React 18 총정리 - 2 (Component)

1. Vite 프로젝트 살펴보기

Vite (한국어로 비트 라고 발음함)

주요 기능

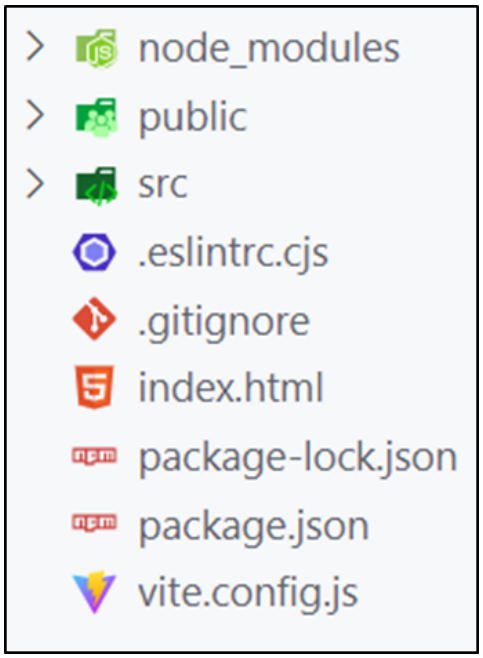
- 빌드 툴 (번들러)
- 기본적인 프로젝트 세팅 제공
- 각각의 컴포넌트를 `.jsx` 파일로 분리해 여러 개의 컴포넌트 사용 가능
- 클라이언트에서 node.js 테스트 서버 사용 가능

왜 이런 툴이 필요할까? React 개발에 필요한 필수적인 npm 패키지 두 가지, `react` 와 `react-dom` 을 설치해 일일히 프로젝트를 세팅하기 너무 어렵고 복잡하기 때문이다.

- 이전엔 webpack 기반 create-react-app 을 사용했으나, React 버전업 이후 추천 패키지에서 제외되었다. Vite 는 webpack 에 비해 훨씬 빠른 성능과 직관적인 구조를 제공하므로 사용하기 적절하다.

- Vite 는 React 공식 번들러가 아니다. 정확하게 말하면, React 에서 추천하는 공식 번들러는 존재하지 않는다. React 는 프레임워크가 아니라 라이브러리이며, 이는 React 에 적용할 수 있는 모든 서드파티 라이브러리에 열려있다는 뜻이기도 하다. 이에 Vite 를 쓰는 게 권장되는지 의문일 수 있는데, 우리가 배우게 될 React Router 의 경우 공식 설치법이 Vite 로 쓰여져있기도 하고, 빠른 성능때문에 Vite 를 사용하는 개발자들이 빠른 속도로 늘고 있다.
- 필자의 개인 견해로서, React 공식문서에서 Vite 를 추천하지 않는 또다른 이유는 Vite 가 경쟁 프레임워크인 Vue.js 팀에서 만든 번들러이기 때문일수도 있다.

어쨌든, 우리 수업은 Vite 로 진행할 것이다. 프로젝트 구조를 살펴보자.



- `node_modules/`
  - 프로젝트에 필요한 패키지가 실제 설치된 곳
- `public/`
  - 서버 절대경로 기준. 현재는 `.svg` 파일만 담겨있음
- `src/`
  - 가장 중요한 곳. 실제 React 코딩의 대부분
- `.eslintrc.cjs`
  - eslint 설정파일
- `.gitignore`

    - git 커밋 시 제외할 리스트. 대표적으로 `node_modules/` 에 포함됨
- `index.html`
    - 실제 React 프로젝트가 작동하는 기준 HTML 파일
- `package-lock.json`
    - `package.json` 의 상세 내역. 700줄이 넘음
- `package.json`
    - 프로젝트에 관한 각종 정보를 요약해놓은 명세서
- `vite.config.js`
    - Vite 설정파일

