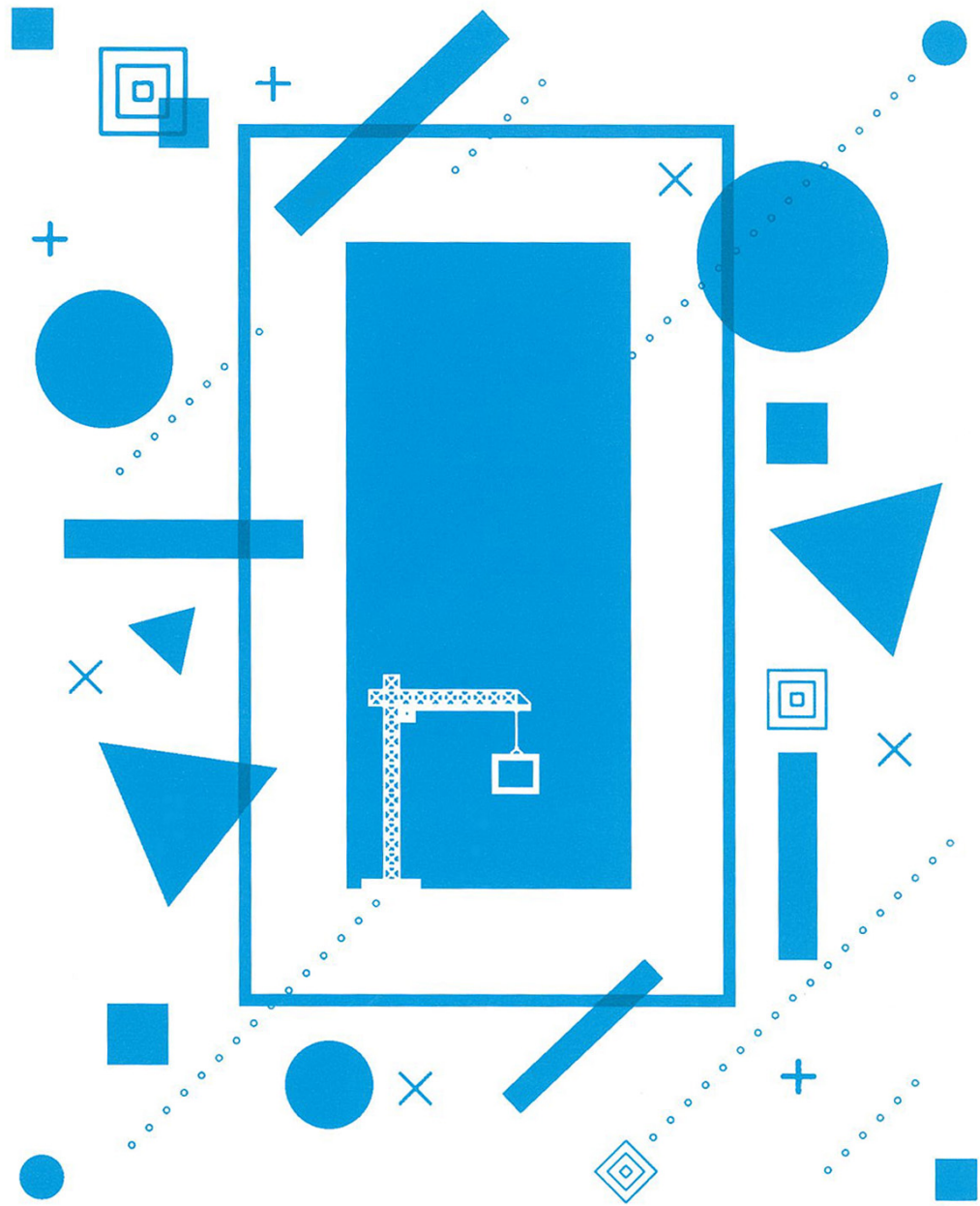


실전 리액트 프로그래밍

리액트 훅부터 Next.js 까지

이재승 지음

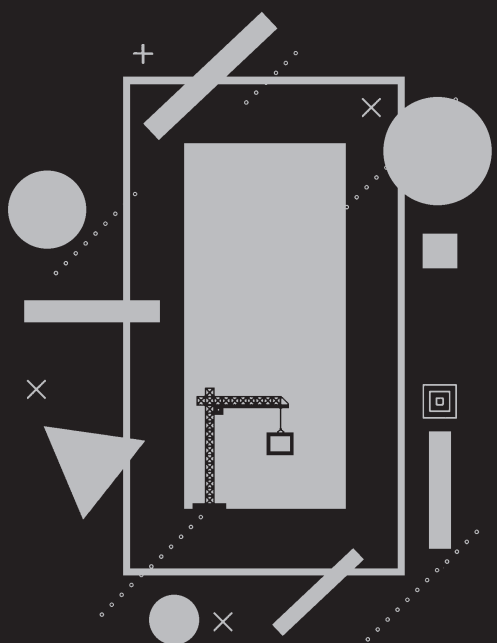
프로그래밍인사이트

실전 리액트 프로그래밍

실전 리액트 프로그래밍

초판 PDF 1.0 2019년 7월 5일 **지은이** 이재승 **펴낸이** 한기성 **펴낸곳** 인사이트 **편집** 문선미 **관리** 박미경 **등록번호** 제2002-000049호 **등록일자** 2002년 2월 19일 **주소** 서울시 마포구 연남로5길 19-5 **전화** 02-322-5143 **팩스** 02-3143-5579 **블로그** http://blog.insightbook.co.kr **이메일** insight@insightbook.co.kr **ISBN** 978-89-6626-249-6

프로그래밍 인사이트



실전 리액트 프로그래밍

리액트 훅부터 Next.js까지

이재승 지음

인사이트 insight

차례

지은이의 글

요즘 대부분의 웹 개발자는 쏟아지는 신기술에 빠져 헤어나오지 못하고 있다. 웹에서의 기술은 10년 전과는 비교도 안 될 정도로 발전이 빨라졌다. 영원할 줄 알았던 제이쿼리(jquery)의 인기가 식었고, 한동안 높은 인기를 누리던 앵귤러(angular)는 점점 외면 받고 있다. 그리고 그 자리를 리액트와 vue.js가 대신하고 있다.

기술의 교체 주기가 짧은 만큼 프레임워크나 라이브러리의 사용법보다는 핵심적인 원리를 이해하는 것이 중요하다. 그래야 새로운 기술이 나오더라도 빠르게 습득할 수 있고, 더 나아가 비판적 사고도 가능하다. vue.js에서는 리액트의 가상 돔 개념을 사용했고, 이후 새로 등장할 프레임워크에서도 비슷한 개념을 적용할 것으로 예상된다. 따라서 리액트의 가상 돔 개념을 이해하면, 가상 돔을 적용한 다른 프레임워크의 동작도 쉽게 이해할 수 있다.

필자는 이 책에서 리액트의 사용법뿐만 아니라, 그렇게 사용하는 이유를 설명하기 위해 노력했다. 리액트 공식 문서를 보면서 누구나 간단한 웹 애플리케이션을 만들 수 있다. 하지만 실전에서 사용하다 보면 다양한 문제를 만난다. 필자의 경우 리액트로 개발하면서 가장 힘들었던 것이 바벨과 웹팩의 설정이었다. 바벨과 웹팩을 제대로 이해하지 못한 상태에서 인터넷에 떠도는 설정을 그대로 가져왔었는데, 한동안 잘 동작하는 듯해도 조금만 수정하려고 하면 에러가 났다. 이 경험을 바탕으로 바벨과 웹팩을 설명하는 장을 만들었다. 이 책의 내용을 모두 이해하면, create-react-app과 같은 도구를 사용하지 않고도 혼자서 리액트 개발 환경을 구축할 수 있을 것이다.

필자는 웹 세상에 발을 들여놓기 전까지 주로 C++, 자바와 같은 정적 타입 언어를 사용했다. 그래서인지 자바스크립트가 편하면서도 동적 타입 언어라 오히려 불편하기도 했다. 동적 타입 언어는 IDE에서 제공해주는 편의 기능이 제한적이다. 따라서 간단한 프로그램을 작성하는 게 아니라면 정적 타입 언어를 사용

하는 것이 더 좋다고 생각한다. 다행히 자바스크립트에 정적 타입 기능을 추가해 주는 다양한 언어가 존재한다. 필자의 경우 그중에서도 커뮤니티의 크기가 가장 큰 타입스크립트를 현업에서 사용하고 있다. 타입스크립트에 매우 만족하고 있으며, 앞으로 새로운 프로젝트를 만들 때도 계속 이용할 생각이다. 다른 분들도 타입스크립트의 장점을 알았으면 하는 마음에 타입스크립트를 설명하는 장을 만들었다.

