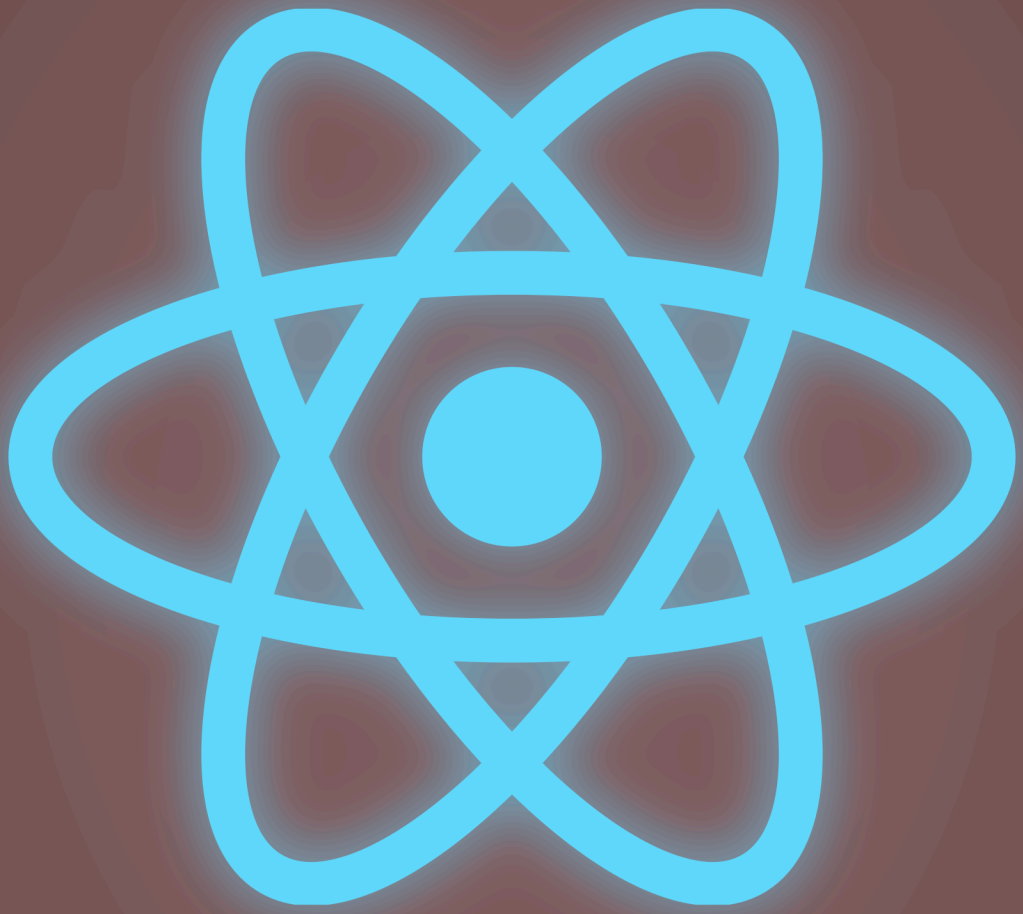
The React Mega-Tutorial



Miguel Grinberg

The React Mega-Tutorial

Miguel Grinberg

Jan 18, 2026

Contents

Preface

Welcome to the React Mega-Tutorial! In this book I will share my experience in developing real-world, non-trivial front end applications using the React[1] library and a handful of related packages.

Unlike most other books and tutorials, the React Mega-Tutorial will take you on a development journey. Instead of teaching you React concepts with isolated examples, it will show you how to develop a complete front end application. You will begin by creating a brand-new React project, and then start adding features and functionality to it as you progress through the chapters. When you reach the end, you will have a complete project of which you will understand every single line of code. More importantly, you will understand the concepts and techniques involved in creating it, in a way that will be directly applicable to your own projects.

Prerequisites

To make the most of this book, you need to have basic experience writing applications in JavaScript. If you have created websites that have bits of interactive behavior using vanilla JavaScript, or maybe jQuery, then you should be fine.

Basic knowledge of HTML and CSS is also assumed. As with JavaScript, a basic level of experience is sufficient. You should know how to create simple HTML markup for things such as paragraphs, headings and images, and how to apply styling changes to them with CSS definitions.

Knowledge of the command line in your operating system will be useful. You will spend most of the time on a code editor and a web browser, but there are some tasks here and there that need to be carried out in a terminal session.

If you are using a Microsoft Windows computer, my recommendation is that you use the Windows Subsystem for Linux (WSL)[2], which provides UNIX-compatible versions of bash, Node.js and all other required packages. Using the Windows command prompt or PowerShell should be possible, but you will need to adjust some command-line instructions.

You are welcome to use the editor that makes you most productive. At the time I'm writing this,

[1] https://reactjs.org/

[2] https://docs.microsoft.com/en-us/windows/wsl/about

Visual Studio Code[3] is the favorite editor of many JavaScript developers, but these things tend to change over time, and I've taken special care to not introduce any dependencies to this or other editors.

To test your application as you develop it, you may use your favorite web browser. If you need a recommendation, many React developers prefer Chrome[4] or Firefox[5], because they are the two browsers that support the React Developer Tools[6] plugin, which can sometimes be useful in debugging React applications.

How to Work with the Example Code

The complete source code for the project you'll work on is available as a GitHub repository at *https://github.com/miguelgrinberg/react-microblog*.

My suggestion is that you type or copy the application code on your own, using the instructions I provide, at least for the first few chapters. I would recommend that you don't rely too much on the code that is on GitHub, as I think it is important that you familiarize yourself with the task of coding the React application, since this is what you will be doing when you work on your own projects.

However, the GitHub repository can be an invaluable reference if you get stuck and can't get the application to work on your own. In the *README.md* file of the repository you will find links to the specific code changes covered in each chapter.

For most web applications, some functionality related to data persistence can only be implemented by a back end service (sometimes also called an API). This book does not cover back end development, but I have created a companion back end application that you will need to set up and run to support your work with React. The back end application is also available on a GitHub repository, at *https://github.com/miguelgrinberg/microblog-api*, in case you are interested in reviewing it. When the time comes, you will be given instructions on how to get this project up and running.

Acknowledgements

Writing a book is a huge task that would have been impossible for me to complete if I didn't have the support and encouragement of my family. My deepest gratitude goes to my wife and my children, for allowing me to spend long hours after a day of work, hunched over the computer to bring this project to life. My dog Hazel gets an honorable mention, for reminding me that it is important to take a break once in a while, go out for a walk (with her, of course), and breathe some fresh air.

I also greatly appreciate the support of my Patreon community[7]. My patrons have been a deciding factor in writing this book, through their questions, suggestions and ideas. In particular, I enjoyed

[3] https://code.visualstudio.com/
[4] https://www.google.com/intl/en_ie/chrome/
[5] https://www.mozilla.org/en-US/firefox/
[6] https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi
[7] https://www.patreon.com/miguelgrinberg

the many discussions we had on best practices around user authentication, which helped me give this topic the importance it deserves in the book.

1. Modern JavaScript

The JavaScript language has evolved significantly in the last few years, but because browsers have been slow in adopting these changes a lot of people have not kept up with the language. React encourages developers to use modern JavaScript, so this chapter gives you an overview of the newest features of the language.

1.1. ES5 vs. ES6

The JavaScript language specification is managed by ECMA[1], a non-profit organization that maintains a standardized version of this language known as ECMAScript[2].

You may have heard the terms "ES5" and "ES6" in the context of JavaScript language versions. These refer to the 5th and 6th editions of the ECMAScript standard respectively. The ES5 version of the language, which was released in 2009, is currently considered a baseline implementation, with wide support across desktop and mobile devices. ES6, released in 2015, introduces significant improvements over ES5, and remains backwards compatible with it. Since the release of ES6, ECMA has been making yearly revisions to the standard, which continue to improve and modernize the language. In many contexts, the "ES6" denomination is loosely used for all the improvements brought to the language after ES5, and not strictly those from the ES6 specification.

