

LẬP TRÌNH REACT THẬT ĐƠN GIẢN

Từng bước tạo ứng dụng từ A-Z

By VNTALKING



MỤC LỤC

LỜI NÓI ĐẦU	4
NỘI DUNG CUỐN SÁCH.....	5
AI NÊN ĐỌC CUỐN SÁCH NÀY?	6
Yêu cầu trình độ	6
Cách học đúng cách	6
Liên hệ tác giả	7
GIỚI THIỆU	8
Ưu điểm của React.....	8
TỔNG QUAN REACT.....	11
Virtual DOM	11
DOM là gì?	11
Vấn đề của DOM là gì?	11
Cơ chế hoạt động của Virtual DOM	12
CÀI ĐẶT REACT	13
Cách 1: Viết React trực tiếp trong HTML	13
Cách 2: Sử dụng Create-React-App CLI.....	17
Cài đặt Code Editor và các extension cần thiết	20
XÂY DỰNG ỨNG DỤNG TODOS	26
Giới thiệu ứng dụng Todos	26
Những kiến thức React cơ bản nhất	27
Tạo các components.....	32
Tạo Header component và thêm Styles cho ứng dụng Todo	37
Thêm State	39
Tạo component hiển thị danh sách Todos	42
Controlled Component	46
Gửi và xử lý sự kiện - Handle Events.....	48
Cập nhật giá trị state sử dụng hàm setState().....	51
Sử dụng Destructuring.....	53

Xóa một Todo.....	54
Thêm một Todo mới.....	56
Cập nhật Todo mới vào danh sách.....	60
FETCHING DỮ LIỆU TỪ API	64
Lý thuyết cơ bản về Request Network	64
Giới thiệu API cung cấp cho ứng dụng Todo	66
Vòng đời Component trong React	67
Cài đặt axios.....	68
Lấy danh sách Todos bằng GET request	69
Thêm Todo bằng POST request	71
Xóa một Todo bằng DELETE request	72
REACT HOOKS.....	74
Tìm hiểu Hooks	74
Refactoring mã nguồn Todo App sử dụng Hooks	78
QUẢN LÝ STATE VỚI REDUX.....	83
3 nguyên tắc của Redux	83
Khi nào thì sử dụng Redux?.....	83
Thành phần của Redux	84
Thực hành sử dụng Redux trong Todo App	85
DEPLOY ỨNG DỤNG REACT	94
Deploy ứng dụng React lên Github Pages.....	95
BÀI TẬP.....	100
Project task	101
Đáp án bài tập tham khảo	103
KẾT NỐI VỚI VNTALKING	105
THÔNG TIN VỀ TÁC GIẢ.....	106
CUỐN SÁCH KHÁC CỦA VNTALKING	107

LỜI NÓI ĐẦU

Nói đến front-end là không thể không nhắc tới ReactJS. Một thư viện Javascript được chống lưng bởi ông lớn Facebook. Lướt qua các trang tuyển dụng lớn, bạn sẽ thấy nhu cầu tuyển người biết ReactJS rất lớn.

Bạn biết CMS nổi tiếng Wordpress không? Từ Wordpress 5.0, hẳn bạn đã nghe tới trình soạn thảo Gutenberg được tích hợp sẵn, nó cũng được xây dựng bằng React đó.

Thêm một ví dụ nữa, đó là Gatsby, một trình tạo trang web tĩnh nổi tiếng, một xu hướng mà nhiều nhà phát triển bắt đầu chuyển sang.

Và rất nhiều ứng dụng nổi tiếng khác cũng đang sử dụng React như: chotot, Shopee, Lazada, MyTV...

Điều này cho thấy rằng, việc lựa chọn ReactJS là một lựa chọn thông minh trong thời điểm hiện tại.

Với mục đích xây dựng nền tảng khi bắt đầu học ReactJS, mình cho ra đời cuốn sách này.

Trong cuốn sách này, bạn sẽ không chỉ học các nguyên tắc cơ bản của ReactJS, mà bạn còn có thể ứng dụng chúng để tự xây dựng các ứng dụng ReactJS hiện đại và triển khai miễn phí lên Internet.

Ngoài ra, trong cuốn sách, mình sẽ giới thiệu những kỹ thuật hỗ trợ bạn xây dựng ứng dụng ReactJS dễ dàng hơn. Ví dụ như quản lý mã nguồn với GIT, lựa chọn và cài đặt Code Editor phù hợp với ReactJS.v.v...

Nếu bạn là người thích sự chi tiết và tỉ mỉ thì cuốn sách này đích thị dành cho bạn.

Hãy tiếp tục đọc và nghiền ngẫm, bạn sẽ cảm thấy yêu thích cuốn sách này.

Mình đảm bảo!

NỘI DUNG CUỐN SÁCH

Chào mừng bạn đến với cuốn sách: "**Lập trình React thật đơn giản**". Mình tin chắc rằng, bạn được nghe không nhiều thì ít về ngôn ngữ lập trình web như HTML, Javascript, CSS.

Về cơ bản, để tạo một ứng dụng web (ám chỉ front-end), bạn chỉ cần biết 3 thứ: HTML, Javascript, CSS. Tất cả những công nghệ, thư viện, frameworks... dù đao to búa lớn thì cũng chỉ xoay quanh 3 ngôn ngữ đó mà thôi.

Thế giới web hiện đại ngày nay, chúng ta không chỉ đơn thuần xây dựng các website cung cấp thông tin một chiều mà còn có các ứng dụng web, khi mà người dùng có thể tương tác, thay đổi nội dung, thay đổi giao diện mà không phải tải lại trang (chính là các ứng dụng web dạng Single Page Applications - gọi tắt là SPA).

Và đây là mảnh đất để các thư viện như ReactJS tung hoành.

Trong cuốn sách này, bạn sẽ được học và thực hành:

- Cách cài đặt và bắt đầu làm việc với ReactJS (2 cách).
- Học các kiến thức nền tảng của React:
 - React component
 - Props và State
 - Class Component và Functional component
 - Truyền dữ liệu giữa các components.
 - Sử dụng Style trong ứng dụng React.
 - Xử lý sự kiện trong ứng dụng React.
 - React Hooks và cách sử dụng.
 - Redux - cách quản lý state.
- Cách lấy dữ liệu từ API và hiển thị lên ứng dụng React.
- Triển khai ứng dụng React lên Internet.
- Thực hành từng bước xây dựng ứng dụng Todos.
- Bài tập: Xây dựng ứng dụng Meme Generator.

Để đảm bảo các bạn tập trung và hiểu rõ những khái niệm, kiến thức nền tảng của ReactJS, mình sẽ không sử dụng bất kỳ thư viện 3rd nào khi xây dựng ứng dụng, ngoại trừ axios và redux.

AI NÊN ĐỌC CUỐN SÁCH NÀY?

Cuốn sách này phù hợp với tất cả những ai quan tâm đến việc phát triển ứng dụng ReactJS. Với những bạn đang định hướng sự nghiệp trở thành front-end developer thì cuốn sách này dành cho bạn.

Đây là cuốn sách "**No Experience Require**", tức là không yêu cầu người có kinh nghiệm ReactJS. Tất cả sẽ được mình hướng dẫn từ con số 0.

Yêu cầu trình độ

Do ReactJS là thư viện Javascript được xây dựng để tạo các ứng dụng web, nên sẽ tốt hơn nếu bạn đã có:

- Kiến thức cơ bản về HTML và CSS.
- Javascript cơ bản (Object, Arrays, điều kiện.v.v...)
- Đã biết tới Javascript ES6 (arrow function.v.v...)

Nếu bạn vẫn còn đang bối ngỡ với Javascript, cũng không sao! Đọc xong cuốn sách này bạn hiểu Javascript hơn.

Cách học đúng cách

Cuốn sách này mình chia nhỏ nội dung thành nhiều phần, mỗi phần sẽ giới thiệu một chủ đề riêng biệt, kèm thực hành. Mục đích là để bạn có thể chủ động lịch học, không bị dồn nén quá nhiều.

Với mỗi phần lý thuyết, mình đều có ví dụ minh họa và code luôn vào dự án. Vì vậy, cách học tốt nhất vẫn là vừa học vừa thực hành. Bạn nên **tự mình viết lại từng dòng code** và kiểm tra nó trên trình duyệt. Đừng copy cả đoạn code trong sách, điều này sẽ hạn chế khả năng viết code của bạn, cũng như làm bạn nhiều khi không hiểu vì sao code bị lỗi.

"Nhớ nhé, đọc đến đâu, tự viết code đến đó, tự build và kiểm tra đoạn code đó chạy đúng không"

Ngoài ra, cuốn sách này mình biên soạn theo mô hình: phần sau được xây dựng từ phần trước. Do vậy, bạn đừng đọc lướt mà bỏ sót đoạn nào nhé.

Tất cả mã nguồn trong cuốn sách sẽ được mình up lên Github:

<https://github.com/vntalking/ebook-reactjs-that-don-gian>

Trong quá trình bạn đọc sách, nếu code của bạn không chạy hoặc chạy không đúng ý muốn, bạn có thể kiểm tra và so sánh với mã nguồn của mình trên Github. Nếu vẫn không được thì đừng ngần ngại đặt câu hỏi trên Group: [Hỏi đáp lập trình - VNTALKING](#)

Liên hệ tác giả

Nếu có bất kỳ vấn đề gì trong quá trình học, code bị lỗi hoặc không hiểu, các bạn có thể liên hệ với mình qua một trong những hình thức dưới đây:

- Website: <https://vntalking.com>
- Fanpage: <https://facebook.com/vntalking>
- Group: <https://www.facebook.com/groups/hoidaplaptrinh.vntalking>
- Email: support@vntalking.com
- Github: <https://github.com/vntalking/ebook-reactjs-that-don-gian>

GIỚI THIỆU

ReactJS (đôi lúc cũng có thể gọi là React.JS, React - cũng đều nó cả) là một thư viện Javascript giúp bạn nhanh chóng xây dựng giao diện ứng dụng. React có thể xây dựng website hoàn toàn sử dụng Javascript (để thao tác với HTML), được tối ưu hiệu năng bởi kỹ thuật **VirtualDOM** (mình sẽ giải thích chi tiết ở phần dưới cuốn sách).

React được Facebook giới thiệu vào năm 2011, họ quảng bá đây là thư viện cho phép developer tạo các dự án ứng dụng web lớn, có giao diện UI phức tạp từ một đoạn mã nguồn nhỏ và độc lập.

Một số diễn đàn, React được coi là Framework vì khả năng và behaviors (hành vi) của nó. Nhưng về mặt kỹ thuật, nó chỉ là một thư viện. Không giống như Angular hay Vue được định nghĩa là một framework. Với React, bạn sẽ cần phải kết hợp sử dụng nhiều thư viện hơn để tạo thành một giải pháp hoàn chỉnh. Điều này khiến **React chỉ tập trung vào rendering** (kết xuất - nghĩa là xuất ra một thứ gì đó cho người dùng nhìn thấy) và đảm bảo nó được đồng bộ hóa với State của ứng dụng. Đó là tất cả những gì mà React thực hiện.

Đọc mấy khái niệm này có vẻ cao siêu khó hiểu nhỉ?

Đừng hoảng nhé!

Bạn sẽ hiểu rõ hơn khi bắt tay vào thực hành xây dựng ứng dụng React ở phần tiếp theo cuốn sách - cứ bình tĩnh nhé.

Ưu điểm của React

Để tìm một thư viện/framework front-end, bạn chỉ cần vào Google gõ nhẹ một cái là ra hàng tá luôn, có thể kể những cái tên đình đám như: Angular, Vue, React, JQuery, Emberjs.v.v...

Do đó, khi phải sử dụng React, chắc hẳn bạn sẽ luôn băn khoăn: *Thư viện React này có ưu điểm gì mà mình phải dùng?*

Ở đây, mình sẽ chia thành từng mục và đưa ra lý do ngắn gọn tại sao sử dụng React là cần thiết nếu bạn đang cân nhắc đến việc xây dựng ứng dụng web hiện đại.

Ok bắt đầu nhé.

1. Component độc lập - có thể tái sử dụng

Trong React, giao diện được xây dựng từ việc kết hợp các components. Bạn có thể hiểu đơn giản component là một hàm, nó nhận giá trị bạn truyền vào và trả về một số đầu ra.

Và cũng giống như function, component có thể tái sử dụng ở bất kỳ đâu. Vì vậy, chúng ta có thể tái sử dụng lại, hợp nhất các component để tạo thành giao diện người dùng phức tạp hơn.

Trong cuốn sách này, bạn sẽ biết cách làm thế nào xây dựng giao diện ứng dụng phức tạp, nhiều tầng, nhiều lớp thông qua component.

2. React làm cho front-end javascript dễ sử dụng hơn, nhanh hơn bằng cách sử dụng Virtual DOM

Khi bạn làm việc với vanilla Javascript, bạn sẽ phải tự làm mọi nhiệm vụ khi thao tác với DOM. Nhưng với React thì khác.

Với React, bạn chỉ cần thay đổi state của UI (bạn sẽ biết khái niệm này ở phần 4 của cuốn sách), và React sẽ tự động cập nhật DOM để phù hợp với state đó. Kỹ thuật được React sử dụng đó là Virtual DOM. Điều này cho phép React chỉ cập nhật những cái cần thiết (kiểu như một phần trang web) thay vì phải tải lại toàn bộ trang.

Việc sử dụng Javascript để tạo ra mã HTML cho phép React có một cây đối tượng HTML ảo — VirtualDOM. Khi bạn tải trang web sử dụng React, một VirtualDOM được tạo ra và lưu trong bộ nhớ.

Mỗi khi state thay đổi, chúng ta sẽ cần phải có một cây đối tượng HTML mới để hiển thị lên trình duyệt. Thay vì tạo lại toàn bộ cây, React sử dụng một thuật toán thông minh để chỉ tạo lại các thành phần khác biệt giữa cây mới và cây cũ. Bởi vì cả hai cây HTML cũ và mới đều được lưu trong bộ nhớ, quá trình xử lý này diễn ra siêu nhanh.

3. Dễ dàng làm việc nhóm

Nếu bạn đang làm việc theo nhóm, thì React là thư viện tuyệt vời. Bởi vì, việc chia các task, chia màn hình ứng dụng sẽ rất đơn giản, mỗi người sẽ được chỉ định làm một thành phần. Các thành phần này độc lập với nhau, việc ghép nối các thành phần được thực hiện dễ dàng.

4. React được đảm bảo bởi Facebook

Khi bạn lựa chọn một thư viện/framework, việc đầu tiên là phải xem nó được bảo chứng bởi ai? Bởi vì một thư viện/framework không được bảo chứng bởi một tổ chức, công ty uy tín, nó có thể sẽ không được phát triển lâu dài, không được mở rộng hoặc maintain khi thư viện/framework phát sinh bugs...

Với React, bạn hoàn toàn yên tâm, vì nó được Facebook chống lưng, theo đó là một cộng đồng React tuyệt vời. Ngoài ra, tài liệu hướng dẫn chính chủ cũng rất chi tiết, đầy đủ.

Nói chung, về nguồn gốc, cộng đồng, tài liệu, bạn hoàn toàn có yên tâm gửi gắm dự án của mình cho React.

5. Nhu cầu tuyển dụng cao

Như mình đã đề cập ở trên, React là một kỹ năng web có nhu cầu tuyển dụng cao ở thời điểm hiện tại.

Hầu như nhà tuyển dụng nào khi cần tuyển một front-end developer đều yêu cầu phải biết React. Vì vậy, việc thành thạo React là một điểm sáng trong CV của bạn, cơ hội việc làm sẽ rộng mở hơn rất nhiều.

Ngoài ra, React có ít API hơn so với các thư viện/framework khác, do vậy việc học React cũng dễ hơn, đặc biệt nếu bạn đã thành thạo Javascript.

Trên đây là những ưu điểm của React mà mình tổng hợp và hệ thống lại. Giờ là lúc chúng ta cùng nhau chinh phục React thôi.

Let's go!

TỔNG QUAN REACT

Khi bạn bắt đầu tìm hiểu bất kỳ một nền tảng, thư viện hay framework nào đó, việc đầu tiên là bạn cần hiểu tư duy/triết lý của người viết ra nó. Hay nói một cách khác, bạn cần cùng hướng nhìn với tác giả của thư viện đó, khi mà cùng "hệ quy chiếu" thì việc học sẽ dễ dàng hơn.

Với React, một trong những thứ mà bạn cần hiểu đầu tiên, đó là: Virtual DOM.

Virtual DOM

Hiểu được cách thức hoạt động của DOM, bạn sẽ nhanh chóng nắm bắt được concept đằng sau Virtual DOM.

Nếu bạn là một Javascript developer, đã từng xây dựng một trang web thì hẳn bạn đã tương tác với DOM. Tuy nhiên, mình vẫn muốn nhắc lại cách thức hoạt động của nó để chúng ta cùng có một hướng nhìn.

DOM là gì?

DOM (viết tắt của Document Object Model) là một giao diện lập trình ứng dụng, cho phép Javascript hoặc các một loại script khác đọc và thao tác nội dung của document (trong trường hợp này là HTML).

Bất cứ khi nào một HTML document được tải xuống trình duyệt dưới dạng một trang web, một DOM sẽ được tạo cho trang đó.

Theo cách này, Javascript có thể thao tác với DOM như thêm, sửa, xóa nội dung... hoặc thực hiện một hành động nào đó như ẩn/hiện một view.

Vấn đề của DOM là gì?

Một số developers tin rằng, việc thao túng DOM là không hiệu quả, có thể gây chậm ứng dụng. Đặc biệt là khi bạn thường xuyên phải cập nhật DOM.

Vấn đề phải cập nhật DOM sẽ còn phức tạp hơn nhiều với các ứng dụng dạng Single Page Applications (SPA). Khi mà DOM chỉ được tải về trình duyệt một lần, sau đó Javascript sẽ phải cập nhật DOM liên tục mỗi khi người dùng thao tác với ứng dụng.

Vấn đề ở đây là không phải do DOM chậm (bởi vì DOM chỉ là một object). Ứng dụng bị chậm là do trình duyệt phải xử lý khi DOM thay đổi. Mỗi khi DOM thay đổi, trình duyệt sẽ phải tính toán lại CSS, chạy bố cục và render lại trang web.

Như một logic dễ hiểu, để tăng tốc ứng dụng, chúng ta cần tìm cách giảm thiểu thời gian khi trình duyệt render lại trang. Đó là lý do mà nhà phát triển React nghĩ tới Virtual DOM.

Cơ chế hoạt động của Virtual DOM

Nói chung, Virtual DOM có cái gì đó giống với cơ chế Cache vậy. Mặc dù không hoàn toàn. Tức là, React sẽ làm việc trên Virtual DOM (được lưu trong bộ nhớ), nó đảm bảo DOM thật chỉ nhận những thứ cần thiết mà thôi.

Cơ chế hoạt động của Virtual DOM cơ bản như sau:

- Bất cứ khi nào có yếu tố mới được thêm vào UI, một Virtual DOM sẽ được tạo.
- Khi state của bất cứ yếu tố nào thay đổi, React sẽ cập nhật Virtual DOM trong khi vẫn giữ phiên bản Virtual DOM trước để so sánh và tìm ra đối tượng Virtual DOM thay đổi. Khi tìm được sự thay đổi, React chỉ cập nhật đối tượng đó trên DOM thật.

Không giống như khi làm việc với vanilla JavaScript, việc cập nhật sẽ được React làm tự động. Tất cả công việc của bạn là thay đổi state của UI, phần còn lại cứ để React lo.

Bất cứ khi nào state thay đổi, React sẽ lập tức có hành động để cập nhật DOM, đó là lý do người ta gọi React là thư viện reactive.

CÀI ĐẶT REACT

Giống như mọi công nghệ web khác, bạn cần phải thiết lập môi trường phát triển trước khi có thể làm việc với React.

Yêu cầu môi trường phát triển:

- **Ubuntu v18.04:** Bạn có thể viết ứng dụng React trên Window hoặc Mac OS cũng được. Nhưng trong cuốn sách này mình sử dụng Ubuntu 18.04, một phần do mình quen dùng OS này rồi, một phần là khi đi làm ở công ty, các bạn cũng chủ yếu dùng Ubuntu để lập trình.
- **Node.js v12.18.3:** Các bạn cứ cài bản Node.js LTS mới nhất nhé. Cách cài đặt thì mình không trình bày trong cuốn sách này vì nó phổ biến quá rồi. Nếu cần có thể tham khảo lại trong bài viết này của mình: [Hướng dẫn cài đặt Node.js chi tiết](#)
- **NPM v6.14.6:** Là công cụ hỗ trợ quản lý và cài đặt các dependencies.
- **create-react-app:** Là công cụ giúp tạo dự án React mới nhanh chóng (mình sẽ trình bày về công cụ này ở phần dưới cuốn sách).

Có 2 cách để làm việc với React phổ biến nhất:

- Nhúng ngay thư viện React trong HTML.
- Sử dụng công cụ *create-react-app* (được tác giả khuyến khích sử dụng).

Mình sẽ hướng dẫn chi tiết cả hai cách sử dụng React này, bạn thấy cách nào phù hợp thì chọn lấy một cái cho dự án của mình nhé.

Cách 1: Viết React trực tiếp trong HTML

Với cách viết React này, bạn sẽ không cần phải cài đặt thêm bất cứ công cụ nào khác. Rất thuận tiện, đặc biệt cho mục đích thử nghiệm.

Đầu tiên, chúng ta sẽ tạo một tệp *index.html*. Trong đó, chúng cần khai báo và tải 3 thư viện cần thiết cho React gồm: *React*, *ReactDOM* và *Babel*.

Phần *<body>*, chúng ta tạo một thẻ *div* và gán *id* là "root".

Cuối cùng là tạo thẻ `<script>` nơi mà chúng ta sẽ viết code React.

Nội dung tệp `index.html` như sau:

```
<!DOCTYPE html>
<html lang="vi">
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
  <title> Phần 3 - Cài đặt ReactJS - cách 1</title>
  <script src="https://unpkg.com/react@16/umd/react.development.js"></scrip
t>
  <script src="https://unpkg.com/react-dom@16/umd/react-
dom.development.js">
  </script>
  <script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
</head>
<body>
  <div id="root"></div>
  <script type="text/babel">
    const element = <h1>VNTALKING: Hello React</h1>;
    console.log(element);
    ReactDOM.render(element, document.getElementById("root"));
  </script>
</body>
</html>
```

Khi bạn mở tệp `index.html` (hoặc truy cập link xem live tại đây:

<https://codesandbox.io/s/quizzical-shockley-k7v1o>), bạn sẽ nhận kết quả như dưới đây:



Hình 3.1: Ứng dụng React: hello world

Trong nội dung của tệp `index.html`, chúng ta tập trung vào thẻ `<script>`. Với thuộc tính `type`, chúng ta bắt buộc sử dụng Babel (mình sẽ nói kỹ hơn ở phần ngay dưới đây).

Bên trong thẻ `<script>`, bạn nhìn các phần tử trông có vẻ giống như là HTML.

```
const element = <h1> VNTALKING: Hello React </h1>;
```

Đến đoạn này, chắc hẳn bạn sẽ tự hỏi: *Tại sao chúng ta lại viết HTML trong mã Javascript?*

Câu trả lời là: thực ra đó không phải là HTML đâu, mà đó là JSX.

JSX là gì

JSX (viết tắt của cụm từ JavaScript XML) là một loại cú pháp mở rộng dành cho ngôn ngữ JavaScript viết theo kiểu XML. Nó transform cú pháp dạng gần như XML thành Javascript. Giúp người lập trình có thể code React bằng cú pháp của XML thay vì sử dụng Javascript. Các XML elements, attributes và children được chuyển đổi thành các đối số truyền vào `React.createElement`.

Tất nhiên, sau tất cả thì mã lệnh viết bằng JSX phải được chuyển sang JavaScript để trình duyệt có thể hiểu được (đó là nhiệm vụ của babel - không phải của bạn ^^).

Cơ chế làm việc của JSX như sau:

Trên thực tế, các trình duyệt ngay cả các trình duyệt mới nhất cũng không hỗ trợ cú pháp của JSX. Do đó mã nguồn sử dụng JSX cần được chuyển về JavaScript thông qua một thư viện có tên là Babel (một JavaScript compiler).

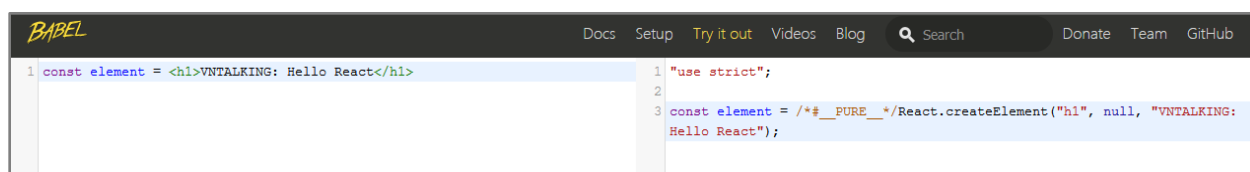
Ví dụ đoạn code sau:

```
const element = <h1>VNTALKING: Hello React</h1>;
```

Sau khi được biên dịch sang JavaScript sẽ tương đương với:

```
const element = React.createElement("h1", null, "VNTALKING: Hello React");
```

Bạn có thể kiểm tra trình biên dịch Babel chuyển đổi code như thế nào qua link: <https://babeljs.io/repl> (copy đoạn mã JSX vào khung bên trái là được).