리액트를 처음 접했을 때가 생각난다. 회사에서 데이터 시스템을 만들면서 백엔드 작업을 주로 하던 터라 지표를 보여줄 웹사이트가 필요했다. 리액트가 뜨고 있다는 소식을 들었기에 웹사이트를 리액트로 만들기 시작했다. HTML 파일을 거의 건드리지 않고 자바스크립트만으로 코딩하는 재미가 쏠쏠했다. 이후에도 여러 개의 백오피스를 리액트로 만들면서 자신감이 붙었고, 급기야 스프링과 JSP로 동작하던 카카오페이지 웹을 리액트로 포팅했다. 그 과정에서 리액트의 버그를 발견했고, 필자의 코드가 리액트에 포함되는 짜릿한 경험도 했다. 물론 그 과정이 달콤하기만 한 건 아니었다. 값진 시행착오를 겪기도 했지만, 알고 보면 아무것도 아닌 것에 많은 시간을 낭비하기도 했다. 필자는 리액트를 공부하는 개발자들이 되도록 유용한 경험을 하길 바라며 이 책을 썼다. 모쪼록 이 책이 리액트를 공부하는 국내 개발자들의 학습 시간을 조금이라도 줄여 주길 바란다.

대상 독자

이 책은 리액트를 사용해 봤지만 아직 이해가 부족한 사람을 대상으로 한다. create-react-app만으로 프로젝트를 생성해 본 사람은 이 책을 읽은 후 리액트 개발 환경을 직접 구축할 수 있게 된다. 서버사이드 렌더링을 해 본 적이 없는 사람은 이 책을 읽은 후 서버사이드 렌더링의 주요 개념을 이해하고 응용할 수 있게 된다. 정적 타입으로 웹 개발을 해 본 적이 없는 사람은 이 책을 읽은 후 타입스크립트를 이용해서 리액트 코드를 작성할 수 있게 된다.

책의 구성

이 책은 리액트의 기본 개념을 다지고, 실전에서 쓰이는 활용법을 익힐 수 있게 구성되어 있다. 책의 후반부에는 타입스크립트로 리액트 코드를 작성하는 방법과 다가올 리액트의 변화에 대해 알아본다.

리액트의 개념, 기본기 다지기

1~3장에서는 리액트로 코드를 작성하기 위한 기본기를 다져본다. 1장에서는 리액트 프로젝트를 구축하는 방법을 알아본다. create-react-app을 사용해서 구축하는 방법과 create-react-app의 도움 없이 구축하는 방법을 알아본다. 2장에서는 자바스크립트 최신 문법을 알아본다. 리액트에서 자주 사용되는 문법 위주로 구성되어 있다. 3장에서는 리액트의 주요 개념을 알아본다.

리액트의 활용, 실전에서 적용해 보기

4~9장에서는 리액트를 실전에서 활용하기 위한 다양한 방법을 알아본다. 4장에서는 컴포넌트 코드를 작성할 때 도움이 되는 고급 활용법을 알아본다. 특히 컴포넌트의 공통 기능을 분리하는 방법인 고차 컴포넌트와 렌더 속성값 패턴을 이해하게 된다. 5장에서는 리액트 훅의 기본 개념을 알아보고 클래스형 컴포넌트의 각 기능을 어떻게 훅으로 구현할 수 있는지 알아본다. 6장에서는 리덕스의 기본 개념과 활용법을 알아본다. 7장에서는 바벨과 웹팩의 기본 개념부터 고급 기능까지 알아본다. 8장에서는 서버사이드 렌더링의 개념과 Next.js로 프로젝트를 구축하는 방법을 알아본다. 9장에서는 정적 타입 언어의 한 종류인 타입스크립트를 이용해서 리액트 코드를 작성하는 방법을 알아본다.

리액트의 미래, 다가올 변화를 미리 확인하기

10장에서는 앞으로 리액트에 추가될 것으로 예상되는 비동기 렌더링에 대해 알아본다.

학습 환경

- 이 책의 실습 과정에서 나오는 모든 코드는 아래 경로에서 확인할 수 있다.
  https://github.com/landvibe/book-react

- 설명을 쉽게 하기 위해 실습 과정에서의 패키지 설치는 `npm install 패키지명`으로 통일한다. 실제로는 배포 환경(dependencies)과 개발 환경(devDependencies)의 패키지를 구분해서 관리하는 게 좋다.

- npm을 사용하기 위해서는 노드(node.js)가 설치되어 있어야 한다. 노드는 다음 경로를 통해서 설치할 수 있다.

  https://nodejs.org

- 이 책의 모든 코드는 프리티어(prettier)를 사용해서 포맷팅했다. 프리티어를 사용하면 코드 포맷팅에 들어가는 시간과 노력을 아낄 수 있다. 생산성을 높이는 데 큰 도움이 되므로 사용하기를 적극 추천한다.

  https://prettier.io/

참고 링크

- 자바스크립트 문법을 공부할 때 참고하면 좋은 사이트다. 세부적인 내용까지 꼼꼼하게 설명하기 때문에 자바스크립트 언어를 제대로 공부하고 싶은 독자에게 추천한다.

  http://2ality.com/

  https://javascript.info

- 자바스크립트로 함수형 프로그래밍을 설명하는 동영상 강의다. 2장에서 지연 평가를 설명하는 부분에서 아래 동영상을 참고했다. 강사 유인동 님은 인사이트 출판사에서 《함수형 자바스크립트 프로그래밍》을 출간하기도 했다.

  https://programmers.co.kr/learn/courses/7637

- 프레젠테이션, 컨테이너 컴포넌트를 설명하는 댄 아브라모프의 유명한 블로그 글이다.

  https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0

- 고차 컴포넌트를 설명한 블로그 글이다. 고차 컴포넌트의 여러 가지 활용법을 설명한다.

  https://medium.com/@franleplant/react-higher-order-components-in-depth-cf9032ee6c3e

- 렌더 속성값 패턴을 설명한 블로그 글이다. 렌더 속성값 패턴을 믹스인(mixin), 고차 컴포넌트와 비교하면서 설명한다.

  https://cdb.reacttraining.com/use-a-render-prop-50de598f11ce

- 리덕스를 만든 댄 아브라모프의 유명한 블로그 글이다. 리덕스를 꼭 사용해야 하는지, 언제 사용하면 좋은지 궁금한 독자에게 추천한다.
  https://medium.com/@dan_abramov/you-might-not-need-redux-be46360cf367

- 나무 흔들기(7.4.1 참고)의 사용법과 주의할 점을 설명하는 글이다.
  https://developers.google.com/web/fundamentals/performance/optimizing-javascript/tree-shaking/

- 서버사이드 렌더링을 스트림과 함께 사용하는 방법을 설명하는 글이다. 스트림을 사용하면 서버사이드 렌더링을 좀 더 효율적으로 할 수 있다.
  https://zeit.co/blog/streaming-server-rendering-at-spectrum

- 댄 아브라모프가 2018년 말부터 시작한 블로그다. 리액트를 자세히 알고 싶은 독자에게 추천한다.
  https://overreacted.io/

책을 써 보자고 제안해 주신 문선미 편집자님께 감사드린다. 처음부터 끝까지 꼼꼼하게 챙겨주신 덕분에 책을 완성할 수 있었다. 책을 쓸 수 있도록 배려해 준 사랑하는 성희와 도현이에게도 감사하다. 아빠는 도대체 언제 책을 다 쓰는 거냐고, 불평하며 혼자 놀던 도현이에게 항상 미안했다.

1장

Practical React Programming

리액트 프로젝트 시작하기

이번 장에서는 리액트가 무엇인지 살펴본 후, 간단한 리액트 애플리케이션을 개발하는 전체 과정을 속성으로 체험해 본다. 구체적으로는 리액트 개발 환경을 구축하여 간단한 단일 페이지 애플리케이션까지 제작해 볼 것이다.

1.1 리액트란 무엇인가

리액트는 페이스북에서 개발하고 관리하는 UI 라이브러리다. 앵귤러(angular)가 웹 애플리케이션 개발에 필요한 다수의 기능을 제공하는 것과는 대조적으로, 리액트는 UI 기능만 제공한다. 따라서 전역 상태 관리, 라우팅, 빌드 시스템 등을 각 개발자가 직접 구축해야 한다. 전반적인 시스템을 직접 구축할 수 있으니 각자의 환경에 맞게 최적화할 수 있다. 반대로 신경 쓸 것이 많기 때문에 초심자에게는 높은 진입 장벽이 되기도 한다.

리액트 팀에서는 리액트의 진입 장벽을 낮추기 위해 create-react-app을 만들었다. create-react-app을 이용하면 리액트를 처음 사용하는 사람도 하나의 명령어로 리액트 개발 환경을 구축할 수 있다. create-react-app에 대한 자세한 사용법은 1.3절에서 설명한다.