How can web browsers keep up with a language that evolves so rapidly? They actually can't and don't! Features that were introduced in ES6 and later updates to the standard aren't guaranteed to be implemented in all browsers. To avoid code failing to run due to missing language features, modern JavaScript frameworks rely on a technique called transpiling[3], which converts modern JavaScript source code into functionally equivalent ES5 code that runs everywhere. Thanks to transpiling, JavaScript developers don't have to worry about what parts of the JavaScript language browsers support.

[1] https://www.ecma-international.org/
[2] https://tc39.es/ecma262/
[3] https://en.wikipedia.org/wiki/Source-to-source_compiler

1.2. Summary of Recent JavaScript Features

The code in this book is written in modern JavaScript. Assuming you are familiar with the ES5 version of the language, you can use the sections that follow to refresh your knowledge of the newer parts of the language and fill any gaps that you may have.

1.2.1 Semicolons

The rules regarding when semicolons are required in JavaScript are confusing. The JavaScript compiler assumes implicit semicolons in some situations, but not in others. In practice, this means that in most cases, semicolons do not need to be typed. What makes things complicated is that in some situations they are still required.

To avoid the confusion generated by the semicolon rules, for this book I have decided to use explicit semicolons after all statements. I do not use a semicolon after a closing } at the end of a function declaration or a control structure, such as a loop or a conditional. Why these exceptions? Most other languages that use the semicolons as statement separators do not require it after the closing brace.

Here are some examples:

```
const a = 1;  // <-- semicolon here

function f() {
  console.log('this is f');  // <-- semicolon here
}  // <-- but not here
```

An interesting situation occurs when an arrow function is assigned to a variable or constant. I consider this an exception to the exception rule above, so I use a semicolon in this case:

```
const f = () => {
  console.log('this is f');
};  // <-- this is an assignment, so a semicolon is used
```

It goes without saying that you do not need to adopt my semicolon choices if you don't like them. If you have developed a personal preference for using or omitting semicolons, you can definitely use it in place of my own.

1.2.2 Trailing Commas

When defining objects or arrays that span multiple lines, it is useful to leave a comma after the last element. Look at the following examples:

```
const myArray = [
  1,
  3,
  5,
];

const myObject = {
  name: 'susan',
```

(continues on next page)

(continued from previous page)

```
  age: 20,
};
```

The commas after the last elements of the array and object might seem like a syntax error at first, but they are valid. In fact, JavaScript is not unique in allowing this, as most other languages support trailing commas as well.

There are two benefits that result from this practice. The most important one is that if you need to reorder the elements, you can just move the lines up and down, without worrying about having to fix the commas. The other one is that when you need to add an element at the end, you do not need to go up to the previous line to add the trailing comma.

1.2.3 Imports and Exports

If you are used to writing old-style JavaScript applications for the browser, you probably never needed to "import" functions or objects from other modules. You simply added `<script>` tags that loaded your dependencies, and that was enough to bring what you needed into the global scope, which is accessible to any JavaScript code running in the context of the current page.

Thanks to the tooling incorporated into modern JavaScript front end frameworks, applications can now use a much more sane dependency model that is based on *imports* and *exports*.

A JavaScript module that wants to make a function or variable available for other modules to use, can declare it as a default export. Let's say there is a *cool.js* module with `myCoolFunction()` inside. Here is how this module could be written:

```
export default function myCoolFunction() {
  console.log('this is cool!');
}
```

Any other module that wants to use the function can then import it:

```
import myCoolFunction from './cool';
```

In this import, `./cool` is the path of the dependent module, relative to the location of the importing file. The path can navigate up or down the directory hierarchy as necessary. The `.js` extension can be included in the import filename, but it is optional.

When using default exports, the name of the exported symbol does not really matter. The importing module can use any name it likes. The next example is also valid:

```
import myReallyCoolFunction from './cool';
```

Importing from third-party libraries works similarly, but the import location uses the library name instead of a local path. For example, here is how to import the `React` object:

```
import React from 'react';
```

A module can have only one default export, but it can also export additional things. Here is an extension of the above *cool.js* module with a couple of exported constants (you'll learn more about constants in the next section):

```
export const PI = 3.14;
export const SQRT2 = 1.41;

export default function myCoolFunction() {
  console.log('this is cool!');
}
```

To import a non-default export, the imported symbol must be enclosed in { and } braces:

```
import { SQRT2 } from './cool';
```

This syntax also allows multiple imports in the same line:

```
import { SQRT2, PI } from './cool';
```

Default and non-default symbols can also be included together in a single import line:

```
import myCoolFunction, { SQRT2, PI } from './cool';
```

If you want to learn about imports and exports in more detail, consult the import[4] and export[5] sections in the JavaScript reference.

1.2.4 Variables and Constants

Older JavaScript versions were very sloppy in terms of how to declare variables. Starting with ES6, the `let` and `const` keywords are used for the declaration of variables and constants respectively. You may have seen the `var` keyword used to declare variables in older versions of JavaScript. The `var` keyword has some scoping quirks, so it is best to replace it with the more predictable `let`.

To define a variable, just prepend it with the `let` keyword:

```
let a;
```

It is also possible to declare a variable and give it an initial value at the same time:

```
let a = 1;
```

If an initial value isn't given, the variable is assigned the special value `undefined`.

A constant is a variable that can only be assigned a value when it is declared:

[4] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import
[5] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export

```
const c = 3;

console.log(c); // 3
c = 4;  // error
```

While it may look confusing, it is perfectly legal to create a constant and assign a mutable object to it. For example:

```
const d = [1, 2, 3];

d.push(4);  // allowed
console.log(d)  // [1, 2, 3, 4]
```

Why does this work? Because the requirement for constants is that they do not have a new assignment after declaration. There are no requirements regarding mutating the initially assigned value.

The JavaScript reference documentation contains more information about let[6] and const[7].

1.2.5 Equality and Inequality Comparisons

Older JavaScript implementations had very strange rules in regard to automatic casting of values between different types. For that reason, the original equality (==) and inequality (!=) operators work in ways that may appear wrong, or at least different to what you would expect.

To avoid breaking older code, these comparison operators preserve the odd behaviors, but recent versions of JavaScript introduced new comparison operators === and !==, so that more predictable comparisons can be used.

In general, all equality and inequality comparisons should use the newer operators. Examples:

```
let a = 1;

console.log(a === 1);  // true
console.log(a === '1');  // false
console.log(a !== '1');  // true
```

Given that many other languages use the == and != operators for comparisons, it is very common to inadvertently use these when writing JavaScript. In a properly set up project (such as the one you will build with this book), static code analysis tools can detect and warn about this mistake.

See the Strict equality[8] and Strict inequality[9] operators in the JavaScript reference documentation for more details.

[6] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let
[7] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const
[8] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Strict_equality
[9] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Strict_inequality

1.2.6 String Interpolation

Many times it is necessary to create a string that includes a mix of static text and variables. ES6 uses *template literals* for this:

```
const name = 'susan';
let greeting = `Hello, ${name}!`;  // "Hello, susan!"
```

There are more examples in the Template literals[10] reference documentation.

1.2.7 For-Of Loops

Older versions of JavaScript only provide strange and contorted ways to iterate over an array of elements, but luckily ES6 introduces the `for ... of` statement for this purpose.

Given an array, a for-loop that iterates over its elements can be constructed as follows:

```
const allTheNames = ['susan', 'john', 'alice'];
for (name of allTheNames) {
  console.log(name);
}
```

1.2.8 Arrow Functions

ES6 introduces an alternative syntax for the definition of functions that is more concise, in addition to having a more consistent behavior for the `this` variable, compared to the `function` keyword.