Hình 3.2: Trình biên dịch Babel online

Nhìn đoạn code chuyển đổi trên, chúng ta có thể thấy là sử dụng code JSX để miêu tả UI có vẻ đọc dễ hiểu hơn hẳn là viết code thuần React đúng không?

Ngoài ra, bạn cần lưu ý những điều sau:

- Bạn có thể sử dụng biểu thức Javascript hợp lệ bên trong JSX thông qua dấu ngoặc nhọn `{ }`.
- Trong JSX, các attributes, event handlers luôn sử dụng quy tắc đặt tên **camelCase**. Ngoại trừ một vài ngoại lệ như *aria-** and *data-* attribute* là chữ thường.

Giờ thì bạn đã hiểu lý do tại sao chúng ta sử dụng JSX và tải Babel trong đoạn mã *hello-world* rồi đúng không?

Giờ thì quay trở lại đoạn code *hello-world* nhé. Bạn mở tệp *index.html* và mở *DevTools*, chuyển sang thẻ *console*.



Hình 3.3: Nội dung đối tượng element

Đối tượng này là output của JSX expression, là một React element và cũng là một phần Virtual DOM.

Để element này hiển thị, chúng ta cần render ra cây DOM thật.

Hàm *render()* có 2 tham số cần truyền vào:

- Tham số đầu tiên xác định những thứ bạn muốn render.
- Tham số thứ 2 xác định nơi bạn muốn render.

Cách 2: Sử dụng Create-React-App CLI

Thay vì phải tải thủ công các thư viện React, Babel... và import trong head của tệp html, bạn có thể thiết lập môi trường React bằng cách sử dụng công cụ *create-react-app*.

Công cụ này sẽ tự động cài đặt React và các thư viện third-party cần thiết như: Webpack, Babel.

Để cài đặt và sử dụng *create-react-app*, bạn cần phải cài sẵn Nodejs và NPM (node package manager) trong máy tính. .



Nodejs là Javascript runtime dùng để phát triển ứng dụng cho server. Nếu bạn muốn học NodeJS để trở thành full stack engineer thì có thể đọc cuốn: [Nodejs thật đơn giản](#) do VNTALKING thực hiện.

Nếu bạn không chắc chắn máy tính của mình đã có NodeJS chưa thì có thể kiểm tra bằng cách vào terminal và gõ lệnh:

`node -v`

và

`npm -v`

Kết quả được như hình bên dưới là được.

```
sonduong@sonduong-PC: ~  
File Edit View Search Terminal Help  
sonduong@sonduong-PC:~$ node -v  
v8.10.0  
sonduong@sonduong-PC:~$ npm -v  
6.13.7  
sonduong@sonduong-PC:~$
```

Hình 3.4: Kiểm tra phiên bản của Nodejs và NPM

Nếu máy tính bạn chưa có sẵn NodeJS và NPM thì bạn có thể tham khảo bài viết hướng dẫn cài đặt chi tiết tại đây: [Cài đặt Nodejs trên Window và Ubuntu](#)

Đầu tiên, bạn cần cài đặt công cụ *create-react-app*:

`$ sudo npm install -g create-react-app`

Sau đó, chúng ta sẽ tạo một ứng dụng React (ví dụ có tên là my-todo-app) như sau:

`$ npx create-react-app my-todo-app`

Bạn có thể đặt tên khác cho dự án, miễn là tất cả ký tự là chữ thường và không có dấu cách.

Để chạy thử ứng dụng React, vẫn ở cửa sổ lệnh, bạn chuyển con trỏ vào thư mục *my-todo-app* và chạy lệnh *start*

```
cd my-todo-app
```

```
npm start
```

```
sonduong@sonduong-PC: ~/VNTALKING/reactjs/reactjs-that-don-gian/chap2/cach2/my-todo-app
File Edit View Search Terminal Help
Compiled successfully!

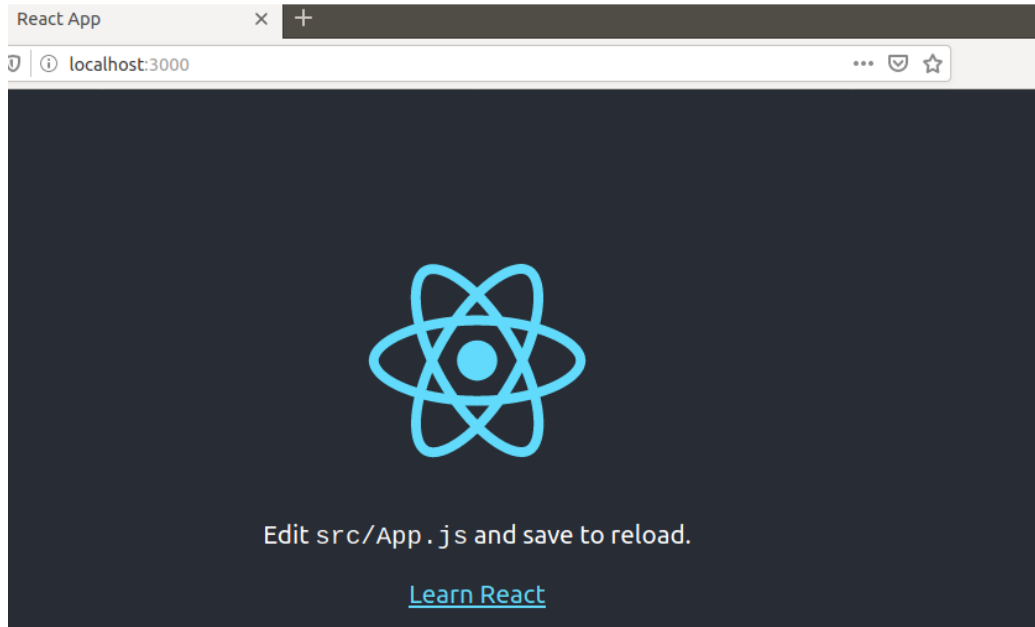
You can now view my-todo-app in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://10.0.2.15:3000

Note that the development build is not optimized.
To create a production build, use npm run build.
```

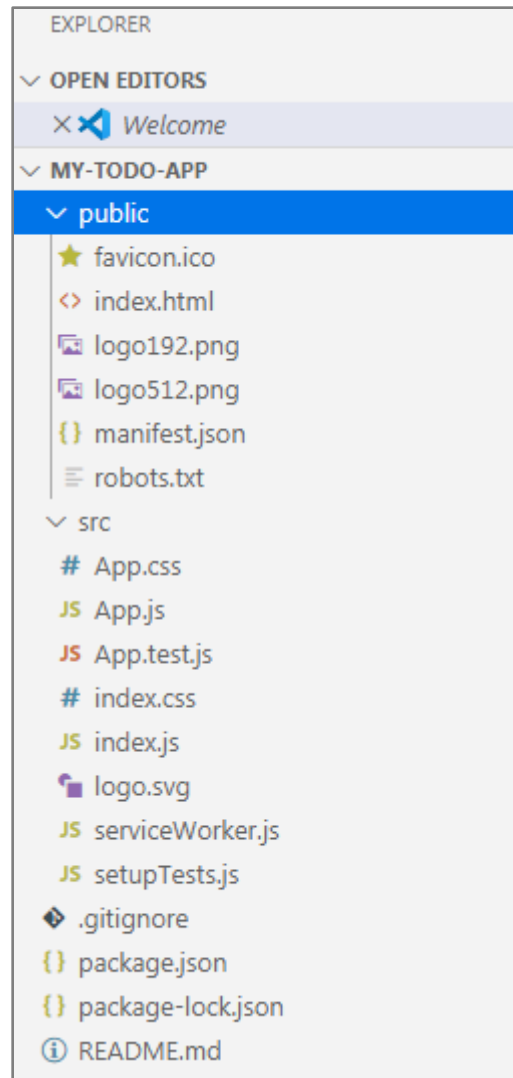
Hình 3.5: Kết quả khi chạy *npm start* thành công

Sau đó bạn vào trình duyệt truy cập vào địa chỉ: <http://localhost:3000>, tận hưởng thành quả:



Hình 3.6 Giao diện mặc định ứng dụng khi tạo bởi *create-react-app CLI*

Còn đây là cấu trúc thư mục được tạo bởi *create-react-app*.



Hình 3.7: Cấu trúc thư mục dự án React

Trong đó:

- **public:** Thư mục chứa tất cả tài nguyên tĩnh (static files) sử dụng trong ứng dụng như: ảnh, css... Những tài nguyên mà trình duyệt có thể truy cập một cách trực tiếp.
- **src:** thư mục này chứa toàn bộ mã nguồn của dự án, là nơi mà bạn sẽ làm việc chính.
- **package.json:** Đây là tệp cấu hình của dự án, nơi bạn khai báo tên ứng dụng, version, các dependencies cần sử dụng.v.v...

Cài đặt Code Editor và các extension cần thiết

Trước đây, khi mới bắt đầu học lập trình, mình cũng hay hỏi nên dùng IDE nào, dùng công cụ gì để viết code... Có rất nhiều tiền bối đưa ra lời khuyên là viết code bằng notepad, sang lắm thì Notepad++ và rất tự hào về khả năng viết "chay" của họ.

Nhưng quả thật, với quan điểm cá nhân mình, khi đã công cụ hỗ trợ thì chẳng tội gì không sử dụng cả. Việc viết code "chay" không làm bạn giỏi hơn, nó chỉ làm bạn chậm đi mà thôi.



Viết code "chay" tức là viết code như viết văn bản. Không sử dụng công cụ hỗ trợ cho viết code, không có các tính năng thiết yếu như auto suggestion, highlight code, hỗ trợ compile... Ví dụ sử dụng Notepad, MS Word, giấy... để viết code gọi là viết code "chay".

Do đó, thay vì viết code "chay", chúng ta nên sẽ sử dụng IDE hoặc Code editor để tăng hiệu suất công việc.

Với Javascript nói chung, React nói riêng, có rất nhiều Code editor tốt mà lại miễn phí như:

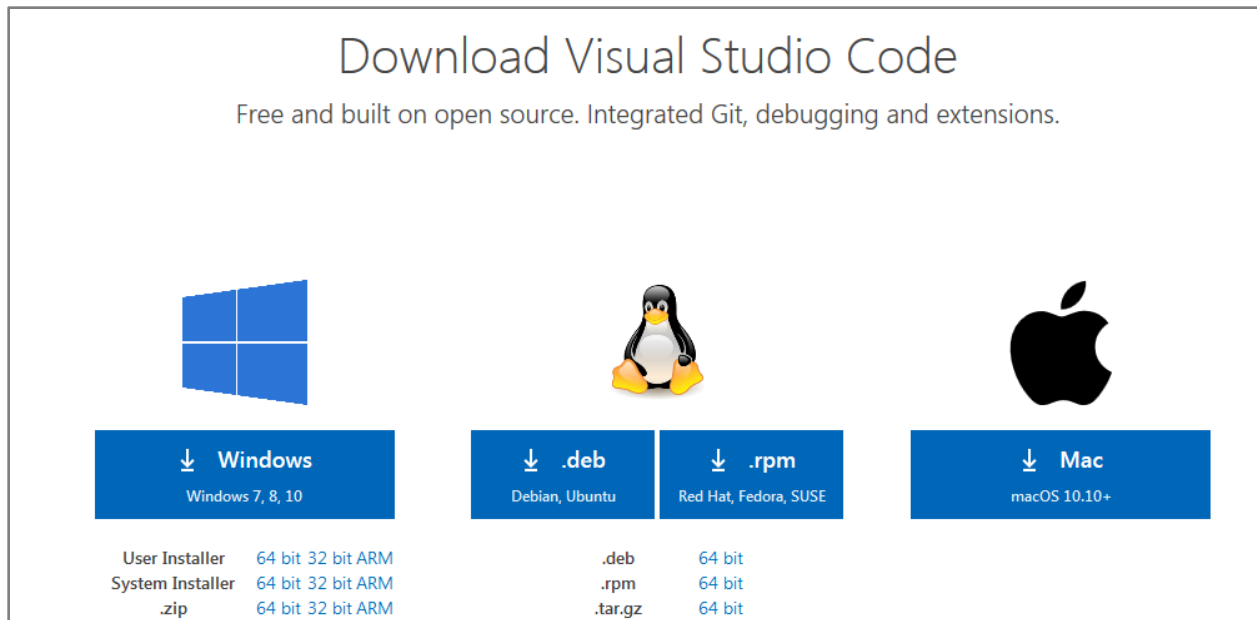
- Coda (Mac Os)
- Atom (cross-platform)
- Visual studio code (cross-platform)
- [Sublime Text](#) (cross-platform)

Không có phần mềm nào là tốt nhất, chỉ có phần mềm nào khiến bạn cảm thấy thoải mái và quen dùng nhất mà thôi. Bạn có thể tùy ý lựa chọn cho riêng mình.

Cá nhân mình thường sử dụng Visual Code để viết code. Đây là phần mềm Code editor do Microsoft phát triển, nhẹ mà lại có nhiều tính năng hay ho. Đặc biệt Visual Code có kho extensions rất phong phú, hoàn toàn miễn phí.

Việc cài đặt Visual Code rất đơn giản, cũng giống như các phần mềm bình thường khác. Đầu tiên, bạn vào trang chủ để tải bản mới nhất:

<https://code.visualstudio.com/download>



Hình 3.8: Trang tải bộ cài đặt visual code mới nhất

Nếu bạn không chắc chắn hệ điều hành mình đang dùng là 32-bit hay 64-bit thì cứ chọn bản 32-bit mà cài đặt.



Bạn có thể tham khảo hướng dẫn chi tiết bằng hình ảnh cách cài đặt Visual Code tại bài viết của mình tại đây: [Hướng dẫn download và cài đặt Visual Code chi tiết](#)

Cài đặt extensions cần thiết

Sau khi cài đặt thành công, việc tiếp theo là cài một số extension cần thiết cho việc viết code React.

1. Emmet

Extension này giúp tăng hiệu suất viết code khi bạn cần tạo các thẻ HTML và CSS. Nó sử dụng các từ viết tắt để xác định các thẻ HTML hay CSS mà bạn muốn tạo.

Mình sẽ ví dụ cách sử dụng extension này.

Đầu tiên, bạn tạo một tệp .html (*index.html* chẳng hạn). Sau đó gõ đoạn lệnh sau trong thẻ <body>

```
ul>li>img+p
```

Gõ xong thì ấn phím TAB để emmet tự động generate ra đoạn mã HTML, kết quả sẽ được như sau:

```
<ul>
  <li>
    <img src="" alt="">
    <p></p>
  </li>
</ul>
```

Cũng đơn giản, dễ hiểu phải không? Bạn có thể tham khảo syntax của emmet tại cheat sheet này: <https://docs.emmet.io/cheat-sheet>

Emmet hỗ trợ rất nhiều loại files như: scss, sass, less, stylus, jsx, xml, xsl, haml, jade, slim.

Trong danh sách này, chúng ta tập trung vào tệp jsx. Vì React sử dụng JSX để viết code xây dựng giao diện (UI). Bạn có thể hiểu JSX như là một extension của Javascript vậy, nên các tệp viết code JSX có thể có đuôi mở rộng là .jsx hoặc .js đều được.

React khuyến cáo nên sử dụng đuôi mở rộng .js. Tuy nhiên, mặc định Emmet thì lại không hỗ trợ JSX khi tệp có đuôi mở rộng là .js

Do vậy, bạn cần phải thay đổi setting để kích hoạt nó!

Cách làm đơn giản là bạn vào setting của Visual Code (File > Preferences > Settings, mở *setting.json*) và thêm đoạn thiết lập sau:

```
"emmet.includeLanguages": {
  "javascript": "javascriptreact"
}
```

Lưu lại là xong. Đây là setting.json trên máy tính của mình:



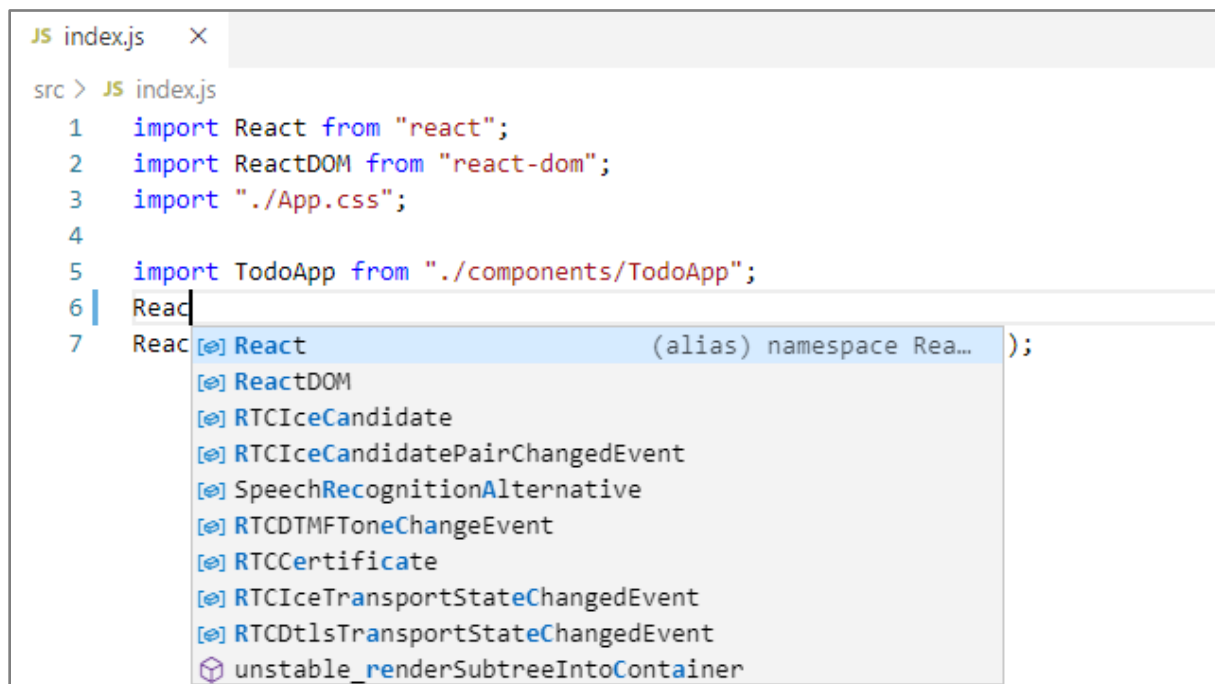
```
C: > Users > AnhSon > AppData > Roaming > Code > User > {} settings.json > ...
1 {
2   "files.autoSave": "afterDelay",
3   "git.autofetch": true,
4   "workbench.colorTheme": "Visual Studio Light",
5   "git.enableSmartCommit": true,
6   "emmet.includeLanguages": {
7     "javascript": "javascriptreact"
8   }
9 }
```

Hình 3.9: Thiết lập Emmet

2. IntelliSense

Đây là extension mà mình thích nhất, giúp mình tiết kiệm rất nhiều thời gian để viết code. Nó dựa trên ký tự bạn gõ mà tự động gợi ý những hàm, class, biến... liên quan tới từ bạn đang nhập.

Như hình bên dưới, khi mình gõ từ react, một danh sách đề xuất được hiển thị.



Hình 3.10: Cách IntelliSense extension hoạt động



Mặc định thì Visual Code hỗ trợ Intellisense cho HTML, CSS, Sass, Less, TypeScript, JSON và Javascript. Còn lại những ngôn ngữ khác thì bạn phải cài đặt extension riêng cho chúng

3. Prettier – code formatter

Đúng như tên gọi của nó, extension này sẽ giúp bạn tự động format mã nguồn theo đúng chuẩn style, nhìn mã nguồn gọn gàng hơn.

4. Es7 React/Redux/GraphQL/React-Native snippets

Với extension này, hiệu suất công việc tăng lên đáng kể khi nó tự động generate các React component khi bạn sử dụng phím tắt.

Cài đặt React Developer Tools (React DevTools)

Debug là một kỹ năng mà bất kỳ developer nào cũng phải có. Mình chưa từng gặp developer nào mà không biết debug cả.

Với React, bạn có một công cụ được phát triển dành riêng cho việc debug ứng dụng React. Đó là **React Developer Tools**.

Công cụ này được cung cấp dưới dạng extension cho Firefox và Chrome.

Bạn có thể cài đặt chúng ta tại đây:

- **Chrome:** <https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi>
- **Firefox:** <https://addons.mozilla.org/en-US/firefox/addon/react-devtools/>

Việc cài đặt cũng rất đơn giản, như mọi extension khác của Firefox hay Chrome thôi. Sau khi cài xong, bạn cũng không cần phải thiết lập gì thêm, cứ thế mà dùng.

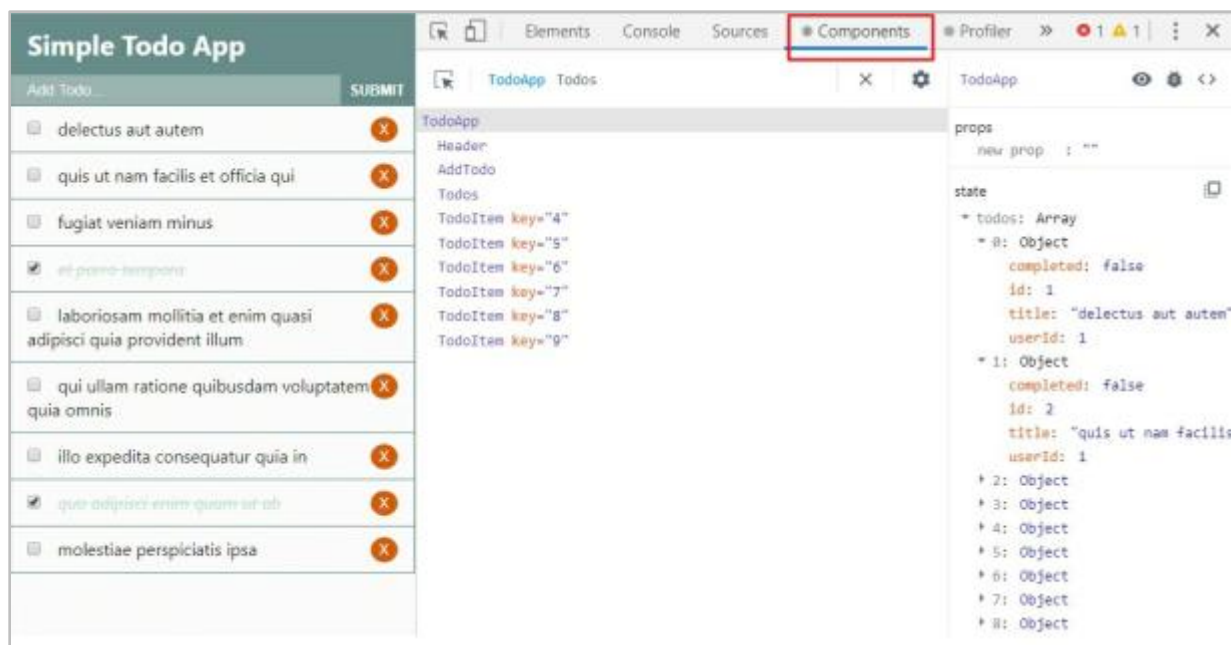
Ngoài ra, bạn có thể cài React DevTools bằng NPM thông qua câu lệnh:

```
$ npm install -g react-devtools@^4
```

Sử dụng React DevTools

Bạn mở ứng dụng web được viết bằng React trên trình duyệt. Bạn chuột phải vào bất kỳ vùng nào trên trang web, chọn *Inspect* (Chrome) hoặc *Inspect Element* (Firefox).

Sau đó bạn chọn tab: *Components* để mở *React DevTools*. Tại đây bạn sẽ nhìn thấy tất cả các thông tin cần thiết phục vụ cho việc debug ứng dụng React.



Hình 3.11: Giao diện React DevTools

Như vậy là chúng ta đã hoàn thành xong phần môi trường phát triển. Phần tiếp theo, chúng ta sẽ bắt tay vào học React và thực hành ngay trên dự án: Todos App

Tổng kết

Qua phần này, chúng ta đã biết cách cài đặt môi trường và khởi tạo dự án React. Trong các dự án thực tế, người ta sẽ ưu tiên cách tạo dự án bằng công cụ *create-react-app*.

Ngoài ra, chúng ta cũng biết tới Code editor mạnh mẽ Visual Code và các extensions hữu ích, tăng năng suất làm việc đáng kể.

Các bạn có thể tham khảo mã nguồn của phần này tại đây:

<https://github.com/vntalking/ebook-reactjs-that-don-gian/tree/master/phan3>

Nếu bạn có bất kỳ thắc mắc hoặc chỗ nào chưa hiểu, đừng ngại liên hệ với mình qua:

support@vntalking.com.