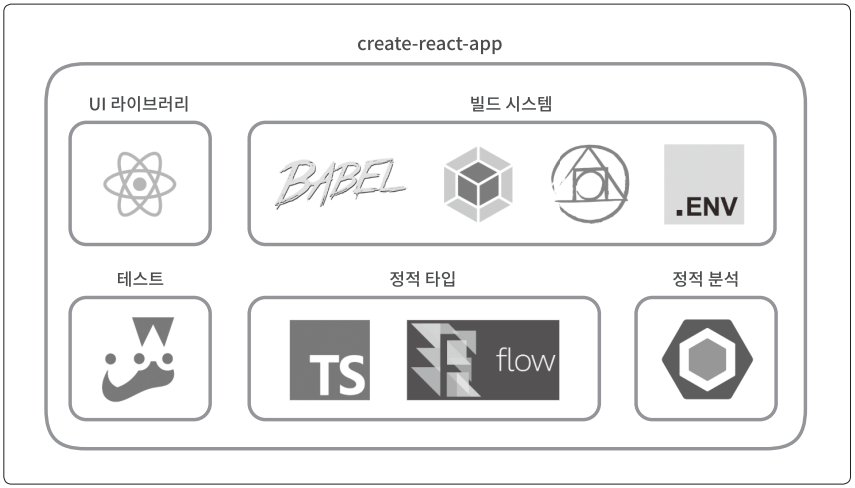


그림 1-1 create-react-app은 여러 패키지를 조합해서 리액트 개발 환경을 구축한다

리액트와 같은 프런트엔드 라이브러리 혹은 프레임워크를 사용하는 이유는 무엇일까? 가장 큰 이유 중의 하나는 UI를 자동으로 업데이트해 준다는 점이다. 대개 프로그램의 상태가 변하면 UI도 변경되는데, 이는 다음과 같이 함축적으로 표현할 수 있다.

```
UI = render(state)
```

우리는 API 통신이나 사용자 이벤트를 통해서 프로그램의 상탯값을 변경한다. 그리고 리액트가 변경된 상탯값을 기반으로 UI를 자동으로 업데이트한다. 리액트와 같은 도구를 사용하지 않으면 브라우저의 돔을 직접 업데이트해야 한다. 돔을 직접 업데이트하는 코드는 잘 관리하지 않으면 프로그램이 커질수록 복잡도가 기하급수적으로 증가한다. 따라서 UI 업데이트를 순수 자바스크립트로 처리하려면 리액트에 상응하는 자체 라이브러리를 만들어서 관리하는 게 좋다.

리액트의 장점은 가상 돔(virtual dom)을 통해서 UI를 빠르게 업데이트한다는 점이다. 가상 돔은 이전 UI 상태를 메모리에 유지해서, 변경될 UI의 최소 집합을 계산하는 기술이다. 가상 돔 덕분에 불필요한 UI 업데이트는 줄고, 성능은 좋아진다. 가상 돔에 대한 자세한 내용은 3장에서 확인할 수 있다.

리액트는 함수형 프로그래밍을 적극적으로 활용한다는 특징이 있다. 리액트에는 다음과 같은 제약 사항이 있는데, 순수 함수와 불변 객체는 함수형 프로그래밍에서 자주 언급되는 개념이다.

- 렌더 함수는 순수 함수로 작성해야 한다.
- 컴포넌트 상탯값은 불변 객체로 관리해야 한다.

앞의 공식에서 `render` 함수는 순수 함수여야 하므로 인수 `state`가 변하지 않으면 항상 같은 값을 반환해야 한다. 그리고 컴포넌트의 상탯값을 수정할 때는 기존 값을 변경하는 게 아니라 새로운 객체를 생성해야 한다. 코드에서 순수 함수와 불변 객체를 적극적으로 사용하면 복잡도가 낮아지고, 찾기 힘든 버그가 발생할 확률이 줄어든다. 더 중요한 것은 리액트에서 이 두 제약 사항 덕분에 렌더링 성능을 크게 향상할 수 있다는 점이다. 자세한 내용은 4장에서 확인할 수 있다.

1.2 리액트 개발 환경 직접 구축하기

이번 절에서는 리액트로 웹 애플리케이션을 만들기 위한 개발 환경을 직접 구축해 본다. 리액트는 UI 라이브러리이기 때문에 UI를 제외한 나머지 요소들은 개발자가 신경 써서 관리해야 한다. 하나의 웹 애플리케이션을 만들기 위해서는 테스트 시스템, 빌드 시스템, 라우팅 시스템 등 UI 외에도 신경 써야 할 부분이 많다. 필자는 리액트 개발 환경을 직접 구축하기보다는 create-react-app과 같은 도구를 사용할 것을 추천한다. 하지만 리액트 웹 애플리케이션의 툴체인(toolchain)을 이해하기 위해서는 한 번쯤 직접 구축해 보는 것도 좋다. 지금부터 리액트 개발 환경을 직접 구축하면서, 바벨과 웹팩의 필요성을 이해해 보자. create-react-app의 사용법은 1.3절에서 설명한다.

1.2.1 Hello World 페이지 만들기

리액트로 웹 애플리케이션을 제작할 때는 다양한 외부 패키지를 활용하는 게 일반적이다. 하지만 이런 다양한 외부 패키지의 존재는 처음 리액트를 접하는 사람에게 오히려 큰 부담이 된다. 사용되는 외부 패키지가 너무 많아서 각 패키지의 용도도 모르는 경우가 많다.

외부 패키지를 전혀 사용하지 않고 리액트로 간단한 웹 페이지를 만들어 보자. 우선 다음 경로에서 리액트 자바스크립트 파일 4개를 내려받는다.

- *https://unpkg.com/react@16/umd/react.development.js* ❶

- *https://unpkg.com/react@16/umd/react.production.min.js* ❷
- *https://unpkg.com/react-dom@16/umd/react-dom.development.js* ❸
- *https://unpkg.com/react-dom@16/umd/react-dom.production.min.js* ❹

이름에서 알 수 있듯이 ❶ ❸ 파일은 개발 환경에서 사용되는 파일이고, ❷ ❹ 파일은 배포 환경에서 사용되는 파일이다. 개발 환경을 위한 파일을 사용하면 개발 시 도움이 되는 에러 메시지를 확인할 수 있다. ❶ ❷ 파일은 플랫폼 구분 없이 공통으로 사용되는 리액트의 핵심 기능을 담고 있다. 따라서 웹뿐만 아니라 리액트 네이티브(react-native)에서도 사용된다. ❸ ❹ 파일은 웹에서만 사용되는 파일이다.

리액트 네이티브

리액트 네이티브를 이용하면 리액트로 안드로이드와 iOS의 네이티브 앱을 만들 수 있다. 웹 애플리케이션을 개발할 때 사용되는 리액트 패키지가 리액트 네이티브에서도 그대로 사용된다. react-dom 패키지는 웹 애플리케이션에서만 사용되며, 웹에서의 react-dom 역할을 하는 리액트 네이티브 코드가 별도로 존재한다.

리액트 네이티브를 이용하면 하나의 소스코드로 안드로이드와 iOS에서 동작하는 앱을 만들 수 있다는 점이 매력적이다. 하지만 인앱 구매나 푸시 알림과 같이 플랫폼에 종속적인 기능을 사용하기 위해서는 플랫폼별로 코드를 작성해야 한다.

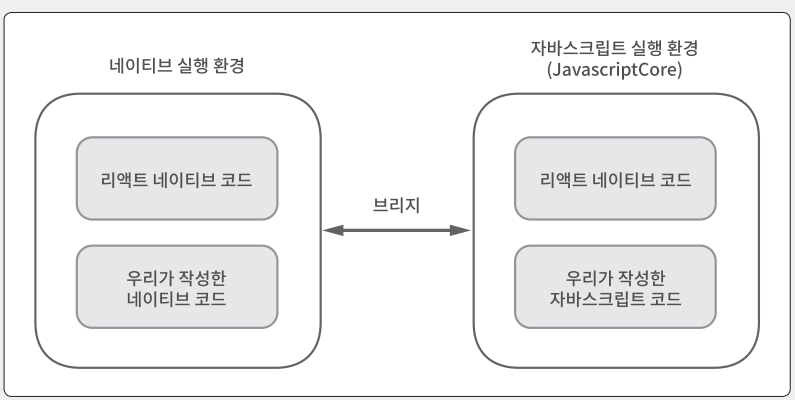


그림 1-2 react-native의 구조

리액트 네이티브는 모바일에서 자바스크립트를 실행하기 위해 JavascriptCore를 사용한다. JavascriptCore는 웹킷(webkit)에 내장된 자바스크립트 엔진이다. 대부분의 모바일 운영체제는 앱에서 C++ 코드를 실행할 방법을 제공해 준다. 따라서 리액트 네이티브는 C++로 작성된 JavascriptCore를 앱 빌드 시 포함함으로써 자바스크립트 실행 환경을 제공할 수 있다.

리액트의 가상 돔(virtual dom)은 마치 웹과 같은 돔 환경에서만 동작할 것만 같은 이름이지만 실제로는 리액트 네이티브에서도 동작한다. 가상 돔은 데이터가 변경됐을 때 UI에서 변경된 부분을 빨리 찾기 위해 사용되는 범용적인 자료구조다. 처음에 리액트는 웹을 위해 만들어졌기 때문에 가상 돔이라는 이름을 사용했지만, 리액트 네이티브가 존재하는 현시점에는 어울리지 않는 이름이 되었다.

이제 리액트 패키지만 사용해서 간단한 웹 애플리케이션을 만들어 보자. hello-world라는 폴더를 만든 다음, 앞에서 언급한 네 개의 파일을 넣는다. 그리고 같은 폴더에 내용이 없는 simple1.html, simple1.js 두 파일을 만든다.

```
hello-world
├── react.development.js
├── react.production.min.js
├── react-dom.development.js
├── react-dom.production.min.js
├── simple1.html
└── simple1.js
```

simple1.html 파일에는 필요한 자바스크립트 파일과 리액트에서 사용할 돔 요소를 정의한다. simple1.html 파일에 다음 내용을 입력하자.