Consider the following function, defined in the traditional way:

```
function mult(x, y) {
  const result = x * y;
  return result;
}

mult(2, 3);  // 6
```

Using the newer arrow function syntax, the function can be written as follows:

```
const mult = (x, y) => {
  const result = x * y;
  return result;
};

mult(2, 3);  // 6
```

Looking at this it isn't very clear why the arrow syntax is better, but this syntax can be simplified in a few ways. If the function has a single statement instead of two, then the curly braces and the `return` keyword can be omitted, and the entire function can be written in a single line:

[10] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals

```
const mult = (x, y) => x * y;
```

If the function accepts a single argument instead of two, then the parenthesis can also be omitted:

```
const square = x => x * x;

square(2);  // 4
```

When passing a callback function as an argument to another function, the arrow function syntax is more convenient. Consider the following example, shown with traditional and arrow function definitions:

```
longTask(function (result) { console.log(result); });

longTask(result => console.log(result));
```

See the Arrow function[11] documentation for more information.

1.2.9 Promises

A *promise* is a proxy object that is returned to the caller of an asynchronous operation running in the background. This object can be used by the caller to keep track of the background task and obtain a result from it when it completes.

The promise object has `then()` and `catch()` methods (among others) that allow the construction of chains of asynchronous operations with solid error handling.

Many internal and third-party JavaScript libraries return promises. Here is an example use of the `fetch()` function to make an HTTP request, and then print the status code of the response:

```
fetch('https://example.com').then(r => console.log(r.status));
```

This executes the HTTP request in the background. When the fetch operation completes, the arrow function passed as an argument to the `then()` method is invoked with the response object as an argument.

Promises can be chained. A common case that requires chaining is when making an HTTP request that returns a response with some data. The following example shows how the request operation is chained to a second background operation that reads and parses JSON data from the server response:

```
fetch('http://example.com/data.json')
  .then(r => r.json())
  .then(data => console.log(data));
```

This is still a single statement, but I have broken it up into multiple lines to increase clarity. Once the `fetch()` call completes, the callback function passed to the first `then()` executes with the response object as an argument. This callback function returns `r.json()`, a method of the response object

[11] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions

that also returns a promise. The second `then()` call is invoked when the second promise completes, receiving the parsed JSON data as an argument.

To handle errors, the `catch()` method can be added to the chain:

```
fetch('http://example.com/data.json')
  .then(r => r.json())
  .then(data => console.log(data))
  .catch(error => console.log(`Error: ${error}`));
```

For additional details on promises, consult the Promise API documentation[12].

1.2.10 Async and Await

Promises are a nice improvement that help simplify the handling of asynchronous operations, but having to chain several actions in long sequences of `then()` calls can still generate code that is difficult to read and maintain.

In the 2017 revision of ECMAScript, the `async` and `await` keywords were introduced as an alternative way to work with promises. Here is the first `fetch()` example from the previous section once again:

```
fetch('http://example.com/data.json')
  .then(r => r.json())
  .then(data => console.log(data));
```

Using async/await syntax, this can be coded as follows:

```
async function f() {
  const r = await fetch('https://example.com/data.json');
  const data = await r.json();
  console.log(data);
}
```

With this syntax, the asynchronous tasks can be given sequentially, and the resulting code looks very close to how it would be with synchronous function calls. A limitation is that the `await` keyword can only be used inside functions declared with `async`.

Error handling in async functions can be implemented with `try/catch`:

```
async function f() {
  try {
    const r = await fetch('https://example.com/data.json');
    const data = await r.json();
    console.log(data);
  }
  catch (error) {
    console.log(`Error: ${error}`);
  }
}
```

[12] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

An interesting feature of functions declared as `async` is that they are automatically upgraded to return a promise. The `f()` function above can be chained to additional asynchronous tasks using the `then()` method if desired:

```
f().then(() => console.log('done!'));
```

Or of course, it can also be awaited if the calling function is also async:

```
async function g() {
  await f();
  console.log('done!');
}
```

The arrow function syntax can also be used with `async` functions:

```
const g = async () => {
  await f();
  console.log('done!');
};
```

The Making asynchronous programming easier with async and await[13] section of the JavaScript documentation is a good place to learn more.

1.2.11 Spread Operator

The *spread operator* (`...`) can be used to expand an array or object in place. This allows for very concise expressions when working with arrays or objects. The best way to learn the spread operator is through some examples.

Let's say you have an array with some numbers, and you want to find the smallest of them. The traditional way to do this would require a for-loop. With the spread operator, you can leverage the `Math.min()` function, which takes a variable list of arguments:

```
const a = [5, 3, 9, 2, 7];
console.log(Math.min(...a));  // 2
```

The basic idea is that the `...a` expression expands the contents of `a`, so the `Math.min()` function receives five independent arguments instead of single array argument.

The spread operator can also be used to create a new array by mixing another array with new elements:

```
const a = [5, 3, 9, 2, 7];
const b = [10, ...a, 8, 0];  // [10, 5, 3, 9, 2, 7, 8, 0]
```

It also allows for a simple way to do a shallow copy of an array:

```
const c = [...a];  // [5, 3, 9, 2, 7]
```

The spread syntax also works with objects:

[13] https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Async_await

```
const d = {name: 'susan'};
const e = {...d, age: 20};  // {name: 'susan', age: 20}
const f = {...d};  // {name: 'susan'}
```

An interesting usage of the spread operator on objects is to make partial updates:

```
const user = {name: 'susan', age: 20};
const new_user = {...user, age: 21};  // {name: 'susan', age: 21}
```

Here, the collision that occurs when having two values for the `age` key is resolved by using the version that appears last.

See the Spread syntax[14] reference for more details.

1.2.12 Object Property Shorthand

In the same league as the spread operator, the object property shorthand provides a simplified syntax for object properties. Consider how the following object is created:

```
const name = 'susan';
const age = 20;
const user = {name: name, age: age};
```

Do you see the repetition of `name` and `age`? These keywords are used as property names, and also as the names of the constants that hold the property values. With the object property shorthand syntax, you can create the same object as follows:

```
const user = {name, age};
```

Objects created as above use the name of the given variable or constant as the property name, and also assign the value to the new property.

Shorthand and regular properties can be combined as well:

```
const user = {name, age, active: true};  // {name: 'susan', age: 20, active: true}
```

1.2.13 Destructuring Assignments

Destructuring assignments are yet another nice syntax shorthand that can be used to simplify assignments of arrays and objects. The idea is that the right side value can be decomposed into its elements on the fly as part of the assignment operation. Here is an array example:

```
const a = ['susan', 20];
let name, age;
[name, age] = a;
```

The square brackets on the left side of the assignment tell JavaScript that the right side is an array that must be taken apart before assigning the elements to the list of variables.

[14] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax

What happens if the number of elements in the left and right sides don't match? If the left side has more elements than the right side, then the extra elements on the left are assigned the `undefined` value. If the right side has more elements than the left side, then the extra elements are discarded.

There is an interesting combination between the destructuring assignment and the spread operator discussed above. Consider the following example:

```
const b = [1, 2, 3, 4, 5];
let c, d, e;
[c, d, ...e] = b;
console.log(c);  // 1
console.log(d);  // 2
console.log(e);  // [3, 4, 5]
```

Destructuring assignments can also be used with objects:

```
const user = {name: 'susan', active: true, age: 20};
const {name, age} = user;
console.log(name);  // susan
console.log(age);  // 20
```

This technique can be applied not only to direct assignments, but also to function arguments. The following example demonstrates it:

```
const f = ({ name, age }) => {
  console.log(name);  // susan
  console.log(age);  // 20
};

const user = {name: 'susan', active: true, age: 20};
f(user);
```

Here the `f()` arrow function accepts an object as its only argument, but instead of accepting the whole object, the function just takes the `name` and `age` properties from the input. As with assignments, if the object has additional properties, they are discarded, and if any of the named properties in the function declaration are not in the object, then they are assigned the `undefined` value.

