# EASY REACT JS FOR BEGINNER DEVELOPERS

*A STEP-BY-STEP VISUAL GUIDE TO LEARN REACT JS AND BUILDING YOUR OWN REACT APPLICATIONS FROM SCRATCH*

***2ND EDITION***

*IBAS MAJID*

**Get the Latest Web Development Tips and Tricks at**

**ibaslogic.com**

# Why You Should Read This Book

It is no doubt that React JS is one of the highly in-demand web skills at the moment. This makes it a necessity for you as a developer to add it on your resume' if you are aiming to create modern real-world web applications.

In this book, we have you covered.

You will not only learn the React fundamentals, but you will also be able to create modern React applications and deploy them on the internet for free.
This will also place you in a position to start working on web application that is built on React. A very good example is the *Gatsby site* which is a static site generator and a platform most developers are now shifting to. Also, if you are the WordPress person, the block editor often called *Gutenberg* is built on React. So you don't have choice than to learn this library if you aim to becoming a better WordPress theme developer.
More so, this book covers some other important topics that will make creating and deploying a React app on the web a breeze.

Some of these include the NPM version control with Git, selecting and installing the right text editor for your project and many more.

In addition to the core React knowledge, this edition covers the **React Hooks**. You'll learn what they are and how you can apply them in your project.

So if you like detailed and visual writing guides, plenty of tasks to be executed, then you'll love this React book.

Go ahead and read carefully, you'll be happy reading it.

I guarantee!

# Table of Contents

**Why You Should Read This Book** ........ 2
Table of Contents ........ 4
Course Introduction ........ 7
Who should read this book? ........ 8
Course Prerequisite ........ 9
What we will cover and build in this course ........ 10
Getting the most from this course ........ 10
What is React? ........ 12
Why should I learn this JavaScript library? ........ 13
Thinking in React Component ........ 16
The Document Object Model (DOM) ........ 19
The Virtual DOM ........ 21
Chapter 1 Setting up Working Environment ........ 23
Writing React directly in HTML ........ 23
What is JSX? ........ 25
Using the Create-React-App CLI ........ 31
A quick look inside the App (my-todo-app) folder ........ 34
Setting Up Text Editor and Installing the React DevTools ........ 37
Chapter 2 Writing the Todos App (First Project) ........ 58
React Component ........ 60
Working with Data (Props and State) ........ 62
Creating the Component Files ........ 65
Adding state data ........ 70

Table of Contents

Converting Class-Based Component to Functional Component .......................... 83
Creating the Header Component and Adding Styles to our App ........................ 90
Controlled Component .......................... 100
Raising and Handling Events .......................... 104
Updating the state using the setState() method .......................... 109
Using Destructuring .......................... 113
Deleting Items from the todos .......................... 115
Adding a text input field and a submit button .......................... 121
Updating the todos list .......................... 132
Chapter 3 Deploy React App to Gh-pages .......................... 141
Initialize the Project folder as a Git Repository .......................... 142
Deploy to local repository .......................... 145
Deploy to remote repository .......................... 148
Deploy Todos app to Gh-pages .......................... 150
Chapter 4 React HTTP Request and Lifecycle Methods .......................... 154
The Lifecycle methods .......................... 158
Making a GET request .......................... 160
Making a POST request .......................... 165
Making a Delete Request .......................... 168
Chapter 5 Project II (Building a Simple Meme Generator App) .......................... 181
Project Tasks .......................... 183
Adding Styles to the Meme Generator .......................... 196
Chapter 6 Getting Started with React Hooks .......................... 208
What are React Hooks? .......................... 209
Using the React Hooks useState .......................... 212
Using the React Hooks useEffect .......................... 221
Running an Effect only on Updates .......................... 233

Table of Contents

Chapter 7 Meme Generator with React Hooks ........ 243

CONNECT WITH IBAS ........ 258

ABOUT THE AUTHOR ........ 259

OTHER BOOK BY IBAS ........ 260

ONE LAST THING ........ 261

# Course Introduction

Hello and welcome to the second edition of my React JS course! I am pretty sure you are familiar with the basic web languages and styles like the HTML, CSS and JavaScript.

These languages are required to build any beautiful and dynamic website.

But have you taught about building modern web applications?

In this era, modern websites are embracing the Single Page Applications (SPA) design approach whereby a user's interaction immediately reflect in the user interface (UI) and does not require a page reload. A very good example is the Gatsby site.

In case you don't know what Gatsby site is.

It is a static site generator that is built on React. It is a platform most developers are now shifting to because it allows the integration of a Content Management System with a static site.

This way, you'll have a nice admin interface to manage your static site content.

***What makes this interesting?***

CMSs like WordPress websites as we know became popular because users can easily update website content rather than modifying the raw HTML.

But these websites are not as fast as static sites.

Now, with a static site generator like Gatsby, you can easily source content from WordPress or other CMSs, create and update content from their dashboard, and still get the benefits of a super-fast static site.

To build this type of website or any high-performance React application, a thorough knowledge of React is required.

Before you proceed,

## *Who should read this book?*

This book is aimed at beginner developer who is interested in developing modern React applications.

Or anyone who wants to customize the block editor of WordPress.

You should also read this book if you want to build a Gatsby website or any other static site generator built on React.

## *Course Prerequisite*

Before you go ahead with this course, you MUST have –

- Basic understanding of HTML and CSS.
- JavaScript fundamentals (object, arrays, conditionals etc).
- Familiarity with JavaScript ES6 features (class syntax, arrow functions, object destructuring etc).

Well, if you are still finding it tough with JavaScript, be rest assured that React will make you a better JavaScript developer.

So, read and code along with me, I will be explaining every task as we write our React application.

## *What we will cover and build in this course*

In this course, you will get to learn the React fundamentals and all that is needed to build a modern React application.

We will build two modern apps (*Todos* and *Simple Meme Generator*) from scratch and also deploy them on the web for **FREE**.

Visit the following links and check them out –

1. Todos application (https://ibaslogic.github.io/todoapp/)

2. Meme Generator (https://ibaslogic.github.io/memegenerator/)

Furthermore, in this edition, we have included the **React Hooks**. You will learn what these Hooks are and how you can apply them in your React project.

We will be converting the logic of our Todos and Meme Generator applications to using the React Hooks.

## *Getting the most from this course*

This book follows a pattern where a section is built on the previous.

So you wouldn't want to skip any.

Also, in this book are several tasks to be completed by you. Make sure you follow the clear instructions on how to execute them.

Lastly, you will find every bit of code used in this book in my GitHub repository.

The *Todos* application source code (Chapter 1- 3) – https://github.com/Ibaslogic/todoapp

*Todos* app (React HTTP Request and Lifecycle Methods) Chapter 4 – https://github.com/Ibaslogic/http-request-lifecycle-methods

Simple *Meme Generator* code (Chapter 5) –

https://github.com/Ibaslogic/memegenerator

*Todos* app with React Hooks (Chapter 6) – https://github.com/Ibaslogic/todos-react-hooks

Meme Generator with React Hooks (Chapter 7) – https://github.com/Ibaslogic/React-Hooks/blob/master/src/components/MemeGenerator.js

If at any point, your code did not work, please check my source code and compare with yours.

Now that you are aware of what you'll learn and how to get the most out of this course.

Let's have fun!

## *What is React?*

React (sometimes called *React.js* or *ReactJS*) is a JavaScript library for building fast and interactive user interfaces (UI). It was originated at Facebook in 2011 and allow developers to create sizeable web applications or complex UIs from a small and isolated snippet of code.

In some quarters, React is often called a framework because of its behaviour and capabilities. But technically, it is a library. Unlike some other frameworks like *Angular* or *Vue*, you'll often need to use more libraries with React to form any solution.

So what exactly does React do?

React makes sure that the view (User Interface) is properly rendered. In other words, it focuses on outputting something to the user and makes sure it is in sync with the state.

Sounds strange?

Don’t worry, you will get to understand better when we start building React Apps.

So keep reading!

## *Why should I learn this JavaScript library?*

In this era of many libraries and frameworks, it is imperative asking yourself this question rather than jumping into learning what might not be useful afterwards no matter how good the library/framework sound.

Here, I will itemize and give a brief reason why learning React is a necessity if you are aiming to build modern-day applications.

Ok, let’s get started!

1. **The use of reusable, independent Components** – In React, we describe User Interface (UI) using component.

You can think of this component as a function that accepts input and returns some output. And as we can reuse function in another function, so also we can reuse and merge components to form a larger user interface. In this book, you'll see how React will make your app more organize through components.

2. **React makes front-end JavaScript much easier and faster by using Virtual DOM** – When working with vanilla JavaScript, you'll have to manually do every little task while interacting with the DOM. But this is not the case with React.

   Here, you'll simply change the state (you will learn about this later) of your UI and React will update the DOM to match that state. It uses what is called a **Virtual DOM**. This allows us to only update what is needed (like a section of your webpage) as opposed to the whole page.

   Let's take for instance if you publish a new blog post in your web app, React will only update that post component to display the new post rather than reload the whole page.

3. **Easy to work with teams** – If you are a team player, you'll love to learn this library. This is because each member of a team can be assigned a different component of a project.

4. **Maintained by Facebook** – Unlike some other libraries/ framework that comes and disappears into thin air which is a valid point for some developers to be afraid to learn new technologies. This is not the case with React.
   One of the reasons it has gained so much popularity so fast is its origin. And this has generated a lot of contributors and a great community. With this, you wouldn't have to bother about getting stuck while building projects.

5. **Hirable** – React is one of the highly in-demand web skills at the moment. This is so because most web development companies are now building an interactive web app with React.
   And with the advent of static site generator like Gatsby, and the introduction of WordPress block editor (the Gutenberg), you are sure to get a good job or work as a freelancer once you are comfortable with this library.

> So my friend, if you want to expand your job opportunities as a front-end developer, you should have React on your resume'.

Additionally, React has less API and easier to learn compared with other libraries or frameworks especially if you have good command of JavaScript.

Now that you are ready to start this awesome journey, the very first thing expected of you is to start ***thinking in React*!**

## *Thinking in React Component*

I mentioned earlier that in React, everything you see in the User Interface (UI) is put into a self-contained component.

What that means is that when building an application with React, you build a bunch of independent, isolated and reusable component.

This individual component can then be merged to form a complex user interface.

If you take a look at the image below, we have a simple *Todos* list application (one of the app we will create from scratch in this book).

It may look simple in the eye but trust me, you will get to understand the concept of React and how it works after building this app.

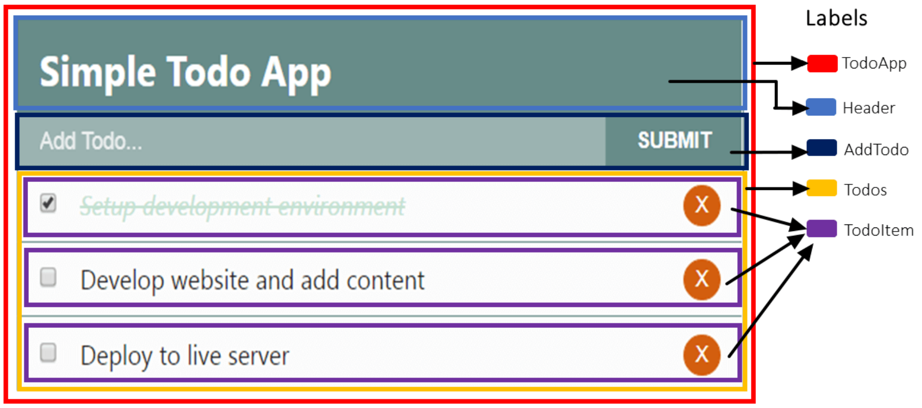


Fig: To-dos Application

To build this type of application or any complex app (even as complex as Twitter), the very first thing to do is to split and decompose the UI design into a smaller and isolated unit as outlined in the image. Where each of these units can be represented as a component which can be built in isolation and then later merge to form a complex UI.

Still on the image.

The parent component (also called the root component), label *TodoApp*, holds all the other components (known as child components). The *Header* component renders the header contents, the *AddTodo* component accepts user's inputs, the *Todos* renders the todos list and finally, the *TodoItem* component takes care of each of the todos items.

With this breakdown, we will be creating five different components in isolation.

By splitting your application into smaller sections ensure that it is well organized and of course easy to follow by other developers.

Once you have this instinct, you are *thinking in React!*

That's just it!

Let's move on.

## *The Document Object Model (DOM)*

Understanding how the DOM works will help you to quickly grasp the concept behind the *Virtual DOM* that React provides for us.

Though as a JavaScript developer, you are sure to have interacted with the DOM while building an interactive website. But you may have been able to avoid understanding how it works.

So let's reiterate.

### *What is DOM?*

The DOM (Document Object Model) is nothing but an interface that allows JavaScript or other scripts to read and manipulate the content of a document (in this case, an HTML document).

Whenever an HTML document is loaded in the browser as a web page, a corresponding Document Object Model (DOM) is created for that page. This model is simply an object-based representation of the HTML document.

This way, JavaScript can connect and dynamically manipulate the DOM because it can read and understand its object-based format.

This makes it possible to add, modify, delete contents or perform an action like toggling navigation menu on web pages.

## *Is there any problem with the DOM (Why the Virtual DOM)?*

Some developers believed that manipulating the DOM is slow and inefficient when frequently updating multiple elements on a page. This mostly happens in Single Page Applications (SPA) where JavaScript frameworks update the DOM much more than they have to.

But in reality, it is not the DOM that is slow but what happens in the browser workflow after manipulations.

Every time the DOM changes, the browser would need to recalculate the CSS, run layout and repaint the web page. And as you can deduce, this is what causes the delay. To speed up things, we need to find a way to minimize the time it takes to repaint the screen. This is where the *Virtual DOM* comes in.

## *The Virtual DOM*

Instead of the browser going through the process of repainting the entire webpage after DOM manipulation, React makes use of Virtual DOM which uses a strategy that updates the DOM without having to redraw all the webpage elements.

It ensures that the real DOM receive only the necessary data to repaint the UI. As the name implies, it is just a virtual representation of the real DOM. This makes it lack the power to directly change what is on the screen.

Let's see how it works in React.

Whenever new elements are added to the UI, a virtual DOM is created.

But if the state of any of these elements changes, React would update the virtual DOM while still maintaining the previous version so that it can compare and figure out which virtual DOM objects have changed.

Once the changes have been figured out, React then update ONLY those objects on the real DOM thereby reduces the performance cost of re-rendering the webpage.

Unlike vanilla JavaScript, whenever you're building an app with React, you will no longer have to manually update the DOM or attach an event handler to the DOM element.

All you have to do is simply change the state of your UI and React will automatically update the DOM to equal that state. The reason this library is called *React*.

Whenever the state changes, React essentially "reacts" to state changes and update the DOM.

Do not worry if all of these seems strange, you will get to see them in practice later.

# Chapter 1
# Setting up Working Environment

Just like every other web technology, you will need to set up a development environment to start interacting with React. Though React recommends setting up an environment through the *create react app* CLI tool (coming to that), I will quickly walk you through how to start working with React by simply writing React code in HTML file.

This will quickly get you up and running and it does not require any installation.

So let's do it.

## *Writing React directly in HTML*

No matter the background you are coming from as a web developer, you can easily get started with React if you have ever worked with HTML, CSS and JavaScript (which I presume you have).

This method of interacting with React is the simplest way and also okay for testing purposes.

Let's see how it is done.

We will start by creating an *`index.html`* file. Inside of it, we will load in three scripts in the head element pointing to their respective CDN– the *React*, *ReactDOM* and *Babel*. Then, we will create a *`div`* element and give it an *`id`* of *`root`*. This is where our application will live. Lastly, we will create a `script` element where we will write our React code.

So your `index.html` file should look like this:

```
 <!DOCTYPE html>
<html lang="en">
  <head>
    <title>React Tutorial</title>
    <script
src="https://unpkg.com/react@16/umd/react.development.js"></s
cript>
    <script src="https://unpkg.com/react-dom@16/umd/react-
dom.development.js"></script>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/babel-
standalone/6.26.0/babel.js"></script>
  </head>

  <body>
```

```
    <div id="root"></div>

    <script type="text/babel">
      const element = <h1>Hello from React</h1>;
      console.log(element);
    </script>
  </body>
</html>
```

The area of focus in the code above is the `script` element. The *type* attribute in the opening tag is compulsory for using Babel (will explain this in a moment).

Inside of the `script` tag, we have what looks like HTML element.

```
const element = <h1>Hello from React</h1>;
```

And you might be wondering why we are writing HTML inside of JavaScript.

Well, that line is **not** HTML but JSX.

## *What is JSX?*

Writing JavaScript/ React code to describe what the user interface (UI) will look like is not as simple as you may think.

This makes the React author to create what looks like a JavaScript version of HTML. This is called *JSX* (JavaScript XML). It is an XML like syntax extension to JavaScript that makes it easier and more intuitive to describe the UI.

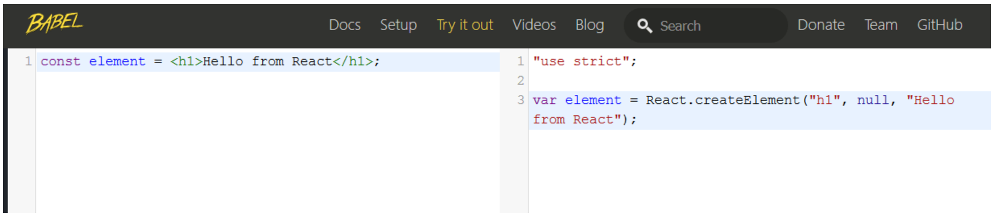
Under the hood, the JSX is being translated to regular JavaScript version of itself at runtime since the browser can't read it.

This is how it works –

The *JSX* code is passed to *Babel* (a JavaScript compiler) which will then convert it to plain JavaScript code that all browser can understand. This compiler also changes any JavaScript ES6 features into what the older browsers would recognize. For instance, it converts the *const* keyword to *var*.

Let's see a quick demo.

Head over to https://babeljs.io/repl and add the JSX code in the Babel editor.



```
const element = <h1>Hello from React</h1>;
```

```
"use strict";

var element = React.createElement("h1", null, "Hello from React");
```

The JSX code is converted to what is called *React.createElement* where its first argument, *h1* is the element type and the third argument is the text in-between the *h1* tags.

As seen in the image above, we can conclude that using JSX to describe what the UI looks like is much easier than writing plain React code.

*Please note: We loaded React library in the head of our HTML code even though we are not explicitly using React object from the library. But under the hood, React is using the React object as you can see on the right side of the Babel editor.*

Also, take note of the following about the JSX:

- You can use a valid JavaScript expression inside the JSX through curly braces, {}.

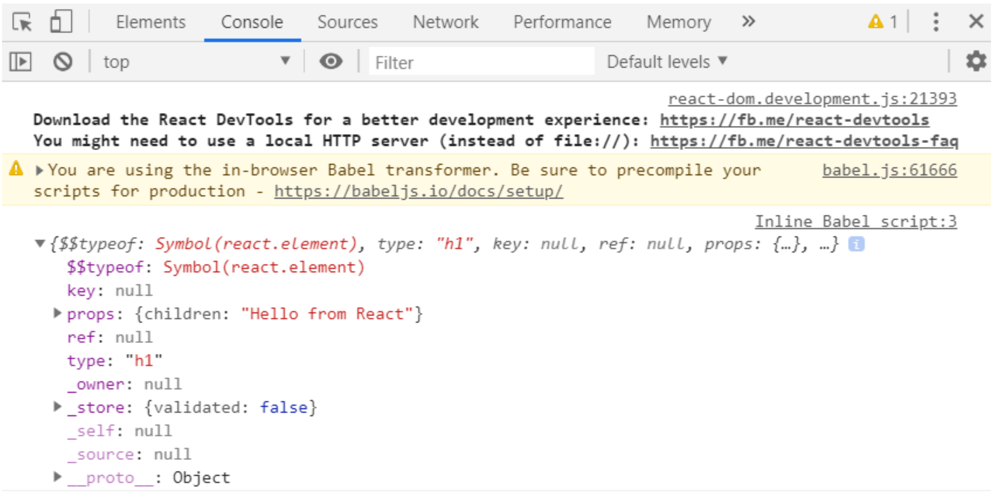
- In JSX, element *attributes, event handlers* are always in *camelCase*. The few exceptions are *aria-** and *data-** attributes, which are *lowercase*.

Now that you understand why we are using JSX and loading Babel, let's go back to our code.

Save the `index.html` file and open it with a web browser.

*Make sure you are connected to the internet as we have included different libraries through CDN.*

At the moment, nothing is displayed in the browser viewport. But if you open the *DevTools* and check the *Console* tab (since we `console.log` the element in our code), you will see an object representing the JSX.



```
                                                       react-dom.development.js:21393
Download the React DevTools for a better development experience: https://fb.me/react-devtools
You might need to use a local HTTP server (instead of file://): https://fb.me/react-devtools-faq
▶You are using the in-browser Babel transformer. Be sure to precompile your       babel.js:61666
scripts for production - https://babeljs.io/docs/setup/
                                                                   Inline Babel script:3
▼{$$typeof: Symbol(react.element), type: "h1", key: null, ref: null, props: {…}, …}
   $$typeof: Symbol(react.element)
   key: null
  ▶props: {children: "Hello from React"}
   ref: null
   type: "h1"
   _owner: null
  ▶_store: {validated: false}
   _self: null
   _source: null
  ▶__proto__: Object
```

This object is the output of the JSX expression. It is a React Element and also part of the Virtual DOM.

In React, this element can have a ***state***. And anytime the state of this object changes, a new React Element is created. React compares these elements and figures out what has changed. Then, it reaches out to the real DOM and update only the changed object.

Next, let's go ahead and render the React Element inside of the real DOM.

To do this, we will call the *`render()`* method that React exposes through the *`ReactDOM`* object. This method accepts two arguments.

The first argument defines what you want to render while the second defines where you want to render it.

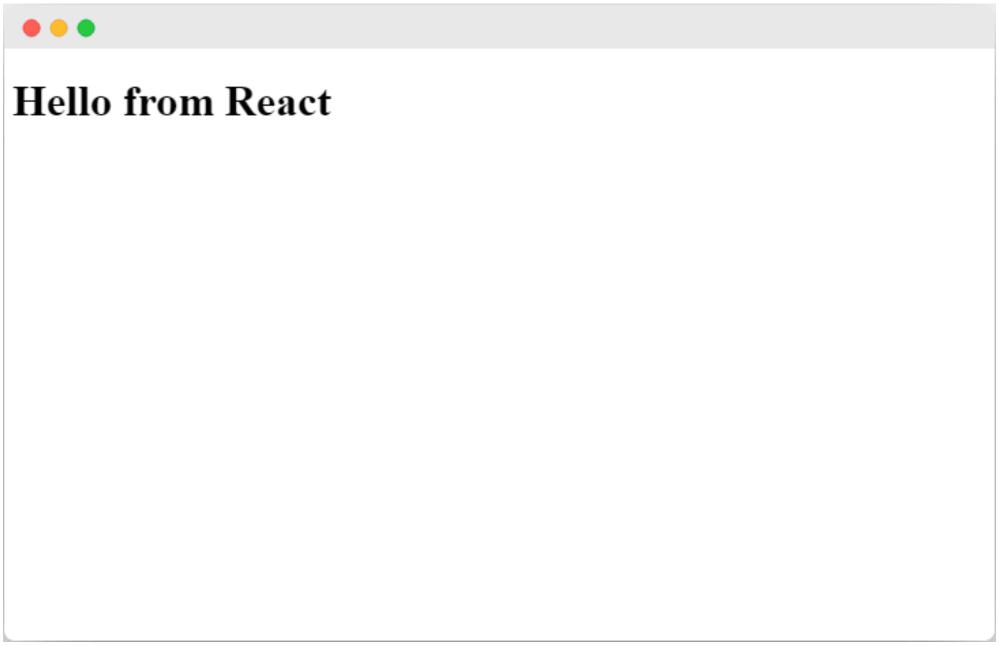
So let's update the `script` element so it looks like this:

```
<script type="text/babel">
  const element = <h1>Hello from React</h1>;
  ReactDOM.render(element, document.getElementById("root"));
</script>
```

The ***render*** method takes the first argument, *element*, and inserts it inside of the DOM element (*`div`*) using plain vanilla JavaScript to reference the *`div`* container.

*Note: We included ReactDOM library in the head of the HTML code so as to use `ReactDOM.render()` method.*

Once you update the HTML file to include the *`ReactDOM.render()`*, save and reload the webpage.



Hello from React

If you successfully render your content on the screen.

Congratulations! You have just created a React application.

## *Using the Create-React-App CLI*

Instead of manually loading scripts pointing to *React* and *Babel* CDN in the *head* element of your file, you can set up a React environment by installing the *create react app* CLI tool.

This tool will install React as well as other third-party libraries you will need. Some of the libraries include *Webpack* for bundling files, a lightweight development server, *Babel* for compiling JavaScript code.

To install and use this CLI, you need to have *Nodejs* installed on your computer. This will give you access to one its tools called *node package manager* (npm) that would allow you to install the CLI as well as any other dependencies or packages you would want to install later.

If you are not sure you have *Nodejs* installed on your computer, open your terminal and run ***node –v*** and ***npm –v*** to check for *Nodejs* and *npm* version respectively. Make sure the Node version is 8.10 or higher and the npm version is 5.2 or higher.

But if you don't have it installed, head over to https://nodejs.org/, download and install the latest stable version.

Then, run the following command in your terminal to create a React App called *my-todo-app*.

```
C:\Users\Your Name> npx create-react-app my-todo-app
```

You can give your app a different name but make sure you use lowercase all through (i.e *my-todo-app* instead of *MyTodoApp*).

Now you should have a new folder called *my-todo-app* in your directory.

Open the folder with your favourite code editor.

If you have VsCode installed on Windows Os., and you want to open your folder with a shortcut from the windows terminal.

You can navigate to the directory with this command:

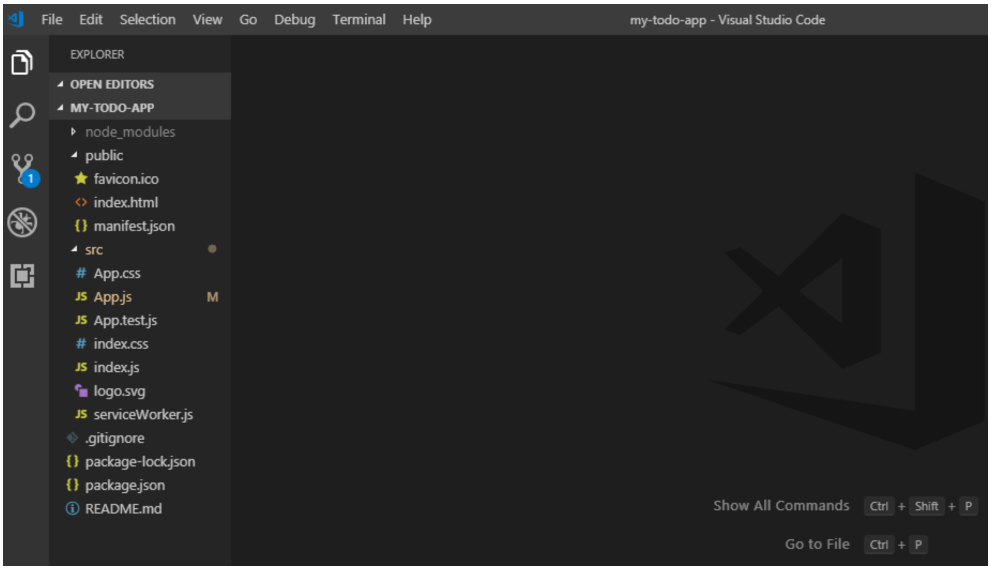
```
C:\Users\Your Name > cd my-todo-app
```

And then, open it using this command (shortcut):

```
C:\Users\Your Name\my-todo-app > code .
```

## *A quick look inside the App (my-todo-app) folder*

In this folder, we have the initial file structure for our project. We will take a look at some of the important folders and files.