`index.html` 파일을 먼저 살펴보자.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Vite + React</title>
  </head>
  <body>
    <div id="root"></div>
    <script type="module" src="/src/main.jsx"></script>
  </body>
</html>
```

이 파일은 React 가 동작하는 HTML 파일이다.

주목해야 할 부분은 `<div id="root"></div>` 한 줄이다. 이 곳이 React 의 영역이다. `index.html` 이 있고, 그 안에 가상의 영역에서 React 가 동작한다고 생각하면 되며, 실제 적용은 바로 아랫줄 `main.jsx` 에서 하게 된다.

우리 수업에서 `index.html` 은 Bootstrap, Google Fonts 등의 CDN 적용을 위해 방문하게 된다.

다음, `package.json` 파일이다.

```
{
 "name": "vite-project",
 "private": true,
```

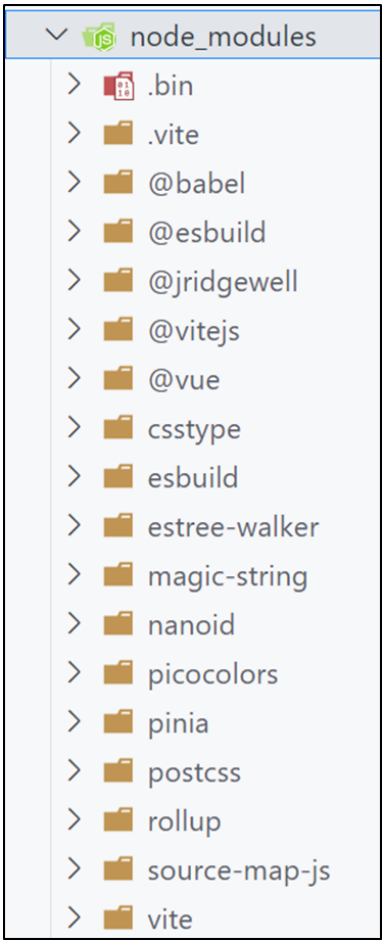
```
  "version": "0.0.0",
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "lint": "eslint src --ext js,jsx --report-unused-disable-directives --max-warnings 0",
    "preview": "vite preview"
  },
  "dependencies": {
    "react": "^18.2.0",
    "react-dom": "^18.2.0"
  },
  "devDependencies": {
    "@types/react": "^18.0.37",
    "@types/react-dom": "^18.0.11",
    "@vitejs/plugin-react-swc": "^3.0.0",
    "eslint": "^8.38.0",
    "eslint-plugin-react": "^7.32.2",
    "eslint-plugin-react-hooks": "^4.6.0",
    "eslint-plugin-react-refresh": "^0.3.4",
    "vite": "^4.3.9"
  }
}
```

프로젝트에 관한 각종 정보를 요약해놓은 명세서라고 할 수 있다.

- `name` : 패키지 이름
- `private` : 패키지 공개 여부. 명시적 목적.
- `version` : 패키지 버전
