmount`에서는 중지하는 것이다.

### 컴포넌트 설계

서버 사이드 렌더링의 성능상 이점을 살리려면 상태를 클라이언트에 어떻게 전달 할지 고민해야 한다. 이는 곧 서버 사이드 렌더링을 고려해서 컴포넌트를 설계해

야 함을 의미한다.

같은 `props`를 전달했다면 같은 초기 렌더링을 하도록 React 컴포넌트를 설계해야 한다. 이로써 컴포넌트의 테스트 가능성을 높이고 서버와 클라이언트에서 같은 렌더링을 하게 보장할 수 있다. 서버 사이드 렌더링의 성능상 이점을 얻기 위해서 반드시 쟁겨야 하는 부분이다.

무작위 숫자를 표시하는 컴포넌트를 만든다고 생각해보자. 매번 결괏값이 다르다는 것이 문제가 될 수 있다. 서버에서 렌더링한 컴포넌트를 클라이언트에서 렌더링하면 체크섬 확인에 실패할 것이다.

---

```
var MyComponent = React.createClass({
  render: function () {
    return <div>{Math.random()}</div>;
  }
});

var result = React.renderToString(<MyComponent/>);
var result2 = React.renderToString(<MyComponent/>);

//result
<div>0.5820949131157249</div>

//result2
<div>0.420401572631672</div>
```

---

무작위 숫자를 `props`로 전달할 수 있게 컴포넌트를 수정해보자. 이제 클라이언트에 `props`를 전달해서 렌더링하는 데 사용할 수 있다.

```
var MyComponent = React.createClass({
  render: function () {
    return <div>{this.props.number}</div>;
  }
});
```

```
var num = Math.random();

//서버 사이드
React.renderToString(<MyComponent number={num}/>);

//클라이언트 사이드로 값을 전달함
React.render(<MyComponent number={num}/>, document.body);
```

---

서버에서 사용하는 props를 클라이언트에 전달하는 방법은 여러 가지가 있다. 가장 간단한 방법은 초기 props를 JavaScript 객체로 클라이언트에 전달하는 것이다.

---

```
<!DOCTYPE html>
<html>
<head>
<title>Example</title>
<!--MyComponent, React 등을 포함한 bundle.js -->
<script type="text/javascript" src="bundle.js"></script>
</head>
<body>
<!--MyComponent의 서버 사이드 렌더링 결과 -->
<div data-reactid=".fgvrzhg2yo" data-react-checks
um="-1663559667">0.5820949131157249</div>

<!-- 서버 사이드에서 사용한 초기화 속성 주입 -->
<script type="text/javascript">
    var initialProps = {"num": 0.5820949131157249};
</script>

<!-- 서버 사이드에서 온 초기화 속성 사용 -->
<script type="text/javascript">
    var num = initialProps.num;
    React.render(<MyComponent number={num}/>, document.body);
</script>
</body>
</html>
```

---

## 비동기 상태

많은 애플리케이션이 데이터베이스나 웹서비스 같은 원격 데이터 출처에서 데이터를 가져온다. 클라이언트에서는 이것이 문제가 되지 않는다. React 컴포넌트는 비동기로 데이터를 가져오는 동안 로딩 화면을 보여줄 수 있다. 하지만 서버 사이드 렌더링 함수는 동기방식으로 동작하기 때문에 같은 처리를 할 수 없다. 비동기로 데이터를 가져와서 보여주려면 데이터를 먼저 가져와서 렌더링 시점에 컴포넌트에 전달해야 한다.

### 예제

컴포넌트에 사용할 사용자 정보를 비동기 저장소에서 가져오고 싶다.

그리고

사용자 기록을 가져와서 렌더링한다. SEO와 성능 향상을 위해 서버 사이드 렌더링을 하기 위해 컴포넌트의 상태가 필요하다.

그리고

컴포넌트가 클라이언트에서의 변경을 감지해서 다시 렌더링할 수 있어야 한다.

### 문제

React.renderToString가 동기 방식으로 동작하기 때문에 비동기로 데이터를 받는 데에 컴포넌트 라이프사이클 메소드를 일절 사용할 수 없다.

### 해결법

statics 함수를 이용해서 비동기 데이터를 가져와 렌더링할 컴포넌트에 전달한다. initialState를 클라이언트에 props로 전달한다. 컴포넌트 라이프 사이클 메소드를 이용해서 변경을 확인하고 상태를 갱신한다. 같은 statics 함수를 사용한다.

---

```
var Username = React.createClass({
  statics: {
    getAsyncState: function(props, setState) {
      User.findById(props.userId)
        .then(function(user) {
          setState({ user: user });
        })
        .catch(function(error) {
          setState({ error: error });
        });
    }
  },
  //클라이언트와 서버
  componentWillMount: function() {
    if (this.props.initialState) {
      this.setState(this.props.initialState);
    }
  },
  //클라이언트에서만 사용
  componentDidMount: function() {
    //속성에 없는 경우 비동기 상태를 가져옴
    if (!this.props.initialState) {
      this.updateAsyncState();
    }
    //변경을 확인
    User.on('change', this.updateAsyncState);
  },
  //클라이언트에서만 사용
  componentWillUnmount: function() {
    //변경 확인을 중단함
    User.off('change', this.updateAsyncState);
  },
  updateAsyncState: function() {
    //인스턴스 내부에서 statics 함수에 접근함
    this.constructor.getAsyncState(this.props, this.setState);
  },
});
```

```

    render: function() {
      if (this.state.error) {
        return <div>{this.state.error.message}</div>;
      }
      if (!this.state.user) {
        return <div>Loading...</div>;
      }
      return <div>{this.state.user.username}</div>;
    }
  });

// 서버 사이드 렌더링
var props = {
  userId: 123 //route에서 나올 수도 있음
};
Username.getAsyncState(props, function(initialState) {
  props[initialState] = initialState;
  var result = React.renderToString(Username(props));
  //initialState와 함께 결과 값을 클라이언트에 전달
});

```

---

이 방법은 렌더링을 서버 쪽에서 할 때만 비동기 데이터를 미리 전달받는다. 클라이언트에서 렌더링할 때는 초기 렌더링할 때 서버가 전달한 initialState만 있으면 된다. 이후 클라이언트 상에서의 route를 변경(HTML5의 pushState나 fragment 변경과 같은) 할 때는 서버에서 전달받은 initialState를 무시해야 한다. 데이터를 가져올 때 로딩 메시지를 보여줄 수도 있다.

## 동형 라우팅

라우팅은 큰 애플리케이션에서 중요한 부분이다. 서버에서 라우터를 이용해서 React 애플리케이션을 렌더링하기 위해서는 라우터가 DOM 없는 렌더링을 지원해야 한다.

라우터와 루트 컨트롤러가 비동기 데이터를 가져와서 전달한다. 비동기 데이터를 사용하는 깊은 중첩 관계로 구성된 컴포넌트가 있다고 가정하자. 검색엔진 최적화

를 위해 데이터가 필요한 경우에는 데이터 전달의 책임을 최상위 루트 컨트롤러가 가져야 하며 위에서 아래로 데이터를 전달해야 한다.

검색엔진 최적화가 필요하지 않다면 클라이언트에서 렌더링할 때 `componentDidMount`에서 데이터를 가져온다. 전통적인 AJAX 데이터 전송과 유사하다.

동형 라우팅을 이용하려면 라우터가 비동기 상태를 지원하는지 비동기 상태를 쉽게 변경할 수 있는지 확인해야 한다. 이상적인 라우터는 `initialState`를 클라이언트에 전달할 수 있다.

### **싱글 톤, 인스턴스, 콘텍스트**

브라우저 환경에서 애플리케이션은 분리된 환경을 갖는다. 애플리케이션의 각 인스턴스는 개별 상태를 가지며 다른 인스턴스의 상태와 섞일 수 없다. 각각의 인스턴스는 대체로 서로 다른 컴퓨터에 존재하며, 같은 컴퓨터에 있더라도 샌드박스로 격리하기 때문이다. 애플리케이션 설계에 싱글 톤을 자주 사용하는 이유다.

코드를 서버에서 사용하기 위해 옮길 때는 애플리케이션의 여러 인스턴스를 같은 시점, 같은 유효범위 안에서 실행할 수 있으므로 주의해야 한다. 두 개의 인스턴스가 같은 싱글 톤 객체의 상태를 변경해서 오작동할 가능성이 있다.

React 렌더링은 동기 방식으로 동작하기 때문에, 애플리케이션을 서버에서 렌더링하기 전에 모든 싱글 톤 객체를 재설정할 수 있다. 비동기 상태가 싱글 톤을 사용하는 경우에는 문제가 될 수 있다. 비동기 상태를 가져와서 렌더링에 사용하는 때도 고려해야 한다.

렌더링 전에 싱글 톤을 재설정하더라도 애플리케이션을 독립적으로 실행하는 게 더 낫다. Contextify 같은 패키지를 이용하면 서버에서 코드를 분리하여 실행할 수 있다. 클라이언트에서 `webworker`를 사용하는 것과 비슷하다. Contextify는 애플리케이션 코드를 별도의 Node.js V8 인스턴스에서 실행한다. 코드가 로

딩되면 해당 콘텍스트에서 함수를 실행할 수 있다. 싱글 톤을 자유롭게 사용할 수 있지만 성능이 떨어지는 것이 단점이다. 요청할 때마다 Node.js V8 인스턴스를 시작해야 한다.

React 코어 개발팀은 콘텍스트와 인스턴스를 컴포넌트 트리 아래로 전달하는 방식을 추천하지 않는다. 이는 컴포넌트 이동을 어렵게 만든다. 또한, 애플리케이션 깊은 곳에 있는 컴포넌트의 의존성에 변화가 생기면 상위 컴포넌트까지 영향을 받는다. 결과적으로 애플리케이션의 복잡도를 증가시켜 애플리케이션을 확장할수록 유지 보수하기 어려워진다.

싱글 톤을 사용하는 방법이나 인스턴스로 콘텍스트를 제어하는 방법 모두 장단점이 있다. 요구사항을 잘 정리해본 후에 적합한 방법을 선택해야 한다. 사용하는 외부 라이브러리의 설계 형태도 함께 고려해야 한다.

### 12.3 정리

서버 사이드 렌더링은 웹사이트나 웹 애플리케이션을 검색 엔진에 최적화 하는 데에 중요하다. React 컴포넌트는 서버와 클라이언트에서 같이 렌더링할 수 있다. 효과적으로 렌더링하려면 전체 애플리케이션이 서버 사이드 렌더링을 지원할 수 있게 설계해야 한다.

다음 장에서는 React 패밀리 라이브러리를 살펴본다.



# React 패밀리

페이스북은 React 외에도 많은 프론트엔드 개발 도구를 만들고 있다. 지금부터 설명 할 도구는 React와 별개로 사용할 수 있으며 React 프로젝트를 할 때 반드시 사용하지 않아도 상관없다. 다만 React와 함께 사용하면 더 좋은 효과를 얻을 수 있다.

이번 장에서 알아볼 도구는 다음과 같다.

- Jest
- Immutable.js
- Flux

## 13.1 Jest

Jest는 Facebook이 Jasmine 테스트 프레임워크를 이용해서 만든 테스트 러너 `test runner`이다. `expect( value ).toBe( other )` 같이 Jasmine과 거의 유사한 조작 방식을 제공한다.

Jest는 기본적으로 `require()`가 반환하는 CommonJS 모듈을 자동으로 모의 객체로 만들어 기존 코드를 테스트할 수 있는 코드로 변경한다. 또한, 모의 DOM API를 이용하는 Jest는 피드백을 빠르게 제공하며 Jest를 이용하면 Node.js 커맨트 라인 유틸리티를 이용해서 테스트를 병렬로 진행할 수도 있다.

다음은 Jasmine 테스트 라이브러리에 익숙할 것이라는 가정하에 세 가지 내용을 설명한다.

- 설치
- 의존성 모듈 자동 모의 객체화
- 의존성 모듈 수동 모의 객체화

## 설치

Jest를 설치하려면 우선 `__tests__`(이름은 변경 가능) 폴더를 만든다. 그다음 만 들어진 폴더에 테스트 코드를 작성하고 저장한다.

---

```
// __tests__/sum-test.js
jest.dontMock('../sum');

describe('sum', function() {
    it('adds 1 + 2 to equal 3', function() {
        var sum = require('../sum');
        expect(sum(1, 2)).toBe(3);
    });
});
```

---

커맨드 라인에 `npm install jest-cli --save-dev`를 입력해서 Jest를 인스톨 한 후 `run jest`를 입력해서 테스트를 시작한다. 테스트 결과가 화면에 출력되면 성공이다.

---

```
[PASS] __tests__/sum-test.js (0.015s)
```

---

## 자동 모의객체화

Jest는 자동으로 소스 파일이 의존하는 모든 모듈을 모의 객체로 만든다. `node`로 필요한 함수를 덮어쓰는 방식이다.

**NOTE**

<https://github.com/backstopmedia/bleeding-edge-sample-app>에서 전체 예제 코드를 볼 수 있다.

여기 TakeSurveyItem 컴포넌트가 있다.

```
var React = require('react');
var AnswerFactory = require('./answers/answer_factory');

var TakeSurveyItem = React.createClass({
  render: function () {
    // ...
  },
  getSurveyItemClass: function () {
    return AnswerFactory.getAnswerClass(this.props.item.type);
  }
});

module.exports = TakeSurveyItem;
```

getSurveyItemClass 함수는 AnswerFactory를 의존한다. TakeSurveyItem을 테스트하는 것이 목적이기 때문에 여기에서 AnswerFactory.getAnswerclass를 테스트할 필요가 없다. 그냥 AnswerFactory의 메소드를 제대로 호출하는지 정도만 확인하면 된다.

Jest는 자동으로 모든 의존 모듈을 모의 객체로 만든다. AnswerFactory를 모의 객체로 대체하면 getAnswerClass 메소드 호출 여부를 간단히 확인할 수 있다. 하지만 지금 테스트해야 할 대상인 TakeSurveyItem은 모의 객체로 만들어선 안 된다. 그래서 TakeSurveyItem#getSurveyItemClass를 테스트하는 코드를 작성했다.

---

```
jest.dontMock('react');
jest.dontMock('app/components/take_survey_item');

var TakeSurveyItem = require('app/components/take_survey_item');
var AnswerFactory = require('app/components/answers/answer_factory');
var React = require('react/addons');
var TestUtils = React.addons.TestUtils;

describe('app/components/take_survey_item', function () {
  var subject;
  beforeEach(function () {
    subject = TestUtils.renderIntoDocument(TakeSurveyItem());
  });

  describe('#getSurveyItemClass', function () {
    it('calls AnswerFactory.getAnswerClass', function () {
      subject.getSurveyItemClass();
      expect( AnswerFactory.getAnswerClass ).toBeCalled();
    });
  });
});
```

---

Jest에 TakeSurveyItem 컴포넌트와 React모듈을 모의 객체로 만들지 않을 것을 요청하고 있다. 모의 객체가 아닌 실제 코드를 이용해서 테스트를 수행할 수 있다.

그다음에 AnswerFactory를 비롯해서 테스트에 필요한 모든 모듈을 요청한다. AnswerFactory를 모의 객체화하지 않겠다는 선언을 하지 않았으므로 require 함수는 TakeSurveyItem의 모의 객체를 반환한다.

이 테스트는 `toBeCalled`를 이용해서 TakeSurveyItem에 있는 `getSurveyItemClass`를 실행하면서 `AnswerFactory.getAnswerClass` 메소드를 호출하는지 확인한다. 이 함수는 Jasmine이 제공하는 스파이<sup>spy</sup> 객체의 `toHaveBeenCalled`와는 다르다. Jest는 스파이 객체의 동작을 방해하지 않는다. 따라서 원한다면 둘 다 사용할 수 있다.

## 수동 모의 객체화

가끔 Jest가 제공하는 자동 모의 객체 기능이 부족할 때가 있다. 그럴 때는 다른 라이브러리를 이용한다. 앞에서 사용했던 AnswerFactory 모듈을 직접 모의 객체화로 만들어본다.

---

```
jest.dontMock('react');
jest.dontMock('app/components/take_survey_item');

// require로 TakeSurveyItem을 가져오기 전에 수동으로 모의 객체를 만든다.
// 그렇지 않으면 AnswerFactory의 다른 모의 객체를 받는다.
// the manual mock needs to happen before we require
// TakeSurveyItem, otherwise TakeSurveyItem will receive a
// different mock of AnswerFactory
jest.setMock('app/components/answers/answer_factory', {
  getAnswerClass: jest.genMockFn().mockReturnValue(TestUtils.
    mockComponent)
});

var TakeSurveyItem = require('app/components/take_survey_item');
var AnswerFactory = require('app/components/answers/answer_factory');
var React = require('react/addons');
var TestUtils = React.addons.TestUtils;

describe('app/components/take_survey_item', function () {
  var subject;

  beforeEach(function () {
    // ...
  });

  describe('#getSurveyItemClass', function () {
    it('calls AnswerFactory.getAnswerClass', function () {
      // ...
    });
  });
});
```

---

Jest가 자동으로 모의 객체로 만들지 못하는 일반 라이브러리나 AnswerFactory 같은 모듈을 직접 모의 객체로 만들고 싶다면, answer\_factory.js 디렉터리에 \_\_mocks\_\_ 폴더를 만들어서 모의 객체를 정의하는 파일을 작성한다.

---

```
// app/components/answers/__mocks__/answer_factory.js
var React = require('react-addons');
var TestUtils = React.addons.TestUtils;

module.exports = {
  getAnswerClass: jest.genMockFn().mockReturnValue(TestUtils.mockComponent)
};
```

---

이제 테스트를 작성할 때 jest.setMock를 호출할 필요가 없다.

---

```
jest.dontMock('react');
jest.dontMock('app/components/take_survey_item');

var TakeSurveyItem = require('app/components/take_survey_item');
var AnswerFactory = require('app/components/answers/answer_factory');
var React = require('react-addons');
var TestUtils = React.addons.TestUtils;

describe('app/components/take_survey_item', function () {
  var subject;

  beforeEach(function () {
    // ...
  });

  describe('#getSurveyItemClass', function () {
    it('calls AnswerFactory.getAnswerClass', function () {
      // ...
    });
  });
});
```

---

불변 자료 구조는 일단 생성한 후에는 데이터를 변경할 수 없는 자료 구조를 말한

다. 사용자가 데이터 변경을 요청하면 원본 데이터를 변경하는 것이 아니라, 복사본을 만들어서 데이터를 변경한 다음 해당 복사본을 돌려준다. 불변 자료 구조는 React와 Flux의 단순함과 아주 잘 어울리며 잘 사용하면 애플리케이션의 성능을 향상할 수 있다.

Immutable.js를 이용하면 JavaScript의 기본 자료 구조를 불변 자료 구조로 변경할 수 있으며 나중에 다시 기본 자료 구조로 되돌릴 수도 있다.

## 13.2 Immutable.Map

JavaScript의 Object를 대신해서 Immutable.Map을 사용할 수 있다.

```
var question = Immutable.Map({description: 'who is your favorite superhero?'});
// get values from the Map with .get
// .get 메소드로 Map에서 값을 가져온다.
question.get('description');

// updating values with .set returns a new object.
// The original object remains intact.
// .set 메소드는 인자로 전달한 값을 갖는 새로운 객체를 돌려준다.
// 원래 객체는 그대로 존재한다.
question2 = question.set('description', 'Who is your favorite comicbook hero?');

// merge 2 objects with .merge to get a third object.
// Once again none of the original objects are mutated.
// .merge를 이용해서 두 객체를 합친 세번째 객체를 돌려준다.
// 그 어떤 객체의 상태도 바뀌지 않았다.
var title = { title: 'Question #1' };
var question3 = question.merge(question2, title);
// { title: 'Question #1', description: 'who is your favorite comicbook hero' }
question3.toObject();
```

## Immutable.Vector

Immutable.Vector는 배열로 사용할 수 있다.

```
var options = Immutable.Vector('Superman' , 'Batman');
var options2 = options.push('Spiderman');
options2.toArray(); // ['Superman', 'Batman', 'Spiderman']
```

서로 다른 자료 구조를 혼용할 수도 있다.

```
var options = Immutable.Vector('Superman' , 'Batman');
var question = Immutable.Map({
  description: 'who is your favorite superhero?',
  options: options
});
```

Immutable.js는 더 많은 특징을 가지고 있다. 더 자세한 내용을 알고 싶다면 <https://github.com/facebook/immutable-js>를 방문하기 바란다.

## 13.3 Flux

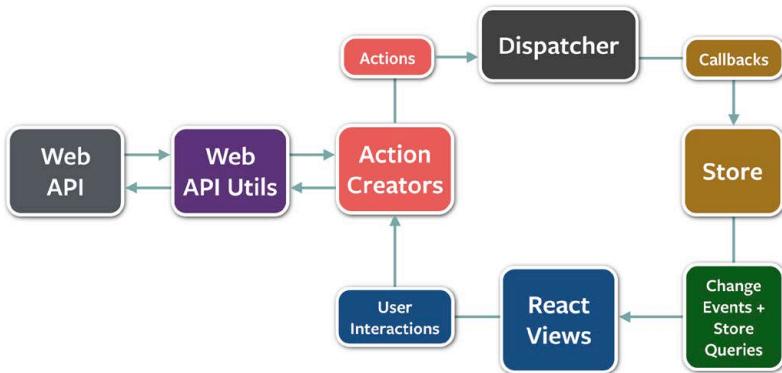
'설계 패턴'에서 자세히 설명하겠지만, Flux는 페이스북이 React와 함께 공개한 일종의 패턴이다. Flux의 가장 큰 특징은 데이터가 단방향으로 흐르도록 엄격하게 통제한다는 점이다. 페이스북이 공개한 Flux 코드는 GitHub, <https://github.com/facebook/flux>에서 볼 수 있다.

Flux는 세 개의 주요 컴포넌트로 구성되어 있다.

- Dispatcher
- Stores
- Views

다음은 세 컴포넌트의 관계를 도식화한 모습이다.

[그림13-1]



이미지 출처: 페이스북

Flux의 각 컴포넌트는 의존성을 갖지 않는다. 얼마든지 자유롭게 원하는 부분만 채택해서 프로젝트에 적용할 수 있다. Flux에 대한 자세한 내용은 16장에서 다룬다.

## 13.4 정리

이번 장에서는 다음 내용을 소개했다.

1. Jest를 이용해서 의존 모듈을 모의 객체로 만드는 방법
2. Immutable.js를 이용해서 기본 JavaScript 자료 구조를 대체하기
3. Flux 개요

다음 장에서는 React 컴포넌트를 개발할 때 이용할 수 있는 무료 도구와 디버깅 기술을 알아본다.



# 개발 도구

React가 제공하는 추상 레이어를 이용하면 쉽게 컴포넌트를 개발하고 애플리케이션의 상태를 파악할 수 있다. 하지만 애플리케이션을 디버깅하거나 빌드 또는 출시할 때는 이 추상 레이어가 오히려 단점으로 작용할 수도 있다.

다행히 애플리케이션을 디버깅하거나 빌드할 때 사용할 수 있는 멋진 개발 도구가 있다. 이 장에서는 React를 개발할 때 사용할 수 있는 도구를 살펴본다.

## 14.1 빌드 도구

빌드 도구를 사용하면 반복 작업을 훨씬 쉽게 처리할 수 있다. React로 개발할 때 가장 많이 하는 반복 작업은 JSX 해석기를 실행해서 모든 React component를 해석하는 일이다. 그리고 브라우저 환경에서 사용하기 위해 개발한 모든 모듈을 하나로 합치거나, 묶음으로 포장하는 일 또한 큰 작업이다.

인기 있는 빌드 도구인 Browserify와 Webpack 사용법을 알아보자.

설문 조사 애플리케이션 예제는 JavaScript를 묶음으로 포장하는 수준의 아주 작은 부분에 초점을 맞춰서 Browerify를 사용했다. 아주 쉽게 사용해 볼 수 있으며 설정 파일을 작성할 필요가 없다.

**NOTE**

예제로 사용한 설문 조사 제작 애플리케이션의 전체 코드는 아래 링크에 있다.

<https://github.com/backstopmedia/bleeding-edge-sample-app>

## 14.2 Browserify

Browserify는 Node.js가 제공하는 `require()` 함수를 브라우저 환경에서도 사용할 수 있게 도와주는 JavaScript 패키지 도구다. 간단히 말해서, Browserify를 이용하면 모든 JavaScript 파일을 브라우저에서 실행 가능한 코드 뮤음으로 만들 수 있다. 어떤 JavaScript 파일이든 필요하다면 `require` 구문으로 가져올 수 있다.

Browserify는 강력하지만, Bower나 Webpack 같은 다른 패키징 솔루션과 달리 오로지 JavaScript만 처리한다.

### Browserify 프로젝트 설치

Browserify를 설치하려면 node 프로젝트를 만들어야 한다. 이미 node와 npm을 설치했다면 터미널에 다음 명령을 입력해서 새로운 프로젝트를 시작한다. `package.json` 파일이 생겼을 것이다.

---

```
npm init
# ... 프로젝트 시작에 필요한 몇 가지 사항을 물어본다.
npm install --save-dev browserify reactify react uglify-js
```

---

다음 빌드 스크립트를 `package.json` 파일 하단에 추가한다.

---

```
...
"devDependencies": {
  "browserify": "^5.11.2",
```

```
"reactify": "^0.14.0",
"react": "^0.11.1",
"uglify-js": "^2.4.15"
},
"scripts": {
  "build": "browserify --debug index.js > bundle.js",
  "build-dist": "NODE_ENV=production browserify index.js | uglifyjs -m >
bundle.min.js"
},
"browserify": {
  "transform": ["reactify"]
}
```

---

`npm run build` 명령을 입력해서 기본 빌드를 실행하면 소스 맵 파일과 함께 번들 파일이 만들어진다. 이제 마치 개별 파일을 가지고 작업하는 것처럼 에러 메시지를 확인하고 브레이크 포인트를 걸어서 디버깅할 수 있다. 또한 컴파일하기 전의 JSX로 작성한 본래 코드를 볼 수 있다.

제품 빌드를 위해 `NODE_ENV` 값을 `production`으로 명시했다. React는 압축 도구인 `uglify`와 함께 `envify`를 이용한다. `envify`를 이용하면 디버깅 코드와 상세 에러 메시지를 제거해서 컴포넌트의 성능을 향상할 수 있으며 파일 사이즈를 더 줄일 수 있다.

화살표 함수나 클래스 같은 ES6 문법을 사용하고 싶다면 `transform` 옵션을 수정한다.

---

```
"transform": [[{"reactify": {"harmony": true}}]]
```

---

이제 React 컴포넌트를 만들어서 패키징 할 준비가 끝났다.

## React 콘텐츠 추가

Index.js 파일에 다음에 있는 React + JSX 코드를 작성하고 저장한다.

---

```
var React = require('react');
React.render(<h1>Hello World</h1>, document.body);
```

---

그리고 간단한 Index.html 파일을 만든다.

---

```
<html>
<head>
  <title>React + Browserify Demo</title>
</head>
<body>
  브라우저 화면에 이 문장이 나와선 안 된다.
  <script src="bundle.js"></script>
</body>
</html>
```

---

이제 루트 디렉터리 안에는 다음 파일과 폴더가 있을 것이다.

- index.html
- index.js
- node\_modules/
- package.json

index.html 파일을 브라우저로 열어보면 JavaScript를 읽지 못한다는 걸 알 수 있다.

아직 최종 패키지를 빌드하지 않았기 때문이다. `npm run build`를 실행한 다음에 폐이지를 새로고침하면 화면에 'Hello World'가 찍혀있는 것을 볼 수 있다.

## Watchify

감시 작업을 추가하면 더 편하게 개발할 수 있다. Watchify는 Browerify를 감싸고 있다. Watchify는 감시 중인 파일의 변경을 감지하면 빌드 작업을 수행한다. 이 때 캐시를 이용해서 빌드 속도를 높인다.

`npm`으로 `watchify`를 설치한다.

---

```
npm install --save-dev watchify
```

---

그리고 package.json의 scripts에 다음 내용을 추가한다.

---

```
"watch": "watchify --debug index.js -o bundle.js"
```

---

npm run build 대신에 npm run watch를 실행한다. 이제 더 부드럽게 개발 프로세스를 진행할 수 있다.

## 빌드

간단하게 빌드 스크립트를 이용해서 React + JSX 파일을 브라우저에서 실행할 수 있는 하나의 파일로 묶어 보자.

---

```
npm run-script build
```

---

방금 새로운 bundle.js 파일을 만들었다. bundle.js 파일을 열어보면 상단에 압축 상태의 JavaScript 코드를 확인할 수 있다. React + JSX 코드를 변환하여 압축한 결과물이다. 이 파일은 이제 index.js의 내용을 모두 가지고 있으며 브라우저에서 실행할 수 있다. index.html을 브라우저로 열어서 확인한다.

## 14.3 Webpack

Browerify처럼 Webpack도 JavaScript 코드를 하나의 패키지로 포장한다. 하지만 Browerify와 Webpack을 비교하는 것은 무리다. Webpack은 Browerify가 아직 지원하지 않는 더 많은 기능을 지원하기 때문이다.

Webpack을 이용하면 다음 작업을 할 수 있다.

- CSS, 이미지 등의 자원을 같은 패키지에 통합
- 파일 통합 전처리(less, coffee script, jsx 등)
- 통합파일 분리
- 개발에 필요한 기능 설정 지원
- 최신 모듈 교체 수행
- 비동기 로딩 지원

Webpack은 Browserify의 역할에 더해서 grunt나 gulp 같은 다른 빌드 도구의 역할도 할 수 있고, 모듈 시스템의 역할도 하므로 플러그인을 추가/삭제할 수도 있다. 또한, 기본적으로 commonJS 해석기 플러그인을 제공한다.

여기서 Webpack을 상세히 설명하기는 힘드므로 React를 이용해서 개발하는 데 필요한 기본만 살펴보겠다.

## 14.4 Webpack과 React

React를 이용해서 애플리케이션에 필요한 컴포넌트를 만들 때 Webpack을 이용하면 JavaScript뿐만 아니라, 필요한 다른 모든 자원까지 하나로 패키징 할 수 있다. 모든 의존 자원을 가지고 있는 컴포넌트를 만들 수 있다는 뜻이다. 컴포넌트가 의존 자원을 함께 달고 다닐 수 있어 컴포넌트의 이동성이 더 좋아진다.

게다가 애플리케이션의 규모가 커지거나 변경이 생겨 컴포넌트를 삭제해야 할 때 의존 자원까지 함께 삭제할 수 있다. 따라서 불필요한 CSS나 이미지가 생기는 것을 막을 수 있다.

---

```
//logo.js
require('./logo.css');
var React = require('react');

var Logo = React.createClass({
  render: function () {
```

```
        return <img className="Logo" src={require('./logo.png')} />
    }
});

module.exports = Logo;
```

---

이 컴포넌트를 패키징 하려면 애플리케이션의 시작점이 있어야 한다.

---

```
//app.js
var React = require('react');
var Logo = require('./logo.js');

React.render(<Logo/>, document.body);
```

---

이제 Webpack 설정 파일을 만들어서 파일 타입에 따라 어떤 로더(loader)를 사용할지 설정한다. 애플리케이션 시작점과 하나로 묶은 파일을 저장할 위치를 알 수 있다.