Starting with the *node_modules*. This folder contains all the third-party libraries as well as React itself. It will also contain packages that you'll be installing through the *npm* later. You don't have to touch the folder.

The *public* folder contains the public asset of your application. The files inside of it can be accessed from the browser address bar. Also, it is where your static files reside. The `index.html` in the `public` folder is similar to the one we created earlier. It also has a `div` container element where your entire application will appear.

The `src` folder contains the working files. One of them is the `index.js` which will serve as the entry point to our application. Don't worry about all the `src` files, we will write everything from scratch. This is where you'll spend most of your time.

Lastly, the `package.json` is a manifest file containing information about your app. It has some dependencies of libraries that are currently installed and if you install other packages, they will be listed as well.

Enough said!

Let's see how the default application looks like by starting the development server. Head over to your terminal and run this command:

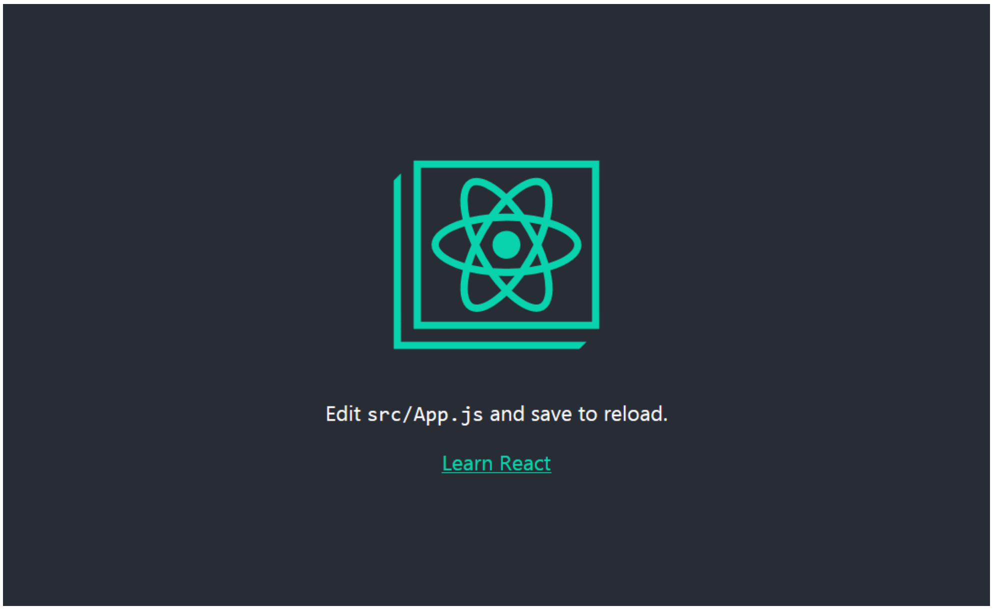
```
C:\Users\Your Name\my-todo-app > npm start
```

*Note: Make sure you are in the project directory before you run the command. In our case, my-todo-app. Also, if you are on VsCode, you can Toggle the terminal by pressing Ctrl + ` or Cmd + `.*

The *npm start* command will lunch the development server on *port 3000* and a new browser window displaying your application will appear automatically.

If nothing happens, visit this URL, *localhost:3000* in the browser address bar.

You should have something similar to the image below.


Edit `src/App.js` and save to reload.

Learn React

As seen in the image, you can edit what is displayed on the banner by modifying the `App.js` file located in the `src` folder.

Let's do that.

Open the `App.js` file and change ***Learn React*** to ***Learn React Now***. Then save the file and go back to your browser.

Notice that you do not have to reload the browser for the change to take effect. This is because *create-react-app* comes bundled with *hot reload*.

## *Setting Up Text Editor and Installing the React DevTools*

As a developer, finding a good text editor that works for your project is very critical.
There are many sophisticated text editors available online for download and purchase, but it can be a bit tough to decide which of them to use.

Well, below is a list of popular and exceptional editor you can choose from –

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

Though, there is no “best” text editor, only the one that makes it more comfortable and easier to work with a particular project.
But in this course, I will advise looking at the Visual studio code (VSC).

This is because it has an integrated terminal where you can perform a quick command-line task without leaving the editor. It also has some extensions pre-installed, unlike some editors where you’ll have to install all necessary extensions.

Finally, it is fast, lightweight, powerful and **free** to use.

### *Installing Visual Studio Code Editor*

Irrespective of the operating system you are using, visit https://code.visualstudio.com/Download.

Download the latest version appropriate for your operating system and the bit version you are running and then install.

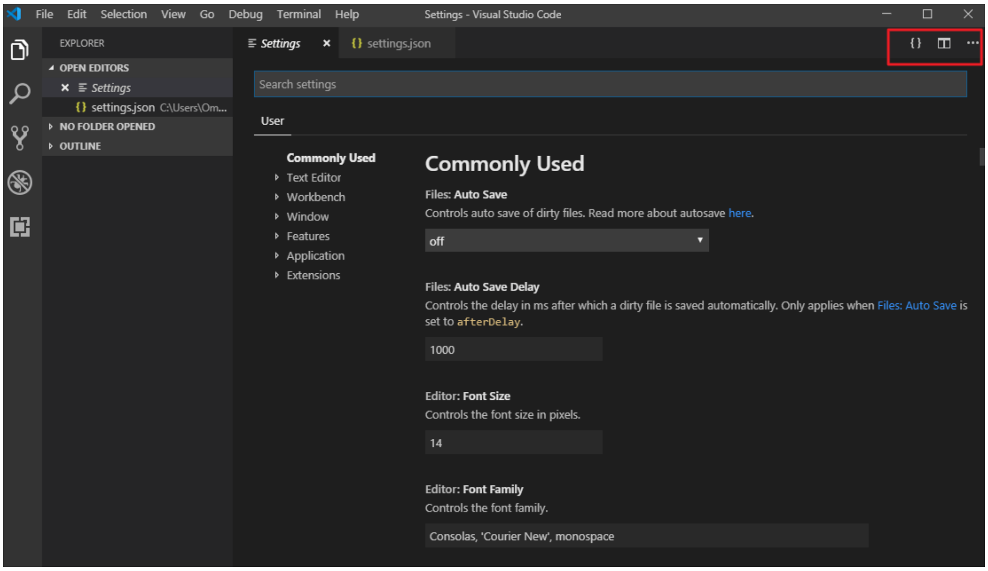
If you're not sure, get the one that is not 64-bit.

Once you install and launch, your text editor is ready to be used. The first thing you would want to do is to take a look at the *Settings* page. There, you can change the default settings like the font size and some other things.

Let’s take a look.

From the editor, go to:

```
File > Preferences > Settings
```



Commonly Used

Files: Auto Save
Controls auto save of dirty files. Read more about autosave here.
off

Files: Auto Save Delay
Controls the delay in ms after which a dirty file is saved automatically. Only applies when Files: Auto Save is set to `afterDelay`.
1000

Editor: Font Size
Controls the font size in pixels.
14

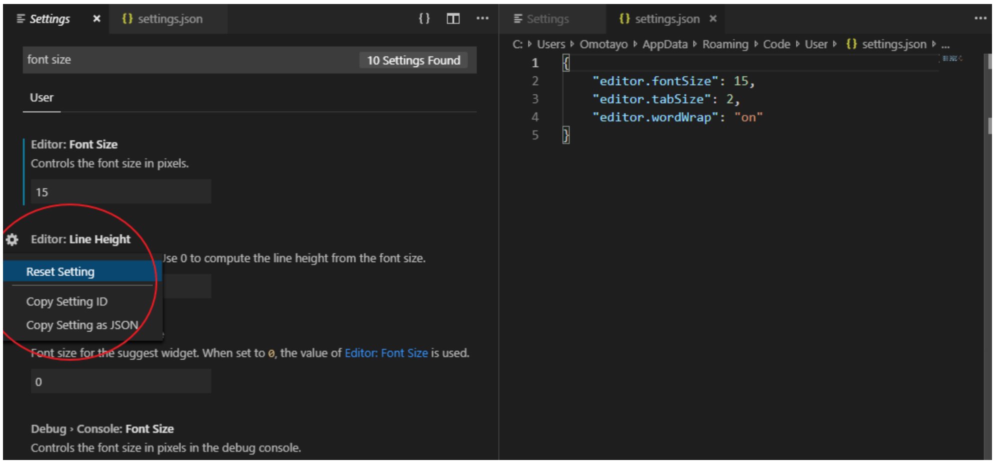
Editor: Font Family
Controls the font family.
Consolas, 'Courier New', monospace

You can scroll through or search for the settings that you want to modify and then change it.

Here, you also have options to see the page in the JSON format. Not only that, you can split the editor to the right and see the changes in real-time.

To do all these, hover on the three icons at the top right as highlighted in the image above and click on any of them.

You should have something like this if you have made any changes.



Editor: **Font Size**
Controls the font size in pixels.
15

Editor: **Line Height**
Use 0 to compute the line height from the font size.

Font size for the suggest widget. When set to 0, the value of Editor: Font Size is used.
0

Debug › Console: **Font Size**
Controls the font size in pixels in the debug console.

C: › Users › Omotayo › AppData › Roaming › Code › User › settings.json › ...

```
{
    "editor.fontSize": 15,
    "editor.tabSize": 2,
    "editor.wordWrap": "on"
}
```

On the right side, you can see the changes in JSON format. Once you make any changes in the UI (on the left side) it will be included in the *settings.json* (on the right side). Or better still, make the changes directly in the JSON file.

Go with the method you are most comfortable with.

Note that you will have a *blue left line* on every modified setting in the UI. And you can restore any setting by hovering on it and then click on the gear icon as seen in the image above.

## *Opening the Editor Terminal*

The terminal is one of my favourite features of this editor. Let's open it.

Go to:

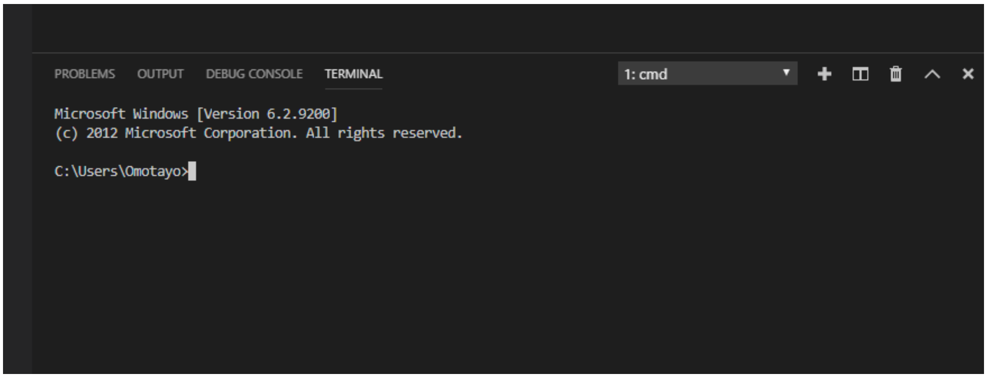
---

View > Terminal

---

Or use shortcut: Ctrl + ` or Cmd + `.

By default, if you are on Windows, you will have the terminal opened as *cmd* or *PowerShell* (if you are on Windows 10).



```
Microsoft Windows [Version 6.2.9200]
(c) 2012 Microsoft Corporation. All rights reserved.

C:\Users\Omotayo>
```

If you don't like this default Windows terminal, let's go ahead and switch it to another common Shell called *Git bash*.
Though the Git bash is also available for *Mac users*, you don't need it because you have a pretty good standard bash editor.

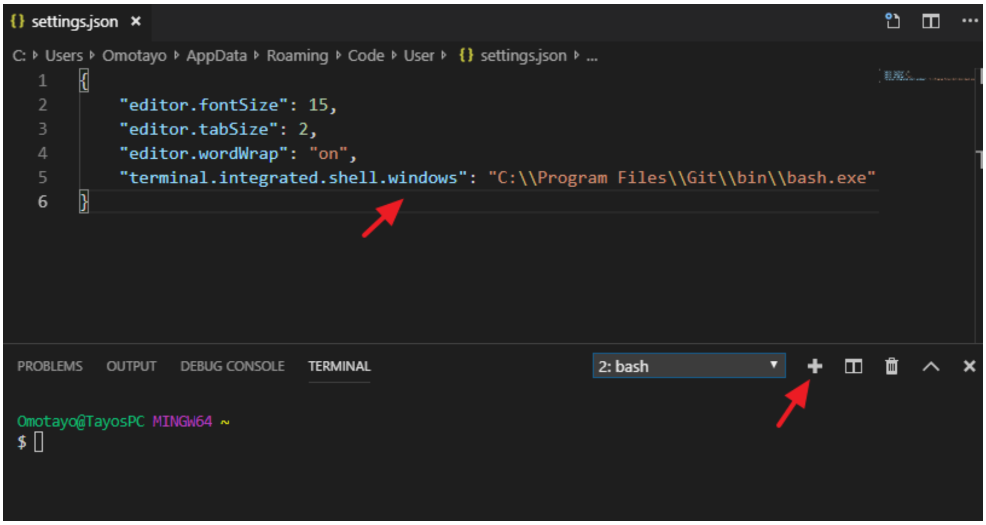
To use Git bash, head over to https://git-scm.com/, download and install Git.
Once you are done with the installation, go back to the VsCode editor and open the *Command Palette* through:

```
View > Command Palette..
```

Or use this shortcut: Ctrl + Shift + P.

In the command palette, search for *Terminal: Select Default Shell* and select it. This will display a list of all the terminals shells currently available on your computer. There, you choose the shell of your choice. In our case, you will choose *Git Bash C:\Program Files\Git\bin\bash.exe*.

To see the change in the editor terminal, you will need to add a new terminal by clicking on the *plus icon* on the top-right of the terminal panel.



{} settings.json
C: › Users › Omotayo › AppData › Roaming › Code › User › {} settings.json › ...
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
2: bash

```
{
    "editor.fontSize": 15,
    "editor.tabSize": 2,
    "editor.wordWrap": "on",
    "terminal.integrated.shell.windows": "C:\\Program Files\\Git\\bin\\bash.exe"
}
```

```
Omotayo@TayosPC MINGW64 ~
$ 
```

Notice in the image above that the terminal setting is added to the *settings.json* file.

At this point, you now have the Git bash as the default editor terminal.

Moving on.

## *Installing Extensions for Visual Code Editor*

Here, we will install some of the plugins that will make writing React application easy for us.
By default, this editor comes with *Emmet* and *IntelliSence* plugins which are very useful. Let's see what these plugins can do and then go ahead and install some other extensions.

1. ***Emmet*** – This extension is designed to speed up the creation of HTML and CSS. It uses abbreviations to define any HTML markup or styles you want to generate.
Let's see a few examples.
First, create a *`.html`* file (*`index.html`*) and type this in-between the body tags:

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

After that, hit the **Tab** key. Emmet then expands the snippet to generate the following code:

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

Another example –

```
section+div+p
```

The above snippet generates:

```
<section></section>
<div></div>
<p></p>
```

Let’s do a couple more.

```
div#main
```

The above generate this:

```
<div id="main"></div>
```

And lastly,

```
div.content.page
```

Generate this:

```
<div class="content page"></div>
```

In a CSS file, every time you enter unknown abbreviation, Emmet will try to find the closest snippet definition.
Try this out in your .css file.

---

**`df`** will be expanded as `display: flex;`

**`w100`** will be expanded as `width: 100%;`

**`ovh`** or even **`oh`** will be expanded as `overflow: hidden;`

---

There are lots of syntaxes when it comes to Emmet. You can check this cheat sheet – https://docs.emmet.io/cheat-sheet/ to learn more.

Apart from HTML and CSS, Emmet snippet expansions are also enabled by default in scss, sass, less, stylus, jsx, xml, xsl, haml, jade, slim files.

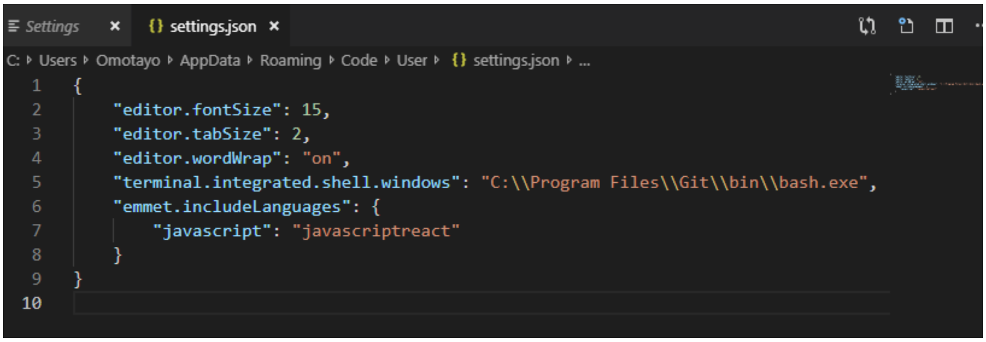
Out of this list, our focus is on the jsx.

In this course, we will be writing a lot of jsx code to describe our React user interface (UI). You can think of jsx as an extension of JavaScript. So in this regard, we can either name our jsx file as `.jsx` or `.js`. Though React recommends using the `.js` file extension and that is what we will use for our project.
But the Emmet abbreviation expansion is not available by default in `.js` file, let's see how to enable it.

Simply add this mapping to your settings:

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

So you now have:


Settings
{} settings.json
C: ▸ Users ▸ Omotayo ▸ AppData ▸ Roaming ▸ Code ▸ User ▸ {} settings.json ▸ ...

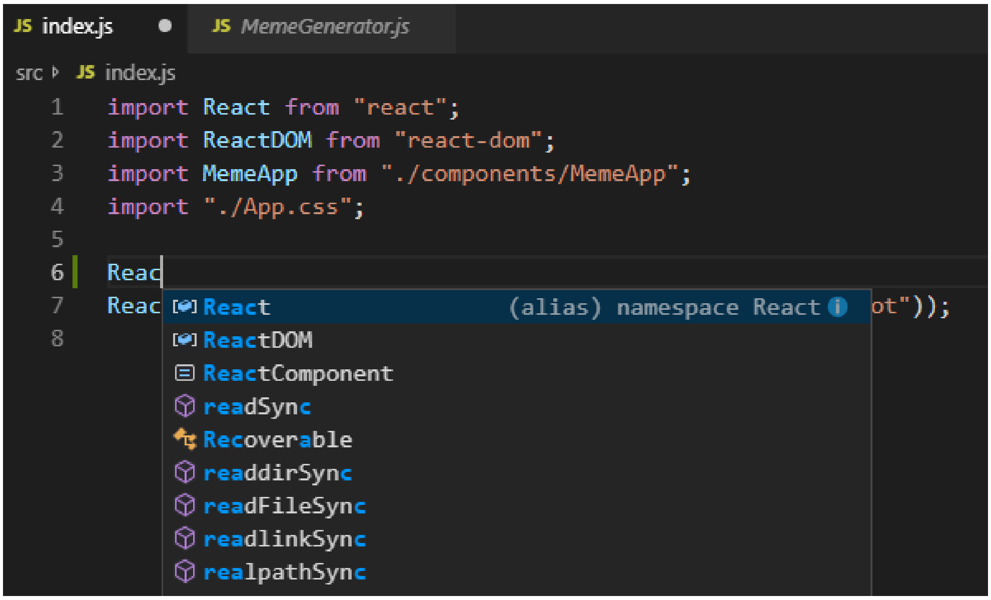
```
{
    "editor.fontSize": 15,
    "editor.tabSize": 2,
    "editor.wordWrap": "on",
    "terminal.integrated.shell.windows": "C:\\Program Files\\Git\\bin\\bash.exe",
    "emmet.includeLanguages": {
        "javascript": "javascriptreact"
    }
}

```

Please be mindful of the commas in the JSON file.

Ok good.

2. ***IntelliSense*** – Just like the Emmet, this extension also comes bundled with Visual Studio Code. It is a feature that analyzes what you are inputting and then provides suggestions on how to finish what you are writing. It is otherwise called *code completion, code hinting* or *content assist.*

See how it displays suggestions while typing.



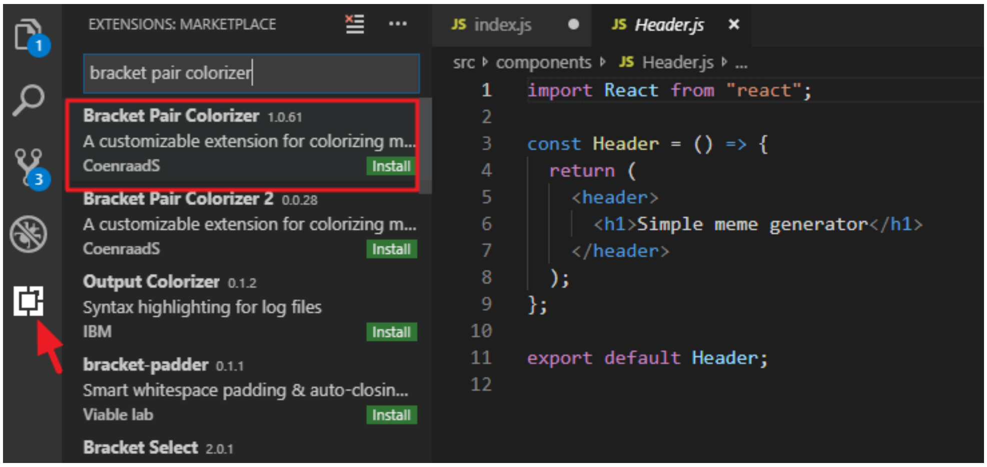
```
JS index.js  ●   JS MemeGenerator.js
src ▸ JS index.js
import React from "react";
import ReactDOM from "react-dom";
import MemeApp from "./components/MemeApp";
import "./App.css";

Reac
Reac                                                        ot"));
```

Note: Visual Studio Code provides IntelliSense for HTML, CSS, Sass, Less, TypeScript, JSON, and JavaScript web languages. For others, you'll need to install their extensions.

Now, we can move on to the extensions we will be installing.

3. ***Bracket Pair Colorizer*** – As the name implies, this extension allows matching brackets to be identified with colours.

To install it, go to the *Extensions* tab on the main navigation bar to your left side of the editor and search for *Bracket Pair Colorizer*. See the image below.


EXTENSIONS: MARKETPLACE
bracket pair colorizer
Bracket Pair Colorizer 1.0.61
A customizable extension for colorizing m...
CoenraadS
Install
Bracket Pair Colorizer 2 0.0.28
A customizable extension for colorizing m...
CoenraadS
Install
Output Colorizer 0.1.2
Syntax highlighting for log files
IBM
Install
bracket-padder 0.1.1
Smart whitespace padding & auto-closin...
Viable lab
Install
Bracket Select 2.0.1
index.js
Header.js
src › components › Header.js › ...

```
import React from "react";

const Header = () => {
  return (
    <header>
      <h1>Simple meme generator</h1>
    </header>
  );
};

export default Header;
```

Once you install it, you will be able to see with ease, how braces in your code editor matched. Even with multiple lines of code.

4. ***Prettier* – code formatter** – This is a very popular extension. It helps format your code automatically and ensure that your codebase has a consistent style.

This is how it works.

It takes in all your code, removes all formatting, and therefore re-formats the code according to its style guidelines.

Now go ahead and install this extension.

Once it has been installed, open the Settings UI and search for *Editor: Format On Save* and make sure its box is *checked*.

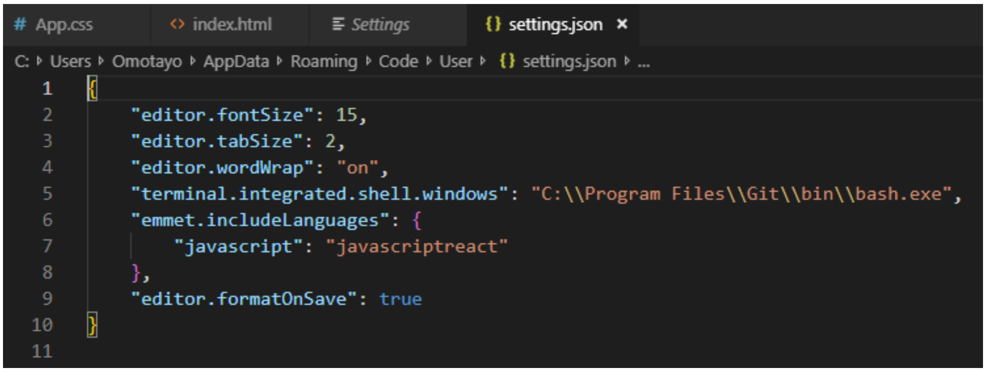
Or simply add the following to the Settings JSON and save it.

---

```
"editor.formatOnSave": true
```

---

So far, the file looks like this:


# App.css
<> index.html
Settings
{} settings.json ×
C: ▸ Users ▸ Omotayo ▸ AppData ▸ Roaming ▸ Code ▸ User ▸ {} settings.json ▸ ...

```
{
    "editor.fontSize": 15,
    "editor.tabSize": 2,
    "editor.wordWrap": "on",
    "terminal.integrated.shell.windows": "C:\\Program Files\\Git\\bin\\bash.exe",
    "emmet.includeLanguages": {
        "javascript": "javascriptreact"
    },
    "editor.formatOnSave": true
}
```

Now, if you have something like this in your JavaScript file:

```
const fname="Dennis";const lname='Roy';const person
={first:fname,last:lname };console.log(person)
```

It will be formatted to this anytime you save the file:

```
const fname = "Dennis";
const lname = "Roy";
const person = { first: fname, last: lname };
console.log(person);
```

As you can see, prettier takes care of the spacing, line wrappings and ensures consistent quotes.

Now you don’t need to worry about your code formatting anymore because prettier will handle it for you perfectly.

Prettier also works with many languages and styles like – TypeScript, Flow, JSX, JSON, CSS, SCSS, Less, HTML, Vue, Angular, GraphQL, Markdown, YAML.

5. ***Es7 React/Redux/GraphQL/React-Native snippets*** – With this plugin, you will be able to speed up development by generating React components (i.e block of code) using shortcuts. Do not worry about this for now. You will learn how it works later in the course.

For now, just go ahead and install it and move on.

***Installing the React Developer Tools (React DevTools)***

Debugging is no doubt one of the most useful skills a developer can acquire. Having this skill will make it very easy to properly navigate and spot errors in your code.
Here, we will quickly look at a tool – *called React Developer Tools* – that will allow us to inspect and debug our React apps components.
Also, we will be able to see the state of our app and how it behaves in real-time.

Let's go ahead and install it.

This tool is available as a browser extension for *Chrome* and *Firefox*.

So, head over to the extension page for your browser of choice and install it.

Chrome here: https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi.

Firefox here: https://addons.mozilla.org/en-US/firefox/addon/react-devtools/.

Once you install it, you are done. It doesn’t require any other setup.

Now, if you are using a different browser, the React DevTools is available as a standalone application.
You can install it from NPM by running this command in your terminal:

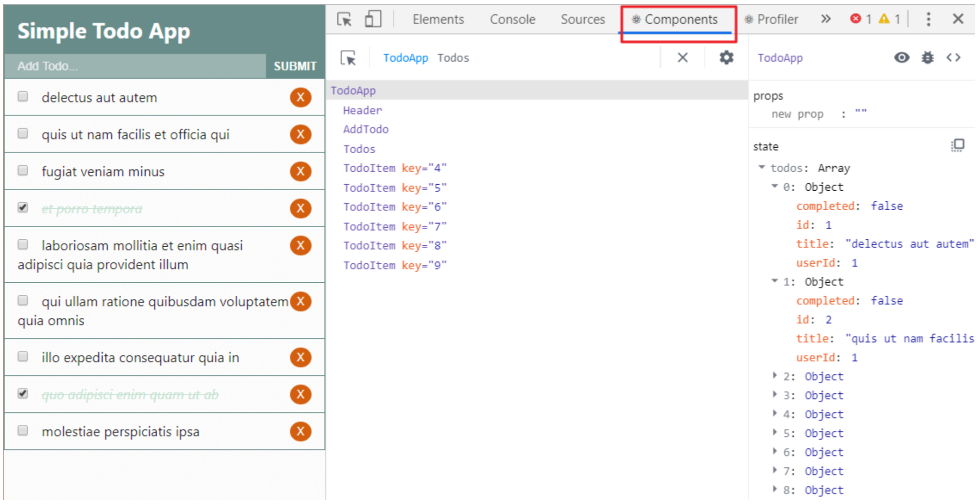
```
npm install -g react-devtools@^4
```

### *Opening the React DevTools*

If you have worked with the browser DevTools before, then you are good to go. Just open it.