- `type` : 값이 `module` 일 시, `require` 를 사용하는 CommonJS 가 아니라 `import` 를 사용하는 ES Module 임을 의미
- `scripts` : 스크립트 명령
  - `npm run dev` : Vite 테스트 서버 동작
  - `npm run build` : 빌드
- `dependencies` : 매우 중요. 한국어로 "의존성" 이라고 하며, 이 패키지를 작동시키기 위해 필요한 다른 패키지 목록
- `devDependencies` : 이 패키지를 "개발할 때만" 필요한 패키지 목록이며, 배포 빌드 작업 시, 해당 목록은 빌드에서 제외됨

`package-lock.json` 이라는 파일이 별도로 존재하는데, 이것은 `package.json` 의 세부 명세서이며, 협업 시 반드시 넘겨줘야한다.

다음은 `node_modules/` 디렉터리이다.

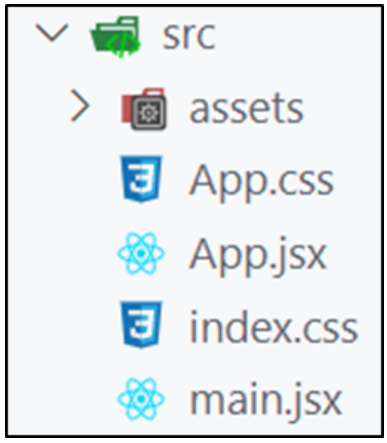


상당히 많은 패키지가 저장되어있음을 알 수 있다.

`package.json` 의 `dependencies` , `devDependencies` 패키지 목록은 실제론 `node_modules/` 디렉터리에 저장되어 있고, 초기 프로젝트 세팅 기준 약 92MB 의 용량이다.

`node_modules/` 디렉터리는 협업 시 반드시 `.gitignore` 처리해야한다. 프로젝트를 진행하면서 용량이 상당히 커지는데, 어차피 `package.json` 명세서와 `package-lock.json` 세부 명세서를 가지고 있기 때문에, 얼마든지 `npm install` 명령으로 명세에 기록된 패키지의 재설치가 가능하기 때문이다.

다음, 가장 중요한 `src/` 디렉터리이다.



React 프로젝트가 진행되는 디렉터리이다.

- `assets/` : 정적 이미지 파일을 저장하는 용도로 쓰임
- `App.css` : `App` 컴포넌트에 해당하는 CSS 파일
- `App.jsx` : 애플리케이션의 최상위 컴포넌트
- `index.css` : 프로젝트 전체에 적용되는 전역 CSS 파일
- `main.jsx` : 애플리케이션의 진입점이며, 기본 세팅 및 라이브러리 세팅 진행

`main.jsx` 파일을 살펴보면 다음과 같다.

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import App from './App.jsx'
import './index.css'

ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
)
```