---

```
//webpack.config.js
module.exports = {
// 애플리케이션 시작점 지정
entry: './app.js',
output: {
    // 최종 파일 생성 위치
    path: './public/build',
    // 모든 url-loader 자원에 붙일 접두사
    publicPath: './build/',
    // 패키징한 애플리케이션 파일 이름
    filename: 'bundle.js'
},
module: {
    loaders: [
    {
        //loader가 지원하는 파일 타입 정규식
        test: /\.js$/,
        // 사용할 loader 타입
        // loader는 쿼리 스트링을 파라미터로 받을 수 있다.
        loader: 'jsx-loader?harmony'
    }
]
```

```
    },
    {
      test: /\.css$/,
      // "!로 구분해서 여러 개의 loader를 연속 나열한다.
      loader: 'style-loader!css-loader'
    },
    {
      test: /\.(png|jpg)$/,
      // url-loader는 인라인 자원에 필요한 base64 인코딩을 지원한다.
      loader: 'url-loader?size=8192'
    }
  ]
}
};
```

이제 npm이나 package.json을 이용해서 Webpack과 로더를 설치한다. 로더를 글로벌이 아닌 로컬에 설치해야 한다는 걸 주의하자.

```
npm install webpack react  
npm install url-loader jsx-loader style-loader css-loader
```

모두 설치하고 나면 Webpack을 실행할 수 있다.

```
// 개발용 코드 빌드  
webpack  
  
// 소스 맵 생성  
webpack -d  
  
// 제품용 코드 빌드를 위해 압축, 난독화, 불필요한 코드를 제거  
webpack -p  
  
// 변경이 발생하면 자동 빌드. 다른 옵션과 함께 사용할 수 있다  
webpack --watch
```

## 14.5 디버깅 도구

인간은 아무리 조심해도 실수를 하기 마련이다. 여기에서 JavaScript 디버깅 방법을 설명하지는 않는다. 대신 React 애플리케이션을 개발할 때 유용한 디버깅 도구를 알아보겠다.

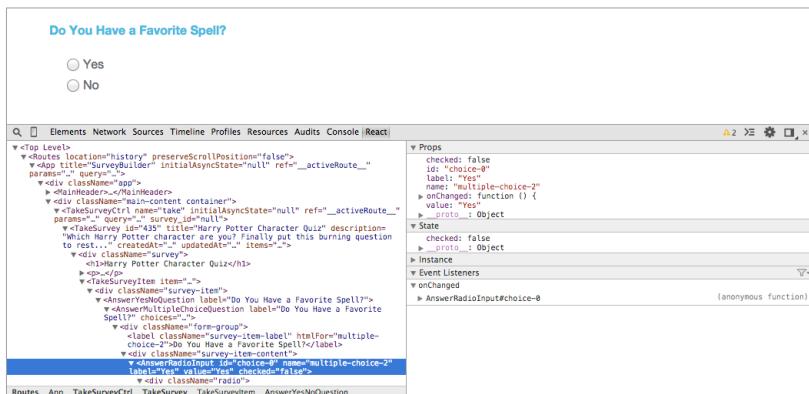
### 시작하기

크롬을 열어서 React Developer Tools addon을 설치한다. React를 가져와서 `window.React = require('react');`

```
window.React = require('react');
```

엘리먼트를 오른 클릭하고 ‘요소 검사’를 선택한다. Elements 탭이 열리고 익숙한 DOM 구조를 볼 수 있다. 하지만 보고 싶은 건 이게 아니다. React 컴포넌트, 그리고 그 컴포넌트의 `props`와 `state`를 보고 싶다. 탭 메뉴 중 우측에 있는 React를 클릭한다.

[그림 14-1]



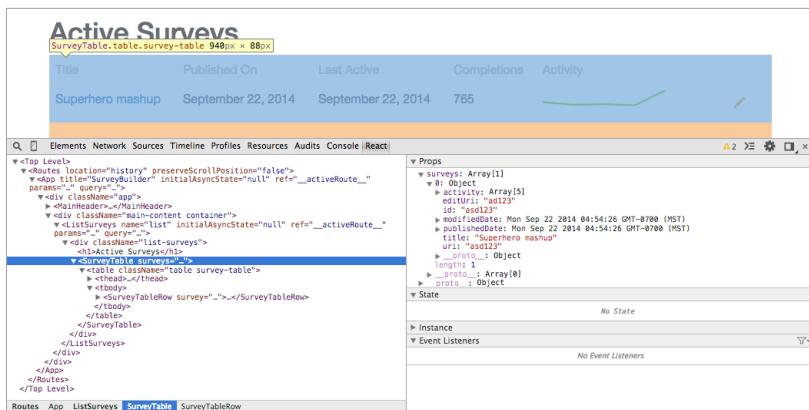
## DISPLAYNAME

JSX 변환기는 displayName을 추론해서 삽입한다. JSX를 사용하지 않는다면 컴포넌트의 displayName을 직접 지정해야 한다.

```
React.createClass({displayName: "MyComponent", ...});
```

왼쪽에는 컴포넌트 계층 구조가 보이고, 오른쪽에는 선택한 인스턴스에 대한 정보가 있다. 일단 state, props와 React로 컴포넌트에 붙인 이벤트 리스너에 대한 많은 정보가 나와 있다. onDragStart 리스너가 ModuleButton::handleDragStart라는 것을 알 수 있다. 버튼이 가지고 있는 클래스와 다른 재밌는 정보도 얻을 수 있다. 개발자 도구는 이보다 더 많은 것을 지원한다.

[그림 14-2]



SurveyTable 컴포넌트에 전달한 surveys 배열이 가지고 있는 타임스탬프 값을 확인할 수 있다. 실제 뷰는 사람이 읽을 수 있는 형식으로 표현한다. 타임스탬프 하나를 더블 클릭해서 새로운 값을 입력하면 새로 입력한 값으로 컴포넌트를 갱신한다.

개발자 도구를 이용하면 문제의 범위를 좁힐 수 있다. 프로젝트에 새로 들어온 동

료는 개발자 도구를 이용해서 컴포넌트를 쉽게 디버깅할 수 있을 것이다.

**NOTE** **JSBin과 JSFiddle**

디버깅이나 브레인 스토밍을 할 때 JSFiddle이나 JSBin 같은 온라인 데모 사이트를 이용하면 아주 좋다. 도움을 요청하거나 프로토 타입을 공유하고 싶을 때 이 사이트를 이용하기로 권한다.

## 14.6 정리

이제 React로 개발할 때 디버깅 도구와 빌드 도구를 사용해서 얻을 수 있는 이점을 알았다. 다음 장에서는 테스트 자동화 방법을 상세하게 알아본다.



# 테스트

지난 장에서는 React로 웹 애플리케이션을 만드는 방법을 설명했다. 아키텍처 패턴을 학습하기 전에 배워야 할 것이 있다. 새로운 애플리케이션을 만들 때 생산성을 높이는 방법은 아주 간단하다. 그저 코드를 빨리 작성하면 된다. 하지만 주의를 기울이지 않으면 코드는 점점 꼬여서 스파게티처럼 되어버릴 것이다. 이런 코드는 나중에 변경하기 어렵다.

그래서 도구를 이용해서 테스트를 자동화해야 한다. 흔히 TDD 워크플로우를 이용하는 테스트 자동화를 도입하면 더 간단하게 모듈화 된 코드를 작성할 수 있다. 이런 코드는 변경에 덜 취약하므로 자신 있게 코드를 변경할 수 있다.

**NOTE** JavaScript 코드를 테스트해 본 적이 없는데?

전에 테스트해 본 적 없어도 상관없다. 테스트 자동화를 달을 수 없는 어딘가에 있는 새로운 개념으로 생각할 수도 있다. 이 장은 JavaScript 테스트를 빼침없이 다루지는 않는다. 이 책의 범위가 아니기 때문이다. 하지만 필요할 때 그 주제를 찾아볼 수 있을 정도의 내용은 실었다.

## 15.1 시작하기

테스트 자동화의 진정한 목표는 더 나은 코드를 작성하는 것이다. 대개 좋지 않은 코드는 테스트하기 어렵다. 자신이 작성한 코드를 테스트하면 영성한 코드를 작성

하는 일을 줄일 수 있다. 자연스럽게 단일 책임 원칙<sup>Single Responsibility Principle</sup><sup>01</sup>과 디미터 법칙<sup>the law of demeter</sup>을 따르며 코드를 모듈화할 수 있다.

테스트 주도 개발<sup>Test Driven Development, TDD</sup>은 보통 “빨강, 초록, 리팩토링” 프로세스를 따르는 테스트 스타일을 말한다. 먼저 실패하는 테스트 코드를 작성한다(빨간색). 그 다음에 테스트를 통과할 수 있는 애플리케이션 코드를 작성한다(녹색). 테스트를 통과하면 방금 작성한 애플리케이션 코드를 리팩토링한다. 리팩토링 도중에는 항상 테스트를 녹색 상태로 유지해야 한다. 이 프로세스를 따르면 코드를 작은 단위로 작성할 수 있다. 그리고 반복하면서 테스트가 통과하여 녹색으로 바뀌는 걸 보면 기분이 좋아진다.

## 테스트의 종류

테스트 자동화가 중요하다는 것을 알았다면, 단위 테스트와 기능 테스트, 이 두 유형의 테스트를 살펴보자. 다양한 형태의 테스트 자동화(통합 테스트, 성능 테스트, 보안 테스트, 시각 테스트<sup>02</sup>)가 있지만, 이 둘을 제외한 나머지는 이 책의 범위를 벗어난다.

- 단위 테스트 : 애플리케이션의 기능 중 아주 작은 부분을 검증하는 테스트다. 주로 특정 입력을 넣어서 기능을 직접 호출한 다음에 출력이 유효한지, 사이드 이펙트는 없는지 확인한다.
- 기능 테스트 : 최종 사용자 입장에서 애플리케이션의 기능이 제대로 동작하는지 검증하는 테스트다. 사용자가 되어서 브라우저로 웹 애플리케이션을 클릭하고, 양식을 작성해 본다. 너무 많은 걸 요구하는 것처럼 들릴 수도 있다. 하지만 이 과정을 거치면서 기능을 관리할 수 있을 뿐만 아니라, 첫 테스트 통과의 기쁨을 느낄 수 있다.

## Tools

다행히 JavaScript 커뮤니티는 훌륭한 테스트 도구 생태계를 가지고 있다. 이 도

---

01 Single Responsibility Principle : <http://blog.8thlight.com/uncle-bob/2014/05/08/SingleResponsibilityPrinciple.html>

02 <https://www.youtube.com/watch?v=1wHr-06gEfc>

구를 이용하면 테스트 스위트를 빠르게 작성할 수 있다. 다음 목록은 이 책에서 소개하는 소프트웨어 스택이다. 그 다음에 대안이 될만한 인기 있는 도구도 함께 소개한다.

- 단위 테스트(클라이언트) : Jasmine, Karma
  - 대안 : Mocha, Chai, Sinon, Vows.js, Qunit
- 단위 테스트(서버) : Mocha, Supertest
  - 대안 : jasmine-node, 나머지는 클라이언트와 동일
- 기능 테스트 : Casper.js
  - 대안 : Nightwatch.js, Zombie.js, Selenium 기반의 도구들(Capybara, Waitr 등)

이제 시작한다.

## 15.2 첫 번째 명세 : 렌더링

React 컴포넌트를 만들 때, 유일한 요구사항은 render 함수를 작성하는 일이다. 따라서 render 함수는 테스트를 시도하기에 아주 좋은 위치다. 간단히 <h1> 태그로 "Hello World!"을 출력하는 <HelloWorld> 컴포넌트를 만들고 싶다고 가정해보자. TDD를 할 것이기 때문에 <HelloWorld> 컴포넌트를 만드는 일은 일단 나중에 필요할 때까지 뒤로 미룬다. 우선 JavaScript 파일을 만들어서 명세(테스트)를 작성하는 거로 테스트를 시작한다.

---

test/client/fundamentals/render\_into\_document\_spec.js

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

describe("HelloWorld", function(){
});
```

---

Jasmine으로 React 단위 테스트를 작성하는 보일러플레이트다. 그동안 배운 내용을 확인하는 의미에서 하나씩 들어보자.

---

```
/** @jsx React.DOM */
```

---

이 테스트 명세가 JSX를 사용한다는 사실을 명시해서 JSX 해석기에 전달한다.<sup>03</sup>

---

```
var React = require("react/addons");
var TestUtils = React.addons.TestUtils;
```

---

애플리케이션 코드에 있는 일반적인 `require("react")` 대신에 정규 React 함수에 일부 테스트 유ти리티 함수를 추가한 `"react/addons"` 패키지를 이용할 수도 있다. `React.addons.TestUtils`는 유용한데 이번 장과 18장에서 설명한다.<sup>04</sup>

---

```
describe("HelloWorld", function(){
});
```

---

`jasmine`의 `describe` 함수를 이용해서 Hello-World 모듈을 위한 테스트 스위트를 만들었다. 이 코드를 이용해서 Hello-World 컴포넌트 렌더링을 테스트한다. 언뜻 보기에도 명세가 아주 간단해서 `React.renderComponent(<HelloWorld>, someDomElement)`를 호출한 결과를 단순히 확인만 하면 될 것 같다. 뒤에서 보겠지만, React의 렌더링 로직은 이보다 좋은 고급 기능이 있다.

---

03 역사주\_ React v.0.12 버전 이후부터는 `@jsx` 컴파일 지시문을 사용하지 않아도 된다.

04 역사주\_ React v.0.14 버전부터는 `addons`가 별도의 패키지로 분리되었다. `React.addons.TestUtils`는 `react-addons-test-utils`를 이용한다.

**NOTE 테스트 작성 시 React.renderComponent 사용**

테스트할 때 `React.renderComponent`를 사용할 경우 하나의 테스트가 뒤에 있는 다른 테스트를 오염시킬 가능성이 매우 크다. 테스트 오염은 알 수 없는 테스트 실패를 만들며 통과해서는 안 되는 테스트가 통과한 것처럼 보이게 만든다. 테스트 오염은 `React`가 이미 엘리먼트로 렌더링한 동일 컴포넌트를 다시 시작하지 않기 때문에 발생한다. 실제 제품에서는 이렇게 함으로써 성능을 높일 수 있지만, 테스트 스위트에서는 문제를 일으킬 수 있다.

테스트 스위트에서 컴포넌트를 렌더링하려면 `React.addons.TestUtils.renderIntoDocument` 함수를 호출한다. 이 함수를 호출할 때 컴포넌트를 하나 인자로 전달한다. 컴포넌트를 삽입하고자 하는 엘리먼트를 두 번째 인자로 받는 `React.renderComponent`와 다르다는 것을 눈치챘을 것이다. 이는 `renderIntoDocument` 함수가 메모리에 별도로 존재하는 DOM에 컴포넌트를 삽입함을 의미 한다. 첫 번째 테스트의 목표는 아주 간단하다. 컴포넌트를 그리는 일이다.

---

```
describe("HelloWorld", function(){

  describe("renderIntoDocument", function(){
    it("should render the component", function(){
      TestUtils.renderIntoDocument(<helloworld></helloworld>);
    });
  });
});
```

---

Jasmine 명세를 모두 작성했고, 명세대로 실행해야 한다. 선택할 수 있는 오픈 소스 프로젝트가 많이 있지만, 여기서는 Karma를 선택했다. Karma는 Google 이 개발한 JavaScript 테스트 러너이다. Testacular라고도 부른다. Karma는 테스트를 여러 브라우저에서 실행하고 결과를 종합하여 알려주는 편리한 도구다. 다음 명령어를 실행해서 카르마 테스트를 시작한다.

---

```
npm run-script test-client
```

---

실행하면 디버깅 결과를 볼 수 있고, 다음과 같은 최종 결과도 확인할 수 있다.

---

0.8.2) `HelloWorld renderIntoDocument`  
should render the component FAILED ReferenceError: `HelloWorld` is not defined

---

`HelloWorld` 컴포넌트를 정의하지 않아서 테스트가 실패했다. 아직 컴포넌트를 작성하지 않았으니 올바른 결과다. 테스트에 사용할 간단한 React 컴포넌트를 만들어보자.

---

```
/** @jsx React.DOM */
var React = require("react");
var HelloWorld = React.createClass({
  render: function(){
    return (
      <div></div>
    );
  }
});

module.exports = HelloWorld;

var React = require("react/addons");
var TestUtils = React.addons.TestUtils;

var HelloWorld = require(' ../../client/testing_examples/hello_world');

describe("HelloWorld", function(){
  describe("renderIntoDocument", function(){
    ...
  })
});
```

---

**NOTE KARMA: SINGLE RUN**

코드를 수정하고 터미널을 확인해보면 테스트가 이미 재실행되고 있는 것을 확인할 수 있다. 이것 은 Karma가 프로젝트 파일에 변경이 있으면 재실행되도록 이미 설정되어 있기 때문이다. 설정은 karma.conf.js에서 확인할 수 있다.

`HelloWorld` 컴포넌트를 정의하고 명세에서 `require` 했기 때문에, 이제 테스트 를 통과할 것이다.

---

```
Chrome 36.0.1985 (Mac OS X 10.8.2): Executed 1 of 1 SUCCESS
(0.905 secs / 0.801 secs)
```

---

두 가지 명세를 더 추가하자. `renderIntoDocument`가 정상 동작하는지 확인 한다.

1. 렌더링하는 HTML 안에 "Hello World!"가 들어있는지 확인한다.
  2. 렌더링한 컴포넌트를 확인한다.
- 

```
...
it("컴포넌트와 HTML을 DOM 노드에 렌더링 해야한다", function(){
  var myComponent = TestUtils.renderIntoDocument(<HelloWorld />);

  // 렌더링된 HTML을 검사한다
  expect(myComponent.getDOMNode().textContent).toContain("Hello World!");
});

it("컴포넌트를 렌더링하고 컴포넌트를 반환한다", function(){
  var myComponent = TestUtils.renderIntoDocument(<HelloWorld />);

  // you can assert things on the component which was rendered
  // 렌더링한 컴포넌트를 확인한다
  expect(myComponent.props.name).toBe("Bleeding Edge React.js Book");
});
...
...
```

---

터미널을 보면 예상대로 테스트 실패 메시지 두 개를 확인할 수 있다.

---

```
Chrome 36.0.1985 (Mac OS X 10.8.2) HelloWorld renderIntoDocument should
render the component and it's html into a dom node FAILED Expected '' to
contain 'Hello World!'. Error: Expected '' to contain 'Hello World!'.
```

```
...
```

```
Chrome 36.0.1985 (Mac OS X 10.8.2) HelloWorld renderIntoDocument should
render the component and return the component as the return value FAILED
Expected undefined to be 'Bleeding Edge React.js Book'. Error: Expected
undefined to be 'Bleeding Edge React.js Book'
```

---

HelloWorld 컴포넌트를 수정해서 테스트를 통과하자.

---

```
it("will never pass if you try to assert on a whole dom node", function(){
  var HelloWorld = React.createClass({
    getDefaultProps: function(){
      return {
        name: "Bleeding Edge React.js Book"
      };
    },
    render: function(){
      return (
        <div>
          <h1>Hello World!</h1>
          <h2 className="subheading">{this.props.name}</h2>
        </div>
      );
    }
  });
});
```

---

하지만 이 명세는 테스트를 통과할 수 없다. subheading이라는 클래스가 있는 DOM 엘리먼트가 없기 때문이다. 다음과 같이 수정하면 이 명세를 통과할 수 있다.

---

```
...
render: function(){
  return (
    <div>
      <h1>Hello World!</h1>
      <h2 className="subheading">{this.props.name}</h2>
    </div>
  );
}
...
...
```

---

### NOTE React와 HTML 확인

테스트를 살펴보다가 왜 다음에 보이는 것처럼 테스트를 작성하지 않는지 궁금해 하는 독자가 있을 것이다.

```
var myComponent = TestUtils.renderIntoDocument(<HelloWorld />);
// 이렇게 하면 작동하지 않는다!
expect(myComponent.getDOMNode().innerHTML).toContain("<h2
class='subheading'>Bleeding Edge React.js Book</h2>");
});
```

아마 jQuery나 Backbone.js 애플리케이션은 이렇게 테스트해도 잘 작동할 것이다. 물론 추천하고 싶은 방법은 아니다. React는 이렇게 테스트할 수 없다. `render` 함수에 정의한 HTML은 DOM에 렌더링된 HTML이 아니기 때문이다. 실제 화면에 렌더링하는 DOM은 다음의 모습을 하고 있다.

```
<h1 data-reactid=".1k.0">Hello World!</h1>
<h2 class="subheading" data-reactid=".1k.1">Bleeding Edge React.js Book</h2>
```

React는 `data` 속성을 사용해서 뛰어난 재렌더링 성능을 보여준다. Ember.js, Angular.js 프레임워크도 비슷한 이유로 프레임워크 자체의 `data` 속성을 사용한다.

## 15.3 모의 컴포넌트

React가 가진 강점 중 하나가 앞에서도 살펴본 것처럼 컴포넌트 안에 다른 컴포넌트를 구성할 수 있다는 점이다. 하나의 컴포넌트 안에서 다른 컴포넌트를 렌더링할 수 있으므로 모듈화와 코드 재사용에 매우 효과적이다. 하지만 이로 인해 테스트할 때 중요하게 고려해야 할 부분이 있다. `UserBadge`와 `UserImage`라고 하는 두 개의 컴포넌트가 있다고 하자. `UserBadge`는 사용자의 이름과 `UserImage`를 렌더링한다. `UserBadge` 컴포넌트에 대한 테스트 명세는 `UserBadge`의 기능만 테스트하고, `UserImage`의 기능은 테스트하지 않아야 한다. 아마 동료 개발자 중에는 두 컴포넌트를 함께 테스트하는 것이 더욱 현실적이라고 이야기하는 사람

도 있을지 모르겠다. 하지만 이렇게 하면 테스트 대상에 집중할 수 없다. 결국, 테스트 작성이 점점 더 어려워지고 유지 보수도 힘들어진다.

**NOTE** 테스트에 귀를 기울이면!

테스트가 너무 복잡해지고 목적이 불분명해지는 것이 걱정이라면, 이 징후를 계속해서 살펴볼 수 있다. 특히 설정 과정이 복잡하거나 반복해서 나타난다면 이런 테스트는 코드 스멜<sup>code smell</sup>이라고 할 수 있다. 이런 테스트는 경계해야 한다. 테스트를 작성하는 것이 고통스러울수록 아키텍처에 최적화하지 않은 부분이 있을 가능성이 크다. 테스트를 작성하다 보면 종종 이런 문제를 발견할 수 있다.

UserBadge 테스트 공략 계획은 이렇다. UserBadge 컴포넌트를 렌더링하기 전에 아무 동작도 하지 않는 UserImage 컴포넌트를 모의 컴포넌트로 교체해야 한다. 애플리케이션 개발에 사용 중인 도구와 아키텍처에 많이 의존하는 방법이다. 설문조사 생성기는 Browserify를 이용했다. UserBadge 테스트와 render 함수 호출을 위한 기본 코드를 살펴보자.

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

var UserBadge = require('../client/testing_examples/user_badge');

describe("UserBadge", function(){
  // 의도와는 다르게 실제 UserImage 컴포넌트를 렌더링한다.
  it("실제 컴포넌트 대신 Mock 컴포넌트를 사용해야 한다", function(){
    var userBadge = TestUtils.renderIntoDocument(<UserBadge />);
  });
});
```

**NOTE**

예제로 사용하는 설문조사 생성기의 전체 소스 코드는 Github 저장소에서 확인할 수 있다.

<https://github.com/backstopmedia/bleeding-edge-sample-app>

rewireify라는 오픈 소스 모듈을 사용해서 `UserImage` 컴포넌트를 테스트에서 제외한다. 이 모듈을 이용하면 지역 변수와 지역 함수를 개별 모듈에서 다시 쓸 수 있다. `UserBadge` 모듈의 소스를 보면 다음과 같은 내용을 확인할 수 있다.

---

```
var UserImage = require("./user_image");
...
render: function(){
  return (
    <div>
      <h1>{this.props.friendlyName}</h1>
      <UserImage slug={this.props.userSlug} />
    </div>
  );
}
...
...
```

---

`UserBadge` 모듈은 React 컴포넌트인 `UserImage`를 지역 변수로 참조한다. `UserBadge` 명세에서 rewireify를 이용해서 지역 변수인 `UserImage`를 모의 컴포넌트로 교체할 것이다. 다음 코드를 살펴보자.

---

```
var mockUserImageComponent = React.createClass({
  render: function(){
    return (<div className="fake"> 가짜 사용자 이미지!!</div>);
  }
});

UserBadge._set_("UserImage", mockUserImageComponent);
```

---

이제 `UserBadge`가 렌더링 될 때 실제 `UserImage` 컴포넌트가 아닌 `mockUserImageComponent`를 렌더링한다. 앞의 예제는 한 가지 중요한 사이드 이펙트를 놓치고 있다. 바로 테스트 오염 `test pollution`이다. 개별 테스트에서 `_set_` 메소드로 변수를 설정한 후에는 다음 테스트 전에 이 설정을 반드시 되돌려야 한다.

다음과 같은 방법으로 다음 테스트의 오염을 막을 수 있다.

---

```
describe("UserBadge", function(){
  describe("rewireify", function(){
    var mockUserImageComponent;

    beforeEach(function(){
      mockUserImageComponent = React.createClass({
        render: function(){
          return (<div className="fake">Fake User Image!!</div>);
        }
      });
    });

    describe("using just rewireify", function(){
      var realUserImageComponent;

      beforeEach(function(){
        // 실제 선언을 넣어두고 테스트를 마치면 둘러놓는다
        realUserImageComponent = UserBadge.__get__("UserImage");
        UserBadge.__set__("UserImage", mockUserImageComponent);
      });

      afterEach(function(){
        UserBadge.__set__("UserImage", realUserImageComponent);
      });

      it("실제 컴포넌트 대신 Mock 컴포넌트를 사용해야 한다", function(){
        var userBadge = TestUtils.renderIntoDocument(<UserBadge />);

        expect(TestUtils.findRenderedDOMComponentWithClass(userBadge,
        "fake").getDOMNode().innerHTML).toBe("가짜 사용자 이미지!!");
      });
    });
  });
});
```

---

**NOTE TESTUTILS.FINDRENDEREDDOMCOMPONENTWITHCLASS**

이 테스트에서는 `TestUtils.findRenderedDOMComponentWithClass`라는 도구를 사용했다. 이 도구의 동작은 이 장의 뒷부분에서 다룰 것이다. 여기서는 이 도구가 `class`로 `fake`가 입력된 컴포넌트를 찾아준다는 점만 알아두자.

앞에서 살펴본 테스트 코드의 기본 알고리즘은 다음과 같다.

1. `mockUserImageComponent` 컴포넌트를 정의한다
2. `UserBadge` 모듈의 변수 `UserImage`의 값을 지역 변수 `realUserImageComponent`에 저장한다.
3. `mockUserImageComponent`를 `UserImage`에 할당한다
4. 테스트를 수행한다.
5. `UserImage`를 다시 `realUserImageComponent`로 되돌린다.

이 방식은 잘 동작한다. 하지만 우리가 작성할 대부분 명세가 다른 컴포넌트를 렌더링할 것이라는 점을 생각해 보면 지금은 준비 코드가 너무 많다. 이런 상황에서는 `Jasmine` 도구를 이용하면 좋다. 변수를 변경했다가 테스트가 끝나면 원래대로 돌려놓는 모듈을 작성해보자. 변경사항을 모두 배열에 넣어두면 반복문을 이용해서 변경사항을 되돌릴 수 있다. 다음 코드는 이런 목적으로 `test/client/helpers/rewire-jasmine.js`에 작성한 모듈이다.

---

```
var rewires = [];
var rewireJasmine = {
  rewire: function(mod, variableName, newValue){
    // 실제 값을 저장하여 되돌릴 수 있게 만든다
    var originalVariableValue = mod._get_(variableName);

    // 이 모듈에서 변경한 사항을 모두 추적한다
    rewires.push({
      mod: mod,
      variableName: variableName,
      originalVariableValue: originalVariableValue,
    });
  }
};
```

```

        newVariableValue: newVariableValue
    });

    // 변수를 새로운 값으로 변경한다
    mod._set_(variableName, newVariableValue);
},
};

unwireAll: function(){
    for (var i = 0; i < rewires.length; i++) {
        var mod = rewires[i].mod,
            variableName = rewires[i].variableName,
            originalVariableValue = rewires[i].originalVariableValue;

        // 변수명을 원래 변수명으로 변경한다
        mod._set_(variableName, originalVariableValue);
    }
}
};

afterEach(function(){
    // 모든 변경을 취소한다
    rewireJasmine.unwireAll();

    // 다음 명세에서 사용할 수 있게 배열을 비운다
    rewires = [];
});

module.exports = rewireJasmine;

```

---

이 모듈을 사용해서 UserBadge 예제를 다음처럼 간단하게 정리했다.

---

```

var rewireJasmine = require("../helpers/rewire-jasmine");
var UserBadge = require('../client/testing_examples/user_badge');

describe("UserBadge", function(){
    describe("커스텀 rewireify 헬퍼를 사용한다.", function(){
        beforeEach(function(){
            rewireJasmine.rewire(UserBadge, "UserImage", mockUserImageComponent);
        });
    });
});

```

```
it("실제 컴포넌트가 아닌 Mock 컴포넌트를 사용한다", function(){
    var userBadge = TestUtils.renderIntoDocument(<UserBadge />);

    expect(
        TestUtils
            .findRenderedDOMComponentWithClass(userBadge, "fake")
            .getDOMNode()
            .innerHTML
    ).toBe("가짜 사용자 이미지!!");
});

});
```

코드가 훨씬 간결해졌다. `rewireJasmine.rewire(UserBadge, "UserImage", mockUserImageComponent);`가 변수값 변경과 테스트 종료 후 뒷정리를 모두 처리한다.

**NOTE** 다른 NPM 구현법

Browserify 대신 Webpack을 사용해서 클라이언트 코드를 관리하고 있다면 `rewireify` 대신 `rewire-webpack`을 사용한다. 클라이언트가 아닌 Node.js 애플리케이션을 테스트하는 경우라면 `rewire`를 사용할 수 있다. `rewireify`나 `rewire-webpack` 모두 `rewire`에서 시작되었으며 인터페이스에 약간의 차이가 있을 뿐이다. 다음에서 더 자세한 내용을 살펴보자.

`npm`이나 `require`를 사용하지 않고 `<script>` 태그로 코드를 불러와서 컴포넌트를 전역변수에 저장하는 프로젝트도 있을 수 있다. 이럴 때에도 대처할 방법이 있다. 방식은 같지만, 코드는 조금 다르다.

```
describe("global variables", function(){
    var mockUserImageComponent, realUserImageComponent;

    beforeEach(function(){
        mockUserImageComponent = React.createClass({
            render: function(){
                return (<div className="fake">Fake Vanilla User Image!!</div>);
            }
        });
    });

    it("실제 컴포넌트가 아닌 Mock 컴포넌트를 사용한다", function(){
        var userBadge = TestUtils.renderIntoDocument(<UserBadge />);

        expect(
            TestUtils
                .findRenderedDOMComponentWithClass(userBadge, "fake")
                .getDOMNode()
                .innerHTML
        ).toBe("가짜 사용자 이미지!!");
    });
});
```

```

});
```

```

// we need to save off the real definition, so we can put it back when
the test is complete
// 실제 값을 저장해뒀다가 테스트가 끝나면 되돌린다.
realUserImageComponent = window.vanillaScriptApp.UserImage;
window.vanillaScriptApp.UserImage = mockUserImageComponent;
});
```

```

afterEach(function(){
    window.vanillaScriptApp.UserImage = realUserImageComponent;
});
```

```

it("실제 컴포넌트 대신 Mock 컴포넌트를 사용해야 한다", function(){
    var UserBadge = window.vanillaScriptApp.UserBadge;
    var userBadge = TestUtils.renderIntoDocument(<UserBadge />);

    expect(
        TestUtils
            .findRenderedDOMComponentWithClass(userBadge, "fake")
            .getDOMNode()
            .innerHTML
    ).toBe("Fake Vanilla UserImage!!");
});
```

```

});
```

---

지금까지 다음 내용을 배웠다.

1. 컴포넌트를 document에 렌더링하는 방법
2. 중첩 컴포넌트를 Mock 컴포넌트로 대체하는 방법

## 15.4 함수를 스파이 객체로 만들기

이번에는 테스트하려는 모듈이 가지고 있는 함수를 스파이 객체로 만드는 방법을 알아보자. 어떤 함수를 스파이 객체로 만드는 데는 여러 가지 이유가 있다.

1. 해당 메소드가 실제 구현대로 동작하지 않게 함으로써 메소드를 독립적으로 테스트할 수 있다.
2. 해당 메소드를 실제 구현대로 동작하지 않게 만들어서 메소드가 의존하는 API, 외부 서비스 등을 테스트 수트<sup>suite</sup>에서 제외할 수 있다.
3. stub을 호출한 함수나 인자를 확인할 수 있다.