To learn about more ways to use this feature consult the Destructuring Assignment[15] section of the JavaScript reference.

1.2.14 Classes

A big omission in the earlier versions of the JavaScript language up to, and including ES5 is *classes*, which are the core component of object-oriented programming. Below you can see an example of an ES6-style class:

```
class User {
  constructor(name, age, active) {  // constructor
    this.name = name;
    this.age = age;
```

(continues on next page)

[15] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

(continued from previous page)

```
    this.active = active;
  }

  isActive() {  // standard method
    return this.active;
  }

  async read() {  // async method
    const r = await fetch(`https://example.org/user/${this.name}`);
    const data = await r.json();
    return data;
  }
}
```

To create an instance of a class, the `new` keyword is used:

```
const user = new User('susan', 20, true);
```

Learn more about classes[16] in the JavaScript reference.

1.2.15 JSX

The last modern JavaScript feature discussed in this chapter is in a category of its own, as it is not part of any ECMAScript specification, and it is not intended to ever be. It is called JSX[17], which is short for JavaScript XML. Its purpose is to make it easier to create inline structured content, mainly to be used in HTML pages.

Let's say that you need to create an HTML paragraph element. Using plain JavaScript and the DOM API, you could create this <p> element as follows:

```
const paragraph = document.createElement('p');
paragraph.innerText = 'Hello, world!';
```

Can you imagine what it would be like to create a more complex tree of elements, like maybe a complete table, using plain JavaScript? With JSX, it becomes much easier, and the resulting code is more readable:

```
const paragraph = <p>Hello, world!</p>;
```

Here is a more complex example of an HTML table:

```
const myTable = (
  <table>
    <tr>
      <th>Name</th>
      <th>Age</th>
    </tr>
    <tr>
      <td>Susan</td>
```

(continues on next page)

[16] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes
[17] https://facebook.github.io/jsx/

(continued from previous page)

```
      <td>20</td>
    </tr>
    <tr>
      <td>John</td>
      <td>45</td>
    </tr>
  </table>
);
```

The JSX syntax is a key component of React applications. While technically not required by React, it makes HTML content and templates much easier to create and maintain.

2. Hello, React!

In this chapter you will take your first steps as a React developer. When you reach the end you will have a first version of a microblogging application running on your computer!

2.1. Installing Node.js

While React is a front end framework that runs in the browser, some related utilities are designed to run on your own computer, using the Node.js[1] engine. The project that creates a brand-new React project, for example, runs on Node.

If you don't have a recent version of Node installed, head over to the Node.js download page[2] to obtain it. My recommendation is that you use the latest available LTS (long term support) version.

You can confirm that Node.js is properly installed on your system by opening a terminal and running the following command to check what version you have installed:

```
node -v
```

2.2. Creating a Starter React Project

There are many ways to create a new React application. The React documentation recommends[3] the Create React App[4] utility when you are creating a brand new single-page application.

Open a terminal window, find a suitable parent directory, and then enter the following command to create a new React project called `react-microblog`:

```
npx create-react-app react-microblog
```

The `npx` command comes with Node.js, along with `npm` which you may be more familiar with. Its purpose is to execute Node.js packages. The first argument to `npx` is the package to execute, which

[1] https://nodejs.org/
[2] https://nodejs.org/en/download/
[3] https://reactjs.org/docs/create-a-new-react-app.html#recommended-toolchains
[4] https://create-react-app.dev/

is called `create-react-app`. Additional arguments are passed to this package once it runs. The Create React App package takes the name of the React project to create as an argument.

You may have noticed that you did not need to install `create-react-app` prior to running it. The nice thing about `npx` is that it downloads and runs the requested package on the fly.

Let's have a look at the newly created project. First change into the *react-microblog* directory:

```
cd react-microblog
```

Get a directory listing (`ls` on Unix and Mac, `dir` on Windows) to familiarize yourself with your new project.

Depending on the version of Create React App that you use the contents of the project may not match mine exactly, but you should expect to have the following files and directories:

- *README.md*: a short document with instructions on how to use your React project.
- *package.json* and *package-lock.json*: the standard Node.js project metadata files, with a description of your project and its dependencies.
- *public*: the directory from where the React application will be served during development. Inside this directory you will find the *index.html* page, which loads the application in the browser, some icon files and other miscellaneous static files.
- *src*: the source code directory for the application. This is where you will do your coding. The starter application comes with a few source files for a simple demo application.
- *node_modules*: the standard directory where Node.js installs all the dependencies of the project.
- *build*: this is not a directory that appears on a freshly created project, but it will be added later, when you create a production build of your project.
- *.git* and *.gitignore*: git repository files. This isn't obvious at first glance because these are hidden, but Create React App also makes the application a local git repository.

Before you continue, you'd want to make sure that the starter project works well. To start the React development web server, run the following command:

```
npm start
```

This command runs an initial development build, and creates a web server that serves the React application at the *http://localhost:3000* URL. It also opens this URL in your default web browser. Figure 2.1 shows what you should see in your browser.

Once the application is up and running in the browser, the `npm start` command enters a source code watch loop. Whenever it detects that changes to source files were made, it automatically rebuilds the application and sends the updated code to the browser. This automated monitoring and refreshing of the application is extremely convenient, so I recommend that you keep the `npm start` command running at all times while working on the application.

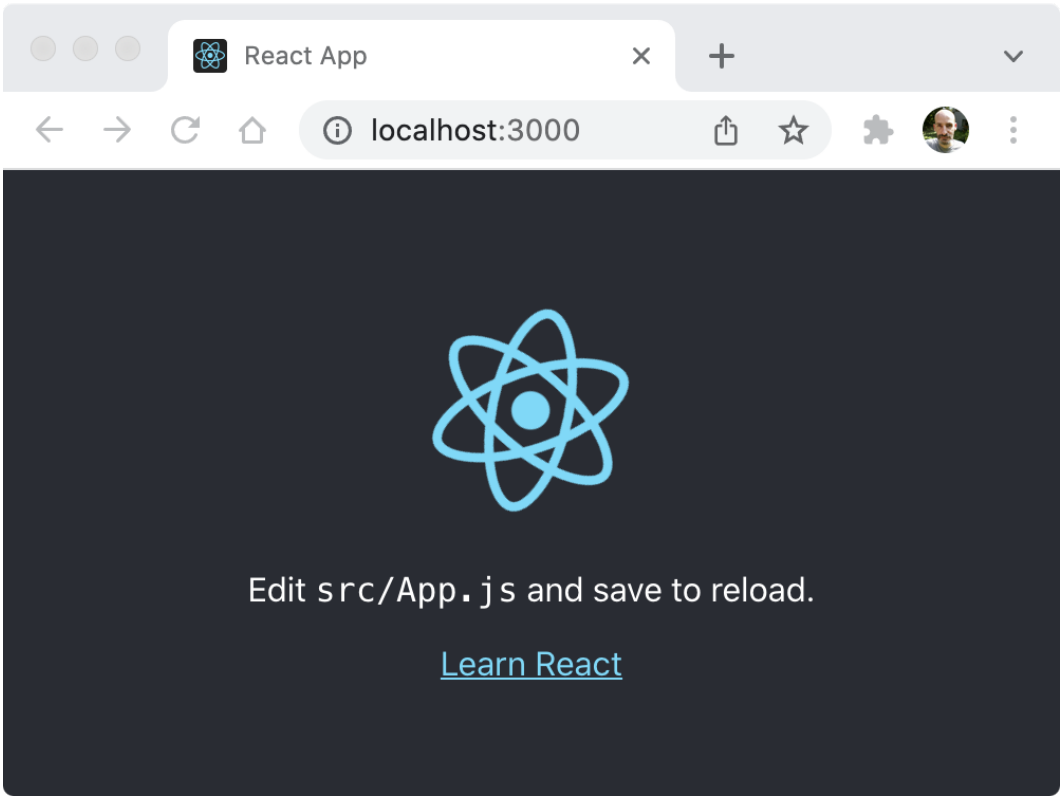


Figure 2.1: React starter application

2.3. Installing Third-Party Dependencies

The project, as created by the Create React App utility, contains a standard Node.js *package.json* file that among other things lists all the third-party dependencies used for the project. The initial list of dependencies includes the `react` and `react-scripts` libraries, plus a few other related packages.