코드 1-1 simple1.html

```
<html>
  <body>
    <h2>안녕하세요. 이 프로젝트가 마음에 드시면 좋아요 버튼을 눌러 주세요.</h2>
    <div id="react-root"></div> ❶
    <script src="react.development.js"/></script>
    <script src="react-dom.development.js"></script> ❷
    <script src="simple1.js"></script> ❸
  </body>
</html>
```

❶ 리액트로 렌더링할 때 사용할 돔 요소를 만들었다. 앞으로 리액트는 이 요소 안쪽에 새로운 돔 요소를 추가한다.

❷ 앞에서 준비한 리액트 파일을 script 태그로 입력했다.

❸ 앞으로 우리는 simple1.js 파일에 리액트 코드를 작성할 것이다.

simple1.js 파일에는 **좋아요** 버튼을 보여 주는 리액트 컴포넌트를 작성해 보자. 버튼을 누르면 **좋아요 취소** 문구를 보여 준다. simple1.js 파일에는 다음 내용을 입력하자.

코드 1-2 **simple1.js**

```
class LikeButton extends React.Component { ❶
  constructor(props) {
    super(props);
    this.state = { liked: false }; ❷
  }
  render() {
    const text = this.state.liked ? '좋아요 취소' : '좋아요'; ❸
    return React.createElement( ❹
      'button',
      { onClick: () => this.setState({ liked: true }) }, ❺
      text,
    );
  }
}
const domContainer = document.querySelector('#react-root'); ❻
ReactDOM.render(React.createElement(LikeButton), domContainer); ❼
```

❶ React 변수는 react.development.js 파일에서 전역 변수로 생성된다. ❷ 초깃값과 함께 컴포넌트의 상탯값을 정의한다. ❸ 컴포넌트의 상탯값에 따라 동적으로 버튼의 문구를 결정한다. ❹ createElement 함수는 리액트 요소를 반환한다. 여기서 생성한 리액트 요소는 최종적으로 버튼 돔 요소가 된다. ❺ 버튼을 클릭하면 onClick 함수가 호출되고, 컴포넌트의 상탯값이 변경된다. ❻ simple1.html 파일에 미리 만들어 뒀던 돔 요소를 가져온다. ❼ react-dom.development.js 파일에서 전역 변수로 만든 ReactDOM 변수를 사용해서 우리가 만든 컴포넌트를 react-root 돔 요소에 붙인다.

드디어 리액트로 만든 첫 번째 웹 페이지가 완성됐다. 페이지를 브라우저에 띄우면 **좋아요** 버튼이 보인다. **좋아요** 버튼을 클릭하면 **좋아요 취소** 버튼이 보이는 것을 확인할 수 있다.

createElement 이해하기

createElement 함수의 구조는 다음과 같다.

```
React.createElement(component, props, …children) => ReactElement
```

첫 번째 매개변수 component는 일반적으로 문자열이나 리액트 컴포넌트다. component의 인수가 문자열이면 HTML 태그에 해당하는 돔 요소가 생성된다. 예를 들어, 문자열 p를 입력하면 HTML p 태그가 생성된다.

두 번째 매개변수 props는 컴포넌트가 사용하는 데이터를 나타낸다. 돔 요소의 경우 style, className 등의 데이터가 사용될 수 있다.

세 번째 매개변수 children은 해당 컴포넌트가 감싸고 있는 내부의 컴포넌트를 가리킨다. div 태그가 두 개의 p 태그를 감싸고 있는 경우에 다음과 같이 작성할 수 있다.

코드 1-3 createElement 사용법

```
<div>
  <p>hello</p>
  <p>world</p>
</div>
```

❶

```
createElement(
  'div',
  null,
  createElement('p', null, 'hello'),
  createElement('p', null, 'world'),
)
```

❷

❶ 일반적인 HTML 코드다. ❷ 같은 코드를 createElement 함수를 사용해서 작성했다. 대부분의 리액트 개발자는 createElement를 직접 작성하지 않는다. 일반적으로 바벨(babel)의 도움을 받아서 JSX 문법을 사용한다. 이는 createElement 함수보다는 JSX 문법으로 작성하는 리액트 코드가 훨씬 가독성이 좋기 때문이다. 바벨을 통한 JSX 문법의 사용은 잠시 후 설명한다.

여러 개의 돔 요소에 렌더링하기

리액트가 돔 요소의 한곳에만 렌더링할 수 있는 것은 아니다. 코드 1-1과 1-2를

조금 수정해서 돔 요소 세 군데에 **좋아요** 버튼을 렌더링해 보자.

simple1.html 파일을 복사해서 simple2.html 파일을 만들자. 그리고 다음과 같이 수정해 보자.

코드 1-4 **simple2.html**

```
<html>
  <body>
    <h2>안녕하세요. 이 프로젝트가 마음에 드시면 좋아요 버튼을 눌러 주세요.</h2>
    <div id="react-root1"></div>
    <!-- ... -->
    <div id="react-root2"></div>
    <!-- ... -->
    <div id="react-root3"></div>
    <script src="react.development.js"/></script>
    <script src="react-dom.development.js"></script>
    <script src="simple2.js"></script> ❷
  </body>
</html>
```

❶

❶ 기존의 react-root 돔 요소는 지우고 세 개의 돔 요소를 만들었다. 각 요소 사이에 다른 코드가 있다는 사실에 주목하자. 다른 코드가 없다면 HTML에서는 하나의 요소만 만들고 리액트 코드에서 여러 개의 버튼을 구성하는 게 낫다. ❷ 새로 만들 자바스크립트 파일 이름으로 변경했다.

코드 1-4에서 작성한 세 개의 div 요소에 LikeButton 컴포넌트를 렌더링해 보자. simple1.js 파일을 복사해서 simple2.js 파일을 만들고 다음과 같이 수정하자.

코드 1-5 **simple2.js**

```
// ... ❶
ReactDOM.render(
  React.createElement(LikeButton),
  document.querySelector('#react-root1'),
);
ReactDOM.render(
  React.createElement(LikeButton),
  document.querySelector('#react-root2'),
);
ReactDOM.render(
  React.createElement(LikeButton),
  document.querySelector('#react-root3'),
);
```

❶ LikeButton 컴포넌트는 수정하지 않는다. 미리 만들어 놓은 세 개의 돔 요소에 LikeButton 컴포넌트를 렌더링한다. simple2.html 파일을 브라우저에서 열어 보면 세 개의 **좋아요** 버튼을 확인할 수 있다.

1.2.2 바벨 사용해 보기

바벨(babel)은 자바스크립트 코드를 변환해 주는 컴파일러다. 바벨을 사용하면 최신 자바스크립트 문법을 지원하지 않는 환경에서도 최신 문법을 사용할 수 있다. ES6가 막 나왔을 때는 대부분의 브라우저가 ES5만 지원하고 있었기 때문에 ES6 문법을 사용할 수 없었다. 그때 바벨(당시 이름은 6to5)이 ES6 문법으로 작성된 자바스크립트 코드를 ES5 문법으로 변환해 줬다. ES6 문법을 사용하고 싶어 하는 개발자가 많았기 때문에 바벨의 인기도 높아졌다.

바벨은 자바스크립트 최신 문법을 사용하는 용도 외에도 다양하게 활용될 수 있다. 이를테면, 코드에서 주석을 제거하거나 코드를 압축하는 용도로 사용될 수 있다. 리액트에서는 JSX 문법을 사용하기 위해 바벨을 사용한다. 바벨이 JSX 문법으로 작성된 코드를 createElement 함수를 호출하는 코드로 변환해 준다.

우리는 지금까지 외부 패키지 없이 리액트 웹사이트를 만들었다. 여기서는 최초의 외부 패키지로 바벨을 추가해 보려고 한다. 현재까지의 코드는 너무 단순하므로 그 전에 몇 가지 컴포넌트를 먼저 추가해 보겠다.

화면에 count 상탯값을 보여 주고 증가, 감소 버튼을 통해서 count 상탯값을 변경하는 코드를 작성해 보자. 먼저 simple1.html, simple1.js 파일을 복사해서 simple3.html, simple3.js 파일을 만들고, simple3.html에 있는 simple1.js 문자열을 simple3.js로 변경하자. 그리고 simple3.js 파일에 Container 컴포넌트를 추가해 보자.

코드 1-6 **simple3.js**

```
class LikeButton extends React.Component {
  // 기존 코드와 같음
}
class Container extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }
```

```
  render() {
    return React.createElement(
      'div',
      null,
      React.createElement(LikeButton),
      React.createElement(
        'div',
        { style: { marginTop: 20 } },
        React.createElement('span', null, '현재 카운트: '),
        React.createElement('span', null, this.state.count),
        React.createElement(
          'button',
          { onClick: () => this.setState({ count: this.state.count + 1 }) },
          '증가',
        ),
        React.createElement(
          'button',
          { onClick: () => this.setState({ count: this.state.count - 1 }) },
          '감소',
        ),
      ),
    );
  }
}
const domContainer = document.querySelector('#react-root');
ReactDOM.render(React.createElement(Container), domContainer); ❷
```

❶

❶ 단순한 기능인데도 render 메서드 코드가 상당히 복잡하다. 바벨의 도움을 받아서 이 부분을 개선해 보자. ❷ 기존에 LikeButton이 들어 있던 코드가 Container로 변경됐다. 그 대신 LikeButton 컴포넌트는 Container 컴포넌트 내부에서 사용되고 있다.