다음 jasmine을 사용한 예제를 통해 "foo" 함수에 스파이를 사용한 방법을 살펴보자.

---

```
var myModule = {
  foo: function(){
    return 'bar';
  }
};

spyOn(myModule, "foo").andReturn('fake foo');
```

---

이것을 React 컴포넌트에서 시도한다면 다음과 같을 것이다.

---

```
var myComponent = React.createClass({
  foo: function(){
    return 'bar';
  },
  render: ...
});

spyOn(myComponent.prototype, "foo").andReturn('fake foo');
```

---

하지만 작동하지 않는다. 이유는 이렇다.

1. React는 프로토타입에 함수를 저장하지 않는다(Backbone.js와 유사하다).
2. React는 자동 바인딩 같은 고급 기능을 제공하기 위해서 함수를 하나 이상 복제하여 보관한다.

3. 이를 해결하는 방법은 React 버전에 따라 다르고, 이 차이는 앞으로 엉망이 될 수 있다.

`jasmineReactHelpers` 모듈을 사용해서 이 문제를 해결할 수 있다. `HelloRandom`이라는 컴포넌트를 테스트한다고 하자. 이 책의 저자를 무작위로 출력하는 컴포넌트다. 이 컴포넌트를 테스트하기 위해서 무작위에 관한 설명은 접어두자. 다음 코드에서 `HelloRandom` 컴포넌트를 살펴본다.

---

```
/** @jsx React.DOM */
var React = require("react");
var authors = [
  { name: "Frankie Bagnardi", githubUsername: "brigand" },
  { name: "Jonathan Beebe", githubUsername: "somethingkindawierd" },
  { name: "Richard Feldman", githubUsername: "rtfeldman" },
  { name: "Tom Hallett", githubUsername: "tommyh" },
  { name: "Simon Hojberg", githubUsername: "hojberg" },
  { name: "Karl Mikkelsen", githubUsername: "karlmikko" }
];

var HelloRandom = React.createClass({
  getRandomAuthor: function(){
    return authors[Math.floor(Math.random() * authors.length)];
  },
  render: function(){
    var randomAuthor = this.getRandomAuthor();

    return (
      <div>저자 {randomAuthor.name}의 github 사용자명은 {randomAuthor.githubUsername}이다.
      </div>
    );
  }
});

module.exports = HelloRandom;
```

---

렌더링 함수를 테스트하는 명세를 작성하자. 테스트는 HTML 문자열의 유효성을 검증한다. 명세는 아마 이럴 것이다.

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

var HelloRandom = require('../client/testing_examples/hello_random');

describe("HelloRandom", function(){
  describe("render", function(){
    it("저자 정보를 출력해야 한다", function(){
      var myHelloRandom = TestUtils.renderIntoDocument(<HelloRandom />);

      expect(
        myHelloRandom.textContent
      ).toBe("Frankie Bagnardi is an author and their github handle is brigand. 저자 Frankie Bagnardi의 gihub 사용자 명은 brigand이다.");
    });
  });
});
```

이 명세는 컴포넌트가 무작위로 출력한 저자가 누군지에 따라서 테스트의 성공여부가 달라지는 문제가 있다.

**NOTE “무작위는” 예시를 위한 설정이다**

여기에서는 스파이 객체로 만들려는 함수의 예로 `randomAuthor`를 사용하고 있다. 이 함수는 실제 애플리케이션에서 다음과 같은 기능을 수행한다.

1. 서버에서 데이터를 가져온다.
2. 테스트에서 설정하기 어려운 컴포넌트의 `state` 값을 사용한다.
3. 이 함수의 동작을 `stub`으로 만들면 사이드 이펙트가 많이 발생한다.
4. 현재 시각이나 시간대에 기반을 둔 데이터를 처리한다.

그래서 `HelloRandom` 컴포넌트의 `getRandomAuthor` 함수를 스파이로 만들어 실제 저자를 반환하는 대신에 가짜 저자를 반환하게 했다.

```
...
var jasmineReact = require("jasmine-react-helpers");
```

```
var HelloRandom = require('../client/testing_examples/hello_random');
...
it("react 클래스의 함수를 스파이 객체로 만들 수 있어야 한다.", function(){
  jasmineReact
    .spyOnClass(HelloRandom, "getRandomAuthor")
    .andReturn({name: "Fake User", githubUsername: "fakeGithub"});

  var myHelloRandom = TestUtils.renderIntoDocument(<HelloRandom />);

  expect(myHelloRandom.getDOMNode().textContent)
    .toBe("저자 가짜 사용자 gibhub 사용자명은 fakeGithub이다.");
});
```

---

jasmine이 제공하는 `spyOn`과 `jasmineReact.spyOnClass`는 같은 값을 반환한다. 따라서 같은 방법으로 `jasmine` 호출을 연결할 수도 있다. 예를 들면 `andReturn( {name: "Fake User", githubUsername: {"fakeGithub"}}` 를 `jasmineReact.spyOnClass`에 연결할 수 있다.

### Assert a spy was called

끝으로 알아볼 내용은 스파이 함수 호출 여부를 확인하는 방법이다. 스파이 함수의 호출 여부를 확인해본 적이 없다면 왜 이걸 알아야 하는지 궁금할 수 있다.

`prop`를 이용해 자식 컴포넌트에 자신의 콜백함수를 전달하는 부모 컴포넌트가 있다고 가정하자. 부모 컴포넌트를 테스트할 때 자식 컴포넌트는 모의객체로 대체할 수 있다. 그렇다 하더라도 두 컴포넌트가 잘 연결되어 있는지 확인해야 한다.

이 내용을 앞에서 살펴본 예제에 접목해보자. `UserBadge`가 부모 컴포넌트, `UserImage`가 자식 컴포넌트가 되고, 부모 컴포넌트는 `props`를 이용해 `imageClicked`를 콜백함수로 전달한다.

---

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

```
 getDefaultProps: function(){
  return {
    friendlyName: "Billy McGee",
    userSlug: "billymcgee"
  };
},
render: function(){
  return (
    <div>
      <h1>{this.props.friendlyName}</h1>
      <UserImage slug={this.props.userSlug} />
    </div>
  );
}
);
};

...

```

---

자식 컴포넌트인 UserImage를 모의객체로 대체하고, 모의객체 컴포넌트에서 imageClicked 함수를 호출하게 만들어보자.

---

```
...
describe("스파이의 호출 여부를 확인한다", function(){
  var mockUserImageComponent;

  beforeEach(function(){
    mockUserImageComponent = React.createClass({
      render: function(){
        return (<div className="fake"> 가짜 사용자 이미지!!</div>);
      }
    });
  });

  rewireJasmine.rewire(UserBadge, "UserImage", mockUserImageComponent);

  it("UserImage 컴포넌트에 콜백 함수인 imageClicked를 전달해야 한다", function(){
    jasmineReact.spyOnClass(UserBadge, "imageClicked");

    var userBadge = TestUtils.renderIntoDocument(<UserBadge />);
    var imageComponent = userBadge.refs.image;
```

```
    imageComponent.props.imageClicked();
    expect(
      jasmineReact
        .classPrototype(UserBadge)
        .imageClicked
    ).toHaveBeenCalled();

  });
}

...

```

---

테스트를 실행하기 전에 테스트 시나리오를 살펴보자.

1. `imageClicked` 함수를 스파이 객체로 만들어 호출 여부를 확인한다.
2. `UserImage` 컴포넌트를 `mockUserImageComponent`로 대체한다.
3. `UserBadge` 컴포넌트를 렌더링한다.
4. `userBadge.refs.image`를 호출하여 `mockUserImageComponent`에 접근한다.
5. `imageComponent.props`에서 `imageClicked` 함수를 호출한다  
(`imageClicked` 함수는 `props`로 전달한다).
6. `UserBadge` 컴포넌트가 정상적으로 함수를 호출하는지 확인한다.

테스트를 실행하면 이런 오류 메시지가 보인다.

---

```
...
PhantomJS 1.9.7 (Mac OS X) HelloRandom assert spy was called
should pass a callback to the imageClicked function to the UserImage
component FAILED imageClicked() method does not exist
...
```

---

`UserImage` 컴포넌트에 `imageClicked` 함수가 없어서 발생한 오류다. 스파이 객체로 만들 함수가 있어야 한다.

```
...
var UserBadge = React.createClass({
  getDefaultProps: function(){
    return {
      friendlyName: "Billy McGee",
      userSlug: "billymcgee"
    };
  },
  imageClicked: function(){
  },
...

```

다시 실행해보자. 이번에는 다른 오류 메시지를 확인할 수 있다.

```
PhantomJS 1.9.7 (Mac OS X) HelloRandom assert spy was called should pass a
callback to the imageClicked function to the UserImage component FAILED
TypeError: 'undefined' is not an object (evaluating 'imageComponent.props')
```

imageComponent가 undefined여서 테스트가 실패했다. UserBadge 컴포넌트에 refs.poster를 설정하지 않았기 때문에 imageComponent가 undefined다. 코드를 추가한다.

```
...
var UserBadge = React.createClass({
  getDefaultProps: function(){
    return {
      friendlyName: "Billy McGee",
      userSlug: "billymcgee"
    };
  },
  render: function(){
    return (
      <div>
        <h1>{this.props.friendlyName}</h1>
      </div>
    );
  }
});
```

```
        <UserImage slug={this.props.userSlug} ref="image"/>
    </div>
);
}

});
```

---

**NOTE**

테스트 편의를 위해 간단하거나 영향을 거의 미치지 않는 코드를 추가하는 것은 괜찮지만, 과도하게 사용하는 것은 권장하지 않는다. 간단한 대체재를 고려하는 게 좋다. 앞의 경우에는 React.addons.TestUtils.findRenderedComponentWithType(userBadge, UserImage)를 사용할 수 있지만 이 메소드에 대해서는 뒤에서 더 자세히 다룬다. 우선 여기서는 ref를 사용하도록 하자.

테스트를 실행하면 다음과 같은 오류 메시지가 등장한다.

---

```
PhantomJS 1.9.7 (Mac OS X) HelloRandom assert spy was called should pass a
callback to the imageClicked function to the UserImage component FAILED
TypeError: 'undefined' is not a function (evaluating 'imageComponent.props.
imageClicked()')
```

---

UserBadge 컴포넌트가 props로 UserImage 컴포넌트에 imageClicked를 전달하지 않아서 오류가 발생했다. 오류를 수정해보자.

---

```
...
var UserBadge = React.createClass({
  get defaultProps: function(){
    return {
      friendlyName: "Billy McGee",
      userSlug: "billymcgee"
    };
  },
  imageClicked: function(){
  },
});
```

```
render: function(){
  return (
    <div>
      <h1>{this.props.friendlyName}</h1>
      <UserImage slug={this.props.userSlug} imageClicked={this.
imageClicked} ref="image"/>
    </div>
  );
}

});

});
```

---

이제 테스트를 통과한다. 부모 컴포넌트가 자식 컴포넌트를 제대로 생성하는 것을 확인했다.

## 15.5 이벤트 시뮬레이션

대부분의 React 컴포넌트는 클릭 이벤트나 품 이벤트 같은 브라우저 이벤트에 반응한다. 이런 시나리오를 유닛 테스트에서 소화하려면, 브라우저 이벤트를 시뮬레이션할 수 있어야 한다.

### SIMULATE 테스트 도구

Simulate는 React 테스트 도구 중에 가장 유용하다. – React 공식 문서

ClickMe 명세를 작성하면서 이 코드를 만들어보자.

---

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

describe("ClickMe", function(){

  describe("Simulate.Click", function(){
    it("렌더링되어야 한다", function(){
```

```
    TestUtils.renderIntoDocument(<ClickMe />);
  });
});

});
```

---

ClickMe를 정의하지 않았으므로 테스트는 실패한다. 모듈을 만들고 테스트에 require를 추가한다.

---

```
/** @jsx React.DOM */
var React = require("react");
var ClickMe = React.createClass({
  render: function(){
    return (
      <div></div>
    );
  }
});

module.exports = ClickMe;
...

var TestUtils = React.addons.TestUtils;
var ClickMe = require('../client/testing_examples/click_me');
...
```

---

렌더링에 성공하고 테스트를 통과한다. ClickMe 컴포넌트에 몇 가지 동작을 추가 한다. 우선 클릭 횟수를 보여주자.

---

```
...
describe("ClickMe", function(){
  describe("Simulate.Click", function(){
    var subject;

    beforeEach(function(){
      subject = TestUtils.renderIntoDocument(<ClickMe />);
```

```
});  
  
it("클릭 수를 표시한다", function(){  
  xpect(subject.getDOMNode().textContent).toBe("횟수: 0");  
});  
});  
});
```

---

### SIMULATE 테스트 도구

#### beforeEach 렌더링 패턴

beforeEach 콜백에서 renderIntoDocument를 호출하고 subject라는 변수에 저장하는 것을 확인했다. 이 패턴을 이용하면 명세에 대한 공통 설정이 있는 경우에 테스트마다 매번 renderIntoDocument를 다시 만들 필요가 없다.

render 함수가 <div> 태그만 반환해서 테스트에 실패했다. 다음과 같이 수정하자.

---

```
...  
var ClickMe = React.createClass({  
  render: function(){  
    return (  
      <h1>횟수: 0</h1>  
    );  
  }  
});  
...
```

---

드디어 테스트를 통과했다. 테스트를 통과하기 위해 컴포넌트에 “0”을 직접 입력했지만 부끄러워할 필요 없다. 테스트를 발전시키다 보면 컴포넌트가 사용하는 실제 값이 필요한 때가 온다. 그때 수정하자.

이제 사용자가 <h1> 태그를 클릭했을 때 글자가 변하는지 확인하는 테스트를 만들어보자.

```
...
it("횟수가 증가해야 한다", function(){
expect(subject.getDOMNode().textContent).toBe("횟수: 0");

// click on the <h1> dom node
// <h1>의 DOM 노드를 클릭한다
TestUtils.Simulate.click(subject.getDOMNode());
expect(subject.getDOMNode().textContent).toBe("횟수: 1");
});

...

```

TestUtils.Simulate 도구의 click 함수를 호출하면서 클릭 이벤트를 받는 DOM 노드를 전달하는 과정을 주목한다. 어떤 이벤트 데이터까지 전달해야 한다면 click 함수의 두 번째 인자를 이용한다.

컴포넌트가 어떠한 클릭 이벤트도 리스닝하지 않으므로 이 테스트는 실패한다. 클릭 이벤트를 받을 수 있게 코드를 수정하자.

```
...
var ClickMe = React.createClass({
getInitialState: function(){
return { clicks: 0 };
},
headingClicked: function(){
var clicks = this.state.clicks;
this.setState({clicks: clicks + 1});
},
render: function(){
return (
<h1 onClick={this.headingClicked}> 횟수: {this.state.clicks}</h1>
);
}
});
...

```

테스트에 통과했다. 지금 우리는 TestUtils.Simulate 사용 방법을 익혔다.

## 15.6 finder 메소드로 컴포넌트 탐색하기

지금쯤이면 컴포넌트 렌더링, 스파이 객체, 모의 객체, 컴포넌트 이벤트 시뮬레이션 같은 기본 테스트 방식에 익숙해졌을 것이다. 이제 앞에서 건너뛰었던 개념을 명확히 살펴보도록 하자. 바로 테스트에서 다른 컴포넌트가 렌더링하는 컴포넌트를 찾는 방법이다. TestUtils의 Finder 메소드를 사용하면 유연하고 간결하며 이해하기 쉬운 테스트를 작성할 수 있다. 여기서는 실례로 설명하기 때문에 쉽게 TDD에 적용할 수 있을 것이다.

```
/** @jsx React.DOM */
var React = require("react");
var CompanyLogo = require("./company_logo");
var NavBar = React.createClass({
  render: function(){
    return (
      <div>
        <CompanyLogo />
        <ul>
          <li className="tab active">Tab 1</li>
          <li className="tab">Tab 2</li>
          <li className="tab">Tab 3</li>
          <li className="tab">Tab 4</li>
          <li className="tab">Tab 5</li>
        </ul>
      </div>
    );
  }
});

module.exports = NavBar;

/** @jsx React.DOM */
var React = require("react");
var CompanyLogo = React.createClass({
  render: function(){
    return ();
  }
})
```

```
});  
  
module.exports = CompanyLogo;
```

---

<NavBar> 컴포넌트가 생성하는 모든 <li> 컴포넌트를 찾아야 하는 경우에는 TestUtils.scryRenderedDOMComponentsWithTag 함수를 사용해야 한다.

**NOTE** **컴포넌트 vs 엘리먼트**

앞의 설명에서 “모든 <li> 엘리먼트를 찾는다”가 아니라 “모든 <li> 컴포넌트를 찾는다”고 설명했다. 이렇게 설명한 이유는 TestUtils의 finder 메서드가 React 컴포넌트를 반환하기 때문이다. 이렇게 하면 해당 컴포넌트의 모든 일급 React 속성에 접근할 수 있어 매우 유용하다. getDOMNode() 도 이 중 하나로 DOM Node를 다뤄야 할 때에 유용하다.

```
/** @jsx React.DOM */  
var React = require("react/addons");  
var TestUtils = React.addons.TestUtils;  
var NavBar = require('../client/testing_examples/nav_bar');  
var CompanyLogo = require('../client/testing_examples/company_logo');  
  
describe("TestUtils Finders", function(){  
  var subject;  
  
  beforeEach(function(){  
    subject = TestUtils.renderIntoDocument(<NavBar />);  
  });  
  
  describe("scryRenderedDOMComponentsWithTag", function(){  
    it("html 태그가 있는 모든 컴포넌트를 찾아야 한다", function(){  
      var results = TestUtils.scryRenderedDOMComponentsWithTag(subject, "li");  
      expect(results.length).toBe(5);  
      expect(results[0].getDOMNode().innerHTML).toBe("Tab 1");  
      expect(results[1].getDOMNode().innerHTML).toBe("Tab 2");  
    });  
  });  
});
```

---

**NOTE** SCRY?

이 함수의 이름이 왜 이런지, 또 어떻게 읽어야 할지 궁금할 수 있다.

이 함수는 “s”와 “cry”를 연결한 것처럼 읽는다. “sky”랑 발음이 비슷하다. 정확하게 발음하고 싶다면 구글에서 검색해서 발음을 들어보라(역자 주: scry는 “수정으로 점을 친다”는 의미가 있다. 한국어로 옮기다면 “스크라이” 정도로 읽을 수 있다).

Scry는 수정을 들여다보면서 어떤 대상을 찾는다는 뜻이다. 여기서는 React 컴포넌트가 곧 수정으로, 내부의 컴포넌트를 찾는다는 의미로 해석할 수 있다.

모든 scry\* 함수에 유사하게 대응하는 find\* 함수가 존재한다. 이 메서드는 같은 동작을 하지만 배열 대신에 단 하나의 컴포넌트만 반환한다. 인자와 일치하는 컴포넌트가 여러 개면 오류가 발생한다.

<NavBar> 컴포넌트가 렌더링하는 <CompanyLogo>라는 컴포넌트를 찾는 경우를 생각해보자. scryRenderedComponentsWithType 함수를 이용해서 이 컴포넌트를 찾을 수 있다.

```
...
it("결합 DOM 컴포넌트를 찾아야 한다", function(){
    var results = TestUtils.scryRenderedComponentsWithType(subject,
    CompanyLogo);
    expect(results.length).toBe(1);
    // <CompanyLogo>를 렌더링하고 찾아냈다고 하더라고
    // 이것은 결합 컴포넌트이기 때문에 실제로는 <img /> 태그이다.
    expect(results[0].getDOMNode().tagName).toBe("IMG");
});
```

형<sup>type</sup>에 따라서 컴포넌트를 탐색하면 유용하다. 구현 방식에 의존하는 CSS 클래스나 ID를 이용하지 않고서도 컴포넌트를 검사할 수 있기 때문이다.

**NOTE** 컴포넌트와 형<sup>type</sup> 제한

React v11.0부터는 결합 컴포넌트를 대체하는 React.DOM.div나 React.DOM.li 같은 네이티브 컴포넌트에는 scryRenderedComponentsWithType를 사용할 수 없다. 이와 관련해서 앞으로 React가 어떻게 바뀔지 지금은 알 수 없다. 자세한 사항은 다음 링크를 참고하길 바란다.

<https://github.com/facebook/react/issues/1533>

CSS 클래스를 기준으로 컴포넌트를 찾으려고 할 때는 scryRenderedDOMComponentsWithClass 함수를 사용한다.

---

```
...
describe("scryRenderedDOMComponentsWithClass", function(){
  it("클래스가 일치하는 모든 컴포넌트를 찾아야 한다",
    function(){
      var tabs = TestUtils.scryRenderedDOMComponentsWithClass(subject,
      "tab");
      var activeTabs = TestUtils.scryRenderedDOMComponentsWithClass(su
bject, "active");
      expect(tabs.length).toBe(5);
      expect(activeTabs.length).toBe(1);
    });
  });
...
});
```

---

## 15.7 믹스인

책의 전반부에서 React 믹스인을 만드는 과정을 살펴보았다. 믹스인과 컴포넌트는 다르다. 믹스인은 어떻게 단위 테스트를 해야 할까? 믹스인을 테스트하는 방법은 크게 세 가지가 있다.

1. 믹스인 객체를 직접 테스트한다.
2. 믹스인을 추가한 가짜 컴포넌트를 테스트한다.
3. 공유하는 동작을 테스트하는 명세를 작성해서 믹스인을 사용하는 컴포넌트의 명세에서 참조한다.

## 믹스인 직접 테스트하기

믹스인은 간단하게 직접 테스트할 수 있다. 믹스인 객체의 함수를 직접 호출하고 예상대로 동작하는지 검증한다. 이 방법을 이용하면 대체로 매우 잘게 나뉜<sup>fine-grained</sup> 여러 개의 테스트가 만들어진다. 테스트를 위해서 믹스인 내에서 호출하는 React 메서드를 스텁<sup>stub</sup>으로 만들어야 한다. 앞에서 믹스인을 설명하면서 작성한 `IntervalMixin`을 예제로 사용하자. 가장 간단한 함수인 `componentDidMount`를 가지고 시작한다.

---

```
/** @jsx React.DOM */
var IntervalMixin = require ('../../../../client/testing_examples/interval_mixin');

describe("IntervalMixin", function(){
    describe("믹스인을 직접 테스트한다", function(){

        var subject;

        beforeEach(function(){
            // 주의: 이렇게 하면 안된다!! 문제가 발생한다. 발생하는 문제는 아래에서 설명
            // 한다.
            subject = IntervalMixin;
        });

        describe("componentDidMount", function(){
            it("인스턴스에 빈 배열인 __intervals 생성한다", function(){
                expect(subject.__intervals).toBeUndefined();
                subject.componentDidMount();
                expect(subject.__intervals).toEqual([]);
            });
        });
    });
});
```

---

이 테스트는 통과한다. `IntervalMixin`을 `subject`로 사용한 것이 보이는가? 이것은 별로 좋은 방법이 아니다. `componentDidMount` 함수가 `this`에 `subject`를

설정하기 때문이다. 따라서 다음 테스트를 실행하면 IntervalMixin 객체가 오염되어 테스트가 실패한다. 테스트 오염을 확인하기 위해서 첫 번째 테스트를 복사해서 명세를 수정한다.

---

```
...
describe("IntervalMixin", function(){
    describe("믹스인을 직접 테스트한다", function(){
        var subject;

        beforeEach(function(){
            // 주의: 이렇게 하면 안된다!! 문제가 발생한다. 발생하는 문제는 아래에서 설명
            // 한다.
            subject = IntervalMixin;
        });

        describe("componentDidMount", function(){
            it("인스턴스에 빈 배열인 __intervals 생성한다 ", function(){
                expect(subject.__intervals).toBeUndefined();
                subject.componentDidMount();
                expect(subject.__intervals).toEqual([]);
            });
        });

        it("인스턴스에 빈 배열인 __intervals 생성한다 (테스트 오염 확인을 위한 테스
        트)", function(){
            expect(subject.__intervals).toBeUndefined();
            subject.componentDidMount();
            expect(subject.__intervals).toEqual([]);
        });
    });
});
```

---

두 번째 테스트가 `expect(subject.__intervals).toBeUndefined();` 라인에서 실패하는 것을 확인할 수 있다. 첫 번째 테스트를 실행했기 때문이다.

---

```
Chrome 37.0.2062 (Mac OS X 10.8.2) IntervalMixin testing the mixin directly
componentDidMount should set an empty array called _intervals on the
instance (testing for test pollution)
FAILED Expected [ ] to be undefined. Error: Expected [ ] to be undefined.
```

---

이 문제를 해결하려면 테스트 케이스마다 새로운 믹스인 사본을 이용할 수 있게 만들어줘야 한다. `Object.create`를 이용하는 방식으로 `beforeEach`의 코드를 변경했다. 두 테스트에 모두 통과한다.

---

```
...
beforeEach(function(){
    subject = Object.create(IntervalMixin);
});
...
```

이제 `componentDidMount`를 통과했다. `setInterval`을 이용해서 테스트를 하자. 여기서 다루는 믹스인의 `setInterval` 함수의 역할은 다음과 같다.

1. 실제 `setInterval` 함수를 연결하는 창구 역할을 한다.
2. `setInterval` id를 배열에 저장해서 나중에 삭제할 수 있다.
3. `setInterval` id를 반환한다.

테스트는 이렇게 구성할 수 있다.

---

```
...
describe("setInterval", function(){

    var fakeIntervalId;

    beforeEach(function(){
        fakeIntervalId = 555;
        spyOn(window, "setInterval").andReturn(fakeIntervalId);
        // 참고: setInterval을 호출하기 전에 componentDidMount를 호출해서 this._intervals를 정의해야한다. 이것은 믹스인 객체의 함수를 직접 호출하는 방법이 가지고 있는 단
```

점이다.

```
        subject.componentDidMount();
    });

it("callback과 interval을 통해 window.setInterval를 호출해야 한다", function()
{
    expect(window.setInterval.callCount).toBe(0);
    subject.setInterval(function(){}, 500);
    expect(window.setInterval.callCount).toBe(1);
});

it("array setInterval id를 this._intervals에 저장해야 한다", function(){
    subject.setInterval(function(){}, 500);
    expect(subject._intervals).toEqual([fakeIntervalId]);
});

it("setInterval id를 반환해야 한다", function(){
    var returnValue = subject.setInterval(function(){}, 500);
    expect(returnValue).toBe(fakeIntervalId);
});
});

...

```

**NOTE 테스트에서 React 기능 제외하기**

앞의 테스트는 React에서만 사용하는 기능을 포함하고 있지 않다. 따라서 앞의 테스트는 일반적인 jasmine 테스트다. 믹스인 함수가 `this.setState({})`처럼 React가 제공하는 메서드를 호출하기 시작하면 `spyOn(subject, "setState")`를 사용해서 React에서 사용하는 함수를 모의 객체로 대체하는 것이 좋다. 이렇게 하면 믹스인 객체만 테스트할 수 있다.

믹스인을 직접 테스트하면 여러 개로 잘게 쪼개진 fine-grained 테스트가 만들어진다. 복잡한 동작이 늘어날 때에는 이 방법이 유용하지만 때로는 기능이 아닌 구현을 테스트하게 되기도 한다. 또한, 이 방법은 믹스인의 함수를 호출할 때 React 라이프사이클 콜백을 흥내내려면 순서에 신경을 써야 한다. `setInterval` 테스트에서 `subject.componentDidMount()`를 호출하는 경우가 그렇다. 다음 절에서 이런 단점을 해결하는 방법을 살펴본다.

## 페이크 컴포넌트를 이용해 믹스인 테스트하기

가짜 컴포넌트인 페이크<sup>fake</sup> 컴포넌트를 이용해 믹스인을 테스트하기 위해서는 React 컴포넌트를 정의해야 한다. 테스트에서 사용하기 위한 컴포넌트를 만들므로써 실제 애플리케이션에서는 사용하지 않을 것이라는 점을 분명하게 밝힐 수 있다. 다음은 여기서 예제로 사용할 가짜 컴포넌트다. 기능을 단순화해서 테스트를 단순하게 만들었다. 이로써 테스트의 목적이 선명해졌다. React가 render 함수를 사용해야 하는 점은 다소 아쉽다.

---

```
describe("가짜 컴포넌트를 이용해 믹스인을 테스트한다", function(){
    var FauxComponent;

    beforeEach(function(){
        // Jasmine으로 가짜 객체를 선언하는 방법을 잘 살펴본다.
        // 이 컴포넌트를 믹스인을 테스트하기 위한 목적으로 만들었다는 사실을 의도적으로 표현하고
        // 있다.
        FauxComponent = React.createClass({
            mixins: [IntervalMixin],
            render: function(){
                return (<div>가짜 컴포넌트가 대세!</div>);
            },
            myFakeMethod: function(){
                this.setInterval(function(){}, 500);
            }
        });
    });

});
```

---

가짜 컴포넌트를 만들었으니 테스트를 작성한다.

---

```
...
describe("setInterval", function(){
    var subject;

    beforeEach(function(){
        spyOn(window, "setInterval");
    });

});
```

```

        subject = TestUtils.renderIntoDocument(<FauxComponent />);
    });

    it("콜백과 인터벌을 통해 window.setInterval를 호출해야 한다 ", function(){
        expect(window.setInterval.callCount).toBe(0);
        subject.myFakeMethod();
        expect(window.setInterval.callCount).toBe(1);
    });
});

describe("unmounting", function(){

    var subject;

    beforeEach(function(){
        spyOn(window, "setInterval").andReturn(555);
        spyOn(window, "clearInterval");
        subject = TestUtils.renderIntoDocument(<FauxComponent />);
        subject.myFakeMethod();
    });

    it("setTimeout이 있으면 제거한다", function(){
        expect(window.clearInterval.callCount).toBe(0);
        React.unmountComponentAtNode(subject.getDOMNode().parentNode);
        expect(window.clearInterval.callCount).toBe(1);
    });
});
...

```

---

앞에서 살펴봤던 믹스인을 직접 테스트하는 방법과 가짜 컴포넌트를 이용해 믹스인을 테스트하는 방법의 차이점 중 중요한 첫 번째는 가짜 컴포넌트를 먼저 렌더링한 다음에 이에 맞춰 테스트 단언문을 만들어야 한다는 점이다. 이런 큰 차이점 외에도 테스트에 큰 영향을 줄 수 있는 미묘한 차이가 더 있다. 가짜 컴포넌트를 사용하는 방법은 `setInterval`과 `unmounting`에 테스트를 수행하고, 믹스인을 직접 테스트하는 경우에는 `setInterval`, `componentDidMount`, `componentWillUnmount`에 테스트를 수행한다.

이 차이점은 그다지 대수롭지 않게 보일 수 있다. `componentDidMount` 함수를 한번 살펴보자. `this._intervals`를 설정하는 것만으로는 자체적으로 아무 값도 제공하지 않는다. 값을 가지려면 다른 함수가 필요하다. 믹스인을 직접 테스트하는 방법에서는 `this._intervals`에 단언문을 작성해서 함수의 구현을 테스트했다. 여기서 `this._intervals`은 기능이 아니라 세부적인 구현 사항이다. 가짜 컴포넌트를 사용할 때는 `componentDidMount`의 구현을 테스트할 필요가 없다. 왜냐하면 `setInterval` 테스트에서 컴포넌트를 문서에 렌더링할 때 내부에서 함께 테스트하기 때문이다.

왜 `describe`에 `componentWillUnmount`가 아니라 `unmounting`이라고 적었을까? 여기에서 중점을 두고 있는 것이 `clearInterval`의 호출 방법이 아니라 컴포넌트를 해제할 때 `clearInterval`를 호출할지 여부이기 때문이다. 따라서 `subject`를 호출하는 대신에 단순히 컴포넌트를 해제하고 React가 적절한 콜백 함수를 바른 순서대로 호출하게 만든다.

**NOTE 어떤 방법을 선택할까?**

앞에서 살펴본 두 가지 방법 중에 무엇이 더 나은 방법이라고 하기 어렵다. 테스트하려는 믹스인의 복잡도나 동작에 따라 방법이 다르기 때문이다. 다만 믹스인을 직접 테스트하는 방법으로 시작하는 것을 추천한다. 이 방법이 좀 더 집중하기 좋다. 그러다가 테스트를 작성하는 것이 힘들어지면 가짜 컴포넌트를 사용하거나 두 가지 방법을 동시에 사용할 수도 있다. 어떤 방법을 고를지 고민하지 말고 일단 시작하자. 테스트 과정에서 보다 나은 방법을 찾을 수 있을 것이다.