The application that you are going to build requires a few more packages, so this is a good time to get them installed. Run the following command from the top-level directory of the project:

```
npm install bootstrap react-bootstrap react-router-dom serve
```

What are all these packages? Here is a brief summary:

- bootstrap[5]: a CSS user interface library for web pages.
- react-bootstrap[6]: a React component library wrapper for the `bootstrap` package.
- react-router-dom[7]: a React component library that implements client-side routing.
- serve[8]: a static file web server that can be used to run the production version of the React application.

You will become familiar with these packages as you work on the project.

2.4. Application Structure

On the one side, generating a starter application with Create React App saves a lot of time, but on the other you end up with an application that has some unnecessary components that need to be removed before embarking on a new project. In this section you will learn about the general structure of the React application you just created, and while you are at it, you will remove some cruft.

Before you continue, make sure that you have the `npm start` command running in a terminal window, and a tab in your browser open on the application. Some of the changes you are going to make soon will cause the application to temporarily break, so this is a great opportunity for you to experience the way the React development server watches your work and notifies you of errors.

2.4.1 The *index.html* File

If you have worked on other web development projects, you probably know that at the core of a web application that is loaded by the browser there is an HTML file. Once the HTML file is parsed, the browser finds all the references to additional resources needed to render the page, such as images, fonts, stylesheets and JavaScript code, and loads those as well.

[5] https://getbootstrap.com/
[6] https://react-bootstrap.github.io/
[7] https://reactrouter.com/
[8] https://www.npmjs.com/package/serve

The *index.html* stored in the *public* directory is the main HTML page for the React application, or to be more accurate, it is a template from which the main HTML page is generated by the React build.

Since React is a *single-page application* library, this is the only page that the browser will load. Once the page and all its referenced resources are downloaded by the browser, state changes to the application will be managed strictly through JavaScript events, always in the context of this page.

Open *public/index.html* in your editor and look through it. In the <head> section of the page, you will find some boilerplate that configures icons, character sets and other important page metadata, all very conveniently generated by Create React App.

Somewhere in this section you will find the page's meta description tag, which is used by search engines to show some information about your website in search results. This is the meta description tag generated by Create React App:

```
<meta
  name="description"
  content="Web site created using create-react-app"
/>
```

The first change you are going to make is to update this description to something that is less generic. You are welcome to be creative and write your own description, but here is an example:

Listing 2.1: *public/index.html*: Updated meta description tag

```
<meta
  name="description"
  content="Microblogging application featured in the React Mega-Tutorial"
/>
```

A few lines below the meta description tag, you will find the page title:

```
<title>React App</title>
```

The browser tab in which you are running the application displays the React favicon[9], followed by this title.

Change the page title to "Microblog", as shown below:

Listing 2.2: *public/index.html*: Updated page title

```
<title>Microblog</title>
```

Save the *index.html* and then watch the application in the browser. A second or two after you save, the title in the browser tab will update.

There are no more changes needed in the *index.html* file, but before you close this file in your editor, scroll down to the <body> section. In this section you are going to find the root element of the application, which looks like this:

[9] https://en.wikipedia.org/wiki/Favicon

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

When the React application starts, it will insert the contents of the page inside this element. In general, you will not need to interact with this `<div>`, but it is good to understand how the React application ends up in the page.

2.4.2 The *manifest.json* File

In addition to *index.html* and the icons, the *public* directory has a file called *manifest.json*. This file provides information about the application in JSON[10] format. Client devices such as smartphones and tablets use the information in this file to provide a better experience when the user installs (or creates a shortcut to) the application in their desktop or home screen.

The `short_name` and `name` keys are set to generic values, similar to those in the index page. You can update these to "Microblog" and "React Microblog" respectively:

Listing 2.3: *public/manifest.json*: Updated application metadata

```
  "short_name": "Microblog",
  "name": "React Microblog",
```

The rest of the file defines icons of various sizes, and theme colors for the application. You can leave these settings as they are.

2.4.3 The Icon Files

The favicon and some larger icons of various dimensions used for desktop and mobile shortcuts are all stored in the *public* directory. If you have the inclination, you can edit or replace these files with your favorite icon design, but this can be done at any time, so it is also fine to leave the files alone for now.

2.4.4 The *index.js* File

Moving on to the *src* directory, the *index.js* file is the main JavaScript file that is loaded by *index.html*. The task of this file is to bootstrap the React application.

The code in *index.js* has the entry point for the React rendering engine. The code in this file may vary depending on what version of React and Create React App you use, but in general you should expect to have a main section with a call to render the application, similar to the following:

```
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
);
```

[10] https://en.wikipedia.org/wiki/JSON

You may notice that there are some strange things in this source file. First, JSX is used in the argument to the `render()` function. Also, a file with extension *.css* is being imported at the top, which does not make a lot of sense in terms of the JavaScript language. Keep in mind that all source files in a React project go through a conversion step before they reach the browser, where any extensions to the JavaScript language are processed and converted into valid code.

What does importing a file with a *.css* extension achieve? One of the main functions of this source code conversion process is to generate optimized JavaScript and CSS bundles that are then downloaded by the browser. Importing a CSS file does not change the JavaScript source code at all, but it informs the build that the referenced CSS file should be added to the application's CSS bundle.

The purpose of the `render()` function is to generate the contents of the application and apply them to the "root" node from the *index.html* file. The argument to the render function is a JSX tree representing the entire application.

The `App` symbol is imported from the *App.js* source file. This is the top-level component of the application. You will learn more about components shortly, but for now, consider that `App` represents a hierarchical collection of elements that represent the entire application. A React component is considered a super-powered HTML element, and for that reason it is rendered using angle brackets, such as a standard HTML element. When looking at a JSX tree, you can tell the difference between native HTML elements and React components because the former use lowercase letters and the latter use CamelCase.

What is the `<React.StrictMode>` component that wraps `<App />`? This is a component that is part of the React library. It does not render anything visible to the page, but enables some internal checks during development that alert you of possible problems with your code.

At the bottom of *index.js* you may see a call to a `reportWebVitals()` function. You are not going to use this in this project, but in case you are interested, generating Web Vitals[11] metrics is an optional feature of React that allows you to analyze the performance of your application.

There is only one change that you are going to make in *index.js*. As you recall, earlier you installed a few third-party packages. One of those packages was `bootstrap`, which is a CSS framework. To add this library to the project, its CSS file must be imported. Insert the following import statement right above the line in which the *index.css* file is imported:

Listing 2.4: *src/index.js*: Add the bootstrap framework

```
import 'bootstrap/dist/css/bootstrap.min.css';
```

The reason why *bootstrap.min.css* is imported before *index.css* is that this gives the application the option to redefine or override the styles that come with the library.

As soon as you save the file, you will notice that the look of the application in the browser changes slightly. This is because the bootstrap styles introduce some minor visual differences on the rendered page.

[11] https://create-react-app.dev/docs/measuring-performance/

2.4.5 Deleting Unnecessary Files

The *src* directory has two files called *logo.svg* and *App.css* that are not going to be used by the application, so you are going to delete them.

Why are these files unnecessary? The *logo.svg* file is the rotating React logo that appears in the center of the page in the starter application. This is a nice graphic for the demo application, but it has no place in your own application. The *App.css* file stores CSS definitions for the top-level application component, but the project also has an *index.css* file with application-wide styles, so there is some redundancy in having two CSS files. For this project I have decided to maintain the entire collection of CSS styles in the *index.css* file.