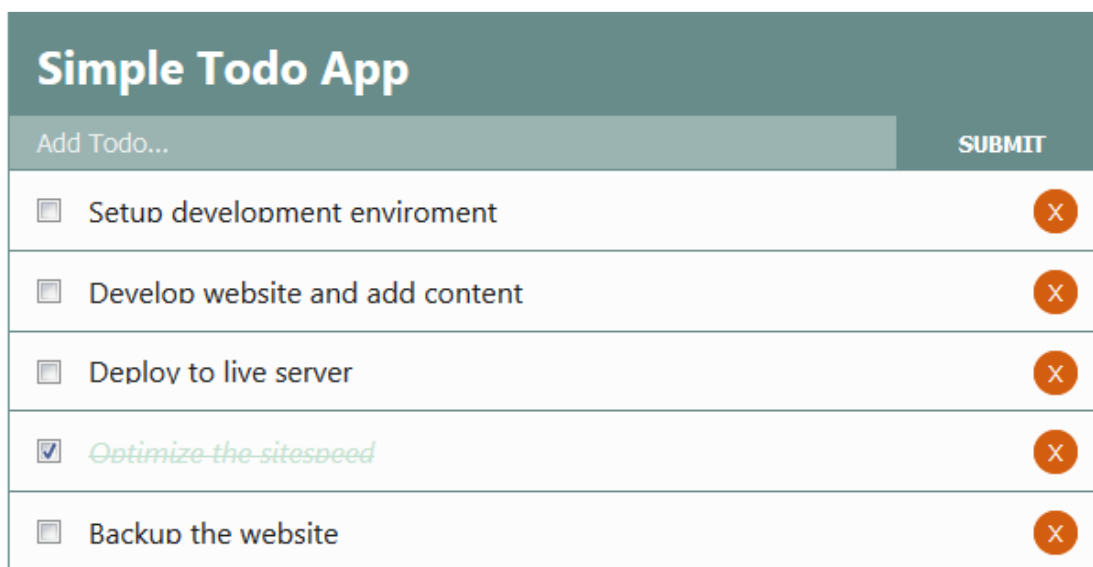
XÂY DỰNG ỨNG DỤNG TODOS

Sau khi hoàn thành xong phần 1 và 2 của cuốn sách này, bạn đã hiểu được khái niệm cơ bản của React, triết lý thiết kế của React cũng như đã chuẩn xong môi trường để phát triển ứng dụng React.

Đến phần này, chúng ta sẽ cùng nhau vừa tìm hiểu lý thuyết của React, vừa áp dụng kiến thức đó để xây dựng ứng dụng Todos.

Giới thiệu ứng dụng Todos

Đây là giao diện ứng dụng Todos mà chúng ta sẽ xây dựng trong phần này.



Simple Todo App	
Add Todo...	SUBMIT
<input type="checkbox"/> Setup development enviroment	X
<input type="checkbox"/> Develop website and add content	X
<input type="checkbox"/> Deploy to live server	X
<input checked="" type="checkbox"/> Optimize the sitespeed	X
<input type="checkbox"/> Backup the website	X

Hình 4.1: Giao diện ứng dụng Todo.

Giống như tên gọi, ứng dụng Todo dùng để lưu lại những công việc, những task cần phải thực hiện. Mỗi Todo được hiểu là một task, một công việc cụ thể mà người dùng cần phải thực hiện.

Ứng dụng này sẽ có những tính năng cơ bản như:

- Hiển thị danh sách các Todo được lấy từ API về.
- Thêm một Todo mới - Todo được thêm vào server thông qua API.
- Xóa một Todo - Todo được xóa trên server thông qua API.
- Đánh dấu một Todo đã hoàn thành.

Đầu tiên, các bạn checkout mã nguồn dự án rỗng tại đây:

<https://github.com/vntalking/ebook-reactjs-that-don-gian/tree/master/phan4/todo-app-starter>



Sau khi checkout về, bạn nhớ cài đặt các dependencies cần thiết bằng câu lệnh: `npm install`. Sau đó gõ: `npm start` để chạy dự án, nếu gặp lỗi: "System limit for number of file watchers reached - react-native", cách sửa như sau link: <https://github.com/facebook/create-react-app/issues/7612#issuecomment-565380404>. Cụ thể: gõ lệnh: `$ echo fs.inotify.max_user_watches=524288 | sudo tee -a /etc/sysctl.conf`. Sau đó gõ tiếp: `$ sudo sysctl -p`

Hoặc bạn có thể code online tại đây (fork về tài khoản của bạn):

<https://codesandbox.io/s/gallant-haze-rplk8>

Mã nguồn dự án hiện tại được tạo bằng công cụ `create-react-app`, và chỉ có tệp `index.js` có nội dung như sau:

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

const element = <h1>Hello from Create React App</h1>;

ReactDOM.render(element, document.getElementById("root"));
```

Trong đoạn code trên, chúng ta thực hiện render trực tiếp một element vào cây DOM thực. Thực tế thì không ai làm thế này cả. Người ta sẽ render một React component (như mình đã giải thích trong phần Virtual DOM)

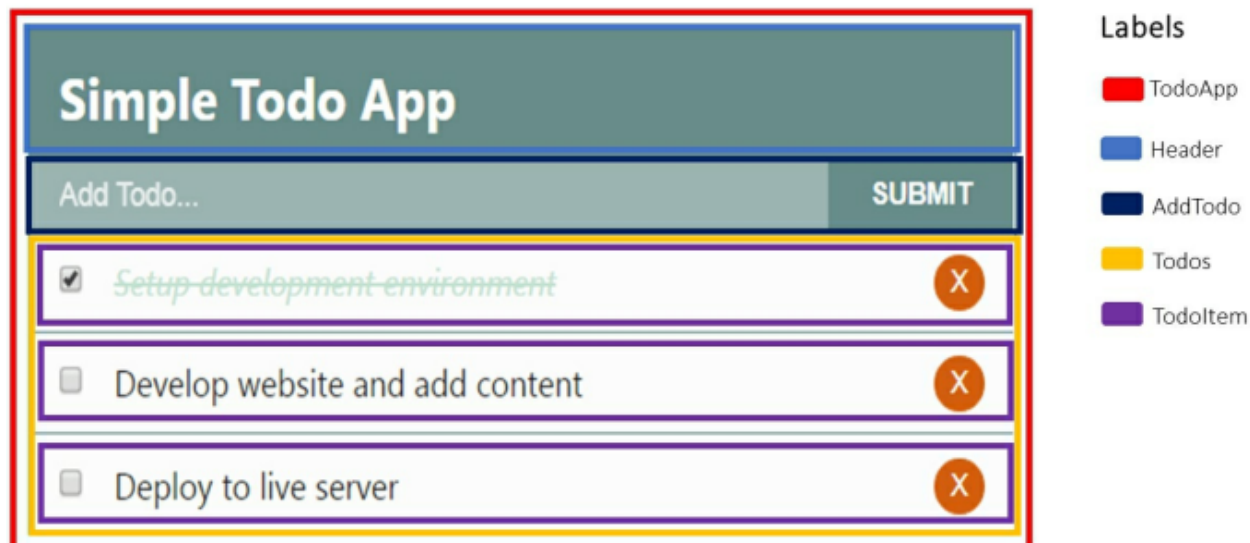
Những kiến thức React cơ bản nhất

1. React Component

Như mình đã cập trước đó, mọi thứ bạn nhìn trên giao diện (UI) đều được chia thành các thành phần độc lập, gọi là Component. Điều này có nghĩa là, khi bạn nhận bản thiết kế giao diện từ đội UX Graphic, bạn cần suy nghĩ làm sao chia nhỏ giao diện đó thành nhiều phần độc lập với nhau, sau đó tái sử dụng chúng.

Nếu bạn nhìn hình ảnh giao diện ứng dụng dưới đây. Thoạt nhìn ban đầu, trong nó cũng "hầm hố" phết, đúng không?

Tư duy mà React muốn bạn làm quen ở đây đó chính là **chia tách**.



Hình 4.2: Giao diện ứng dụng Todos List

Để xây dựng các ứng dụng như này, thậm chí cả những ứng dụng phức tạp như Facebook, Twitter... Việc đầu tiên là bạn cần hình thành tư duy là chia tách. Bạn cần chia thiết kế UI thành nhiều phần nhỏ hơn, càng độc lập với các thành phần khác càng tốt.

Như hình trên, sau khi mình phân tích và khoanh vùng, mình tạo được 3 thành phần nhỏ nhất, có thể tái sử dụng: *header*, *AddTodo*, *TodoItem*.

Các thành phần nhỏ nhất này được tái sử dụng để tạo thành một thành phần phức tạp hơn. Ví dụ: nhiều thành phần *TodoItem* sẽ tạo ra thành phần *Todos*.

Về mặt code, React component không có gì đặc biệt cả, điểm nổi bật duy nhất là nó **độc lập và có thể tái sử dụng**.

Đây cũng chính là một trong những yếu tố giúp React phổ biến.

Có thể coi component như một tính năng đơn giản mà bạn có thể gọi ở bất kỳ đâu, chỉ cần bạn truyền giá trị đầu vào và nó sẽ hiển thị kết quả cho bạn.

React có 2 kiểu viết component: Functional component và Class component.

1. Functional component

Functional component là một hàm Javascript (hoặc ES6) trả về 1 React element. Theo tài liệu chính thức của React, hàm dưới đây là một component hợp lệ.

```
function Welcome(props) {  
  return <h1>Hello, {props.name}</h1>;  
}
```

Function này là một component React hợp lệ vì nó nhận một "props" làm tham số và trả về 1 React element.

Vì vậy mình có thể định nghĩa 1 component như 1 hàm Javascript thông thường:

```
function exampleFunctionalComponent() {  
  return ( <h1>Tôi là một functional component!</h1> );  
};
```

Hoặc theo ES6 arrow function:

```
const exampleFunctionalComponent = () => {  
  return ( <h1>Tôi là một functional component!</h1> );  
};
```

Tóm lại, 1 React function component:

- Là một function Javascript / ES6 function
- Phải trả về 1 React element.
- Nhận props làm tham số nếu cần.



Functional component cũng được biết tới với cái tên là stateless components. Bởi vì chúng không thể làm nhiều thứ phức tạp như quản lý React State (data) hoặc xử lý vấn đề liên quan tới life-cycle trong functional components.

Tuy nhiên, từ phiên bản React 16.8, nhà phát hành giới thiệu tính năng React Hooks. Với Hooks, chúng ta có thể sử dụng state và những features khác trong functional components. Chúng ta sẽ tìm hiểu và thực hành về React Hooks ở phần 6 cuốn sách.

2. Class component

Các Class components là những class ES6. Chúng phức tạp hơn functional components một chút.

Đó là Class component còn có:

- Phương thức khởi tạo, có hàm về vòng đời component, hàm *render()*
- State (dữ liệu ứng dụng).

Ví dụ một class component:

```
class Welcome extends React.Component {  
  render() {  
    return <h1>Hello, {this.props.name}</h1>;  
  }  
}
```

Tóm lại, một Class component là:

- Là một class kế thừa từ *React.Component*
- Có thể nhận props (trong hàm khởi tạo) nếu cần.
- Phải có hàm *render()* và trong đó trả về 1 React element hoặc NULL.

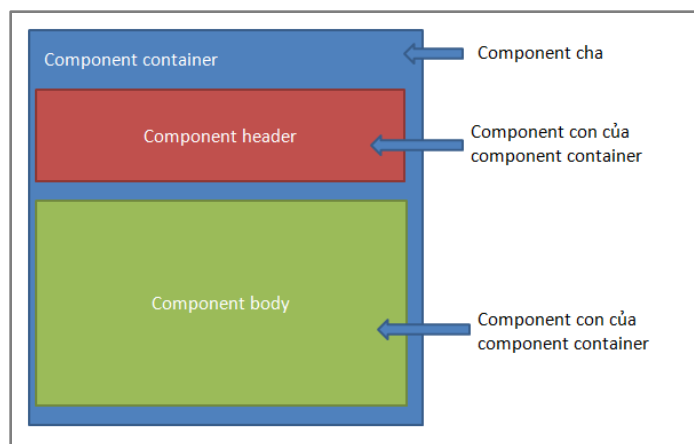
Props và State

Khi học React, bạn sẽ được nghe tới rất nhiều hai khái niệm Props và State.

Với một ứng dụng nói chung, hay ứng dụng React nói riêng, việc phải lưu dữ liệu hay truyền dữ liệu giữa các màn hình, giữa các components... là điều xảy ra như cơm bữa vậy. Đây chính là lúc sinh ra khái niệm props và state.

Trước khi trình bày lý thuyết về props và state, mình muốn thống nhất cách gọi: thế nào là component cha, component con. Theo cách gọi của mình, component cha là component chứa component con. Ở đây, khái niệm "chứa" tức là giao diện (UI) component nằm bên trong nhau, chứ không phải là kế thừa như trong ngôn ngữ lập trình.

Bạn có thể xem hình bên dưới để rõ thêm.



Hình 4.3: Giải thích về thuật ngữ component con và component cha

Props

Props (nó là từ viết tắt của properties) có thể được coi là các thuộc tính của phần HTML.

Ví dụ: thuộc tính type, checked... của input element.

```
<input type="checkbox" checked={true} />
```

Props là cách để bạn có thể truyền dữ liệu từ component cha xuống component con. Khi thực hiện truyền dữ liệu qua props, dữ liệu trong component con chỉ được đọc, không thể thay đổi. Nhờ đó mà component được sử dụng ở bất kỳ đâu cũng luôn hiển thị cùng 1 đầu ra khi có cùng 1 giá trị đầu vào. Điều này giúp chúng ta dễ dàng kiểm soát hơn.

Ví dụ cách sử dụng props:

```
class Welcome extends React.Component {  
  render() {  
    return <h1>Hello {this.props.name}</h1>;  
  }  
}
```

```
const element = <Welcome name="Sơn Dương" />;
```

Trong ví dụ trên, trong Welcome component có một props có tên là *name*. Và khi nó được gọi thì chúng ta có thể truyền giá trị vào props đó: `<Welcome name="Sơn Dương" />`

State

Như mình đã đề cập ở trên, một component có thể nhận dữ liệu từ một component cha hoặc nhận trực tiếp từ người dùng (ví dụ trong trường hợp người dùng nhập liệu qua form input).

Vì props chỉ cho phép một component nhận dữ liệu từ component cha. *Vậy điều gì sẽ xảy ra nếu người dùng nhập liệu trực tiếp trên component đó?*

Đây là lúc khái niệm state ra đời.

State bản chất chỉ là một Object để lưu dữ liệu.

Khi một component nhận dữ liệu trực tiếp từ người dùng (ví dụ như cập nhập giá trị từ các trường input, trạng thái của checkbox...), những dữ liệu này sẽ được lưu trong state và chỉ có thể được cập nhật bởi component định nghĩa state đó.

Tóm lại:

- State của một component là dữ liệu mà chỉ có component đó sử dụng và quản lý.
- Props là cách để truyền dữ liệu từ component cha xuống component con và chỉ component cha mới có thể thay đổi giá trị được.

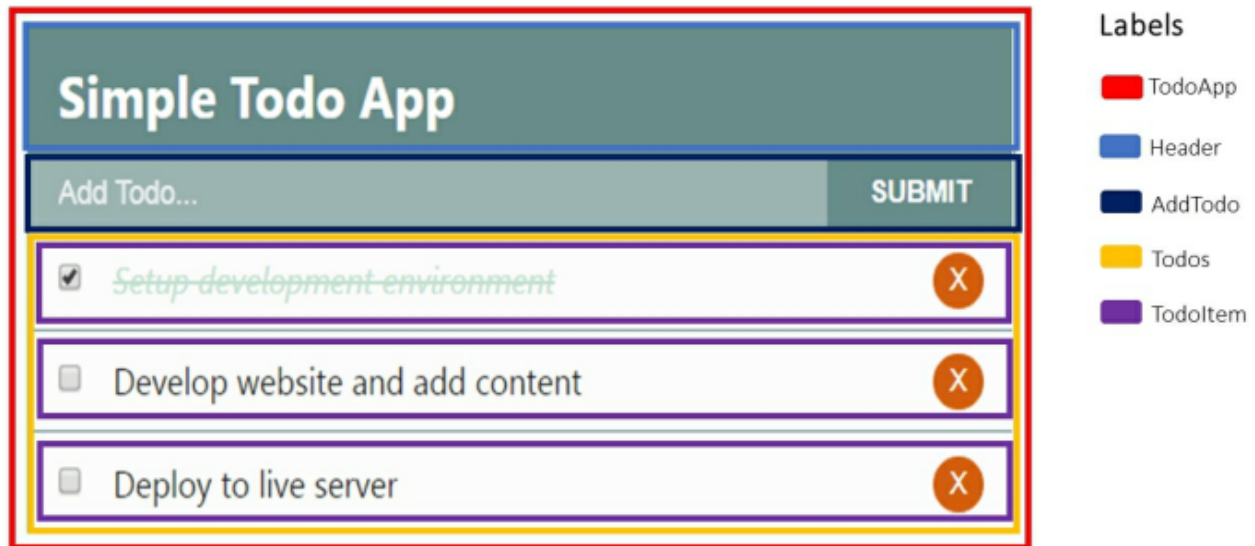
Bạn hiểu rõ về props và state rồi đúng không?

Giờ chúng ta bắt tay vào thực hành, áp dụng kiến thức về component, props và state vào ứng dụng todo.

Tạo các components

Để bắt đầu tạo các tệp trong dự án, bạn cần suy nghĩ tới cấu trúc thư mục của dự án. Điều này sẽ giúp bạn đỡ stress hơn khi dự án trở lên phức tạp hơn.

Chúng ta cùng xem lại thiết kế giao diện ứng dụng Todo:



Hình 4.4: Giao diện ứng dụng todo

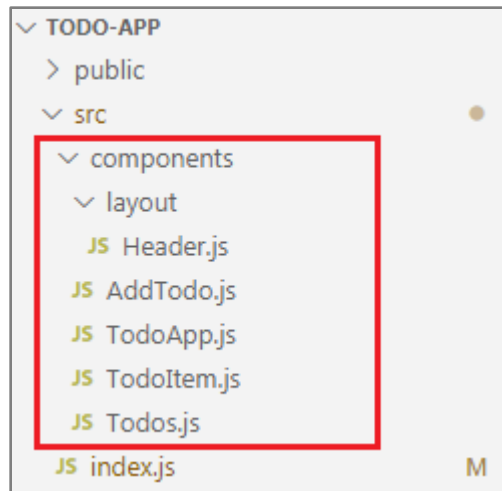
Từ giao diện này, chúng ta sẽ phân chia thành các component độc lập với nhau:

- **TodoApp**: Component cha chứa tất cả các component.
- **Header**: Hiển thị thông tin chung của ứng dụng, ở đây là tên ứng dụng.
- **AddTodo**: Nhận dữ liệu mà người dùng nhập vào.
- **Todos**: Hiển thị danh sách các todo.

Trong đó, component *Todos* là một danh sách, do vậy mình cần tạo thêm một *TodoItem* component nữa.

Ok, giờ tiến hành tạo file thôi.

Trong thư mục *src*, bạn tạo các tệp: *TodoApp.js*, *AddTodo.js*, *Todos.js*, *TodoItem.js*, *Header.js* và được đặt trong các thư mục như hình bên dưới đây.



Hình 4.5: Các components trong dự án



Mình quyết định để *Header.js* trong thư mục *layout* vì nó được tái sử dụng ở tất cả các màn hình. Còn bạn có thể tùy ý đặt nó bên ngoài thư mục *components* cũng được.

Tiếp theo, thêm đoạn mã sau vào component cha *TodoApp.js*

```
import React from "react";
class TodoApp extends React.Component {
  render() {
    return (
      <div>
        <h1>Hello from Create React App</h1>
        <p>I am in a React Component!</p>
      </div>
    );
  }
}
export default TodoApp;
```

Sau đó, bạn import *TodoApp* component này vào trong *index.js* (phần mình bôi đậm là chỗ code thay đổi).

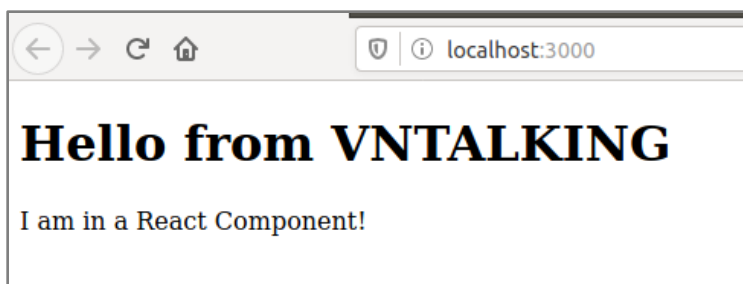
```
import React from "react";
import ReactDOM from "react-dom";

import TodoApp from "./components/TodoApp";

ReactDOM.render(<TodoApp />, document.getElementById("root"));
```

Đến đây bạn thu được kết quả như hình bên dưới (Xem online tại đây:

<https://codesandbox.io/s/relaxed-snow-wbmyr>)



Hình 4.6: Kết quả khi import `TodoApp` component

Giải thích cách viết component

Từ đoạn code ngắn trên, mình muốn giải thích 3 điểm để các bạn rõ hơn:

1. Quy tắc của hàm `render()`

Như bạn cũng đã thấy, mình tạo các components thành các file riêng và đều có ý đồ riêng cả.

Chúng ta tạo một component cha `TodoApp.js` (có thể coi là class-based component), mục đích là chứa toàn bộ nội dung giao diện ứng dụng như mình đã phân tích ở hình 3.2

Chúng ta định nghĩa một React component bằng cách tạo một class và kế thừa (extend) từ `Component` class của thư viện React. Trong class này, chúng ta phải sử dụng hàm `render()` trả về một JSX element để render giao diện lên màn hình.



Bạn không thể trả về nhiều hơn một JSX element ngang hàng nhau. Trả về nhiều element nhưng chúng phải lồng nhau, đảm bảo chỉ có duy nhất một element gốc. Có một giải pháp là bạn cho hết vào một thẻ `<div>`.

Bạn không thể làm như này:

```
render() {  
  // không được  
  return (  
    <h1>Hello from Create React App</h1>  
    <p>I am in a React Component!</p>  
  );  
}
```

Trong trường hợp bạn không muốn tạo nhiều element lồng nhau thì có thể sử dụng *React Fragment* (một element ảo, không hiển thị ra ngoài DOM).

Ví dụ sử dụng `<React.fragment>` thay vì sử dụng `<div>`

```
<React.Fragment>
  <h1>Hello from Create React App</h1>
  <p>I am in a React Component!</p>
</React.Fragment>
```

2. Cách sử dụng component trong JSX

Khi bạn tạo TodoApp component, nó sẽ không hiển thị ra ngoài ứng dụng cho đến khi bạn import và render trong *index.js*.

Trong *index.js*, chúng ta đã render TodoApp component bằng cách sử dụng một custom tag, tương tự như HTML vậy (*<TodoApp />*)

Ok, đến lúc này thì thay vì render một JSX element đơn giản, chúng ta đã render một React component.

Có một vài điểm mà mình muốn các bạn lưu ý:

- Nên sử dụng quy tắc đặt tên UpperCamelCase khi tạo các tệp component (ví dụ: *TodoApp.js*, *AddTodo.js*...)
- Tên của component phải được viết hoa (ví dụ: *class TodoApp*). Điều này là cần thiết bởi vì khi sử dụng (ví dụ: *<TodoApp />*) trong JSX, chúng không được coi là HTML tag.

3. Cách import một module/component

Khi bạn sử dụng bất kỳ tài nguyên nào bên ngoài file đang làm việc, bạn đều phải khai báo/import chúng.

Ví dụ: bạn cần sử dụng module React, hoặc component mà bạn tự tạo... bạn sẽ phải import chúng.

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

import TodoApp from "../components/TodoApp";
```

Khi import, bạn cần chỉ định đường dẫn tương đối tới tệp component đó từ thư mục hiện tại. Ví dụ, trong trường hợp TodoApp: *"../components/TodoApp"*, nghĩa là tệp TodoApp nằm ở thư mục components trong thư mục hiện tại (thư mục hiện tại là *src*).

Phần extension của tệp mặc định là *.js* nên bạn có thể bỏ qua khi import.

Tạo Header component và thêm Styles cho ứng dụng Todo

Bạn mở *Header.js* trong thư mục *src/components/layout* và thêm đoạn code sau:

```
import React from "react";

class Header extends React.Component {
  render() {
    return (
      <header>
        <h1>Simple Todo App</h1>
      </header>
    );
  }
};

export default Header;
```

Tiếp theo, để hiển thị header, bạn cần import và gọi nó trong *TodoApp* component.

```
import React from "react";
import Header from "../components/layout/Header";

class TodoApp extends React.Component {
  render() {
    return (
      <div>
        <Header/>
      </div>
    );
  }
}

export default TodoApp;
```

Kết quả được như hình bên dưới đây.



Hình 4.7: Hiển thị header của ứng dụng *Todo*

Nhìn có vẻ hơi đơn giản nhỉ (^_^)

Để "trang điểm" cho nó đẹp hơn, giống như yêu cầu giao diện ban đầu, chúng ta cần thêm style (CSS) cho nó.

Có 2 cách để thêm style cho ứng dụng:

- **Inline style:** Tức là thêm trực tiếp các thuộc tính CSS vào trong component.
- **External style:** Tức là mình sẽ tạo riêng một tệp CSS, sau đó import nó vào ứng dụng.

Cá nhân thì mình thích cách thứ 2 hơn, tạo một file CSS riêng. Chúng ta tiến hành thôi!

Trong thư mục *src*, tạo thêm một tệp mới *App.css*, sau đó import tệp CSS này vào ứng dụng.

Mở tệp *index.js* để import tệp CSS.

```
import './App.css';
```

Do khuôn khổ cuốn sách không đi sâu vào CSS và HTML nên mình sẽ đi nhanh phần "trang điểm" cho Header nhé.