이 파일은 애플리케이션의 진입점이자 기초 세팅이 진행되는 곳이다.

실전에선, 특정 패키지를 설치하고 세팅 코드를 작성할 때 종종 방문한다.

- ex) `react-bootstrap` , `react-router-dom` …

`import` 구문을 확인해보면,

- `React` , `ReactDOM` : React 를 사용하기 위한 가장 기본적인 객체
- `App` : 루트 컴포넌트

- `index.css` : 전역 CSS 파일

그리고 `index.html` 파일의 `<div id="root"></div>` 코드 한 줄에 React 를 적용시키는 구문이 보인다.

- 이런 방식을 VirtualDOM 이라고 하며, 잠시후에 살펴보자.

마지막으로, 루트 컴포넌트 파일인 `App.jsx` 파일을 살펴보자.

```
import "./App.css";
import { useState } from "react";

export default function App() {
  const [ message, setMessage ] = useState("Hello React");
  return (
    <>
      <div>message: { message }</div>
    </>
  );
}
```

`App` 컴포넌트는 모든 컴포넌트의 최상위 컴포넌트 역할을 한다. 기본적으로 모든 요청이 `App` 을 거쳐간다.

컴포넌트가 무엇인지 잠시 후 상세히 배우게 될 것이다.

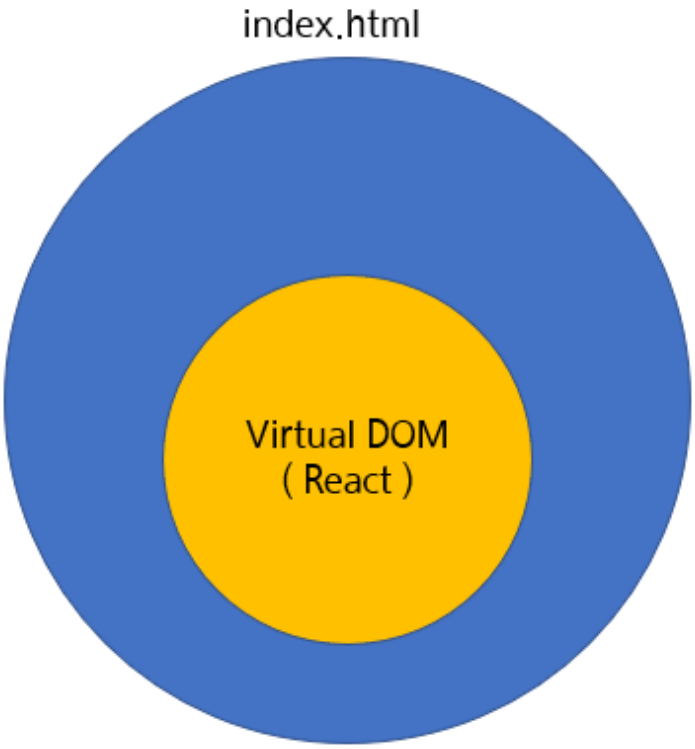
2. Virtual DOM

`index.html` 을 다시 한 번 살펴보자.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Vite + React</title>
  </head>
  <body>
    <div id="root"></div>
    <script type="module" src="/src/main.jsx"></script>
  </body>
</html>
```

여기서, `<div id="root"></div>` 한 줄이 React 의 영역이라는것은 이미 설명했다. HTML 은 DOM 객체로 이루어져 있고, 실제 DOM 을 Real DOM 이라고 하는데, 이 안에 React 만의 가상 DOM 이 올라가게 된다. 이를 Virtual DOM 이라고 한다.