## 공유 동작 명세

앞에서 살펴본 두 가지 방법은 실제 애플리케이션의 컴포넌트는 건드리지 않고서 믹스인 자체만 테스트한다. 이번에 살펴볼 방법은 실제로 믹스인을 사용하는 컴포넌트를 이용해서 믹스인을 테스트한다. 이 접근법을 “**공유 동작**” 명세라고 한다. 대부분의 테스트 명세가 `interval_mixin_spec.js`에 있지 않다는 점이 이 테스트 방법의 가장 큰 특징이다. `since_2014_spec.js`가 테스트를 수행하고 공유 동작은 다른 곳에 저장한다. `Since2014` 명세를 작성해보자.

---

```
/** @jsx React.DOM */
var React = require("react/addons");
var TestUtils = React.addons.TestUtils;
var Since2014 = require ('../../../../client/testing_examples/since_2014');

describe("Since2014", function(){
});
```

---

Since2014 명세를 위한 보일러플레이트를 준비했다. 공유 명세 예제를 추가해보자.

---

```
...
describe("Since2014", function(){
    describe("공유 명세 예제", function(){
        IntervalMixinSharedExamples();
    });
});
...
...
```

---

IntervalMixinSharedExamples를 정의하지 않아서 테스트에 실패한다. 함수를 정의하자.

---

```
...
var Since2014 = require ('../../../../client/testing_examples/since_2014');
var IntervalMixinSharedExamples = require('../../../shared_examples/interval_mixin_
shared_examples');
...
...
```

---

그리고 interval\_mixin\_shared\_examples.js 파일에 보일러플레이트 예제 코드를 추가한다.

---

```
/** @jsx React.DOM */
var React = require("react/addons");
var TestUtils = React.addons.TestUtils;
var SetIntervalSharedExamples = function(attributes){
```

```
var componentClass;

beforeEach(function(){
    componentClass = attributes.componentClass;
});

describe("SetIntervalSharedExamples", function(){
});

};

module.exports = SetIntervalSharedExamples;
```

---

앞의 코드를 자세히 보면 SetIntervalSharedExamples 함수는 여러 개의 jasmine 테스트를 포함하고 있는 함수라는 것을 알 수 있다. Since2014 명세에서 IntervalMixinSharedExamples();를 호출함으로써 이 테스트를 수행한다.

attributes.componentClass도 공유 동작 보일러플레이트의 중요한 부분이다. Since2014 예제에서는 attributes.componentClass를 이용해서 컴포넌트를 테스트에 전달하는 방법으로 의존성을 주입한다.

---

```
...
describe("Since2014", function(){
describe("Since2014", function(){
describe("Since2014", function(){
    describe("shared examples", function(){
        IntervalMixinSharedExamples({componentClass: Since2014});
    });
});
```

---

SetIntervalSharedExamples 명세를 작성해보자.

---

```
/** @jsx React.DOM */
var React = require("react/addons");
var TestUtils = React.addons.TestUtils;
var SetIntervalSharedExamples = function(attributes){
```

```

var componentClass;
beforeEach(function(){
    componentClass = attributes.componentClass;
});

describe("SetIntervalSharedExamples", function(){
    describe("setInterval", function(){

        var subject, fakeFunction;
        beforeEach(function(){
            spyOn(window, "setInterval");
            subject = TestUtils.renderIntoDocument(<componentClass />);
            fakeFunction = function(){};
        });

        it("콜백과 인터벌을 통해 window.setInterval를 호출해야 한다", function()
{
            expect(window.setInterval).not.toHaveBeenCalledWith(fakeFunction,
                jasmine.any(Number));
            subject.setInterval(fakeFunction, 100);
            expect(window.setInterval).toHaveBeenCalledWith(fakeFunction,
                jasmine.any(Number));
        });
    });
});

describe("unmounting", function(){
    var subject, fakeFunction;
    beforeEach(function(){
        fakeFunction = function(){};
        spyOn(window, "setInterval").andCallFake(function(func,
interval){
// 이 명세에서 발생한 setInterval 호출을 확인하는 단언문을 만들어야 한다.
// 이 믹스인을 사용하는 다른 컴포넌트는 테스트 대상이 아니다.
// 따라서 setInterval 호출이 "fakeFunction"이 다른 id 번호를 반환하게 강제한다.
        if(func === fakeFunction){
            return 444;
        } else {
            return 555;
        }
    });
});

```

```
    });
    spyOn(window, "clearInterval");
    subject = TestUtils.renderIntoDocument(<componentClass />);
    subject.setInterval(fakeFunction, 100);
});

it("setTimeout이 있으면 제거한다", function(){
    expect(window.clearInterval).not.toHaveBeenCalledWith(444);
    React.unmountComponentAtNode(subject.getDOMNode());
    parentNode);
    expect(window.clearInterval).toHaveBeenCalledWith(444);
});
});
});
};

module.exports = SetIntervalSharedExamples;
```

---

**NOTE** 공유 명세

기억해야 할 중요한 차이가 한 가지 있다. Since2014와 관련한 동작은 반드시 since\_2014\_spec.js에만 있어야 하고, IntervalMixin 믹스인을 이용해서 만든 Since2014의 기능은 반드시 interval\_mixin\_shared\_examples.js 명세 파일에서 테스트해야 한다.

공유 동작에 대한 명세가 가짜 컴포넌트의 명세와 유사한 것을 알 수 있다. 다만 공유 동작 명세가 조금 더 복잡하다. 가짜 컴포넌트 예제에서는 이것을 다음처럼 unmounting 명세에서 할 수 있다.

---

```
spyOn(window, "setInterval").andReturn(555);
```

---

그렇지만 공유 동작 예제에서는 unmounting 명세에서 해야 한다.

---

```
spyOn(window, "setInterval").andCallFake(function(func, interval){
    if(func === fakeFunction){
        return 444;
```

```
    } else {
      return 555;
    }
});
```

---

Since 2014 컴포넌트가 `setInterval`을 호출할 때 공유 동작 명세의 `setInterval`을 추가로 호출하기 때문에 테스트가 복잡해졌다. 따라서 이 둘을 구분해야 한다. 그렇지 않으면 공유 동작 명세로 믹스인을 제대로 테스트하지 못 할 수 있다. 이는 공유 동작을 테스트하는 방식이 가짜 컴포넌트를 이용하는 경우 보다 더 복잡하고 잡음도 많다는 뜻이다. 다음 같은 경우에는 이런 복잡함을 감수 할 만하다.

- 믹스인이 React 컴포넌트에 특정한 함수나 동작에 의존하는 경우, React 컴포넌트가 해당 함수나 동작을 가졌는지 공유 동작 명세를 이용해 검증할 수 있다(예: 믹스인 인터페이스와 함께 테스트함).
- 믹스인이 제공하는 동작을 컴포넌트가 쉽게 덮어쓰거나 망가뜨릴 수 있는 경우 공유 동작 명세를 이용하면 이런 문제를 방지할 수 있다.

여기까지 살펴본 세 가지의 테스트 방법은 모두 장단점을 가지고 있다. 믹스인을 테스트할 때는 어떤 방법이든 일단 선택해서 진행하다가, 뜻대로 되지 않는다 싶을 때 다른 방법으로 변경한다. 믹스인의 각 부분을 다른 방식으로 테스트하는 것도 좋은 방법이다.

## 15.8 <body>에 렌더링 하기

여기까지 React 컴포넌트를 테스트하는 방법에 대해서 많은 부분을 살펴봤다. 이제 다시 첫 번째 주제였던 컴포넌트 렌더링을 좀 더 자세히 살펴보자. `React.addons.TestUtils.renderIntoDocument` 함수는 `renderIntoBody`나 `render`가 아니라 `renderIntoDocument`라는 이름을 가지고 있다. React를 만

든 사람은 이 함수가 메모리 상에 존재하는 별개의 HTML Document에 컴포넌트를 렌더링한다는 점을 분명하게 밝히려는 의도에서 이런 이름을 지었다. 따라서 컴포넌트가 <body> 태그 안에 들어가지 않으며 jasmine 테스트 화면에서 보이지도 않는다. 이 방법은 대부분 애플리케이션에서 문제가 없다. 많은 개발자가 선호하는 방법이기도 하다. 하지만 테스트를 위해 <body>에 컴포넌트를 삽입하고 화면에서 확인해야 한다면 어떨까?

이런 시나리오를 제대로 테스트하기 위해서는 `React.renderComponent`를 사용해야 한다. 이 함수는 컴포넌트를 DOM 엘리먼트에 렌더링한다. 렌더링한 엘리먼트는 실제 애플리케이션 코드처럼 <body> 태그 안에 위치한다. 이 과정에서 부모 DOM 엘리먼트의 상태를 제거한다는 데 주의를 기울여야 한다. 그렇지 않으면 렌더링한 엘리먼트가 테스트를 오염시켜 이후의 테스트에 영향을 미친다. 이 기법을 이해하기 위해 컴포넌트의 높이와 너비를 픽셀로 반환하는 컴포넌트를 만들었다. 아주 간단하다. 예제를 살펴보자.

---

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

describe("Footprint", function(){
  describe("render", function(){
    it("컴포넌트의 너비를 보여줍니다", function(){
      React.renderComponent(<Footprint />);
    });
  });
});
```

---

이 테스트는 `ReferenceError: Footprint is not defined` 참조오류: Footprint를 정의하지 않았습니다.라는 메시지를 보여주며 실패한다. 오류를 고쳐보자.

...

```
var HelloWorld = require('../client/testing_examples/hello_world');
var Footprint = require('../client/testing_examples/footprint');
...

/** @jsx React.DOM */
var React = require("react");
var Footprint = React.createClass({
  render: function(){
    return (
      <div></div>
    );
  }
});

module.exports = Footprint;
```

---

이 테스트를 실행하면 Error: Invariant Violation: \_registerComponent(...): Target container is not a DOM element. 오류: 불변식 위반: \_registerComponent(...) 대상 컨테이너가 DOM 엘리먼트가 아닙니다.라는 오류가 발생한다.

TestUtils.renderIntoDocument와 다르게, React.renderComponent는 두 번째 인자로 렌더링할 DOM 엘리먼트를 받는다. 이 문제를 해결하기 위해서 명세에 <div>를 만들어서 <body>안에 추가한다.

---

```
// 주의: 이 테스트는 미완성으로 테스트 오염과 관련한 이슈가 있으니 따라서는 안됩니다.
it("컴포넌트의 너비를 보여줘야 한다", function(){
  var el = document.createElement("div");
  document.body.appendChild(el);
  React.renderComponent(<Footprint />, el);
});
```

---

이제 테스트가 컴포넌트를 실제 DOM 노드에 성공적으로 렌더링했다. 컴포넌트의 너비를 HTML로 출력한 결과를 검증할 수 있게 테스트를 개선하자.

---

```
// 주의: 이 테스트는 미완성으로 테스트 오염과 관련한 이슈가 있으니 따라해서는 안 됩니다.
it("컴포넌트의 너비를 보여줘야 한다", function(){
    var el = document.createElement("div");
    document.body.appendChild(el);
    var myComponent = React.renderComponent(<footprint>/footprint>, el);
    expect(myComponent.getDOMNode().textContent).toContain("컴포넌트 너비: 100");
});
```

---

이 테스트를 실행하면 Expected '' to contain 'component width: placeholder-value'. 해당 위치에 component width라고 입력되어 있었을 것으로 기대하는 오류 메시지를 볼 수 있다. 기능을 추가하자.

---

```
var Footprint = React.createClass({
  getInitialState: function(){
    return { width: undefined };
  },
  componentDidMount: function(){
    var componentWidth = this.getDOMNode().offsetWidth;
    this.setState({width: componentWidth});
  },
  render: function(){
    var divStyle = {width: "100px"};
    return (<div style={divStyle}>component width:{this.state.width}</div>);
  }
});
```

---

테스트에 통과했다. 하지만 아직 끝나지 않았다. 다른 테스트에서도 지금 렌더링한 컴포넌트를 사용하므로 다음 테스트를 시작하기 전에 반드시 컴포넌트를 언마운트해야 한다. 제거하는 과정을 명세에 추가한다.

---

```
describe("Footprint", function(){
  describe("render", function(){
    var el;
    beforeEach(function(){
```

```
    el = document.createElement("div");
    document.body.appendChild(el);
});
afterEach(function(){
    // React가 컴포넌트를 언마운트하여 제거한다.
    React.unmountComponentAtNode(el);
    // <div id="content"></div> 도 함께 제거해서
// beforeEach 함수가 개별 테스트마다 새로 만들게 한다
    el.parentNode.removeChild(el);
});

it("컴포넌트의 너비를 보여줍니다 ", function(){
    var myComponent = React.renderComponent(<Footprint />, el);
    expect(myComponent.getDOMNode().textContent).toContain("컴포넌트 너
비: 100");
});
});
});
});
```

---

이제 컴포넌트를 DOM 안에 성공적으로 렌더링했고, 테스트 오염을 걱정하지 않아도 된다. 이 기능이 자주 사용한다면 `jasmineReactHelpers`를 이용하면 편하다. `jasmineReactHelper`는 각각의 테스트가 끝날 때마다 컴포넌트를 자동으로 언마운트한다. 다음 예제에 `jasmine-react-helpers`의 `renderComponent`를 사용하는 방법이 나와 있다.

---

```
...
var jasmineReact = require("jasmine-react-helpers");
...
describe("jasmineReact.renderComponent", function(){
    var el;
    beforeEach(function(){
        // DOM 엘리먼트를 <body> 안에 추가한다
        el = document.createElement("div");
        document.body.appendChild(el);
    });
    afterEach(function(){
        // <div></div> 도 함께 제거해서
```

```

// beforeEach 함수가 개별 테스트에 따라 새로 만들게 한다
el.parentNode.removeChild(el);
});
it("마운트한 컴포넌트를 반환한다", function(){
var myComponent = jasmineReact.renderComponent(<HelloWorld />, el);
// 컴포넌트에 단언문을 추가할 수 있다
expect(myComponent.props.name).toBe("Bleeding Edge React.js Book");
});

it("컴포넌트를 DOM 안에 배치한다", function(){
var myComponent = jasmineReact.renderComponent(<HelloWorld />, el);
// DOM 노드의 너비와 높이가 실제 값인지 확인한다
expect(myComponent.getDOMNode().offsetWidth).not.toBe(0);
expect(myComponent.getDOMNode().offsetHeight).not.toBe(0);
});
});

```

---

`React.unmountComponentAtNode ( el );`를 호출하지 않는다는 것이 이전과 다른 중요한 차이점이다. `jasmineReactHelpers`는 렌더링하는 것은 무엇이든 테스트가 끝나면 제거한다.

**NOTE** <BODY>에 렌더링할 필요가 있을까?

DOM 제거와 테스트 오염에 대한 논의를 살펴보고 나면 컴포넌트를 실제 DOM에 렌더링하는 것이 과연 의미 있는 행동인지 의문이 생긴다. 대부분의 경우에 대답은 '그렇지 않다'이다. 가능한 `React.addons.TestUtils.renderIntoDocument`를 사용하고. DOM에 컴포넌트를 렌더링해야 할 때만 `renderComponent`를 사용할 것을 추천한다.

## 15.9 서버 사이드 테스트

지금까지 브라우저나 클라이언트에서 사용하는 React 컴포넌트를 테스트하는 방법을 학습했다. 12장에서 살펴본 것처럼 React는 Node.js 애플리케이션이나 서버 측에서도 사용할 수 있으므로 서버 측에서 테스트하는 방법도 살펴봐야 한다. 이번에는 Mocha를 테스트 프레임워크로 사용할 생각이다. `Jasmine-node`를 사용

할 수도 있지만, mocha가 비동기 테스트를 잘 지원하다 보니 Node 생태계에서 Mocha의 선호도는 매우 높다. 따라서 여기에서는 Mocha를 사용하겠다.

**NOTE** **Jasmine과 Mocha의 비교는 불필요하다**

Jasmine과 Mocha 모두 훌륭한 테스트 프레임워크이며, 어떤 쪽을 선택해도 만족스러운 결과를 얻을 수 있다. React 코드 베이스는 Jasmine을 이용해 테스트를 한다(물론 앞에서도 다뤘지만 실제로는 Jasmine을 기반으로 해서 만든 Jest를 사용한다). 그러므로 앞에서는 Jasmine을 이용했다. 하지만 Node.js 프로젝트에서는 Mocha의 인기가 높다. 또한, Mocha로 작성한 테스트는 Chai를 이용해서 단언 문을 작성하기에도 좋다. 따라서 여기서는 Mocha 사용법을 보여주려고 한다.

설문조사 생성기에서 애플리케이션 라우팅에 사용하는 `react-router`는 클라이언트와 서버, 양쪽 모두에서 사용한다. 덕분에 동형 JavaScript 코드를 작성할 수 있다. 서버 쪽에서 라우팅 코드를 어떻게 사용하는지 살펴보자. `client/app/app_router.js`를 보면 몇 가지 사실을 알 수 있다.

1. 애플리케이션 라우터를 `require`로 가져와서 추가한다.

---

```
var app_router = require("../client/app/app_router");
```

---

2. `express`의 Router를 미들웨어로 사용한다.

---

```
var router = require('express').Router({caseSensitive: true, strict: true});
...
router.use(function (req, res, next) {
...
});
```

---

3. `react-router`를 호출하면서 URL을 전달한다.

---

```
Router.renderRoutesToString(app_router, req.originalUrl)
```

---

#### 4. 템플릿에 HTML 결과물을 렌더링하는 핸들러를 작성한다.

---

```
var template = fs.readFileSync(_dirname + "/../../client/app.html",
{encoding:'utf8'});
...
Router.renderRoutesToString(app_router, req.originalUrl)
.then( function (data) {
    var html = template.replace(/\{\{body\}\}/, data.html);
    html = html.replace(/\{\{title\}\}/, data.title);
    res.status(data.httpStatus).send(html);
}, ...);
```

---

이 과정을 테스트하기 위해서 `test/server/routing.test.js`라는 명세 파일을 만들고 다음 보일러플레이트 코드를 추가하자.

```
var request = require('supertest');
var app = require('../server/server.js');
describe("serverside routing", function(){
});
```

---

이 코드의 내용은 다음과 같다.

1. Node.js 애플리케이션인 `server.js`를 `require`로 가져온다.
2. `supertest` 모듈을 사용한다. 실제 돌아가는 서버 없이도 Node 서버로 요청을 보낼 수 있다.
3. `describe` 블록은 `Jasmine`이 아닌 `Mocha` 함수다. 뒤에서 `Mocha API`가 `Jasmine`과 어떤 점이 다른지 살펴본다.

`test/server/main.js`를 만들어서 테스트를 실행한다.

---

```
require('./routing.test.js');
```

---

커맨드 라인에서 `npm run-script test-server`를 실행하면 다음 결과를 볼 수 있다.

---

```
tom:bleeding-edge-sample-app (master) $ npm run-script testserver
> bleeding-edge-sample-app@0.0.1 test-server /Users/tom/workspace/bleeding-
edge-sample-app
> mocha test/server/main.js

0 passing (5ms)
```

---

훌륭하다. Mocha 테스트에서 오류가 발생하지 않았다. 보일러플레이트를 잘 만들었다는 뜻이다. 이제 첫 번째 테스트를 추가해서 Node 서버에 `/add_survey`에 GET 요청을 보내보자.

---

```
...
describe("serverside routing", function(){
  it("/add_survey 경로를 잘 렌더링한다", function(done){
    request(app)
      .get('/add_survey')
      .expect(200)
      .end(done);
  });
});
```

---

이 테스트는 Jasmine의 테스트와 비슷해 보인다. 그렇지만 중요한 한 가지 차이점이 있다. 바로 `done` 함수다. 익명 함수가 `done`을 인자로 받아서 호출하는 것을 확인할 수 있다. 이것은 단언 문과 테스트를 마치면 호출해야 하는 콜백 함수다. 이 테스트는 `server.js`의 `/add_survey`로 GET 요청을 보낸다. 이 요청의 결과로 돌아오는 응답 코드는 200이다. 마지막으로 `done`을 호출해서 Mocha에게 단언 문을 호출했다는 것을 알려준다.

**NOTE** SUPERTEST

request, get, expect, end 함수는 모두 Supertest 프로젝트의 일부다. 궁금한 점은 Supertest의 문서를 살펴보길 바란다.

<https://github.com/visionmedia/supertest>

이 테스트도 통과한다.

```
tom@bleeding-edge-sample-app (master) $ npm run-script testserver
> bleeding-edge-sample-app@0.0.1 test-server /Users/tom/workspace/bleeding-
edge-sample-app
> mocha test/server/main.js
```

serverside routing should render the AddSurvey component for the /add\_survey path (87ms)

1 passing (97ms)

이제 우리가 받아올 실제 내용에 관한 단언을 추가할 차례이다.

**NOTE** DONE() 호출에는 주의를 요한다.

앞의 테스트를 다음처럼 작성하지 않아야 한다. 이렇게 하면 예상 코드를 실행하기 전에 done 함수를 먼저 호출해버려 테스트가 원하는대로 동작하지 않는다.

```
it("/add_survey 경로를 잘 렌더링한다 ", function(done){
  request(app)
    .get('/add_survey')
    .expect("666");

  // DON'T DO THIS!! This test will pass every time, even though the
  assertion is not true
  // 이렇게 하면 안된다!! 이렇게 하면 단언문이 true가 아니어도 테스트는 항상 통과한다.
  done();
});
```

AddSurvey 컴포넌트를 제대로 렌더링했는지 검증하려면, 반환한 HTML을 확인하는 단언 문이 필요하다. 우선 테스트를 간단히 수정하자.

---

```
...
  it("/add_survey 경로에 AddSurvey 컴포넌트를 렌더링 해야 한다", function(done){
    request(app)
      .get('/add_survey')
      .expect(200)
      .end(function (err, res) {
        console.log("OUR HTML IS:" + res.text)
        done();
      });
  });
...

```

---

결과는 다음과 같다.

```
tom:bleeding-edge-sample-app (master) $ npm run-script testserver
> bleeding-edge-sample-app@0.0.1 test-server /Users/tom/workspace/bleeding-
edge-sample-app
> mocha test/server/main.js

serverside routing
OUR HTML IS: <!DOCTYPE html>
<html>
<head lang='en'>
...
<title>Add Survey to SurveyBuilder</title>
...
</head>
<body>
<div class="app" data-reactid=".qajmfw0740" data-
reactchecksum="2024417999">...</div>
<script src="/build/bundle.js" type="text/javascript"></script>
</body>
</html>
should render the AddSurvey component for the /add_survey path (89ms)

1 passing (100ms)
```

---

좋다. AddSurvey 컴포넌트를 정확하게 문자열로 <body>의 템플릿에 렌더링하고 있는 것을 확인했다. 이제 res.text와 기본 문자열을 비교할 수 있다. 그런데 이는 매우 골치 아프고 불안정한 방법이다. HTML 응답을 파싱하는 단언 문을 작성하자. Cheerio라는 라이브러리를 이용해서 HTML을 파싱한다. Cheerio는 읽어드린 HTML을 jQuery를 이용하는 것처럼 처리할 수 있는 인터페이스를 제공하는 Node.js 모듈이다. 다음처럼 테스트를 작성해보자.

---

```
var cheerio = require('cheerio');
...
it("add_survey 경로에 AddSurvey 컴포넌트를 렌더링 해야 한다 ", function(done){
  request(app)
    .get('/add_survey')
    .expect(200)
    .end(function (err, res) {

      var doc = cheerio.load(res.text);
      expect(doc("title").html()).to.be("Add Survey to SurveyBuilder");
      expect(doc(".main-content .surveyeditor").length).to.be(1);
      done();
    });
});
...
...
```

---

end 함수는 다음 작업을 수행한다.

1. res.text에서 HTML 응답을 받는다.
2. cheerio.load를 호출해서 응답 결과를 파싱한다.
3. <title> 엘리먼트의 innerHTML을 검증한다.
4. main-content .survey-editor 선택자가 페이지에 존재하는지 검증한다.

이 테스트는 이상 없이 통과한다. 다음 테스트는 404 오류 페이지를 정상 처리하는지 확인한다.

```
it("404 오류 페이지를 렌더링해야 한다", function(done){
    request(app)
        .get('/not-found-route')
        .expect(404)
        .end(function (err, res) {
            var doc = cheerio.load(res.text);
            expect(doc("body").html()).to.contain("The Page you were
looking for isn't here!");
            done();
        });
});
```

## 15.10 브라우저 테스트 자동화

이 장의 첫 부분에서 다양한 테스트 방법을 정의했는데, 지금까지는 주로 단위 테스트를 중점적으로 살펴봤다. 기능 테스트<sup>functional testing</sup> 역시 관심을 가져야 할 주제다. 기능 테스트는 최종 사용자 관점에서 애플리케이션이 바르게 동작하는지 검증하는 테스트다. 웹 애플리케이션의 경우 실제 사용자처럼 웹 브라우저를 이용해 마우스를 클릭하고, 양식을 작성해 보는 과정이 바로 기능 테스트다.

### NOTE 소개

이번에는 웹 애플리케이션의 기능 테스트를 작성하는 기본적인 방법을 알아본다. 여기에서 모든 내용을 완벽하게 다룰 수는 없다. 이 주제와 관련해 더 많은 정보를 얻고 싶다면 『The Cucumber Book』(2012, Pragmatic)과 『Instant Testing with CasperJS』(2014, Packt) 같은 책을 참고하길 바란다.

웹 브라우저가 특정 동작을 수행하게 한 다음에, 단언 문을 이용해 웹 페이지가 제대로 동작했는지 확인한다. 이런 테스트에는 CasperJS라는 훌륭한 도구를 이용할 수 있다. CasperJS 같은 브라우저 테스트 자동화 도구를 처음 접한다면 여기에서 소개하는 용어를 잘 살펴보길 바란다.

1. CasperJS - 웹 브라우저를 이용한 테스트를 쉽게 할 수 있게 도와주는 테스트 도구다. CasperJS는 브라우저 엔진으로 PhantomJS를 사용한다.
2. PhantomJS - Javascript API를 지원하는 헤드리스<sup>Headless</sup> 웹 브라우저다. 렌더링 엔진으로 Webkit을 사용한다.

3. 헤드리스 웹 브라우저 - 크롬, 파이어폭스, IE처럼 흔히 사용하는 브라우저에서 화면에 보이는 시각 인터페이스를 제거한 상태를 생각하면 쉽다. 명령어로 동작하며 터미널에서 실행한다.
4. 웹 브라우저 조작<sup>driving</sup> - 링크 클릭, 품 입력, URL 이동, 요소의 드래그와 드롭 같은 동작을 수행하는 것을 뜻한다. 최종 사용자가 브라우저에서 하는 모든 행동을 포함한다.
5. 웹킷<sup>Webkit</sup> - 사파리의 렌더링 엔진이다. 크롬은 웹킷을 포크<sup>fork</sup>한 블링크<sup>Blink</sup>를 사용하고, 파이어폭스는 게코<sup>Gecko</sup>, IE는 트라이던트<sup>Trident</sup>를 렌더링 엔진으로 사용한다.

본격적으로 기능 테스트를 작성하기 전에 간단한 CasperJS 테스트를 살펴보자.

```
casper.test.begin('설문 추가하기', 1, function suite(test) {  
    casper.start("http://localhost:8080/", function(){  
        test.assertTitle("SurveyBuilder", "홈페이지 title이 바르게 입력되었다");  
    });  
    casper.run(function() {  
        test.done();  
    });  
});
```

이 테스트는 고수준에서 설문조사 애플리케이션 홈페이지에 들어가 페이지의 title이 무엇인지 확인한다. 우선 React에 대한 언급이 전혀 없다는 점에 주목하자. CasperJS는 사용자처럼 브라우저를 조작하기 때문에 React로 애플리케이션을 구현했다는 사실은 별로 중요하지 않다. 테스트는 title, 정수값, 테스트 실행 함수를 인자로 받는다. 인자 중 정수값은 테스트에서 수행하는 단언 문의 개수다. 애플리케이션의 “Add a Survey”<sup>설문조사 추가하기</sup>를 테스트하는 test/functional/adding\_a\_survey.js를 작성하자.