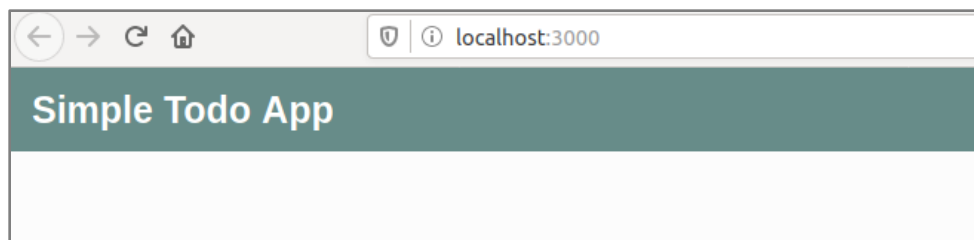
Trong *Header.js*, thêm *className* cho các tag HTML như sau (mục đích là để định danh dùng cho CSS):

```
class Header extends React.Component {  
  render() {  
    return (  
      <header className="header-container">  
        <h1 className="header-title">Simple Todo App</h1>  
      </header>  
    );  
  }  
};
```

Trong *App.css* thì thêm các định dạng như sau:

```
.header-container {  
  background-color: #678c89;  
  color: #fff;  
  padding: 10px 15px;  
}  
.header-title {  
  font-size: 25px;  
  line-height: 1.447em;  
  margin: 0px;  
}
```

Kết quả thu được như sau (xem online tại đây: <https://codesandbox.io/s/patient-tdd-7n7z8>).



Hình 4.7: Header sau khi thêm style

Thêm State

Như chúng ta đã biết, để có thể lưu giá trị có thể từ người dùng nhập, hoặc giá trị khởi tạo mặc định... thì bạn sẽ phải cần đến state.

Theo như thiết kế ban đầu, AddTodo component sẽ chịu trách nhiệm nhận dữ liệu mà người dùng nhập vào.

Ngoài ra, chúng ta cũng cần tạo State để lưu danh sách các todos (dù sau này thì danh sách này sẽ được lấy từ API trên internet về). Dù là cách nào thì chúng ta cũng cần phải sử dụng đến state.



Để component truy xuất vào state tối ưu nhất, bạn cần định nghĩa state trong component cha gần với các component đó nhất.

Chẳng hạn, trong ứng dụng Todos, các component như *AddTodo*, *TodoItem* đều muốn truy xuất vào state. Do vậy, state sẽ được định nghĩa trong *TodoApp* component (đây là component cha gần các component kia nhất).

Hi vọng đến đây bạn đã hiểu rõ!

Để thêm một state, chúng ta đơn giản là tạo một đối tượng, đặt tên là state. Trong đó có chứa mảng các todos, mảng todos là một mảng các đối tượng.

Bên trong tệp *TodoApp.js*, thêm đoạn mã sau, ngay phía trên hàm *render()*.

```

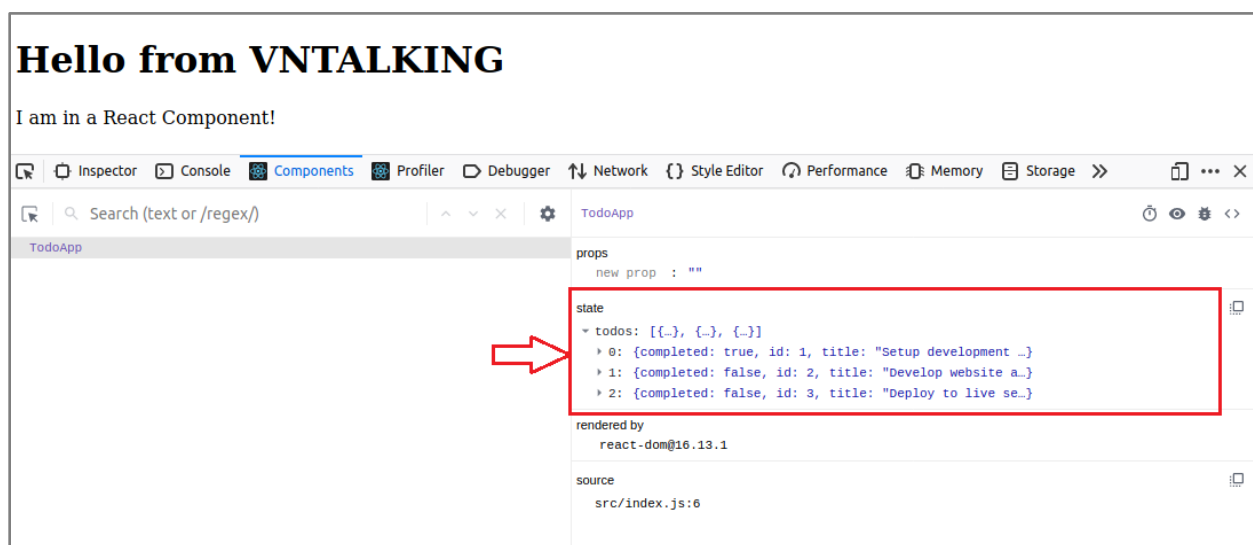
state = {
  todos: [
    {
      id: 1,
      title: "Setup development environment",
      completed: true
    },
    {
      id: 2,
      title: "Develop website and add content",
      completed: false
    },
    {
      id: 3,
      title: "Deploy to live server",
      completed: false
    }
  ]
};

```



Chúng ta tạo state với giá trị có sẵn, coi như là default data để hiển thị ra ngoài màn hình. Sau này thì giá trị của state sẽ được thêm bởi người dùng nhập thông qua input form hoặc lấy từ API trên máy chủ về (sẽ hướng dẫn ở phần sau - cứ bình tĩnh ^^)

Sau khi thêm đoạn code trên, bạn có thể xem state của ứng dụng thông qua React DevTools (đã cài đặt ở phần 3 cuốn sách).

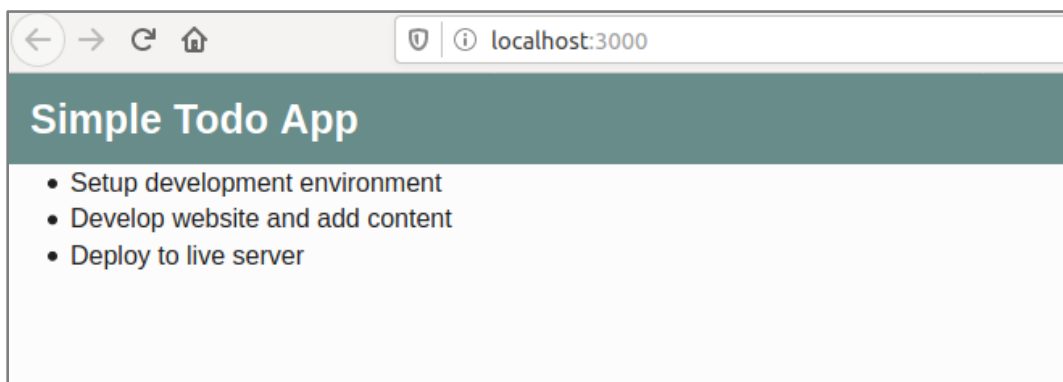


Hình 4.8: Kiểm tra dữ liệu của state bằng React DevTools

Tiếp theo, để hiển thị dữ liệu của state ra ngoài danh sách, chúng ta cần sửa lại hàm `render()` trong `TodoApp.js`

```
render() {  
  return (  
    <div>  
      <Header />  
      <ul>  
        {this.state.todos.map(todo => (  
          <li key={todo.id}>{todo.title}</li>  
        ))}  
      </ul>  
    </div>  
  );  
}
```

Lưu lại và kiểm tra kết quả (xem online tại đây: <https://codesandbox.io/s/phan-4-them-state-wzod6>).



Hình 4.9: Hiển thị dữ liệu state lên màn hình

Giải thích

Sau khi định nghĩa state, chúng ta truy xuất dữ liệu của state trong hàm `render()` thông qua từ khóa `this` (như ở đây là: `this.state.todos`).

Bởi vì dữ liệu `todos` là một mảng các đối tượng, nên chúng ta cần phải duyệt mảng để xuất từng phần tử `todo` ra màn hình.



Trong JSX, bạn có thể sử dụng các hàm Javascript bằng cách viết chúng trong dấu ngoặc nhọn `{}`

Ở đây, mình mới chỉ hiển thị kết quả kiểu đơn giản thôi. Gọi là ví dụ để bạn biết cách truy xuất vào state thôi. Còn trong ứng dụng Todos, chúng ta sẽ hiển thị với giao diện phức tạp hơn. Đây là lúc cần áp dụng kiến thức về Props.

Tạo component hiển thị danh sách Todos

Đầu tiên, bạn mở tệp *Todos.js* và tạo một component có tên là Todos.

Mình cứ tạm thời tạo Todos với nội dung như sau:

```
import React from "react";

class Todos extends React.Component {
  render() {
    return (
      <div>
        Hello from Todos
      </div>
    );
  }
}
export default Todos;
```

Tiếp theo bạn mở *TodoApp.js* để gọi Todos component.

```
import React from "react";
import Todos from "../Todos";
...
render() {
  return (
    <div>
      <Header />
      <Todos todos={this.state.todos} />
    </div>
  );
}
```

Ở đoạn code trên, chúng ta truyền dữ liệu cho component con thông qua props có tên là *todos*. Đây chính là cách để truyền dữ liệu từ component cha xuống component con thông qua props.

Do vậy, chúng ta có thể sử dụng dữ liệu này trong Todos component.

Giờ chúng ta mở lại *Todos.js* và cập nhập lại mã nguồn.

```
import React from "react";

class Todos extends React.Component {
  render() {
    return (
      <div>
        <ul>
          {this.props.todos.map(todo => (
            <li key={todo.id}>{todo.title}</li>
          ))}
        </ul>
      </div>
    );
  }
}
export default Todos;
```

Khi hoàn thành mã nguồn đến đoạn này, chúng ta vẫn thu được giao diện danh sách giống như trước.

Tiếp theo, thay vì render các Todo ngay trong Todos component, chúng ta sẽ tách ra một component riêng cho từng phần tử Todo, đó chính là TodoItem component.

Cách làm tương tự như đã làm trước đó. Để cho nhanh hơn, giờ mình sẽ copy đoạn code trong Todos sang TodoItem.

Nội dung TodoItem component (*TodoItem.js*) sẽ như sau:

```
import React from "react";

class TodoItem extends React.Component {
  render() {
    return (
      <li>{this.props.todo.title}</li>
    );
  }
}
export default TodoItem;
```

Sau đó import và gọi TodoItem trong Todos component.

```
import React from "react";
import TodoItem from "./TodoItem";

class Todos extends React.Component {
  render() {
    return (
      <div>
        <ul>
```

```

        {this.props.todos.map(todo => (
          <TodoItem key={todo.id} todo={todo} />
        ))}
      </ul>
    </div>
  );
}
}
export default Todos;

```

Lưu lại và kiểm tra kết quả. Giao diện ứng dụng vẫn như cũ là được.

Giờ mình "trang điểm" một chút cho phần danh sách này.

Trong *TodoApp.js* thêm `className` là *container*

```

<div className="container">
  <Header />
  <Todos todos={this.state.todos} />
</div>

```

Tương tự trong *TodoItem.js*, chúng ta thêm `className` là *todo-item*.

```

<li className="todo-item">{this.props.todo.title}</li>

```

Cuối cùng là thêm CSS trong *App.css*

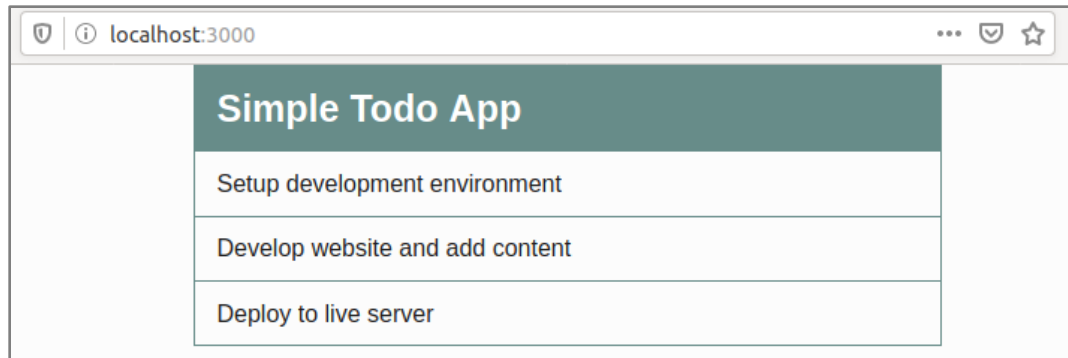
```

* {
  box-sizing: border-box;
  margin: 0 0 0 0;
  padding: 0;
}
body {
  font-family: "Segoe UI", Arial, sans-serif;
  line-height: 1.4;
  background-color: #fcfcfc;
  color: #1f1f1f;
}
.container {
  max-width: 500px;
  margin: 0 auto;
  border: 1px solid #678c89;
}
.todo-item {
  list-style-type: none;
  padding: 10px 15px;
  border-top: 1px solid #678c89;
}

```

Kết quả thu được cũng đẹp hơn chút ít (xem online tại đây:

<https://codesandbox.io/s/phan-4-tao-component-hien-thi-danh-sach-todos-lvdzl>)



Hình 4.10: Chính giao diện danh sách Todos

Thêm checkbox trong Todo Items

Có thể bạn đã làm việc nhiều với các kiểu Input trong HTML rồi. Trong React thì có đôi chút khác biệt.

Ok. Để tạo checkbox, bạn chỉ cần thêm thẻ input với type là "checkbox" như bên dưới đây. Bạn sửa trong TodoItem component nhé.

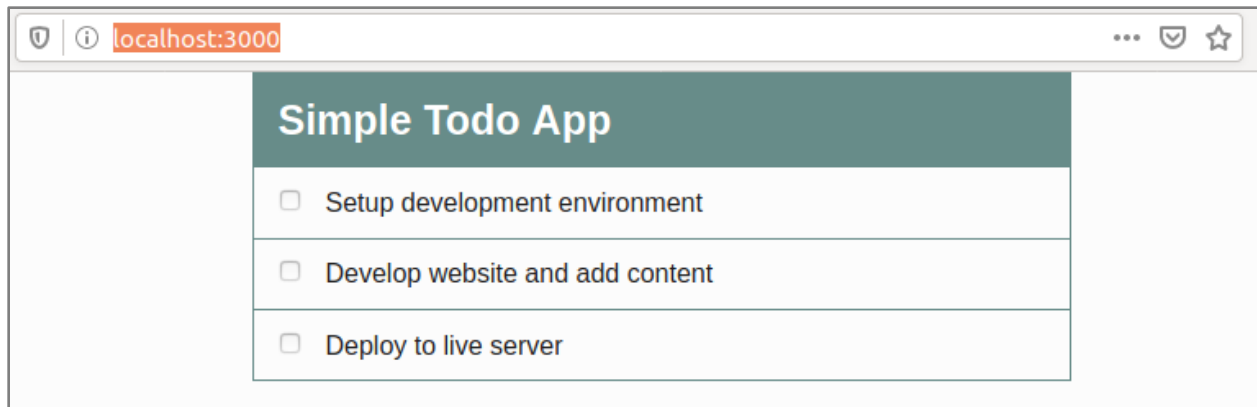
```
<li className="todo-item">
  <input type="checkbox" />{this.props.todo.title}
</li>
```

Sau đó thì thêm style cho thẻ Input đó.

```
.todo-item input {
  margin-right: 15px;
}
```

Lưu lại và kiểm tra thành quả trên trình duyệt nhé (xem online tại đây:

<https://codesandbox.io/s/epic-gagarin-r82z7>)



Hình 4.11: Thêm checkbox cho *TodoItem* component

Mặc định, các input (trong trường hợp này là checkbox) được xử lý bởi DOM. Tức là nó có các hành vi làm việc mặc định của HTML, ví dụ như bạn có thể check hoặc uncheck vào box đó mà không cần phải viết thêm code gì nữa. Kiểu input này được gọi là Uncontrolled Input - tức là Input không kiểm soát.

Tuy nhiên, trong React các input được hiểu là các input có thể kiểm soát được.

Điều này dẫn chúng ta tới một chủ đề quan trọng khác. Đó là điều khiển - quản lý Component.

Controlled Component

Để làm cho Input fields có thể kiểm soát được, giá trị của input (ví dụ là trạng thái check/uncheck của checkbox) phải được xử lý bởi component state thay vì browser DOM.

Chúng ta sẽ cùng nhau phân tích xem state làm việc như thế nào trong trường hợp này nhé.

Hãy xem lại đối tượng State mà chúng ta đã định nghĩa ở phần trước. Trong mảng *todos*, mỗi *TodoItem*, chúng ta có một key là: *completed* có kiểu dữ liệu là Boolean (true/false). Giá trị true hay false sẽ tương ứng với trạng thái check/uncheck của checkbox.

Quay trở lại *TodoItem.js*, sử dụng attribute *checked* để thiết lập giá trị trạng thái của checkbox.

```
<input type="checkbox" checked={this.props.todo.completed} />
```

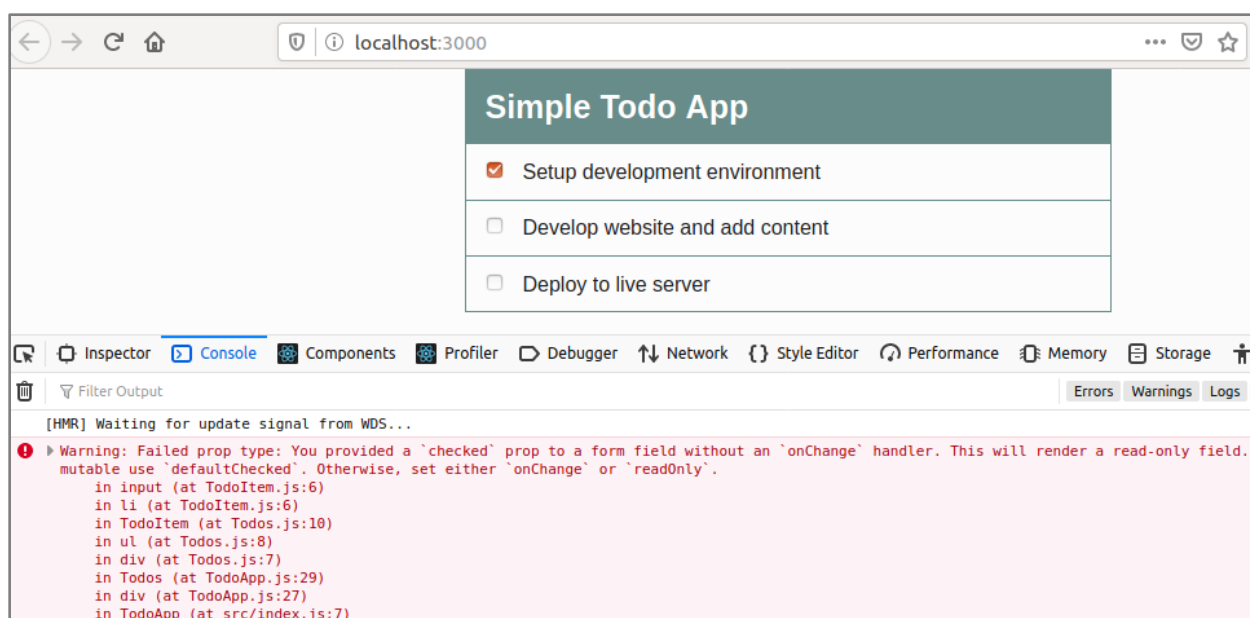
Lưu lại và kiểm tra kết quả trên trình duyệt.

Tại thời điểm này, bạn thử tick vào các checkbox thì cũng chẳng có chuyện gì xảy ra, checkbox không thay đổi trạng thái. Lý do là vì trạng thái của checkbox bây giờ được gán bằng giá trị hiện tại trong state.

Trong giá trị state ban đầu, chỉ có todo task đầu tiên là đã hoàn thành nên checkbox có trạng thái là được checked.

Chúng ta cần phải có cách để thay đổi giá trị *complete* trong state khi người dùng tick vào checkbox.

Giải pháp đó chính là sử dụng listener: *onChange*. Giải pháp này cũng được React gợi ý, không tin, bạn có thể mở console của trình duyệt lúc này nên mà xem.

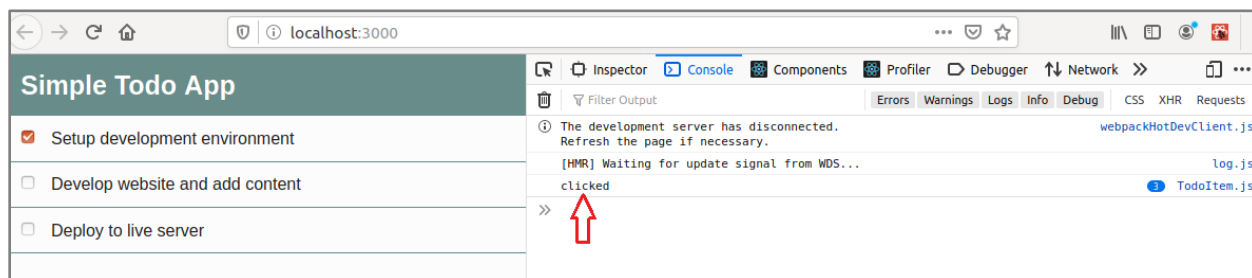


Hình 4.12: Cảnh báo của React khi chưa xử lý sự kiện check cho checkbox

OK, giờ chúng ta mở *TodoItem.js* để thêm *onChange*.

```
<input
  type="checkbox"
  checked={this.props.todo.completed}
  onChange={() => console.log("clicked")}
/>
```

Lưu lại và kiểm tra trong trình duyệt. Trong tab console, cảnh báo đã biến mất. Bạn click vào checkbox thì thấy in log như trong hình. Điều đó chứng tỏ hàm *onChange* đã hoạt động.



Hình 4.13: Kiểm tra hoạt động hàm *onChange()*

Công việc tiếp theo là xử lý logic mỗi khi người dùng click vào checkbox thay vì chỉ in ra mỗi dòng console log.

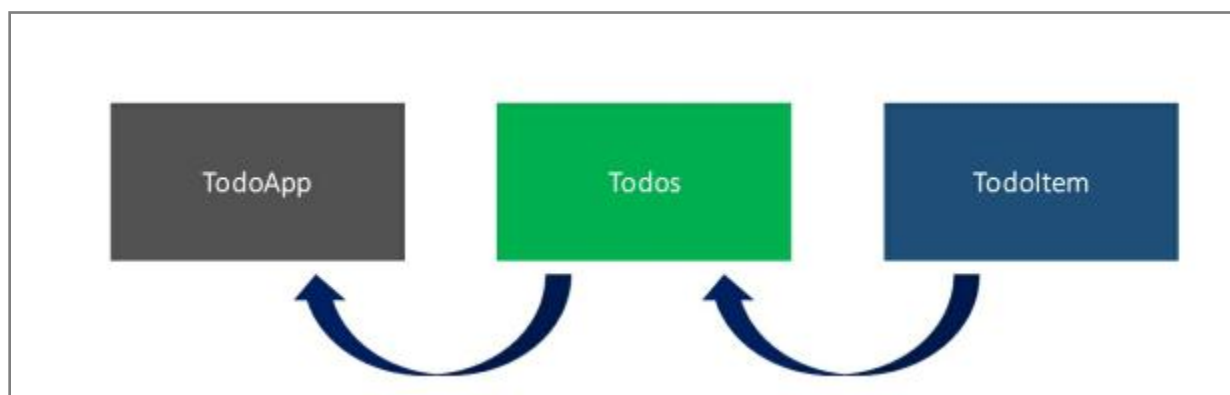
Gửi và xử lý sự kiện - Handle Events

Trong ứng dụng Todo, chỉ có TodoApp component là giữ giá trị state. Do vậy, chỉ có TodoApp component là có thể thay đổi giá trị của state.

Trong khi chúng ta muốn người dùng click vào checkbox (checkbox nằm trong TodoItem component) và thay đổi được giá trị của state (chính là cập nhập giá trị trường *completed*). Để làm được điều này, bạn chỉ có cách đó là "bắn event" từ TodoItem lên TodoApp component.



Để truyền dữ liệu từ component cha xuống component con thì truyền qua Props. Còn truyền dữ liệu từ component con lên component cha, sử dụng event là cách phổ biến nhất



Hình 4.14: Đường đi của event

Chúng ta cùng thực hành nhé.

Đầu tiên là thêm code để cho phép các component có thể giao tiếp với nhau.

Bắt đầu từ component cha (*TodoApp.js*), thêm phương thức *handleCheckboxChange()* ngay trên hàm *render()*

```
handleCheckboxChange = () => {  
    console.log("clicked");  
};
```

Tên phương thức *handleCheckboxChange()* là mình tự đặt nhé, bạn có thể đặt khác đi cũng được. Miễn là lúc gọi thì gọi đúng tên là được.