If not, right-click anywhere on your web page viewport and select *Inspect* or *Inspect Element* depending on your browser.

Then, on the browser inspection window, you will need to select the *Components* tab to see the view of your application hierarchy.



- [ ] delectus aut autem
- [ ] quis ut nam facilis et officia qui
- [ ] fugiat veniam minus
- [x] ~~et porro tempora~~
- [ ] laboriosam mollitia et enim quasi adipisci quia provident illum
- [ ] qui ullam ratione quibusdam voluptatem quia omnis
- [ ] illo expedita consequatur quia in
- [x] ~~quo adipisci enim quam ut ab~~
- [ ] molestiae perspiciatis ipsa

```
TodoApp
  Header
  AddTodo
  Todos
  TodoItem key="4"
  TodoItem key="5"
  TodoItem key="6"
  TodoItem key="7"
  TodoItem key="8"
  TodoItem key="9"
```

```
props
  new prop  : ""

state
 ▾ todos: Array
   ▾ 0: Object
       completed: false
       id: 1
       title: "delectus aut autem"
       userId: 1
   ▾ 1: Object
       completed: false
       id: 2
       title: "quis ut nam facilis
       userId: 1
   ▸ 2: Object
   ▸ 3: Object
   ▸ 4: Object
   ▸ 5: Object
   ▸ 6: Object
   ▸ 7: Object
   ▸ 8: Object
```

Please note that you will not be able to see the *Components* tab if your webpage is not using React at the moment.

To see it, just start the React development server and go back to reopen the browser DevTools.

Now, we can move on to development.

And this is where the fun begins.

# Chapter 2
# Writing the Todos App (First Project)

At this point, we can start creating our React App. The files that describe what you are seeing in the frontend live in the *src* folder.

Since this book focuses on the beginners, we will write all the *src* files from scratch.

So let's start by deleting all the files in the *src* folder.

The frontend breaks immediately you do that. This is because React needs an *index.js* file present in the *src* folder. This file is the entry point.

Let's create this file.

In the *src* folder, create an *index.js* file and add the following code:

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

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

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

Once you save the file, you'll see a heading text displayed in the frontend.

Comparing this code to the one we write directly in the HTML file at the beginning. You'll see that we didn't do anything special except that we are importing `React` and `ReactDOM` instead of loading their respective CDN. Also, notice we are not loading *Babel* to compile the JSX to JavaScript. It comes bundled with the *create-react-app* CLI.

The `import` statement as used in the code is an ES6 feature that allows us to bring in objects (`React` and `ReactDOM`) from their respective modules (`react` and `react-dom`).

*Note: A module is just a file that usually contains a class or library of function. And create-react-app CLI have both files installed for us to use.*

If you are not familiar with ES6 modules, don't worry, just code along with me.

Back to our code,

At the moment, we are rendering the JSX element directly in the real DOM through the `ReactDOM.render`. This is not practicable.

Imagine having an app with hundreds of elements. You'll agree with me that it would be hard to maintain.

So, instead of rendering a simple element in the DOM, we will render a React component.

Before we proceed, let's get to know what this component is.

## *React Component*

A component in React is nothing but an independent and reusable bit of code. It is one of the reasons React became popular.

Earlier, I mentioned that the UI is described using the component and that you can think of components as a simple function that you can call with some input and they render some output.

Just like reusable functions, you can also reuse components, merge them and thereby creating a complex user interface.

In React, the component comes in two types, namely –

1. *Functional component* and
2. *Class component*.

As the name implies, the ***functional components*** are created by writing function. This component type optionally accepts *props* (think of this as the attributes in HTML element) as an argument and returns React element that would be rendered on the screen.

This type is the simplest form of React component because it is often associated with the presentational concept (i.e primarily concerned with how things look).

Before React version 16.8, this type of component is referred to as a ***stateless*** component because it cannot maintain a state (think of the ***state*** as the data you can store to a specific component) and lifecycle logic.

Don't worry about these terms (*State* and *Props*) for now. It will be clear in a moment.

On the other hand, the ***class components*** (also called ***stateful*** component) are created using the ES6 class syntax. They can also receive *props* just like the functional component. Plus, they provide additional features such as the ability to naturally have internal *state* and/or *lifecycle* method (more on this later) that can control what is rendered on the screen.

## *Working with Data (Props and State)*

Oftentimes, we have mentioned these two terms – *props* and *state*. And you may be wondering what they are.

When creating React app, you cannot do without having components receiving and/or passing data. It may be a child component receiving data from its parent or maybe the user directly input data to the component.

Understanding how to work with data is very crucial to building React component.

Starting with the *props*.

The *props* (which stands for properties) can be thought of as the attributes in HTML element.

For instance, the attributes – *type, checked* – in the input tag below are props.

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

They are the primary way to pass data and/or event handlers down the component tree – i.e data from the parent to its child component.

When this happens, the data that is received through props in the child component becomes read-only and immutable (i.e they cannot be changed by the child component).

This is because the data is owned by the parent component and can only be changed by the same parent component.

***The state***

I mentioned that component in React can receive data either from its parent or directly from the user's interaction as in the case of form input.

Since *props* allow a component to receive data from its parent, what happens if a user inputs data directly to the component?

That is why we have the *state*.

When a component receives data from the user's interaction (like updating the form input fields, toggling menu button etc.), the data is stored in the state and can only be updated by the component holding it. Making the state local to the specific component.

Please note –

The child component can receive the state of its parent as props.

But don't forget, only the component that owns it (in this case, the parent) can update it.

As a recap!

The state of a component is owned by that component and can be managed within. Props are passed into a component from its parent and are managed by the parent component.

Since the state of a component will often become the props of its child component, to change the immutable props, all you have to do is to change its internal state from its parent and React will propagate it to the child component.

Hope that is clear?

Ok, move on.

## *Creating the Component Files*

To start creating your project files, you need to have a good organizational structure. This will save you the stress as your application becomes more and more complex.

If you revisit the To-dos diagram at the beginning, we decomposed our application into a tree of isolated components. Where the parent component, *TodoApp* holds three children components (*Header*, *AddTodo* and *Todos*). Then, *Todos* holds another component called *TodoItem*.

That means we will create five components in total.

Again, see the image below:

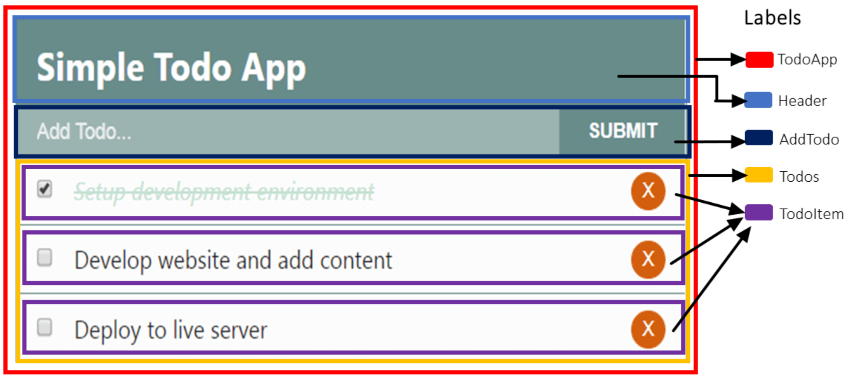


Fig: To-dos Application

Let's create these files.

Start by creating a folder called *`components`* in the *`src`* directory. Inside the *`components`* folder, create another folder called *`Layout`*.

Still in the *`components`* folder, create the following component files *`TodoApp.js`*, *`AddTodo.js`*, *`Todos.js`* *and* *`TodoItem.js`*.

Then go inside the *`Layout`* folder and create the *`Header.js`* *file*.

**Note**: I decided to place the *`Header.js`* file in a *`Layout`* folder. You can decide to place it alongside the other component files in the *`components`* folder.

Next, add the following code in the parent component file, *`TodoApp.js`* and save it:

```
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;
```

Also, go inside the `index.js` file and update it so it looks like this:

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

import TodoApp from "./components/TodoApp";

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

Save the file and check the frontend.

You should have a heading and a paragraph text being rendered on the screen.

What did we do?

We started by creating a *class-based* component in the `TodoApp.js` file. This component will serve as the container for the entire application as we have it in our app diagram.

In this file, we defined a React component class called *TodoApp* and extends the Component class in the React library. Inside this class, we have the ***`render()`*** method where we return the JSX that is rendered on the screen.

Make sure you import *`React`* object for the JSX to work.

*Please note: You cannot return more than one JSX element next to each other except you wrap them in a single element. In our case, we wrapped them inside a `<div>`. But in case you don't want a redundant wrapper around your elements, you can wrap everything in a React Fragment (a virtual element that doesn't get shown in the DOM).*

For instance, use *`<React.fragment>`* (or use shortcut, `<></>`) instead of `<div>`.

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

The *TodoApp* component which is the parent component was isolated from the app until we imported and rendered it in the `index.js` file.

In this file, we rendered the *TodoApp* component using a custom tag similar to HTML, `<TodoApp />`.

Now, instead of rendering a simple JSX element, we are rendering a React component.

A few notes –

- It's a good convention to use *UpperCamelCase* for the component file name (i.e `TodoApp.js`).
- Component names in React must be capitalized. In our case, *TodoApp*. This is necessary so that its instance (e.g `<TodoApp />`) in JSX is not considered as DOM/HTML tag.

If you are bringing in your own module/file, you will need to specify the relative path of that file from the current directory.

In our case, we specified *"`./components/TodoApp`".* Meaning the `TodoApp` file is located in the `components` folder inside the current directory.

The file extension defaults to ***.js***, so you don't need to append it.

## *Adding state data*

As we have it in the app diagram, the *AddTodo* component will carry the responsibility of accepting the user's input.

That means you will need a place to store the data received through the input. From there, you can display the data in the frontend.

Though, you can decide to have some default todos items displayed in the frontend. Or if you like, you can start by displaying an empty todos item.

Either way, you will need to define the *state* in your file.

For every component that will be accessing the state data, you will need to declare the state in the file of their closest common parent.

For instance, in our app diagram, the input field in the *AddTodo* component will be adding data to the *state*. Also, the delete button in the *TodoItem* component will be accessing the state data and removing todos item.

For that reason, the state data will live in the *TodoApp* component, which is their closest common parent.

Hope it's clear?

To add a state in a class component, we simply create a ***`state`*** object with different *key-value* pair. The value can be of any data type.

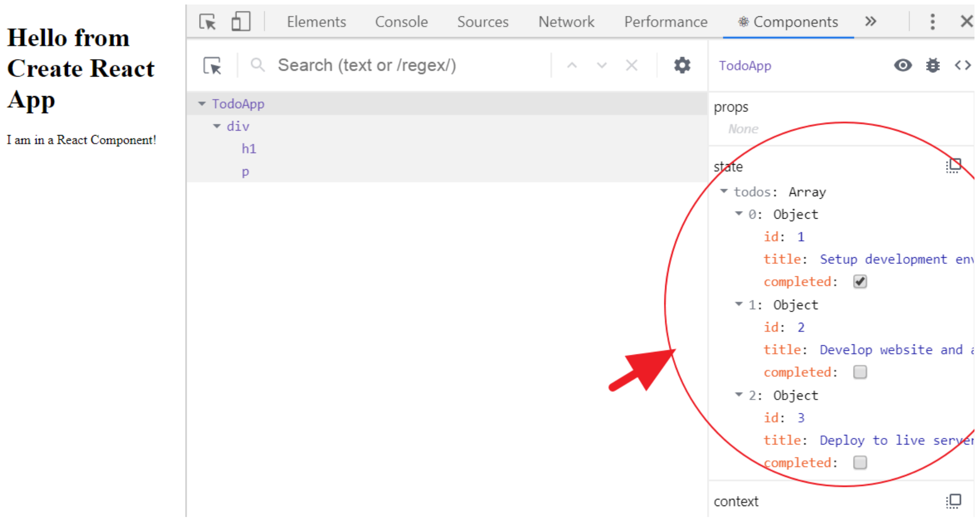
In the case of the todos data, we will have an array of objects.

So add the following code just above the ***`render()`*** method in the ***`TodoApp.js`*** file and save it:

```
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
    }
  ]
};
```

At this point, you can view the state of your application by looking at the *React DevTools*. Remember, we installed it in the previous chapter.

So open the browser DevTools and navigate to the *Components* tab to see the state of your application.



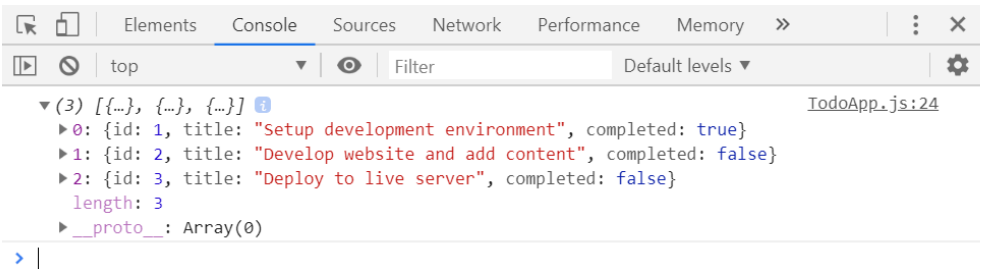
```
todos: Array
  0: Object
    id: 1
    title: Setup development env
    completed: ✓
  1: Object
    id: 2
    title: Develop website and a
    completed: ☐
  2: Object
    id: 3
    title: Deploy to live server
    completed: ☐
```

In case you don't have *React DevTools* installed, you can view the *state* of your application by logging *`this.state.todos`* in the *`render()`* method just above the *`return`* statement.

i.e

```
console.log(this.state.todos)
```

And then check the *Console* tab of your DevTools.



Elements Console Sources Network Performance Memory

top Filter Default levels

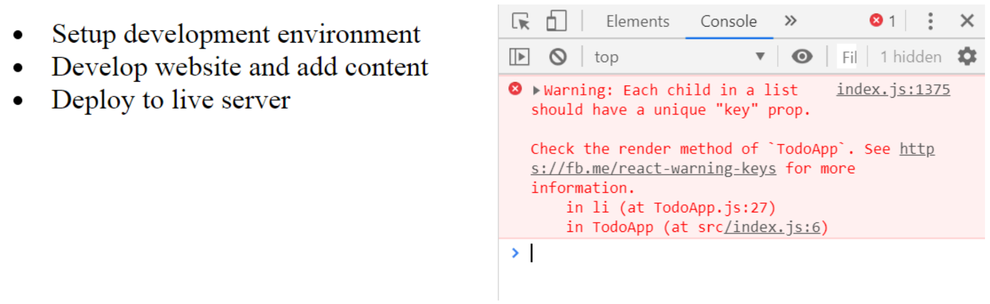
```
▼(3) [{…}, {…}, {…}]                                              TodoApp.js:24
  ▶0: {id: 1, title: "Setup development environment", completed: true}
  ▶1: {id: 2, title: "Develop website and add content", completed: false}
  ▶2: {id: 3, title: "Deploy to live server", completed: false}
   length: 3
  ▶__proto__: Array(0)
>
```

Now that we have each of the todos data in the state, let's render them on the screen.

To do this, update your *`render()`* method so it looks like this:

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

Save the *`TodoApp.js`* file and check the frontend.



- Setup development environment
- Develop website and add content
- Deploy to live server

```
▶Warning: Each child in a list     index.js:1375
should have a unique "key" prop.

Check the render method of `TodoApp`. See https://fb.me/react-warning-keys for more
information.
    in li (at TodoApp.js:27)
    in TodoApp (at src/index.js:6)
```

So what did we do?

After we defined the state data, we accessed it in the *`render()`* method using *`this.state.todos`*. Since its value is an array of objects as declared in the state, we looped through this array and output each of the todos item i.e *title*.

In React, we make use of the `map()` method which is a higher-order function to do this iteration.

This method returns a new array by applying a function on every array element.

So with the `map()` method, we are saying that for each of the todos data that we are looping through, we want to display the todos *title*.

*Remember that you can use a valid JavaScript expression inside the JSX through curly braces,* `{}`.

If you check the *Console* tab of your DevTools, You'll see React warnings just like what you have in the image above. We will take care of that in a moment.

For now, I want you to compare the frontend result and the app diagram. You will realize that another component called *Todos* has the responsibility to handle the todos list.

This is where we will apply the knowledge of *props* earlier explained.

What we want to do is to pass the state data from the *TodoApp* to the *Todos* component.

Recall that we can pass data down the tree through *props*. And I mentioned that the *prop* is just like the HTML attribute.

Let's apply that.

First, go inside the *`Todos.js`* file and create a component called *Todos*. At this point, you can render anything. We will update it soon.

In my case, I have this:

```
import React from "react";

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

export default Todos;
```

After that, open the *TodoApp.js* file and modify the *render()* method so it looks like this:

```
render() {
  return (
    <div>
      <Todos todos={this.state.todos} />
    </div>
  );
}
```

Since we are using an instance of a component, *<Todos />* in another file, you have to import the component. So, add this at the top of the *TodoApp.js* file.

```
import Todos from "./Todos";
```

At this point, you now have the *state* data in the *todos* prop. Thanks to this line – *<Todos todos={this.state.todos} />*

Now, we can access this data through *props* in the *Todos* component.

So let's update the *Todos.js* file so it looks like this:

```
import React from "react";

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

export default Todos;
```

Save your file. You should have the todos *title* rendered on the screen just like before.

Notice how we accessed the state data from within the child component, *Todos,* using `this.props.todos`.

Always remember, with props, we can access *state* data at different levels of the component hierarchy.

This is called ***prop drilling***.

It is a way to pass data (in this case, *todos* state data) from a parent component to the child component.

Once you have your data in the child component, you can also loop through it using the `map()` method as we have it in our code.

As a recap,

The *todos* data that come from the state of the *TodoApp* component is passed as *props* using `todos={this.state.todos}`. Then, we accessed it through `this.props.todos` from within the *Todos* component.

*Let's fix the console warnings.*

Remember that React is displaying warnings in the *Console* tab of the DevTools.

Fixing it is very simple.

Whenever you map through something (in this case, an array of objects), a list is created. And React want each child in the list to have a unique *key* prop. This helps React to identify which items have changed, added or removed.

To add a unique key prop to each of the lists, we will take advantage of the `id` we provided in the *TodoApp* state. We can access these `ids` the same way we accessed the `title`.

So go ahead and update the *`<li>`* element in the *Todos* component so you have:

```
<li key={todo.id}>{todo.title}</li>
```

Save the file and the error goes away.

*A quick question.*

Since our concern is assigning a unique key prop to the list. Can we use *indexes* as keys when looping through an array like this?

```
render() {
  return (
    <div>
      {this.props.todos.map((todo, index) => (
        <li key={index}>{todo.title}</li>
      ))}
    </div>
  );
}
```

Notice how we grabbed each array index inside the *`map`* method and assigned it to the *`key`* prop.

While this would work, React advises against using it except in some rare cases. This is because it could lead to unstable component behaviour.

One of the dangers of using the indexes as keys is that it causes issues with the component state when you reorder list items, add or remove items from a list. This is so because reordering an item changes the index, hence the component state may use the old key for a different component instance.

You are only safe to use the index as keys if you are sure that your list is never going to change or re-ordered or if there are no *`ids`* for the list items.

In our case, we have the *`ids`* and the list is going to change. So we grabbed it using *`key={todo.id}`*. And of course, this is the best approach to assign unique keys to the list items.

Before we go further, let's take some time and understand better the *Stateless* and *Stateful* component.

We have seen a *class component* (*Stateful*) but not a *functional component*.

***A function component*** prior to React 16.8 cannot maintain a state and lifecycle logic (ignore this for now). And as such, it is referred to as a ***stateless*** component.

But now, things have changed with the introduction of React Hooks.

You can now manage the class-based logic (state and lifecycle) inside of the function component. This gives us the flexibility to create a React application ONLY with function component.

Later in this book, you will learn how to use these React Hooks to manage this logic in a function component.

For now, we will manage them in the class component so that you can grab the fundamentals.

Back to our application.

If you take a look at the components we've created, only one of them is holding the *state* data. And that is the parent component, *TodoApp*.

That means we will retain this component as a *class-based*. The other component, *Todos* which is presently a class component can also be a function component.

This is because it does not hold state data.

For clarity, let's convert the class component, *Todos* to a function component.

## *Converting Class-Based Component to Functional Component*

In the `Todos.js` file, replace the code with the following:

```
import React from "react";

function Todos(props) {
  return (
    <div>
      {props.todos.map(todo => (
        <li key={todo.id}>{todo.title}</li>
      ))}
    </div>
  );
}

export default Todos;
```

If you save the file and check your application, you'll still have the todos items displayed.

So what changes?

Here, we created a function with the same component name instead of using ES6 class that extends *`React.Component`*.

The functional component does not require a *`render()`* method which is part of the lifecycle method (coming to that).

Also, notice that *`this.props`* in the class component was replaced by `props`. And to use this props, we included it as the function argument.

As you start with React, you may not always know whether to use a functional or class component. A lot of times, you will realize after a while that you chose the wrong type.

But as you create more components, making this choice will get easier.

Well, this has been solved as we can now create React components ONLY with functions. Thanks to React Hooks.

We will get to that later in this book.

Until then, one helpful tip is that a class component that only has markup within the `render()` method can safely be converted to a functional component.

In this React app, we will use the functional component simply for presentation as in the case of the *Header* component. There, we are rendering a simple heading text.

So, revert the *Todos* component to class component.

Before we proceed, let's use this opportunity to apply one of the React extensions we installed in chapter 1.

The *Es7 React/Redux/GraphQL/React-Native snippets*.

Remember?

Recall, we said this plugin will allow us to generate React components using shortcuts. This will no doubt help us to speed up development.

So let's apply it.

We use this abbreviation to generate a *class component*:

```
rce + tab
```

It will generate a class component in the name of the file. You see why we capitalized the component file names. Yes, the component names have to be capitalized!

Likewise, to generate a functional component, use this shortcut:

```
rfce + tab
```

Ok good.

If you go back to the app diagram, you'll see that another component called *TodoItem* has the responsibility to handle each of the todos items.

We did something like this earlier.

Open the `TodoItem.js` file and create a component called *TodoItem*. For the meantime, you can render anything.

Since this component will not be holding state, you can make it either class or functional component.

In my case, I will make it a class component.

```
import React from "react";

class TodoItem extends React.Component {
  render() {
    return (
      <div>
        Hello from TodoItem
      </div>
    );
  }
}

export default TodoItem;
```

Next, import the component in the `Todos.js` file using this line:

```
import TodoItem from "./TodoItem"
```

After that, replace the *`<li>`* element in the *`map()`* method with this line:

```
<TodoItem key={todo.id} todo={todo} />
```

*Note: Since we are mapping through the todos, don't forget to add key prop.*

Your code should look like this:

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

class Todos extends React.Component {
  render() {
    return (
      <div>
        {this.props.todos.map(todo => (
          <TodoItem key={todo.id} todo={todo} />
        ))}
      </div>
    );
  }
}

export default Todos;
```

At this point, each of the state data is present in the *`todo`* prop.

You can now access this data through *props* in the *TodoItem* component.

So let's update the *`TodoItem.js`* file so it looks like this:

```
import React from "react";

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

export default TodoItem;
```

Save all your files. You should have the frontend display as expected.

In the *TodoItem* component, take note of how we accessed the *title* using *`this.props.todo.title`*.

## *Creating the Header Component and Adding Styles to our App*

This is going to be straight forward since the *Header* component will not be holding the state. It will be rendering a simple heading text on the screen.

To avoid complications, let's make it a *functional* component. Here, also, you will learn how to start styling a React application.

To get started,

Open the `Header.js` file and add the following code:

```
import React from "react";

const Header = () => {
  return (
    <header>
      <h1>Simple Todo App</h1>
    </header>
  );
};

export default Header;
```

Save the file.

Next, go inside the parent component file, *`TodoApp.js`* and bring in the file at the top just like this:

```
import Header from "./layout/Header";
```

Then, add its instance, *`<Header />`* in the *`render()`* method so it looks like this:

```
render() {
  return (
    <div>
      <Header />
      <Todos todos={this.state.todos} />
    </div>
  );
}
```

Save the file. You should have the heading text displayed in the frontend.

Before we move on, notice how we are using the ES6 arrow function:

```
const Header = () => {
```

The above line is the same as this:

```
function Header() {
```

So go with the one you are most comfortable with.

## *Styling the Todos App*

Here, we will take a look at different ways to style a React app.

Just like adding styles to HTML file, you can style React JSX using the inline CSS styles and CSS classes.

React app can also be styled using the *CSS modules*. But this book covers only the inline and the CSS classes.

### *Starting with the inline styling.*

Recall, to use inline styling in HTML document, we pass a string of all the styles to the *`style`* attribute.

But here, we will assign a JavaScript object to the attribute.

Go inside the `Header.js` file and update the `h1` element to include a `style` attribute so it looks like this:

```
return (
  <header>
    <h1 style={{ fontSize: "25px", lineHeight: "1.447em",
margin: "0px" }}>
      Simple Todo App
      </h1>
  </header>
);
```

Save the file. Check the frontend for changes or inspect the `h1` element to see your CSS style declaration.

In the code, you'll notice two curly braces.

We already know that valid JavaScript expressions in JSX are written inside curly braces. The second curly brace is for the inline styling in the form of a JavaScript object.

Also, notice that the style keys are in *camelCase*.

This is also true for all *attributes, event handlers* in React with few exceptions like *aria-** and *data-** attributes which should be in lowercase.

We’ve said that before.

Another way to use an inline style in React is to use variables.

Still in the *Header.js* file, add the following code above the *return()* statement:

```
const headerStyle = {
  backgroundColor: "#678c89",
  color: "#fff",
  padding: "10px 15px"
};
```

Then update the *<header>* opening tag so you have:

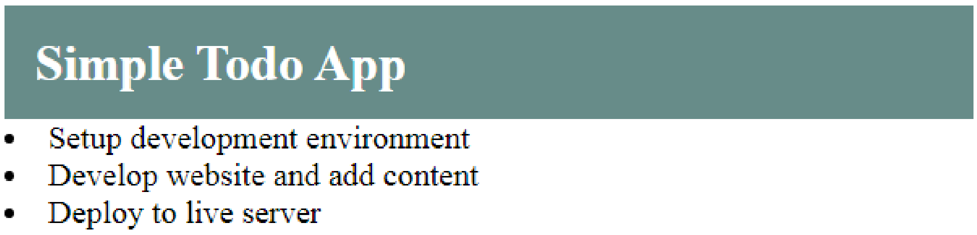
```
<header style={headerStyle}>
```

Save the file.

Now, your code should look like this:

```
import React from "react";

const Header = () => {
  const headerStyle = {
    backgroundColor: "#678c89",
    color: "#fff",
    padding: "10px 15px"
  };

  return (
    <header style={headerStyle}>
      <h1 style={{ fontSize: "25px", lineHeight: "1.447em",
margin: "0px" }}>
        Simple Todo App
      </h1>
    </header>
  );
};

export default Header;
```

And your frontend should look like this:



Simple Todo App

- Setup development environment
- Develop website and add content
- Deploy to live server

In the code, we created an object, ***headerStyle*** with the styling information and then refer to it in the ***style*** attribute.

Here, we used a curly brace.

## *Importing External CSS Stylesheet and using CSS Classes*

Unlike HTML where we add CSS classes to elements using the class syntax. In React JSX, we make use of a special syntax called *className*.

So let's see how to use this syntax.

But first, we need to create and import a CSS stylesheet.

Go inside the *`src`* folder and create a new file called *`App.css`*. Inside the file, add the following CSS styles:

```
* {
  box-sizing: border-box;
  margin: 0;
  padding: 0;
}
body {
  font-family: "Segoe UI", Arial, sans-serif;
  line-height: 1.4;
  background-color: #fcfcfc;
  color: #1f1f1f;
}
```

Save the file.

Next, decide where to import the file. You can either do this in the parent component file, *`TodoApp.js`* or in the entry file, *`index.js`*.

To be on the same page, open the *`index.js`* file and add this line of code above the *`ReactDOM.render()`* method:

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

Save the file.

Now open the *`TodoApp.js`* file and add a `className` called *`container`* to the *`div`* that wraps the instances of the components.

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

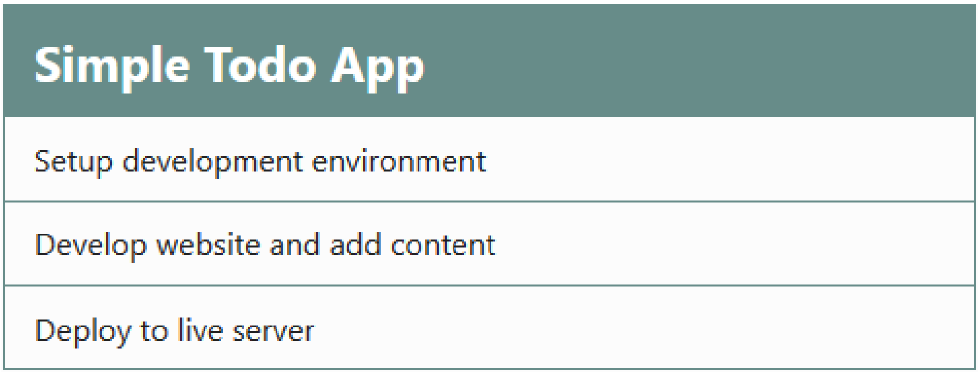
Also, open the *`TodoItem.js`* file and add a *`className`* called *`todo-item`* to the *`<li>`* element.