도식화하면 다음과 같다.



Virtual DOM 은 화면 변경이 잦은 반응형 사이트에서 상당한 성능 향상을 보인다. 왜냐면, Real DOM 에서 화면이 변경된다면 변경될때마다 렌더링하게 되는데, Virtual DOM 에서는 모든 변경이 끝나고 맨 마지막에 딱 한번만 렌더링하기 때문이다.

※ 이를 비판하며 등장한 React 의 경쟁 프레임워크가 Svelte 이다. Svelte 는 Virtual DOM 이 더 빠르다는건 오로지 밈 (meme) 일 뿐이며, Virtual DOM 을 사용하지 않은 자신들의 프레임워크가 훨씬 빠르다고 주장한다.

https://svelte.dev/blog/virtual-dom-is-pure-overhead

어쨌든, React 의 기반이 Virtual DOM 이기 때문에, 반드시 지켜야 할 주의사항이 있다.

- Real DOM 직접 접근 금지
  - JavaScript에서 사용하는, DOM에 직접 접근하는 메서드 사용 금지

- ex) `querySelector`, `addEventListener`, `createElement`, `innerHTML` 등

간혹, Real DOM 에 접근할 일이 있다면 `useRef` 를 사용해야 하는데, 매우 특수한 상황이므로 우리 수업에선 다루지 않는다.

3. Component 소개

우선 다음 예시를 보자.

React 는 Meta 에서 만든 라이브러리이기 때문에, 자사 제품 인스타그램의 예시를 보는 것은 적절하다.

위 화면은 하나의 페이지이며, 하나의 HTML 문서이다. `index.html` 이라고 가정하자.

화면의 내용이 많아지면 많아질수록 코드의 양도 당연히 많아진다. 현재 `index.html` 은 약 5천줄 정도 된다고 가정하겠다.

여러분은 인스타그램에 입사한 신입 개발자다. 다음 두 가지 업무를 해야만 한다.

상황1: 하트 아이콘을 별 모양으로 변경

- `index.html` 을 연다.
- 몇 천줄의 코드에서 하트를 찾는다. (총 두 군데)
- 다른 아이콘으로 교체한다.

상황2: `<footer>` 태그의 새 아이콘 리스트 적용

- 디자이너는 신입 개발자에게 `<footer>` 태그의 새 아이콘 리스트 제공
- 다른 선배 개발자들은 메인 피드 영역 작업중
- 하나의 `index.html` 파일을 양쪽에서 작업 - 다소 위험!

이 두 가지 상황 모두, 몇 천줄의 코드를 다루는 상황이기에 위험한 건 똑같다.