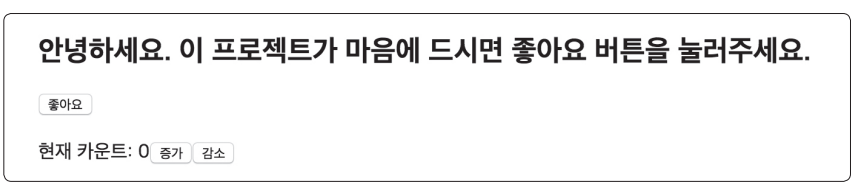
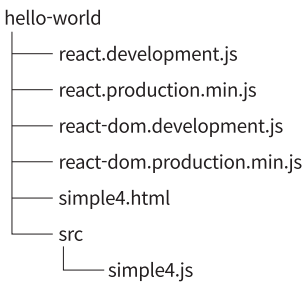


그림 1-3 simple3.html의 결과 화면

JSX 문법 사용해 보기

Container 컴포넌트의 render 메서드는 JSX 문법을 사용하면 가독성이 좋아진다. 코드 1-6을 JSX 문법을 사용한 버전으로 작성해 보자. 우선 simple3.html 파

일을 복사해서 simple4.html 파일을 만들자. simple4.html에 있는 `simple3.js` 문자열을 `simple4.js`로 변경하고, hello-world 폴더 밑에 src 폴더를 만든다. 그 다음 simple3.js 파일을 복사해서 src 폴더 밑에 simple4.js 파일을 만든다. 여기까지 잘 따라 했다면 다음과 같은 구조가 된다.



이제 `createElement` 함수를 호출하는 코드를 JSX 문법으로 변경해 보자. simple4.js 파일에서 `Container` 컴포넌트를 다음과 같이 변경한다.

코드 1-7 simple4.js

```
class Container extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }
  render() {
    return (
      <div> ❶
        <LikeButton />
        <div style={{ marginTop: 20 }}>
          <span>현재 카운트: </span>
          <span>{this.state.count}</span>
          <button
            onClick={() => this.setState({ count: this.state.count + 1 })}
          >
            증가
          </button>
          <button
            onClick={() => this.setState({ count: this.state.count - 1 })}
          >
            감소
          </button>
        </div>
      </div>
    );
  }
}
```

❶ render 메서드에서 createElement 함수를 사용하지 않고 JSX 문법을 사용했다.

JSX 문법 알아보기

JSX는 HTML에서 태그를 사용하는 방식과 유사하다. createElement 함수를 사용해서 렌더 함수를 작성하는 것보다는 JSX 문법을 사용하는 게 간결하고 가독성도 좋다. HTML 태그와의 가장 큰 차이는 속성값을 작성하는 방법에 있다.

코드 1-8 **JSX 문법 사용 예**

```
<div className="box"> ❶
  <Title text="hello world"  width={200} /> ❷
  <button onClick={() => {}}>좋아요</button> ❸
  <a href="/home" style={{ marginTop: '10px', color: 'red' }}> ❹
    홈으로 이동
  </a>
</div>
```

❶ HTML에서 돔 요소에 CSS 클래스 이름을 부여할 때 class 키워드를 사용했다면, JSX에서는 className 키워드를 사용한다. 이는 class라는 이름이 자바스크립트의 class 키워드와 같기 때문이다.

❷ Title은 리액트 컴포넌트다. JSX에서는 돔 요소와 리액트 컴포넌트를 같이 사용할 수 있다. Title 컴포넌트는 text, width라는 두 개의 속성값을 입력받는다. width처럼 문자열 리터럴이 아닌 속성값은 중괄호를 사용해서 입력한다. Title 컴포넌트가 두 개의 속성값을 이용해서 어떤 돔 요소로 치환될지는 Title 컴포넌트의 렌더 함수가 결정한다.

❸ 이벤트 처리 함수는 브라우저마다 다르게 동작할 수 있기 때문에 리액트와 같은 라이브러리를 사용하지 않을 때는 주의해야 한다. 다행히 리액트에서는 이벤트 처리 함수를 호출할 때 브라우저에 상관없이 통일된 이벤트 객체(SyntheticEvent)를 전달해 준다.

❹ HTML에서 돔 요소에 직접 스타일을 적용하는 것과 같이 JSX에서도 스타일을 적용할 수 있다. 다만 자바스크립트에서는 속성 이름에 대시(-)로 연결되는 이름을 사용하는 게 힘들기 때문에 카멜 케이스(camel case)를 사용한다.

JSX 문법을 바벨로 컴파일하기

JSX 문법은 자바스크립트 표준이 아니기 때문에 simple4.js 파일을 그대로 실행하면 에러가 발생한다. 바벨을 이용해서 JSX 문법으로 작성된 simple4.js 파일을 createElement 함수로 작성된 파일로 변환해 보자. 먼저 파일을 변환하기 위해서 다음 패키지를 설치해야 한다.

```
npm install @babel/core @babel/cli @babel/preset-react
```

@babel/cli에는 커맨드 라인에서 바벨을 실행할 수 있는 바이너리 파일이 들어 있다. @babel/preset-react에는 JSX로 작성된 코드를 `createElement` 함수를 이용한 코드로 변환해 주는 바벨 플러그인이 들어 있다.

> **바벨 플러그인과 프리셋**
>
> 바벨은 자바스크립트 파일을 입력으로 받아서 또 다른 자바스크립트 파일을 출력으로 준다. 이렇게 자바스크립트 파일을 변환해 주는 작업은 플러그인(plugin) 단위로 이루어진다. 두 번의 변환이 필요하다면 두 개의 플러그인을 사용한다. 하나의 목적을 위해 여러 개의 플러그인이 필요할 수 있는데, 이러한 플러그인의 집합을 프리셋(preset)이라고 부른다. 예를 들어, 바벨에서는 자바스크립트 코드를 압축하는 플러그인을 모아 놓은 babel-preset-minify 프리셋을 제공한다. @babel/preset-react은 리액트 애플리케이션을 만들 때 필요한 플러그인을 모아 놓은 프리셋이다.

설치된 패키지를 이용해서 자바스크립트 파일을 변환해 보자.

```
npx babel --watch src --out-dir . --presets @babel/preset-react
```

`npx` 명령어는 외부 패키지에 포함된 실행 파일을 실행할 때 사용된다. 외부 패키지의 실행 파일은 `./node_modules/.bin/` 밑에 저장된다. 따라서 `npx babel`은 `./node_modules/.bin/babel`을 입력하는 것과 비슷하다. 오래된 npm 버전을 사용한다면 `npx` 명령어가 동작하지 않으므로, 최신 버전의 npm을 설치하거나 `./node_modules/.bin/babel`을 입력하자.

위 명령어를 실행하면 src 폴더에 있는 모든 자바스크립트 파일을 @babel/preset-react 프리셋을 이용해서 변환 후 현재 폴더에 같은 이름의 자바스크립트 파일을 생성한다. watch 모드로 실행했기 때문에 src 폴더의 자바스크립트 파일을 수정할 때마다 자동으로 변환 후 저장한다. 바벨로 변환 후 simple4.html을 실행해 보면 simple3.html과 같은 결과 화면을 볼 수 있다. 바벨에 대한 자세한 설명은 7장에서 다루기로 한다.

1.2.3 웹팩의 기본 개념 이해하기

웹팩(webpack)은 자바스크립트로 만든 프로그램을 배포하기 좋은 형태로 묶어 주는 툴이다. 배포하기 좋은 형태란 무엇일까? 반대로 웹팩을 사용하지 않고 배포할 때는 어떤 어려운 점이 있는지 알아보자.

2000년대 초반의 웹 페이지는 페이지가 전환될 때마다 새로운 HTML 파일을 요청해서 화면을 새로 그리는 방식이었다. 그 당시 자바스크립트는 돔을 조작하는 간단한 역할만 했기 때문에 코드의 양이 많지 않았다. 한두 개의 자바스크립트 파일을 HTML의 script 태그를 이용해서 서비스하는 방식이면 충분했다. Ajax가 유행했을 때는 자바스크립트의 비중이 조금 더 커졌지만 많아 봐야 페이지당 자바스크립트 파일 열 개 정도 수준이었다.

웹사이트 제작 방식이 단일 페이지 애플리케이션(single page application)으로 전환되면서 상황은 달라졌다. 한 페이지에 자바스크립트 파일 수십 또는 수백 개가 필요했기 때문에 더는 기존 방식이 통할 리 없었다.

코드 1-9 전통적인 방식으로 개발된 웹사이트의 HTML 코드

```
<html>
    <head>
        <script type="text/javascript" src="javascript_file_1.js"></script>
        <script type="text/javascript" src="javascript_file_2.js"></script>
        <!-- ... -->
        <script type="text/javascript" src="javascript_file_999.js"></script>
    </head>
    <!-- ... -->
</html>
```

사실 이런 방식으로는 계속 늘어나는 자바스크립트 파일을 관리하기가 힘들다. 파일 간의 의존성 때문에 선언되는 순서를 신경 써야 하기 때문이다. 그리고 뒤에 선언된 자바스크립트 파일이 앞에 선언된 파일에서 생성한 전역 변수를 덮어쓰는 위험도 존재한다.