```
// casper.js 설정  
casper.options.verbose = true;  
casper.options.logLevel = "debug";  
casper.options.viewportSize = {width: 800, height: 600};  
casper.test.begin('설문조사 추가하기', 0, function suite(test) {  
});
```

보일러플레이트를 만들었으니 어떤 테스트를 자동화할지 생각해보자.

1. 홈페이지로 이동한다.
  2. 홈페이지가 바르게 출력되었는지 확인한다.
  3. “Add Survey” 설문조사 추가하기 링크를 클릭한다.
  4. 올바른 페이지로 이동하는지 검증한다.
- 

```
casper.test.begin('설문조사 추가하기', 0, function suite(test) {  
    casper.start("http://localhost:8080/", function(){  
        console.log("홈페이지에 들어갔다!")  
    });  
});
```

---

테스트를 실행하려면 CasperJS를 설치해야 한다. 모듈을 설치하고 테스트를 실행해보자.

---

```
npm install -g casperjs  
casperjs test test/functional
```

---

다음 결과가 나타난다.

---

```
tom:bleeding-edge-sample-app (master) $ casperjs test test/functional/  
Test file: /Users/tom/workspace/bleeding-edge-sample-app/test/functional/  
adding_a_survey.js  
# 설문조사 추가하기  
[info] [phantom] Starting...  
[info] [phantom] Running suite: 2 steps  
[debug] [phantom] opening url: http://localhost:8080/, HTTP GET  
[debug] [phantom] Navigation requested: url=http://localhost:8080/,  
type=Other, willNavigate=true, isMainFrame=true  
[warning] [phantom] Loading resource failed with status=fail:  
http://localhost:8080/  
[debug] [phantom] Successfully injected Casper client-side utilities  
홈페이지에 들어갔다!
```

```
[info] [phantom] Step anonymous 2/2: done in 53ms.  
[info] [phantom] Done 2 steps in 72ms  
WARN Looks like you didn't run any test.
```

---

터미널에 보이는 로그를 자세히 들여다보자.

1. 브라우저에서 홈페이지 가져오기를 시도한다.
2. 이 요청은 실패했다: Loading resource failed with status=fail:  
`http://localhost:8080/`
3. 홈페이지를 가져오는 데 실패한 후에 `console.log`를 출력한다.
4. 아무런 테스트도 수행하지 않았다는 사실을 알려준다.

1번과 3번은 정상이다. 단연 문을 작성하지 않았으므로 4번도 문제가 없다. 2번은 문제가 있다. 테스트를 시작하기 전에 애플리케이션이 실행 중인지 확인해야 한다. 애플리케이션을 실행한다.

---

```
npm start
```

---

이제 CasperJS 테스트를 다시 실행하면 다음과 같은 결과를 볼 수 있다.

---

```
tom:bleeding-edge-sample-app (master) $ casperjs test test/functional/  
Test file: /Users/tom/workspace/bleeding-edge-sample-app/test/functional/  
adding_a_survey.js  
# 설문조사 추가하기  
[info] [phantom] Starting...  
[info] [phantom] Running suite: 2 steps  
[debug] [phantom] opening url: http://localhost:8080/, HTTP GET  
[debug] [phantom] Navigation requested: url=http://localhost:8080/,  
type=Other, willNavigate=true, isMainFrame=true  
[debug] [phantom] url changed to "http://localhost:8080/"  
[debug] [phantom] Successfully injected Casper client-side utilities  
[info] [phantom] Step anonymous 2/2 http://localhost:8080/  
(HTTP 200)  
홈페이지에 들어갔다!
```

```
[info] [phantom] Step anonymous 2/2: done in 1773ms.  
[info] [phantom] Done 2 steps in 1792ms  
WARN Looks like you didn't run any test.
```

---

훨씬 좋아졌다. 홈페이지의 <title>이 SurveyBuilder가 맞는지 확인하는 단언문을 추가하자.

---

```
...  
casper.start("http://localhost:8080/", function(){  
    // 홈페이지 title이 SurveyBuilder인지 확인하는 단언문을 작성한다  
    test.assertTitle("SurveyBuilder", "홈페이지 title이 정확하다");  
});  
...  
...
```

---

title을 제대로 입력했는지 확인하는 코드를 추가했다. 두 번째 인자로 테스트에 대한 설명을 작성해서 전달한다. 터미널에서 CasperJS가 PASS를 보여줄 때 이 내용을 함께 보여준다.

---

```
tom:bleeding-edge-sample-app (master) $ casperjs test test/functional/  
Test file: /Users/tom/workspace/bleeding-edge-sample-app/test/functional/  
adding_a_survey.js  
# 설문조사 추가하기  
[info] [phantom] Starting...  
[info] [phantom] Running suite: 2 steps  
[debug] [phantom] opening url: http://localhost:8080/, HTTP GET  
[debug] [phantom] Navigation requested: url=http://localhost:8080/,  
type=Other, willNavigate=true, isMainFrame=true  
[debug] [phantom] url changed to "http://localhost:8080/"  
[debug] [phantom] Successfully injected Casper client-side utilities  
[info] [phantom] Step anonymous 2/2 http://localhost:8080/  
(HTTP 200)  
PASS 홈페이지 title이 정확하다  
[info] [phantom] Step anonymous 2/2: done in 1864ms.  
[info] [phantom] Done 2 steps in 1883ms  
PASS 1 test executed in 1.89s, 1 passed, 0 failed, 0 dubious, 0  
skipped.
```

---

**NOTE** CasperJS 문서

CasperJS로 할 수 있는 것들에 대해 더 관심이 있다면 [casperjs.org](http://casperjs.org)를 방문하길 바란다

처음으로 테스트에 통과했다. 이제 링크를 클릭하게 만들 차례다.

---

```
...
casper.start("http://localhost:8080/", function(){
    // 홈페이지 title이 SurveyBuilder인지 확인하는 단언문을 작성한다
    test.assertTitle("SurveyBuilder", "the title for the homepage is
correct");
    // navbar의 두번째에 있는 Add Survey 링크를 클릭한다
    this.click(".navbar-nav li:nth-of-type(2) a");
});
...
...
```

click 함수는 CSS 선택자를 받는다. 앞의 예제 테스트에서는 navbar의 두 번째 링크를 클릭한다. 링크가 가지고 있는 고유 CSS 클래스나 ID 선택자를 전달할 수 있다. 설문조사 추가 링크를 클릭한 후 설문조사 추가 페이지로 이동하는지 확인하자.

---

```
...
casper.start("http://localhost:8080/", function(){
    // 홈페이지 title이 SurveyBuilder인지 확인하는 단언문을 작성한다
    test.assertTitle("SurveyBuilder", "홈페이지 title이 정확하다");
    // navbar의 두번째에 있는 Add Survey 링크를 클릭한다
    this.click(".navbar-nav li:nth-of-type(2) a");
});

casper.then(function(){
    // /add_survey 페이지가 나오는지 검증한다
    test.assertTitle("설문조사 생성기에 설문조사 추가하기", "설문조사 페이지의 title이
정확하다");
    test.assertTextExists("모듈을 왼쪽에서 드래그 앤 드롭 한다",
        "설문조사 화면에 드래그 앤 드롭에 대한 설명이 있다");
});
...
...
```

이 코드를 살펴보면 재미있는 사실을 발견할 수 있다. 가장 마지막으로 작성한 단언 문이 `start`로 작성한 이전 코드와 달리 `then` 콜백으로 되어 있다. 왜 그럴까? CasperJS는 `click` 메소드를 호출하면 새로운 페이지 불러오고 클릭을 할 때까지 대기한다. 이 때문에 `click`은 비동기로 동작하고, 따라서 다음 동작은 `then`으로 실행해야 한다. 설문조사 링크를 클릭하는 동작을 마치고 나면 `then`을 실행한다.

`casperjs test test/functional/ 명령`을 입력해서 테스트를 다시 실행해 보면 통과하는 것을 볼 수 있다.

## 서버 구동하기

앞에서는 CasperJS 테스트를 하기 위해 `npm` 서버를 실행해야 했다. 이는 번거로운 일인 데다가 CI 서버를 다룰 때 골치 아픈 일이기도 하다. 테스트를 실행하기 전에 서버를 실행하는 `bash` 스크립트를 작성하자. 프로젝트의 최상위 디렉터리에 `run_casperjs.js`라는 스크립트를 작성한다.

---

```
// Node 웹서버를 3040 포트에 가동한다
var app = require("./server/server"),
    appServer = app.listen(3040),

    // CasperJS 테스트 도구를 child process에서 실행한다
    spawn = require('child_process').spawn,
    casperJs = spawn('./node_modules/casperjs/bin/casperjs', ['test', 'test/functional']);

// CasperJS에서 모든 데이터를 받아 화면으로 전달한다
casperJs.stdout.on('data', function (data) {
    console.log(String(data));
});

casperJs.stderr.on('data', function (data) {
    console.log(String(data));
});

// CasperJS 테스트가 끝나면, Node 웹서버를 종료한다.
casperJs.on('exit', function(){
```

```
    appServer.close();  
});
```

---

이 파일은 다음 작업을 실행한다.

1. Node.js 서버를 불러온다.
2. 3040 포트로 서버를 실행한다.
3. Node.js의 `spawn`을 이용해서 하위 프로세스로 CasperJS 테스트를 실행한다.  
터미널에서 `casperjs test test/functional`을 실행하는 것과 같다.
4. CasperJS의 모든 결과 값을 터미널에 전달한다.
5. CasperJS 테스트가 끝나면 웹 서버를 종료한다.

이제 테스트 명세에 8080 포트를 3040 포트로 수정하면 끝이다. `run_casperjs.js`를 실행해서 CasperJS 테스트와 서버 실행을 모두 자동으로 할 수 있다.

지금까지 설명한 내용을 잘 살펴봤다면 CasperJS 테스트가 어떤 형태인지 이해하고 기본적인 테스트를 작성할 수 있게 되었을 것이다. 여기에서 CasperJS를 깊게 설명하지는 않았는데 더 공부하고 싶다면 다음 자료를 살펴보길 권한다.

- Joseph Scott의 CasperJS를 이용한 사이트 테스트: <https://www.youtube.com/watch?v=fIhjYUNCo-U>
- CasperJS 테스트 프레임워크: <http://docs.casperjs.org/en/latest/testing.html>
- CasperJS casper 문서: <http://docs.casperjs.org/en/latest/modules/casper.html>

## 15.11 정리

이번 장에서는 렌더링부터 시작해서 스파이, 모의객체, 이벤트, 믹스인, 컴포넌트 탐색, 서버 측 테스트, 기능 테스트까지 다양한 테스트의 개념을 살펴보았다. 이 장의 내용을 잘 이해했다면 실제 애플리케이션에 사용하고 있는 React 컴포넌트

를 테스트하는 데 어려움이 없을 것이다.

지금까지 React 애플리케이션을 어떻게 단위 테스트하고 기능 테스트를 하는지 살펴봤다. 다음 장에서는 React 애플리케이션에 적용할 수 있는 설계 패턴을 학습한다.

# 설계 패턴

React는 HTML을 렌더링한다. MVC에서 V의 역할을 하지만 다음처럼 간단한 AJAX 요청을 전송할 때 어떤 간섭도 하지 않는다.

```
var TakeSurvey = React.createClass({  
  getInitialData: function () {  
    return {  
      survey: null  
    };  
  },  
  componentDidMount: function () {  
    $.getJSON('/survey/' + this.props.id, function (json) {  
      this.setState({survey: json});  
    });  
  },  
  render: function () {  
    if (!this.state.survey) return null;  
    return <div>{this.state.survey.title}</div>;  
  }  
});
```

MVC 프레임워크를 사용하고 있다면 대부분은 애플리케이션 구조에 React를 통합할 수 있다.

**NOTE**

예제로 사용하는 설문조사 생성기의 전체 소스 코드는 Github 저장소에서 확인할 수 있다.

<https://github.com/backstopmedia/bleeding-edge-sample-app>

이번 장에서는 React 애플리케이션을 설계할 때 활용할 수 있는 몇 가지 접근 방법과 라이브러리를 살펴본다.

## 16.1 라우팅

라우터는 단일 페이지 애플리케이션에서 URL과 특정 핸들러를 연결한다. 사용자가 /surveys URL로 접근하면 서버에서 사용자 정보를 불러오고 <ListSurveys /> 컴포넌트를 렌더링하는 함수를 실행하는 경우를 생각해볼 수 있다.

라우터의 종류는 다양하다. 라우터는 주로 서버 쪽에 존재하며 때에 따라서 클라이언트와 서버 양쪽에 존재하기도 한다.

렌더링 라이브러리인 React는 라우터를 가지고 있지 않다. 대신 React와 함께 사용할 수 있는 도구가 많이 있다. 이번 절에서 이런 도구들을 살펴본다.

### Backbone.Router

Backbone은 단일 페이지 애플리케이션 라이브러리로 MV\*Model-View-Whatever 패턴으로 설계되어 있다. MV\*에서 \*은 컨트롤러Controller 역할을 하며, 흔히 라우터Router인 경우가 많다. Backbone도 그렇다.

Backbone의 컴포넌트는 모듈화되어 있어 오로지 Router만 별도로 사용할 수 있다. 그래서 React와 잘 어울린다. 앞에서 살펴본 /surveys 예제에 Backbone Router를 적용해보자.

---

```
var SurveysRouter = Backbone.Router.extend({
  routes: {
    "surveys": "list"
  },
  list: function() {
    React.renderComponent(
```

```
        <ListSurveys />,
        document.querySelector('body')
    );
}
});
```

---

라우터는 URL이나 queryString의 동적으로 바뀌는 부분을 처리할 수 있어야 한다. Backbone.Router는 이를 다음과 같은 방식으로 지원한다.

```
// surveys_router.js
var SurveysRouter = Backbone.Router.extend({
    routes: {
        'surveys': 'list',
        'surveys/:filter': 'list'
    },
    list: function(filter) {
        React.renderComponent(
            <ListSurveys filter={filter}/>,
            document.querySelector('body')
        );
    }
});
```

---

앞의 예제는 /surveys/active 같은 URL을 받으면 SurveysRouter#list의 인자인 filter를 활성화한다.

Backbone.Router에 대한 자세한 정보를 원한다면 <http://backbonejs.org/#Router>를 참고하길 바란다.

## Aviator

Aviator는 Backbone.Router와 달리 독립형 라우터 라이브러리이다. Aviator를 사용하면 Route와 RouteTarget을 분리하여 정의할 수 있다. Aviator는 RouteTarget의 형태나 동작에 대해서 고민하지 않고 단순히 할당된 함수를 호

출한다. RouteTarget은 이런 모습이다.

---

```
// surveys_route_target.js
var SurveysRouteTarget = {
    list: function () {</div><div>&nbsp; &nbsp; React.renderComponent(</div><div><ListSurveys/>,</div><div>document.querySelector('body')</div><div>);</div><div>}</div><div>};</div>
```

---

코드에 보이는 것처럼 객체를 정의하고, 여기에 더해 라우트를 설정해야 한다. 애플리케이션 전체에 하나의 라우트만 설정할 수 있다. 대체로 설정 파일은 분리하여 작성한다.

---

```
// routes.js
Aviator.setRoutes({
    '/surveys': {
        target: UsersRouteTarget,
        '/': 'list'
    }
});

// Aviator에게 대상지점으로 URL을 보낼 것을 요청한다.
Aviator.dispatch();
```

---

매개 변수를 다루기 위해서는 RouteTargets를 다음과 같이 설정한다.

---

```
// routes.js
Aviator.setRoutes({
    '/surveys': {
        target: UsersRouteTarget,
        '/': 'list',
        '/:filter': 'list'
    }
});

// surveys_route_target.js
var SurveysRouteTarget = {
```

```
list: function (request) {
  React.renderComponent(
    <ListSurveys filter={request.params.filter}>/>,
    document.querySelector('body')
  );
}
};
```

---

Aviator는 여러 개의 target을 가질 수 있는 장점이 있다. 다음 예제에서 라우트를 정의한 방법을 살펴보자.

---

```
// routes.js
Aviator.setRoutes({
  target: AppRouteTarget,
  '/*': 'beforeAll',
  '/surveys': {
    target: UsersRouteTarget,
    '/': 'list',
    '/:filter': 'list'
  }
});
```

---

'/surveys/active'로 요청이 들어오면 Aviator는 AppRouteTarget.beforeAll을 호출한 다음 UsersRouteTarget.list를 호출한다. 이런 식으로 필요한 동작을 몇 개라도 연결할 수 있다. 사용자가 현재 경로에서 밖으로 나갈 때, 역순으로 호출할 종료 함수를 라우트로 등록할 수도 있다.

Aviator에 대한 더욱 자세한 정보는 <https://github.com/swipely/aviator>에 잘 나와 있다.

## react-router

React-router는 앞에서 살펴본 도구들과 달리 오직 React 컴포넌트 전용으로 만들어졌다.

컴포넌트로 라우트를 정의했으며, 라우트의 핸들러 역시 컴포넌트다. 다음은 React-router로 작성한 라우터 예제다.

---

```
var appRouter = (
  <Routes location="history">
    <Route title="SurveyBuilder" handler={App}>
      <Route name="list" path="/" handler={ListSurveys} />
      <Route title="Add Survey to SurveyBuilder" name="add" path="/add_survey" handler={AddSurvey} />
      <Route name="edit" path="/surveys/:surveyId/edit" handler={EditSurvey} />
      <Route name="take" path="/surveys/:surveyId" handler={TakeSurveyCtrl} />
      <NotFound title="Page Not Found" handler={NotFoundHandler}/>
    </Route>
  </Routes>
);
```

---

각각의 핸들러는 특정 페이지를 위한 컴포넌트이다. 라우터는 최상위 레벨 컴포넌트처럼 웬더링해서 실행한다.

---

```
React.renderComponent(
  appRouter,
  document.querySelector('body')
);
```

---

다른 라우터와 마찬가지로 react-router도 비슷한 매개 변수 형태를 가지고 있다. 예를 들어 /surveys/:surveyId 라우트는 surveyId를 이용해 값을 TakeSurveyCtrl 컴포넌트에 전달한다.

React-router가 Link 컴포넌트를 지원한다는 점은 장점이다. Link 컴포넌트를 내비게이션에 적용하면 자동으로 라우트에 연결한다. 또한, 링크에 활성화 클래스를 추가해서 현재 페이지를 자동으로 나타내기도 한다.

다음은 React-router의 Link 컴포넌트를 사용한 <MainNav/> 컴포넌트의 예제다.

---

```
var MainNav = React.createClass({
  render: function () {
    return (
      <nav className='main-nav' role='navigation'>
        <ul className='nav navbar-nav'>
          <li><Link to="list">All Surveys</Link></li>
          <li><Link to="add">Add Survey</Link></li>
        </ul>
      </nav>
    );
  }
});
```

---

React-router에 대해서 더 살펴보고 싶다면 <https://github.com/rackt/reactrouter>를 참고하자.

## 16.2 Om(ClojureScript)

Om은 인기 있는 ClojureScript의 인터페이스를 React에 적용한 라이브러리다. Om은 ClojureScript의 영속적인 데이터 구조를 이용해서 애플리케이션의 근본적인 부분에서 매우 빠르게 다시 렌더링한다. 각각의 렌더링은 스냅샷으로 저장되므로 실행 취소와 같은 작업에 이용할 수 있다. 다음은 Om 컴포넌트를 이용한 예제다.

---

```
(ns example
  (:require [om.core :as om :include-macros true]
            [om.dom :as dom :include-macros true]))
(defn App [data owner]
  (reify
    om/IRender
    (render [this]
      (dom/h1 nil (:text data)))))
```

```
(om/root App {:text "Survey Builder"}  
{:target (. js/document (querySelector "body"))})
```

---

이 예제는 <h1>Survey Builder</h1>를 렌더링한다.

## 16.3 Flux

앞에서 살펴본 설계 패턴은 Facebook이 React를 오픈 소스로 공개한 이후에 등장했다. 이와 달리 Flux는 React 개발자가 처음부터 의도한 설계 패턴이다.

Flux는 Facebook이 소개한 설계 패턴이다. Flux는 React의 데이터 흐름을 단방향<sup>unidirectional data-flow</sup>으로 만든다. 단방향 데이터 흐름은 이해하기가 쉽고 아주 간단한 골격<sup>scaffolding</sup>만으로도 실행할 수 있다.

Flux는 크게 Store, Dispatcher, View의 세 부분으로 나뉘어 있으며, 이 셋은 모두 React 컴포넌트다. 여기에 더해 Action이 있는데, Dispatcher에 시맨틱한 인터페이스를 만드는 도우미 메소드<sup>helper method</sup>다.

최상위 React 컴포넌트를 컨트롤러-뷰<sup>Controller-View</sup>로 볼 수 있다. 컨트롤러-뷰 컴포넌트는 Store와 인터페이스로 연결되어 있으면서 자식 뷰 컴포넌트와의 커뮤니케이션을 쉽게 만든다. iOS의 뷰-컨트롤러<sup>View-Controller</sup>와 비슷하다.

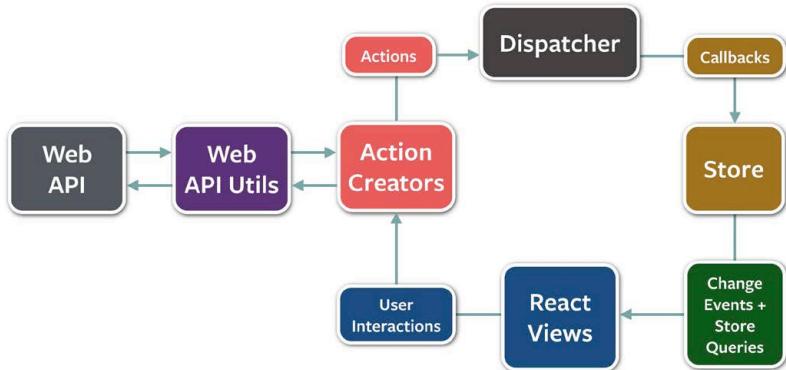
Flux의 각 노드는 독립적이며 관심사를 분리해 격리하고 있어 테스트하기 쉽다.

### 데이터 흐름

정보 흐름이 단일 방향이라는 점은 Flux가 가진 독특한 특징이다. 최신 MVC 프레임워크가 가진 철학에 비춰보면 이런 특징이 다소 이상하게 느껴질 수도 있겠지만, 여기에는 분명한 이점이 있다. Flux는 양방향 데이터 바인딩을 이용하지 않기 때문에 애플리케이션을 이해하기 쉽다. 데이터를 담고 있는 Store에서 상태

`State`를 관리하기 때문에 더욱 쉽게 추적할 수 있다. `Store`는 데이터를 변경 메소드 `change method`를 이용해서 전달한다. 데이터를 전달받으면 뷰는 렌더링을 수행한다. 사용자 입력 이벤트가 발생하면 `dispatcher`는 요청을 전달하는 작업을 수행하는데, 이때 특정 동작을 담당하는 `Store`에 페이로드를 전달한다.

[그림 16-1]



이미지 출처: 페이스북

## Flux 구성 요소

Flux는 개별적인 책임을 지는 네 개의 요소를 가지고 있다. 각 구성 요소는 다음 스트림에서 데이터 입력을 전달받아 처리한 후 업스트림으로 결괏값을 전달하는 일련의 과정을 단방향으로 처리한다.

- `Dispatcher` — 애플리케이션의 중앙 허브
- `Actions` — 애플리케이션의 도메인 특정 언어<sup>DSL, Domain Specific Language</sup>
- `Store` — 비즈니스 로직과 데이터 인터랙션
- `Views` — 애플리케이션 렌더링을 위한 컴포넌트 트리

여기서는 각 구성 요소가 담당하는 역할을 살펴보고 React 애플리케이션에 효율적으로 사용하는 방법을 논의한다.

## Dispatcher

Dispatcher를 가장 먼저 소개하는 이유는 모든 사용자 인터랙션과 데이터 흐름에 관여하는 중앙 허브이기 때문이다. Dispatcher는 Flux 패턴 안에 있는 싱글톤 singleton이다.

Dispatcher는 Store에 콜백 함수를 등록하고, Store와 콜백 함수 사이의 의존성을 관리한다. 사용자의 행위로 발생한 이벤트 정보는 Dispatcher로 흘러간다. 데이터 페이로드는 해당 페이로드에 등록된 Store에 전달된다.

설문조사 생성기는 간단하지만, 효과적인 Dispatcher를 가지고 있다. 이 Dispatcher는 한 개의 Store를 관리한다. 애플리케이션을 확장해감에 따라 여러 개의 Store가 생겨 각각의 의존성을 관리해야 하는 경우가 반드시 생긴다. 이 부분은 뒤에서 좀 더 자세히 설명하겠다.

## Actions

사용자 관점에서 볼 때 Actions은 Flux의 시작점이다. 사용자가 UI를 조작함으로써 만들어지는 모든 Action은 Dispatcher로 보내진다.

Actions은 Flux 패턴을 구성하는 공식 요소는 아니지만 애플리케이션의 도메인 특정 언어로 여겨진다. 사용자 동작은 의미 있는 Dispatcher 동작으로 변환되어 Store를 작동시킨다.

설문조사 응답 애플리케이션은 3개의 서로 다른, 사용자의 동작을 처리한다.

1. 새로운 설문조사 저장
2. 기존 설문조사 삭제
3. 설문조사 결과 기록

SurveyActions 객체에서 이 경우를 찾아볼 수 있다.

---

```
var SurveyActions = {
  save: function(survey) {...},
  delete: function(id) {...},
  record: function(results) {...}
};
```

---

예를 들어, 설문조사에 응답하는 사용자가 있다고 생각해보자. 사용자가 필수 입력 사항을 모두 입력하고 나서 Save 버튼을 클릭했다. 이때 React 컴포넌트는 사용자의 클릭 요청을 onSave 메소드 호출로 전환한다.

---

```
var TakeSurvey = React.createClass({
  ...
  handleClick: function() {
    this.props.onSave(this.state.results);
  },
  render:function(){
    return (
      <div className="survey">
        ...
        <button ... onClick={this.handleClick}>Save</button>
      </div>
    );
  }
});
```

---

TakeSurveyCtrl 컨트롤러-뷰가 onSave를 처리하면서 사용자가 Save 버튼을 클릭하는 동작을 Store가 record action으로 변환한다.

---

```
var TakeSurveyCtrl = React.createClass({
  ...
  handleSurveySave: function(results) {
    SurveyActions.record(results);
  },
  render:function () {
    var props = merge({}, this.state.survey, {
      onSave: this.handleSurveySave
    });
  }
});
```

---

```
    });
    return TakeSurvey(props);
}
});
```

---

사용자 동작은 이런 방식으로 컴포넌트를 통해 Dispatcher까지 흘러간다. 이것 이 사용자 동작을 Flux 애플리케이션의 도메인 특정 언어로 변환하는 과정이다.

## Store

Store는 비즈니스 로직을 캡슐화하고 애플리케이션의 데이터와 관련한 인터랙션을 처리한다. Store는 Dispatcher에 등록한 Action 중에서 응답해야 할 Action을 선택한다. Store는 change 이벤트를 이용해서 React 컴포넌트 계층으로 데이터를 전달한다.

Store의 관심사는 엄격하게 분리해야 한다.

- Store는 애플리케이션의 모든 데이터를 가지고 있다.
- 애플리케이션의 다른 구성 요소를 데이터 인터랙션을 처리하지 않는다. Store는 애플리케이션에서 데이터를 조작하는 유일한 곳이다.
- Store는 setter가 없다. 모든 변경 사항은 Dispatcher를 거쳐 Store로 들어간다. 새로운 데이터는 Store의 change 이벤트를 타고 애플리케이션으로 되돌아간다.

사용자가 설문조사 결과를 기록하는 앞의 예제에서 Dispatcher는 record action을 Store에 전달한다.

---

```
Dispatcher.register(function(payload) {
  switch(payload.actionType) {
    ...
    case SurveyConstants.RECORD_SURVEY:
      SurveyStore.recordSurvey(payload.results);
      break;
  }
});
```

---

Store는 동작을 전달받고 결과를 저장한다. 저장이 끝나면 change 이벤트를 일으킨다.

---

```
SurveyStore.prototype.recordSurvey = function(results) {  
    // 여기에서 결과 저장과 관련한 작업을 수행한다  
    this.emitChange();  
}
```

---

App.js에 정의한 주 컨트롤러-뷰가 이 change 이벤트를 핸들링한다. React 컴포넌트 계층이 새로운 state를 전달받으면 React는 컴포넌트를 필요에 따라 다시 렌더링한다.

### 컨트롤러-뷰

보통 애플리케이션의 컴포넌트 계층에는 Store에 대한 인터랙션을 담당하는 최상위 컴포넌트가 있다. 간단한 애플리케이션에는 단 하나만 있을 것이고, 복잡한 애플리케이션이라면 여러 개의 컨트롤러-뷰가 존재할 수 있다.

예제 애플리케이션은 app.js에 컨트롤러-뷰를 가지고 있다. Store와 연결하는 과정은 간단하다.

1. 컴포넌트를 탑재<sup>mount</sup>하면 change 리스너를 추가한다.
2. Store에 변경이 발생하면 새로운 데이터를 요청하고 결과에 따라 처리한다.
3. 컴포넌트 탑재가 끝나면<sup>unmount</sup> change 리스너를 제거한다.

다음은 app.js에서 Store와 관련한 인터랙션을 처리하는 코드다.

---

```
var App = React.createClass({  
    handleChange: function() {  
        SurveyStore.listSurveys(function(surveys) {  
            // 설문조사 데이터 핸들링  
        });  
    },
```

```
componentDidMount: function() {
    SurveyStore.addChangeListener(this.handleChange);
},
componentWillUnmount: function() {
    SurveyStore.removeChangeListener(this.handleChange);
},
...
});
```

---

## 여러 개의 Store 관리하기

예제로 살펴본 설문조사 생성기는 하나의 Store만 사용하고 있지만, 애플리케이션은 시간이 지남에 따라 반드시 규모가 커지기 마련이므로 여러 개의 Store가 필요하다. 하나의 Store가 다른 Store를 의존하는 경우에는 한 Store가 동작을 수행하기 위해 의존하는 Store가 먼저 동작을 마쳐야 하므로 처리하기 까다롭다.

예를 들어 설문조사의 결과를 요약한 내용을 관리하기 위해서 별도의 Store를 만들어서 모든 설문조사 응답 결과를 검수한다고 했을 때, 주 Store가 record action 처리를 마쳐야만 요약 Store를 안전하게 변경할 수 있다.

따라서 몇 가지를 변경해야 한다.

- Dispatcher에 action 대기열 queue을 만들어야 한다.
- 동작이 끝날 때까지 Dispatcher를 대기시켜야 한다.
- Dispatcher에 등록된 콜백은 어떤 동작을 대기시킬 것인지 정의해야 한다.

Store는 이런 기능을 수행하지 않는다. Dispatcher가 action 대기열을 만들고 Dispatcher에 등록한 콜백 메소드가 Store에 대한 호출 흐름을 제어한다. 따라서 Dispatcher에 등록한 함수가 이런 역할을 맡아야 한다.

Dispatcher를 완벽하게 리팩토링하는 것은 주제를 벗어나므로, 여기에서는 여러 개의 Store를 다룰 때 주의할 점에 대해서만 설명했다. 더욱 자세한 해결책은 Facebook의 Flux 웹사이트([github.com/facebook/flux](https://github.com/facebook/flux))를 참고하기 바란다.

## Dispatcher 간단하기

앞에서 살펴본 Dispatcher는 단순히 새로운 콜백을 배열에 밀어 넣기만 한다. 하지만 순서를 강제하려면 각각의 콜백을 추적할 수 있어야 한다. 따라서 Dispatcher를 리팩토링해서 등록한 각각의 콜백에 id를 부여한다. 이 id는 다른 Store의 콜백을 잠시 대기시켜야 한다는 것을 Dispatcher에 알려야 할 때 사용할 수 있는 토큰이다.

---

```
Dispatcher.prototype.register = function(callback) {
    var id = uniqueId('ID-');
    this.handlers[id] = {
        isPending: false,
        isHandled: false,
        callback: callback
    };
    return id;
};
```

---

다음은 `waitFor` 메소드를 추가한다. `waitFor` 메소드를 이용하면 Store에 진행 전에 특정 콜백을 실행할 것을 요청할 수 있다. 다음 `waitFor` 메소드의 예제가 있다. `register` 함수가 반환한 id 목록인 `ids`를 전달해서 진행 전에 실행한다.

---

```
Dispatcher.prototype.waitFor = function(ids) {
    for (var i = 0; i < ids.length; i++) {
        var id = ids[i];
        if(!this.isPending[id] && !this.isHandled[id]) {
            this.invokeCallback(id);
        }
    }
};
```

---

## 의존성 있는 Action 등록하기

RECORD\_SURVEY action과 연관된 두 개의 Store가 있다.

- SurveyStore는 설문조사 결과를 기록해야 한다.
- SurveySummaryStore는 모든 응답 결과를 검수해야 한다.

따라서 SurveyStore가 RECORD\_SURVEY action 처리를 끝내야만 SurveySummaryStore가 작업을 종료할 수 있다. 애플리케이션의 최상위에 SurveyStore를 등록할 때, Dispatcher 토큰에 대한 참조를 저장한다.

---

```
// SurveyStore와 동작 Dispatcher의 연결
SurveyStore.dispatchToken = Dispatcher.register(function(payload) {
    switch(payload.actionType) {
        ...
        case SurveyConstants.RECORD_SURVEY:
            SurveyStore.recordSurvey(payload.results);
            break;
        ...
    }
});
```

---

그다음에 SurveySummaryStore를 등록하고 SurveyStore 데이터에 접근하기 전에 waitFor를 호출한다.

---

```
SurveySummaryStore.dispatchToken = Dispatcher.register(function(payload) {
    switch(payload.actionType) {
        case SurveyConstants.RECORD_SURVEY:
            Dispatcher.waitFor(SurveyStore.dispatchToken);
            // 이 지점에서는 SurveyStore 콜백이 이미 실행되었기 때문에
            // 안전하게 데이터에 접근하여 SurveySummaryStore의 summarize를 실행할
            수 있다.
            SurveySummaryStore.summarize(SurveyStore.listSurveys());
            break;
    }
});
```

---

## 16.4 정리

이번 장에서는 다양한 현대적 설계 패턴을 React에 적용하는 방법을 살펴봤다. 전통적인 MVC를 사용한 기존의 프로젝트에 React를 통합하는 것부터 시작해서 Flux와 같은 새로운 패턴에도 적용해보면서 어디에 가져다 놔도 꽤 잘 어울린다는 점을 확인했다.

설계 패턴 외에 React에서 사용할 수 있는 다른 라이브러리나 도구도 있다. 다음 장에서는 React를 이용해서 데스크톱 애플리케이션, 게임, 이메일, 차트를 만드는 법을 알아본다.



# 그밖의 사용법

React는 강력한 대화형 UI를 만들 수 UI 렌더링 라이브러리다. 또한, React를 이용하면 데이터와 사용자 입력을 멋지게 처리할 수 있다. 재사용하기 좋고 단위 테스트하기 쉬운 작은 컴포넌트를 만들 수 있다. React의 이런 장점은 웹이 아닌 다른 기술에도 적용할 수 있다.

이번 장에서는 React를 적용할 수 있는 다른 기술을 알아보자.

- 데스크톱 애플리케이션
- 게임
- 이메일
- 차트

## 17.1 데스크톱

atom-shell이나 node-webkit을 이용하면 웹 애플리케이션을 데스크톱에서도 실행할 수 있다. Github이 만든 Atom Editor는 atom-shell과 React를 이용해서 만들어졌다. 설문조사 생성기 앱을 atom-shell을 이용해 작동시켜 보자.

### NOTE

예제로 사용하는 설문조사 생성기의 전체 소스 코드는 Github 저장소에서 확인할 수 있다.

<https://github.com/backstopmedia/bleeding-edge-sample-app>

우선 <https://github.com/atom/atom-shell>에서 atom-shell을 내려 받아 설치한다. 다음의 desktop.js 스크립트를 이용해 Atom-shell을 실행하면 애플리케이션을 브라우저 창에서 확인할 수 있다.

---