Sau đó chúng ta truyền phương thức này xuống component con thông qua props. Làm như sau:

Vẫn trong *TodoApp* component, bạn cập nhật lại chỗ gọi *Todos* trong hàm *render()* như sau:

```
render() {  
    return (  
        <div className="container">  
            <Header />  
            <Todos todos={this.state.todos} handleChange={this.handleCheckboxChange} />  
        </div>  
    );  
}
```

Tiếp theo chúng ta sửa lại mã nguồn *Todos.js* để truyền tiếp xuống *TodoItem* component.

```
render() {  
    return (  
        <div>  
            <ul>  
                {this.props.todos.map(todo => (  
                    <TodoItem  
                        key={todo.id}  
                        todo={todo}  
                        handleChange={this.props.handleChange} />  
                ))}  
            </ul>  
        </div>  
    );  
}
```

Cuối cùng thì bạn sửa lại hàm `onChange()` trong `Todoltem` component (`Todoltem.js`).

```
render() {
  return (
    <li className="todo-item">
      <input
        type="checkbox"
        checked={this.props.todo.completed}
        onChange={() => this.props.handleChange()}
      />
      {this.props.todo.title}</li>
    );
  }
}
```

Lưu lại và kiểm tra kết quả trên trình duyệt. Lúc này, bạn click vào các checkbox, bạn sẽ thấy `console.log(...)` trong hàm `handleCheckboxChange` được gọi. Về cơ bản thì chúng ta đã truyền event khi thay đổi trạng thái của checkbox từ `Todoltem` lên `TodoApp` rồi đấy.

Giờ chúng ta làm thêm một bước nữa. Thay vì chỉ console log đơn giản, chúng ta sẽ in ra chính xác id của checkbox nào được click.

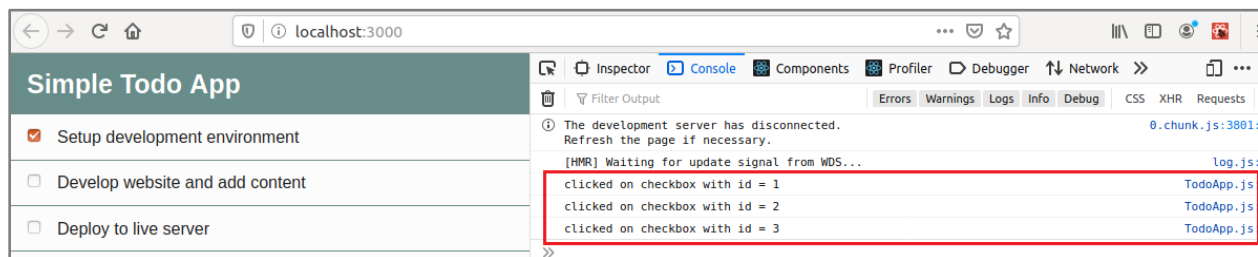
Sửa lại `onChange` trong `Todoltem` component một chút.

```
onChange={() => this.props.handleChange(this.props.todo.id)}
```

Sau đó cập nhật lại hàm `handleCheckboxChange()` trong `TodoApp` component (`TodoApp.js`)

```
handleCheckboxChange = id => {
  console.log("clicked on checkbox with id = " + id);
};
```

Lưu lại và kiểm tra thành quả trên trình duyệt nhé.



Hình 4.15: xử lý sự kiện click vào checkbox

Nhưng có vẻ như trạng thái UI của checkbox không có thay đổi khi click vào đúng không?

Mời bạn đọc phần tiếp theo để được giải đáp.

Cập nhật giá trị state sử dụng hàm `setState()`

Hoàn thành đến bước này, chúng ta mới chỉ in ra được id của checkbox được người dùng click vào. Còn giao diện thì checkbox đó vẫn không có gì thay đổi. Vì bản chất trạng thái của checkbox đang được gán bằng giá trị của trường `completed` trong state như mình đã đề cập trước đó.

Do vậy, chúng ta cần phải thay đổi giá trị của state mỗi khi người dùng click vào checkbox.



Mình muốn giải thích thêm chỗ này cho bạn hiểu rõ hơn về React. Không phải bạn tick vào checkbox thì bạn gọi hàm để thay đổi trạng thái của checkbox thành `true` (checkbox được ticked). Thực tế là khi bạn tick vào checkbox, bạn gọi hàm thay đổi giá trị state (cụ thể là trường `completed` trong state), hết, tất cả chỉ có vậy! Khi giá trị của state thay đổi, React sẽ tự động cập nhật UI của checkbox (cụ thể là checkbox được tick). Và cách làm việc này sẽ xuyên suốt các ứng dụng React.

Trong React, chúng ta không thay đổi giá trị state một cách trực tiếp, mà thông qua hàm `setState()`.

Mục đích của việc sử dụng hàm `setState()` khi thay đổi giá trị state là để React có hành động cập nhật UI.

Đây là đoạn code để thay đổi state trong `TodoApp` component (`TodoApp.js`)

```
handleCheckboxChange = id => {  
  this.setState({  
    todos: this.state.todos.map(todo => {  
      if (todo.id === id) {  
        todo.completed = !todo.completed;  
      }  
      return todo;  
    })  
  });  
};
```

Nhìn logic của đoạn code cũng dễ hiểu đúng không?!

Giờ bạn kiểm tra trên trình duyệt nhé. Chúng ta đã có thể check và uncheck các checkbox của mỗi `Todo` item. Xem online tại đây: <https://codesandbox.io/s/exciting-tu-e0p6m>

Thêm style cho các todo đã hoàn thành

Khi mỗi todo item được check, tức là todo đó đã hoàn thành. Để cho ứng dụng todo nhìn chuyên nghiệp hơn, chúng ta sẽ thêm style gạch ngang những todo đã check (coi như đã hoàn thành).

Mở `App.css` và thêm style này:

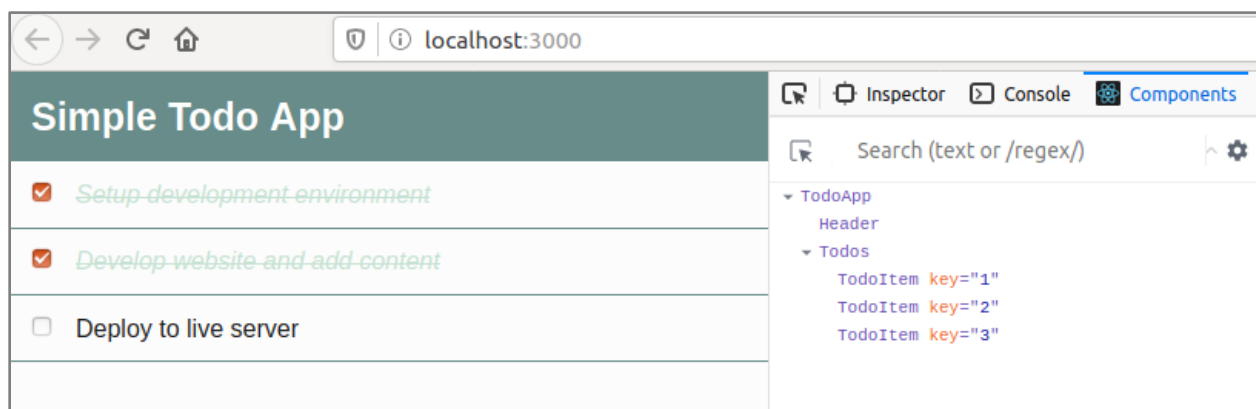
```
.completed {  
  font-style: italic;  
  color: #c5e2d2;  
  text-decoration: line-through  
}
```

Sau đó cập nhật lại `TodoItem` component, thêm thẻ `span` để định dạng cho mỗi item.

```
render() {  
  return (  
    <li className="todo-item">  
      <input  
        type="checkbox"  
        checked={this.props.todo.completed}  
        onChange={() => this.props.handleChange(this.props.todo.id)}  
      />  
      <span className={this.props.todo.completed ? "completed" : null}>  
        {this.props.todo.title}  
      </span>  
    </li>  
  );  
}
```

Kết quả thu được như hình dưới đây. Xem online tại đây:

<https://codesandbox.io/s/unruffled-dust-9mx1h>



Hình 4.16: Thêm style cho todo đã hoàn thành

Sử dụng Destructuring

Bạn có để ý là trong `TodolItem` component, chúng ta gọi rất nhiều tới `this.props.todo`. Đây có thể nguyên nhân của "bệnh đau đầu kinh niên" nếu ứng dụng trở nên phức tạp hơn, code bị thừa thãi rất nhiều. Mình chỉ muốn gọi ngắn gọn `id` thay vì cứ phải là `this.props.todo.id`.

Để đỡ đau đầu, chúng ta sẽ tách các biến ra khỏi đối tượng `Todo`, người ta gọi kỹ thuật này là "destructure".

Để làm điều đó, chúng ta mở `TodolItem` component, và thêm đoạn code sau ngay phía trên hàm `return()` trong hàm `render()`.

```
render() {  
  const { completed, id, title } = this.props.todo  
  return (  
    <li className="todo-item">  
    ...  
  )  
}
```

Sau đó, thay thế các biến `this.props.todo` bằng biến tương ứng.

Ví dụ: `this.props.todo.completed` được thay bằng biến `completed`

Kết quả cuối cùng được đoạn code như dưới đây:

```
render() {  
  const { completed, id, title } = this.props.todo  
  return (  
    <li className="todo-item">  
      <input  
        type="checkbox"  
        checked={completed}  
        onChange={() => this.props.handleChange(id)}  
      />  
      <span className={completed ? "completed" : null}>  
        {title}  
      </span>  
    </li>  
  );  
}
```

Bạn thấy chưa, nhìn code "ngon" hơn hẳn (^_O)

Xóa một Todo

Cách để xóa một Todo cũng tương tự như cách chúng ta xử lý với checkbox vậy. Tức là chúng ta cũng cần phải xóa dữ liệu trong state ở TodoApp component mỗi khi người dùng click vào nút xóa.

Chúng ta bắt tay vào thực hành code luôn nhé.

Đầu tiên, thêm một nút xóa vào bên cạnh mỗi todo Item. Mở *TodoItem.js* và thêm đoạn code tạo button bên dưới thẻ *<input>* như sau:

```
return (  
  <li className="todo-item">  
    <input  
      type="checkbox"  
      checked={completed}  
      onChange={() => this.props.handleChange(id)}  
    />  
    <span className={completed ? "completed" : null}>  
      {title}  
    </span>  
    <button className="btn-style"> X </button>  
  </li>  
)  
);
```

Mở *App.css* để thêm định dạng cho nút xóa.

```
.btn-style {  
  background: #d35e0f;  
  color: #fff;  
  border: 1px solid #d35e0f;  
  padding: 3px 7px;  
  border-radius: 50%;  
  cursor: pointer;  
  float: right;  
  outline: none;  
}
```

Tiếp theo, chúng ta thực hiện gửi sự kiện từ *TodoItem* lên *TodoApp* component.

Mở *TodoApp* component (*TodoApp.js*) và thêm hàm xử lý sự kiện xóa.

```
deleteTodo = id => {  
  console.log("deleted", id);  
};
```

Vẫn trong TodoApp component, cập nhật `<Todos/>` để thêm:

```
<div className="container">
  <Header />
  <Todos todos={this.state.todos}
    handleChange={this.handleChange}
    deleteTodo={this.deleteTodo}/>
</div>
```

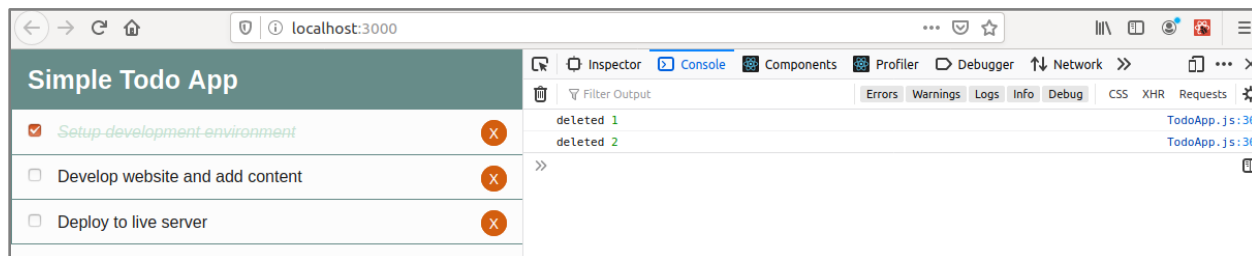
Tiếp tục chuyển sang Todos component và cập nhật `<TodoItem />`

```
<TodoItem
  key={todo.id}
  todo={todo}
  handleChange={this.props.handleChange}
  deleteTodo={this.props.deleteTodo}
/>
```

Cuối cùng, quay trở lại `TodoItem (TodoItem.js)` để thêm hàm `onClick()`. Hàm này được gọi khi người dùng click vào nút xóa.

```
<button className="btn-style" onClick={() => this.props.deleteTodo(id)}> X </button>
```

Lưu lại và kiểm tra kết quả trên trình duyệt.



Hình 4.17: Click vào nút xóa

Đến đây, khi bạn click vào nút xóa, ứng dụng mới chỉ in ra màn hình `id` của todo bị xóa mà thôi.

Để thực sự xóa được todo khi click nút xóa, chúng ta cần phải xóa chúng trong state.

Trong Javascript, có nhiều cách để xóa một item trong một mảng. Cách mình thích nhất đó là sử dụng hàm `filter()`. Hàm này sẽ trả về một mảng các phần tử thỏa mãn điều kiện nào đó.

Trong trường hợp này, chúng ta chỉ trả về các phần tử trong mảng `todos` có `id` khác với `id` được truyền vào.

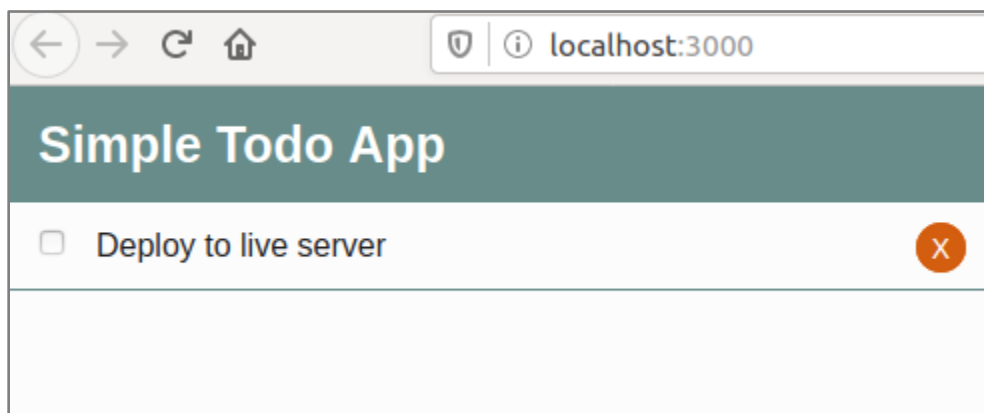
Quay trở lại hàm `deleteTodo()` trong `TodoApp` component và cập nhật code như sau:

```
deleteTodo = id => {  
  this.setState({  
    todos: [  
      ...this.state.todos.filter(todo => {  
        return todo.id !== id;  
      })  
    ]  
  });  
};
```



Dấu (...) trong đoạn code được gọi là toán tử *spread operator*. Nó cho phép chúng ta lấy todos hiện tại. Bạn có thể tìm hiểu kỹ hơn về toán tử này [tại đây](#).

Vậy là xong cho tính năng xóa Todo rồi đấy. Mời bạn hưởng thụ thành quả trên trình duyệt.



Hình 4.18: Tính năng xóa Todo

Xem online tại đây: <https://codesandbox.io/s/epic-kirch-i3idk>

Thêm một Todo mới

Trong React, tất cả các input field dù có kiểu type khác nhau thì đều có chung một cách tiếp cận.

Tương tự như cách chúng ta đã làm với input có kiểu là checkbox thì với text input field, chúng ta cũng làm tương tự.

Đầu tiên, chúng ta mở tệp `AddTodo.js` (tệp này chúng ta đã tạo sẵn lúc trước rồi), sau đó tạo `AddTodo` component.


```
import React from "react";

class AddTodo extends React.Component {
  render() {
    return (
      <form className="form-container">
        <input type="text" placeholder="Add Todo..." className="input-
-text" />
        <input type="submit" value="Submit" className="input-
submit" />
      </form>
    );
  }
}
export default AddTodo;
```

Ở AddTodo component, chúng ta sẽ phải lấy dữ liệu thông qua tương tác với người dùng, cụ thể là dữ liệu mà người dùng nhập vào text input field. Do đó, component này sẽ cần phải tạo thêm state nữa (mình sẽ nói sau).

Thêm style cho nó đã. Mở App.css và thêm đoạn CSS này.

```
.form-container {
  display: flex;
  width: 100%;
}
.input-text {
  flex: 8;
  font-size: 14px;
  padding: 6px 15px;
  background: rgba(103, 140, 137, 0.65);
  border: none;
  color: #fff;
  outline: none;
  font-weight: 400;
  width: 80%;
}
.input-text::placeholder {
  color: #fff;
  opacity: 0.8;
}
.input-submit {
  flex: 2;
  border: none;
  background: #678c89;
```

```

    color: #fff;
    text-transform: uppercase;
    cursor: pointer;
    font-weight: 600;
    width: 20%;
    outline: none;
}

```

Cuối cùng thì import AddTodo component này vào TodoApp component.

Mở *TodoApp.js* và thêm đoạn import này.

```
import AddTodo from "../AddTodo"
```

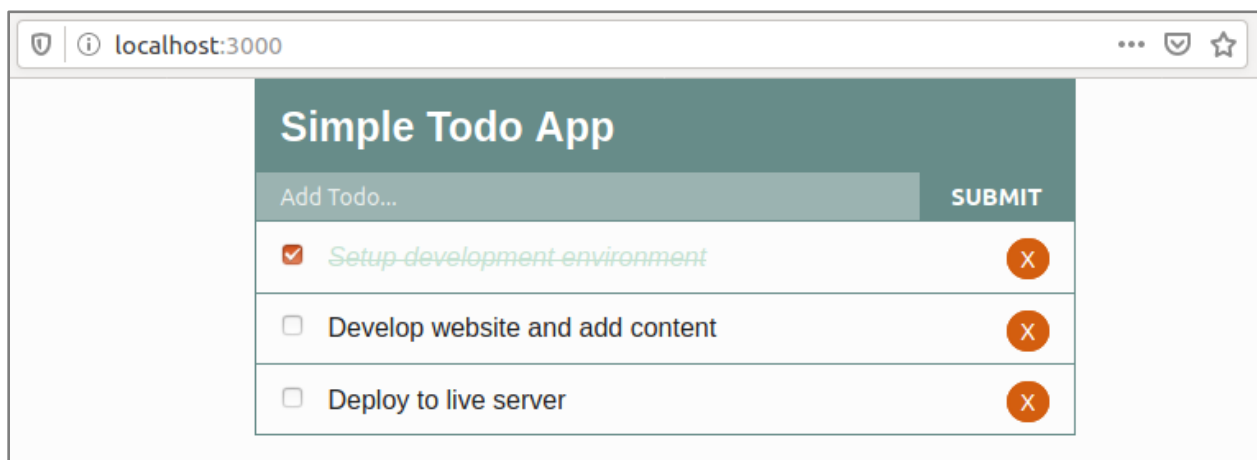
Và update lại phần render. Lưu ý là theo như thiết kế UI ban đầu, phần để thêm mới Todo sẽ nằm bên dưới header. Trong code cũng thế.

```

render() {
  return (
    <div className="container">
      <Header />
      <AddTodo />
      <Todos todos={this.state.todos}
        handleChange={this.handleClickboxChange}
        deleteTodo={this.deleteTodo}/>
    </div>
  );
}

```

Lưu lại và chạy thử xem thế nào.



Hình 4.19: Tính năng thêm mới Todo

Kết quả là được giao diện như hình trên. Tuy nhiên, khi click vào nút "Submit" thì ứng dụng chưa làm gì cả, đơn giản là do chúng ta chưa xử lý logic cho nó.

Về tư duy cho đoạn xử lý này là: Lưu giá trị mà người dùng nhập ở input field vào state trong AddTodo component. Sau đó, khi người dùng click nút "Submit" thì gửi giá trị đó lên state trong TodoApp component.

OK, tiến hành luôn.

Đầu tiên, khai báo state trong AddTodo component (*AddTodo.js*).

```
state = {  
  title: ""  
};
```

Tiếp theo, gán state vào text input field.

```
<input  
  type="text"  
  placeholder="Add Todo..."  
  className="input-text"  
  value={this.state.title}  
/>
```



Chúng ta sử dụng attribute checked cho input kiểu checkbox, còn attribute value cho text input.

Đến đây, giá trị của Input field được quản lý bởi state. Do vậy, bạn thử gõ bất kỳ giá trị nào trong input cũng không được.

Cũng giống với cách làm của checkbox, chúng ta cần thêm listener *onChange()* để lắng nghe sự kiện người dùng nhập, sau đó cập nhật giá trị của state thông qua hàm *setState()*.

Thêm *onChange()* vào input như sau:

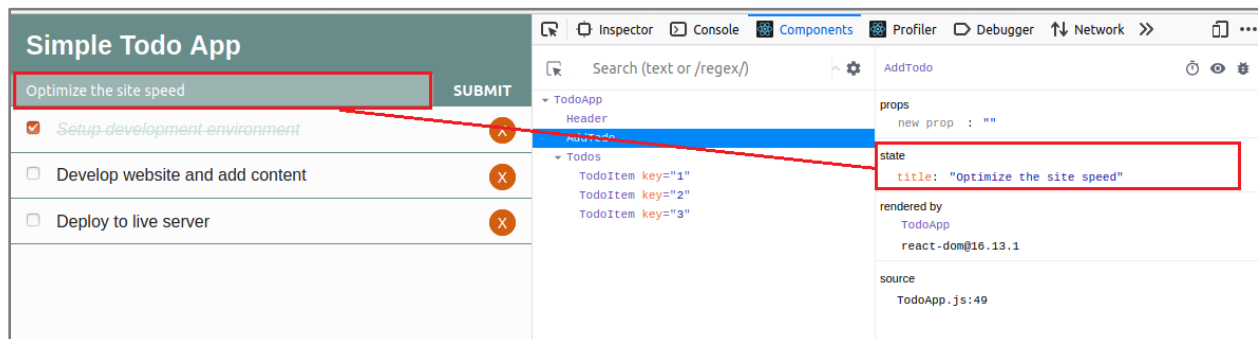
```
<input  
  type="text"  
  placeholder="Add Todo..."  
  className="input-text"  
  value={this.state.title}  
  onChange={this.onInputChange}  
/>
```

Và định nghĩa hàm xử lý khi *onChange()* được gọi.

```
onInputChange = e => {
  this.setState({
    title: e.target.value
  });
};
```

Lưu lại và kiểm tra trên trình duyệt.

Bạn có thể nhập một cái gì đó trong text input field. Ngoài ra, nếu bạn bật React tools, bạn sẽ thấy state được thay đổi real-time mỗi khi bạn gõ chữ.



Hình 4.20: Thay đổi giá trị state mỗi khi nhập trong input field.

Cập nhật Todo mới vào danh sách

Công việc tiếp theo, chúng ta sẽ xử lý khi người dùng nhập xong và click vào nút "Submit".

Để thêm một Todo, chúng ta sẽ sử dụng `onSubmit` listener trong `<form>` element

Thêm đoạn code: `onSubmit = {this.onSubmit}` vào thẻ `<form>` trong `AddTodo` component.

```
<form className="form-container" onSubmit={this.addTo} >
```

Và hàm xử lý khi `onSubmit` được gọi.

```
addTo = e => {
  e.preventDefault();
  console.log(this.state.title);
};
```

Đến đây, chúng ta mới chỉ đơn giản là in ra log giá trị mà người dùng đang nhập khi click nút "Submit". Việc cần làm bây giờ là "bắt" giá trị mà người dùng nhập đó lên `TodoApp` component để update vào state.

Cách làm tương tự như đã thực hiện với checkbox và nút delete vậy.

Bắt đầu từ `TodoApp` component (`TodoApp.js`). Thêm hàm `addTodo()` như dưới đây (đặt hàm này bên trên hàm `render()`).

```
addTodo = title => {  
  console.log(title);  
};
```

Tiếp theo cập nhật lại `<AddTodo>` tag trong hàm `render` trong `TodoApp` component.

```
render() {  
  return (  
    <div className="container">  
      <Header />  
      <AddTodo addTodo={this.addTodo} />  
      <Todos todos={this.state.todos}  
        handleChange={this.handleClickboxChange}  
        deleteTodo={this.deleteTodo}  
      />  
    </div>  
  );  
}
```

Quay trở lại `AddTodo` component (`AddTodo.js`). Chúng ta sẽ sửa lại đoạn mã xử lý khi người dùng click vào nút "Submit".

```
addTodo = e => {  
  e.preventDefault();  
  this.props.addTodo(this.state.title);  
};
```

Đến đây, bạn có thể chạy thử trên trình duyệt. Nếu màn hình console vẫn in ra được nội dung mà bạn nhập trong text input field là OK.

Nhưng có gì đó thiếu thiếu thì phải?

Giờ mình muốn thêm: sau khi click nút "Submit", giá trị trong text input field được chuyển xuống danh sách todos, đồng thời text input field cũng phải reset giá trị về rỗng.

Để reset giá trị trong text input field sau khi click nút "Submit", chúng ta thêm một đoạn code nhỏ trong hàm `addTodo()` là được (vẫn trong `AddTodo` component nhé)

```
addTodo = e => {  
  e.preventDefault();  
  this.props.addTodo(this.state.title);  
  this.setState({  
    title: ""  
  });  
};
```

Cuối cùng, để cập nhập danh sách todos, chúng ta thêm đoạn code để thêm phần tử vào mảng trong TodoApp component (*TodoApp.js*).

```
addTodo = title => {  
  const newTodo = {  
    id: 4,  
    title: title,  
    completed: false  
  };  
  this.setState({  
    todos: [...this.state.todos, newTodo]  
  });  
};
```

Lưu lại và kiểm tra trên trình duyệt.