자바스크립트의 모듈 시스템

C++나 Java에서는 include, import 키워드를 이용해서 한 파일에서 다른 파일의 코드를 가져다 사용할 수 있다. 하나의 파일이 하나의 모듈이 되고 사용하는 쪽에서는 여러 모듈을 가져다 쓸 수 있다. 이때 모듈 측에서는 필요한 부분만 내보내는 방법이 필요하고, 사용하는 측에서는 필요한 것만 가져다 쓸 방법이 필요하다. 이렇게 내보내고 가져다 쓸 수 있도록 구현된 시스템이 모듈 시스템이다.

자바스크립트에는 ES6부터 모듈 시스템이 언어 차원에서 지원된다. 현재 모든 최신 브라우저에서는 ES6의 모듈 시스템을 지원한다. 하지만 예전 버전의 브라우저에서는 모듈 시스템을 사용할 수 없다. 또한 상당히 많은 수의 오픈 소스가 ES6 모듈로 작성되지 않았다는 것도 큰 걸림돌이다.

ES6가 나오기 이전부터 자바스크립트의 모듈 시스템을 요구하는 개발자가 많았다. 그래서 등장한 대표적인 자바스크립트 모듈 시스템이 commonJS이다. node.js가 commonJS 표준을 따르면서 commonJS가 널리 퍼지기 시작했다. 현재 많은 수의 오픈 소스가 commonJS 모듈 시스템으로 구현되어 있다.

웹팩은 ESM(ES6의 모듈 시스템)과 commonJS를 모두 지원한다. 이들 모듈 시스템을 이용해서 코드를 작성하고 웹팩을 실행하면 예전 버전의 브라우저에서도 동작하는 자바스크립트 코드를 만들 수 있다. 웹팩을 실행하면 보통은 하나의 자바스크립트 파일이 만들어지고, 원한다면 여러 개의 파일로 분할할 수도 있다. 우리가 할 일은 웹팩이 만들어 준 자바스크립트 파일을 HTML의 `script` 태그에 포함시키는 것이다.

ESM 문법 익히기

ESM 문법을 익히기 위해 모듈을 내보내고 가져오는 코드를 작성해 보자. 다음 코드는 세 파일의 내용을 보여 준다. file1.js 파일은 코드를 내보내는 쪽이고 file2.js, file3.js 파일은 코드를 사용하는 쪽이다.

코드 1-10 ESM 예제 코드

```
// file1.js 파일
export default function func1() {} ❶
export function func2() {}
export const variable1 = 123;
export let variable2 = 'hello'; ❷

// file2.js 파일
import myFunc1, { func2, variable1, variable2 } from './file1.js'; ❸

// file3.js 파일
import { func2 as myFunc2 } from './file1.js'; ❹
```

❶, ❷ 코드를 내보낼 때는 `export` 키워드를 사용한다. ❸ 코드를 사용하는 쪽에서는 `import`, `from` 키워드를 사용한다. ❶ `default` 키워드는 한 파일에서 한 번만 사용할 수

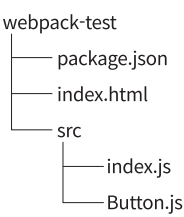
있다. ❸ default 키워드로 내보내진 코드는 괄호 없이 가져올 수 있고, 이름은 원하는 대로 정할 수 있다. ❶번 코드에서 내보낸 func1 함수는 ❸번 코드에서 myFunc1이라는 이름으로 가져왔다. ❸ default 키워드 없이 내보내진 코드는 괄호를 사용해서 가져온다. 가져올 때 이름은 내보낼 때 사용된 이름 그대로 가져와야 한다. ❹ 원한다면 as 키워드를 이용해서 이름을 변경해서 사용할 수 있다.

1.2.4 웹팩 사용해 보기

웹팩을 사용해서 리액트의 두 파일을 자바스크립트의 모듈 시스템으로 포함시켜 보자. webpack-test라는 폴더를 만들고 그 폴더에서 다음 명령어를 실행한다.

```
npm init -y
```

명령어를 실행하면 package.json 파일이 만들어진다. simple1.html 파일을 복사해서 webpack-test 폴더 밑에 index.html 파일을 만들고, index.html에 있는 `simple1.js` 문자열을 `dist/main.js`로 변경하자. 그 다음 react.development.js, react-dom.development.js 파일을 포함하고 있는 script 태그를 지운다. 이 두 리액트 파일은 모듈 시스템을 이용해서 main.js 파일에 포함될 예정이다. webpack-test 폴더 밑에 src 폴더를 만들자. src 폴더 밑에 내용이 없는 index.js, Button.js 파일을 만든다. 여기까지 잘 따라 했다면 파일 구조는 다음과 같을 것이다.



이제 필요한 외부 패키지를 설치해 보자.

```
npm install webpack webpack-cli react react-dom
```

웹팩과 함께 리액트 패키지도 설치했다. react 패키지에는 우리가 위에서 내려

받았던 react.production.min.js, react.development.js 파일이 포함되어 있다. 마찬가지로 react-dom 패키지에는 react-dom.production.min.js, react-dom.development.js 파일이 포함되어 있다. 이전에는 url을 직접 입력해서 각각의 파일을 내려받았지만, 이제는 모듈 시스템과 npm 덕분에 외부 패키지를 프로젝트에 쉽게 포함할 수 있게 되었다.

ESM 문법을 이용해서 다른 모듈을 가져오는 코드를 작성해 보자. 먼저 index.js 파일에 다음 내용을 입력한다.

코드 1-11 index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import Button from './Button.js';

function Container() {
  return React.createElement(
    'div',
    null,
    React.createElement('p', null, '버튼을 클릭해 주세요.'),
    React.createElement(Button, { label: '좋아요' }),
    React.createElement(Button, { label: '싫어요' }),
  );
}
const domContainer = document.querySelector('#react-root');
ReactDOM.render(React.createElement(Container), domContainer);
```

❶ ESM 문법을 이용해서 필요한 모듈을 가져오고 있다. ❷ 이제까지는 클래스형 컴포넌트만 사용했지만 Container 컴포넌트는 함수형 컴포넌트로 작성됐다. 렌더 함수만 필요한 경우에는 함수형 컴포넌트로 작성하는 게 좋다.

클래스형 컴포넌트와 함수형 컴포넌트

리액트 컴포넌트는 클래스형 컴포넌트 또는 함수형 컴포넌트로 작성될 수 있다. 둘 사이에는 분명한 차이점이 있기 때문에 각각의 장단점을 잘 이해하고 사용하는 게 좋다. 기능적인 측면에서 보면 클래스형 컴포넌트는 함수형 컴포넌트가 할 수 있는 모든 일을 할 수 있다. 리액트 16.8 이전 버전의 함수형 컴포넌트가 할 수 없는 일은 다음과 같다.

- 상탯값을 가질 수 없다.
- 리액트 컴포넌트의 생명 주기 함수를 작성할 수 없다.

> 리액트 버전 16.8부터 훅(hook)이라는 기능이 추가되면서 함수형 컴포넌트에서도 상탯값과 생명 주기 함수 코드를 작성할 수 있게 되었다. 새로운 프로젝트를 만든다면 되도록 클래스형 컴포넌트를 지양하고 훅과 함께 함수형 컴포넌트를 작성하는 게 좋다. 기존 프로젝트를 관리하고 있다면 클래스형 컴포넌트의 생명 주기 메서드를 잘 이해하고 있어야 한다. 페이스북 내부에도 클래스형 컴포넌트로 작성된 코드가 상당히 많기 때문에 리액트에서 클래스형 컴포넌트를 오랫동안 지원할 것으로 보인다.

Button.js 파일에는 Button 컴포넌트를 작성하고, ESM 문법을 이용해서 필요한 모듈을 가져오고 Button 컴포넌트를 내보내도록 하자.

코드 1-12 **Button.js**

```
import React from 'react';                    ❶

function Button(props) {
  return React.createElement('button', null, props.label);
}
export default Button;
```

❶ react 모듈을 가져오고 Button 컴포넌트를 내보내기 위해 ESM 문법을 사용했다.

이제 웹팩을 이용해서 두 개의 자바스크립트 파일을 하나의 파일로 합쳐 보자.

```
npx webpack
```

위 명령어를 실행하면 dist 폴더 밑에 main.js 파일이 생성된다. 이제 index.html 파일을 브라우저에서 실행해 보자. 화면에 두 개의 버튼이 보인다면 성공이다.

웹팩에는 이 외에도 다양한 기능이 있다. 자바스크립트 파일 압축, CSS 전처리 등 유용한 기능이 많다. 웹팩에 대한 자세한 설명은 7장에서 살펴보기로 한다.

1.3 create-react-app으로 시작하기

create-react-app은 리액트로 웹 애플리케이션을 만들기 위한 환경을 제공한다. 만약 리액트 네이티브에만 관심 있고 웹 애플리케이션 제작에 관심이 없다면 이 절은 건너뛰어도 좋다. 대신 리액트 네이티브의 개발 환경을 자동으로 구축해

주는 expo를 이용하기 바란다.

앞에서 다룬 바벨과 웹팩도 create-react-app에 포함되어 있다. 그 밖에 테스트 시스템, HMR(hot-module-replacement), ES6+ 문법, CSS 후처리 등 거의 필수라고 할 수 있는 개발 환경도 구축해 준다. 이러한 개발 환경을 직접 구축할 경우 시간이 오래 걸릴 뿐 아니라 유지 보수도 해야 한다. create-react-app를 이용하면 기존 기능을 개선하거나 새로운 기능을 추가했을 때 패키지 버전만 올리면 된다.

또 다른 장점은 어떤 문제를 해결하기 위한 선택지가 여러 개일 때 create-react-app에서 가장 합리적인 선택을 해 준다는 점이다. 현재 자바스크립트 생태계는 춘추전국 시대라 할 만하다. 지금 이 순간에도 수많은 새로운 패키지가 쏟아져 나온다. 기존에 가장 좋은 방법이라 여겨졌던 것이 새로운 방법으로 대체되는 경우도 있다. 그만큼 우리에게는 하나의 문제를 해결하기 위한 다양한 선택지가 있고, 이들 가운데 하나를 선택해야 하는 상황을 자주 접하게 된다. 여러 선택지 가운데 하나를 고르기 위해서는 각 선택지의 장단점을 공부해야 한다. 그렇기에 create-react-app을 사용하면 우리의 시간을 좀 더 중요한 곳에 할애할 수 있다.

1.3.1 create-react-app 사용해 보기

다음 명령어를 입력하면 create-react-app을 이용한 개발 환경이 설치된다.