Since the application was created as a git repository, the preferred way to delete these files is to use `git rm` from a terminal window, as shown below:

```
git rm src/logo.svg src/App.css
```

Right after you delete these two files the `npm start` process is going to get upset and spew a few error messages on your terminal and on the browser. This is because the *App.js* source file references the two files that are now gone.

In spite of the errors, `npm start` will continue to monitor your files and wait for you to fix the errors to restart the application.

2.5. A Basic React Application

The *App.js* file in the *src* directory is currently broken, as it references files that don't exist anymore in the project. In this section you are going to replace the code in this file with a simpler base application that does not require any external files.

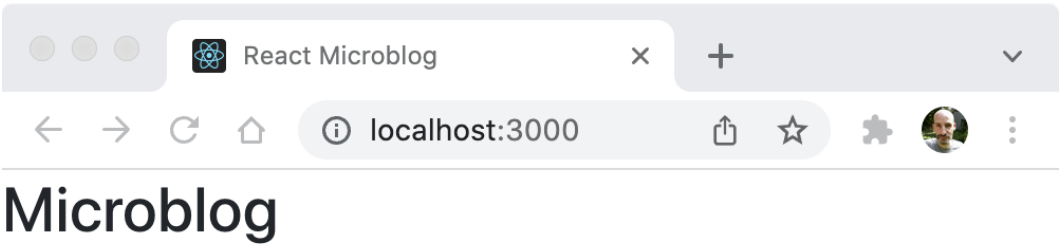
Open *src/App.js* in your favorite editor or IDE, delete all of its contents and replace them with the following code:

Listing 2.5: *src/App.js*: Basic application component

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

As soon as you save the new version of *App.js*, the application will automatically reload in your browser and display the <h1> heading in the top-left corner of the page, as shown in Figure 2.2.

In React, components can be written as classes or as functions. Function-based components are newer and use a more concise syntax, so that is what you'll learn. Functional components were introduced in React 16.8 and are associated with another important feature from that release called *hooks*. You will learn about hooks in future chapters.



Microblog

Figure 2.2: Basic application

You now know that a React component is implemented as a JavaScript function. The name of the component is the name of the function, in this case `App`. Component names must begin with a capital letter, and in general are written in CamelCase.

The top-level component of an application is often given the name `App`, but this is just a convention, not a rule. Component functions need to be exported, so that they can be imported and used from other files. For that reason, component functions are always defined with `export default function ...`.

To keep the source code well organized, a component is written in a source file of the same name, so for example, the `App` component is written in a source file named *App.js*. file.

Now on to the most important question. What is a component function supposed to do? In its simplest form, a component must return a representation of itself as an HTML element tree. The function is said to be the component's *render* function for that reason. The `App` component above renders itself as an `<h1>` element with the application's name in it. When the application runs in the browser, this `<h1>` element is inserted as a child of the root `<div>` element defined in the application's *index.html* file.

Note the parenthesis that enclose the JSX that the function returns. Due to the strange and somewhat unpredictable semicolon rules in JavaScript, the opening parenthesis needs to be in the same line as the `return` keyword, to prevent the JavaScript compiler from inserting a virtual semicolon on that line. The opening parenthesis tells the compiler that the expression continues in the next line.

2.6. Dynamic Rendering

Rendering chunks of HTML returned by component functions is nice, but insufficient for the vast majority of applications you may want to build. A key feature that most applications need is the ability to render content that is dynamic. For example, the Microblog application you are going to build needs to render blog posts that are not known in advance, and will be retrieved from a back end service.

The JSX syntax can be expanded with *templating* expressions, which make it possible to render content stored in variables that can be single values or lists, and it is even possible to define conditional rendering rules.

2.6.1 Rendering Variables

A JSX definition can include JavaScript expressions, given inside curly brackets. For example, if a variable `name` is set to the string `'susan'`, the JSX expression `<h1>Hello, {name}!</h1>` would render `<h1>Hello, susan!</h1>` to the page. This not only works for text, but also for attributes of elements, so for example, `<img src={image_url} />` would render an image that references the URL stored in the `image_url` variable.

This application is going to render blog posts, but these will be obtained from a back end, but at this early stage none of that is available. To avoid getting a stuck with too many problems that need a

solution, a useful technique is to *mock* parts of the application that aren't ready yet.

Given that there is no back end yet, let's create a mock blog post that can be rendered to the page. Replace the code in *App.js* with the following:

Listing 2.6: *src/App.js*: Render a blog post

```
export default function App() {
  const post = {
    id: 1,
    text: 'Hello, world!',
    timestamp: 'a minute ago',
    author: {
      username: 'susan',
    },
  }

  return (
    <>
      <h1>Microblog</h1>
      <p>
        <b>{post.author.username}</b> &mdash; {post.timestamp}
        <br />
        {post.text}
      </p>
    </>
  );
}
```

The `post` constant defined at the start of the `App()` function is a fake blog post that can be used to implement the code that renders posts to the page, without having to implement the connection to a back end first, something that is a much more complex task.

The `return` statement renders the same `<h1>` element as before, followed by a `<p>` element that includes the author's username, the timestamp and the text of the blog post, with minimal HTML formatting. Text that is included directly in the JSX block is rendered verbatim to the page, while text that is inside `{ }` is evaluated as a JavaScript expression, and the result is what is rendered to the page.

In the previous version of this code, a single `<h1>` element was returned, but in this version there's a `<p>` as well. React requires that the render tree returned by a component is a proper tree, with a single top-level node. It would be easy to add a parent `<div>`, but that will render an unnecessary element to the page. Using empty tags `<>` and `</>` is more efficient, as these do not produce any render output. These tags create a *fragment*, which is an invisible parent that allows the grouping of multiple nodes into a single tree.

The `<br />` notation might also look strange if you are used to often see `<br>` to insert line breaks in HTML pages. JSX requires a strict XML syntax, so all elements must be properly closed.

Figure 2.3 shows how the blog post looks on the browser.

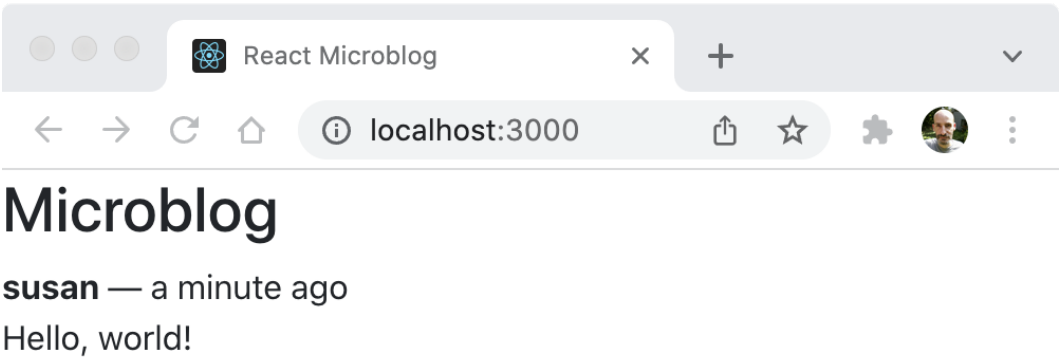


Figure 2.3: Render a blog post

2.6.2 Rendering Lists of Elements

The techniques shown in the previous section can be used with variables or constants that are assigned simple values or objects. But what happens if you need to render an array?

When the expression inside curly brackets is an array, React outputs the elements of the array one after another. Consider the following example:

```
export default function RenderArray() {
  const data = ['one', 'two', 'three'];

  return (
    <>{data}</>
  );
}
```

The output of this component is going to be `onetwothree`, which isn't very useful, as there is no separation between the items and no way to add HTML markup for each element of the array.