```
return (
  <li className="todo-item">{this.props.todo.title}</li>
);
```

Next, go inside the *`App.css`* file and add the following styles:

```
.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;
}
```

Save all the files and check the frontend.


Simple Todo App

Setup development environment

Develop website and add content

Deploy to live server

Later in this book, you will learn how to do dynamic styling. For instance, you will be able to apply styles to your todos items once a task is completed.

But first, let’s add checkboxes to the todos list.

### *Adding Checkboxes to the Todo Items*

You should be familiar with handling this type of input field in HTML form. But in React, form handling works a little bit different.

Let’s start by adding the input checkbox to our app.

Inside the *`li`* element, add this *`input`* element just before the *`{this.props.todo.title}`* in the *TodoItem* component.

```
<input type="checkbox" />
```

Update the *`App.css`* file by adding this style:

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

Save your files. Check the frontend and see the checkboxes.

By default, the input type (in this case, checkboxes) are being handled by the DOM – i.e they have the default HTML behaviour.

That is why you can toggle the boxes. This type of input is called ***uncontrolled input***.

But in React, the input fields are meant to be controllable.

This takes us to another important subtopic.

## *Controlled Component*

To make the input fields controllable, the input data (toggling of checkboxes) has to be handled by the component *state* and not the browser DOM. With this, the state will serve as a *single source of truth*.

Meaning, the input checkbox would no longer listen to its internal state (i.e the browser DOM) but the state in your app.

This is necessary because the component state will not change unless you change it.

Let's see how it works.

If you take a look at the state, we have a Boolean value (*`true`* or *`false`*) assigned to every *`completed`* key in the todos data. We can tap into that to toggle the checkboxes.

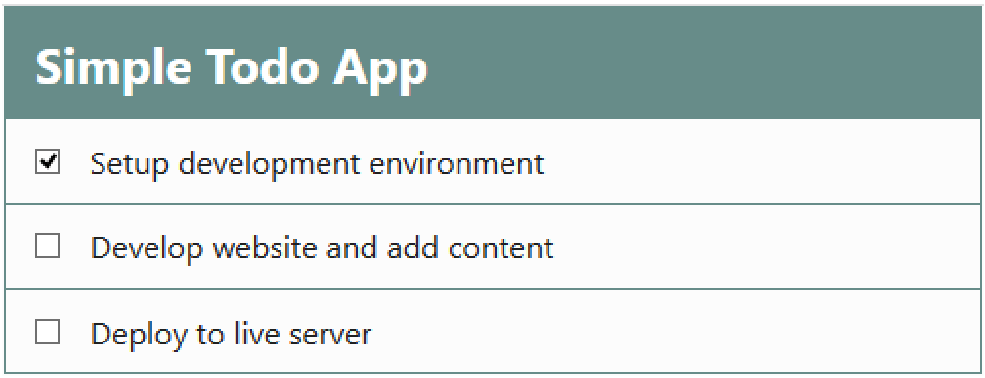
So go inside the *`TodoItem.js`* file and add *`checked`* props to the input tag and then assign *`{this.props.todo.completed}`*.

So you have:

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

Remember, just like the *`title`*, we have access to the *`completed`* value in this component.

Save the file.


Simple Todo App
☑ Setup development environment
☐ Develop website and add content
☐ Deploy to live server

At this point, you have succeeded in making the input checkbox a controlled input because it now listens only to the state in your application.

Now, if you try to toggle any of the checkboxes, nothing will happen. This is because each of the *checked* attributes is assigned a value equal to the current value of the state.

Remember, only the first task is assigned to be completed.

We need a way to change the state whenever users click any of the checkboxes.

React already gives us a hint through the Console tab of the browser DevTools.

Open the console.

You'll see a warning displayed as a result of the added *checked* attribute.

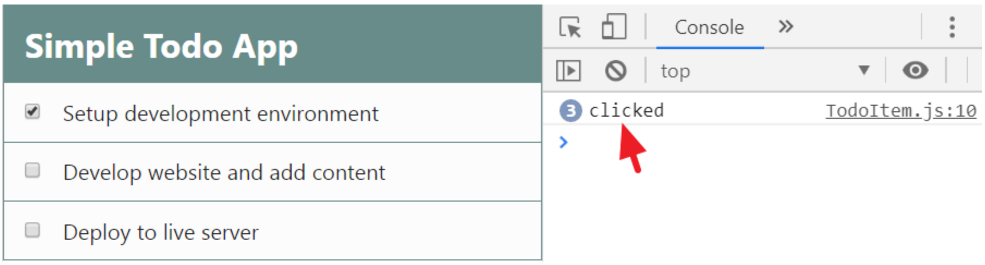
React is telling us to add an *onChange* handler to keep track of any changes in the field. Else, it wouldn't know if the input field is checked or not.

So let's update the *`input`* tag in the *`TodoItem.js`* file so you have:

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

Save the file and look inside the Console tab, you'll see that the warning is gone.

For the meantime, we are assigning to the handler, a callback function that will log a *"clicked"* text in the Console whenever the checkbox is clicked.


Simple Todo App
☑ Setup development environment
☐ Develop website and add content
☐ Deploy to live server
Console
top

```
3 clicked                                   TodoItem.js:10
```

Now, instead of logging a text in the console, we want to toggle the checkboxes anytime they are being clicked.

To do this, we need to understand how to raise and handle events.

## *Raising and Handling Events*

In our app, the parent component, *TodoApp* is the one that holds the state data. This component, therefore, is the *ONLY* one that can change it. Meaning the *TodoItem* component, which is the one handling the checkboxes, cannot change the state data in the parent component, *TodoApp.*

We need to find a way to access the *state* data from the *TodoItem* (where we have the input checkboxes) and toggle the `completed` value to `true` or `false` in the *TodoApp* component.

To do this, we will need to raise an event from the *TodoItem* up a level to *Todos*, and then into *TodoApp* component. In other words, we need to climb a ladder.

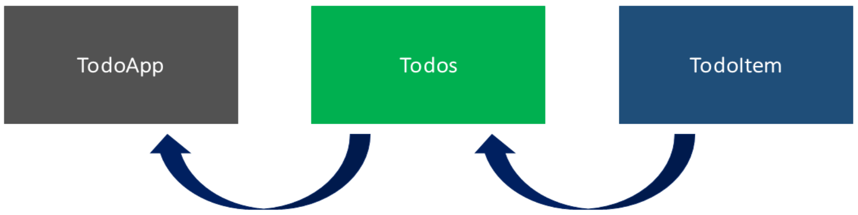


Fig: Raising and Handling Event

The *TodoItem* component will raise the event while the parent component, *TodoApp* will handle the event.

And the way we do that is through *props*.

This is kind of tricky but trust me it's very simple.

You can either go from the child to parent component or the other way round. I prefer the latter.

Let's do it.

We will first enable communication between these components.

Starting from the parent component, *TodoApp*, add this handler method, ***`handleChange`*** just above the ***`render()`*** method:

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

You can name this method anything you like.

Let's see how we can communicate with this method from the child component.

Start by passing this method to the *Todos* component through the *props*.

So update the *Todos* instance, ***`<Todos />`*** so you have:

```
<Todos todos={this.state.todos}
handleChange={this.handleChange} />
```

Again, you can name the ***`handleChange`*** prop anything you like.

**Note** – We are using ***`this.handleChange`*** to reference the ***`handleChange()`*** method because it is part of the class.

Now, you have the ***`handleChange()`*** method assigned to the ***`handleChange`*** prop. Its data can be accessed through props in the *Todos* component. From there, we can pass it to the *TodoItem* component.

Let's update the ***`<TodoItem />`*** instance in the ***`Todos.js`*** file so you have:

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

At this point, the *handleChange()* data can be accessed from the *TodoItem* component.

So update the *onChange* handler in the *TodoItem* component so you have:

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

This time, make sure you have parenthesis, `()` attached to the *handleChange*.

Save all your files.

Now, if you click any of the checkboxes, the *onChange* event will trigger and will call the *handleChange()* method in the parent component, *TodoApp*.

For now, we are only logging a text in the console.

Let's go a step further.

We need to identify which one of the checkboxes is clicked. To do this, we need to pass along their respective *ids* through the callback function.

Update the *onChange* handler in the *TodoItem* component so it looks like this:

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

Remember, just like the *title* and the *completed* value, we also have access to the *id* in this component.

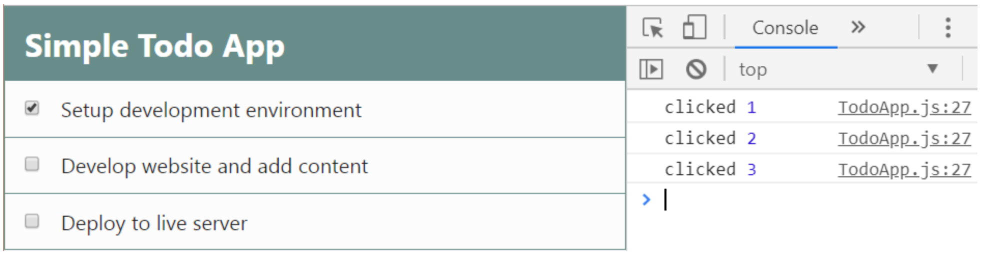
Save the file.

Then go inside the *TodoApp* component and also update the *handleChange* method.

```
handleChange = id => {
  console.log("clicked", id);
};
```

Note how we are receiving and logging the *id*.

Save the file. Look inside the console and click on the checkboxes. This time, you will see their respective *ids*.


Simple Todo App
Setup development environment
Develop website and add content
Deploy to live server
Console
top

```
clicked 1    TodoApp.js:27
clicked 2    TodoApp.js:27
clicked 3    TodoApp.js:27
```

Good. We are heading somewhere.

## *Updating the state using the setState() method*

Our aim here is to change the state of the checkbox whenever it is clicked.

In React, we do not modify the state directly. Instead, we update the state through a method we inherited by extending `React.Component`.

This method is called `setState().` It tells React that we are updating the state. Then React figures out what part of the state is changed and update ONLY that part in the real DOM.

All we need to do in the `handleChange` method is to check if the `id` of the clicked checkbox matches any of the todos items.

If it does, we will flip the `completed` value in the *todos state* from `true` to `false` and vice versa.

Now go inside the *TodoApp* component and update the `handleChange` method so it looks like this:

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

The `id` on the first line comes from the *TodoItem* component (it contains the checked `id`). You know that already!

On looping through the `todos` data, we check if any of the items `id` matches the checked `id`. Then, we flip the `completed` value.

Save the file and check your application. You should be able to toggle the checkboxes.

***Adding Styles when any of the todos is Completed***

This is straight forward. You can give it a try!

Ok good try.

Let's do it together.

In the *TodoItem* component, add the following styles in the ***render()*** method but above the ***return*** statement:

```
const completedStyle = {
  fontStyle: "italic",
  color: "#c5e2d2",
  textDecoration: "line-through"
};
```

Then, update the ***return*** statement so it looks like this:

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

In the code, we introduced a new tag, ***span*** (you can use any HTML tag you like) and then added a ***style*** attribute to it.

We also used the *ternary operator* in the ***style*** attribute to dynamically change the CSS style if any of the todos item(s) is/are *completed*.

As a refresher,

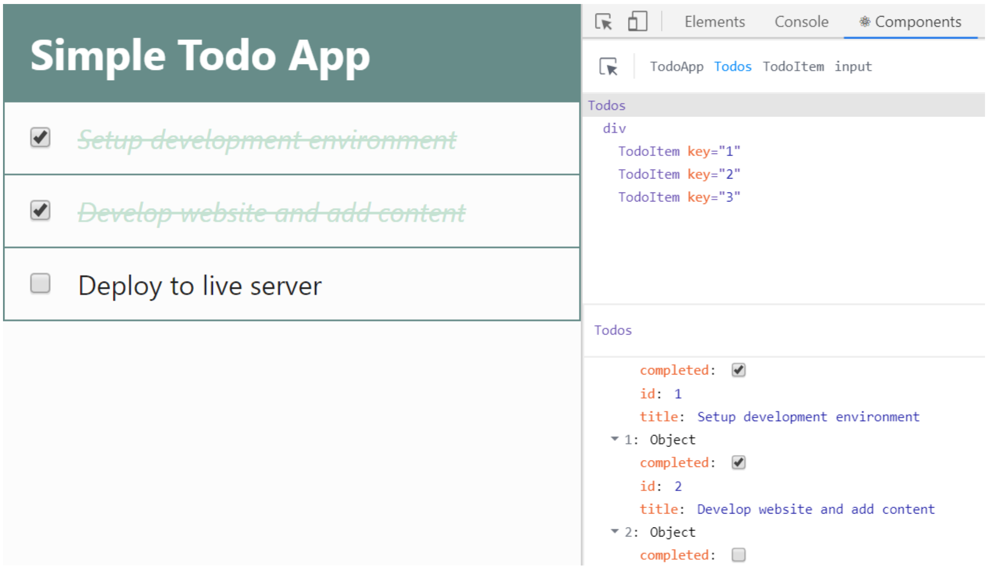
Ternary operator (or inline if-statement) as used here will check if any of the todos items is completed, else, no CSS styles would be assigned to the attribute.

This is how it works:

```
(condition) ? (true return value) : (false return value)
```

i.e if the condition is ***true*** (in our case, if the task is mark completed), we apply the second statement, ***completedStyle*** (we created this variable as an object holding the styling information in the same component), else, we apply *null*, i.e no style.

Ok. Save the file and check the frontend.


Simple Todo App
~~Setup development environment~~
~~Develop website and add content~~
Deploy to live server
Elements
Console
Components
TodoApp Todos TodoItem input

```
Todos
  div
    TodoItem key="1"
    TodoItem key="2"
    TodoItem key="3"
```

```
Todos
      completed: ✓
      id: 1
      title: Setup development environment
  ▾ 1: Object
      completed: ✓
      id: 2
      title: Develop website and add content
  ▾ 2: Object
      completed: ☐
```

In the DevTools, you can also play around and familiarize yourself with the React tools now called *Components* as seen on the right side of the image above.

## *Using Destructuring*

If you take a look at the *TodoItem* component, we were writing multiples *`this.props.todo`* to grab the *`id`*, *`title`* and *`completed`* values.

This can be a pain in the neck if your application gets complex.

Instead of doing these, we can pull variables out of the *todo*. In other words, we will "destructure" the *todo* and get these variables from it.

To do this, go inside the *TodoItem* component and add this code just above the *return* statement:

```
const { completed, id, title } = this.props.todo
```

Then, replace every *this.props.todo* with their corresponding variables. For instance, *this.props.todo.completed* should be replaced with *completed* and so on.

So your *return* statement now looks like this:

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

## *Deleting Items from the todos*

This will be similar to how we handled the input checkbox. As you know, the items we will be deleting live in the *TodoApp* component, while the delete button will live in the *TodoItem* component.

That means we will need to raise an event from the *TodoItem* component and move up levels until we get to the *TodoApp* component where the event will be handled.

Let's get down.

Start by adding a delete button in the *TodoItem* component.

So, add this ***button*** element below the input tag:

```
<button className="btn-style"> X </button>
```

We will update it later.

Let's give it a style by adding the following in the *App.css* file:

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

Save the files.

Please take note of the different methods we are using to add styles to our application.

Now, you should have the styled buttons displayed in the app. Though not doing anything yet.

As usual, we will enable communication between these components to raise an event.

So go inside the *TodoApp* component and add a ***deleteTodo*** method above the ***render()***.

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

Still in the component, update the `<Todos />` to include `deleteTodo={this.deleteTodo}`. So you have:

```
<Todos
  todos={this.state.todos}
  handleChange={this.handleChange}
  deleteTodo={this.deleteTodo}
/>
```

Save the file and move a level down inside the *Todos* component and update the `<TodoItem />` so you have:

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

Finally, back in the *TodoItem* component. Update the `button` element to include an `onClick` event handler that will trigger the `deleteTodo` method in the parent component.

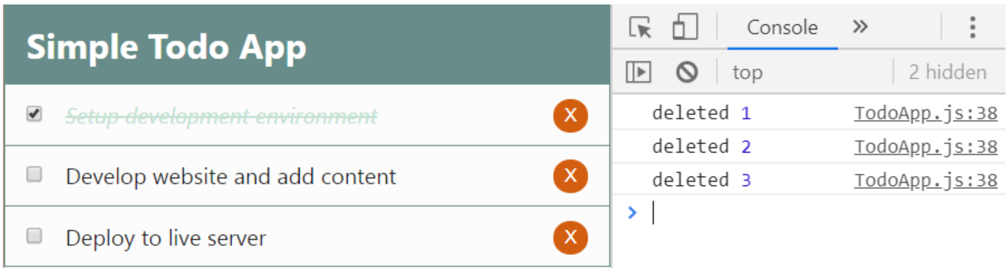
You should have this:

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

Remember we used destructuring earlier to pull the `id` from the `todo`.

Save all your files and check the frontend.

Open the Console tab and try to delete any of the *todos* items.


Simple Todo App
~~Setup development environment~~
Develop website and add content
Deploy to live server
X
Console
top
2 hidden

```
deleted 1    TodoApp.js:38
deleted 2    TodoApp.js:38
deleted 3    TodoApp.js:38
```

At the moment, we are logging *"deleted"* text alongside the `id` of the deleted items.

Up to this point, we are repeating what we did for the checkbox. If it's not clear, revisit the earlier explanation.

Next, we will manipulate the state and remove any of the deleted items from the list.

The way we do that is by using the *filter()* method.

Just like the *map()* method, the *filter()* is also a higher-order function. It returns a new array by applying a condition on every array element.

In this case, we only want to return the todos items that do not match the *id* that will be passed in. Any *id* that matches will be deleted.

Now, update the *deleteTodo* method so you have:

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

With the *filter()* method, we are saying that for each of the todos data that we are looping through, we want to retain the once that the *id* is not equal to the *id* passed in.

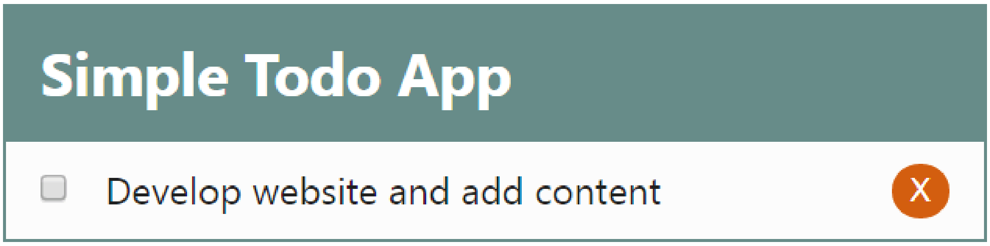
Please note the spread operator (…) in the code. It allows us to grab the current todos item(s) at every point. As this is necessary for the code to work.

Save the file.

Test your application by trying to delete any todos item.

Yeah! It works.

I deleted the first and the last todos items. This is what I have left.



If you get to this point, good job!

Moving on.

## *Adding a text input field and a submit button*

In React, all the different types of input fields follow the same approach. We've seen how the checkbox type of input field works.

Using the same pattern, we will add a text input field that will allow users to add todos items to the list.

Let's start by adding the following code inside the empty ***`AddTodo.js`*** file (remember we created the file earlier):

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

class AddTodo extends 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;
```

Since we will be getting data through the user's interaction (i.e through the *input* field), the component will, therefore, hold state. For that reason, it will be a *class-based* component.

Don't forget, it can be a functional component if we use React Hooks. Coming to that later in this book.

In the code, we added some attributes including the class name to the *input* elements. We will style these elements using their respective CSS class names in the *App.css* file.

Also, notice how we are extending the *Component* class in the React library.

Unlike the previous class components where we used *React.Component*. Here, we are using *Component* after importing it from the *react* module like this:

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

Now, you are exposed to different methods of using a class component.

Ok. Let's move on.

To use this component in our application, we will import it inside the *`TodoApp.js`* file using this line of code:

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

Then, add *`<AddTodo />`* inside the *`render()`* method just below the *`<Header />`*.

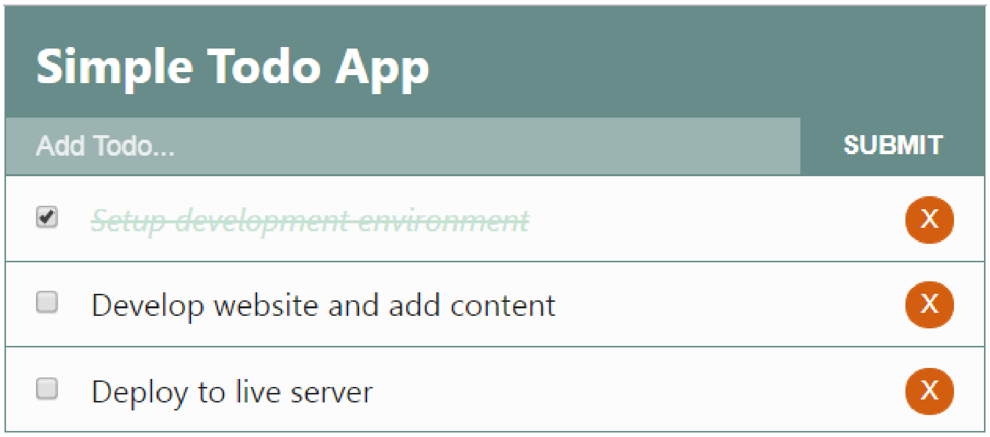
You should have this:

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

Finally, add the following styles to the *App.css* file:

```
.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;
}
```

Save all your files and check the frontend.



- [x] ~~Setup development environment~~
- [ ] Develop website and add content
- [ ] Deploy to live server

As we did for the checkbox, we have to make the form input field a *controlled* field. The first step is to have a *state* manage our data.

So, add this code just above the `render()` method in the *AddTodo* component:

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

We can now take this data and assign it to a `value` prop in the text `input` element.

So you have:

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

**Note** – We use *checked* prop for input checkbox and *value* prop for the text input.

Now, the text input field is being controlled by the component state and not the DOM. You will not be able to write anything in the field because it is assigned a value equal to the current value of the state.

The value is empty as declared in the state.

To change the state, we need to update it through the *setState()* method.

Again, just like the *checked* prop used for checkboxes, the *value* prop also exhibits a warning in the console.

React is reminding us that we need to add an *onChange* handler that will keep track of any changes in the field.

So let's update the text `input` so it includes this:

```
onChange={this.onChange}
```

Then add this code above the `render()` method:

```
onChange = e => {
  console.log("hello");
};
```

Remember we are using `this.onChange` because the method, `onChange()` is part of the class.

Save your file.

If you try to write in the input field, you'll see *"hello"* text being displayed in response to every keystroke inside the *Console*.

Next, we need to handle the event and update the state.

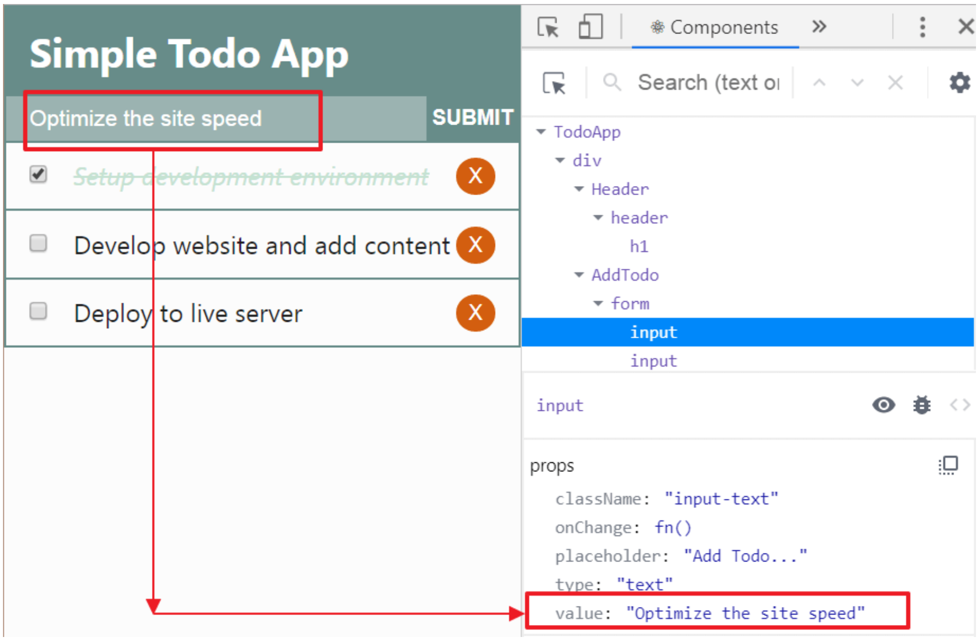
Let's update the *onChange* method to this:

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

Save the file.

Now you should be able to write something in the input field.

You will also see the state of your app in real-time if you open the *React tools*. See the image below.



```
className: "input-text"
onChange: fn()
placeholder: "Add Todo..."
type: "text"
value: "Optimize the site speed"
```

What's happening?

If you type anything inside the input field, the *onChange* event handler will be triggered.

This will then call the *`onChange()`* class method that will re-render the state using the *`setState()`* method.

In the *`setState()`* method, we are passing the current value of the state (i.e the input text) to the *`title`* using *`e.target.value`*.

And if you recall from vanilla JavaScript DOM API, the predefined parameter, ***`e`***, hold some important information about the event.

From there, you can target the specific input field and grab the updated value.

As a recap!

Handling input form fields in React requires you to constantly keep track of the state data. That is, for every keystroke, you update the state and have the most updated version of what the user is typing into the form.

*Handling form that has more than one text input field*

For instance, if your form requires fields for the *name*, *email* and *password*. First, you would want all those fields included in the state and assigned an empty string.