```
// desktop.js
var app = require('app');
var BrowserWindow = require('browser-window');

// SurveyBuilder 서버를 require하고 실행한다.
var server = require('./server/server');
server.listen('8080');

// 서버에 충돌을 보고한다.
require('crash-reporter').start();

// 브라우저 윈도우 객체를 참조하는 전역 변수를 선언해서 참조를 계속 유지한다.
// 이렇게 하지 않으면 윈도우가 가비지 콜렉팅의 대상이 되어 자동으로 닫혀버린다.
var mainWindow = null;

// 모든 윈도우를 닫으면 앱을 종료한다.
app.on('window-all-closed', function() {
    if (process.platform != 'darwin')
        app.quit();
});

// atom-shell이 초기화를 마치고 브라우저 윈도우를 생성할 준비가 끝나면 이 메소드를 호출한다.
app.on('ready', function() {
    // Create the browser window.
    // 브라우저 윈도우를 생성한다.
    mainWindow = new BrowserWindow({width: 800, height: 600});

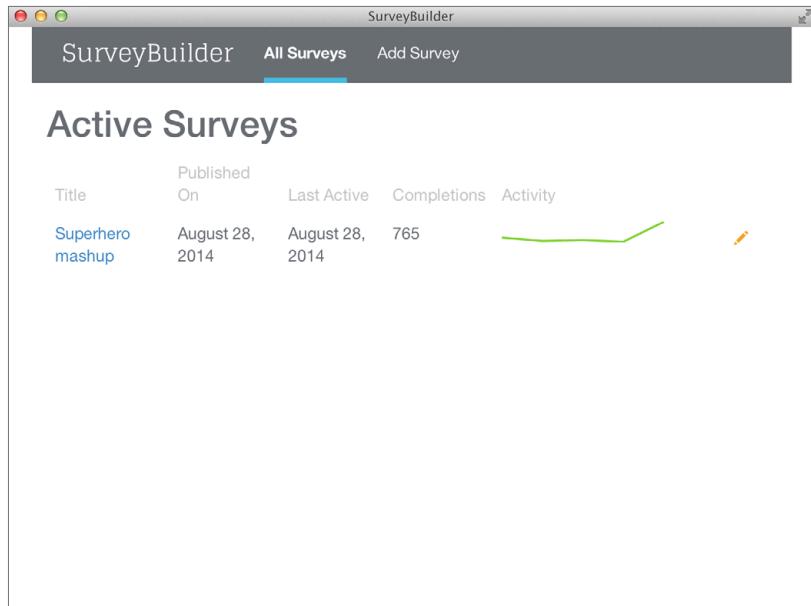
    // 앱의 index.html을 로드한다.
    // mainWindow.loadUrl('file://' + __dirname + '/index.html');
    mainWindow.loadUrl('http://localhost:8080/');

    // 윈도우를 닫을 때 처리
    mainWindow.on('closed', function() {
        // 윈도우 객체 참조를 제거한다.
    });
});
```

```
// 앱이 다중 윈도우를 지원할 경우 윈도우는 주로 배열에 저장하는데,  
// 이 시점에 해당 요소를 제거한다.  
        mainWindow = null;  
    };  
});  
});
```

---

[그림 17-1]



atom-shell이나 node-webkit을 이용하면 웹 기술로 데스크톱 애플리케이션을 개발할 수 있다. 웹과 마찬가지로 React를 이용해서 데스크톱에서 동작하는 강력한 대화형 애플리케이션을 만들 수 있다.

## 17.2 게임

게임은 보통 높은 수준의 사용자 상호작용이 필요하며 사용자는 게임의 상태가 변화에 따라 응답해야 한다. 이 점은 대부분의 웹 애플리케이션에서 사용자가 콘텐

츠를 소비나 생산하는 것과는 차이가 있다. 게임은 본질적으로 상태 시스템이며 두 가지 기본 규칙을 갖는다.

### 1. 화면 변경

### 2. 이벤트 응답

이 책의 도입부에서 React는 아주 작은 범위의 두 가지 관심사만 처리한다고 설명했다.

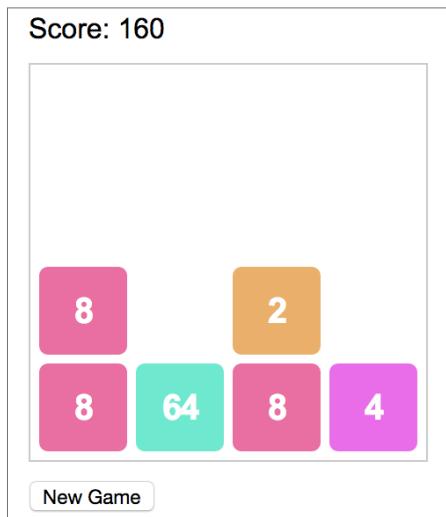
### 1. 화면 갱신

### 2. 이벤트 응답

React와 게임의 유사성은 이뿐만이 아니다. React의 가상 DOM 구조는 고성능 3D 게임 엔진을 많이 닮았다. 필요한 화면 상태를 받는 점이나 화면과 DOM을 효율적으로 갱신하는 점 등이 비슷하다.

React로 구현한 2048 게임을 보자. 같은 숫자를 합쳐서 2048에 도달하는 것이 이 게임의 목표다.

[그림 17-2]



jsfiddle을 이용해서 실제 구현 내용을 볼 수 있다(<http://jsfiddle.net/karlmikko/cdnh399c/>). 소스는 크게 두 부분이다. 하나는 전역 네임스페이스에 함수로 구현한 게임 로직이고, 다른 하나는 React 컴포넌트다. 게임 보드의 초기 데이터 구조가 가장 먼저 눈에 들어온다.

---

```
var initial_board = {  
    a1:null, a2:null, a3:null, a4:null,  
    b1:null, b2:null, b3:null, b4:null,  
    c1:null, c2:null, c3:null, c4:null,  
    d1:null, d2:null, d3:null, d4:null  
};
```

---

게임 보드의 기본 데이터 구조는 객체이고, 이 객체는 CSS에 정의한 그리드를 키로 갖는다. 기본 데이터 구조 다음에는 이 데이터를 처리하는 함수를 살펴보자. 모든 함수는 입력 값을 불변<sup>immutable</sup> 방식으로 처리하며, 입력 값을 직접 변경하지 않고 새로운 게임 보드를 만들어서 반환한다. 이런 방식으로 게임 로직에 따라 보드의 이동 전과 후를 비교하며, 게임 상태를 변경하지 않고 보드를 이동시킬 수 있다.

데이터 구조에서 또 하나 살펴봐야 할 점은 보드의 구조를 공유하는 방식이다. 모든 보드는 보드 상에서 바뀌지 않은 타일에 대한 참조를 공유한다. 이렇게 함으로써 아주 빠르게 새로운 보드를 만들 수 있으며 참조 일치 비교를 사용해서 보드를 비교할 수 있다. 게임은 GameBoard와 Tiles라는 두 개의 React 컴포넌트를 사용한다.

Tiles는 기본 React 컴포넌트다. 보드를 props로 받아서 계속해서 모든 타일을 렌더링한다. 여기에 CSS3 트랜지션을 사용해서 애니메이션을 적용할 수 있다.

---

```
var Tiles = React.createClass({  
    render: function(){
```

```

var board = this.props.board;
// 먼저 보드 키를 정렬해서 DOM 엘리먼트의 순서를 정렬하는 것을 막는다
var tiles = used_spaces(board).sort(function(a, b) {
    return board[a].id - board[b].id;
});
return <div className="board">{
    tiles.map(function(key){
        var tile = board[key];
        var val = tile_value(tile);
        return <span key={tile.id} className={key + " value" +
val}>{val}</span>;
    })
}
};

<!— 타일 렌더링 예시 →>
<div class="board" data-reactid=".0.1">
<span class="d2 value64" data-reactid=".0.1.$2">64</span>
<span class="d1 value8" data-reactid=".0.1.$27">8</span>
<span class="c1 value8" data-reactid=".0.1.$28">8</span>
<span class="d3 value8" data-reactid=".0.1.$32">8</span>
</div>

/* CSS transition applied to animate tiles */
/* 타일의 애니메이션 적용을 위한 CSS 트랜지션 */
.board span{
    /* ... */
    transition: all 100ms linear;
}

```

---

상태 시스템<sup>state machine</sup>인 GameBoard는 사용자의 방향키 이벤트와 버튼 입력에 응답한다. 또한, 게임 로직 함수와 상호작용해서 새로운 보드의 상태를 변경한다.

---

```

var GameBoard = React.createClass({
getInitialState: function(){
    return this.addTile(this.addTile(initial_board));
},
keyHandler:function(e){
    var directions = {

```

```

37: left,
38: up,
39: right,
40: down
};

if(directions[e.keyCode]
    && this.setBoard(fold_board(this.state, directions[e.keyCode]))
    && Math.floor(Math.random() * 30, 0) > 0){
    setTimeout(function(){
        this.setBoard(this.addTile(this.state));
    }.bind(this), 100);
}
},
setBoard:function(new_board){
    if(!same_board(this.state, new_board)){
        this.setState(new_board);
        return true;
    }
    return false;
},
addTile:function(board){
    var location = available_spaces(board).sort(function() {
        return .5 - Math.random();
    }).pop();
    if(location){
        var two_or_four = Math.floor(Math.random() * 2, 0) ? 2 : 4;
        return set_tile(board, location, new_tile(two_or_four));
    }
    return board;
},
newGame:function(){
    this.setState(this.getInitialState());
},
componentDidMount:function(){
    window.addEventListener("keydown", this.keyHandler, false);
},
render:function(){
    var status = !can_move(this.state)? " - Game Over! ":"";
    return <div className="app">

```

```
<span className="score">
  Score: {score_board(this.state)}{status}
</span>
<Tiles board={this.state}/>
<button onClick={this.newGame}>New Game</button>
</div>
}
});
```

---

GameBoard 컴포넌트에 보드와 상호작용하는 키보드 핸들러를 설정해야 한다. 방향키를 입력할 때마다 setBoard를 호출하면서 게임 로직에서 새로 생성한 보드를 인자로 전달한다. 만약 새로운 보드가 기존의 보드와 다르면 GameBoard 컴포넌트의 상태를 변경한다. 이 방법을 이용하면 불필요한 사이클을 피할 수 있어 성능을 향상할 수 있다.

render 함수는 현재 보드에다가 Tiles 컴포넌트를 렌더링한다. 또한, 점수도 현재 보드를 기준으로 계산하여 표시한다.

addTile 함수는 방향키를 사용할 때마다 새로운 타일을 추가하는 작업을 게임이 끝날 때까지 계속한다. 보드가 가득 차서 숫자를 합칠 수 없으면 게임을 종료한다.

앞의 구현을 살펴보면 어렵지 않게 게임에 실행 취소 기능을 추가할 수 있다. GameBoard 컴포넌트에 모든 보드의 변경 사항을 기록하고, 실행 취소 버튼을 누르면 현재 보드를 변경하게 만들 수 있다. <http://jsfiddle.net/karlmikko/ouxn3qc1/>를 참고한다.

React를 이용하면 게임 로직과 사용자 상호작용을 만드는 데 집중할 수 있고, 화면 동기화를 신경 쓸 필요가 없어 쉽게 이 게임을 구현할 수 있다.

### 17.3 HTML 이메일

React가 웹 환경의 대화형 UI를 개발하는 데 최적화되어 있기는 하지, 핵심은 결국 HTML 렌더링이다. 이것은 HTML 이메일 제작처럼 번거로운 일에도 React를 사용할 수 있다는 뜻이다.

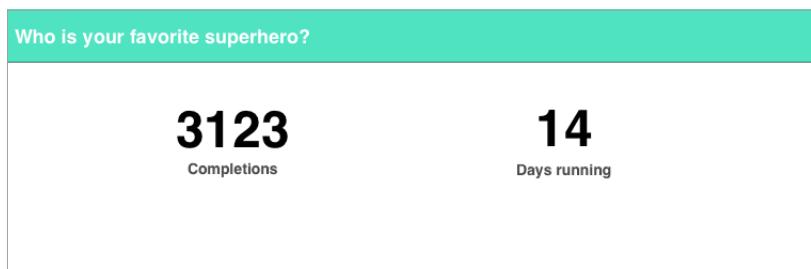
HTML 이메일을 만들 때는 각 이메일 클라이언트에 맞게 여러 개의 테이블을 사용해야 한다. HTML 이메일을 작성할 때는 시간을 되돌려 마치 1999년으로 돌아간 것 같은 느낌으로 HTML을 작성해야 한다.

이메일을 이메일 클라이언트에 제대로 렌더링하기는 쉽지 않다. React로 디자인을 구현하려면 React 사용 여부와 관계 없이 이메일을 만들 때 겪어야 하는 일을 알아두는 것이 좋다.

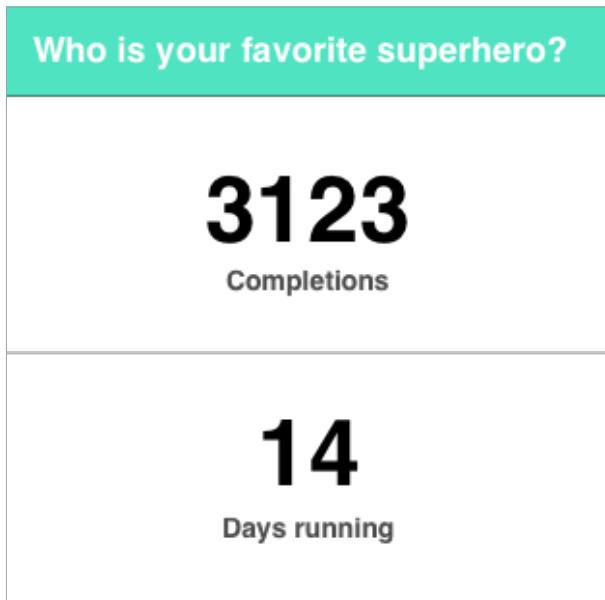
React를 이용한 HTML 이메일 렌더링의 기본 원칙은 `React.renderToString`이다. 이 함수는 하나의 최상위 컴포넌트를 기준으로 전체 컴포넌트 트리를 포함하는 HTML 문자열을 반환한다. `React.renderToString`과 `React.renderToStaticMarkup`는 `data-react-id` 같은 추가 DOM 속성을 만들지 않는 점이다. 이메일은 브라우저 환경의 클라이언트에서 실행하는 것이 아니므로 이런 속성이 필요 없다.

데스크톱과 모바일 디자인을 기준으로 React를 이용해서 이메일을 만들어보자.

[그림 17-3]



[그림 17-4]



이메일 발송에 필요한 HTML을 반환하는 간단한 스크립트를 만들었다.

```
// render_email.js
var React = require('react');
var SurveyEmail = require('survey_email');
var survey = {};

console.log(
  React.renderToString()
);
```

SurveyEmail의 핵심 구조를 만든다. 우선 Email 컴포넌트를 작성한다.

```
var Email = React.createClass({
  render: function () {
    return (
      <html>
        <body>
```

```
        {this.props.children}
    </body>
</html>
);
}
});

```

---

<SurveyEmail/> 컴포넌트에서 <Email/>를 사용한다.

---

```
var SurveyEmail = React.createClass({
  propTypes: {
    survey: React.PropTypes.object.isRequired
  },
  render: function () {
    var survey = this.props.survey;
    return (
      <Email>
        <h2>{survey.title}</h2>
      </Email>
    );
  }
});

```

---

다음으로 디자인에 두 개의 KPI를 나란히 렌더링한다. 각 KPI는 구조가 비슷하기 때문에 같은 컴포넌트를 공유할 수 있다.

---

```
var SurveyEmail = React.createClass({
  render: function () {
    return (
      <table className='kpi'>
        <tr><td>{this.props.kpi}</td></tr>
        <tr><td>{this.props.label}</td></tr>
      </table>
    );
  }
});

```

---

이 컴포넌트를 <SurveyEmail/> 컴포넌트에 추가한다.

---

```
var SurveyEmail = React.createClass({
  propTypes: {
    survey: React.PropTypes.object.isRequired
  },
  render: function () {
    var survey = this.props.survey;
    var completions = survey.activity.reduce(function (memo, ac) { return
      memo + ac; }, 0);
    var daysRunning = survey.activity.length;
    return (
      <Email>
        <h2>{survey.title}</h2>
        <KPI kpi={completions} label='Completions' />
        <KPI kpi={daysRunning} label='Days running' />
      </Email>
    );
  }
});
```

---

KPI 컴포넌트를 쌓아 올리는데 데스크톱이라면 컴포넌트가 옆으로 나란히 놓인다. 다음에는 데스크톱과 모바일에서 모두 사용할 수 있게 만드는 방법을 알아본다. 그 전에 먼저 살펴봐야 할 팀이 있다. CSS 파일을 추가하듯이 <Email/>을 추가하자.

---

```
var fs = require('fs');
var Email = React.createClass({
  propTypes: {
    responsiveCSSFile: React.PropTypes.string
  },
  render: function () {
    var responsiveCSSFile = this.props.responsiveCSSFile;
    var styles;
    if (responsiveCSSFile) {
      styles = <style>fs.readFileSync(responsiveCSSFile)</style>;
    }
  }
});
```

```
        return (
            <html>
                <body>
                    {styles}
                    {this.props.children}
                </body>
            </html>
        );
    }
});
```

---

<SurveyEmail/>을 완성한 모습은 다음과 같다.

```
var SurveyEmail = React.createClass({
  propTypes: {
    survey: React.PropTypes.object.isRequired
  },
  render: function () {
    var survey = this.props.survey;
    var completions = survey.activity.reduce(function (memo, ac) { return
      memo + ac; }, 0);
    var daysRunning = survey.activity.length;
    return (
      <Email responsiveCSS='path/to/mobile.css'>
        <h2>{survey.title}</h2>
        <table className='for-desktop'>
          <tr>
            <td>
              <KPI kpi={completions} label='Completions'/>
            </td>
            <td>
              <KPI kpi={daysRunning} label='Days running'/>
            </td>
          </tr>
        </table>
        <div className='for-mobile'>
          <KPI kpi={completions} label='Completions'/>
          <KPI kpi={daysRunning} label='Days running'/>
        </div>
      </Email>
    );
  }
});
```

```
        </Email>
    );
}
});
```

---

이메일을 데스크톱과 모바일 버전으로 구분했다. 안타깝게도 `float:left` 같은 속성을 이메일에서 사용하는 것을 대부분의 모던 브라우저가 지원하지 않는 데다가 HTML 명세가 더 이상 `align`이나 `valign` 속성을 지원하지 않기 때문에 React 역시 이러한 속성을 지원하지 않는다. 하지만 두 개의 `div`에 `float` 속성을 적용한 것과 비슷한 효과를 줄 수는 있다. 두 개의 그룹 대신에 스크린 사이즈에 따라 노출 여부를 결정하는 반응형 스타일 시트를 이용하면 된다.

테이블을 반드시 사용해야 할 때에도 브라우저에서 컴포넌트를 사용할 때 얻을 수 있는 이점은 이메일 렌더링에 React를 사용할 때도 그대로 얻을 수 있다. 재사용 할 수 있고, 결합할 수 있으며, 테스트하기 좋은 대화형 UI를 만들 수 있다.

## 17.4 차트

설문조사 생성기 애플리케이션에서, 하루에 조사한 설문조사의 수를 차트로 표현해보자. 설문조사 응답자를 나타내는 표에 간단한 스파크라인<sup>Sparkline</sup>을 표현한다.

React가 SVG 태그를 지원하기 때문에 간단하게 SVG를 구현할 수 있다. 스파크라인을 렌더링하려면 몇 가지를 설정하고 `<Path/>`를 사용한다. 여기 완성된 예제가 있다.

---

```
var Sparkline = React.createClass({
  propTypes: {
    points: React.PropTypes.arrayOf(React.PropTypes.number).isRequired
  },
  render: function () {
    var width = 200;
```

```

var height = 20;
var path = this.generatePath(width, height, this.props.points);

return (
  <svg width={width} height={height}>
    <path d={path} stroke="#7ED321" strokeWidth='2' fill='none' />
  </svg>
);
},
generatePath: function (width, height, points) {
  var maxHeight = arrMax(points);
  var maxWidth = points.length;
  return points.map(function (p, i) {
    var xPct = i / maxWidth * 100;
    var x = (width / 100) * xPct;
    var yPct = 100 - (p / maxHeight * 100);
    var y = (height / 100) * yPct;

    if (i === 0) {
      return 'M' + x + ',' + y;
    } else {
      return 'L' + x + ',' + y;
    }
  }).join(' ');
}
);

```

---

앞의 스파크라인 컴포넌트는 위치를 나타내는 숫자의 배열을 기준으로 Path를 이용해서 SVG를 만든다. generatePath 함수가 가장 중요한데, 이 함수는 렌더링할 위치를 계산해서 SVG Path를 반환한다.

이 함수는 "M0,30 L10,20 L20,50" 같은 문자열을 반환한다. SVG Path는 이 문자열을 그리기 명령으로 변환한다. 각 명령은 빈 여백으로 구분한다. "M0,30"은 커서를 x0, y30으로 이동하라는 의미다. 그리고 "L10, 20"은 현재 커서에서 x10, y20까지 선을 그린다.

큰 차트에 이런 식으로 확대/축소 함수를 만드는 일은 지루할 수 있다. D3.js 같은

라이브러리가 제공하는 함수를 이용하면 간단하게 처리할 수 있다. 다음 예를 살펴보자.

---

```
var Sparkline = React.createClass({
  propTypes: {
    points: React.PropTypes.arrayOf(React.PropTypes.number).isRequired
  },
  render: function () {
    var width = 200;
    var height = 20;
    var points = this.props.points.map(function (p, i) {
      return { y: p, x: i };
    });
    var xScale = d3.scale.linear()
      .domain([0, points.length])
      .range([0, width]);
    var yScale = d3.scale.linear()
      .domain([0, arrMax(this.props.points)])
      .range([height, 0]);
    var line = d3.svg.line()
      .x(function (d) { return xScale(d.x) })
      .y(function (d) { return yScale(d.y) })
      .interpolate('linear');

    return (
      <svg width={width} height={height}>
        <path d={line(points)} stroke='#7ED321' strokeWidth='2'
fill='none'/>
      </svg>
    );
  }
});
```

---

## 17.5 정리

이 장에서 살펴본 내용은 다음과 같다.

1. React는 브라우저 밖에서도 사용할 수 있다. 데스크톱 애플리케이션이나 이메일에 사용할 수 있다.
2. 게임 개발에 React를 사용하는 방법을 알아봤다.
3. React를 이용해서 개발한 차트는 D3.js 같은 라이브러리에도 사용할 수 있다.

이제 이 책의 모든 내용을 살펴봤으니 React를 이용해서 다양한 애플리케이션을 개발해 볼 수 있길 바란다.



# 부록: 릴리스로그

## 18.1 React v.0.11.2<sup>01</sup>

몇 가지 기능을 추가한 React v0.11.2를 오늘 공개합니다.

두 가지 새로운 DOM 엘리먼트인 `<dialog>`와 `<picture>`, 그리고 이 요소를 사용할 때 필요한 속성인 `open`, `media`, `sizes`를 지원합니다. 현재 이 기능을 모든 브라우저가 지원하지는 않지만, 사용하기 원하는 사용자를 위해 React에 추가했습니다.

또한, v0.12를 준비하면서 `React.createElement`를 `React.createDescriptor`로 변경해서 버전 간 호환성을 높였습니다. `createDescriptor`를 사용하면 경고를 노출합니다. v0.12에서는 없어질 예정이지만 특별한 영향은 없을 겁니다.

끝으로 어제 `@Scale`<sup>02</sup>에서 발표한 Flow를 통해 TypeScript 같은 타입 주석을 쓸 수 있는 기능을 `jsxttransform`에 추가할 것입니다. `--strip-types`를 명령창에서 추가하거나 API를 호출할 때 `stripTypes`를 `options` 객체에 설정해서 사용할 수 있습니다. Flow에 대해서는 앞으로 몇달 간 더 말씀을 드려야 할 것 같네요. 지금 알려드릴 수 있는 것은 Flow가 흐름에 민감한<sup>flow-sensitive</sup> 자바스크립트

01 <http://facebook.github.io/react/blog/2014/09/16/react-v0.11.2.html>

02 <http://www.atscaleconference.com/>

타입 체커이고, 오픈 소스화를 앞두고 있다는 것뿐입니다.

## React Core

### 새로운 기능

- <dialog> 요소와 open 속성 지원 추가
- <picture> 요소와 media, sizes 속성 지원 추가
- v.0.12 버전을 위해 React.createElementAPI 추가
  - React.createDescriptor는 지원 중단(deprecated)

### JSX

- 이제부터 <picture>는 React.DOM.picture로 파싱할 수 있음

### React Tools

- 정확도 설정을 위해 esprima와 jstransform을 업데이트함
- jsx를 실행할 때 --strip-types 옵션을 추가해서 TypeScript 같은 타입 주석을 제거할 수 있음
  - 이 옵션은 require('react-tools').transform에도 stripTypes로 설정할 수 있음

## 18.2 React v.0.12 RC<sup>03</sup>

드디어 지난 몇 달간의 노력의 산물을 공유합니다. 수정이 많았습니다. 그래서 정식 릴리스 전에 잠재적인 문제를 확실히 정리하기 위해 React v0.12 RC(베포 예정판)을 공개합니다. 이슈를 발견하면 리포트를 부탁합니다. 흥미롭거나 크게 바뀐 부분은 이번에 소개하고 상세한 변경 사항은 정식 릴리스 때 소개하겠습니다.

### React Core

v0.12의 가장 큰 컨셉 변화는 React Elements로 바뀐 부분입니다. 이 내용은 이번 주 초에 언급한 적이 있습니다. 상세한 설명은 Introducing React Elements<sup>04</sup>를 참고해주세요.

---

03 <http://facebook.github.io/react/blog/2014/10/16/react-v0.12-rc1.html>

04 <http://facebook.github.io/react/blog/2014/10/14/introducing-react-elements.html>

## JSX Changes

올해 초에 JSX 명세<sup>05</sup>를 작성하기로 했습니다. 이 덕분에 React를 위한 JSX를 만드는 데 집중할 수 있었고, 같은 공간에서 다른 이들도 혁신을 시도할 수 있게 되었습니다.

### @jsx 컴파일 지시문이 필요하지 않습니다

사실 이 부분은 React를 오픈 소스화 하기 전부터 고민해왔습니다. 이제 `/** @jsx React.DOM */`는 필요 없습니다. React를 위한 JSX transform은 스코프에 이미 React가 있다고 가정합니다(그래도 옵션을 `true`로 설정은 해야합니다). 이 기능을 지원하기 위해 `JSXTransformer`와 `react-tools`를 업데이트했습니다.

### JSX를 함수 호출로 변환하지 않습니다

이제 React는 JSX를 함수 호출로 변환하지 않습니다. 대신 `React.createElement`를 사용하고 이 함수에 인자를 전달합니다. 이렇게 하면 최적화를 할 수 있고 React를 `0m` 같은 도구의 컴파일 대상으로 더욱 잘 지원할 수 있습니다. 더 자세한 사항은 [Introducing React Elements](#)에서 확인하세요.

결과적으로 기존에 임의로 사용하던 함수 호출방식은 이제 더 이상 사용할 수 없습니다. 많은 사람이 이 방법을 사용한다는 것을 알고 있지만, 잃는 것보다 얻는 것이 더 많다고 판단했습니다.

### JSX 소문자 컨벤션

그동안은 HTML 태그 중 일부를 JSX에서 whitelist로 특별히 다뤄왔습니다. 하지만 새로운 HTML 태그가 명세에 추가되고, SVG 태그를 더 많이 지원하면서 매번 이 태그 목록을 업데이트하는 일은 불편했습니다. 게다가 잘 작동할지도 의문이었으며, 언제든 새로운 태그가 등장해서 코드에 영향을 미칠 수도 있죠. 다음 예를 볼까요.

---

05 <https://facebook.github.io/jsx/>

---

```
return <component />;
```

---

component는 HTML 태그일 수도 있고, 아니면 지금은 아니지만 언젠가는 태그가 될 수도 있습니다. 컨벤션을 추가해서 이 문제를 해결했습니다. 소문자로 시작하거나 -를 포함하는 JSX 태그는 HTML로 취급합니다.

새로운 태그를 지원할 때까지 JSX 업데이트를 기다릴 필요가 없다는 것을 의미합니다. 또한, 웹 컴포넌트<sup>Web Components</sup> 같은 커스텀 엘리먼트도 흡수할 수 있습니다. 다만 커스텀 어트리뷰트는 아직 완벽하게 지원하지 않습니다.

세니터 체크를 위해 여전히 whitelist를 사용하고 있습니다. 알려지지 않은 태그는 사용하면 변환할 수 없습니다. 이렇게 하면 제품 코드를 오류없이 업데이트 할 수 있습니다.

덧붙여서 HTML 태그는 React.DOM으로 직접 변환하지 않고 문자열로 변환합니다. 예를 들어 <div/>는 React.DOM.div()가 아니라 React.createElement('div')로 변환합니다.

## JSX 어트리뷰트 나열

예전에는 JSX에 동적으로 속성을 추가할 수 없었습니다. 이제는 ...을 이용해서 속성을 추가할 수 있습니다.

---

```
var myProps = { a: 1, b: 2 };
return <MyComponent {...myProps} />;
```

---

이렇게 하면 객체의 값들을 MyComponent의 속성으로 추가합니다. 더 자세한 사항은 JSX Spread Attributes<sup>06</sup>를 참고하세요.

기존에 속성 객체를 전달하기 위해 함수 호출을 사용했다면 이제는 ...을 이용할 수 있습니다.

---

<sup>06</sup> <https://gist.github.com/sebmarkbage/07bbe37bc42b6d4aef81>

---

```
return MyComponent(myProps);

// JSX Spread Attributes
return <MyComponent {...myProps} />;
```

---

### 중요 변경 사항 : this.props에서 key와 ref 제거

key와 ref는 이미 사용하고 있는 속성 이름입니다. 이 때문에 어떤 객체라도 추가로 props를 받을 수 있으므로 이 값을 명시적이고 정적으로 입력하기가 어려웠습니다. 또한, 최신 가상 머신 상에서 React 내부를 JIT 최적화할 때 방해합니다.

React는 컴포넌트가 생성되기 전부터 외부에서 컴포넌트를 관리하기 때문에, 이 같은 props 일부가 되어서는 안 됩니다.

구분을 명확하게 하려고 key와 ref를 props에서 제외하고 ReactElement로 옮겼습니다. 따라서 이름을 변경해야 합니다.

---

```
someElement.props.key -> someElement.key
```

---

이제 컴포넌트 인스턴스 안에서 this.props.ref나 this.props.key로 접근할 수 없습니다. 다른 이름을 써야 합니다. 하지만 key나 ref를 선언하는 방법을 바꿀 필요는 없습니다. <div key="my-key" />나 div( { key: 'my-key' } )도 여전히 동작합니다.

### 중요 변경사항 : 기본 속성 해결

이것은 미묘한 차이입니다만, defaultProps는 이제 마운트될 때가 아니라 ReactElement를 생성할 때 적용<sup>resolve</sup>됩니다. 속성을 적용하려고 객체를 추가로 만들 필요가 없습니다. 주로 transferPropsTo를 사용할 때 이 문제를 만납니다.

## 지원 중단 : transferPropsTo

transferPropsTo는 v0.12에서 지원을 중단하고 v0.13에서 제거할 계획입니다. 이 기능은 일부 적용 가능한 속성만을 자동으로 추가하는 기능이었습니다. 너무 많은 속성을 전달했고, 적용 가능한 속성을 계속 관리해야 하는 어려움이 있었습니다. 속성의 오탈자도 확인하기 어려웠습니다. 대신 앞에서 소개한 Spread Attributes나 JSX를 사용하기 바랍니다.

---

```
// Spread Attributes  
return <div {...this.props} />;  
  