Hình 4.21: Thêm mới một todo vào danh sách

Nhưng bạn có nhận thấy là mình đã hardcode giá trị ID không? Điều này là không nên. Vì ít nhất là lúc xóa Todo, chúng ta dựa vào ID để xóa chính xác Todo.

Do vậy, chúng ta cần phải tạo unique ID cho mỗi Todo.

Giải pháp của mình là sử dụng module "UUID".

Vẫn trong *TodoApp.js*, bạn import module "UUID".

```
import uuid from "uuid";
```

Sau đó, thay giá trị của id bằng hàm: *uuid.v4()* như dưới đây.

```

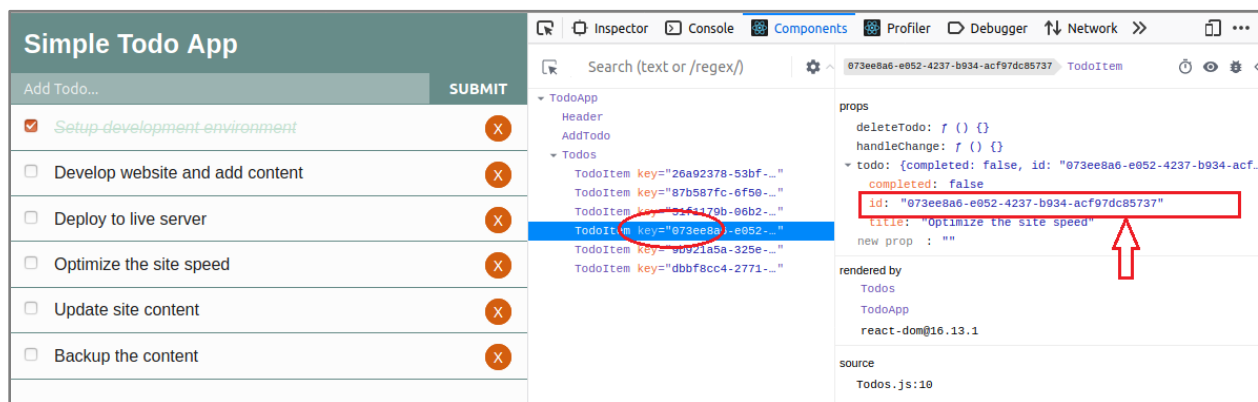
addTodo = title => {
  const newTodo = {
    id: uuid.v4(),
    title: title,
    completed: false
  };
  this.setState({
    todos: [...this.state.todos, newTodo]
  });
};

```



Nếu bạn gặp lỗi "uuid module not found" thì cần cài đặt UUID bổ sung bằng lệnh:
npm i uuid

Giờ thì lưu lại và kiểm tra trên trình duyệt.



Hình 4.22: Tạo Unique Id cho Todo

Xem online tại đây: <https://codesandbox.io/s/phan-4-them-mot-todo-moi-ug6mm>

Tổng kết

Qua phần này, chúng ta đã hiểu về Component, Props và State, những khái niệm rất quan trọng của React. Ngoài ra, bạn cũng đã biết cách truyền dữ liệu giữa các component, cập nhật giá trị state, xử lý sự kiện trong ứng dụng React.

Các bạn có thể tham khảo mã nguồn của phần này tại đây:

<https://github.com/vntalking/ebook-reactjs-that-don-gian/tree/master/phan4/todo-app>

Nếu bạn có bất kỳ thắc mắc hoặc chỗ nào chưa hiểu, đừng ngại liên hệ với mình qua:
support@vntalking.com.

FETCHING DỮ LIỆU TỪ API

Từ đầu tới giờ, ứng dụng Todo của chúng ta mới chỉ lưu dữ liệu vào biến State, đây có thể coi là biến tạm thời. Vì khi reload lại ứng dụng, dữ liệu sẽ bị reset lại từ đầu.

Thực tế, dữ liệu sẽ được lưu vào cơ sở dữ liệu (database) và đặt tại một máy chủ nào đó. Các ứng dụng front-end như Todo App sẽ chỉ có nhiệm vụ request tới máy chủ đó để đọc và ghi dữ liệu. Máy chủ cần phải cung cấp REST API để ứng dụng front-end như Todo App sử dụng.

Trong trường hợp ứng dụng Todo, những tác vụ sau có thể request tới API của máy chủ:

- Lấy danh sách Todos.
- Thêm một Todo mới.
- Xóa một Todo.

Các request tới máy chủ được thực hiện trên giao thức HTTP, nên nhiều khi có thể gọi là HTTP request.

Để tạo HTTP request tới REST API, chúng ta có thể sử dụng native fetch API hoặc dùng thư viện ngoài, phổ biến nhất là thư viện axios.

Lý thuyết cơ bản về Request Network

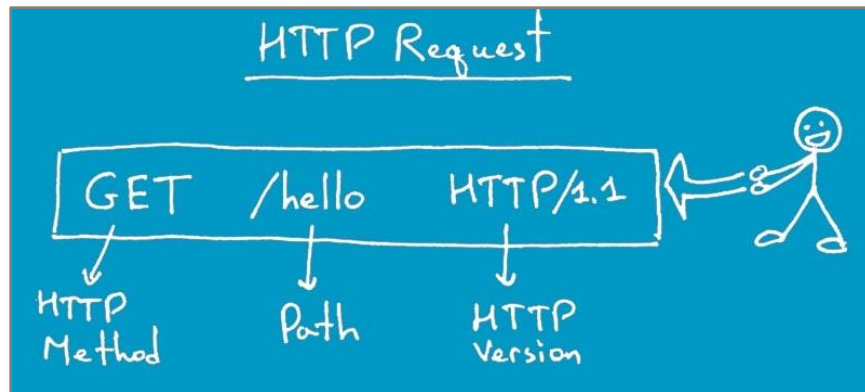
Phần này, chúng ta cùng tìm hiểu qua về HTTP request, Fetch API, Axios và REST API phục vụ cho việc xây dựng ứng dụng Todo.

Tìm hiểu HTTP request

Một HTTP request là một gói thông tin mà client gửi đến server, yêu cầu server làm một việc gì đó.

Request có thể được tạo bằng nhiều cách, trực tiếp hoặc gián tiếp.

Ví dụ: khi người dùng nhấp vào một liên kết trên trình duyệt. Hoặc một video được đính kèm trong một website và việc request đến website này cũng đồng nghĩa sẽ có một request tới video ấy.



Hình 5.1: HTTP request

Bắt đầu của HTTP Request sẽ là dòng Request-Line bao gồm 3 thông tin đó là:

- **Method:** là phương thức mà HTTP Request này sử dụng, thường là GET, POST. Ngoài ra còn một số phương thức khác như HEAD, PUT, DELETE, OPTION, CONNECT.
- **URI:** là địa chỉ định danh của tài nguyên. Trong trường hợp này URI là / - tức request cho tài nguyên gốc, nếu request không yêu cầu một tài nguyên cụ thể, URI có thể là dấu *.
- **HTTP version:** là phiên bản HTTP đang sử dụng, ở đây là HTTP 1.1.

Tiếp theo là các trường request-header, cho phép client gửi thêm các thông tin bổ sung về thông điệp HTTP request và về chính client. Một số trường thông dụng như:

- **Accept:** loại nội dung có thể nhận được từ thông điệp response. Ví dụ: text/plain, text/html...
- **Accept-Encoding:** các kiểu nén được chấp nhận. Ví dụ: gzip, deflate, xz, exi...
- **Connection:** tùy chọn điều khiển cho kết nối hiện thời. Ví dụ: keep-alive, Upgrade...
- **Cookie:** thông tin HTTP Cookie từ server.
- **User-Agent:** thông tin về user agent của người dùng

Fetch API

Fetch API cho phép gửi một yêu cầu tới máy chủ và xử lý kết quả được trả về từ máy chủ.

Bạn có thể sử dụng fetch API mặc định có sẵn của Javascript. Nhưng cá nhân mình thì thích sử dụng thư viện hỗ trợ cho việc này hơn. Phổ biến trong số đó là axios.

Axios

Axios là một HTTP client được viết dựa trên Promises được dùng để hỗ trợ cho việc xây dựng các ứng dụng API từ đơn giản đến phức tạp và có thể được sử dụng ở trình duyệt hoặc Node.js

Một trong những lợi thế khi sử dụng Axios đó là nó hỗ trợ trên hầu hết các trình duyệt, bao gồm cả những trình duyệt cổ điển như IE8.

Việc hỗ trợ bởi các trình duyệt là điểm bạn cần cân nhắc khi sử dụng Axios hay fetch API mặc định.

Repository chính thức của Axios: <https://github.com/axios/axios>

REST API

REST API là một tiêu chuẩn dùng trong việc thiết kế API cho các ứng dụng web hay web services để tiện cho việc quản lý các resource và được truyền tải qua HTTP.

Hiểu REST API là cần thiết nếu bạn muốn lấy dữ liệu từ các nguồn khác nhau trên internet. Chẳng hạn, muốn lấy danh sách các todos khi bạn request tới một URL cụ thể.

Đến đây, chúng ta đã hiểu cơ bản về những khái niệm cần thiết được sử dụng trong ứng dụng Todo App. Tiếp theo đây, mình sẽ giới thiệu phần máy chủ phục vụ cho ứng dụng Todo.

Giới thiệu API cung cấp cho ứng dụng Todo

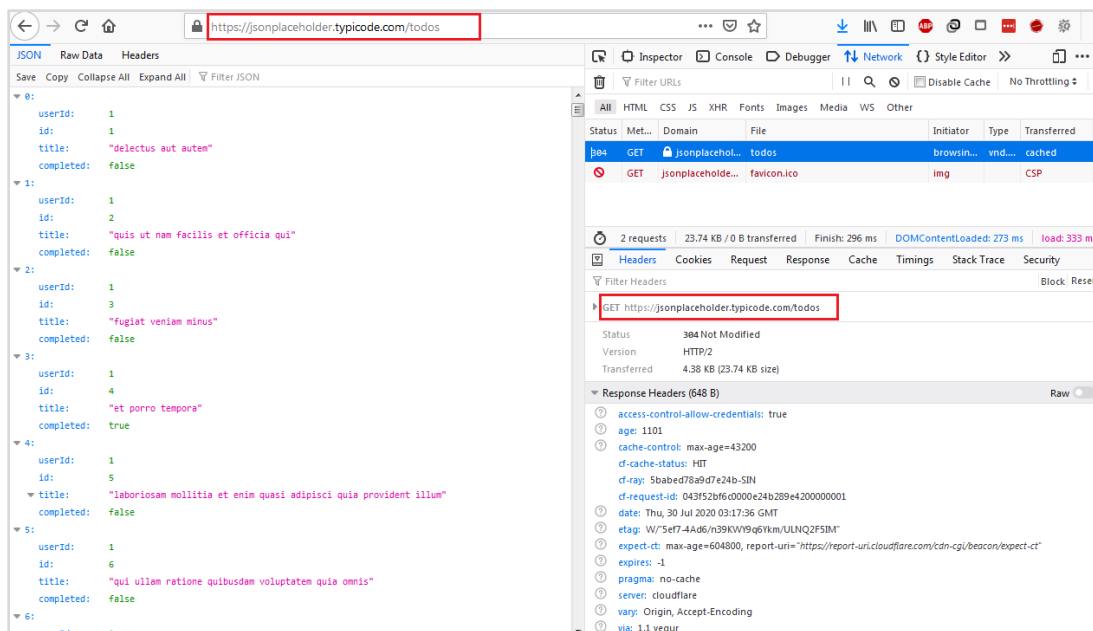
Về nguyên tắc thì chúng ta sẽ cần tự xây dựng một web service trên một máy chủ để lưu trữ và quản lý các Todo, sau đó cung cấp REST API để tương tác với ứng dụng Todo.

Trong khuôn khổ cuốn sách này, mình sẽ không hướng dẫn các bạn các xây dựng web service này, cũng như cách tạo các REST API. Thay vào đó, mình sẽ sử dụng luôn một dịch vụ fake REST API online: <https://jsonplaceholder.typicode.com/>

Dưới đây là các APIs sử dụng trong ứng dụng Todo:

Chức năng	Methods	URI
Lấy danh sách các Todos	GET	https://jsonplaceholder.typicode.com/todos
Thêm một Todo mới	POST	https://jsonplaceholder.typicode.com/todos
Xóa một Todo	DELETE	https://jsonplaceholder.typicode.com/todos/\${id}

Bạn có thể kiểm tra các API này qua các công cụ như Postman. Với API có phương thức GET thì kiểm tra luôn trên trình duyệt cũng được. Ví dụ như API lấy danh sách todos nhé.



Hình 5.2: Kết quả trả về của API

Phần API coi như đã xong, phần tiếp theo, chúng ta sẽ cần fetching danh sách todos lúc bắt đầu vào ứng dụng Todo. Để làm điều này, chúng ta cần hiểu và sử dụng đến các hàm liên quan tới vòng đời component (life-cycle method).

Vòng đời Component trong React

Tất cả các component bạn tạo ra sẽ phải trải qua một loạt các events hoặc các giai đoạn từ khi được gọi tới khi bị hủy.

Cũng giống cuộc sống của con người vậy, ai cũng phải trải qua đủ các giai đoạn từ khi sinh ra, đến lúc trưởng thành rồi "về với đất mẹ".

Trong React, một component sẽ phải trải qua 3 giai đoạn chính:

1. Mounting: Đúng như tên gọi của nó, đây là giai đoạn mà Component được tạo và thêm vào cây DOM. Đây có thể coi là giai đoạn bắt đầu của một Component. Các methods được gọi trong giai đoạn này:

- `componentWillMount()`
- `render()`
- `componentDidMount()`.

2. Updating: Sau khi Component qua giai đoạn đầu tiên, tức là đã được thêm vào DOM, giai đoạn tiếp theo là cập nhập nếu có yêu cầu. Component sẽ được update khi giá trị state hoặc props thay đổi, lúc này Component sẽ được render lại. Các methods trong giai đoạn này gồm:

- `componentWillReceiveProps()`
- `shouldComponentUpdate()`
- `componentWillUpdate()`
- `render()`
- `componentDidUpdate()`

3. Unmounting: Đây là giai đoạn kết thúc vòng đời của Component, khi nó được xóa khỏi DOM. Chỉ có duy nhất 1 method trong giai đoạn này:

- `componentWillUnmount()`

Như bạn thấy, trong mỗi giai đoạn, React sẽ cung cấp các life-cycle methods để theo dõi và thao tác với những gì xảy ra trong component.

Từ đầu cuốn sách đến giờ, có lẽ bạn gặp nhiều nhất là hàm `render()`.

Đối với người mới tiếp cận React, ngoài `render()` là dùng nhiều nhất, bạn còn sử dụng hàm `componentDidMount()` nhiều không kém. Chủ yếu là lúc bạn cần tạo một HTTP request để lấy dữ liệu ban đầu.

Ok, giờ chúng ta quay trở lại với ứng dụng Todo nhé.

Cài đặt axios

Như mình đã nói ở trên, chúng ta sẽ sử dụng thư viện axios cho dự án này. Do vậy, chúng ta sẽ cần cài axios đã nhé.

Từ cửa sổ terminal của visual code, bạn gõ lệnh cài axios qua npm:

```
npm install axios
```

Sau khi cài đặt thành công, npm sẽ tự động thêm vào `package.json` như dưới đây:

```
"dependencies": {  
  "axios": "^0.19.2",  
  "react": "^16.13.1",  
  ...  
},
```

Lấy danh sách Todos bằng GET request

Vì chúng ta muốn tải danh sách todos về mỗi khi mở ứng dụng Todo, do đó, phương thức `componentDidMount()` là nơi thích hợp nhất để gọi http request.

Đầu tiên, mở `TodoApp.js` và import thư viện axios:

```
//khai báo thư viện axios
import axios from "axios";
```

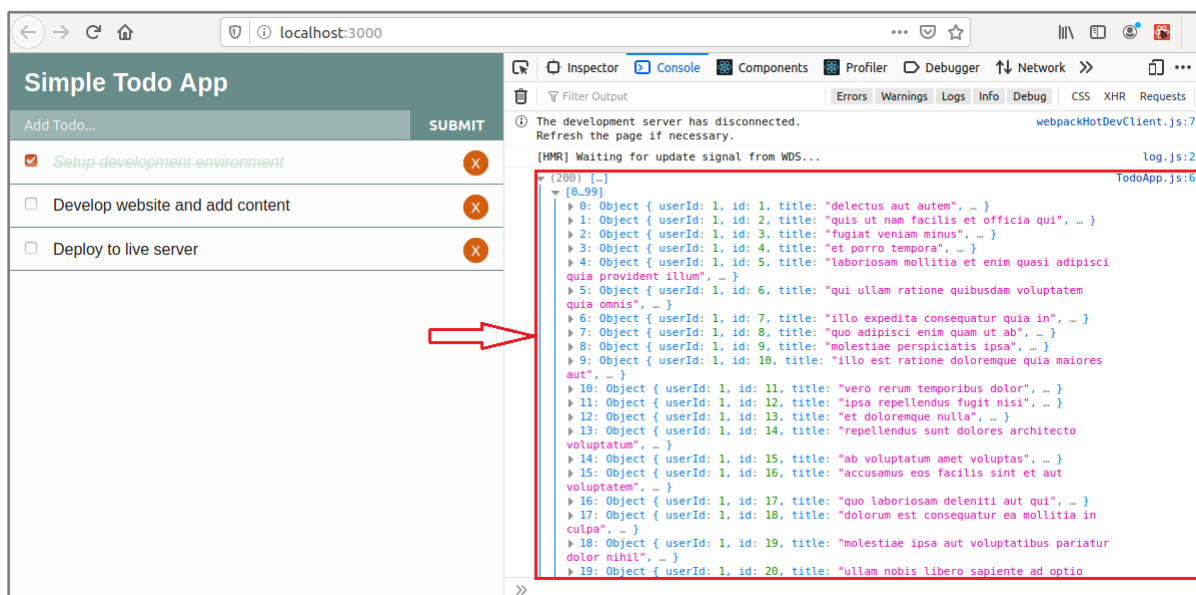
Sau đó thêm đoạn code này ngay phía trên hàm `render()`

```
componentDidMount() {
  //tạo GET request để lấy danh sách todos
  axios.get("https://jsonplaceholder.typicode.com/todos")
    .then(response => console.log(response.data));
}
```

Lưu lại và kiểm tra trong console tab.



Do lúc này ứng dụng chúng ta đang kết nối tới một máy chủ trên internet, nên đảm bảo máy tính của bạn có kết nối mạng nhé.



Hình 5.3: Danh sách todos được lấy từ máy chủ.

Nếu bạn thấy dữ liệu trả về nhiều quá không cần thiết cho dự án kiểu simple thế này, hoặc sau này bạn làm thêm tính năng phân trang... bạn có thể giới hạn số lượng todo trả về cho mỗi lần request bằng cách thêm `param_limit`.

Có 2 cách để làm điều này:

Cách 1. Thêm param trực tiếp vào URL:

```
axios.get("https://jsonplaceholder.typicode.com/todos?_limit=10")
```

Cách 2: Tạo một đối tượng config và truyền vào hàm axios.get()

```
componentDidMount() {  
  const config = {  
    params: {  
      _limit: 5  
    }  
  }  
  //tạo GET request để lấy danh sách todos  
  axios.get("https://jsonplaceholder.typicode.com/todos", config)  
    .then(response => console.log(response.data));  
}
```

Cá nhân thì mình thích cách làm thứ 2 hơn, vì nó linh động hơn cách 1. Với cách 2, ngoài việc muốn thêm *param*, bạn còn có thể cấu hình cả những thông số như *header request*...

Công việc tiếp theo, sau khi đã nhận được danh sách todos từ server, thay vì chỉ in ra màn hình console, chúng ta sẽ gán dữ liệu đó vào state.

Giá trị state lúc trước là chúng ta fix cứng thôi, giờ thì giá trị đó được lấy về từ server nên chúng ta sẽ xóa hết những giá trị cũ đó (*TodoApp.js*).

```
state = {  
  todos: []  
};
```

Cập nhật lại đoạn mã để gán trị cho state (vẫn trong *TodoApp.js*)

```
componentDidMount() {  
  const config = {  
    params: {  
      _limit: 5  
    }  
  }  
  //tạo GET request để lấy danh sách todos  
  axios.get("https://jsonplaceholder.typicode.com/todos", config)  
    .then(response => this.setState({ todos: response.data }));  
}
```

Lưu lại và xem kết quả. Xem online tại đây: <https://codesandbox.io/s/pensive-chaum-bl8b5>



Simple Todo App	
Add Todo...	SUBMIT
<input type="checkbox"/> delectus aut autem	X
<input type="checkbox"/> quis ut nam facilis et officia qui	X
<input type="checkbox"/> fugiat veniam minus	X
<input checked="" type="checkbox"/> et porro tempora	X
<input type="checkbox"/> laboriosam mollitia et enim quasi adipisci quia provident illum	X

Hình 5.4: Hiển thị danh sách Todos lấy từ máy chủ về.

Nếu bạn thấy giao diện bị tràn chữ vì nội dung todo server trả về quá dài thì có thể thêm đoạn CSS sau để cắt bớt. Trong `App.css` thêm đoạn sau:

```
.todo-item span {
  white-space: nowrap;
  overflow: hidden;
  text-overflow: ellipsis;
  width: 80%;
  height: 18px;
  display: inline-block;
}
```

Thêm Todo bằng POST request

Với tính năng thêm Todo, chúng ta phải sử dụng phương thức POST. Mục tiêu mong muốn là tạo một Todo mới, gửi lên máy chủ thông qua POST request, sau đó máy chủ phản hồi và cập nhật lên UI.

Tuy nhiên, dịch vụ JSONPlaceholder lại không có lưu giá trị vào cơ sở dữ liệu của họ, mà chỉ trả về một phản hồi là lưu thành công - Hàng fake mà (T_T). Thôi thì phần này, khi nhận được phản hồi thành công thì coi như đã thêm được Todo mới rồi.

Vẫn trong *TodoApp.js*, sửa lại mã code của hàm *addTodo()* để gọi axios

```
addTodo = title => {
  const todoData = {
    title: title,
    completed: false
  }
  axios.post("https://jsonplaceholder.typicode.com/todos", todoData)
    .then(response => {
      console.log(response.data)
      this.setState({
        todos: [...this.state.todos, response.data]
      })
    })
};
```



Do trong dữ liệu trả về của REST API đã có giá trị unique Id rồi, nên chúng ta không cần dùng tới thư viện UUID nữa. Giờ bạn có thể xóa nó đi nếu gặp lỗi compile.

Lưu lại và kiểm tra trên trình duyệt thôi. Hoặc xem online tại đây:

<https://codesandbox.io/s/hardcore-wilbur-5ngky>

Xóa một Todo bằng DELETE request

Tương tự như GET hay POST request, việc xóa một todo trong danh sách, chúng ta sẽ dùng đến delete request.

Cách làm hoàn toàn tương tự như trước đó. Vẫn trong *TodoApp.js*, cập nhật lại mã hàm *deleteTodo()*

```
deleteTodo = id => {
  axios.delete(`https://jsonplaceholder.typicode.com/todos/${id}`)
    .then(reponse => this.setState({
      todos: [
        ...this.state.todos.filter(todo => {
          return todo.id !== id;
        })
      ]
    )))
};
```

Lưu lại và kiểm tra trên trình duyệt. Bạn thử xóa một todo xem có ổn không? Nó chạy như mong muốn đúng không nhé!?

Xem online tại đây: <https://codesandbox.io/s/determined-elion-nrpg7>

Tổng kết

Qua phần này, chúng ta đã biết cách tạo request tới các API. Các ứng dụng front-end hiện đại thường sẽ phải để dữ liệu trên một máy chủ, hoặc để máy chủ thực hiện các tác vụ chuyên biệt nào đó, còn nhiệm vụ của front-end chỉ là tương tác với người dùng, nhận và hiển thị dữ liệu - không làm gì to tát hơn nữa.

Ngoài ra, chúng ta cũng đã biết cách sử dụng thư viện axios, một thư viện hỗ trợ thực hiện request network mạnh mẽ.

Các bạn có thể tham khảo mã nguồn của phần này tại đây:

<https://github.com/vntalking/ebook-reactjs-that-don-gian/tree/master/phan5/todo-app>

Nếu bạn có bất kỳ thắc mắc hoặc chỗ nào chưa hiểu, đừng ngại liên hệ với mình qua:

support@vntalking.com.

REACT HOOKS

Như trong phần tìm hiểu về Component, chúng ta biết rằng, có 2 kiểu viết component: Class Component và functional Component.

Trong đó, functional Component có những hạn chế nhất định như: không có state, không có life-cycle method...

Nhưng functional component có ưu điểm là dễ unit test hơn, phù hợp với trường phái viết code theo kiểu hướng function thay vì hướng đối tượng (OOP).

React Hooks ra đời để khắc phục những nhược điểm của functional component và còn hơn thế nữa.

React Hooks cho phép chúng ta sử dụng state và các tính năng khác của React mà không phải dùng đến Class.

Có thể thấy, các nhà phát triển React họ đang muốn hướng đến 1 tương lai **Functional Programming** thay vì sử dụng những Class mà chỉ nghe cái tên thôi là ta đã nghĩ ngay đến OOP (lập trình hướng đối tượng). Cộng với việc không sử dụng Class kế thừa từ React Component nữa nên giờ đây kích thước bundle sẽ được giảm đáng kể.