The way to make this work is to use the `map()` method of the `Array` class to transform each element into the desired JSX expression. The `map()` method takes a function as an argument. This function is called once for each of the elements in the array, with the element as its only argument. The return values for all these calls are collected and returned by `map()` as a new array, and this becomes the render output of the component.

To render the above list as an HTML unordered list, you could rewrite the component as follows:

```
export default function RenderArray() {
  const data = ['one', 'two', 'three'];

  return (
    <ul>
      {data.map(element => {
        return <li>{element}</li>
      })}
    </ul>
  );
}
```

If you aren't familiar with the `Array.map()` method, the inner `return` statement in this last example may seem strange. Remember that `map()` takes a function as its argument, so this inner `return` is returning values back to `map()`, which is the caller of the inner function.

With the `map()` method, the output of the component becomes a proper HTML list:

```
<ul><li>one</li><li>two</li><li>three</li></ul>
```

This technique can be applied to Microblog. In the previous section, the application rendered a single post. That can be extended to work with a list of posts. Replace the code in *src/App.js* with the following:

Listing 2.7: *src/App.js*: Render a list of blog posts

```
export default function App() {
  const posts = [
    {
      id: 1,
      text: 'Hello, world!',
      timestamp: 'a minute ago',
      author: {
        username: 'susan',
      },
    },
    {
      id: 2,
      text: 'Second post',
      timestamp: 'an hour ago',
      author: {
        username: 'john',
      },
    },
  ];

  return (
    <>
      <h1>Microblog</h1>
      {posts.map(post => {
        return (
          <p>
            <b>{post.author.username}</b> &mdash; {post.timestamp}
            <br />
            {post.text}
          </p>
        );
      })}
    </>
  );
}
```

You can see how the list of posts looks in the browser in Figure 2.4.

This looks great, but there is a hidden problem. If you look at the browser's debugging console, you will see a warning message.

```
Warning: Each child in a list should have a unique "key" prop.

Check the render method of `App`.
```

React has a performance optimization that triggers when a list that is rendered by a component changes. The goal is to update lists efficiently, by only updating the elements that were added, removed or changed. To be able to determine which elements of a list need to be updated, React requires each list element to be given a unique `key` attribute. When all elements have a key, React can compare the current and new versions of the list and determine which elements are new, removed or updated. More importantly, it allows React to know which of the elements have not changed at all, so it can optimize the render process by reusing those items from the previous page update.

When running in development mode, React triggers this warning when it finds a list that is rendered without keys. To remove the warning, a `key` attribute needs to added to the top-level JSX node

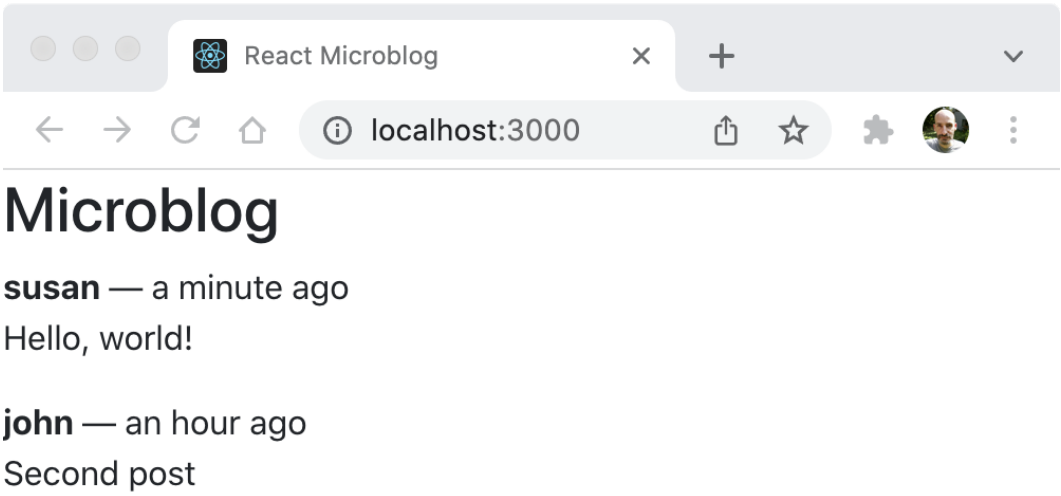


Figure 2.4: Render a list of blog posts

rendered for each element. Depending on the elements, you will need to find a unique value that can serve as identifier. Objects that are retrieved from a server back end often have an `id` attribute that works perfectly as keys.

Below is the posts loop, modified to use the `id` attributes defined in the fake blog posts as keys.

Listing 2.8: *src/App.js*: Render a list of blog posts with unique keys

```
      {posts.map(post => {
        return (
          <p key={post.id}>
            ...  // <-- no changes to the post JSX
          </p>
        );
      })}
```

2.6.3 Conditional Rendering

The last templating trick you are going to learn gives the React application the ability to render parts of the JSX tree only when certain condition is true.

Revisiting the `RenderArray` example component from the previous section, see how it can be extended to show a message when the array has no elements to render:

```
export default function RenderArray() {
  const data = [];

  return (
    <>
      <ul>{data.map(element => <li>{element}</li>)}</ul>
      {data.length === 0 &&
        <p>There is nothing to show here.</p>
      }
    </>
  );
}
```

Here the "and" operator is used to create an expression that will only include the JSX contents on the right side of the && if the condition on its left is true.

The above example is not perfect, because when the list is empty an empty <ul> element is rendered to the page before the error message. The && operator maps nicely to an if-then construct, but in this situation it would be ideal to be able to use an if-then-else, which can be implemented similarly, but using the ternary conditional operator (`?:`).

```
export default function RenderArray() {
  const data = [];

  return (
    <>
      {data.length === 0 ?
        <p>There is nothing to show here.</p>
      :
        <ul>{data.map(element => <li>{element}</li>)}</ul>
```

(continues on next page)

(continued from previous page)

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

The same idea can be implemented in the Microblog application. Replace the contents of *src/App.js* with the code below.

Listing 2.9: *src/App.js*: Render a list of blog posts with empty warning

```
export default function App() {
  const posts = [
    {
      id: 1,
      text: 'Hello, world!',
      timestamp: 'a minute ago',
      author: {
        username: 'susan',
      },
    },
    {
      id: 2,
      text: 'Second post',
      timestamp: 'an hour ago',
      author: {
        username: 'john',
      },
    },
  ];

  return (
    <>
      <h1>Microblog</h1>
      {posts.length === 0 ?
        <p>There are no blog posts.</p>
      :
        posts.map(post => {
          return (
            <p key={post.id}>
              <b>{post.author.username}</b> &mdash; {post.timestamp}
              <br />
              {post.text}
            </p>
          );
        })
      }
    </>
  );
}
```

With this version of the application you can comment out the `posts` constant and put an empty array in its place, and the page will automatically change to show the message in the "else" part, as seen in Figure 2.5.

Once you confirm that the conditional logic is working, remember to restore the fake blog posts.

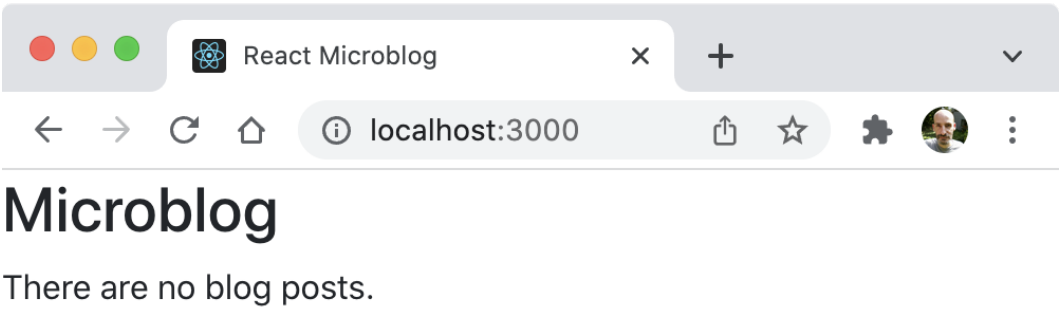


Figure 2.5: Render a warning that the post list is empty

2.7. Chapter Summary

- To start a new React project easily, use Create React App[12] (CRA).
- Remove the cruft added by CRA before you begin coding your application.
- A React component is a JavaScript function that renders a JSX tree and returns it.
- Insert JavaScript expressions in your JSX by enclosing them in curly brackets.
- Render lists of elements with `map()`, and include a `key` attribute with a unique value per element.
- Add conditional rendering expressions with the `&&` (if-then) and `?:` (if-then-else) operators.