```
npx create-react-app cra-test
```

create-react-app 패키지가 설치되어 있지 않더라도 npx가 자동으로 가져와서 실행한다. 만약 npm 버전이 낮아서 실행이 안 된다면 다음과 같이 입력한다.

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

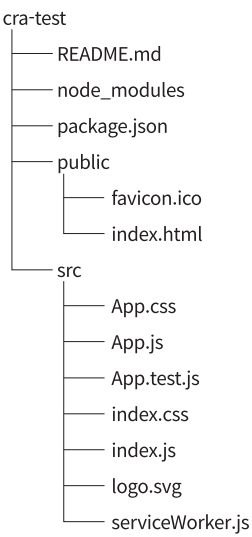
실행 후에는 cra-test 폴더가 생성되고 그 안에 몇 개의 폴더와 파일이 들어 있다. 거두절미하고 웹 페이지를 띄워 보자.

```
cd cra-test
npm start
```

빌드가 끝나면 자동으로 브라우저에서 새 탭이 열리고 렌더링된 페이지를 볼 수 있다. 이렇게 자동으로 브라우저를 띄워 주는 소소한 기능이 create-react-app의 매력 포인트다.

이 상태로 App.js 파일에서 `Welcome to React` 부분을 다른 텍스트로 수정해 보자. 브라우저의 화면이 자동으로 업데이트되는 것을 확인할 수 있다. 이번에는 App.css 파일에서 `20s` 부분을 `2s`로 변경해 보자. 마찬가지로 수정된 내용이 바로 적용되어 로고 이미지가 빠르게 회전하는 것을 볼 수 있다. 이는 HMR이라는 이름의 기능인데, `npm start` 실행 시 create-react-app이 로컬 서버를 띄워 주기 때문에 가능한 일이다. 참고로 `npm start`는 개발 모드에서 동작하므로 배포할 때 사용하면 안 된다.

create-react-app에서 자동으로 생성한 파일을 살펴보자. 다음은 자동으로 생성된 cra-test 폴더의 내부 구조이다.



index.html, index.js 파일은 빌드 시 예약된 파일 이름이므로 지우면 안 된다. index.html, index.js, package.json 파일을 제외한 나머지 파일은 데모 앱을 위한 파일이기 때문에 마음대로 수정하거나 삭제해도 괜찮다. index.js로부터 연결된 모든 자바스크립트 파일과 CSS 파일은 src 폴더 밑에 있어야 한다. src 폴더 바깥에 있는 파일을 `import` 키워드를 이용해서 가져오려고 하면 실패한다.

index.html에서 참조하는 파일은 public 폴더 밑에 있어야 한다. public 폴더

밑에 있는 자바스크립트 파일이나 CSS 파일을 `link`나 `script` 태그를 이용해서 index.html에 포함시킬 수 있다. 하지만 특별한 이유가 없다면 index.html에 직접 연결하는 것보다는 src 폴더 밑에서 `import` 키워드를 사용해서 포함시키는 게 좋다. 그래야 자바스크립트 파일이나 CSS 파일의 경우 빌드 시 자동으로 압축된다.

이미지 파일이나 폰트 파일의 경우도 마찬가지로 src 폴더 밑에서 `import` 키워드를 사용해서 포함시키는 게 좋다. 웹팩에서 해시값을 이용해서 url을 생성해 주기 때문에 파일의 내용이 변경되지 않으면 브라우저 캐싱 효과를 볼 수 있다.

파일을 참조하는 경우 외에도 index.html의 내용을 직접 수정해도 괜찮다. 대표적으로 `title` 태그에서 제목을 직접 입력하는 경우를 예로 들 수 있다. 그런데 제목을 페이지별로 다르게 줘야 한다면 문제가 될 수 있다. 만약 사내에서만 쓰는 웹사이트라면 react-helmet과 같은 패키지를 사용하면 된다. 반대로 일반 사용자를 대상으로 하는 웹사이트라서 검색 엔진 최적화를 해야 한다면 문제는 복잡해진다. 검색 엔진 최적화가 중요하다면 create-react-app으로 해결하는 것보다는 서버사이드 렌더링에 특화된 넥스트(next.js)를 사용하는 게 좋다. 넥스트에 대한 자세한 설명은 8장에서 살펴보기로 한다.

serviceWorker.js 파일에는 PWA(progressive web app)와 관련된 코드가 들어 있다. PWA는 오프라인에서도 잘 동작하는 웹 애플리케이션을 만들기 위한 기술이다. create-react-app으로 프로젝트를 생성하면 PWA 기능은 기본적으로 꺼져 있는 상태다. PWA 기능을 원한다면 index.js 파일에 `serviceWorker.register();` 코드를 넣으면 된다.

1.3.2 주요 명령어 알아보기

package.json 파일을 열어 보면 네 가지 npm 스크립트 명령어를 확인할 수 있다. 자주 사용되는 명령어이므로 하나씩 살펴보자.

개발 모드로 실행하기

`npm start`는 개발 모드로 프로그램을 실행하는 명령어이다. 개발 모드로 실행하면 HMR이 동작하기 때문에 코드를 수정하면 화면에 즉시 반영된다. HMR이 없다면 코드를 수정하고 브라우저에서 수동으로 새로고침을 해야 하므로 번거롭다. 개발 모드에서 코드에 에러가 있을 때는 브라우저에 에러 메시지가 출력된다.

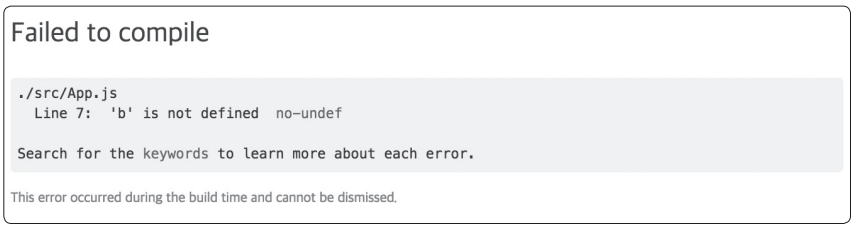


그림 1-4 개발 모드에서 발생한 에러는 브라우저 화면에 출력된다

에러 메시지가 출력된 영역을 클릭하면 에러가 발생한 파일이 에디터에서 열린다. 이는 create-react-app의 섬세함을 느낄 수 있는 기능이다.

때에 따라 API 호출을 위해서 https로 실행해야 할 수도 있다. https 환경을 직접 구축하기란 여간 귀찮은 일이 아닌데, 고맙게도 create-react-app에서는 https로 실행하는 옵션을 제공한다.

- 맥: `HTTPS=true npm start`
- 윈도우: `set HTTPS=true && npm start`

이 명령어를 실행하면 자체 서명된 인증서(self-signed certificate)와 함께 https 사이트로 접속한다. 자체 서명된 인증서이기 때문에 안전하지 않다는 경고 문구가 뜨지만 무시하고 진행하면 된다.

빌드하기

`npm run build` 명령어는 배포 환경에서 사용할 파일을 만들어 준다. 빌드 후 생성된 자바스크립트 파일과 CSS 파일을 열어 보면 사람이 읽기 힘든 형식으로 압축된 것을 확인할 수 있다. 이렇게 생성된 정적 파일을 웹 서버를 통해서 사용자가 내려받을 수 있게 하면 된다. 로컬에서 웹 서버를 띄워서 확인해 보자.

```
npx serve -s build
```

serve 패키지는 노드(node.js) 환경에서 동작하는 웹 서버 애플리케이션이다. 정적 파일을 서비스할 때 간단하게 사용하기 좋다.

build/static 폴더 밑에 생성된 파일의 이름에 해시값이 포함되어 있다. 파일의 내용이 변경되지 않으면 해시값은 항상 같다. 새로 빌드를 하더라도 변경되지 않은 파일은 브라우저에 캐싱되어 있는 파일이 사용된다. 따라서 재방문의 경우

빠르게 페이지가 렌더링되는 효과를 볼 수 있다.

자바스크립트 파일에서 import 키워드를 이용해서 가져온 CSS 파일은 다음 경로에 저장된다.