Tìm hiểu Hooks

Sử dụng Hook sẽ có nhiều thay đổi so với sử dụng Class component. Tuy nhiên, chúng ta chỉ cần hiểu và biết cách sử dụng Hooks vào 2 thứ: **State** và **lifecycle methods (vòng đời component)**.

Khởi tạo State bằng Hook

Chúng ta sử dụng hook có tên là `useState()` để khai báo và cập nhật giá trị State.

`useState()` trả về 1 cặp, gồm:

- Giá trị của state hiện tại.
- Phương thức để cập nhật state đó.

Để hiểu hơn về hook này, chúng ta cùng xem ví dụ dưới đây:

```
import React, { useState } from 'react';
function ViduHook() {
  //Khai báo một biến trạng thái mới, đặt tên là là "count"
  const [count, setCount] = useState(0);

  const handleChangeCount = () => {
    setCount(count + 1);
  }
  return (
    <div>
      <p>Bạn đã click {count} lần</p>
      <button onClick={handleChangeCount}>
        Click vào đây
      </button>
    </div>
  );
}
```

Hàm `setCount` có tác dụng như khi dùng `this.setState()` trong class component. Chúng giúp chúng ta update giá trị của state khi cần.

Còn giá trị "0" truyền vào hook `useState(0)` chỉ là giá trị khởi tạo mặc định ban đầu của state count.

Với đoạn code ví dụ trên, nếu bạn viết theo kiểu class component thì sẽ như sau:

```
class ViduHook extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }
  handleChangeCount = () => {
    this.setState({ count: this.state.count + 1})
  }
  render() {
    return (
      <div>
        <p>Bạn đã click {this.state.count} lần</p>
        <button onClick={handleChangeCount}>
          Click vào đây
        </button>
      </div>
    );
  }
}
```

Vậy là chúng ta có thể sử dụng state trong functional component một cách rất dễ dàng đúng không!

Ngoài ra nếu bạn cần nhiều state hơn thì có thể sử dụng cú pháp tương tự để khai báo.

```
...
const [count, setCount] = useState(0);
const [name, setName] = useState('SonDuong');
const [age, setAge] = useState(30);
...
```

NHƯNG? Nếu mình có khoảng 100 state cần khai báo thì sao? Nếu viết như này thì cũng "tù túng" nhỉ!?

Cách khai báo state trên chỉ phù hợp khi component của bạn có ít state thôi, tầm một hoặc hai state.

Còn khi bạn có nhiều state hơn thì phải dùng cách khác. Rất may là Hooks cũng có giải pháp để biến nó vừa ngắn vừa rất giống với cách khai báo state trong class component vậy.

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

function ViduHook() {
  //Khai báo một biến trạng thái mới, đặt tên là là "count"
  const [state, setState] = useState({
    count: 0,
    name: 'SonDuong'
  });

  const handleChangeCount = () => {
    setState({
      count: state.count + 1,
      name: "vntalking"
    });
  }

  return (
    <div>
      <p>Bạn đã click {count} lần</p>
      <button onClick={handleChangeCount}>
        Click vào đây
      </button>
    </div>
  );
}
```

Sử dụng Hook thay thế cho life-cycle methods

Như trong phần tìm hiểu về vòng đời của Component trong ứng dụng React, chắc hẳn bạn vẫn còn nhớ những hàm như:

- `componentDidMount()`
- `componentDidUpdate()`
- `componentWillUnmount()`
- v.v...

giờ chúng được gói gọn trong một hook duy nhất, đó là `useEffect()`

Chúng ta sẽ cùng nhau tìm hiểu cách sử dụng hook `useEffect()` này như nào nhé? Xem nó xử lý tương ứng với các hàm vòng đời ra sao?

1. `ComponentDidMount()`

```
useEffect(() => {  
  // Bạn viết code xử lý logic tại đây  
}, []);
```

Cách viết này sẽ xử lý tương tự `componentDidMount()`. Tức nó được gọi một lần khi component được mounted.

Chúng ta chú ý vào tham số thứ 2: `[]` là một array rỗng. Điều này khiến đoạn code bên trong hàm `useEffect()` chỉ chạy đúng một lần.

2. `ComponentDidUpdate()`

```
useEffect(() => {  
  // Bạn viết code xử lý logic tại đây  
});
```

Khi bạn không truyền tham số thứ 2 trong hàm `useEffect()`, lúc này đoạn code bên trong sẽ chạy mỗi khi component được render. Điều này làm nó giống với hàm `componentDidUpdate()`.

3. `ComponentWillUnmount()`

```
useEffect(() => {  
  // hàm được trả về sẽ được gọi khi component unmount  
  return () => {  
    // Bạn viết code xử lý logic tại đây khi component unmount.  
  }  
}, [])
```

Khi `useEffect()` return về 1 function thì hàm được return đó sẽ có tác dụng giống như `componentWillMount()`. Tức hàm đó sẽ được gọi khi component unmount.

Như vậy là chúng ta đã hiểu cơ bản về React Hooks rồi đúng không? Đến đây, bạn hoàn toàn có đủ nền tảng để khám phá những hooks khác khi dự án thực tế yêu cầu.

Refactoring mã nguồn Todo App sử dụng Hooks

Quay trở lại ứng dụng Todo, để áp dụng những kiến thức mới học về React Hooks, chúng ta sẽ cùng nhau thay đổi mã nguồn để sử dụng Hook nhé.

Xem lại mã nguồn của ứng dụng Todo hiện tại, chúng ta thấy có 2 component có sử dụng đến State là: `AddTodo` và `TodoApp`.

Chúng ta sẽ cùng nhau tiến hành chuyển 2 component trên thành functional component và sử dụng Hooks nhé. Còn các component khác thì để lại, nếu bạn thích thì có thể tự thực hành chuyển nhé.

Đầu tiên là `AddTodo.js`. Do component này chỉ có 1 state là title nên mình sẽ dùng cách khai báo thứ nhất.

```
import React, { useState } from "react";

function AddTodo (props) {
  const [title, setTitle] = useState("");

  const onInputChange = e => {
    setTitle(e.target.value)
  };

  const addTodo = e => {
    e.preventDefault();
    props.addTodo(title);
    setTitle("");
  };

  return (
    <form className="form-container" onSubmit={addTodo}>
      <input
        type="text"
        placeholder="Add Todo..."
        className="input-text"
        value={title}
        onChange={onInputChange}
      />
      <input
        type="submit"

```

```

        value="Submit"
        className="input-submit"/>
    </form>
  );
}
export default AddTodo;

```

Khi bạn chuyển sang sử dụng functional component, việc sử dụng props sẽ không thông qua từ khóa *this* mà sẽ là truyền qua tham số của hàm:

function AddTodo (props)

Đến TodoApp component (*TodoApp.js*) thì phức tạp hơn một chút vì chúng ta xử lý nhiều logic trong này. Nhưng không sao, cứ lần lượt làm thôi.

Trong *TodoApp.js*, đầu tiên là import hai hooks: *useState*, *useEffect*

```
import React, { useState, useEffect } from "react";
```

Tiếp theo là chuyển Class component thành functional component. Khi chuyển thành functional component, chúng ta không cần đến hàm *render()* nữa, và chuyển nó thành *return(...)*

```

function TodoApp() {
  ...
  return (
    <div className="container">
      <Header />
      <AddTodo addTodo={this.addTodo} />
      <Todos todos={this.state.todos}
        handleChange={this.handleCheckboxChange}
        deleteTodo={this.deleteTodo}
      />
    </div>
  );
}

```

Tiếp theo đến phần khai báo State, lần này thì mình sẽ dùng cách khai báo State thứ 2 (kiểu khai báo hỗ trợ nhiều state như mình đã nói ở trên).

```

const [state, setState] = useState({
  todos: []
});

```

Sau đó thì bạn bỏ hết từ khóa *this* trong code đi, vì *this* chỉ dùng cho class thôi. Ví dụ như hàm này:

```
const handleCheckboxChange = id => {
  setState({
    todos: state.todos.map(todo => {
      if (todo.id === id) {
        todo.completed = !todo.completed;
      }
      return todo;
    })
  });
};
```



Nhớ thêm từ khóa `const` trước tên các hàm nhé.

Trong `TodoApp` component có một chỗ cần phải làm việc với life-cycle, cụ thể là `componentDidMount()`, đó là khi truy cập vào ứng dụng thì tải danh sách `todo` từ API về.

```
componentDidMount() {
  ...
}
```

Với `React Hooks`, chúng ta sẽ sử dụng `useEffect`. Do vậy, hàm `componentDidMount()` chuyển thành như sau:

```
useEffect(() => {
  const config = {
    params: {
      _limit: 5
    }
  }

  //tạo GET request để lấy danh sách todos
  axios.get("https://jsonplaceholder.typicode.com/todos", config)
    .then(response => setState({ todos: response.data }));
}, []);
```

Kết quả cuối cùng thì toàn bộ `TodoApp` component sẽ như sau:

```
import React, { useState, useEffect } from "react";
import Todos from "../Todos";
import Header from "../components/layout/Header";
import AddTodo from "../AddTodo"

//khai báo thư viện axios
import axios from "axios";

function TodoApp() {
  const [state, setState] = useState({
```



```

    todos: []
  });
  const handleCheckboxChange = id => {
    setState({
      todos: state.todos.map(todo => {
        if (todo.id === id) {
          todo.completed = !todo.completed;
        }
        return todo;
      })
    });
  };
  const deleteTodo = id => {
    axios.delete(`https://jsonplaceholder.typicode.com/todos/${id}`)
      .then(reponse => setState({
        todos: [
          ...state.todos.filter(todo => {
            return todo.id !== id;
          })
        ]
      })))
  };

  const addTodo = title => {
    const todoData = {
      title: title,
      completed: false
    }
    axios.post("https://jsonplaceholder.typicode.com/todos", todoData)
      .then(response => {
        console.log(response.data)
        setState({
          todos: [...state.todos, response.data]
        })
      });
  };

  useEffect(() => {
    const config = {
      params: {
        _limit: 5
      }
    }

    // tạo GET request để lấy danh sách todos
    axios.get("https://jsonplaceholder.typicode.com/todos", config)
      .then(response => setState({ todos: response.data }));
  }, []);

```

```

    return (
      <div className="container">
        <Header />
        <AddTodo addTo={addTo} />
        <Todos todos={state.todos}
          handleChange={handleCheckboxChange}
          deleteTodo={deleteTodo} />
      </div>
    );
  }
  export default TodoApp;

```

Vậy là xong rồi đấy. Bạn lưu tất cả và kiểm tra trên trình duyệt xem ứng dụng đã chạy giống như cũ chưa nhé. Xem online tại đây: <https://codesandbox.io/s/wizardly-shockley-5qck0>

Tổng kết

Như vậy, qua phần này chúng ta đã biết Hooks là gì và ứng dụng của Hooks trong dự án React. Theo thông tin "năm vùng", có vẻ đội ngũ phát triển React đang khuyến khích sử dụng Hooks thay vì Class component.

Các bạn có thể tham khảo mã nguồn của phần này tại đây:

<https://github.com/vntalking/ebook-reactjs-that-don-gian/tree/master/phan6/todo-app>

Nếu bạn có bất kỳ thắc mắc hoặc chỗ nào chưa hiểu, đừng ngại liên hệ với mình qua:

support@vntalking.com.

QUẢN LÝ STATE VỚI REDUX

Redux là một thư viện Javascript giúp quản lý State của ứng dụng.

Khi mà ứng dụng ngày càng phức tạp, state ngày càng nhiều hơn... đó là lúc chúng ta cần phải quản lý state.

State có thể bao gồm:

- Dữ liệu trả về từ phía máy chủ và được cached lại.
- Hoặc dữ liệu được tạo ra và thao tác ở phía client nhưng chưa được đẩy lên phía server.
- UI state cho các ứng dụng phức tạp. Chúng ta cần quản lý việc active Routes, selected tabs, spinners, điều khiển phân trang.v.v...

Để giải quyết những khó khăn khi quản lý State, Redux ra đời. Redux nổi lên như 1 hiện tượng, nó thậm chí thay thế luôn kiến trúc Flux của Facebook dùng cho React. Hiện tại Facebook cũng khuyến cáo các lập trình viên chuyển qua dùng Redux vì nhiều ưu điểm được cải tiến so với Flux.

3 nguyên tắc của Redux

Khi sử dụng Redux, bạn cần ghi nhớ kỹ 3 nguyên tắc sau:

- **Chỉ có một nguồn dữ liệu tin cậy duy nhất:** Tức là State của toàn bộ ứng dụng được lưu trong 1 store duy nhất. Đó là 1 Object theo mô hình cây (tree).
- **State chỉ được phép đọc:** Tức là chỉ có 1 cách duy nhất để thay đổi giá trị state. Đó là tạo ra một ACTION (cũng là 1 object để mô tả những gì xảy ra).
- **Thay đổi chỉ bằng hàm JS thuần túy:** Để chỉ ra cách mà State được biến đổi bởi Action chúng ta dùng các pure function gọi là Reducer.

Khi nào thì sử dụng Redux?

Về bản chất thì Redux cũng chỉ là một công cụ giúp bạn quản lý State hiệu quả hơn. Do vậy, mỗi người, mỗi dự án lại có đặc điểm riêng mà bạn sử dụng Redux hợp lý.

Về cá nhân mình thì coi Redux để gom các State dùng chung về một "tổng kho". Tức là sẽ có một nơi để lưu các State được dùng trong toàn bộ ứng dụng. Ví dụ: trạng thái login, theme.v.v... Các state được lưu bằng redux có thể gọi là global state.

Nhiệm vụ của Redux đơn giản là lưu giá trị vào global state đó, hỗ trợ lấy giá trị ra để sử dụng. Còn việc sử dụng giá trị của global state đó như thế nào? dùng làm gì?... đó không phải là nhiệm vụ của Redux.

Thành phần của Redux

Về cơ bản, Redux có những thành phần sau:

1. Actions

Trong Redux, Action là nơi mang các thông tin để gửi từ ứng dụng đến store.

Action là 1 đối tượng định nghĩa 2 thuộc tính:

- **type**: kiểu mô tả action.
- **payload**: giá trị tham số truyền lên (không bắt buộc).

Ví dụ một action:

```
{
  type: 'ADD_TODO',
  text: 'Optimize the site speed'
}
```

2. Reducers

Action có nhiệm vụ mô tả những gì xảy ra nhưng lại không chỉ rõ cụ thể phần state nào thay đổi. Việc này sẽ do Reducer đảm nhiệm.

Reducer nhận 2 tham số đầu vào:

- State cũ.
- Action được gửi lên.

Khi thao tác với state, reducer không được phép sửa đổi state. Thay vào đó, reducer dựa trên giá trị state cũ, cộng với dữ liệu đầu vào để tạo nên đối tượng state mới.

3. Store

Store thực chất là 1 đối tượng để lưu trữ state của toàn bộ ứng dụng.

Store có 3 phương thức sau:

- **getState():** Truy cập vào state và lấy ra state hiện tại.
- **dispatch(action):** Thực hiện gọi 1 action.
- **subscribe(listener):** Nó có vai trò cực quan trọng, luôn luôn lắng nghe xem có thay đổi gì không rồi ngay lập tức cập nhật ra View.

Đến đây, chúng ta cũng hiểu cơ bản về Redux rồi, giờ mình sẽ áp dụng Redux vào dự án xây dựng ứng dụng Todo.

Thực hành sử dụng Redux trong Todo App

Trong ứng dụng Todo, chúng ta thêm tính năng mới là thay đổi theme. Tức là có thể thay đổi màu của toàn ứng dụng.

Giá trị mã màu sẽ được lưu trong global state, và được quản lý bởi redux.

Giao diện ứng dụng khi thêm tính năng thay đổi theme như sau:

The screenshot shows a web application titled "Simple Todo App". It features a header with the title, a text input field labeled "Add Todo...", and a "SUBMIT" button. Below the input is a list of five todo items, each with a checkbox and a delete button (an orange circle with an 'X'). The first four items are unchecked, and the fifth is checked. The checked item, "et porro tempora", is highlighted in green. At the bottom, there is a "Choose Theme" section with three radio buttons: a red one (selected), a blue one, and a grey one.

Simple Todo App	
Add Todo...	SUBMIT
<input type="checkbox"/> delectus aut autem	
<input type="checkbox"/> quis ut nam facilis et officia qui	
<input type="checkbox"/> fugiat veniam minus	
<input checked="" type="checkbox"/> et porro tempora	
<input type="checkbox"/> laboriosam mollitia et enim quasi adipisci quia provident illum	
Choose Theme <input checked="" type="radio"/> <input type="radio"/> <input type="radio"/>	

Hình 7.1: Tính năng thay đổi theme

Trước khi đi vào thực hành sử dụng Redux, chúng ta cần tạo sẵn giao diện phần Footer, nơi để người dùng thay đổi theme. Cách thực hiện tương tự như đã làm với các phần trước, chúng ta sẽ đi nhanh nhé.

Tạo giao diện tính năng thay đổi theme

Trong thư mục component, tạo thêm một component: *Footer.js* có nội dung như sau:

```
import React from "react";

const RED = "#ff0000";
const BLUE = "#0000ff";
const GRAY = "#678c89";

class Footer extends React.Component {
  constructor(props) {
    super(props)

    this.submitThemeColor = this.submitThemeColor.bind(this)
  }

  submitThemeColor(color) {
    // lưu giá trị mã màu theme vào Store - redux
    // Xử lý sau
  };

  render() {
    return (
      <div className="footer">
        <div className="vertical-center">
          <span>Choose Theme </span>
          <button onClick={()=>this.submitThemeColor(RED)} className="
dot red"/>
          <button onClick={()=>this.submitThemeColor(BLUE)} className="
"dot blue"/>
          <button onClick={()=>this.submitThemeColor(GRAY)} className="
"dot gray"/>
        </div>
      </div>
    );
  }
};
export default Footer;
```

Sau đó thêm Footer vào TodoApp component. Mở *TodoApp.js*

```
import Footer from "../components/layout/Footer";
...

return (
  <div className="container">
    <Header />
    <AddTodo addTo={addTo} />
    <Todos todos={state.todos}
      handleChange={handleCheckboxChange}
      deleteTodo={deleteTodo} />
    <Footer/>
  </div>
);
```

Thêm style CSS cho phần Footer. Mở *App.css* và thêm đoạn CSS sau vào bên dưới cùng:

```
.footer {
  height: 50px;
  background: #678c89;
  color: white;
  position: relative;
}
.vertical-center {
  margin: 0 0 0 10px;
  position: absolute;
  top: 50%;
  -ms-transform: translateY(-50%);
  transform: translateY(-50%);
}
.dot {
  height: 25px;
  width: 25px;
  border-radius: 50%;
  display: inline-block;
  margin-left: 5px;
  margin-right: 5px;
}
.red {
  background-color: #ff0000;
}
.blue {
  background-color: #0000ff;
}
.gray {
  background-color: #678c89;
}
```

Để phục vụ việc thay đổi theme, chúng ta sẽ cần sử dụng variable css. Tức là có thể thay đổi giá trị CSS bằng javascript.

Trong *App.css*, chúng ta khai báo thêm variable sau:

```
:root {  
  --main-color: #678c89;  
}
```

Sau đó, sửa lại đoạn css thiết lập background của những nơi muốn thay đổi màu tự động.

```
.header-container {  
  background-color: var(--main-color);  
  color: #fff;  
  padding: 10px 15px;  
}  
.container {  
  max-width: 600px;  
  margin: 0 auto;  
  border: 1px solid var(--main-color);  
}  
.todo-item {  
  list-style-type: none;  
  padding: 10px 15px;  
  border-top: 1px solid var(--main-color);  
}  
  
.footer {  
  height: 50px;  
  background: var(--main-color);  
  color: white;  
  position: relative;  
}
```

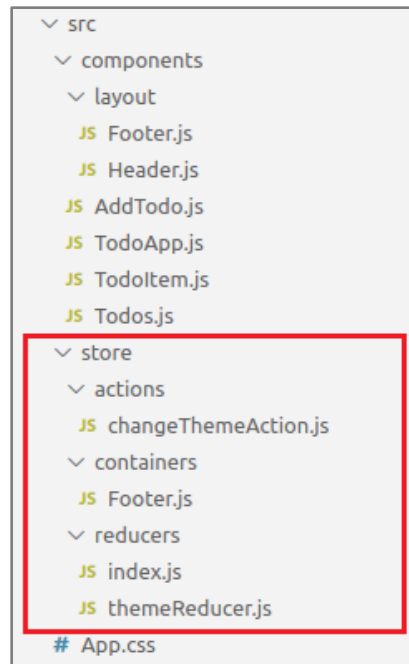
Như vậy là đã xong phần giao diện. Phần tiếp theo là sử dụng Redux.

Việc đầu tiên là cài đặt thư viện redux cho dự án Todo đã. Từ terminal, bạn gõ lệnh:

```
$ npm install --save redux react-redux
```

Dựa vào các thành phần của Redux, chúng ta sẽ tạo các thư mục tương ứng trong mã nguồn để tiện quản lý và theo dõi.

Chúng ta sẽ tạo thêm thư mục *store*, trong đó gồm: *actions*, *reducers* và *containers*. Và các files tương ứng như hình bên dưới.



Hình 7.2: Tạo thư mục cho Redux

Có thể bạn sẽ thắc mắc, tại sao lại tạo thêm thư mục *containers* ? dùng để làm gì?

Mình xin được giải thích ngắn gọn là: nó cũng tương tự như component nhưng khác ở chỗ, *Containers* có nhiệm vụ thao tác với Store. Tức là nó có thể thay đổi state rồi sau đó truyền state xuống Component để render ra View.

Actions

Với tính năng thay đổi theme thì chúng ta có duy nhất một action là thay theme. Trong *changeThemeAction.js*, thêm action này:

```
export const saveTheme = color => ({
  type: "CHANGE_THEME",
  payload: {
    color
  }
});
```

Như bạn thấy, action gồm *type* và *payload*. Với tính năng thay theme thì payload chỉ đơn giản là mã màu thôi.

Reducers

Trong *themeReducer.js*, chúng ta thêm đoạn code dưới đây:

```

let initState = {
  color: "#FFFFFF"
};

export default function themeReducer(state = initState, action) {
  switch (action.type) {
    case 'CHANGE_THEME':
      console.log('themeReducer: ' + JSON.stringify(state))
      return Object.assign({}, state, {
        color: action.payload.color
      });
    default:
      return initState;
  }
}

```

Trong đoạn code trên, chúng ta đã khởi tạo một global state để lưu mã màu của theme. Sau khi nhận action, reducers sẽ không trực tiếp thay đổi state mà nó nhận được, mà tạo ra các bản copy và thay đổi trên đó. Đó là lý do mình phải sử dụng *Object.assign* thay vì return luôn state hoặc dùng phép gán "=".

Để reducer có thể kết nối với Store thì ta cần phải combine tất cả các reducer lại. Trong file *store/reducers/index.js*, bạn thêm nội dung như sau:

```

import { combineReducers } from 'redux';
import color from './themeReducer';

export default combineReducers({
  color
});

```

Với Reducer đã xong. Giờ chúng ta quay trở lại Footer component (*components/layout/Footer.js*). Hẳn bạn còn nhớ hàm:

```

submitThemeColor(color) {
  // lưu giá trị mã màu theme vào Store - redux
  // Xử lý sau
};

```

Hàm này sẽ được gọi mỗi khi người dùng click vào các nút màu trong Footer. Theo như luồng hoạt động trong Redux mà chúng ta áp dụng cho ứng dụng Todo, khi người dùng click vào các nút màu để chọn theme thì action: *'CHANGE_THEME'* được sinh ra và dispatch đến reducer mà chúng ta vừa định nghĩa ở trên.

Vậy làm thế nào để hàm *submitThemeColor* có thể làm được việc đó? Đây là lúc chúng ta xử lý điều này trong Container.

Containers

Containers là những component giao tiếp với Redux thông qua `connect()` của `react-redux`.

`connect()` có thể nhận tối đa 4 tham số, trong đó có 2 tham số mà chúng ta sẽ sử dụng trong ứng dụng Todo:

- **mapDispatchToProps:** Đại loại là nó sẽ map việc dispatch action đến reducer thành props quen thuộc mà ta vẫn hay thường dùng React.
- **mapStateToProps:** Có nhiệm vụ map giá trị state với props để bạn sử dụng trong component.

Do đó, ta có nội dung của file `store/comainers/Footer.js` như sau:

```
import { connect } from 'react-redux';
import { saveTheme } from '../actions/changeThemeAction';
import Footer from '../components/layout/Footer';

const mapDispatchToProps = dispatch => ({
  dispatch,
  saveColorTheme: color => dispatch(saveTheme(color)),
});

function mapStateToProps(state){
  return {
    themeColor: state.color
  };
};

export default connect(
  mapStateToProps,
  mapDispatchToProps
)(Footer);
```

Sau khi map xong props thì quay trở lại `components/layout/Footer.js` để cập nhật code hàm submit:

```
submitThemeColor(color) {
  if (color) {
    console.log('handleChangeTheme')
    this.props.saveColorTheme(color);
  }
};
```

Bước cuối cùng liên quan tới Redux đó là kết nối Reducer tới Store.

Mở `src/index.js`, cập nhật lại code như sau:

```
import React from "react";
import ReactDOM from "react-dom";
import "./App.css";

import TodoApp from "../components/TodoApp";

import { createStore } from 'redux';
import { Provider } from 'react-redux';
import rootReducer from './store/reducers';

const store = createStore(rootReducer);

ReactDOM.render(
  <Provider store={store}>
    <TodoApp />
  </Provider>, document.getElementById('root'));

```

Trong *TodoApp.js* thì đổi phần import Footer component sang container.