만약, 하나의 HTML 파일을 여러개의 파일로 분할할 수 있다면 좀 더 안전하게 작업할 수 있지 않을까?

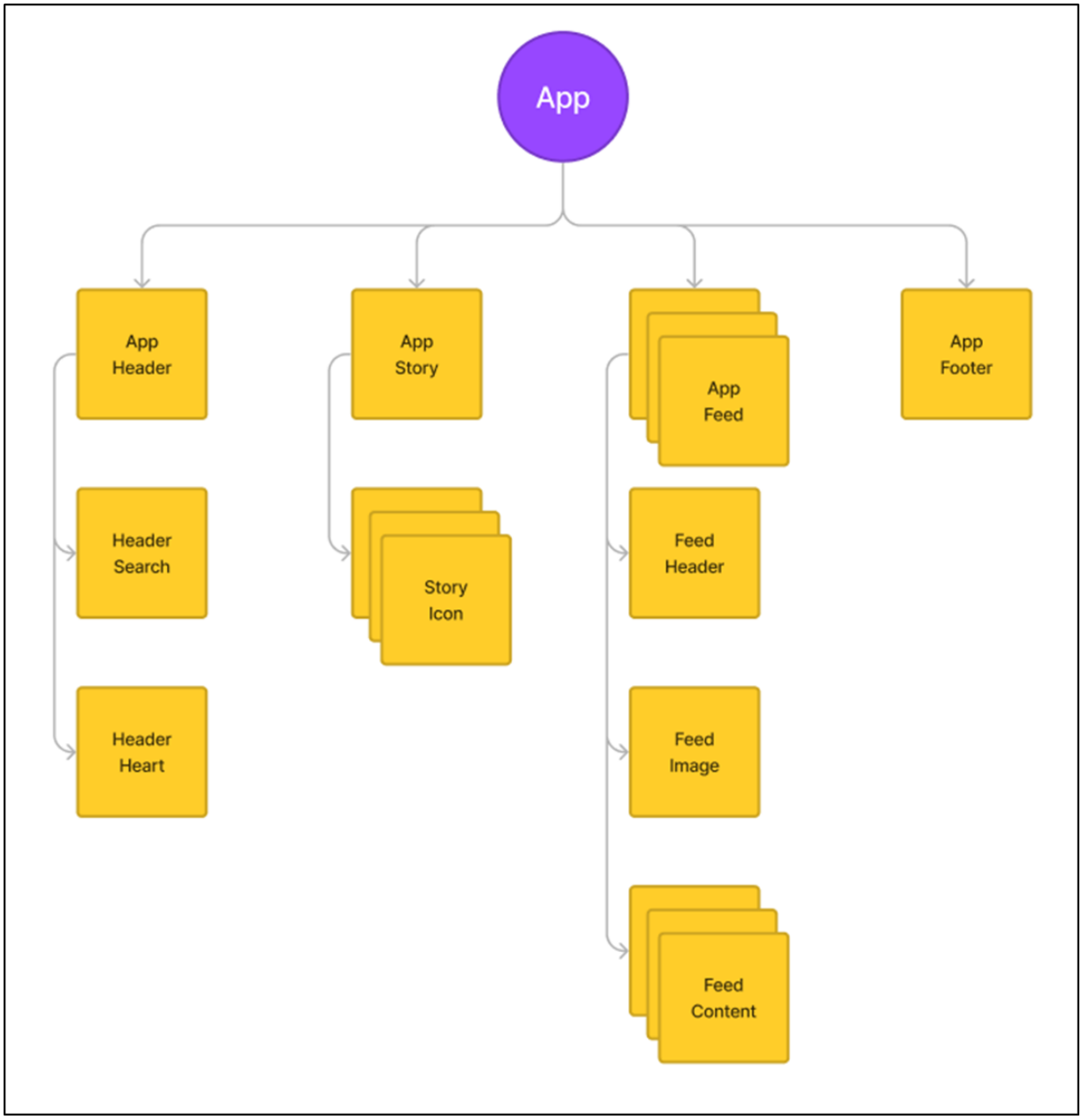
여기서 컴포넌트라는 개념이 등장한다. 컴포넌트 (component) 는 "부품" 이라는 뜻이다. 레고를 생각하면 된다. 레고를 가지고 놀 때 우리는 각각의 부품을 따로 만들고, 모아서 조립한다.

즉, `index.html` 각각의 컴포넌트는 필요에 따라 다수의 `.jsx` 파일로 나눌 수 있다. 예시 페이지에서 컴포넌트를 나눠보면 다음과 같다.

- `AppHeader`
- `AppStory`
- `AppFeed`
- `AppFooter`

그리고, 각각의 컴포넌트는 필요에 따라서 자식 컴포넌트를 둘 수 있다.

이 예제는 다음 도식으로 변경할 수 있다.



트리의 개별 요소는 개별 파일, 즉 `.jsx` 이며, 각각은 컴포넌트가 된다.

컴포넌트가 반복될 경우, 겹쳐서 표현했다. 여기선 `StoryIcon` , `AppFeed` , `FeedContent` 에 해당한다.

이제 앞에서 제시했던 두 가지 상황을 돌아보자.

상황1: 하트 아이콘을 별 모양으로 변경

- `HeartIcon.jsx` 파일을 연다.
- 하트를 별 모양으로 바꾼다.
- 다른 컴포넌트는 건드리지 않는다.

`HeartIcon` 라는 이름에서도 알 수 있듯이, 특정 부모에게 종속된 컴포넌트가 아니다. 따라서, 이 컴포넌트를 여러 곳에서 재사용할 수 있고, `HeartIcon` 변경 시 해당 컴포넌트를 사용하는 모든 부분은 하트모양에서 별 모양이 된다.

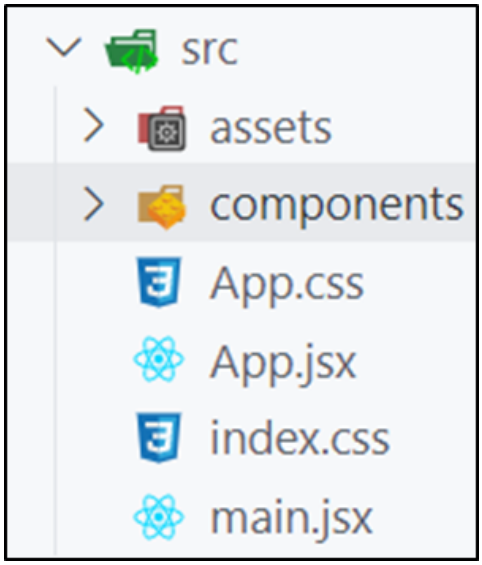
상황2: `<footer>` 태그의 새 아이콘 리스트 적용

- 디자이너는 신입 개발자에게 `<footer>` 태그의 새 아이콘 리스트 제공
- 다른 선배 개발자들은 메인 피드 영역 작업중
- 한쪽에선 `AppFooter.jsx` 작업
- 다른 한쪽에선 `AppFeed.jsx` 작업
- 별개의 파일을 작업하므로 안전

즉, 컴포넌트 방식으로 개발할 때, 유지보수와 협업에 유리해진다.

4. Component 구현

먼저, `src/` 디렉터리에 `components/` 디렉터리를 생성하자.



다음, `App.jsx` 를 아래 기본 코드로 변경한다.