[12] https://create-react-app.dev/

3. Working with Components

In *Chapter 2*, you wrote your first React component. In this chapter you will delve deeper into React as you learn how to create robust applications by combining and reusing components, not only your own but also some imported from third-party libraries.

3.1. User Interface Components

Creating a great user interface for the browser is not an easy task. Styling HTML elements with CSS requires a lot of time and patience, which most people don't have.

As you might expect, a myriad of libraries and frameworks that provide nice looking user interface primitives exist. At the time I'm writing this, there are three leading user interface libraries for React:

- MUI (Material UI)[1]
- React-Bootstrap[2]
- Ant Design[3]

I evaluated them and settled on using React-Bootstrap for this book, because it is the most straightforward of the three to learn and use.

React-Bootstrap is a library that provides React component wrappers for Bootstrap[4], a very popular CSS framework for the browser. You have imported Bootstrap's CSS file in *src/index.js* in *Chapter 2*, so some of its default styles are already in use. Now it is time to start actively using Bootstrap elements through the components provided by React-Bootstrap.

The React-Bootstrap library provides *grids* and *stacks* as the building blocks to help you create the layout of your website. Grids use the `Container` component (and optionally also `Row` and `Col`) to organize subcomponents. Stacks use the `Stack` component to render its subcomponents vertically or horizontally within their allocated space on the page. You will use the `Container` and `Stack` components later in this chapter.

[1] https://mui.com/
[2] https://react-bootstrap.github.io/
[3] https://ant.design/
[4] https://getbootstrap.com/

These two primitives may seem too simple to create complex layouts, but their power comes from their ability to be embedded recursively, as you will soon see.

3.2. The Container Component

Go back to the Microblog application you left running in the browser. You may have noticed that the text of the <h1> heading and the two fake blog posts are stuck to the left border of the window, without any margin, and this does not look good. The `Container` component, which is the main part of React-Bootstrap's grid system, addresses this by adding a small margin around all its children components. The next task is to add a top-level container to the application.

The component's function returns an <h1> element and a list of <p> paragraphs with the contents of the two made up blog posts. All these elements were grouped into a fragment, with the <> and </> tags, because React requires that components return a JSX tree with a single root node. The fragment tags can now be replaced with a <Container> component, which will be the root node of the tree.

Open *src/App.js* in your editor, add an import statement for the `Container` component, and then replace the fragment tags with it:

Listing 3.1: *src/App.js*: Add a container wrapper to the application

```
import Container from 'react-bootstrap/Container';

export default function App() {
  const posts = [
    ...  // <-- no changes to fake blog posts
  ];

  return (
    <Container fluid className="App">
      ...  // <-- no changes to JSX content
    </Container>
  );
}
```

Components in React-Bootstrap can be imported individually, using the format `import X from 'react-bootstrap/X'`. It may feel tedious to import each component you need individually, but importing the entire library in a single import is discouraged, because that inflates the size of the application considerably.

The `Container` component has a `fluid` attribute. A fluid container automatically changes its width to fill the browser window. Without the fluid option, the width of the container "snaps" to one of a few predefined widths associated with standard device screen sizes.

The `className` attribute is the equivalent of `class` in plain HTML. The name had to be changed to avoid a conflict with the `class` keyword from JavaScript. You can use `className` to provide a CSS class for any primitive HTML element, but many components also implement this attribute and assign it to the top-level element they render. Giving the component a class name is useful to later

be able to customize its appearance with CSS. As a naming convention to keep the CSS styles well organized, the name of the component is used as the CSS class.

Save the changes and note how the `Container` component adds a nice margin to the page. Also try resizing the browser window to see how the container resizes with it. You may notice that the font sizes slightly increase as you make the window larger. Bootstrap styles elements of the page differently for different screen sizes, a technique that helps make websites look their best on phones, tablets, laptops and desktop computers. Figure 3.1 shows how the fluid container looks.

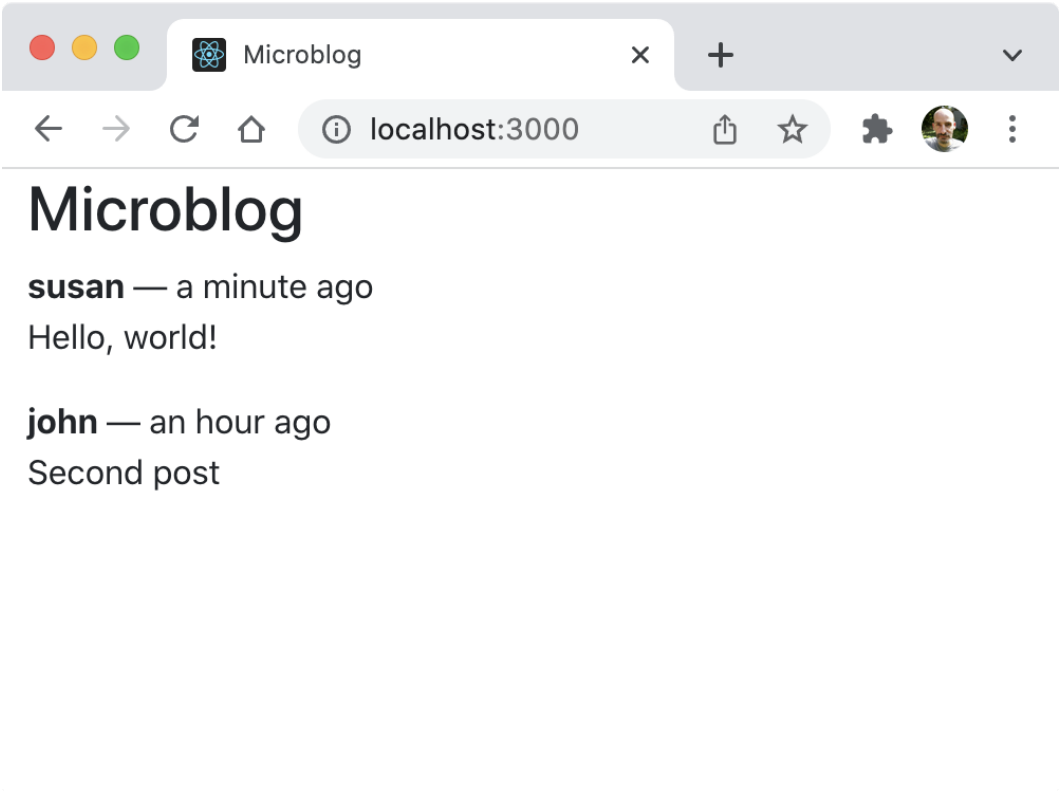


Figure 3.1: Fluid container from React-Bootstrap

An interesting experiment you can do is to look at the structure of the page in your browser's debugging console. Note how the `Container` component renders itself as a `<div>` element with the `App` class name (in addition to other Bootstrap-specific classes).

For more examples and information about the `Container` component, consult its documentation[5]. Do not worry if the difference between fluid and non-fluid containers isn't very clear yet. In the next

[5] https://react-bootstrap.github.io/docs/layout/grid/#container