After that, you'll have to modify the *onChange* method to this:

```
onChange = e => {
  this.setState({
    [e.target.name]: e.target.value
  });
};
```

Then, you add a *name* attribute to each of the input tags and assign a value with the same name you declared in the state. For instance, in our case, we will have *name="title"* included in the text *input* tag.

If you apply these changes in your code, your *AddTodo* component should look like this:

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

class AddTodo extends Component {
  state = {
    title: ""
  };
  onChange = e => {
    this.setState({
      [e.target.name]: e.target.value
    });
  };
  render() {
    return (
      <form className="form-container">
        <input
```

```
          type="text"
          placeholder="Add Todo..."
          className="input-text"
          name="title"
          value={this.state.title}
          onChange={this.onChange}
        />
        <input type="submit" value="Submit" className="input-
submit" />
      </form>
    );
  }
}

export default AddTodo;
```

With these changes, you can add as many text *input* fields as you want.

Instead of having multiple methods to handle different input fields, we modified the *setState()* method to this:

```
[e.target.name]: e.target.value
```

As long as the value of the *name* attribute in the *input* element matches what you have in the state. It will work perfectly.

## *Updating the todos list*

At the moment, the page will reload if you try to submit a todos item and update the state. We need to handle that.

To submit todos items, we will make use of the ***onSubmit*** event handler on the form element.

Let's quickly do that.

Update the *<form>* tag in the *AddTodo* component to include *onSubmit = {this.onSubmit}*.

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

Then, add the following code above the ***render()*** method and save the file:

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

For the meantime, if you submit your todos items, they will be displayed in the Console tab of your browser.

Note how we are preventing the default behaviour of the form submission.

Now, instead of logging the user's input (titles) in the console, we want to pass them from this component to the *TodoApp* component and update the *state* data.

To do that, we will need to raise and handle the event just like we have been doing.

But this time, we will raise an event from the *AddTodo* component (that accept the user's input) to the parent component, *TodoApp* where the state data to be updated live.

If you check the app diagram or open the React Tools to see the component hierarchy, you will see that we are moving up a level from the *AddTodo* to the *TodoApp* component.

Let's do it again.

As usual, let's start by enabling communication between those components.

Starting from the parent component, *TodoApp.* Add this class method above the ***render()*** method:

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

Since we are expecting the todos *title* from the *AddTodo* component, you have to include it as the function argument as seen in the code above.

Next, pass this class method to the *AddTodo* component by updating the *<AddTodo />* so you have:

```
<AddTodo addTodo={this.addTodo} />
```

Now, the ***addTodo*** method can be accessed through props in the *AddTodo* component.

So update the ***onSubmit*** method in the *AddTodo* component so you have:

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

Save the file. You should still be able to see your todos in the console.

Before we move on, let's clear the input field once we have submitted a todos item for subsequent entry.

Simply update the ***`onSubmit`*** method so you have:

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

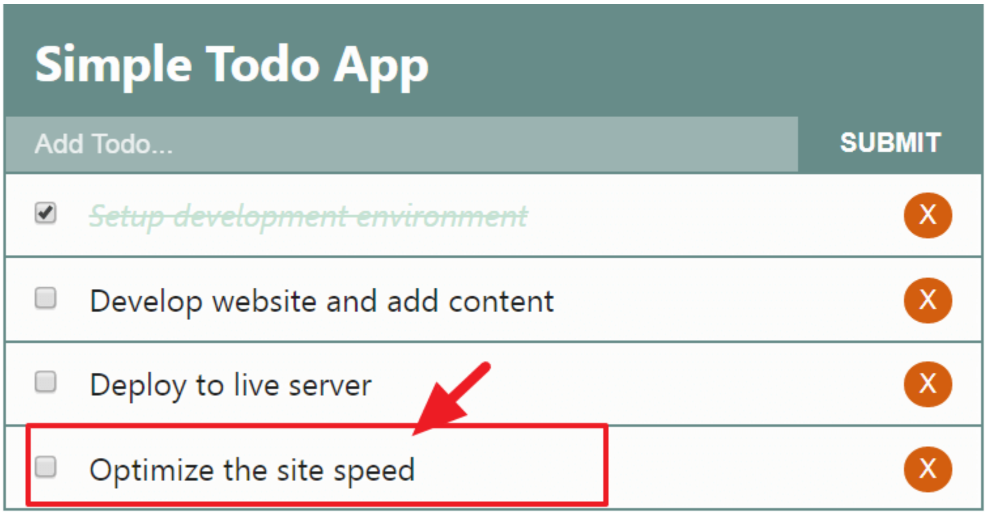
Finally, we can update the state.

Back to the *TodoApp* component, update the ***`addTodo()`*** method so you have:

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

Save the files.

Go to the frontend and add a new todos item to the list.



What did we do?

In the code, we started by defining an object for the new item. In this object, we are passing a set of key-value pair. Here, we have the `title` from the user's input, the `completed` key assigned a `false` value so that the checkbox is not selected by default.

Then, for the meantime, we are working with hardcoded `id`.

With the `setState()` method, we are re-rendering the state. We are adding the new item to the current todos list which can be grabbed using the spread operator (…).

*Generating random ids for the todos list items*

If you try to submit more than one todos item, React will trigger a console warning telling you that the *ids/keys* are meant to be unique.

This is happening because we are assigning an `id` of 4 to every new todos item received through the input field.

That is not ideal.

The `ids` help React to identify which item is added or removed. To fix this warning, we will install something that will generate random `ids` that we can use. This is called the UUID (Universal Unique Identifier).

To install it, stop the server with CTRL + C.

From your terminal, run:

```
C:\Users\Your Name\my-todo-app > npm i uuid
```

Once installed, start the server again with *npm start*.

In case you encounter any error while restarting the server, simply delete the *node_modules* folder.

Run *npm install* to recreate the folder and then re-run *npm i uuid*.

You should be all set.

To use these *`ids`* in your app, you need to import the UUID in the *`TodoApp.js`* file.

So go ahead and add this line below the list of *`import`* statements.

```
import { v4 as uuidv4 } from 'uuid';
```

After that, replace any hardcoded *`id`* value with *`uuidv4()`*.

For instance, instead of having:

```
id: 1,
```

You'll have:

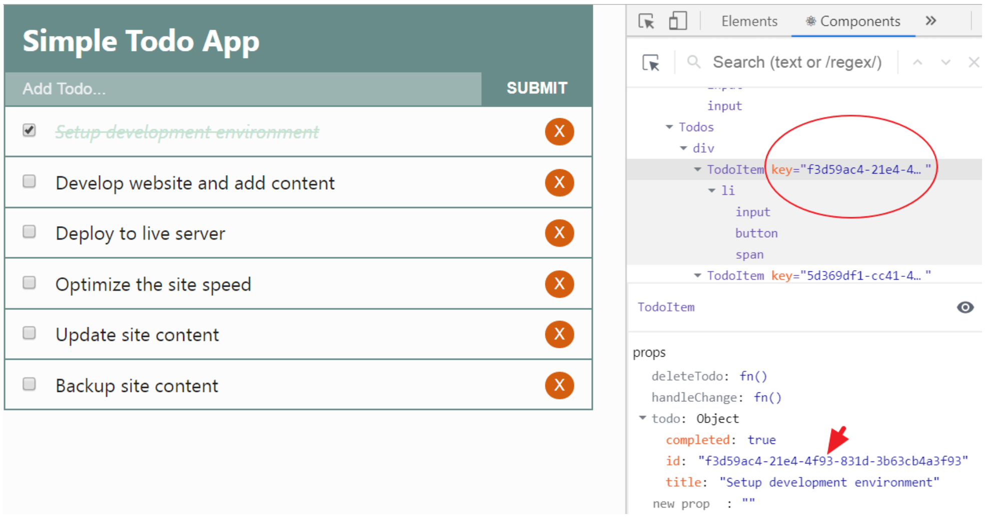
```
id: uuidv4(),
```

Do the same for the other *`ids`*.

Save the file and test your application.

You should be able to add as many todos to the list without any console error.

If you check the React tools, you'll see how the todos items are assigned unique *`ids`* from the UUID.



~~Setup development environment~~
Develop website and add content
Deploy to live server
Optimize the site speed
Update site content
Backup site content

```
        input
  ▾ Todos
    ▾ div
      ▾ TodoItem key="f3d59ac4-21e4-4… "
        ▾ li
            input
            button
            span
      ▾ TodoItem key="5d369df1-cc41-4… "
TodoItem

props
  deleteTodo: fn()
  handleChange: fn()
▾ todo: Object
    completed: true
    id: "f3d59ac4-21e4-4f93-831d-3b63cb4a3f93"
    title: "Setup development environment"
  new prop   : ""
```

At this point, we have reached another milestone.

Now, let’s see how we can make our awesome application available to the world.

# Chapter 3
# Deploy React App to Gh-pages

Here, we will deploy our React application to GitHub pages for free so that we can access it on the web.

"GitHub Pages is a static site hosting service that takes HTML, CSS and JavaScript files straight from a repository on GitHub, optionally runs the files through a build process and publishes a website/app"

To get started, you have to create a GitHub account (https://github.com/) if you don't have one. Also, install Git (https://git-scm.com/) – a version control system (VCS) for tracking changes in computer files – for your operating system and then, set it up (https://www.atlassian.com/git/tutorials/install-git).

Once that's done,

You will need to move all your React code to your GitHub account. And this can be done in two phases.

First, you will move your files to the local repository and then to the remote repository.

Let's get started.

Stop the server with CTRL + C

## *Initialize the Project folder as a Git Repository*

The first thing you would want to do when setting up a git project is to initialize your local Git repository. This will create a ***.git*** folder (hidden by default) in your project directory. Fortunately, this is available by default when you set up a React project with *create-react-app* CLI.

To see the ***.git*** folder, open your project folder, go to the *View* tab and click on *Options* located at the top right side. This pops up a new window. Next, click on *View* and then select *Show hidden files, folders, or drives* radio button under the *Advance settings*. Finally, uncheck *Hide extensions for known file types*.

Click the *Apply* button and then *Ok*.

Now you should be able to see the ***.git*** folder in your project directory.

But if you do not set up your project with the CLI, run this command from your terminal to create it:

```
C:\Users\Your Name\my-todo-app > git init
```

*Make sure you are in your project directory in the terminal.*

For the rest of us that set up the project using the *create-react-app* CLI, we don't need to reinitialize the git repository.

Instead, we just have to make sure that all of the new files become part of the repo.

Let's do that.

If you are just starting with Git, you can optionally set up a *username* and a commit *email* address.

You can do these using the following commands:

```
git config --global user.name 'ibaslogic'
```

And…

```
git config --global user.email 'ibaskunle@gmail.com'
```

Ensure you use your *username* and *email* and take note of them because you'll need them to push your application online.

You can confirm that you set them up correctly using these commands:

```
git config --global user.name
```

```
git config --global user.email
```

They should return your username and email respectively if all went well.

## *Deploy to local repository*

Now let's deploy our code into the local repository.

Run this command:

```
C:\Users\Your Name\my-todo-app > git add .
```

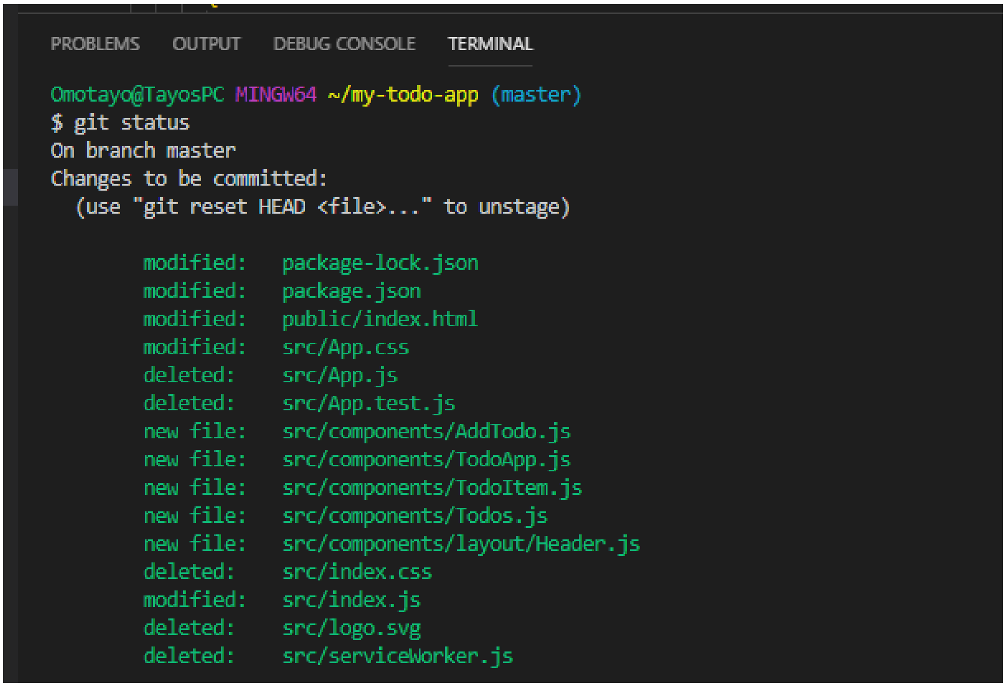
This keeps all your working files in the staging area. Please don't forget the dot(.)

*Note: The dot(.) indicates that you are adding all the files in the staging area and putting them in the local repository. If you want to add a specific file for instance `index.js`, you would run something like this – git add `index.js`*

To see what is in the staging area. Let's run this command:

```
C:\Users\Your Name\my-todo-app > git status
```

This allows you to check the current status of the working tree. You'll have the files in *green* if they are in the staging area. Else, you'll have them in *red*.



```
Omotayo@TayosPC MINGW64 ~/my-todo-app (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   package-lock.json
        modified:   package.json
        modified:   public/index.html
        modified:   src/App.css
        deleted:    src/App.js
        deleted:    src/App.test.js
        new file:   src/components/AddTodo.js
        new file:   src/components/TodoApp.js
        new file:   src/components/TodoItem.js
        new file:   src/components/Todos.js
        new file:   src/components/layout/Header.js
        deleted:    src/index.css
        modified:   src/index.js
        deleted:    src/logo.svg
        deleted:    src/serviceWorker.js
```

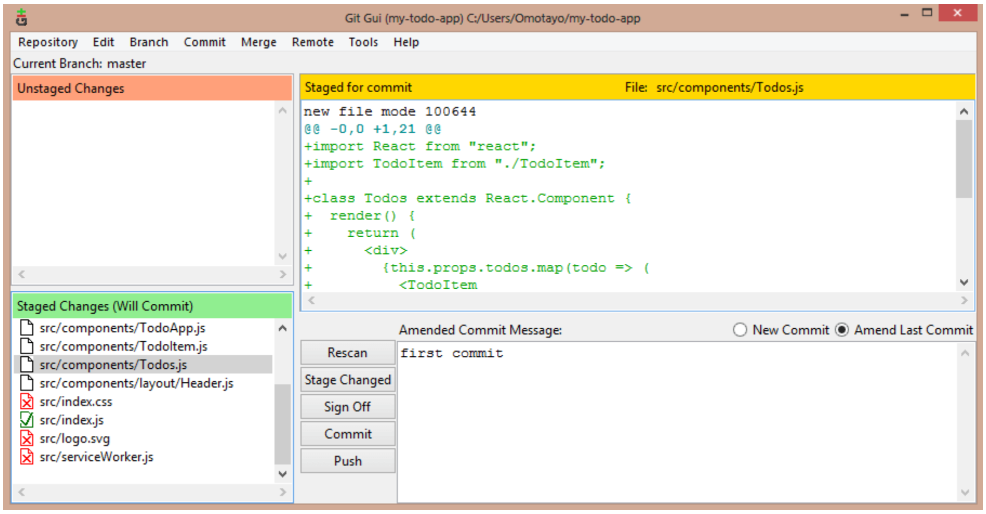
Next, run this command to commit all the changes:

```
C:\Users\Your Name\my-todo-app > git commit -m 'first commit'
```

You specify your commit message within the quote. Modify to whatever suit you.

At this point, your project files are now in the local repository. To view them, open the Git GUI you installed in your machine. Click on the *Open Existing Repository* and then search for your project folder.

Open it and select the *Amend Last Commit* button to view your staged Changes.


Git Gui (my-todo-app) C:/Users/Omotayo/my-todo-app
Repository Edit Branch Commit Merge Remote Tools Help
Current Branch: master
Unstaged Changes
Staged for commit
File: src/components/Todos.js
Staged Changes (Will Commit)
Amended Commit Message:
New Commit [✓ Amend Last Commit]
Rescan
Stage Changed
Sign Off
Commit
Push

```
new file mode 100644
@@ -0,0 +1,21 @@
+import React from "react";
+import TodoItem from "./TodoItem";
+
+class Todos extends React.Component {
+  render() {
+    return (
+      <div>
+        {this.props.todos.map(todo => (
+          <TodoItem
```

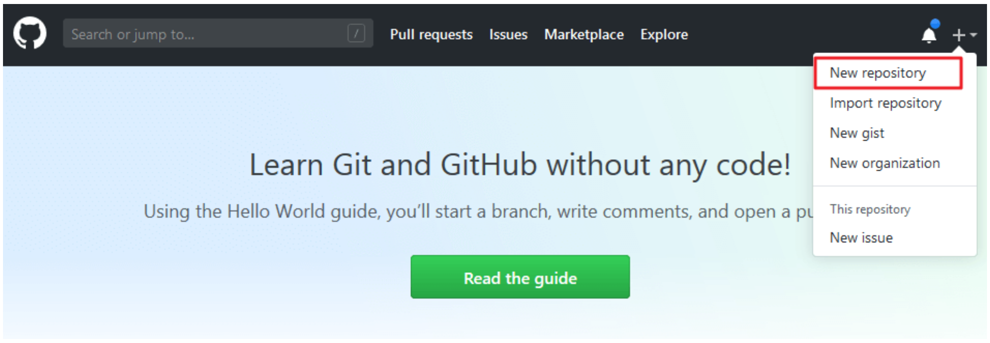
src/components/TodoApp.js
src/components/TodoItem.js
src/components/Todos.js
src/components/layout/Header.js
src/index.css
src/index.js
src/logo.svg
src/serviceWorker.js

```
first commit
```

Now, you are ready to push these files to the remote repository.

## *Deploy to remote repository*

Go ahead and log into your GitHub account and create a new repository.



On the new page, you will be required to name your repository. In my case, I will name it *todoapp*. Other fields are optional and self-explanatory.

*Note that React already included a README.md file in your project folder. So DON'T Initialize this repository with a README.*

Once you are done, click on the *Create repository* button.

This takes you to a page where you will find additional information on what to do.

Find the command that looks like this and run it in your terminal:

```
git remote add origin https://github.com/Ibaslogic/todoapp.git
```

If you can’t find it, make sure you modify the URL above to reflect your repository. What this command does is add the repo as the remote repository.

Finally, run this command to push to the master branch:

```
git push -u origin master
```

Now reload your GitHub page. You should have your project files and folders already pushed to your account.

Good.

We are almost done! Now we can publish our app to gh-pages.

## *Deploy Todos app to Gh-pages*

Back to the terminal, let's install a package that will create a gh-pages branch on GitHub.

So run:

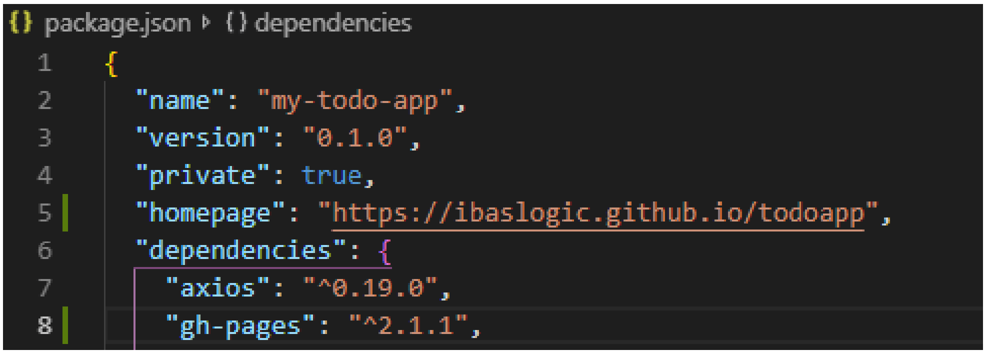
```
C:\Users\Your Name\my-todo-app > npm install gh-pages
```

After that, open the ***`package.json`*** file in your root directory and add this line of code at the top level.

```
"homepage": "https://username.github.io/repository-name",
```

Modify the above URL to include your GitHub *username* and *repository name*.

In my case, the *package.json* file now looks like this:



```
{} package.json ▸ {} dependencies
{
  "name": "my-todo-app",
  "version": "0.1.0",
  "private": true,
  "homepage": "https://ibaslogic.github.io/todoapp",
  "dependencies": {
    "axios": "^0.19.0",
    "gh-pages": "^2.1.1",
```

Still, in the *package.json* file, locate the *scripts* property and add these lines of code:

```
"predeploy": "npm run build",
"deploy": "gh-pages -d build"
```

Your *scripts* should look like this:

```
"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"
},
```

Save the file.

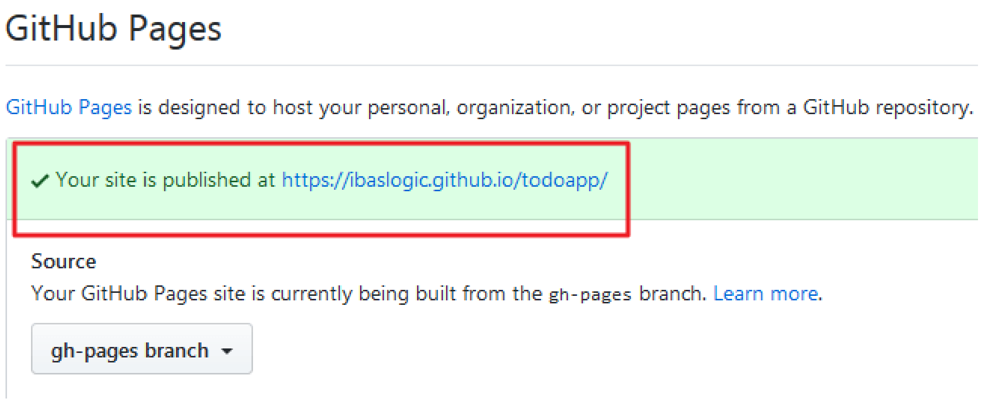
Lastly, run this command to push your file to the gh-pages:

```
C:\Users\Your Name\my-todo-app > npm run deploy
```

This command will create a *`build`* folder in the root directory. This folder will contain production-ready files that will be deployed.

Once your app is successfully deployed, you can visit the URL you assigned to the *`homepage`* property in the *`package.json`* file and see your application.

Alternatively, you can go back to your GitHub repository and click on the *Settings* tab. Then scroll down to GitHub pages section to see the URL.



GitHub Pages

GitHub Pages is designed to host your personal, organization, or project pages from a GitHub repository.

✓ Your site is published at https://ibaslogic.github.io/todoapp/

Source

Your GitHub Pages site is currently being built from the `gh-pages` branch. Learn more.

Congratulations!

Test your application and proudly share with the world.

Before we get started with the second application, *Simple Meme Generator*, let’s go a step further by learning how to fetch data from a URL endpoint.

# Chapter 4
# React HTTP Request and Lifecycle Methods

In the last chapter, we deployed our working todos app on the web (app available here, https://ibaslogic.github.io/todoapp/).

Up to this moment, we have the default todos items hardcoded in the app component. While that is fine, in reality, making an HTTP request or fetching data from an API is most likely what you'll be doing when building an application with a frontend framework. You can either work with a third party API or have your server to work with.

In the case of our *Todos* app, we are going to request some sort of todos data from the server and list them on the frontend instead of manually adding them in the *state*. To do this, we can make use of the native *Fetch API* or *Axios* to perform an HTTP request to a REST API.

*HTTP request? Fetch API? Axios? REST API?*

*What are they?*

***An HTTP*** request is a packet of information that one computer (client) sends to another computer (server) to convey something.

Once the message is received by the server, a response is expected by the client.

***The Fetch API*** allows us to perform an HTTP request to a server and handle responses. **Axios**, on the other hand, is a 3rd party HTTP library that allows us to make this request as well.

In this book, we will show you how both of them works.

This is necessary because you may be working on an existing project that is using either of the two.

We will use *Axios* in our *Todos* application to fetch data and then, the native *Fetch API* in the next project, *Simple Meme Generator*.

One of the benefits of using *Axios* over *Fetch API* is that it supports all modern browsers, including support for IE8 and higher by default. This is because it is compiled with *Babel*. For *Fetch API* to support older browsers, it needs what is called *polyfill* – a piece of code used to provide modern functionality on older browsers.

Finally, Understanding the ***REST API*** is necessary if you want to get data from another source on the internet. In our case, we will be getting a piece of data (a list of todos) when we link to a specific URL endpoint.

To get a clearer picture of how this works, let's get the hands dirty.

We will be working with a FREE online REST API called *JSONPlaceholder*. This allows us to mimic a real live server and have a backend to work with. From there, we can get a list of fake todos items into our Todos app.

Let's dive in.

First, you need to start your server by running *npm start*. After that, delete all the hardcoded todos data in the *TodoApp* component.

The state now looks like this:

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

Save the file. You will see that your app is now displaying empty todos items.

Next, head over to *JSONPlaceholder* website, https://jsonplaceholder.typicode.com. Scroll down to *Resources* section and click on the *todos* link.

Here, you will see a list of 200 todos that we will be working with.

Please note the URL, we will make an HTTP request to use its todos data.

But before we do that, we will need to install *Axios* to perform this request.

So stop the server with CTRL + C and run this from your terminal:

```
C:\Users\Your Name\my-todo-app > npm i axios
```

Now, to start making this request, you'll need to understand the lifecycle methods.

## *The Lifecycle methods*

Every React component you create always goes through a series of events or phases from its birth to death. For instance, if you create a component to render something on the screen, it will go through a couple of phases to display the content. You can think of this component going through a cycle of birth, growth and finally death.

In React, these phases are mainly three –

*Mounting* – As the name implies, this is the phase when React component mounts (created and inserted) the DOM. In this phase, the component is birthed.

*Updating* – Once the component is mounted, next is to get updated if necessary. Remember that component gets updated when there is/are *state* or *prop* changes, hence trigger re-rendering. All of that happens in this phase.

*Unmounting* – This phase ends the component lifecycle because, in it, the component is removed from the DOM.

In each of these phases, React provides lifecycle methods that we can use to monitor and manipulate what happens within the component. Though, we have been using one of these lifecycle methods all this while.

The `render()` method in our class components is one of the methods.

This method is the only required lifecycle method within a React class component. It's responsible for rendering React elements in the DOM and it is called during the *mounting* and *updating* phase.

React has several optional lifecycle methods, of which some are deprecated. But the common once include –

`componentDidMount()` – This method is called after the component is rendered.

`componentDidUpdate()` – This is called immediately after updating occurs.

`componentWillUnmount()` – This is called just before a component is unmounted or destroyed.

### ***The componentDidMount() method***

As a beginner, you will most likely be working with the *`componentDidMount()`* in addition to the *`render()`* lifecycle method. This is because, most of the time, you will be making an HTTP request.

And one of the common uses of this lifecycle method is to get data from somewhere.

Remember, we want to load todos data from a remote endpoint. So, the *`componentDidMount()`* method is a good place to make this type of HTTP request. This ensures that your data is fetched as soon as possible.

So let's apply this method. Start your development server with *npm start*.

## *Making a GET request*

As expected, we use the GET HTTP method to fetch data. So let's start by importing the *Axios* library in the *`TodoApp.js`* file:

```
import axios from "axios";
```

Then add the following code right above the ***render()*** method in the *TodoApp* component:

```
componentDidMount() {
  axios.get("https://jsonplaceholder.typicode.com/todos")
    .then(response => console.log(response.data));
}
```

Save the file and check the console. You will see a list of objects containing the todos data.

Make sure you are connected to the internet.

*Note: If you are using the VsCode Prettier – Code Formatter extension, your code may re-format on save. Do not worry, all that matter is that it works.*

We can limit the number of todos data to 10 just for brevity.

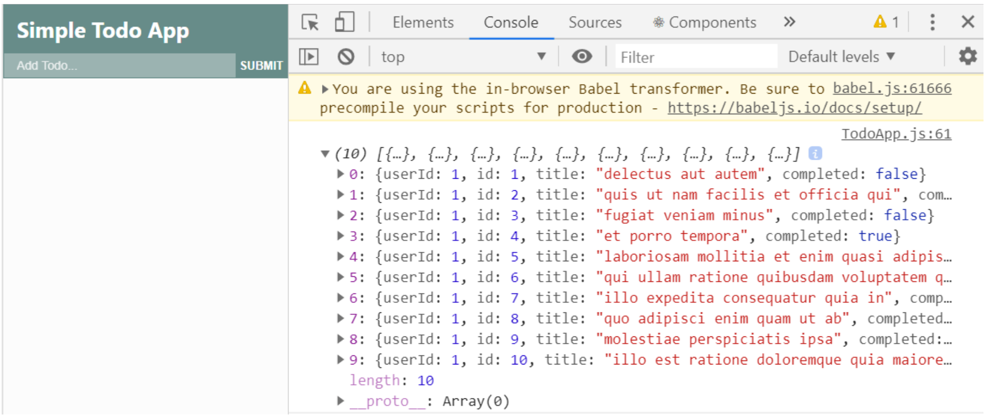
This can be done either by appending the query string parameter (*_limit=10*) to the URL or by adding them as a second argument in the ***get()*** method.

I will show you both options.

First, update the endpoint URL so you now have this:

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

Save the file and go back to the console. You should see the todos data limited to 10.