```
import React, { useState, useEffect } from "react";
import Todos from "../Todos";
import Header from "../components/layout/Header";
import Footer from "../store/containers/Footer";
import AddTodo from "../AddTodo"

```

Về cơ bản, đến đây là bạn đã hoàn thành việc lưu mã màu của theme vào global state rồi đấy.

Tuy nhiên, bạn không thấy ứng dụng thay đổi màu đúng không?

Bởi vì chúng ta chưa có xử lý việc thay đổi màu. Sau khi theme được lưu vào global state, việc tiếp theo là lấy giá trị trong state ra và đổi màu thôi.

Trong React, mỗi khi State thay đổi, nó sẽ trigger tới những hàm để bạn sử dụng nếu cần. Trong trường hợp này, chúng ta sẽ sử dụng hàm *componentWillReceiveProps*

Thêm đoạn code thay đổi màu theme trong *components/layout/Footer.js*

```
componentWillReceiveProps(nextProps){
  console.log('UNSAFE_componentWillReceiveProps: ' +JSON.stringify(nextProps))
  document
    .documentElement
    .style
    .setProperty("--main-color", nextProps.themeColor.color);
}
```

Vậy là xong rồi đấy, lưu tất cả lại và tận hưởng thành quả trên trình duyệt nhé. Hoặc xem online tại đây: <https://codesandbox.io/s/sparkling-grass-dq11n>

Tổng kết

Phần này chúng ta đã tìm hiểu thư viện Redux, cách ứng dụng Redux để quản lý State của ứng dụng. Việc sử dụng thành thạo Redux là kỹ năng cần thiết của React developer, vì các ứng dụng React sẽ cần tới nó rất nhiều.

Các bạn có thể tham khảo mã nguồn của phần này tại đây:

<https://github.com/vntalking/ebook-reactjs-that-don-gian/tree/master/phan7/todo-app>

Nếu bạn có bất kỳ thắc mắc hoặc chỗ nào chưa hiểu, đừng ngại liên hệ với mình qua:

support@vntalking.com.

DEPLOY ỨNG DỤNG REACT

Về cơ bản, ứng dụng do React tạo ra là các website tĩnh, tức là chỉ gồm có HTML, CSS, JS. Sau khi xây dựng ứng dụng xong, bạn có thể build dự án và triển khai lên môi trường Internet.

Mình xin giới thiệu 2 cách để triển khai ứng dụng React:












Cách 1: Build dự án, sau đó triển khai lên bất kỳ static server nào.

Tại màn hình terminal của dự án, gõ lệnh: `npm run build`

Hình 5.1: Lệnh build dự án

Khi lệnh build chạy thành công, nó sẽ tạo ra thư mục build trong dự án của bạn.

Hình dưới đây là nội dung thư mục build được tạo ra sau khi thực hiện lệnh build.

Name	Size	Modified
 asset-manifest.json	1,0 kB	16:46
 favicon.ico	3,1 kB	16:46
 index.html	2,3 kB	16:46
 logo192.png	5,3 kB	16:46
 logo512.png	9,7 kB	16:46
 manifest.json	492 bytes	16:46
 precache-manifest.c8e7253d8f1da22fc78c4f924a512ad0.js	657 bytes	16:46
 robots.txt	67 bytes	16:46
 service-worker.js	1,2 kB	16:46
 static	2 items	16:46
 style.css	0 bytes	16:46

Hình 8.1: Nội dung của thư mục build của ứng dụng Todo



Bạn không thể chạy ứng dụng trực tiếp bằng cách mở tệp index.html bằng trình duyệt được. Bởi vì React sử dụng client-side routing nên nó không chạy với file:// URL

Công việc tiếp theo là bạn copy toàn bộ các files trong thư mục build và đẩy lên bất kỳ static server nào.

Static server có thể bạn tự xây dựng bằng cách sử dụng Nginx, hoặc Apache... Hoặc sử dụng dịch vụ static server như: Gatsby, AWS Amplify, Heroku...

Cách 2: Triển khai lên static server trực tiếp từ mã nguồn để trên github

Cách thứ 2 này cũng gần tương tự với cách thứ nhất, chỉ khác là bạn sẽ triển khai dự án trực tiếp từ mã nguồn trên Github. Ưu điểm nhất của phương pháp này là dự án có thể triển khai nhanh chóng và miễn phí.

Trong khuôn khổ cuốn sách này, mình sẽ hướng dẫn các bạn cách deploy thứ 2. Cụ thể dịch vụ static server là Github Pages

Deploy ứng dụng React lên Github Pages

Github Pages cho phép bạn deploy static website (trang web tĩnh) lên đó với source code từ các Github repository.

Đầu tiên, mã nguồn đầy đủ ứng dụng Todo được mình đặt tại đây:

<https://github.com/vntalking/ebook-reactjs-that-don-gian/tree/master/phan7/todo-app>

Nếu các bạn thực hành đầy đủ từng bước theo cuốn sách thì cũng sẽ được mã nguồn giống như trên thôi.

Ngoài ra, mình có thể deploy trực tiếp từ repository trên cũng được. Nhưng để các bạn hiểu được quy trình một cách đầy đủ, chúng ta sẽ làm từ bước tạo một repository mới trên Github, rồi từ đó mới deploy sang Github Pages.

Bắt đầu thôi!

Bước 1: Tạo Github Repository trên Github

Tạo Github Repository trên Github. Phần này mình sẽ không hướng dẫn chi tiết, các bạn có thể tham khảo bài viết trên VNTALKING: [Cách tạo Repository trên Github bằng hình ảnh](#)

Giả sử, mình vừa tạo một repository mới có tên là: **reactjs-todo-app** cho ứng dụng Todo, có đường dẫn là: <https://github.com/vntalking/reactjs-todo-app.git>

Bạn ghi nhớ tên repository này lại, để tí nữa còn cấu hình cho URL của ứng dụng trên Github Pages.

Bước 2: Đồng bộ ứng dụng React ở local lên Github repository

Trước mắt, chúng ta sẽ clone repository rỗng mà chúng ta đã tạo ở trên về local đã nhé.

```
$ git clone git@github.com:vntalking/reactjs-todo-app.git
```



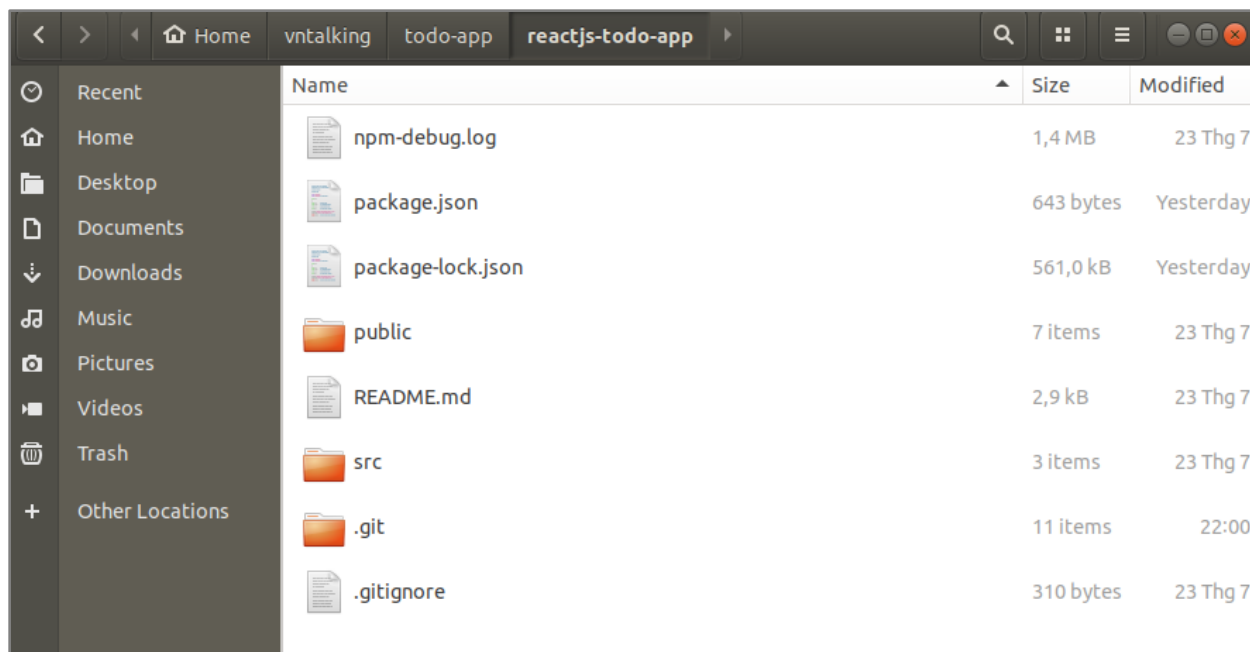
Ở trên mình dùng SSH để clone về nên không cần nhập mật khẩu đăng nhập Github. Nhưng làm theo cách này thì bạn phải tạo SSH public key và thêm vào tài khoản Github của bạn. Còn không thì bạn chọn kiểu clone qua HTTPS và nhập username và mật khẩu bình thường.

Bước tiếp theo, chúng ta copy toàn bộ mã nguồn ứng dụng Todo vào thư mục vừa được tạo khi clone repository rỗng từ Github về.

Các bạn khi copy nhớ bỏ qua hai thư mục (nếu có trong mã nguồn):

- `node_modules`
- `build`

Ngoài ra, nếu có cảnh báo ghi đè mấy file `.gitignore` hay `README.md` thì tùy bạn nhé. Chọn replace hay ignore đều được.



Hình 8.2: Copy toàn bộ mã nguồn ứng dụng todo vào thư mục sau khi clone

Tiếp theo thì đẩy toàn bộ mã nguồn lên repository bằng cách gõ lần lượt các câu lệnh sau:

```
# Thêm tất cả các file vào git
$ git add *

# Commit sự thay đổi
$ git commit -m "add todo app"

# Push các file lên Github
$ git push origin
```

```
sonduong@SonDuong-VNTALKING:~/vntalking/todo-app/reactjs-todo-app$ git push origin
Counting objects: 23, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (21/21), done.
Writing objects: 100% (23/23), 166.18 KiB | 662.00 KiB/s, done.
Total 23 (delta 0), reused 0 (delta 0)
To github.com:vntalking/reactjs-todo-app.git
 62b9753..eb19c41 master -> master
sonduong@SonDuong-VNTALKING:~/vntalking/todo-app/reactjs-todo-app$
```

Hình 8.3: Đẩy toàn bộ mã nguồn Todo app lên Github

Bạn có thể xem toàn bộ mã nguồn mình đã push lên Github:

<https://github.com/vntalking/reactjs-todo-app>

Bước 3: Cài đặt gh-pages

Sau khi đã đẩy thành công toàn bộ mã nguồn ứng dụng lên Github. Bước tiếp theo là cài đặt *gh-page* module để hỗ trợ deploy ứng dụng trực tiếp từ repository sang Github Pages.

Bạn có thể sử dụng NPM hoặc yarn để cài đặt *gh-page* đều được.

Vẫn ở cửa sổ terminal lúc trước, bạn gõ lệnh:

```
$ npm install --save gh-pages
```

Chờ đợi một chút để npm cài đặt.

Bước 4: Sửa file package.json

Bạn mở *package.json* để thay đổi một số thông tin liên quan khi deploy.

Như ứng dụng Todo, chúng ta có được *package.json* như dưới đây:

```
{
  "name": "my-todo-app",
  "version": "0.1.0",
  "private": true,
  "homepage": "https://vntalking.github.io/reactjs-todo-app/",
  "dependencies": {
    "axios": "^0.19.2",
    "gh-pages": "^3.1.0",
    "react": "^16.13.1",
    "react-dom": "^16.13.1",
    "react-scripts": "3.4.1"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject",
    "predeploy": "npm run build",
    "deploy": "gh-pages -d build"
  }
}
```

Trong đó:

- **homepage:** Là địa chỉ của ứng dụng sau khi được deploy, có dạng: `https://[your-user-name].github.io/[your-repo-name]/`
- **predeploy:** xác định câu lệnh để build ứng dụng trước khi deploy.
- **deploy:** Xác định câu lệnh để deploy ứng dụng.

Bước 5: Chạy câu lệnh deploy ứng dụng

```
$ npm run deploy
```

Như ứng dụng Todo thì kết quả như hình bên dưới đây.

```
Creating an optimized production build...
Compiled successfully.

File sizes after gzip:

 44.88 KB    build/static/js/2.f85904ea.chunk.js
 1.15 KB    build/static/js/main.20724521.chunk.js
 784 B (+4 B) build/static/js/runtime-main.69fa2210.js
 634 B      build/static/css/main.be516758.chunk.css

The project was built assuming it is hosted at /reactjs-todo-app/.
You can control this with the homepage field in your package.json.

The build folder is ready to be deployed.

Find out more about deployment here:

  bit.ly/CRA-deploy

> my-todo-app@0.1.0 deploy /home/sonduong/vntalking/todo-app/reactjs-todo-app
> gh-pages -d build

Published
```

Hình 5.5: Kết quả khi gõ lệnh deploy thành công

Sau khi deploy ứng dụng React thành công, bạn cần **chờ một vài phút**, rồi truy cập vào địa chỉ ứng dụng theo địa chỉ đã định nghĩa homepage trong package.json (ví dụ: <https://vntalking.github.io/reactjs-todo-app>) để kiểm tra thành quả.

BÀI TẬP

Với mục đích để bạn tự thực hành và củng cố kiến thức React đã học ở các phần trước. Mình tạo ra một bài tập nhỏ nhỏ để bạn tự thực hành. Tại đây, mình sẽ hướng dẫn các bạn thực hành bằng cách chia dự án thành các task nhỏ để các bạn thực hiện. Trong các task, mình sẽ có gợi ý và giải thích cụ thể nhiệm vụ cần thực hiện.

Bạn có sẵn sàng để thực hiện dự án này không?

Nếu đã sẵn sàng thì bắt đầu thôi!

Đây là ứng dụng mà chúng ta sẽ xây dựng.



Hình 9.1: Giao diện ứng dụng Meme Generator

Ứng dụng này làm được gì?

Ứng dụng Meme Generator này nhận những dòng text được người dùng nhập vào, sau đó tạo một hình ảnh ngẫu nhiên khi nhấn vào nút "Generate". Ảnh sẽ được lấy từ trên Internet thông qua HTTP request.

Cũng đơn giản phải không?

Giờ các bạn bắt tay vào xây dựng ứng dụng này thôi.

Project task

Dưới đây là những gợi ý và những task cần thực hiện khi làm ứng dụng.

Task 1: Phân tích UI

Dựa trên giao diện ứng dụng, bạn cần phân tích UI và chia thành các component, giống như đã làm với ứng dụng Todo App vậy. Ví dụ, bạn có thể chia thành các component như: MemeApp, Header, MemeGenerator...

Mỗi người có cách phân tích UI khác nhau, cái này tùy bạn tự chọn, miễn sao hợp lý là được.

Task 2: Tạo dự án với các files cần thiết, render gì đó lên màn hình

Ở phần này, bạn bắt đầu tạo mới dự án React, đặt tên là meme-generator, có thể sử dụng công cụ create-react-app như phần trước.

Nếu sử dụng công cụ create-react-app thì có thể xóa hết các files trong thư mục *src*, chỉ để lại file *index.js* với nội dung như sau:

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

const element = <h1>React Meme Application</h1>;

ReactDOM.render(element, document.getElementById("root"));
```

Sau đó, bạn cần tạo các tệp component như đã phân tích ở task 1. Ví dụ: *MemeApp.js*, *header.js*, *MemeGenerator.js*...

Công việc cuối cùng ở task này là thêm MemeApp component vào *index.js*

Task 3: Thêm các child component vào MemeApp và hiển thị gì đó

Sau khi bạn đã tạo các component xong ở task 2, giờ thì bạn include những component này vào component chính, đúng như kết quả đã phân tích UI ở task 1.

Ví dụ: Mình tạo 2 child component và đặt trong thư mục *src/components*: Header và MemeGenerator.

Trong đó:

- **Header:** Chỉ đơn giản hiển thị tiêu đề của ứng dụng
- **MemeGenerator:** Hiển thị các input field và hình ảnh sau khi tạo meme

Theo như phân tích UI thì cả component sẽ cùng xuất hiện trong component MemeApp. Kiểu như sau:

```
import React from "react";
import Header from "../Header";
import MemeGenerator from "../MemeGenerator";

const MemeApp = () => {
  return (
    <div>
      <Header />
      <MemeGenerator />
    </div>
  );
};

export default MemeApp;
```

Task 4: Thêm input fields và hiển thị một ảnh mặc định trong MemeGenerator

Ở phần này, bạn sẽ tiến hành thêm các input field để người dùng nhập text. Tất nhiên là bạn cũng cần xử lý logic cho các input field đó nữa. Ví dụ như sử dụng *onChange()* để gửi event lên MemeApp...

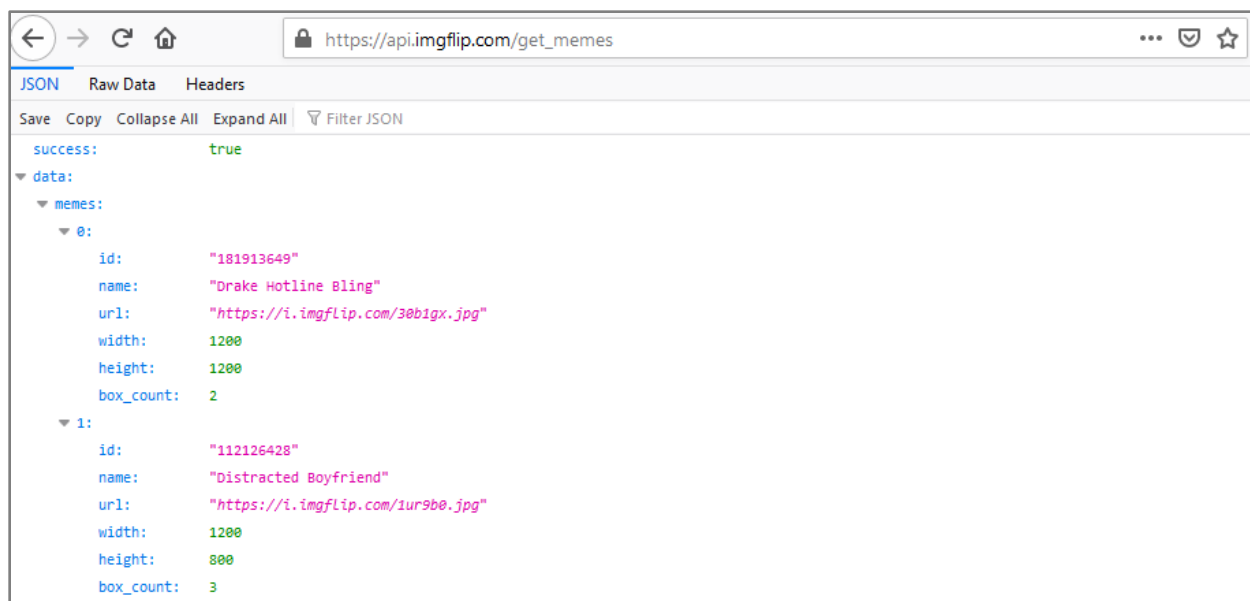
Tiếp theo, bạn cần hiển thị một ảnh nào đó lên màn hình MemeGenerator. Mình ví dụ một link ảnh: <https://i.imgflip.com/26am.jpg>. Lưu ý, chưa cần phải tạo HTTP request để hiển thị ảnh ở task này. Bạn chỉ cần thêm link ảnh vào thẻ *src* của ** tag là được.

Thêm CSS để định dạng ứng dụng giống với giao diện yêu cầu ban đầu.

Task 5: Tạo request để lấy danh sách ảnh

Đến bước này, bạn cần tạo một request tới một API nào đó để lấy một danh sách các URL ảnh.

Đây là API mà mình gợi ý sử dụng: https://api.imgflip.com/get_memes



Hình 9.2: Response của API

Để tạo request, bạn có thể sử dụng thư viện Axios hoặc native api của Javascript đều được.

Task 6: Generate ảnh ngẫu nhiên

Sau khi bạn đã danh sách các URL ảnh, giờ mỗi khi người dùng nhập text và nhấn nút "Generate" thì bạn sẽ cần hiển thị ngẫu nhiên một ảnh trong danh sách.

Đáp án bài tập tham khảo

Sau khi hoàn thành các task trên là bạn có thể hoàn thành ứng dụng Meme Generator này rồi.

Bước tiếp theo, bạn có thể publish ứng dụng lên Github Pages để chia sẻ với bạn bè. Cách deploy ứng dụng Meme cũng tương tự như đã làm với ứng dụng Todo nhé.

Cuối cùng là mã nguồn của ứng dụng Meme Generator để bạn tham khảo:

<https://github.com/vntalking/ebook-reactjs-that-don-gian/tree/master/phan9/meme-generator>

Trong quá trình thực hành nếu có thắc mắc, khó khăn cần giúp đỡ thì có thể lên Group để mình hỗ trợ: <https://www.facebook.com/groups/hoidaplaptrinh.vntalking>

GREAT!

Xin chúc mừng bạn đã hoàn thành nội dung cuốn sách!

Bây giờ, bạn hoàn toàn có thể tự tin để phát triển các ứng dụng React từ đầu, cho lúc triển khai ứng dụng.

Nhờ những kiến thức nền tảng, tư duy phát triển ứng dụng, bạn hoàn toàn có thể tự tìm hiểu thêm những kiến thức chuyên sâu, những tính năng nâng cao hơn về React.

Thay mặt các bạn trong đội ngũ VNTALKING, xin chúc bạn mọi điều tốt đẹp nhất trong hành trình trở thành một lập trình viên React chuyên nghiệp.

Hi vọng bạn sẽ thích cuốn sách này, và muốn tìm hiểu thêm về thế giới lập trình.

Trong quá trình biên soạn cuốn sách sẽ không tránh khỏi những thiếu sót. Mình rất mong nhận được phản hồi từ bạn, hãy gửi email tới support@vntalking.com.

Cuốn sách là một phần trong dự án học lập trình của VNTALKING. Mong bạn ủng hộ website hướng dẫn học lập trình tại: <https://vntalking.com>

Hẹn gặp lại ở những cuốn sách sau.

VNTALKING!

KẾT NỐI VỚI VNTALKING

VNTALKING đánh giá rất cao khả năng nỗ lực của bạn, chứng tỏ bởi việc bạn đã đọc hết cuốn sách và đọc đến tận trang sách này. Cảm ơn bạn rất nhiều ^_^

Đặc biệt, VNTALKING cũng rất vui khi được đồng hành cùng bạn trên con đường học để trở thành một lập trình viên chuyên nghiệp nói chung và React nói riêng.

Vì vậy, bất cứ khi nào bạn cần tư vấn, có thắc mắc hay khó khăn hãy "trút bầu tâm sự" với VNTALKING.

Liên hệ với VNTALKING bằng bất kỳ hình thức nào dưới đây.

- Website: <https://vntalking.com>
- Fanpage: <https://facebook.com/vntalking>
- Group: <https://www.facebook.com/groups/hoidaplaptrinh.vntalking>
- Email: support@vntalking.com

THÔNG TIN VỀ TÁC GIẢ

VNTALKING.COM là một website được thành lập từ 25/12/2016 và đang được vận hành bởi nhóm Dương Anh Sơn (một developer “kì cựu” – chuẩn bị về quê chăn lợn).

Bọn mình luôn hướng tới một trải nghiệm miễn phí mà hiệu quả. Bọn mình gồm những thành viên luôn muốn đem đến cho độc giả những kiến thức, kinh nghiệm thực tiễn, được cập nhật nhanh nhất. Đồng hành cùng VNTALKING để khám phá niềm đam mê lập trình trong bạn.

Thông tin thêm về tác giả.



Tên đầy đủ là Dương Anh Sơn, gọi tắt là Sơn Dương ^_^ . Tốt nghiệp ĐH Bách Khoa Hà Nội. Mình bắt đầu nghiệp coder khi ra trường không xin được việc đúng chuyên ngành. Mình tin rằng chỉ có chia sẻ kiến thức mới là cách học tập nhanh nhất.

CUỐN SÁCH KHÁC CỦA VNTALKING

Đến thời điểm hiện tại, VNTALKING đã hoàn thành 2 dự án sách học lập trình. Sách học React này là cuốn sách thứ 2 mà VNTALKING thực hiện.

Nếu bạn muốn tìm hiểu nhiều hơn về back-end, cụ thể là Node.JS, có thể tham khảo cuốn sách:



Lập trình Node.JS thật đơn giản

(<https://vntalking.com/sach-hoc-lap-trinh-node-js-that-don-gian.html>)

Hì vọng trong thời gian tới, VNTALKING sẽ tiếp tục nhận được sự ủng hộ của độc giả. Thành công của bạn chính là động lực để VNTALKING cho ra nhiều cuốn sách với chất lượng tốt hơn nữa, đáp ứng nhu cầu học lập trình của mọi người.