```
build/static/css/main.{해시값}.chunk.css
```

여러 개의 CSS 파일을 임포트하더라도 모두 앞의 파일에 저장된다. 자바스크립트 파일에서 import 키워드를 이용해서 가져온 폰트, 이미지 등의 리소스 파일은 build/static/media 폴더 밑에 저장된다. 이미지 파일의 크기가 10킬로바이트보다 작은 경우에는 별도의 파일로 생성되지 않고 *data url* 형식으로 자바스크립트 파일에 포함된다. 이는 작은 크기의 파일을 여러 번 요청하는 것보다는 한 번의 요청으로 처리하는 게 효율적이기 때문이다.

이미지 파일의 크기가 10킬로바이트보다 작은 파일과 큰 파일을 하나씩 준비해서 src 폴더 밑에 저장해 보자. 다음과 같이 App.js 파일에서 두 개의 이미지 파일을 불러오는 코드를 작성해 보자.

코드 1-13 App.js

```
// ...
import smallImage from './small.jpeg';
import bigImage from './big.jpeg';

class App extends Component {
  render() {
    return (
      <div className="App">
        <img src={bigImage} />
        <img src={smallImage} />
// ...
```

❶ 일반적인 자바스크립트 모듈처럼 이미지 파일을 자바스크립트로 가져왔다.

❷ bigImage, smallImage 변수는 해당 이미지의 경로를 나타내는 문자열이다.

지금까지 작업한 내용을 빌드해 보자. media 폴더에는 big.{해시값}.jpeg 파일이 생성된다. small.{해시값}.jpeg 파일은 생성되지 않는다. small.jpeg 파일의 내용은 어디에 있을까? main.{해시값}.js 파일에서 data:image를 키워드로 검색해 보면 이미지 파일이 문자열 형태로 자바스크립트 파일에 포함된다는 사실을 확인할 수 있다.

테스트 코드 실행하기

`npm test`를 입력하면 테스트 코드가 실행된다. create-react-app에는 제스트(jest)라는 테스트 프레임워크를 기반으로 테스트 시스템이 구축되어 있다. create-react-app으로 프로젝트를 생성하면 App.test.js 파일이 생성된다. create-react-app에서는 자바스크립트 파일이 다음 조건을 만족하면 테스트 파일로 인식한다.

- __tset__ 폴더 밑에 있는 모든 자바스크립트 파일
- 파일 이름이 .test.js로 끝나는 파일
- 파일 이름이 .spec.js로 끝나는 파일

util.js 파일을 생성해서 간단한 함수를 작성해 보자.

코드 1-14 **util.js**

```
export function addNumber(a, b) {
  return a; ❶
}
```

❶ 코드에 버그가 있기 때문에 테스트 코드를 작성하면 실패할 것이다.

이번에는 util.test.js 파일을 생성해서 `addNumber` 함수를 테스트하는 코드를 작성해 보자.

코드 1-15 **util.test.js**

```
import { addNumber } from './util';

it('add two numbers', () => { ❶
  const result = addNumber(1, 2);
  expect(result).toBe(3); ❷
});
```

❶, ❷ `it`, `expect`는 제스트에서 테스트 코드를 작성할 때 사용되는 함수이다.

제스트를 실행해서 테스트 결과를 확인해 보자.

```
npm test
```

App.test.js 파일은 성공하고 util.test.js 파일은 실패한다. util.js 파일의 버그를 수정해서 저장해 보자. 테스트 프로그램이 watch 모드로 동작하고 있기 때문에

util.test.js 테스트가 성공하는 것을 바로 확인할 수 있다.

CI(continuous integration)와 같이 watch 모드가 필요 없는 환경에서는 다음 명령어로 테스트 코드를 실행한다.

- 맥: `CI=true npm test`
- 윈도우: `set "CI=true" && npm test`

설정 파일 추출하기

`npm run eject`를 실행하면 숨겨져 있던 create-react-app의 내부 설정 파일이 밖으로 노출된다. 이 기능을 사용하면 바벨이나 웹팩의 설정을 변경할 수 있다. 이 기능의 단점은 create-react-app에서 개선하거나 추가된 기능이 단순히 패키지 버전을 올리는 식으로 적용되지 않는다는 점이다. 이 기능은 리액트 툴체인에 익숙한 사람이 아니라면 추천하지 않는다. `npm run eject` 외에도 create-react-app의 설정을 변경할 방법이 있다.

- 방법 1: react-scripts 프로젝트를 포크(fork)해서 나만의 스크립트를 만든다.
- 방법 2: react-app-rewired 패키지를 사용한다.

방법 1은 자유도가 높기 때문에 원하는 부분을 얼마든지 수정할 수 있다. 이렇게 수정된 내용을 여러 프로젝트에서 공통으로 사용할 수 있다는 장점도 있다. 방법 2는 자유도는 낮지만 비교적 쉽게 설정을 변경할 수 있다는 장점이 있다. 하지만 두 가지 방법 모두 create-react-app의 이후 버전에 변경된 내용을 쉽게 적용할 수 없다는 단점이 있다.

1.3.3 자바스크립트 지원 범위

create-react-app에서는 ES6의 모든 기능을 지원한다. ES6 이후에 추가되거나 제안된 기능 중에서 create-react-app(v2.1.3)에서 지원하는 기능은 다음과 같다.

- 지수 연산자(exponentiation operator)
- async await 함수
- 나머지 연산자(rest operator), 전개 연산자(spread operator)
- 동적 임포트(dynamic import)
- 클래스 필드(class field)

- JSX 문법
- 타입스크립트(typescript), 플로(flow) 타입 시스템

create-react-app에서는 타입스크립트와 플로 타입 시스템을 지원한다. 필자는 정적 타입을 사용할 것을 추천한다. 자바스크립트에서 정적 타입 시스템을 적용할 수 있는 방법은 여러 가지가 있지만, 현재로서는 타입스크립트가 가장 괜찮은 선택이라고 생각한다. 타입스크립트에 대한 자세한 내용은 9장에서 살펴보기로 한다.

create-react-app의 기본 설정에서는 아무런 폴리필(polyfill)도 포함되지 않는다. ES6+에서 추가된 객체나 함수를 사용하고 싶다면 직접 폴리필을 넣도록 하자. ES8에 추가된 String.padStart 함수를 사용하고 싶다고 가정해 보자. core-js 패키지를 사용하면 다양한 폴리필을 선택적으로 사용할 수 있다. 우선 core-js 패키지를 설치해 보자.

```
npm install core-js
```

그리고 다음과 같이 작성하면 폴리필이 추가된다.

코드 1-19 core-js 패키지를 이용한 폴리필

```
// index.js
import 'core-js/features/string/pad-start'; ❶

// someFile.js
const value = '123'.padStart(5, '0'); // '00123'
```

❶ index.js 파일에서 한 번만 가져오면 모든 곳에서 자유롭게 사용할 수 있다.

바벨에서도 @babel/polyfill 혹은 @babel/preset-env 프리셋을 이용하면 폴리필을 추가할 수 있다. @babel/polyfill은 사용하지 않는 기능의 폴리필까지 모두 포함되기 때문에 번들(bundle) 크기가 커지는 단점이 있다. @babel/preset-env 프리셋을 이용하면 필요한 폴리필만 추가할 수 있지만 동적 타입 언어의 한계 때문에 core-js로 직접 추가하는 것보다는 몇 가지 불필요한 폴리필이 포함되는 단점이 있다. 바벨을 이용해서 폴리필을 추가하는 방법은 7장에서 살펴보기로 한다.

폴리필

새로운 자바스크립트 표준이 나와도 대다수 사용자의 브라우저에서 지원하지 않으면 사용할 수 없다. 언어 표준에는 새로운 문법도 추가되고 새로운 객체나 함수도 추가된다. 새로운 문법은 대부분의 브라우저에서 지원하지 않더라도 바벨을 이용하면 어느 정도 사용이 가능하다. 바벨을 사용하면 빌드 시점에 코드가 변환된다.

새로운 객체나 함수는 성격이 조금 다르다. 물론 새로운 객체나 함수로 작성한 코드도 빌드 시점에 변환할 수 있다. 하지만 이들은 실행 시점에 주입할 수 있다는 장점이 있다. 따라서 실행 시점에 주입하고자 하는 객체나 함수가 현재 환경에 존재하는지 검사해서 존재하지 않는 경우에만 주입하는 게 좋다. 이렇게 기능이 존재하는지 검사해서 그 기능이 없을 때만 주입하는 것을 폴리필이라고 부른다.

1.3.4 코드 분할하기

코드 분할(code splitting)을 이용하면 사용자에게 필요한 양의 코드만 내려 줄 수 있다. 코드 분할을 사용하지 않으면 전체 코드를 한 번에 내려 주기 때문에 첫 페이지가 뜨는 시간이 오래 걸린다. 코드를 분할하는 한 가지 방법은 이전에 언급했던 동적 임포트를 이용하는 것이다.

코드 분할을 이해하기 위해 간단하게 할 일 목록을 만들어 보자. src 폴더 밑에 Todo.js 파일을 생성해서 다음 내용을 입력해 보자.

코드 1-20 **Todo.js**

```
import React from 'react';

export function Todo({ title }) {
  return <div>{title}</div>;
}
```

TodoList.js 파일을 생성해서 `Todo` 컴포넌트를 이용하는 `TodoList` 컴포넌트 코드를 입력해 보자.

코드 1-21 **TodoList.js**

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