// JSX  
return div(this.props);
```

---

너무 많은 속성을 전달하는 것을 피하기는 했지만, ES7의 rest 속성 같은 것들을 사용해야 하는 경우가 있을 수 있겠죠. 이에 대한 더 자세한 사항은 upgrading from transferPropsTo<sup>07</sup>에서 확인 바랍니다.

## 지원 중단 : 이벤트 핸들러에서 false를 반환하지 않습니다

그동안은 이벤트 핸들러에서 preventDefault를 할 목적으로 false를 반환할 수 있었습니다. 대부분 브라우저에서 이 기능을 지원했기 때문에 React에서도 지원해왔습니다. 하지만 이는 혼란스러운 API입니다. 다른 값을 반환해야 하는 예도 있습니다. 그래서 지원을 중단합니다. 따라서 event.preventDefault()를 사용하세요.

## 이름이 변경된 API

새로운 React 용어 모음<sup>08</sup>의 일부로 기존 API를 새로운 네이밍 컨벤션에 맞게 변경했습니다.

---

07 <https://gist.github.com/sebmarkbage/a6e220b7097eb3c79ab7>

08 <https://gist.github.com/sebmarkbage/fcb1b6ab493b0c77d589>

- `React.renderComponent` → `React.render`
- `React.renderComponentToString` → `React.renderToString`
- `React.renderComponentToStaticMarkup` → `React.renderToStaticMarkup`
- `React.isValidComponent` → `React.isValidElement`
- `React.PropTypes.component` → `React.PropTypes.element`
- `React.PropTypes.renderable` → `React.PropTypes.node`

이전 API는 경고를 노출하지만 똑같이 작동합니다. v0.13에서 제거할 예정입니다.

### 18.3 React v.0.12<sup>09</sup>

React v0.12를 공개할 수 있어 기쁩니다. 배포 예정 판을 공개한 후 몇 가지 작은 이슈를 수정했습니다. 업데이트를 도와주고 피드백을 준 여러분 감사합니다.

이번 출시에서 생긴 큰 변화에 대해서는 계속해서 이야기를 해왔습니다. 새로운 용어<sup>10</sup>를 소개했고, React의 컨셉을 단순하게 정리하려고 API를 개선했습니다. JSX에도 몇 가지 변화<sup>11</sup>가 있었고, 몇 가지 기능을 더는 지원하지 않습니다. 이 변화에 대해서는 연결된 문서를 참고하길 바랍니다. 다음에서는 이런 변경 사항을 요약합니다. 이외에 다른 변경 사항과 이 변경이 어떤 영향을 주는지도 설명하겠습니다.

#### 새로운 용어와 API 업데이트

v0.12부터 새로운 용어를 사용하기 시작했습니다. 2주 전에 이를 공개하고

---

09 <http://facebook.github.io/react/blog/2014/10/28/react-v0.12.html>

10 <http://facebook.github.io/react/blog/2014/10/14/introducing-react-elements.html>

11 <http://facebook.github.io/react/blog/2014/10/16/react-v0.12-rc1.html>

문서에도 포함<sup>12</sup>했습니다. 이 용어에 맞추어 상위 레벨 API도 수정했습니다. Component는 모든 React.render 메서드에서 제거했습니다. 이 메서드에 전달 하던 인자를 Component라고 했지만 이제는 ReactElements를 전달합니다. 컴포넌트 클래스의 render 메서드에 맞추기 위해 React.renderComponent를 React.render로 변경했습니다.

몇 가지 잘못된 이름도 수정했습니다. React.isValidComponent는 실제로는 인자가 ReactElement인지 확인하기 때문에 React.isValidElement로 변경했습니다. 같은 흐름에서 React.PropTypes.component는 React.PropTypes.element로, React.PropTypes.renderable는 React.PropTypes.node로 변경했습니다.

기존 메서드를 사용하면 경고를 노출하고, v0.13에서 제거할 예정입니다.

## JSX 변경 사항

v0.12 배포 예정 판을 소개하면서 이 부분을 다뤘기 때문에 특징만 소개합니다.

- `/** @jsx React.DOM */`를 더는 입력할 필요가 없음
- 함수 호출로 변환하지 않기 때문에 `<Component/>`를 `React.createElement(Component)`로 변환함
- DOM 컴포넌트는 `React.DOM`을 사용하지 않고 태그 이름을 바로 사용함
- `<div>`를 `React.createElement('div')`로 변환함
- Spread Attributes를 사용해서 props를 더욱 쉽게 전달함

## 개발자 도구 개선, `_internals` 제거

지난 몇 달간 React 개발자 도구의 메시지에 대한 불만사항을 접수했습니다. 원래는 이미 개발자 도구를 사용하고 있으므로 로그가 필요하지 않습니다. 안타깝지만 이는 React 개발자 도구가 React를 아는 형태로 개발했기 때문에 이런 문제가 발생했습니다. 이 문제에 대해 인지를 하게 되었고, React가 개발자 도구가 설

---

12 <http://facebook.github.io/react/docs/glossary.html>

치되어 있는지 알 수 있게 했습니다. 이것이 가능하도록 몇 주 전에 개발자 도구를 업데이트했습니다. 크롬 확장 프로그램은 자동으로 업데이트되기 때문에 아마도 이미 업데이트가 설치되어 있을 것입니다.

이 업데이트를 하고 나면 몇 가지 내부 모듈을 더는 외부로 노출하지 않습니다. 이러한 상세 구현을 이용하고 있었다면 코드가 동작하지 않을 겁니다. 다시는 React.\_\_internals를 사용할 수 없습니다.

## 라이선스 변경 – BSD

React의 라이선스를 BSD 3-Clause로 변경했습니다. 이전에는 Apache 2 라이선스를 사용하고 있었습니다. 이 라이선스는 서로 매우 유사하고 추가적인 특허 허가는 아파치 라이선스에서 제공하는 것과 동등합니다. Facebook의 주요 오픈 소스 프로젝트와 같은 라이선스 정책으로 변경했습니다. 라이선스 전문<sup>13</sup>과 PATENTS 파일은 Github에서 확인할 수 있습니다.

## React Core

### 중요 변경 사항

- key와 ref를 props에서 제거하였고 엘리먼트에서 바로 접근할 수 있음
- React의 라이선스가 특허권과 함께 BSD 라이선스로 변경됨
- 기본 속성 적용이 마운트 시점이 아니라 엘리먼트 생성 시점으로 변경됨
- React.\_\_internals를 제거함. 개발자 도구를 위해 사용했으나 이제 필요가 없게 됨
- 컴포넌트 구성 함수를 직접 호출하지 않음. 반드시 먼저 React.createFactory로 감싸야 함

### 새로운 기능

- this.transferPropsTo 지원 중단을 위해 {...}을 추가함
- HTML 속성 acceptCharset, classID, manifest를 추가로 지원함

---

<sup>13</sup> <https://github.com/facebook/react/blob/master/LICENSE>

## 지원 중단

- `React.renderComponent` → `React.render`
- `React.renderComponentToString` → `React.renderToString`
- `React.renderComponentToStaticMarkup` → `React.renderToStaticMarkup`
- `React.isValidComponent` → `React.isValidElement`
- `React.PropTypes.component` → `React.PropTypes.element`
- `React.PropTypes.renderable` → `React.PropTypes.node`
- DEPRECATED `React.isValidClass`
- DEPRECATED `instance.transferPropsTo`
- DEPRECATED `preventDefault`를 목적으로 이벤트 핸들러에서 `false` 반환하는 기능
- DEPRECATED 컴포넌트 구성은 `React.createFactory`로 감싸는 것으로 변경
- DEPRECATED `key={null}`를 키를 할당하는 기능을 지원 중단

## 버그 수정

- 중첩된 결과에서 이벤트와 업데이트 핸들링을 개선하고, 겹쳐진 제어 컴포넌트에서 값 복원을 수정
- `event.getModifierState`를 대소문자에 맞게 다름
- `event.charCode` 표준화 개선
- `autobound` 메서드와 관련된 경우, 더 나은 오류 스택
- 개발자 도구가 설치되어 있는 경우, 개발자 도구 메시지를 표시하지 않게 함
- 브라우저 간 필요 언어 기능을 정확하게 감지함
- 몇 가지 HTML 속성에 대한 지원을 수정
- `list`가 정상적으로 업데이트됨
- `scrollLeft`, `scrollTop`이 제거되었으며, `prop`로 선언하지 않음
- 오류 메시지 개선

## React with Add-Ons

### 새로운 기능

- 업데이트 사이클에 `React.addons.batchedUpdates`를 추가함. ##### 중요 변경사항

- React.addons.update가 copyProperties 대신 assign을 사용함. assign에서 hasOwnProperty 검사를 포함함. 프로토 타입의 속성을 더 이상 업데이트하지 않음. #####  
버그 수정
- CSS 트랜지션과 관련한 이슈를 수정

## JSX

### 중요 변경 사항

- 소문자 태그명은 항상 HTML 태그로 간주하고, 대문자 태그는 컴포넌트 조합으로 간주
- JSX를 함수 호출로 변환하지 않음

### 새로운 기능

- @jsx React.DOM 불필요
- {...}으로 더 쉽게 prop를 사용할 수 있음

### 버그 수정

- JSXTransformer : API를 직접 사용할 때 소스 맵 옵션으로 만들(예: react-rails)

## 18.4 React v.0.12.2<sup>14</sup>

지난 두 달간 master에 추가한 몇 가지 작은 수정사항을 모아 v0.12.2를 배포합니다.

v0.12.1을 건너뛴 것을 아는 분이 계실지 모르겠네요. v0.12.1은 몇 가지 transform과 관련한 내용만 변경했기에 별도로 안내하지 않았습니다. JSX를 실행할 때 옵션을 추가하여 안전하게 Flow 기반의 코드를 Javascript로 변환할 수 있습니다. v0.12.1을 적용하지 않고 바로 v0.12.2를 적용해도 아무 문제가 없습니다.

---

14 <http://facebook.github.io/react/blog/2014/12/18/react-v0.12.2.html>

## 변경 사항

### React Core

- HTML 속성 지원 추가: formAction, formEncType, formMethod, formTarget, marginHeight, marginWidth
- strokeOpacity를 단위 없는 CSS 속성에 추가
- 후행 쉼표(trailing commas) 제거(npm 모듈 번들과 IE8 사용이 가능하게 됨)
- React.createElement에 undefined를 전달할 때 오류가 생기는 버그를 수정하여 경고가 나오게 함

### React Tools

- JSX 관련 변환은 props와 displayName에 쌍따옴표를 사용함

## 18.5 React v.0.13 Beta 1<sup>15</sup>

React v0.13은 여러 가지 좋은 기능을 가지고 있습니다. 그 중에도 특별한 한 가지가 있습니다. 내일 아침 React.js Conf가 정말 기다려지는 이유 중 하나입니다.

컨퍼런스에 참석하는 사람도 있고, 운이 좋지 않아서 컨퍼런스에 참석하지 못할 도 있겠죠. 어떤 경우든 컨퍼런스 전까지 v0.13을 재미있게 살펴보시길 바랍니다.

### 자바스크립트 클래스 지원

원래 자바스크립트는 클래스를 지원하지 않았습니다. 대부분의 인기 있는 프레임워크가 클래스를 지원하기 때문에 React도 클래스를 만들었습니다. 이는 곧 프레임워크마다 조금씩 다르게 제공하는 클래스 사용 방식을 배워야 한다는 뜻입니다.

클래스 시스템을 만드는 건 우리가 해야 할 일이 아니라는 걸 깨달았습니다. 자바스크립트가 지원하는 고유의 방식으로 클래스를 만들 수 있길 원했죠.

React 0.13.0부터는 React.createClass를 사용하지 않고도 React 컴포넌트

---

<sup>15</sup> <http://facebook.github.io/react/blog/2015/01/27/react-v0.13.0-beta-1.html>

를 만들 수 있습니다. 트랜스파일러를 사용한다면 ES6 클래스를 사용할 수도 있습니다. react-tools에 추가한 트랜스파일러를 이용해서 harmony 옵션을 추가하세요: `jsx --harmony`

## ES6 클래스

---

```
class HelloMessage extends React.Component {  
  render() {  
    return <div>Hello {this.props.name}</div>;  
  }  
}  
  
React.render(<HelloMessage name="Sebastian" />, mountNode);
```

---

API는 `getInitialState`를 제외하면 여러분이 생각한 것과 크게 다르지 않을 것입니다. 간단한 인스턴스 속성을 사용해서 클래스의 상태를 자연스럽게 작성할 수 있습니다. `getDefaultProps`이나 `propTypes` 같은 경우도 생성자의 속성입니다.

```
export class Counter extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {count: props.initialCount};  
  }  
  tick() {  
    this.setState({count: this.state.count + 1});  
  }  
  render() {  
    return (  
      <div onClick={this.tick.bind(this)}>  
        Clicks: {this.state.count}  
      </div>  
    );  
  }  
}  
  
Counter.propTypes = { initialCount: React.PropTypes.number };  
Counter.defaultProps = { initialCount: 0 };
```

---

## ES67+ Property Initializer

클래스를 선언할 때 속성을 할당하는 일은 매우 중요합니다. 그런데도 이런 방식을 지원하는 이유는 이것이 더 자연스럽기 때문입니다. 속성에 대해서는 다음 버전의 자바스크립트에서 좀 더 서술적인 문법을 지원할 예정입니다. 아마도 다음과 같은 방식이 되겠죠.

---

```
// 미래 버전
export class Counter extends React.Component {
  static propTypes = { initialCount: React.PropTypes.number };
  static defaultProps = { initialCount: 0 };
  state = { count: this.props.initialCount };
  tick() {
    this.setState({ count: this.state.count + 1 });
  }
  render() {
    return (
      <div onClick={this.tick.bind(this)}>
        Clicks: {this.state.count}
      </div>
    );
  }
}
```

---

TypeScript의 Property Initializer에서 영감을 얻었습니다.

## 자동바인딩(Autobinding)

React.createClass는 모든 메서드를 this에 바인딩합니다. 다른 형태의 클래스에서 이런 기능을 다뤄보지 않았거나 React를 사용하다가 다른 형태의 클래스를 다뤄야 하는 일이 생긴다면 조금 혼란스러울 수 있습니다. 따라서 React 클래스 모델에서 이 기능을 제거했습니다. 생성자에서 별도로 메서드를 바인딩합니다.

---

```
class Counter extends React.Component {
  constructor() {
    super();
    this.tick = this.tick.bind(this);
```

```
    }
    tick() {
      ...
    }
    ...
}
```

---

Property Initializer를 사용하면 문법적으로 이 부분을 좀 더 쉽게 풀어낼 수 있을 것입니다.

---

```
class Counter extends React.Component {
  tick = () => {
    ...
  }
  ...
}
```

---

## 믹스인

안타깝지만 React에 ES6 클래스를 위한 믹스인 지원은 하지 않을 예정입니다. 믹스인은 자연스러운 자바스크립트 컨셉을 지원하는 목적에서 어긋납니다.

자바스크립트로 믹스인을 만드는 표준이나 일반적인 방법이 없습니다. 사실 믹스인 지원을 위한 몇 가지 기능이 ES6에서 제외되었습니다. 여러 가지 라이브러리가 서로 다른 방법을 사용합니다. 우리는 믹스인을 선언해서 어떤 자바스크립트 클래스에도 사용할 수 있는 한 가지 방법을 찾고 있습니다. React에서 새로운 방법을 만드는 것은 이런 노력에 아무런 도움이 되지 않는다고 판단했습니다.

따라서 커다란 JS 커뮤니티와 함께 믹스인 표준을 만들기 위해 노력할 것입니다. 믹스인 없이 일반적인 작업을 쉽게 할 수 있는 API 설계를 시작했습니다. 어떤 형태의 Flux Store에도 적용할 수 있는 일급 구독First-class subscriptions 같은 것 말이죠.

여전히 `React.createClass`를 이용해서 믹스인을 사용할 수 있습니다.

참고: `React.createClass`를 이용한 기존 클래스 생성 방법도 여전히 유효합니다.

## 다른 언어의 클래스까지

이 클래스는 평범한 자바스크립트 클래스이기 때문에 TypeScript나 CoffeeScript의 클래스도 사용할 수 있습니다.

---

```
div = React.createFactory 'div'

class Counter extends React.Component
  @propTypes = initialCount: React.PropTypes.number
  @defaultProps = initialCount: 0

  constructor: (props) ->
    super props
    @state = count: props.initialCount

  tick: =>
    @setState count: @state.count + 1

  render: ->
    div onClick: @tick,
      'Clicks: '
      @state.count

    div onClick: @tick,
      'Clicks: '
      @state.count
```

---

ES3의 모듈 패턴도 쓸 수 있습니다.

---

```
function MyComponent(initialProps) {
  return {
    state: { value: initialProps.initialValue },
    render: function() {
      return <span className={this.state.value} />
    }
  };
}
```

---

## 18.6 React v.0.13 RC<sup>16</sup>

주말에 걸쳐 React v0.13의 첫 번째 배포 예정 판을 공개했습니다.

수정사항에 대해서는 이미 간단히 설명했습니다. ES6 클래스를 지원하는 게 눈에 띕니다. 이 내용은 v0.13.0 Beta 릴리스에서 좀 더 자세하게 설명했습니다. 이번 업데이트는 매우 흥분됩니다. Sebastian은 오늘 아침 ReactElement에 대한 수정사항<sup>17</sup>을 포스팅하기도 했습니다. 이런 변화가 성능과 개발자 경험에 개선을 가져올 수 있길 기대합니다.

### React Core

#### 중요 변경 사항

- props를 엘리먼트 생성 후에 변경하는 기능을 중단하고 사용할 때는 경고를 표시할 예정. 향후 React는 props가 변경되지 않을 것을 가정하고 성능을 최적화할 예정
- statics에 선언한 정적 메서드가 더는 컴포넌트 클래스에 자동으로 바인딩 되지 않음
- ref 적용 순서를 변경해 컴포넌트의 ref를 componentDidMount 메서드 호출 직후 바로 사용할 수 있음. 이 변화는 componentDidMount 내부에서 부모 컴포넌트의 콜백을 호출하는 경우에만 확인할 수 있음. 이런 방식은 앤티패턴(anti-pattern)이며 권장하지 않음
- 라이프사이클 메서드에서 setState에 대한 호출은 이제 비동기로 일괄 처리함. 이전에는 첫 번째 마운트의 첫 번째 호출은 동기로 처리했음
- 마운트되지 않은 컴포넌트의 setState와 forceUpdate는 오류 대신 경고를 표시. Promise를 사용하는 경우에 대비한 변화임
- 내부 속성을 접근할 수 없음. this.\_pendingState와 this.\_rootNodeID를 포함

#### 새로운 기능

- ES6 클래스를 사용해서 React 컴포넌트를 만들 수 있습니다. 자세한 사항은 v0.13 Beta1의 설명을 참고하세요.
- 새로운 최상위 API로 React.findDOMNode(component)를 추가했습니다. component.

<sup>16</sup> <http://facebook.github.io/react/blog/2015/02/24/react-v0.13-rc1.html>

<sup>17</sup> <http://facebook.github.io/react/blog/2015/02/24/streamlining-react-elements.html>

`getDOMNode()`를 대신해 사용할 수 있습니다. ES6 클래스를 이용해 생성한 컴포넌트는 `getDOMNode`을 사용할 수 없습니다. 이제 더 나은 다른 패턴을 사용하세요.

- 새로운 `ref` 스타일은 이름에 사용할 콜백을 지정할 수 있습니다: `<Photo ref={c} => this._photo = c</>`를 사용하면 `this._photo`와 관련하여 컴포넌트를 참조할 수 있습니다 (`ref="photo"`가 `this.refs.photo`가 되는 것과 다릅니다).
- `this.setState()`는 이제 `state` 업데이트를 위해 첫 번째 인자로 함수를 받습니다. `this.setState(state, props) => ({count: state.count + 1})`; 같은 형태입니다. 이는 즉 `this._pendingState`를 사용할 필요가 없다는 뜻이며, 이 기능은 더는 지원하지 않습니다.
- 이터레이터와 `immutable-js` 시퀀스를 자식으로 지원합니다.

## 지원 중단

- ES6 클래스를 사용해서 React 컴포넌트를 만들 수 있습니다. 자세한 사항은 v0.13 Beta1의 설명을 참고하세요.

## React with Add-Ons

### 지원 중단

- `React.addons.classSet`를 지원하지 않습니다. 이 기능은 다른 모듈을 이용해서 대체 할 수 있습니다. `classnames18`가 한 가지 예입니다.

## React Tools

### 중요 변경 사항

- ES6 문법으로 변환할 때 `class` 메서드를 이제 더는 기본으로 열거하지 않습니다. 이 메서드는 `Object.defineProperty`를 필요로 합니다. IE8 같은 브라우저 지원이 필요한 경우에는 `--target es3`를 옵션으로 추가할 수 있습니다.

### 새로운 기능

- `jsx` 명령에 `--target` 옵션을 추가하여 대상 ECMAScript 버전을 지정할 수 있습니다.
- 기본값은 ES5입니다.
- ES3으로 설정하면 기존의 기본 기능을 복원합니다. 이 경우에는 안전한 예약어 사용을 위해 추가로 변환합니다. 예를 들면 IE8 호환을 위해서 `this.static`을

---

<sup>18</sup> <https://www.npmjs.com/package/classnames>

'''this['static']으로 변환합니다.

- {...}도 변환이 가능합니다.

## JSX

### 중요 변경 사항

- JSX 파싱에 변경이 있으며, 이는 특히 >이나 }를 엘리먼트 내부에서 사용하는 경우에 관련되어 있습니다. 이전에는 문자열로 간주하였지만 이제부터는 파싱 오류가 됩니다. JSX 코드의 잠재적인 이슈를 찾고 수정하기 위해 독립적인 실행 파일을 공개할 예정입니다.

## 18.7 React v.0.13 RC2<sup>19</sup>

배포 예정 판을 테스트해준 여러분께 감사 인사를 전합니다. 몇 가지 좋은 피드백을 받아서 두 번째 배포 예정 판을 공개합니다. 변경 사항은 많지 않습니다. 지난 번 배포 예정 판에서 공개한 API에 대한 변경은 없습니다. 변경 사항을 간단히 정리해보면 이렇습니다.

- 새로운 API로 `React.cloneElement` 추가
- `React.addons.createFragment` API를 사용하는 경우 `propTypes` 검증과 관련된 오류를 수정
- 몇 가지 경고 메시지 개선
- `jstransform`과 `esprima` 업데이트

### React.cloneElement

React v0.13 RC2에서 추가한 새로운 API로, `React.addons.cloneWithProps`와 유사하며 다음과 같이 사용합니다.

---

```
React.cloneElement(element, props, ...children);
```

---

`cloneWithProps`와 다르게, 이 새로운 기능은 `style`과 `className`을 자동으로 병합

---

<sup>19</sup> <http://facebook.github.io/react/blog/2015/03/03/react-v0.13-rc2.html>

하는 내장 기능 같은 것은 가지고 있지 않습니다. 자동으로 지원하는 기능은 아무도 정확히 알 수 없습니다. 따라서 style에 다른 내용이 있다면 코드에 대한 이해와 재사용을 어렵게 합니다. `React.cloneElement`는 다음과 거의 비슷합니다.

---

```
<element.type {...element.props} {...props}>{children}</element.type>
```

---

`ref`도 포함한다는 점이 JSX나 `cloneWithProps`와 다릅니다. `ref`로 자식을 가져오면 실수로 부모에게서 가져올 일이 없다는 것입니다. 새로운 엘리먼트에 추가한 `ref`를 똑같이 가져옵니다.

자식을 순회하며 새로운 속성을 추가하는 것은 흔한 패턴입니다. `cloneWithProps`가 `ref`를 놓치는 경우가 있다는 보고를 수차례 받았습니다. 이 때문에 코드를 이해하기 어려웠을 것입니다. `cloneElement`를 이용해서 같은 패턴을 구현하면 기대한 대로 작동합니다. 다음 예를 살펴보세요.

---

```
var newChildren = React.Children.map(this.props.children, function(child) {
  return React.cloneElement(child, { foo: true })
});
```

---

참고: `React.cloneElement(child, { ref: 'newRef' })`는 `ref`를 덮어씁니다. 따라서 콜백 `ref`를 사용하지 않는 한, 두 개의 부모가 한 자식에 `ref`를 갖는 것은 불가능합니다.

이 기능은 React 0.13에서 중요한 기능이기도 합니다. 이제 `props`가 불변이기 때문입니다. 업그레이드 경로로 자주 엘리먼트를 복제하는데, 이 때문에 `ref`를 잊어버리는 경우가 있습니다. 따라서 좀 더 나은 업그레이드 경로가 필요했습니다. Facebook의 `callsites`를 업그레이드하면서 이 메서드가 필요해졌습니다. 커뮤니티에서도 같은 피드백을 받았습니다. 이 때문에 정식 배포 전에 마지막으로 배포 예정 판을 내놓아 확인하는 절차를 추가했습니다.

결과적으로는 `React.addons.cloneWithProps`는 지원을 중단할 예정입니다. 지금 당장은 아니지만 `React.cloneElement`로 대체하는 것을 고려해볼 만한 좋은 시점입니다. 사전에 이 기능의 지원 중단에 대해 공지할 예정이기 때문에 당장 수정이 필요한 것은 아닙니다.

## 18.8 React v.0.13<sup>20</sup>

React v0.13을 공개합니다.

가장 주목할만한 기능은 ES6 클래스 지원입니다. 이제 컴포넌트를 더 유연하게 작성할 수 있습니다. ES6 클래스로 `React.createClass`를 완전히 대체하는 것이 React의 최종 목표입니다. 그렇지만 현재의 믹스인 사용 방법과 클래스 속성 초기화를 얻어 수준에서 지원하기 전까지는 `React.createClass` 지원을 중단하지는 않을 것입니다.

지난주에 있었던 EmberConf와 ng-conf에서 Ember와 Angular가 React와 견줄 만큼 속도를 개선한 것에 감탄했습니다. 성능이 React를 선택하는 이유라고 생각하지 않습니다만, React를 더욱 빠르게 만들 최적화를 계획하고 있습니다.

최적화 진행을 위해서는 `ReactElement` 객체가 불변이어야 합니다. 자연스러운 React 코드를 작성할 때 언제나 가장 좋은 방법이기도 합니다. 이번 버전에서는 엘리먼트를 생성하거나 렌더링할 때 `props`를 추가하거나 변경하면 경고 메시지를 노출합니다.

새로운 버전을 적용하면서 아마 `React.createElement` API의 사용을 고려할 수 있습니다. 이 기능은 `React.addons.cloneWithProps`와 비슷하지만, `key`와 `ref`를 보전하고 `style`과 `className`을 자동으로 병합하지 않습니다. 최적화

---

<sup>20</sup> <http://facebook.github.io/react/blog/2015/03/10/react-v0.13.html>

에 대한 더욱 자세한 계획에 관심이 있다면 Github의 이슈 #3226<sup>21</sup>, #3227<sup>22</sup>, #3228<sup>23</sup>을 확인하길 바랍니다.

## React Core

### 중요 변경 사항

- v0.12에서 경고 메시지를 노출했던 지원 중단된 패턴을 제거하여 사용할 수 없습니다. 특히 컴포넌트 클래스를 JSX나 `React.createElement` 사용하지 않거나 JSX 또는 `createElement`와 함께 컴포넌트가 아닌 함수를 사용하는 경우가 이에 해당합니다.
- `props`를 엘리먼트 생성 후에 변경하는 기능을 중단하고 사용할 때에는 경고를 표시합니다. 향후 React는 `props`가 변경되지 않을 것을 가정하고 성능을 최적화할 예정입니다.
- `statics`에 선언한 정적 메서드가 더는 컴포넌트 클래스에 자동으로 바인딩 되지 않습니다.
- `ref` 적용 순서를 변경해 컴포넌트의 `ref`를 `componentDidMount` 메서드 호출 직후 바로 사용할 수 있습니다. 이 변화는 `componentDidMount` 내부에서 부모 컴포넌트의 콜백을 호출하는 경우에만 확인할 수 있습니다. 이런 방식은 안티패턴(anti-pattern)이며 권장하지 않습니다.
- 라이프사이클 메서드에서 `setState`에 대한 호출은 이제 비동기로 일괄처리합니다. 이전에는 첫 번째 마운트의 첫 번째 호출은 동기로 처리했었습니다.
- 마운트되지 않은 컴포넌트의 `setState`와 `forceUpdate`는 오류 대신 경고를 표시합니다. `Promise`를 사용하는 때를 대비한 변화입니다.
- 내부 속성에 접근할 수 없습니다. `this._pendingState`와 `this._rootNodeID`를 포함합니다. ##### 새로운 기능
- React 엘리먼트 복제를 위한 새로운 최상위 API인 `React.cloneElement(el, props)`가 추가되었습니다. RC2의 설명을 참고바랍니다.
- ES6 클래스를 사용해서 React 컴포넌트를 만들 수 있습니다. 자세한 사항은 v0.13 Beta1의 설명을 참고하세요.
- 새로운 최상위 API로 `React.findDOMNode(component)`를 추가했습니다. `component.getDOMNode()`를 대신해 사용할 수 있습니다. ES6 클래스를 이용해 생성한 컴포넌트는

---

21 <https://github.com/facebook/react/issues/3226>

22 <https://github.com/facebook/react/issues/3227>

23 <https://github.com/facebook/react/issues/3228>

`getDOMNode`를 사용할 수 없습니다. 이제 더는 다른 패턴을 사용하세요.

- 새로운 `ref` 스타일은 이름에 사용할 콜백을 지정할 수 있습니다: `<Photo ref={ (c) => this._photo = c} />`를 사용하면 `this._photo`와 관련하여 컴포넌트를 참조할 수 있습니다 (`ref="photo"`가 `this.refs.photo`가 되는 것과 다릅니다).
  - `this.setState()`는 이제 `state` 업데이트를 위해 첫번째 인자로 함수를 받습니다. `this.setState( (state, props) => ({count: state.count + 1}))`; 같은 형태입니다. 이는 즉 `this._pendingState`를 사용할 필요가 없다는 뜻이며, 이 기능은 더이상 지원하지 않습니다.
  - 이터레이터와 `immutable-js` 시퀀스를 자식으로 지원합니다.

## 지원 중단

- `ComponentClass.type` 지원을 중단합니다. `ComponentClass`를 사용하세요(보통 `element.type === ComponentClass` 같은 형태입니다).
- `createClass`기반의 컴포넌트에서 사용할 수 있었던 몇 가지 메서드를 ES6 클래스에서 지원 중단하거나 제거했습니다(`getDOMNode`, `replaceState`, `isMounted`, `setProps`, `replaceProps`).

## React with Add-Ons

### 새로운 기능

- 전체 자식에 키를 추가할 수 있는 `React.addons.createFragment`를 추가<sup>24</sup>했습니다.

## 지원 중단

- `React.addons.classSet`는 지원하지 않습니다. 이 기능은 다른 모듈을 이용해서 대체할 수 있습니다. `classnames`<sup>25</sup>가 한 가지 예입니다.
- `React.addons.cloneWithProps`를 `React.cloneElement`로 대체할 수 있습니다. `style`과 `className`은 필요에 따라 수동으로 병합해야 한다는 점을 주의하세요.

## React Tools

### 중요 변경 사항

- ES6 문법으로 변환할 때, `class` 메서드를 이제 더는 기본으로 열거하지 않습니다. 이 메서드는 `Object.defineProperty`를 필요로 합니다. IE8 같은 브라우저 지원이 필요한 경우

---

<sup>24</sup> <http://facebook.github.io/react/docs/create-fragment.html>

<sup>25</sup> <https://www.npmjs.com/package/classnames>

에는 --target es3를 옵션으로 추가할 수 있습니다.

## 새로운 기능

- jsx 명령에 '--target' 옵션을 추가하여 대상 ECMAScript 버전을 지정할 수 있습니다.
- 기본값은 ES5입니다.
- ES3로 설정하면 기존의 기본 기능을 복원합니다. 이 경우에는 안전한 예약어 사용을 위해 주가적인 변환을 진행합니다. 예를 들면 IE8 호환을 위해서 `this.static`를 `'''this['static']'`으로 변환합니다.
- `{...}`도 변환이 가능합니다.

## JSX

### 중요 변경 사항

- JSX 파싱에 변경이 있으며, 이는 특히 `>`이나 `}`를 엘리먼트 내부에서 사용하는 경우에 관련되어 있습니다. 이전에는 문자열로 간주하였지만 이제부터는 파싱 오류가 됩니다. JSX 코드의 짐작적인 이슈를 찾고 수정하기 위해 독립적인 실행파일을 공개할 예정입니다.

## 18.9 React v.0.13.1<sup>26</sup>

v0.13.0을 내놓은 지 한 주도 채 되지 않아 작은 이슈를 수정한 새로운 버전을 배포합니다. 애플리케이션을 업그레이드하고 이슈를 보고해 준 분들께 감사 인사를 전합니다. 거기에 이슈를 보고하고 직접 수정까지 해준 분들께는 더 큰 감사를 전하고 싶습니다. 6개의 수정사항 중 2개는 React Core 팀원이 아닌 분들이 수정해주셨습니다.

### React Core

#### 버그 수정

- 비어 있는 `<select>` 요소를 렌더링할 때 오류를 발생시키지 않습니다.
- `style`을 `null`에서 트랜지션할 때 확실히 업데이트합니다.

---

26 <http://facebook.github.io/react/blog/2015/03/16/react-v0.13.1.html>

## React width Add-Ons

### 버그 수정

- TestUtils : ES6 클래스에서 getDOMNode에 관한 경고를 하지 않습니다.
- TestUtils: 래핑한 전체 페이지 컴포넌트(〈html〉, 〈head〉, 〈body〉)를 DOM 컴포넌트로 간주합니다.
- Perf: DOM 컴포넌트 이중 계산을 중단했습니다.

## React Tools

### 버그 수정

- --non-strict-es6module 옵션 파싱을 수정했습니다.

## 18.10 React v.0.13.2<sup>27</sup>

어제 React Native Team이 v0.4를 공개했습니다. 몇 빌자 옆에서 일하는 웹 팀 소속인 우리도 가만히 있을 수 없죠. v0.13.2를 공개합니다. 몇 가지 버그를 수정했습니다. v0.14를 위한 작업은 계속 진행하고 있습니다.

## React Core

### 새로운 기능

- strokeDashoffset, flexPositive, flexNegative를 단위 없는 CSS 속성에 추가
- DOM 속성 지원 추가
- 〈style〉 요소의 scoped 속성
- 〈meter〉 요소의 high, low, optimum 속성
- unselectable: IE에서만 지원하는 사용자 선택 방지를 위한 속성

### 버그 수정

- null을 렌더링 한 후 재렌더링할 때 문맥을 제대로 전달하지 못하는 경우를 수정

---

<sup>27</sup> <http://facebook.github.io/react/blog/2015/04/18/react-v0.13.2.html>

- style={null}로 렌더링한 후에 재렌더링할 때 적절히 style을 업데이트하지 못하는 현상을 수정
- IE8의 버그를 방지하기 위해 uglify 의존성 업데이트
- 경고 개선

## React width Add-Ons

### 버그 수정

- Immutability Helpers: hasOwnProperty를 객체 키로 지원합니다.

## React Tools

- 새로운 옵션에 관한 문서 개선

## 18.11 React v.0.13.3<sup>28</sup>

오늘 v0.13 브랜치에 새로운 패치를 내놓았습니다. 아주 작은 변화가 몇 가지 있고, 문서화는 하지 않았지만 이미 많이 사용하고 있는 context와 관련한 이슈가 있었습니다. 또한, 몇 가지 경고 메시지를 개선했습니다.

## React Core

### 새로운 기능

- clipPath 요소와 SVG를 위한 속성 추가
- 자바스크립트 클래스에서 더는 지원하지 않는 기능에 대한 경고 메시지 개선

### 버그 수정

- dangerouslySetInnerHTML에 대한 제한을 느슨하게 하여 {\_\_html: undefined} 가 오류를 발생하지 않게 함.
- 비순수 getChildContext에서 발생하는 무관한 문맥 경고를 수정
- replaceState(obj)가 obj의 프로토 타입을 유지하도록 수정

---

<sup>28</sup> <http://facebook.github.io/react/blog/2015/05/08/react-v0.13.3.html>

### 버그 수정

- TestUtils: 컴포넌트에 contextTypes을 선언했을 때 얇은 렌더링이 작동하게 했습니다.

## 18.12 React v.0.14 Beta 1<sup>29</sup>

React 커뮤니티의 많은 사람이 이번 주 파리에서 열린 ReactEurope<sup>30</sup>에 참석했습니다. 이번 행사는 두 번째로 열리는 React 컨퍼런스이기도 합니다. 지난번 컨퍼런스에서는 React 0.13 Beta를 출시했습니다. 이번 컨퍼런스를 앞두고도 지난번처럼 React 0.14의 첫 번째 베타 버전을 출시합니다. 컨퍼런스에 참석했다가 돌아가는 분은 물론이고, 참석하지 못한 분들에게도 즐거운 소식이길 바랍니다.

React 0.14에서도 React의 발전은 계속되며, 이번에는 API를 정착시키기 위한 작은 변화를 담았습니다. 여기에서는 두 가지 큰 변화를 소개합니다. 최종 릴리스를 선보일 때 문서를 수정하면서 모든 변경사항도 함께 소개하겠습니다.

이번 베타를 설치하려면 `npm install react@0.14.0-beta1`과 `npm install react-dom@0.14.0-beta1`을 실행합니다. `react-tools` 지원 중단 안내<sup>31</sup>에서 소개해 드렸지만, 더는 `react-tools` 패키지를 업데이트하지 않기 때문에 이번 업데이트는 `react-tools`의 최신 버전을 포함하지 않습니다.

새로운 버전을 살펴보고, 의견이 있거나 새로운 이슈가 있다면 GitHub 저장소에 이슈를 등록해주세요.

---

29 <http://facebook.github.io/react/blog/2015/07/03/react-v0.14-beta-1.html>

30 <https://www.react-europe.org/>

31 <https://facebook.github.io/react/blog/2015/06/12/deprecating-jstransform-and-react-tools.html>

## React와 React DOM의 분리

react-native<sup>32</sup>, react-art<sup>33</sup>, react-canvas<sup>34</sup>, react-three<sup>35</sup> 같은 패키지를 살펴보면 React의 아름다움과 정수는 브라우저나 DOM에 국한되지 않는다는 것이 확실합니다.

컴포넌트와 엘리먼트에 대한 단순한 발상이 React의 본질이라고 생각합니다. 렌더링하고자 하는 대상을 서술적으로 표현할 수 있죠. 앞에서 언급했던 패키지들도 이런 생각을 토대로 만들어졌습니다. React를 렌더링에 사용할 수 있는 영역을 한정해서 생각하고 싶지 않습니다. 한 예로 DOM diff 비교를 이용하면 브라우저 환경에서 React를 아주 유용하고 빠르게 이용할 수 있습니다. 하지만 DOM이 상태를 기반으로 하는 명령형 API를 가지고 있지 않았다면 이런 비교 방식은 전혀 필요 없었겠죠. 이것을 보다 명확하게 하여 더 다양한 환경에서 React를 이용해 렌더링할 수 있도록 react 패키지를 react와 react-dom으로 분리하기로 했습니다.

react 패키지는 React.createElement, React.createClass, React.Component, React.PropTypes, React.Children을 포함하며, 엘리먼트와 컴포넌트 클래스를 사용하는 데 필요한 API를 제공합니다. 이 API는 동형<sup>Isomorphic<sup>36</sup></sup>(또는 보편적 Universal<sup>37</sup>) 자바스크립트 컴포넌트를 개발하는 데 필요하기 때문입니다.

react-dom 패키지는 ReactDOM.render, ReactDOM.unmountComponentAtNode, ReactDOM.findDOMNode를 포함하며, react-dom/server는 서버 측 렌더링에 필요한 ReactDOMServer.renderToString과 ReactDOMServer.renderToStaticMarkup API를 지원합니다.

---

32 <https://github.com/facebook/react-native>

33 <https://github.com/reactjs/react-art>

34 <https://github.com/Flipboard/react-canvas>

35 <https://github.com/lzzimach/react-three>

36 <http://nerds.airbnb.com/isomorphic-javascript-future-web-apps/>

37 <https://medium.com/@mjackson/universal-javascript-4761051b7ae9>

---

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

var MyComponent = React.createClass({
  render: function() {
    return <div>Hello World</div>;
  }
});

ReactDOM.render(<MyComponent />, node);
```