```
⚠ ▶You are using the in-browser Babel transformer. Be sure to babel.js:61666
  precompile your scripts for production - https://babeljs.io/docs/setup/
                                                              TodoApp.js:61
 ▼(10) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}]
  ▶0: {userId: 1, id: 1, title: "delectus aut autem", completed: false}
  ▶1: {userId: 1, id: 2, title: "quis ut nam facilis et officia qui", com…
  ▶2: {userId: 1, id: 3, title: "fugiat veniam minus", completed: false}
  ▶3: {userId: 1, id: 4, title: "et porro tempora", completed: true}
  ▶4: {userId: 1, id: 5, title: "laboriosam mollitia et enim quasi adipis…
  ▶5: {userId: 1, id: 6, title: "qui ullam ratione quibusdam voluptatem q…
  ▶6: {userId: 1, id: 7, title: "illo expedita consequatur quia in", comp…
  ▶7: {userId: 1, id: 8, title: "quo adipisci enim quam ut ab", completed…
  ▶8: {userId: 1, id: 9, title: "molestiae perspiciatis ipsa", completed:…
  ▶9: {userId: 1, id: 10, title: "illo est ratione doloremque quia maiore…
   length: 10
  ▶__proto__: Array(0)
```

The second option is to add a config object as the second parameter in the *`get()`* method like so:

```
componentDidMount() {
  axios.get("https://jsonplaceholder.typicode.com/todos", {
      params: {
        _limit: 10
      }
    })
    .then(response => console.log(response.data));
}
```

You should have the same result as the first method.

Here we use the *`params`* option to set a query string parameter in the config object.

To be on the same page, let's go with the first method.

Before we go ahead and display the todos list in our app, let's explain what we did.

First, I guess you remember why we called the *`componentDidMount()`* method. Again, it is a good place to make an HTTP request.

Here, we are using HTTP GET method to retrieve data from an endpoint.

By using *Axios*, we make use of *`axios.get()`* method. This accepts the URL of the endpoint and an optional config object as the second parameter.

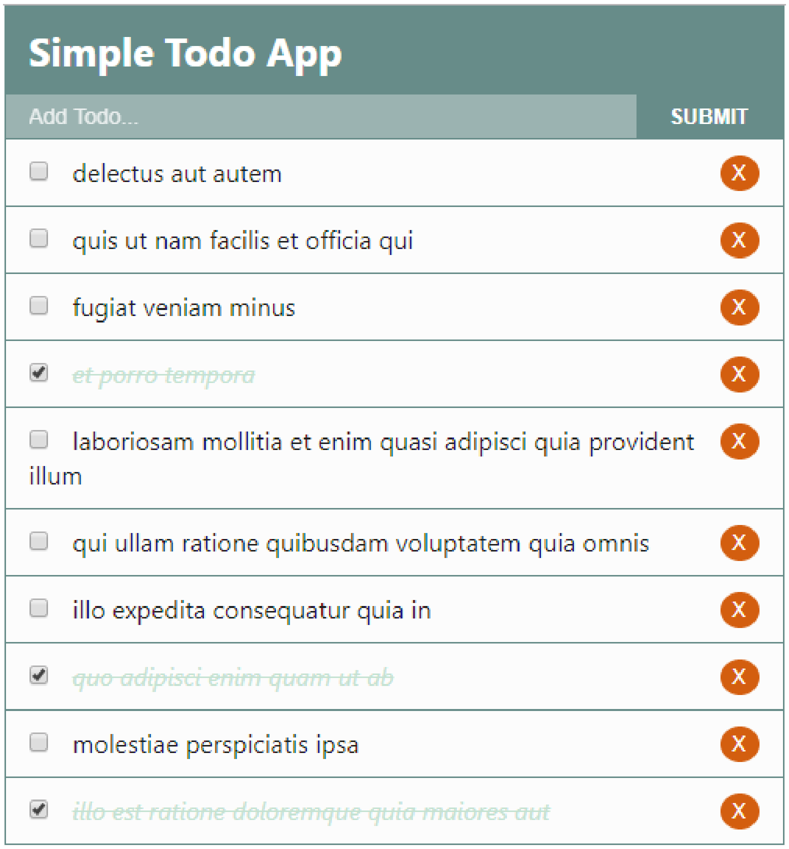
This method, *`axios.get()`* returns a promise which must be resolved to access the data. To do that, we use the *`.then()`* method which will receive a response that contains the data we need.

Now, we can update the state with these data.

Update the code so it looks like this:

```
componentDidMount() {

axios.get("https://jsonplaceholder.typicode.com/todos?_limit=
10")
    .then(response => this.setState({ todos: response.data
}));
}
```

Save the file and see the update in the frontend.



Simple Todo App

Add Todo...

SUBMIT

- [ ] delectus aut autem X
- [ ] quis ut nam facilis et officia qui X
- [ ] fugiat veniam minus X
- [x] ~~et porro tempora~~ X
- [ ] laboriosam mollitia et enim quasi adipisci quia provident illum X
- [ ] qui ullam ratione quibusdam voluptatem quia omnis X
- [ ] illo expedita consequatur quia in X
- [x] ~~quo adipisci enim quam ut ab~~ X
- [ ] molestiae perspiciatis ipsa X
- [x] ~~illo est ratione doloremque quia maiores aut~~ X

Thanks to the `setState()` call. React knows that the todos have changed from the empty data we assigned in the state to the data we received from the JSONPlaceholder API. It then calls the `render()` method once again to display the new data.

As you can see, some are marked as *completed* because they are assigned a `true` value. You can visit the endpoint URL in your browser and see what is assigned to each of the `completed` keys.

## *Making a POST request*

Now if you add more entries to the todos list, you will only be updating the UI and not the backend.

What we want is to make a POST request to the REST API (*JSONPlaceholder*), get a response and update the UI.

If you check the *JSONPlaceholder* home page, you will see a number of the request you can make under the *Routes* section. For instance, you can make a POST request, DELETE request and so on.

You would want to make a POST request whenever you are adding something.

Remember, we are working with a Fake Online REST API, so our data doesn't get saved to the *JSONplaceholder* server. However, it does complete the request and gives us a response. Thereby mimics a real-life backend.

Now, let's modify the ***addTodo*** class method in the *TodoApp* component so you have:

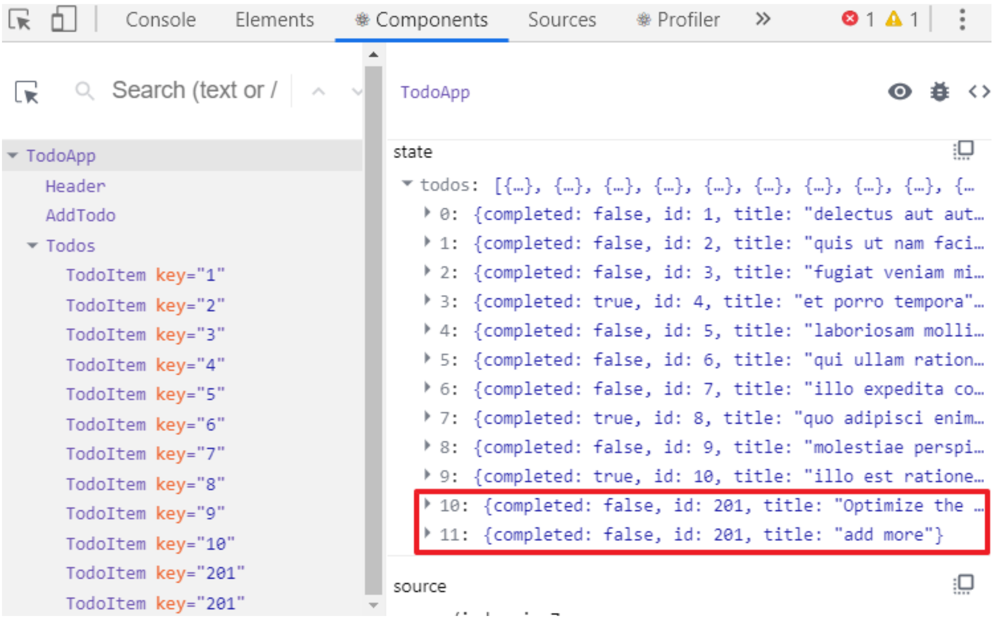
```
addTodo = title => {
  axios.post("https://jsonplaceholder.typicode.com/todos", {
      title: title,
      completed: false
  })
  .then(response =>
    this.setState({
      todos: [...this.state.todos, response.data]
    })
  );
};
```

Save the file.

Please note that when you make a POST request to the JSONplaceholder server, an *id* is generated for the returned data. So, you don't need the UUID anymore. You can delete them.

However, this *id* is static for every submission. This is because we are not actually updating their database.

As seen below (in the React DevTools), it returned an *id* of 201 for every submission. Remember, the endpoint has a total of 200 todos items.



```
▾ TodoApp
    Header
    AddTodo
  ▾ Todos
      TodoItem key="1"
      TodoItem key="2"
      TodoItem key="3"
      TodoItem key="4"
      TodoItem key="5"
      TodoItem key="6"
      TodoItem key="7"
      TodoItem key="8"
      TodoItem key="9"
      TodoItem key="10"
      TodoItem key="201"
      TodoItem key="201"
```

```
state
▾ todos: [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…
  ▸ 0: {completed: false, id: 1, title: "delectus aut aut…
  ▸ 1: {completed: false, id: 2, title: "quis ut nam faci…
  ▸ 2: {completed: false, id: 3, title: "fugiat veniam mi…
  ▸ 3: {completed: true, id: 4, title: "et porro tempora"…
  ▸ 4: {completed: false, id: 5, title: "laboriosam molli…
  ▸ 5: {completed: false, id: 6, title: "qui ullam ration…
  ▸ 6: {completed: false, id: 7, title: "illo expedita co…
  ▸ 7: {completed: true, id: 8, title: "quo adipisci enim…
  ▸ 8: {completed: false, id: 9, title: "molestiae perspi…
  ▸ 9: {completed: true, id: 10, title: "illo est ratione…
  ▸ 10: {completed: false, id: 201, title: "Optimize the …
  ▸ 11: {completed: false, id: 201, title: "add more"}
source
```

As expected, React will throw a console warning expecting a unique key. This will not be the case if you are working with a real server. Obviously, the next *id* will be 202 and so on.

With that out of the way, what exactly is happening in the code?

Unlike the *`axios.get()`* where the second parameter (config object) is optional. Here, it is *required* because it will contain the data that you want to send. Together with the todos *`title`*, we are assigning *`false`* value to the *`completed`* property so that a new entry by default will not be checked.

The *`axios.post()`* will also return a promise which must be resolved using the *`.then()`* method. The data it receives from its response is appended to the todos list as seen in the *`setState()`* method.

## *Making a Delete Request*

At this point, you should be comfortable making requests to the server. They all follow the same pattern. And a DELETE request is not an exception.

Let's do it together.

As expected, we will modify the `deleteTodo` method in the *TodoApp* component so it looks like this:

```
deleteTodo = id => {

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

};
```

Save the file.

Test your application by deleting any of the todos item(s) on the list.

Did it work? Yes.

Good.

Did we do anything special?

No! Except that to make a DELETE request, you need not only the URL but also the `id` of the item to delete.

Fortunately, we have the `id` as received in the function argument. And in JavaScript, we can insert dynamic expression (in our case, the `id`) within a string using the back-tick (\`  \`).

Now, if you delete any item from the list, you will get a response from the virtual server and the UI get updated.

*Notice that you can only* ***add*** *or* ***delete*** *items if you have internet access. This shows there is a communication between the client and the backend.*

Again, if you are working with a real backend, the todos item(s) will be deleted from the database or added to the database as the case may be and then update the UI.

## *The componentDidUpdate() method*

This is another important lifecycle method in React. As I mentioned earlier, this method is called immediately after updating occurs – i.e after state or props changes.

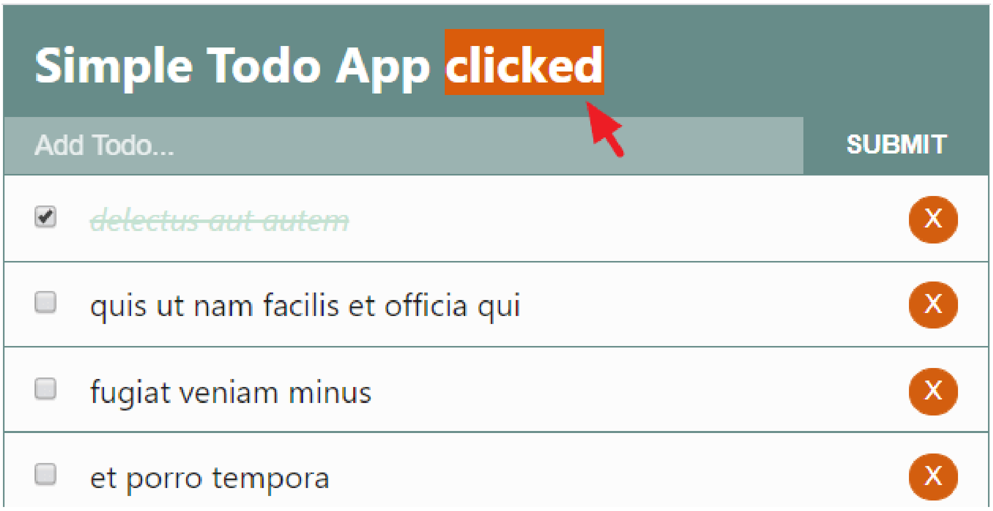
It is a perfect place to manipulate the DOM when the component has been updated. Also, in this method, you can make API calls after specific conditions have been met.

So let's see how we can apply this method in our todos application.

First, what is the goal here?

We want a situation whereby when we click on the checkboxes, a text with random background color is displayed in the header of our app.

Like so:



How can we achieve this?

We will define another property in the `state` object of the parent component, *TodoApp*, assign a default Boolean value and pass it down to the *Header* component through props. Once this prop changes as a result of a clicked checkbox, this lifecycle method will run and execute whatever we want.

Let's see it in practice.

We'll start by converting the *Header* component from function to class component. This is necessary since this component will manage the lifecycle logic.

*In the coming chapter, you'll learn how you can manage the state and lifecycle logic in a functional component with React Hooks.*

For now, update the `Header.js` file so you have:

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

class Header extends Component {
  render() {
    const headerStyle = {
      backgroundColor: "#678c89",
      color: "#fff",
      padding: "10px 15px"
    };
    return (
      <header style={headerStyle}>
        <h1 style={{ fontSize: "25px", lineHeight: "1.447e
m", margin: "0px" }}>
          Simple Todo App <span id="inH1"></span>
        </h1>
      </header>
    );
  }
}

export default Header;
```

The conversion is pretty straight forward. Please note that we've added a *span* element to the *h1* heading. In there, we will display our text.

Now, add this lifecycle method just above the *render()* method:

```
componentDidUpdate(prevProps, prevState) {
  // update logic here
}
```

This method accepts the previous props and state as parameters. These are simply the state and props before the update.

We can use them to check for updates by comparing them to the current snapshot which we can get through *this.state* or *this.props*.

At the moment, they are null. So we need to pass data to them.

We can either define a state in this *Header* component to get the *`prevState`* or pass down data via props from the parent component to get the *`prevProps`*.

As I mentioned earlier, we will define another property in the *`state`* object of the parent component and pass it down to the *Header* component.

With this, we will have access to the *`prevProps`*.

So, update the state in the *TodoApp* component to include the *`show`* property and pass it to the *`<Header />`*.

```
...
class TodoApp extends React.Component {
  state = {
    todos: [],
    show: false
  };
  ...
  render() {
    return (
      <div className="container">
        <Header headerSpan={this.state.show} />
        ...
      </div>
    );
  }
}
...
```

Save the file.

Now, we have access to the default data through the ***headerSpan*** prop in the *Header* component.

Next, update the lifecycle method so you have:

```
componentDidUpdate(prevProps, prevState) {
  if (prevProps.headerSpan !== this.props.headerSpan) {
    console.log("props change");
  }
}
```

In the code, all we are doing is comparing the previous and current snapshot.

Remember, we want these changes to happen once we click the checkboxes. So we need to get the current prop by updating the method that is called whenever we click on the input checkbox.

In the parent component, *TodoApp*, update the *`handleChange`* method to include the *`show`* object property:

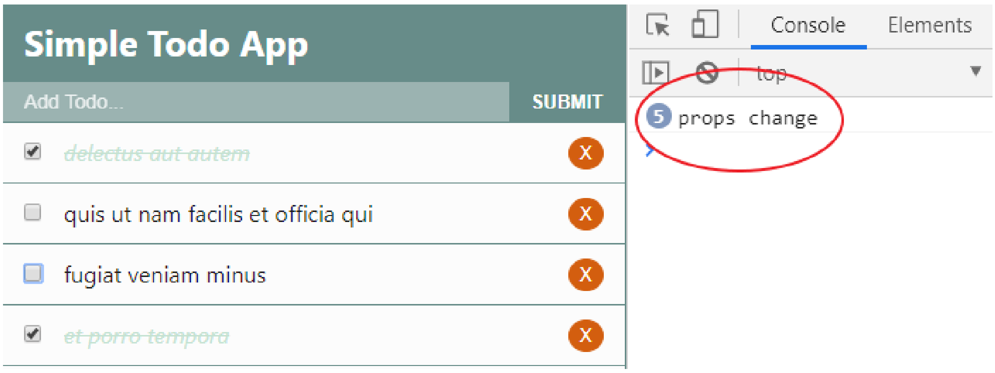
```
handleChange = id => {
  this.setState({
    todos: this.state.todos.map(todo => {
      if (todo.id === id) {
        todo.completed = !todo.completed;
      }
      return todo;
    }),
    show: !this.state.show,
  });
};
```

As seen in the code, we are inverting the current value of the *`show`* property. This way, anytime you click on the checkbox, you'll always get a negated value.

Once this happens, the *`componentDidUpdate()`* method compares these props and detect changes. It then executes whatever you define in it.

Let's test it.

Save the file. Head over to the console and click on the checkboxes.



We are almost there.

Instead of logging text in the console, we want to manipulate the DOM.

Update the method so you have:

```
componentDidUpdate(prevProps, prevState) {
  var x = Math.floor(Math.random() * 256);
  var y = Math.floor(Math.random() * 256);
  var z = Math.floor(Math.random() * 256);
  var bgColor = "rgb(" + x + "," + y + "," + z + ")";

  if (prevProps.headerSpan !== this.props.headerSpan) {
    document.getElementById("inH1").innerHTML = "clicked";
    document.getElementById("inH1").style.backgroundColor
= bgColor;
  }
}
```

This is self-explanatory.

All we are doing is referencing the *span* element in the JSX and then add text and dynamic background color.

Once again, test your application.

***The componentWillUnmount() method***

This lifecycle method is called when a component is about to be destroyed or removed from the DOM. Most of the time, it is used to perform cleanups. For instance – cancelling the network request, removing event listeners.

In our todos app, we will use this method to trigger an alert when an item is about to be deleted from the todos list.

This is pretty straight forward.

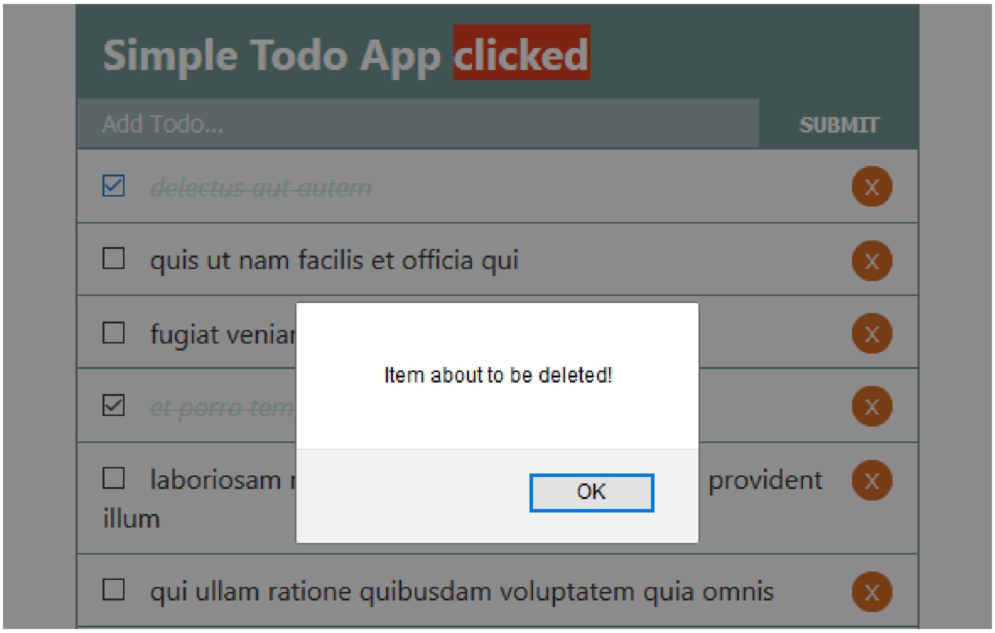
Go inside the *`TodoItem.js`* file and add this code above the *`render()`* method:

```
componentWillUnmount() {
  alert("Item about to be deleted!");
}
```

That's all.

Whenever any of the todos items is about to be deleted, this method will be called and executed.

Save your file and test your work.



Item about to be deleted!

Good job.

A quick recap. In this chapter, you've learned how to make an HTTP request to a server endpoint, how you can handle responses and finally, how you can use the React lifecycle methods in your application.

This brings us to the end of the first project.

You can find the source code for this chapter here:

https://github.com/Ibaslogic/http-request-lifecycle-methods

Before we get started with React Hooks, let's solidify our React fundamentals by creating another project.

Moving on to the second project!

# Chapter 5
# Project II (Building a Simple Meme Generator App)

The purpose of this project is to solidify and build on our already made React foundation. The way we will build this app will be a bit different compared to the Todos app. Here, you'll be given tasks to be accomplished. After which I will explain how the tasks were executed.

Are you ready to dive in? Then let's get started!

This is what we will build.

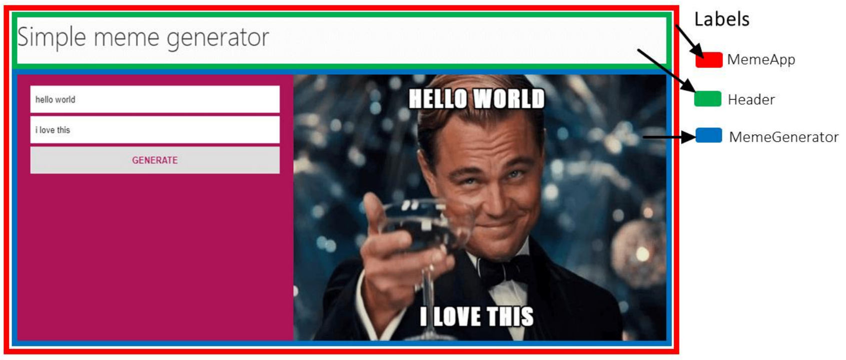


Fig: Simple Meme Generator

What this app does is simple.

The user inputs the *top* and *bottom* text through the input fields and then generate random images by clicking on the *Generate* button.

We will get a random image by making an HTTP request to an online database.

As expected, you need to wear your React cap by *thinking in React!*

So the first thing is to decompose your design UI into smaller units, each representing component.

As highlighted in the image, the parent component is called the *MemeApp* while the *Header* and *MemeGenerator* components are its children.

Since the *Header* simply renders a simple heading text, this should be a *functional component*. The *MemeGenerator* will not only accept inputs from the user (meaning, it will hold state), it will also take care of fetching data from an endpoint URL through a lifecycle class method known as `componentDidMount()`.

All these indicate that the *MemeGenerator* component will be a *class component*.

Enough said. Now you have the hints!

So start the first task.

*Project Tasks*

## Task – Create the primary files and render something on the screen

1. Start by creating a new React app called *meme-generator* by running this command:

```
C:\Users\Your Name> npx create-react-app meme-generator
```

*From the terminal, you can change directory to the Desktop or wherever you want to save your project and run the command.* The command will create a new folder called *meme-generator* inside your choosing directory.

2. Next, open the project folder with your code editor and start the development server using this command:

```
C:\Users\Your Name\meme-generator > npm start
```

*Remember, with VsCode, you can Toggle the terminal by pressing Ctrl + ` or Cmd + `.*

3. Delete all the files in the `src` folder so that you can start from scratch.

   This breaks your app immediately. Do not worry, we will fix it.

4. Create `index.js` file inside the `src` folder and get React to render something on the screen.

So the file will look like this:

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

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

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

5. Save the file and see the content displayed in the frontend.

Now, instead of rendering *`h1`* text, we will render a React component that will serve as the parent component.