```
import "./App.css";

export default function App() {
  return (
    <>
      <h1>App</h1>
    </>
  );
}
```

컴포넌트 구현은 총 3단계로 이루어진다.

1단계: 파일 생성

2단계: 자식 `import`

3단계: 부모에 붙이기

우선, 파일 생성이다.

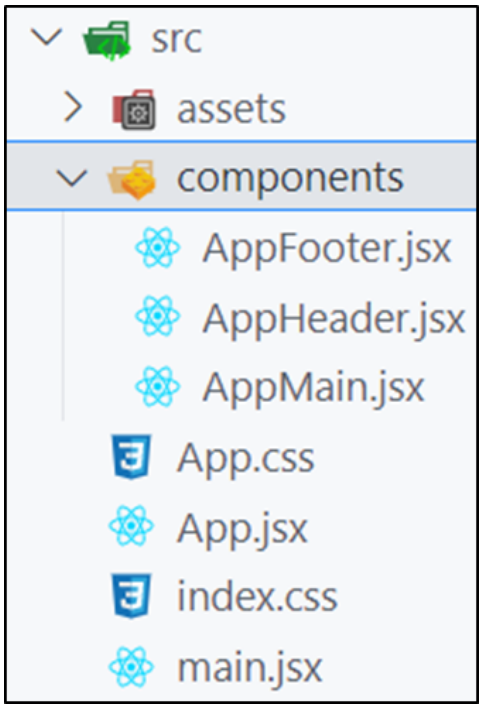
`App` 컴포넌트는 총 3개의 자식 컴포넌트를 가질 것이다.

`components/` 디렉터리에 다음 세 개의 파일을 제시된 양식으로 작성한다.

- `AppHeader.jsx`

- `AppMain.jsx`
- `AppFooter.jsx`

```
export default function 컴포넌트이름() {
  return (
    <>
      <h2>컴포넌트이름</h2>
    </>
  );
}
```



반드시 대문자로 시작해야 하며, 파일명과 컴포넌트 함수명은 일치해야한다.

2단계: 자식 `import`

`App` 컴포넌트를 다음과 같이 작성한다.

```
import "./App.css";
import AppHeader from "./components/AppHeader";
import AppMain from "./components/AppMain";
import AppFooter from "./components/AppFooter";

export default function App() {
  return (
    <>
      <h1>App</h1>
    </>
```

```
  );
}
```

자식 컴포넌트를 가져오되, 경로를 정확히 써주도록 하고, 확장자 `.jsx` 는 떼도 되고 붙여도 된다.

3단계: 부모에 붙이기

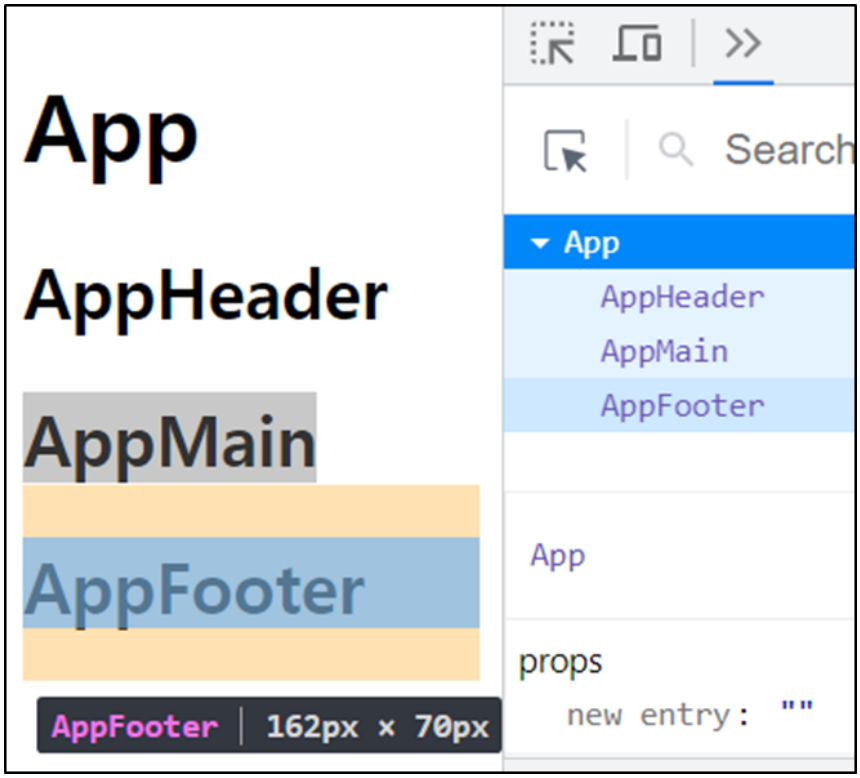
`App` 컴포넌트를 다음과 같이 작성한다.