---

대부분 컴포넌트가 가볍고, 어떠한 렌더링 로직도 가지고 있지 않은 react 패키지만을 의존할 것으로 예상합니다. 따라서 브라우저에서 React를 사용하기 위해서는 react-dom이 필요합니다. 이와 비슷한 DOM을 기반으로 하는 컴포넌트인 awesome-faster-react-dom(훨씬 빠른 react-dom)을 사용할 수도 있습니다. 컴포넌트를 정의하는 부분과 렌더링을 하는 부분을 분리함으로써 이런 변화를 만들었습니다.

더 중요한 것은, 이제 React와 React Native에서 공유할 수 있는 컴포넌트를 만들 수 있는 길이 열렸다는 점입니다. 아직은 공유 컴포넌트를 만들기가 쉽지 않습니다만 앞으로 보다 쉽게 만들 수 있게 개선해서 웹사이트와 네이티브 앱이 서로 React 코드를 공유할 수 있게 만들 생각입니다.

애드온<sup>addon</sup>도 별도의 패키지로 분리했습니다. react-addons-clone-with-props, react-addons-create-fragment, react-addons-css-transition-group, react-addons-linked-state-mixin, react-addons-pure-render-mixin, react-addons-shallow-compare, react-addons-transition-group, react-addons-update를 분리했으며, react-dom의 ReactDOM.unstable\_batchedUpdates도 분리했습니다.

버전 간의 문제를 막기 위해서는 같은 버전의 react와 react-dom을 사용하길

바랍니다. 나중에는 이런 의존성을 제거할 예정입니다. 이번 버전까지는 기존 메서드를 계속 지원하는 대신 경고 문구를 노출하며, 0.15 버전에서는 완전히 제거 할 예정입니다.

## DOM 노드의 refs

이번 버전의 또 다른 큰 변화는 DOM 컴포넌트의 refs가 DOM 노드를 참조하도록 변경한 부분입니다. 우리는 많은 사람이 DOM 컴포넌트에서 ref를 사용하는 이유가 대체로 `this.refs.giraffe.getDOMNode()`로 DOM 노드를 가져오기 위해 서란 사실을 알았습니다. 이번 버전부터 `this.refs.giraffe`는 실제 DOM 노드를 참조합니다.

커스텀 컴포넌트 클래스의 refs는 변경없이 이전과 똑같이 동작합니다.

---

```
var Zoo = React.createClass({
  render: function() {
    return (
      <div>
        Giraffe's name: <input ref="giraffe" />
      </div>
    );
  },
  showName: function() {
    // Previously:
    // var input = this.refs.giraffe.getDOMNode();
    var input = this.refs.giraffe;

    alert(input.value);
  }
});
```

---

이 변경 내용은 상위 컴포넌트를 DOM 노드로 `ReactDOM.render`에 전달해도 반환 하는 결과값에 똑같이 적용됩니다. refs의 경우와 마찬가지로 `<MyFancyMenu>`

나 <MyContextProvider> 같은 커스텀 컴포넌트는 예외입니다.

이 변경과 함께 `component.getDOMNode()`를 `ReactDOM.findDOMNode(component)`로 변경했습니다. `findDOMNode` 메서드는 인자로 전달하는 컴포넌트로 인해 렌더링이 발생한 DOM 노드를 탐색하는데, DOM 노드를 넘기면 인자를 그대로 반환하기 때문에 DOM 컴포넌트에 사용해도 안전합니다. 우리는 이 메서드를 지난번 릴리스에서 조용히 추가했는데, 이제는 `.getDOMNode()`를 완전히 지원중단하기로 결정했습니다. 기존의 코드를 `ReactDOM.findDOMNode`로 쉽게 변경할 수 있습니다. 또한, 변경 작업을 도와줄 자동화 스크립트<sup>38</sup>를 제공합니다. 앞의 예제처럼 DOM 컴포넌트에 `ref`가 있다면 `findDOMNode`를 사용할 필요가 없다는 점을 주의하세요.

저희처럼 여러분에게도 이번 릴리스가 흥미로웠기를 바랍니다. 언제든 의견 주세요.

## 18.13 React v.0.14 RC<sup>39</sup>

React 0.14의 첫 번째 배포 예정 판을 공개합니다. 지난 7월에 베타를 출시하면서 변경사항을 소개한 적이 있습니다. 그 사이 좀 더 안정화를 하여 이번에 배포 예정 판을 공개합니다. 사용 중에 발생한 이슈는 GitHub 저장소에 등록해주세요.

### 설치

`react`를 `npm`으로 설치하고 `browserify`나 `webpack` 같은 도구를 이용해서 패키지로 빌드하는 것을 추천합니다.

---

```
npm install --save react@0.14.0-rc1
npm install --save react-dom@0.14.0-rc1
```

---

<sup>38</sup> <https://www.npmjs.com/package/react-codemod>

<sup>39</sup> <http://facebook.github.io/react/blog/2015/09/10/react-v0.14-rc1.html>

React는 기본적으로 development 모드일 때 추가 검사를 하며 경고 메시지를 출력합니다. 애플리케이션을 배포할 때 NODE\_ENV 환경 변수를 production으로 변경한 후 빌드하면 React는 이런 경고 메시지를 노출하지 않습니다. 따라서 React를 더욱 빠르게 빌드할 수 있습니다. bower에서도 react, react-dom으로 찾을 수 있습니다.

## React와 React DOM의 분리

react-native, react-art, react-canvas, react-three 같은 패키지를 살펴보면, React의 아름다움과 정수는 브라우저나 DOM에 국한되지 않는다는 것이 확실합니다.

컴포넌트와 엘리먼트에 대한 단순한 발상이 React의 본질이라고 생각합니다. 렌더링하고자 하는 대상을 서술적으로 표현할 수 있죠. 앞에서 언급했던 패키지들도 이런 생각을 토대로 만들어졌습니다. React를 렌더링에 사용할 수 있는 영역을 한정해서 생각하고 싶지 않습니다. 한 예로 DOM diff 비교를 이용하면 브라우저 환경에서 React를 아주 유용하고 빠르게 이용할 수 있습니다. 하지만 DOM이 상태를 기반으로 하는 명령형 API를 가지고 있지 않았다면 이런 비교 방식은 전혀 필요 없었겠죠.

이것을 보다 명확하게 하여 더 다양한 환경에서 React를 이용해 렌더링할 수 있도록 react 패키지를 react와 react-dom으로 분리하기로 했습니다.

react 패키지는 React.createElement, React.createClass, React.Component, React.PropTypes, React.Children을 포함하며, 엘리먼트와 컴포넌트 클래스를 사용하는 데 필요한 API를 제공합니다. 이 API는 동형<sup>isomorphic</sup> (또는 보편적<sup>Universal</sup>) 자바스크립트 컴포넌트를 개발하는 데 필요하기 때문입니다.

react-dom 패키지는 ReactDOM.render, ReactDOM.unmountComponentAtNode, ReactDOM.findDOMNode를 포함하며,

`react-dom/server`는 서버 측 렌더링에 필요한 `ReactDOMServer`.  
`renderToString`과 `ReactDOMServer.renderToStaticMarkup` API를 지원합니다.

---

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

var MyComponent = React.createClass({
  render: function() {
    return <div>Hello World</div>;
  }
});

ReactDOM.render(<MyComponent />, node);
```

---

대부분 컴포넌트가 가볍고, 어떠한 렌더링 로직도 가지고 있지 않은 `react` 패키지만을 의존할 것으로 예상합니다. 따라서 브라우저에서 `React`를 사용하기 위해서는 `react-dom`이 필요합니다. 이와 비슷한 DOM을 기반으로 하는 컴포넌트인 `awesome-faster-react-dom`(훨씬 빠른 `react-dom`)을 사용할 수도 있습니다. 컴포넌트를 정의하는 부분과 렌더링을 하는 부분을 분리함으로써 이런 변화를 만들었습니다.

더 중요한 것은, 이제 `React`와 `React Native`에서 공유할 수 있는 컴포넌트를 만들 수 있는 길이 열렸다는 점입니다. 아직은 공유 컴포넌트를 만들기가 쉽지 않습니다만 앞으로 더욱 쉽게 만들 수 있게 개선해서 웹사이트와 네이티브 앱이 서로 `React` 코드를 공유할 수 있게 만들 생각입니다.

애드온도 별도의 패키지로 분리했습니다. `react-addons-clone-with-props`, `react-addons-create-fragment`, `react-addons-css-transition-group`, `react-addons-linked-state-mixin`, `react-addons-pure-render-mixin`, `react-addons-shallow-compare`, `react-addons-transition-group`,

`react-addons-update`를 분리했으며, `react-dom`의 `ReactDOM.unstable_batchedUpdates`도 분리했습니다.

버전 간의 문제를 막기 위해서는 같은 버전의 `react`와 `react-dom`을 사용하길 바랍니다. 나중에는 이런 의존성을 제거할 예정입니다. 이번 버전까지는 기존 메서드를 계속 지원하는 대신 경고 문구를 노출하며, 0.15 버전에서는 완전히 제거할 예정입니다.

### DOM 노드의 refs

이번 버전의 또 다른 큰 변화는 DOM 컴포넌트의 `refs`가 DOM 노드를 참조하도록 변경한 부분입니다. 우리는 많은 사람이 DOM 컴포넌트에서 `ref`를 사용하는 이유가 대체로 `this.refs.giraffe.getDOMNode()`로 DOM 노드를 가져오기 위해 서란 사실을 알았습니다. 이번 버전부터 `this.refs.giraffe`는 실제 DOM 노드를 참조합니다.

커스텀 컴포넌트 클래스의 `refs`는 변경 없이 이전과 똑같이 동작합니다.

---

```
var Zoo = React.createClass({
  render: function() {
    return (
      <div>
        Giraffe's name: <input ref="giraffe" />
      </div>
    );
  },
  showName: function() {
    // Previously:
    // var input = this.refs.giraffe.getDOMNode();
    var input = this.refs.giraffe;

    alert(input.value);
  }
});
```

---

이 변경 내용은 상위 컴포넌트를 DOM 노드로 ReactDOM.render에 전달해도 반환하는 결과값에 똑같이 적용됩니다. refs의 경우와 마찬가지로 <MyFancyMenu>나 <MyContextProvider> 같은 커스텀 컴포넌트는 예외입니다.

이 변경과 함께 component.getDOMNode()를 ReactDOM.findDOMNode(component)로 변경했습니다. findDOMNode 메서드는 인자로 전달하는 컴포넌트로 인해 렌더링이 발생한 DOM 노드를 탐색하는데, DOM 노드를 넘기면 인자를 그대로 반환하기 때문에 DOM 컴포넌트에 사용해도 안전합니다. 우리는 이 메서드를 지난번 릴리스에서 조용히 추가했는데, 이제는 .getDOMNode()를 완전히 지원을 중단하기로 결정했습니다. 기존의 코드를 ReactDOM.findDOMNode로 쉽게 변경할 수 있습니다. 또한, 변경 작업을 도와줄 자동화 스크립트를 제공합니다. 앞의 예제처럼 DOM 컴포넌트에 ref가 있다면 findDOMNode를 사용할 필요가 없다는 점을 주의하세요.

### 무상태 함수 컴포넌트(Stateless function components)

보통의 React 코드를 보면 작성하는 컴포넌트 대부분이 상태를 갖지 않으며, 다른 컴포넌트로 구성되어 있습니다. 이제 새롭고, 이전보다 단순한 문법을 이용할 수 있습니다. 다음 코드는 props를 인자로 받아서 렌더링하길 원하는 엘리먼트를 반환합니다.

---

```
// ES2015 (ES6) arrow 함수를 사용한다:  
var Aquarium = (props) => {  
  var fish = getFish(props.species);  
  return <Tank>{fish}</Tank>;  
};
```

```
// 보다 단순한 방법도 가능하다.  
var Aquarium = ({species}) => (  
  <Tank>  
    {getFish(species)}  
  </Tank>
```

```
);  
// 사용할 때는 <Aquarium species="rainbowfish" />라고 작성한다.
```

---

이 패턴을 이용하면 애플리케이션의 많은 부분을 차지하는 단순한 컴포넌트를 더 쉽게 생성할 수 있습니다. 나중에는 불필요한 검사 단계와 메모리 할당을 제거함으로써 성능을 최적화할 수도 있습니다.

### react-tools 지원 중단

react-tools 패키지와 브라우저용 JSXTransformer.js 파일에 대한 지원을 중단했습니다. 0.13.3 버전의 파일을 사용할 수는 있지만 더는 지원하지 않으며, React와 JSX를 기본적으로 지원하는 Babel을 사용하기를 권장합니다.

### 컴파일러 최적화

React는 Babel 5.8.23 이후부터 두 가지 컴파일러 최적화 옵션을 지원합니다. 이 최적화는 코드를 압축하는 production 단계에서만 이용해야 합니다. 실행 성능을 개선하지만, 알 수 없는 경고 메시지를 노출하며 propTypes처럼 development 모드에서 확인할 수 있는 검사 단계를 생략하기 때문입니다.

두 가지 최적화 옵션은 다음과 같습니다.

- 인라인 React 엘리먼트: optimisation.react.inlineElements은 JSX 엘리먼트를 React.createElement 호출 방식 대신 {type: 'div', props: ...} 같은 객체 리터럴로 변경합니다.
- React 엘리먼트를 위한 상수 호이스팅 : optimisation.react.constantElements는 하위 트리에 있는 정적인 엘리먼트 생성을 최상위로 끌어올립니다. 이렇게 해서 React.createElement 호출과 이로 인한 메모리 할당을 줄일 수 있습니다. 더 중요한 것은, 하위 트리에 변경이 없다는 사실을 React에 전달함으로써 조정(reconciling) 과정을 생략할 수 있다는 점입니다.

### 주의해야 할 변경 사항

이번 버전에서도 주의해야 할 변경사항이 있습니다. 많은 변화가 있을 때는 릴리

스를 하면서 이런 부분을 안내하여 코드를 업데이트할 수 있는 시간을 가질 수 있도록 하고 있습니다. Facebook의 코드 베이스에는 15,000개 이상의 React 컴포넌트가 존재합니다. 따라서 React 팀은 변경으로 인해 발생할 수 있는 고통을 최소화하기 위해 최대한 노력하고 있습니다.

다음 세 가지 주의사항은 0.13 버전에서 경고 메시지를 노출했고, 자신의 코드에서 경고 메시지가 발생하지 않는다면 수정하지 않아도 좋습니다.

props는 불변이며, 컴포넌트 엘리먼트를 생성한 후 조작하는 것은 이제 불가능합니다. 대부분의 경우 `React.cloneElement`<sup>40</sup>를 대신 사용할 수 있습니다. 이 변경 덕분에 컴포넌트를 이해하기가 더 쉬워졌으며, 앞에서 설명했던 컴파일러 최적화를 할 수 있었습니다.

React 자체 노드로 순수 객체는 더 이상 사용할 수 없습니다. 대신 배열을 사용하길 바랍니다. 배열을 반환하는 `createFragment`<sup>41</sup>를 사용하세요.

- Add-Ons : `classSet`을 제거하였습니다. 대신 `classnames`<sup>42</sup>를 사용하세요.

다음 두 가지 변경은 0.13 버전에서 경고를 노출하지 않았지만 쉽게 찾아서 제거할 수 있습니다.

- `React.initializeTouchEvents`는 더는 필요 없으며 완전히 제거하였습니다. 터치 이벤트를 자동으로 적용합니다.
- Add-Ons : 앞에서 설명한 DOM 노드의 `refs`에 대한 변경과 관련하여 `TestUtils.findAllInRenderedTree`, 그리고 이와 관련 있는 메서드는 더는 커스텀 컴포넌트가 아닌 DOM 컴포넌트를 가져올 수 없습니다.

## 지원 중단

다음 항목은 지원을 중단하며 이번 버전까지는 경고 메시지를 노출하고, 다음 버

---

<sup>40</sup> <http://facebook.github.io/react/docs/top-level-api.html#react.cloneelement>

<sup>41</sup> <http://facebook.github.io/react/docs/create-fragment.html>

<sup>42</sup> <https://github.com/JedWatson/classnames>

전에서 제거할 예정입니다.

앞에서 설명한 DOM 노드의 refs에 대한 변경과 관련하여 `this.getDOMNode()`를 이제 사용할 수 없습니다. `ReactDOM.findDOMNode(this)`를 사용하세요. 추가로 이제부터는 `findDOMNode`를 사용해야 할 일이 별로 없습니다. DOM 노드의 refs 항목을 살펴보세요. 작업하고 있는 코드의 양이 많다면 자동화 스크립트를 추천합니다.

`setProps`와 `replaceProps`는 더는 사용할 수 없습니다. 대신에 새로운 최상위에서 새로운 `props`를 인자로 전달하여 `ReactDOM.render`를 호출하세요.

ES6 컴포넌트 클래스는 `React.Component`를 상속해야 무상태 함수 클래스를 이용할 수 있습니다. ES3 모듈 패턴은 계속 사용할 수 있습니다.

더 이상 `style` 객체를 렌더링 중간에 재사용하거나 조작할 수 없습니다. `props`를 변경할 수 없게 만든 것과 같은 의도입니다.

- Add-Ons : `cloneWithProps`를 이제 사용할 수 없습니다. 대신 `React.cloneElement`를 사용하세요. 이 API는 `cloneWithProps`와 달리 `className`이나 `style`을 자동으로 병합하지 않습니다. 필요에 따라 이 항목을 병합합니다.
- Add-Ons : 신뢰도를 높이기 위해서 `CSSTransitionGroup`은 이제 트랜지션 이벤트를 리스닝하지 않습니다. 대신에 `props`를 이용해 `transitionEnterTimeout={500}`과 같은 형식으로 트랜지션 시간을 직접 설정해야 합니다.

## 주요 개선 사항

`React.Children.toArray`를 추가하였습니다. 이 API는 중첩된 자식 객체를 전달받아, 각 자식에 할당한 키를 원소로 하는 배열을 반환합니다. 이 메서드를 이용하면 `render` 메서드에서 자식 콜렉션을 쉽게 처리할 수 있으며, 특히 `this.props.children`을 전달하기 전에 순서를 변경하거나 잘라야 할 때 유용합니다. 이제 `React.Children.map`가 배열을 반환한다는 점도 참고하세요.

`console.warn` 대신 `console.error`를 사용합니다. 브라우저 콘솔에서 전체 스택 트레이스를 확인할 수 있습니다. 릴리스 후에 문제가 될 수 있거나 예상치 못한 결과를 발생시킬 수 있는 코드가 있을 때 경고 메시지를 출력하므로 반드시 수정해야 합니다.

이전에는 신뢰할 수 없는 객체를 React 자식으로 사용하면 XSS 보안 취약점 문제<sup>43</sup>가 발생했습니다. 이 문제를 해결하려면 애플리케이션 레이어에서 입력값을 적절히 검증하여 애플리케이션 코드로 신뢰할 수 없는 객체를 전달하지 않게 해야 합니다. 보호 레이어를 추가하기 위해 React는 ES2015 (ES6) `Symbol`<sup>44</sup>을 지원하는 브라우저에서 엘리먼트에 태그를 적용<sup>45</sup>합니다. 이로써 React가 신뢰할 수 없는 JSON을 유효한 객체로 판단하는 것을 방지합니다. 구형 브라우저에서 이런 추가 보안 대책이 필요하다면 `Symbol`을 사용할 수 있는 폴리필(polyfill)을 사용하세요. Babel은 `Symbol` 폴리필<sup>46</sup>을 제공합니다.

이제 React DOM은 최대한 XHTML에 호환되는 마크업을 만듭니다. React DOM은 다음과 같은 HTML 표준 또는 비표준 속성을 지원합니다.

- 표준 속성 : `capture`, `challenge`, `inputMode`, `is`, `keyParams`, `keyType`, `minLength`, `summary`, `wrap`
- 비표준 속성 : `autoSave`, `results`, `security`

React DOM은 다음과 같은 SVG 속성을 지원합니다. 네임스페이스 속성으로 렌더링 됩니다.

- `xlinkActuate`, `xlinkArcrole`, `xlinkHref`, `xlinkRole`, `xlinkShow`, `xlinkTitle`, `xlinkType`, `xmlBase`, `xmlLang`, `xmlSpace`

---

<sup>43</sup> <http://danlec.com/blog/xss-via-a-spoofed-react-element>

<sup>44</sup> <http://www.2ality.com/2014/12/es6-symbols.html>

<sup>45</sup> <https://github.com/facebook/react/pull/4832>

<sup>46</sup> <http://babeljs.io/docs/usage/polyfill/>

image SVG 태그를 지원합니다. React DOM을 사용할 때 커스텀 엘리먼트의 태그명에 `-`를 넣거나 `is="..."`와 같은 임의의 속성을 사용할 수 있습니다.

React DOM은 audio와 video에서 다음 미디어 이벤트를 지원합니다.

- `onAbort`, `onCanPlay`, `onCanPlayThrough`, `onDurationChange`, `onEmptied`, `onEncrypted`, `onEnded`, `onError`, `onLoadedData`, `onLoadedMetadata`, `onLoadStart`, `onPause`, `onPlay`, `onPlaying`, `onProgress`, `onRateChange`, `onSeeked`, `onSeeking`, `onStalled`, `onSuspend`, `onTimeUpdate`, `onVolumeChange`, `onWaiting`

이외에도 성능을 많이 개선했습니다. 더욱 자세한 맥락을 경고 메시지에 추가했습니다.

- Add-Ons : ES6 클래스의 `PureRenderMixin`을 대체하기 위해서 `shallowCompare`가 추가되었습니다.
- Add-Ons : `CSSTransitionGroup`에 `-enter-active` 같은 이름을 넣는 대신 별도의 클래스 명을 사용할 수 있습니다.

### 새로운 경고 메시지

개발자가 HTML 엘리먼트를 잘못 중첩한 경우, 이제 ReactDOM은 경고 메시지를 출력합니다. 업데이트 동안에 발생하는 당황스러운 오류를 피할 수 있습니다.

`ReactDOM.render`에 컨테이너로 `document.body`를 직접 전달하면 DOM을 조작하는 브라우저 확장 도구가 오작동할 수 있다는 경고를 노출합니다.

여러 개의 React 인스턴스를 사용할 수 없습니다. 따라서 이런 문제가 있으면 경고를 노출하여 문제를 사전에 방지합니다.

### 주요 버그 수정

모바일 브라우저, 특히 모바일 사파리에서 React DOM이 클릭 이벤트를 처리하는 방식을 개선했습니다.

올바른 네임 스페이스를 갖는 SVG 엘리먼트를 생성하는 경우가 더 많아졌습니다.

React DOM은 이제, 여러 개의 텍스트 자식 노드를 갖는 <option> 엘리먼트를 제대로 렌더링하며, 서버에서도 <select>의 선택한 옵션을 정확하게 표현하여 렌더링합니다.

브라우저 확장 도구를 사용하는 경우는 물론, 서로 다른 두 개의 React가 같은 도큐먼트에 노드를 추가하는 경우에도 React DOM은 이벤트를 처리할 때 최대한 예외를 던지지 않습니다.

React DOM을 사용할 때 HTML 태그명을 대문자로 작성할 수 있습니다(예: `React.createElement('DIV')`). 하지만 여전히 JSX 컨벤션과의 통일성 유지하기 위해 소문자로 작성할 것을 권장합니다. 소문자 이름은 내장 컴포넌트를 참조하고, 대문자 이름은 커스텀 컴포넌트를 참조합니다.

React DOM은 다음 CSS 속성에 단위를 적용하지 않기 때문에 값은 부여할 때 "px"를 붙이지 않습니다.

- `animationIterationCount`, `boxOrdinalGroup`, `flexOrder`, `tabSize`, `stopOpacity`
- Add-Ons : 테스트 도구의 `Simulate.mouseEnter`와 `Simulate.mouseLeave`를 사용할 수 있습니다.
- Add-Ons : `ReactTransitionGroup`은 이제 여러 개의 노드를 동시에 제거할 때도 제대로 동작합니다.