6. So, create a folder called *`components`* in the *`src`* directory. Inside this folder, create a file called *`MemeApp.js`* and add the following code:

```
import React from "react";

const MemeApp = () => {
  return (
    <div>
      <h1>Hello from parent app</h1>
    </div>
  );
};

export default MemeApp;
```

7. Save the file and update the `index.js` file so you have:

```
import React from "react";
import ReactDOM from "react-dom";
import MemeApp from "./components/MemeApp";

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

8. Save and check your app.

## Explanation

So far, what we have done is pretty straight forward. In the code, we rendered a simple *functional component* called *MemeApp* that will serve as the parent component. If you are wondering, why *functional* and not a *class-based* component. This is simply because it will neither hold state nor any lifecycle methods.

Its duty here is just to hold the other two components – *Header* and *MemeGenerator*.

Again, if you are not sure initially what type of component to use, go with the *class component*. Later when things become clear, you can then adjust accordingly.

Moving on.

## Task – Add the children components to the MemeApp and display something

1. Create two new components called *Header* and *MemeGenerator* inside the `components` folder.
   The *Header* will only display a simple heading text while *MemeGenerator* will be making a call to an API and also accept the user's input. This information helps determine what type of component to create.

Your *`Header.js`* file should look like this:

```
import React from "react";

const Header = () => {
  return (
    <header>
      <h1>Simple meme generator</h1>
    </header>
  );
};

export default Header;
```

While the *`MemeGenerator.js`* file should look like this:

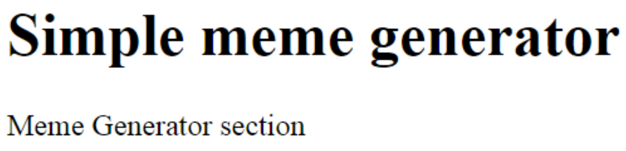
```
import React, { Component } from "react";

class MemeGenerator extends Component {
  render() {
    return <div>Meme Generator section</div>;
  }
}

export default MemeGenerator;
```

2. Next, import both files inside the parent component, *MemeApp*, and render them by calling their instances within the *`return`* statement.

Your code should look like this:

```
import React from "react";
import Header from "./Header";
import MemeGenerator from "./MemeGenerator";

const MemeApp = () => {
  return (
    <div>
      <Header />
      <MemeGenerator />
    </div>
  );
};

export default MemeApp;
```

3. Save the files and check the frontend. Your app should look like this:



Simple meme generator

Meme Generator section

For the Meme Generator section, we will need to display the *top text*, the *bottom text* and a *random image*.

And to get them, we will initialize them in the state.

To get random images, we will make use of a website called *imgflip* (https://api.imgflip.com/). We will make an API call to this endpoint URL, https://api.imgflip.com/get_memes that provides a bunch of new images.

Remember, in our last project, *Todos* app, we used a third-party library called *Axios* to make API calls. But here, we will use the native *Fetch API*. This exposes you to different ways of fetching data.

So follow the next task.

## Task – Add input fields to the MemeGenerator Component and display a default image

1. In the *`MemeGenerator.js`*, add this *`state`* object just above the *`render()`* method:

```
state = {
  topText: "",
  bottomText: "",
  randomImage: "https://i.imgflip.com/26am.jpg",
}
```

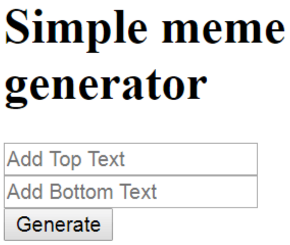
2. Then, create a *`form`* element within the *`render()`* method and add two *`input`* elements (one for the *top text* and the other for the *bottom text*) and a *`button`*. Remember that this will be a *"controlled form"*.

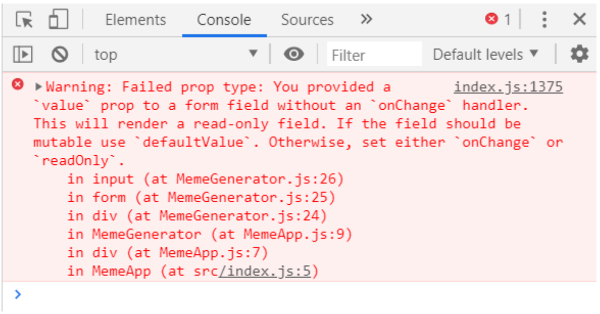
   So make sure to include all the attributes you'll need for this to work.

   Your render method should look like this:

```
render() {
  return (
    <div>
      <form>
        <input
          type="text"
          name="topText"
          placeholder="Add Top Text"
          value={this.state.topText}
        />
        <input
          type="text"
          name="bottomText"
          placeholder="Add Bottom Text"
          value={this.state.bottomText}
        />
        <button>Generate</button>
      </form>
    </div>
  );
}
```

3. Save the file and check the frontend.





```
Warning: Failed prop type: You provided a        index.js:1375
`value` prop to a form field without an `onChange` handler.
This will render a read-only field. If the field should be
mutable use `defaultValue`. Otherwise, set either `onChange` or
`readOnly`.
    in input (at MemeGenerator.js:26)
    in form (at MemeGenerator.js:25)
    in div (at MemeGenerator.js:24)
    in MemeGenerator (at MemeApp.js:9)
    in div (at MemeApp.js:7)
    in MemeApp (at src/index.js:5)
```

By assigning to each of the input fields, a value equal to the current value of the state, you've made the form *controllable*.

This therefore prompt React to display a console warning as seen in the image above telling you to provide an *onChange* handler.

React need this handler to keep track of any changes in the fields.

4. Update both `input` fields to include this:

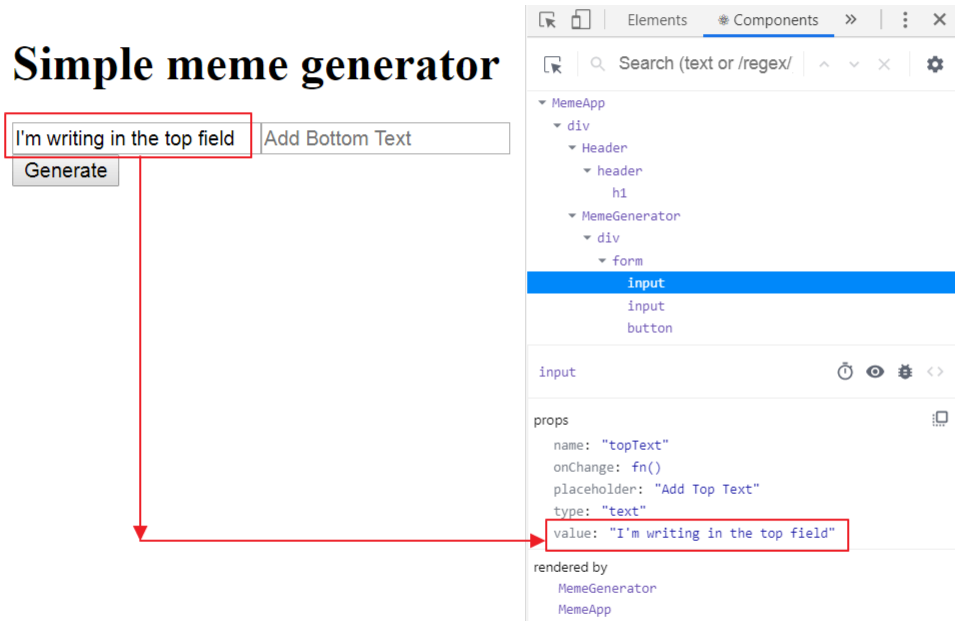
```
onChange={this.handleChange}
```

5. Then, add this above the *render()* method:

```
handleChange = e => {
  this.setState({
    [e.target.name]: e.target.value
  });
};
```

6. Save the file. Go back to the frontend and write something in the input fields. You will see the state in real-time if you open the React tools.

   See image below.



```
name: "topText"
onChange: fn()
placeholder: "Add Top Text"
type: "text"
value: "I'm writing in the top field"
```

Notice that the state is being updated for every keystroke.

7. Next, add the following code after the *form* element to display the text and the default meme image in the frontend:

```
<div>
  <img src={this.state.randomImage} alt="" />
  <h2>{this.state.topText}</h2>
  <h2>{this.state.bottomText}</h2>
</div>
```

8. Save the file and check the frontend.

   It works but quite ugly!

   Before we add styles to our app, let's quickly explain what we did!

## Explanation

Anytime you want to work with data that will later be updated, the first thing you would want to do is to initialize them in the state with a default value. In our case, we assigned empty strings to the value of the input fields and a default image URL for the meme.

In React, input fields should be *controlled*. Meaning they should be handled by the component state and not the DOM. And we do that by assigning to the value attribute, a value equal to the current value of the state.

We have said these over and over again.

In the *`setState()`* method, we are assigning to the target input, its current value through *`e.target.value`*. And as long as the name you have in the input field matches what you have in the state, this will always work for as many text input fields as you have.

Please revisit the *Todos* app if you need a refresher!

## *Adding Styles to the Meme Generator*

This is pretty straight forward. In the *`src`* folder, create a new file called *`App.css`* and add the following CSS styles:

```
* {
  box-sizing: border-box;
  margin: 0;
  padding: 0;
}
body {
  font-family: "Segoe UI", Arial, sans-serif;
  line-height: 1.4;
  background-color: #fcfcfc;
  color: #1f1f1f;
}
h1 {
  font-size: 40px;
  font-weight: 300;
  margin-bottom: 27px;
}
.meme-container {
  display: flex;
}
form {
  background: #ad1457;
  text-align: center;
  padding-top: 20px;
}
input,
button {
  width: 90%;
  height: 40px;
  margin-bottom: 5px;
  border: none;
  outline: none;
}
input {
  padding: 5px 8px;
}
button {
  cursor: pointer;
  text-transform: uppercase;
  color: #ad1457;
  font-weight: bold;
}
.meme {
  position: relative;
}
.meme > img {
  width: 100%;
  background: cover;
  margin-bottom: -5px;
  background-repeat: no-repeat;
}
.meme > h2 {
  position: absolute;
  max-width: 36em;
  font-size: 36px;
  text-align: center;
  left: 49%;
  transform: translateX(-50%);
  margin: 0.3em 0;
  padding: 0;
  font-family: impact, sans-serif;
  text-transform: uppercase;
  color: white;
  letter-spacing: 2px;
  text-shadow: 2px 2px 0 #000, -2px -2px 0 #000, 2px -2px 0
#000,
    -2px 2px 0 #000, 0 2px 0 #000, 2px 0 0 #000, 0 -2px 0
#000, -2px 0 0 #000,
    2px 2px 5px #000;
}
.meme > .top {
  top: 0.1em;
}
.meme > .bottom {
  bottom: 0.1em;
}

/* On screens that are 768px or less */
@media screen and (max-width: 768px) {
  h1 {
    font-size: 30px;
    margin-bottom: 14px;
  }
  .meme-container {
    flex-direction: column;
  }
  .meme > h2 {
    font-size: 30px;
  }
}
```

Save the file and import it in the *`index.js`* file using this line of code:

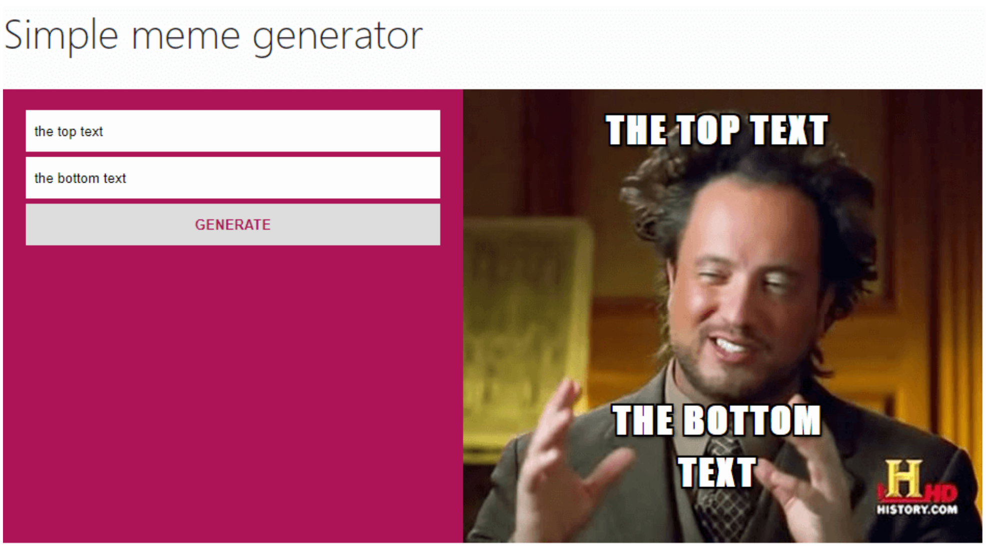
```
import "./App.css";
```

Save the file.

Next, update the *`render()`* method in the *`MemeGenerator.js`* file to include the CSS class names.

```
render() {
  return (
    <div className="meme-container">
      <form>
        <input
          type="text"
          name="topText"
          placeholder="Add Top Text"
          value={this.state.topText}
          onChange={this.handleChange}
        />
        <input
          type="text"
          name="bottomText"
          placeholder="Add Bottom Text"
          value={this.state.bottomText}
          onChange={this.handleChange}
        />
        <button>Generate</button>
      </form>
      <div className="meme">
        <img src={this.state.randomImage} alt="" />
        <h2 className="top">{this.state.topText}</h2>
        <h2 className="bottom">{this.state.bottomText}</h2>
      </div>
    </div>
  );
}
```

Save the file and check the frontend.



This looks good!

So what next?

We want a situation whereby we will display random images whenever the *GENERATE* button is clicked. To do this, we will first make an HTTP request to fetch the images from an API.

And we know that the best place to do that is in the `componentDidMount()` method.

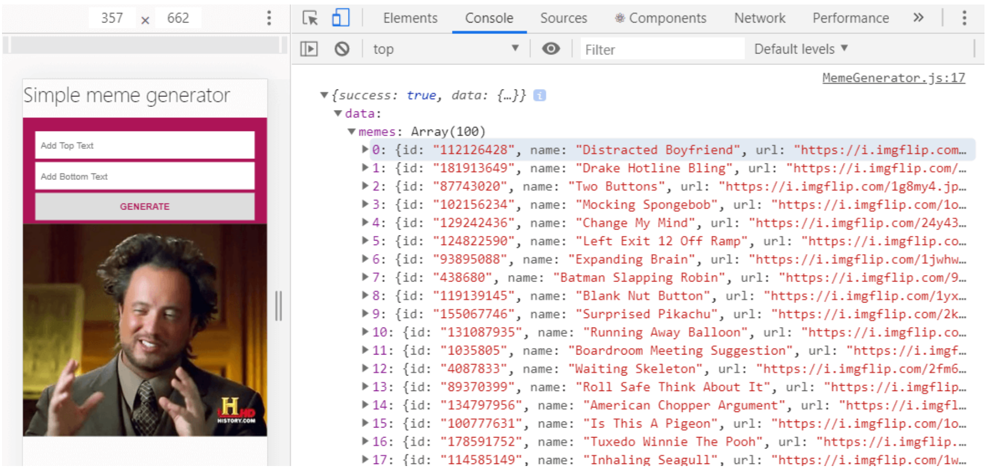
So let's follow the next task.

## Task – Request image data by making an API call

1. Add the following code above the *render()* method in the *MemeGenerator.js* file:

```
componentDidMount() {
  fetch("https://api.imgflip.com/get_memes")
    .then(response => response.json())
    .then(response => console.log(response));
}
```

2. Save the file and check the *Console* for the returned data.



```
                                                        MemeGenerator.js:17
▼{success: true, data: {…}}
  ▼data:
    ▼memes: Array(100)
      ▶0: {id: "112126428", name: "Distracted Boyfriend", url: "https://i.imgflip.com…
      ▶1: {id: "181913649", name: "Drake Hotline Bling", url: "https://i.imgflip.com/…
      ▶2: {id: "87743020", name: "Two Buttons", url: "https://i.imgflip.com/1g8my4.jp…
      ▶3: {id: "102156234", name: "Mocking Spongebob", url: "https://i.imgflip.com/1o…
      ▶4: {id: "129242436", name: "Change My Mind", url: "https://i.imgflip.com/24y43…
      ▶5: {id: "124822590", name: "Left Exit 12 Off Ramp", url: "https://i.imgflip.co…
      ▶6: {id: "93895088", name: "Expanding Brain", url: "https://i.imgflip.com/1jwhw…
      ▶7: {id: "438680", name: "Batman Slapping Robin", url: "https://i.imgflip.com/9…
      ▶8: {id: "119139145", name: "Blank Nut Button", url: "https://i.imgflip.com/1yx…
      ▶9: {id: "155067746", name: "Surprised Pikachu", url: "https://i.imgflip.com/2k…
      ▶10: {id: "131087935", name: "Running Away Balloon", url: "https://i.imgflip.co…
      ▶11: {id: "1035805", name: "Boardroom Meeting Suggestion", url: "https://i.imgf…
      ▶12: {id: "4087833", name: "Waiting Skeleton", url: "https://i.imgflip.com/2fm6…
      ▶13: {id: "89370399", name: "Roll Safe Think About It", url: "https://i.imgflip…
      ▶14: {id: "134797956", name: "American Chopper Argument", url: "https://i.imgfl…
      ▶15: {id: "100777631", name: "Is This A Pigeon", url: "https://i.imgflip.com/1o…
      ▶16: {id: "178591752", name: "Tuxedo Winnie The Pooh", url: "https://i.imgflip.…
      ▶17: {id: "114585149", name: "Inhaling Seagull", url: "https://i.imgflip.com/1w…
```

As seen in the image, we were able to get data from the URL endpoint we specified in the `fetch()` API method. This is similar to how we requested for data using the *Axios* library. But with *Fetch API*, you'll need to first resolve the promise to the JSON format using the `.then()` method before you can receive the data in the format that you can work with.

What we need from the response is the memes data which is an array (see the image above). If you take a careful look at the response in console, this data is located in *response.data.memes*.

To use this data, we need to save it.

But first, we will initialize it as an empty array in the state.

Update the state so it looks like so:

```
state = {
  topText: "",
  bottomText: "",
  randomImage: "https://i.imgflip.com/26am.jpg",
  allMemeImgs: []
};
```

After that, update the *`componentDidMount()`* method so you have:

```
componentDidMount() {
  fetch("https://api.imgflip.com/get_memes")
    .then(response => response.json())
    .then(response =>
      this.setState({
        allMemeImgs: response.data.memes
      })
    );
}
```

At this point, we have all the images information saved in the *`allMemeImgs`*.

Finally, let's see how we can grab and display a random image from the *`allMemeImgs`* array whenever we click on the *GENERATE* button.

To display images, all we need is the image URL. And if you take a look again at the last screenshot, the URLs are located in *`response.data.memes[i].url`*. Where **`i`** is the array index.

And from what we have in the *`state/setState`*,

you'll agree with me that –

*`response.data.memes[i].url == this.state.allMemeImgs[i].url`*

So let's do this.

## Task – Generate random image on form submission

1. Update the *<form>* opening tag to include *onSubmit={this.handleSubmit}*.

```
<form onSubmit={this.handleSubmit}>
```

2. Then, add the following code above the *render()* method:

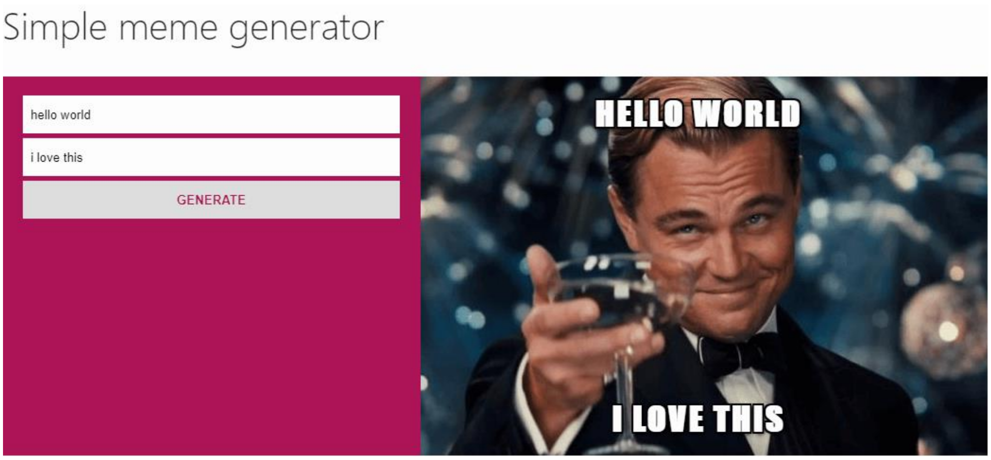
```
handleSubmit = e => {
  e.preventDefault();
  const randNum = Math.floor(Math.random() *
this.state.allMemeImgs.length);
  const randMemeImgUrl = this.state.allMemeImgs[randNum].url;
  this.setState({ randomImage: randMemeImgUrl });
};
```

3. Save the file.

Now you should have your app working as expected!

Go ahead and add both *top* and *bottom* text and click on the *GENERATE* button. This will keep generating a random image for your meme.

Enjoy!



Congratulations!

## Explanation

In the *handleSubmit* method, all we are doing is grabbing a random image URL and then updating the *src* attribute of the component *<img />* tag.

Notice how we are getting the random number (between 1 and the number of elements in the *allMemeImgs* array) and passing it as the array index to grab a random image URL.

Next, publish your beautiful app on the web and share with the world.

You can revisit the *Todos* app and replicate the steps here.

This brings us to the end of the first part of this course.

Congratulations on getting to this point.

Now, you should have the boldness and confidence to develop a modern React application from scratch. Also, you now have the tools to follow a Gatsby site project or any framework that is based on React!

In the next chapter, you will get started with React Hooks. There, you'll learn how to manage the state and lifecycle logic in a functional component.

It promises to be fun!

# Chapter 6
# Getting Started with React Hooks

Imagine having to switch your React component from a function to a class-based simply because you want to manage a state and/or lifecycle logic.

With React Hooks, that's a thing of the past.

The functional component is more than just being a presentational component. It can now manage a state and the class-based lifecycle logic.

The benefit here is that you'll be able to write a more readable, concise and clear code. You'll also have one way of creating a component.

In this chapter, you will learn how to get started with React Hooks practically. We will be working with our todos and meme generator project where the stateful logic and lifecycle methods are being managed by the class component. Our task now is to switch from managing this logic from the class component into a function-based component.

This way, you'll not only learn the fundamentals, but you'll also learn how it applies in a real project.

Let's dive in.

## *What are React Hooks?*

React Hooks (introduced in React since version 16.8) are JavaScript functions that allow us to build our React component ONLY with a function component.

React comes bundled with a few Hooks that allow us to manage most of the use cases of the class logic. It also allows us to create custom Hooks whenever we want to reuse component logic.

Here, we will explore the common use cases of built-in Hooks.

To get started, let's get our project files ready. We will be working with our todos application.

You'd want to clone or duplicate the project so that you can have the original version of the code in place.

Let's go ahead and clone it using this command:

```
git clone https://github.com/Ibaslogic/todos-app-hooks-starter
```

Once the download is done, don't forget to run *npm install* after you change directory (cd) inside the project folder.

Now, if you open the project, You'll notice a slight change in the `TodoApp.js` file compared to our original todos app code.

In the just downloaded project, we are only fetching data from the remote endpoint. We are not making use of the HTTP POST and DELETE methods. This way, we don't have to rely on the internet to post or delete items.

This gives us a smooth ride writing our code.

This is optional though!

Anyway, let's get started.

At the moment, our components are class-based.

To optimize the code using the React Hooks, let's start with the component where only the state logic (and not lifecycle logic) is being managed.

So let's take a look at `src/components/AddTodo.js` file.

Presently, it has a *`state`* object (where we assign a default empty string to the *`title`* property) and class methods at the top level of the component.

Let's start by commenting out all the code.

Then add this starting code at the top to avoid page break:

```
import React from "react"

const AddTodo = () => {
  return <div></div>
}

export default AddTodo
```

This is the first conversion. Notice we are now using a function instead of class.

## *Using the React Hooks useState*

To add state in a function component, React provides us with a Hook called *`useState`*.

If you revisit the class component, the data defined in the *`state`* object is accessed using *`this.state`*. It is as well updated using *`this.setState`* method.

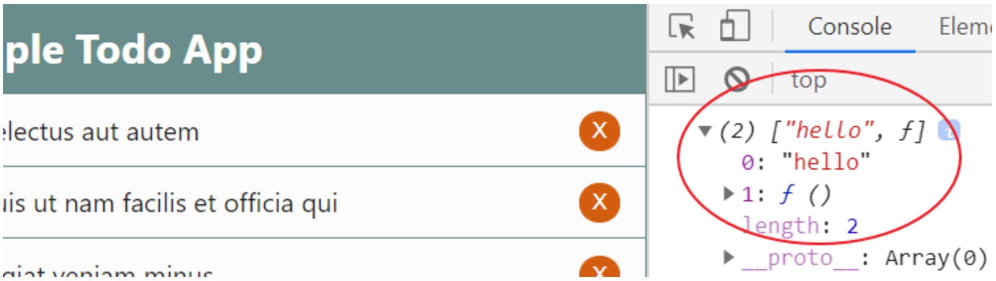
Now, let's replicate that in a function component.

First, import the *useState* Hook from the react module like so:

```
import React, { useState } from "react";
const AddTodo = () => {
  console.log(useState("hello"));
  return <div></div>;
};

export default AddTodo;
```

Notice we are logging the Hook to see what we have in return.

Save the file. Start your development server and open the console of your browser DevTools.



ılectus aut autem

ıis ut nam facilis et officia qui

```
▼ (2) ["hello", ƒ]
    0: "hello"
  ▶ 1: ƒ ()
    length: 2
  ▶ __proto__: Array(0)
```

As seen in the image, the *useState* Hook returns an array which ALWAYS contains two items. The first item is the current value passed-in (in our case, **hello**), and the second is a function that will allow us to update the value.

We can get these items from the array using the JavaScript array destructuring.

For instance,

```
const [title, setTitle] = useState("hello")
```

Here, we declared a state variable called *title* (which holds the current state i.e **hello**) and a function called *setTitle* to update the state.

This is similar to *this.state.title* and *this.setState* in our class component.

Unlike the class component, the state doesn't have to be an object. It can hold an array, number or string (as seen above).

Also, note that you are not limited to one state property as in the case of class component. Here, you can define multiple states.

You'll see how this works later in this book.

*But keep in mind, it's good to keep related data together.*

**Now that you have some basic understanding, let's take a look at the rules to use these Hooks.**

All you have to keep in mind is that you ONLY call Hooks at the top level of your function component or from custom Hooks. Not inside a loop, condition or regular function.

This ensures that all your component logic is visible to React.

Back to our code, let's update the component so you have:

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

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

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

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

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

export default AddTodo;
```

Save the file. You should see the input fields back in the frontend.

Test it and it should work perfectly.

**What's happening in the code?**

If you revisit the class version, we declared a *`state`* object where we assigned a key-value pair. But now, we are doing this using the *`useState`* React Hook.

Here, instead of using *`this.state`* to access the current state value, we simply use the variable, *`title`*. Likewise, we are now updating the state using the second element returned by the *`useState`*.

As seen in the *`onChange`* and *`onSubmit`* function, we are using the *`setTitle`* instead of *`this.setState`* used in the class component.

**Note:** ***`this`*** keyword in a class component does not exist in a function component.

This also applies to the methods in the class component (*`onChange`* and *`onSubmit`*). Remember we cannot use class methods in a function but we can define functions in a function.

So all we did here was to convert the class methods to function by adding the *`const`* keyword to them. With this simple change, you can call the function within the JSX without using ***`this`*** keyword.

Another area of concern is the *`onChange`* method. This method is called whenever the input text field changes.

If you are vigilant, you'd ask yourself why we are not using the *`e.target.name`* in the *`onChange`* method as we have it in the class version. And if you follow this book from the beginning, you'd know that this target allows us to manage many input fields with a single method/function as the case may be.

Now read carefully.

In our code, we are assigning a string to the *`title`* state variable through the *`useState`*. This is the simplest use case of the Hook.

With this setup, you can only manage an input field in a function call. If you add more fields, you'll need to define a separate *`useState`* Hook and a function to manage it.

While this is fine, it is better to group related data.

Just like the class version of the code, we will write our code in a way that we can manage as many input fields with a function.