```
import "./App.css";
import AppHeader from "./components/AppHeader";
import AppMain from "./components/AppMain";
import AppFooter from "./components/AppFooter";

export default function App() {
  return (
    <>
      <h1>App</h1>
      <AppHeader />
      <AppMain />
      <AppFooter />
    </>
  );
}
```

컴포넌트를 붙일 땐, HTML 태그와 구분되도록 PascalCase 를 사용하고, 닫는 기호 `/` 를 넣어준다.

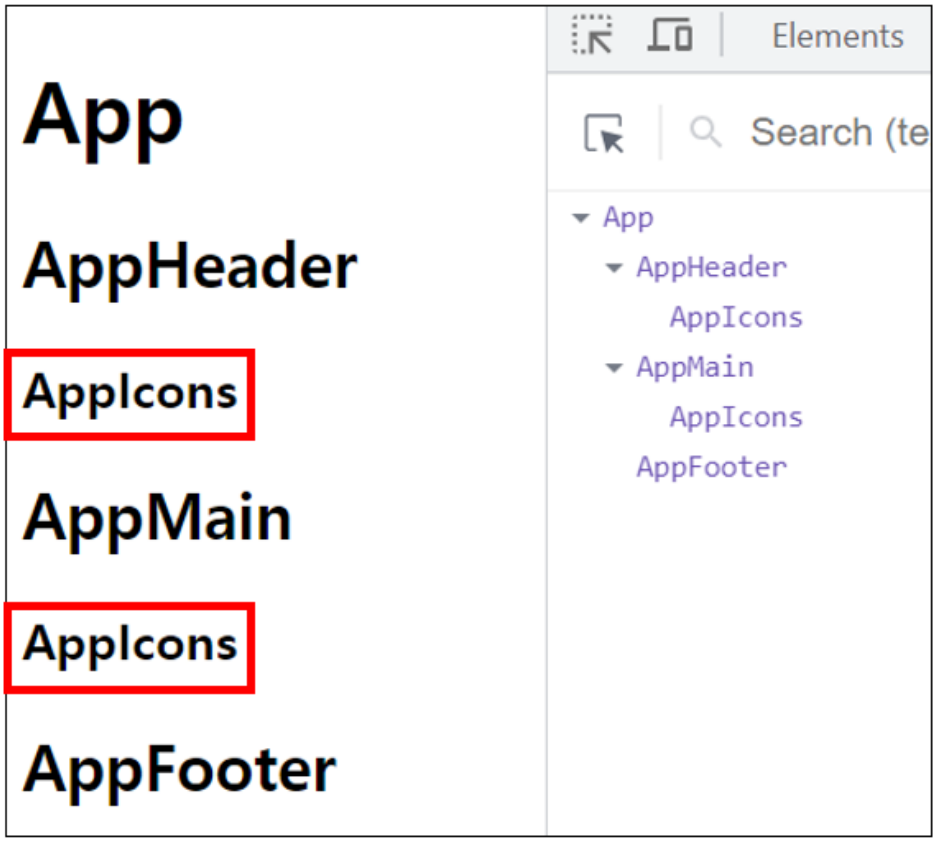
이제, React Developer Toos 를 개발자 도구로 열고, 부모 자식 관계가 잘 설정되었는지 확인해보자.



도전: 컴포넌트 재사용

`AppIcons` 컴포넌트 생성 후, 다음 두 컴포넌트에서 `import`

- `AppHeader`
- `AppMain`


App
AppHeader
AppIcons
AppMain
AppIcons
AppFooter
Elements
Search (te
App
AppHeader
AppIcons
AppMain
AppIcons
AppFooter