Let's update the *AddTodo* component so you have:

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

const AddTodo = (props) => {
  const [inputText, setInputText] = useState({
    title: "",
  });

  const onChange = (e) => {
    setInputText({
      ...inputText,
      [e.target.name]: e.target.value,
    });
  };

  const onSubmit = (e) => {
    e.preventDefault();
    props.addTodo(inputText.title);
    setInputText({
      title: "",
    });
  };

  return (
    <form className="form-container" onSubmit={onSubmit}>
```

```
      <input
        type="text"
        placeholder="Add Todo..."
        className="input-text"
        name="title"
        value={inputText.title}
        onChange={onChange}
      />
      <input type="submit" value="Submit" className="input
-submit" />
    </form>
  );
};

export default AddTodo;
```

Save your file and test your work.

Now, you can manage as many input fields in your app with a single function (In our case, the *`onChange`* function). All you have to do is to add another property alongside the *`title`* in the *`useState`* and then assign the property name to the *`name`* prop in the *`input`* element.

So what changes?

First, anytime you are grouping related data in an object as in the case of the *`inputText`* state variable, the state returned by the *`useState`* Hook is not merged with that of the update passed to it.

Meaning it doesn't merge the old and new state. Instead, it overrides the entire state with that of the current.

The way out is to manually merge them by passing the entire state using the spread operator (the three dots before *`inputText`*) and override the part of it.

If you don't feel comfortable grouping related data like this, then you can split them into different *`useState`*. But don't forget, you'd need separate functions to manage them.

Hope it's clear?

Now that you've learned about managing the state in a function component using the React built-in *`useState`* Hook, let's see how we can replicate the lifecycle logic in a function component.

## *Using the React Hooks useEffect*

For now, our focus will be on the *`src/components/TodoApp.js`* file. This file manages a lifecycle method called the *`componentDidmount()`*.

Let's replicate its logic in a functional component. I believe you can convert the state logic in this component to use the *`useState`* Hook.

Well, let's start with that.

As expected, comment-out all the code in this file and add the following at the top.

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

const TodoApp = (props) => {
  const [todos, setTodos] = useState([]);
  const [show, setShow] = useState(false);

  const handleChange = (id) => {
    setTodos(
      todos.map((todo) => {
        if (todo.id === id) {
          todo.completed = !todo.completed;
        }
        return todo;
      })
    );
    setShow(!show);
  };

  const deleteTodo = (id) => {
    setTodos([
      ...todos.filter((todo) => {
        return todo.id !== id;
      }),
    ]);
  };

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

  return (
    <div className="container">
      <Header headerSpan={show} />
      <AddTodo addTodo={addTodo} />
      <Todos
        todos={todos}
        handleChange={handleChange}
        deleteTodo={deleteTodo}
      />
    </div>
  );
};

export default TodoApp;
```

Save your file and test your application.

Notice we are not including the lifecycle logic yet, hence no data is being fetched. We will take care of that in a moment.

So what is happening?

In the code, we started by defining a separate *`useState`* Hook for the state variables and assigning a default value to them.

Now, comparing the entire code to that of the class version, you'll notice that we removed all occurrence of *`this.state`* since it doesn't apply in the function component.

Likewise, the *`setTodos`* and *`setShow`* function which are used to update the state value replaces their respective *`this.setState`*.

That out of the way,

If you take a look at the class version of our code, we are fetching the default todos data using the HTTP GET method in the *`componentDidMount`* lifecycle method.

But in a function component, we cannot use this method. Instead, we will make use of another Hook called *useEffect*.

As the name implies, it is used to perform side effects. An example is a data we fetch via an HTTP request.

React allows us to combine different lifecycle logic using this single Hook. So you can think of *useEffect* Hook as *componentDidMount*, *componentDidUpdate*, and *componentWillUnmount* combined.

Though, just like the *useState* Hook, you can also have multiple *useEffect* to separate unrelated logic.

Let's see how to apply this Hook.

In the *src/components/TodoApp.js* file, import the *useEffect* Hook from the *react* module. So your import should look like this:

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

Then add this Hook above the *return* statement and save your file:

```
useEffect(() => {
  console.log("test run");
});
```

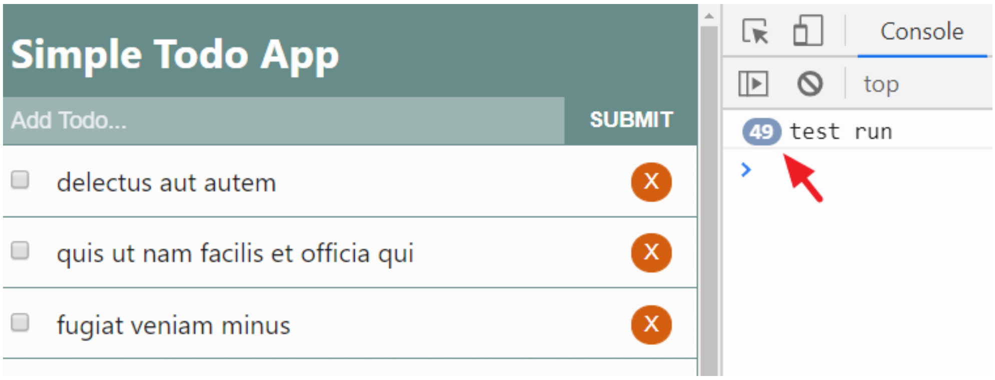
With this simple addition, if you reload the frontend, you should see the log message displayed in the browser console.

This Hook takes in a function as an argument and an optional array (I omitted that for now). The function defines the side effect to run (in our case, making an HTTP request) and the optional array will define when to re-run the effect.

Now, let's update this Hook to include our HTTP request.

```
useEffect(() => {
  console.log("test run")
  axios.get("https://jsonplaceholder.typicode.com/todos?_l
imit=10")
    .then(response => setTodos(response.data))
})
```

If you save the file and take a look at the console once again, you will see that your log keeps incrementing. This shows the Hook is running infinitely.



```
49 test run
```

What's happening?

Unlike *`componentDidMount`* lifecycle that only runs once it fetches data for the first time, the *`useEffect`* Hook by default runs not only after the first render but also after every update – i.e when there is a prop or state changes.

In our code, a change occurs. The *`todos`* state variable is being updated through the *`setTodos`* function when the data is fetched from the endpoint. Thereby causing an infinite loop.

*This is happening because the Hook combines different lifecycle logic. It is our responsibility to control it to the logic we want.*

How can we control it?

That's where the optional array of dependencies comes in.

```
useEffect(() => {
  ...
}, []);
```

By specifying this array, we can control whether or not the effect should re-run. If we pass-in variable(s), the effect will only re-run if its value(s) changes between re-renders. Else, it will skip applying the effect.

But if we pass an empty array, React will only execute the effect once because no data is changing.

Taking a closer look at this, we have the equivalent of *`componentDidMount`* when the array is empty and *`componentDidUpdate`* when it includes variable(s) that will trigger re-rendering.

Update the Hook to include the optional array:

```
useEffect(() => {
  console.log("test run");
  axios.get("https://jsonplaceholder.typicode.com/todos?_l
imit=10")
    .then((response) => setTodos(response.data));
}, []);
```

Save the file and test your application.

It should work as expected.

Next, let's see how the *`useEffect`* handles the logic of the *`componentDidUpdate`* and *`componentWillUnmount`*.

Starting with the *`componentDidUpdate`*,

Remember that component gets updated when there is/are state or prop changes, thereby trigger re-rendering.

If you take a look at the `src/components/Layout/Header.js` file, we are using this lifecycle method to update the DOM whenever there is prop change. This happens every time the checkbox is clicked.

To apply this logic using the Hook,

Let's start by converting the component to a function-based.

```
import React from "react";

const Header = (props) => {
  const headerStyle = {
    backgroundColor: "#678c89",
    color: "#fff",
    padding: "10px 15px",
  };
  return (
    <header style={headerStyle}>
      <h1 style={{ fontSize: "25px", lineHeight: "1.447em"
, margin: "0px" }}>
        Simple Todo App <span id="inH1"></span>
      </h1>
    </header>
  );
};

export default Header;
```

At this point, we don't have the lifecycle logic in place.

Let's do that now.

Import the *`useEffect`* from the react module like so:

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

Then add this Hook in your *Header* component (above the *`return`* statement):

```
useEffect(() => {
  var x = Math.floor(Math.random() * 256)
  var y = Math.floor(Math.random() * 256)
  var z = Math.floor(Math.random() * 256)
  var bgColor = "rgb(" + x + "," + y + "," + z + ")"

  document.getElementById("inH1").innerHTML = "clicked"
  document.getElementById("inH1").style.backgroundColor =
bgColor
}, [props.headerSpan])
```

Save your file and check your application.

Oops! The heading text, “clicked” is displaying on the initial render – without the checkbox being clicked.

What’s happening?

As mentioned earlier, the Hook is designed to run not only when the component first renders but also on every update. That is why the call to manipulate the DOM as defined in it is being executed on the initial rendering.

Once it renders for the first time, it checks for an update in the dependency to run subsequently.

Remember, this dependency gets updated whenever you click the checkbox.

While this is the common use case of this lifecycle logic using the Hook, sometimes, we want the Hook to run only on updates and right after any user action. In our case, whenever the user clicks on the checkbox.

## *Running an Effect only on Updates*

If you revisit the class version of our code, we are checking for update (i.e if a checkbox is clicked) by comparing the *`prevProps`* and the current prop.

With React Hooks, we can get the previous props or state as the case may be using the *`useRef()`* Hook.

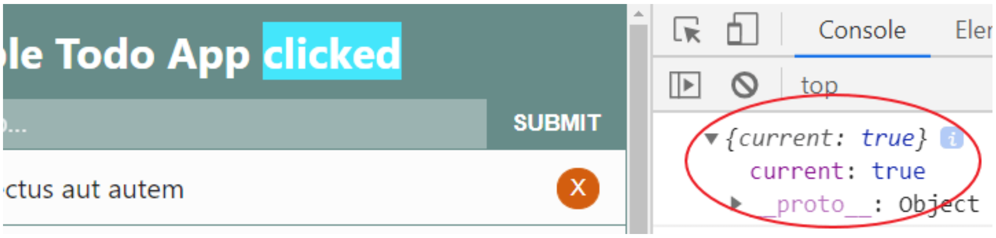
For instance, add this above the *`useEffect`* Hook:

```
const isInitialMount = useRef(true)
```

Then, log the *`isInitialMount`* variable to the console. Make sure you import *`useRef`* from the *`react`* module.

```
import React, { useEffect, useRef } from "react";
const Header = props => {
  const headerStyle = {
    ...
  };

  const isInitialMount = useRef(true);
  console.log(isInitialMount);
  useEffect(() => {
    ...
  }, [props.headerSpan]);
  return (
    ...
  );
};
export default Header;
```

If you save your file and check the console, you should see this:



```
▼{current: true}
   current: true
 ▶ __proto__: Object
```

The *useRef* Hook returns an object containing the *current* property. This property holds the value we passed to the Hook.

This is good because we can track whether we are on the first render or subsequent render.

Next, let’s update the *`useEffect`* Hook so you have:

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

const Header = props => {
  const headerStyle = {
    ...
  };

  const isInitialMount = useRef(true);

  console.log(isInitialMount);

  useEffect(() => {
    var x = Math.floor(Math.random() * 256);
    var y = Math.floor(Math.random() * 256);
    var z = Math.floor(Math.random() * 256);
    var bgColor = "rgb(" + x + "," + y + "," + z + ")";

    if (isInitialMount.current) {
      isInitialMount.current = false;
    } else {
      document.getElementById("inH1").innerHTML = "clicked
";
      document.getElementById("inH1").style.backgroundColo
r = bgColor;
    }
  }, [props.headerSpan]);

  return (
    ...
  );
};

export default Header;
```

Save your file and test your application.

It should work as expected.

What’s happening in the code?

In the `useEffect` Hook, we are checking if the `current` property of the `useRef` is true.

By default, we set the value to `true` to track when the component has just mounted. When this happens, we ignore any action and immediately set the value to `false`.

At this point we know we can do whatever we want. In our case, we can perform DOM manipulation right after a clicked checkbox.

Moving on.

Next, the *`componentWillUnmount`* logic.

Here, our focus is on the *`src/components/TodoItem.js`* file.

Normally, we do cleanups (for instance, cancelling the network request, removing event listeners) in the *`componentWillUnmount`*. This is because it is invoked just before a component is unmounted and destroyed.

But in our app, we are using this lifecycle logic to trigger an alert when an item is about to be deleted from the todos list.

Now, how can we replicate the same logic with Hooks?

While you are aware that the *`useEffect`* Hook run on every render (except you control it), React allows us to clean up effects from the previous render before running another cycle and also before the component is unmounted.

Well, let's see this in action.

As expected, we will convert the *TodoItem* class component to a function-based.

This should be straight forward.

Here you have it:

```
import React from "react";

const TodoItem = (props) => {
  const completedStyle = {
    fontStyle: "italic",
    color: "#c5e2d2",
    textDecoration: "line-through",
  };

  const { completed, id, title } = props.todo;

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

export default TodoItem;
```

Save the file.

Now let’s apply the Unmount logic.

In the same file, import the *`useEffect`* like so:

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

Then, add the following code above the *`return`* statement.

```
useEffect(() => {
  return () => {
    alert("Item about to be deleted!");
  };
}, []);
```

Save your file and try to delete todos items in your application.

The code is pretty simple. Anytime you return a function inside the *`useEffect`* Hook, it will execute before the Hook run the next time (in case you are triggering a re-run) and also before the component is unmounted.

In our case, we don't have any array dependency. So, the effect will run just once and the *`return`* function will be called when the component is about to unmount.

At this point, you have total control over the type of component to create.

Now, the logic of our todos app is managed in the functional component using the React Hooks. Though, we still have a component constructed with class in the *`src/components/Todos.js`* file.

Mind you, this component has no state or lifecycle logic. This makes the conversion easy and direct.

Can you give it a try?

Good!

Here is the conversion.

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

const Todos = (props) => {
  return (
    <div>
      {props.todos.map((todo) => (
        <TodoItem
          key={todo.id}
          todo={todo}
          handleChange={props.handleChange}
          deleteTodo={props.deleteTodo}
        />
      ))}
    </div>
  );
};

export default Todos;
```

Now, we have a complete React app written only with a function component.

Thanks to the Hooks.

While you have learned a lot and covered most of the use cases of the React Hooks, there are more to learn like creating a custom Hook for logic reusability.

But this is a great start! You can start using these Hooks in your new and existing project.

Please note, you don't have to rewrite your existing logic, but you can start applying these Hooks to new updates.

That's it for this project.

You can find the source code at:

https://github.com/Ibaslogic/todos-react-hooks

Do you still want to have fun?

Move on to the Meme Generator Hooks version.

# Chapter 7
# Meme Generator with React Hooks

In the previous chapter, we discussed the fundamentals of React Hooks and also apply them in our Todos app project.

Here, we aim to solidify our React Hooks foundation by working on our simple meme generator.

If you are ready, let's dive in.

As you are aware, the simple meme generator is built on the class logic (state and lifecycle method). We will, however, manage this logic using the React Hooks in a function component.

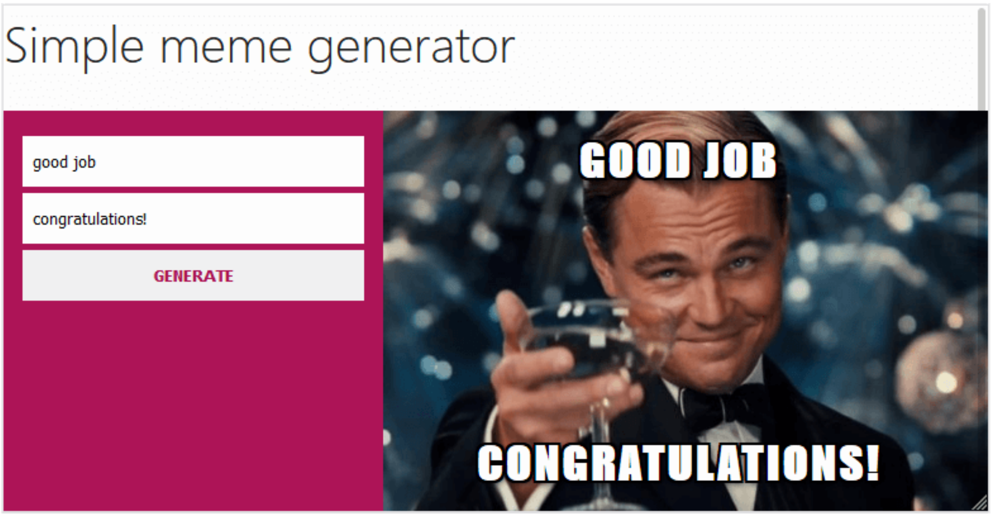
So get your project folder ready. Or go ahead and clone it using this command.

```
git clone https://github.com/Ibaslogic/memegenerator
```

Once the download is done, change directory (cd) inside the project folder and run *npm install*. This creates a `node_module` folder consisting of the necessary dependencies.

Finally, start your server with *npm start*.

You should see this app at http://localhost:3000/



Let's quickly reiterate what this app does.

The user inputs the top and bottom text through the input fields and then generate random images by clicking on the GENERATE button.

As expected, you should know that the files that make up this UI live in the `src` folder. If you take a look inside the `src/components` folder, we have three files.

Both the `Header.js` and MemeApp.js are already a function component. The `MemeGenerator.js` manages the state and a lifecycle method, hence constructed using a class component.

Let's optimize the component to use the React Hooks. So, open the `src/components/MemeGenerator.js` file.

Presently, it has a `state` object (consisting of four different properties with default values) and class methods including a lifecycle (`componentDidMount`) at the top level of the component.

You can start by commenting out all the code.

Then add this starting code at the top to avoid page break:

```
import React from "react"

const MemeGenerator = () => {
  return <div></div>
}

export default MemeGenerator
```

This is our first conversion. As seen, we are now using a function to construct our component instead of class.

Next, let’s manage the state in this function component.

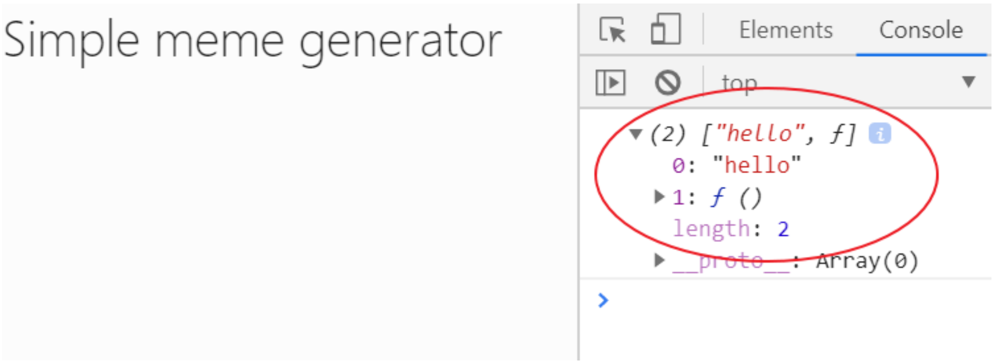
So, import the *`useState`* Hook from the react module like so:

```
import React, { useState } from "react";
const MemeGenerator = () => {
  console.log(useState("hello"));
  return <div></div>;
};

export default MemeGenerator;
```

Save the file.

Notice that we are logging the *`useState`* Hook to see what we have in return.

You should know that the Hook will return an array containing two items. The current state and a function to update the state.



```
▼(2) ["hello", ƒ]
   0: "hello"
  ▶1: ƒ ()
   length: 2
  ▶__proto__: Array(0)
>
```

Recall, we can destructure the returned value of this Hook to have access to its initial state (i.e default value) as well as a callback to modify it.

Something like this.

```
const [topText, setTopText] = useState("hello")
```

Where `topText` holds the initial state i.e **hello** and the `setTopText` is a function to update the state.

If you apply this in the *MemeGenerator* component, you should have:

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

const MemeGenerator = () => {
  const [inputText, setInputText] = useState({
    topText: "",
    bottomText: "",
  })
  const [randomImage, setRandomImage] = useState(
    "https://i.imgflip.com/26am.jpg"
  )
  const [allMemeImgs, setAllMemeImgs] = useState([])

  const handleChange = e => {
    setInputText({
      ...inputText,
      [e.target.name]: e.target.value,
    })
  }

  const handleSubmit = e => {
    e.preventDefault()
    console.log("submitted")
  }

  return (
    <div className="meme-container">
      <form onSubmit={handleSubmit}>
        <input
          type="text"
          name="topText"
          placeholder="Add Top Text"
          value={inputText.topText}
          onChange={handleChange}
        />
        <input
          type="text"
          name="bottomText"
          placeholder="Add Bottom Text"
          value={inputText.bottomText}
          onChange={handleChange}
        />
        <button>Generate</button>
      </form>
      <div className="meme">
        <img src={randomImage} alt="" />
        <h2 className="top">{inputText.topText}</h2>
        <h2 className="bottom">{inputText.bottomText}</h2>
      </div>
    </div>
  )
}

export default MemeGenerator
```

Save the file. You should see your app rendered in the frontend.

For now, if you click to generate new images, you’ll get a "submitted" text in the console of your browser DevTools.

What's happening in the code?

Remember, in the class version, we declared a *`state`* object where we assigned default values to the object properties. But now, we are doing this using the *`useState`* React Hook.

Here, instead of defining all the properties in the state object, we have options to split them into multiple state Hook.

But take note, we are keeping related data (the *`topText`* and *`bottomText`*) together.

Now, we don't have to use *`this.state`* to access the current state value. Instead, we are using their respective variables. Likewise, we are now updating the state using the second element returned by the *`useState`* Hook.

Also, take note of what's happening in the *`handleChange`*. This function is called whenever the input text field changes.

You should know that anytime you are grouping related data in an object, the *`useState`* Hook doesn't merge the old and new state. Instead, it overrides the entire state with that of the current.

To get a clearer picture,

For the meantime, comment-out the …`inputText` from the function so you have:

```
const handleChange = e => {
  setInputText({
    // ...inputText,
    [e.target.name]: e.target.value,
  })
}
```

Save your file and try to input the top and bottom text in your application.

You'll notice that they are overriding each other.

So we merged them by passing the entire state using the spread operator (the three dots before `inputText`) and override the part of it.

Please remember to uncomment the …`inputText` in the function.

Sometimes, manually merging the state might be cumbersome.

So an alternative is to split the `topText` and `bottomText` into different `useState` Hook. But in this case, you will need separate functions to manage the changes in these inputs.

This is not going to scale especially if you want to manage many input fields in your application.

Hope it's clear?

Ok. Let's move on.

## *The useEffect React Hook*

At the moment, if we click the *GENERATE* button in our app to display a random image, nothing will happen except that we are logging a simple text in the console.

If you take a look at the class version of the code, we are fetching these images using the fetch API in the `componentDidMount` lifecycle method.

And as you know, this method is not available in a function component.

So we will use another Hook called *useEffect*.

We have covered much of this in the last chapter, but to reemphasise, you can think of this Hook as *componentDidMount*, *componentDidUpdate*, and *componentWillUnmount* combined.

Let's see how to apply it in our application.

Back in your *src/components/MemeGenerator.js* file, import *useEffect* Hook from the *react* module.

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

Then add this Hook above the *return* statement and save your file:

```
useEffect(() => {
  console.log("test run")
  fetch("https://api.imgflip.com/get_memes")
    .then(response => response.json())
    .then(response => setAllMemeImgs(response.data.memes))
})
```

If you visit the console, you'll see that this Hook is running infinitely.

What's happening?

As you can see, this Hook accepts a function that defines the side effect to run. In our case, making HTTP request.

By default, it runs not only after the first render but also after every update – i.e when there is a prop or state changes.

In our code, a change occurs. The *setAllMemeImgs* function updates its state variable when the data is fetched from the endpoint. Thereby causing the Hook to run continuously.

So, we have the responsibility to control it to the logic we want. Remember, we want to replicate the logic of the *componentDidMount* which should only run once it fetches data for the first time.

To do that, we will add an optional array of dependencies to the Hook so it looks like this:

```
useEffect(() => {
  console.log("test run")
  fetch("https://api.imgflip.com/get_memes")
```

```
    .then(response => response.json())
    .then(response => setAllMemeImgs(response.data.memes))
}, [])
```

With this simple addition, the Hook now depends on the array of dependencies to re-run.

But in case there are no dependencies in the array as seen above, React will only execute the Hook once because no data is changing.

With this, you are safe to say that you have the equivalent of *`componentDidMount`* when you have an empty array as the second parameter.

Now, we can update the *`handleSubmit`* function.

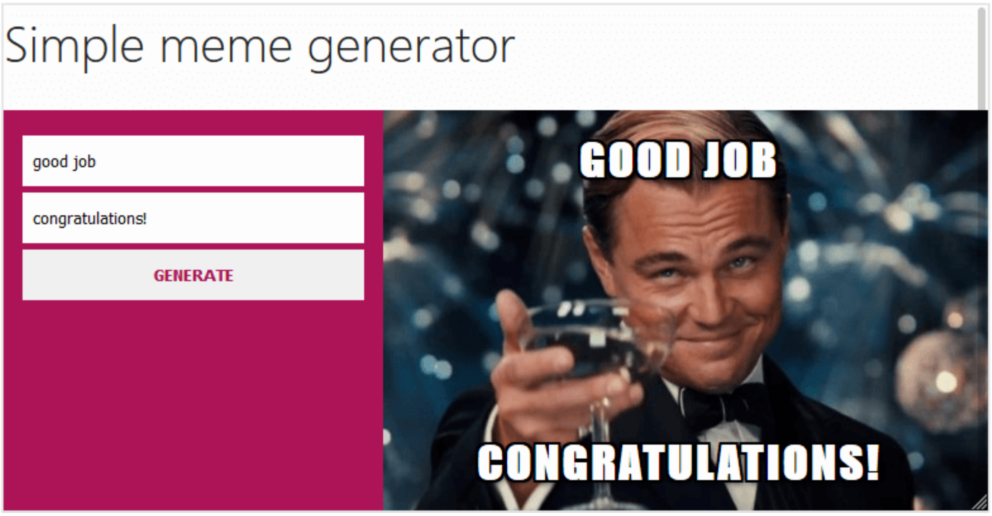
```
const handleSubmit = e => {
  e.preventDefault()
  const randNum = Math.floor(Math.random() * allMemeImgs.l
ength)
  const randMemeImgUrl = allMemeImgs[randNum].url
  setRandomImage(randMemeImgUrl)
}
```

Save the file.

Comparing this code to that of the class version, you'll notice that we've removed all the occurrence of *`this.state`* since it doesn't apply in the function component. Likewise, the *`setRandomImage`* function which update its state value replaces *`this.setState`*.

Now check your app and test it.

It should work as expected.



Good job!

This is a quick one. I hope you had fun learning React and the Hooks. Now, all you need is to practice as much as possible.

Also, you should be able to work with any framework based on React.

So my friend, if you like this React book, consider writing a short review.

Find the Meme Generator source code here:

https://github.com/Ibaslogic/React-Hooks/blob/master/src/components/MemeGenerator.js

# CONNECT WITH IBAS

I appreciate your time for reading this book. Thank you so much. I'm excited for you to start your journey to become a better React developer. I want to assure you that you are not alone in this profitable journey.

We are together!

So if at any point you have questions of any kind, feel free to contact me directly at ibas@ibaslogic.com.

You can also get in touch via Twitter: @ibaslogic.

So have fun while you code. Happy coding!

# ABOUT THE AUTHOR

***Ibas .M*** is a front-end developer and a creative writer. He specializes in developing realistic websites and web applications using the latest tools and tech. He also writes technical contents around this subject.

Ibas currently teaches JavaScript, React, Gatsby and WordPress theme development. And he has helped companies of different sizes launch their production websites/apps. He has a Computer Engineering degree from the University of Lagos, and he has been programming for about 9 years.

# OTHER BOOK BY IBAS

# ONE LAST THING

Thank you for reading! Don't forget to get all the bit of code used in this React book in my GitHub repository (https://github.com/Ibaslogic/).

***The Todos application code (Chapter 1 – 3) –***
github.com/Ibaslogic/todoapp

***Todos app (React HTTP Request and Lifecycle Methods) Chapter 4*** – github.com/Ibaslogic/http-request-lifecycle-methods

***Simple Meme Generator code (Chapter 5) –***
github.com/Ibaslogic/memegenerator.

***Todos app with React Hooks (Chapter 6) –***
github.com/Ibaslogic/todos-react-hooks

***Meme Generator with React Hooks (Chapter 7) –***
github.com/Ibaslogic/React-Hooks/blob/master/src/components/MemeGenerator.js

Please send me feedback

**If you found this book useful, I'd be very grateful if you'd contact me and send me feedback.**

Thank you!