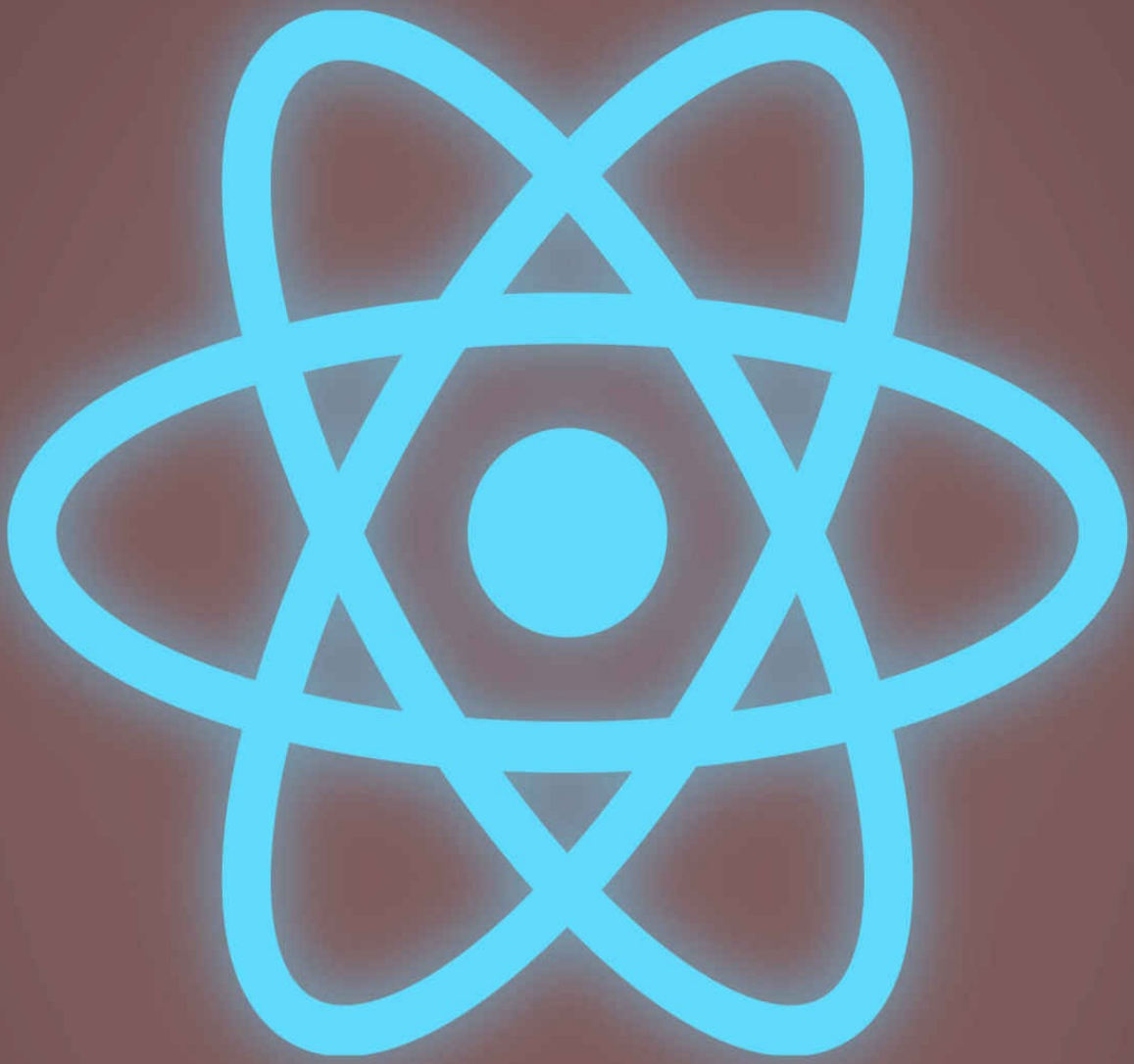
The React Mega-Tutorial



Miguel Grinberg

The React Mega-Tutorial

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 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), 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, Visual Studio Code 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 or Firefox, because they are the two browsers that support the React Developer Tools 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. My patrons have been a deciding factor in writing this book, through their questions, suggestions and ideas. In particular, I enjoyed the many discussions we had on best practices around user authentication, which helped me give this topic the importance it deserves in the book.

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.

ES5 vs. ES6

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

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, 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.

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.

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.

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',
  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.

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 and export sections in the JavaScript reference.

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:

```
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 and const.

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 and Strict inequality operators in the JavaScript reference documentation for more details.

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 reference documentation.

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);
}
```

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:

```
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 documentation for more information.

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 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.

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}`);
  }
}
```

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 section of the JavaScript documentation is a good place to learn more.

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:

```
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 reference for more details.

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,
```

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.

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 section of the JavaScript reference.

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;
    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 in the JavaScript reference.

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, 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>
      <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.

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!

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 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 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
```

Creating a Starter React Project

There are many ways to create a new React application. The React documentation recommends the Create React App 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 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 1 shows what you should see in your browser.

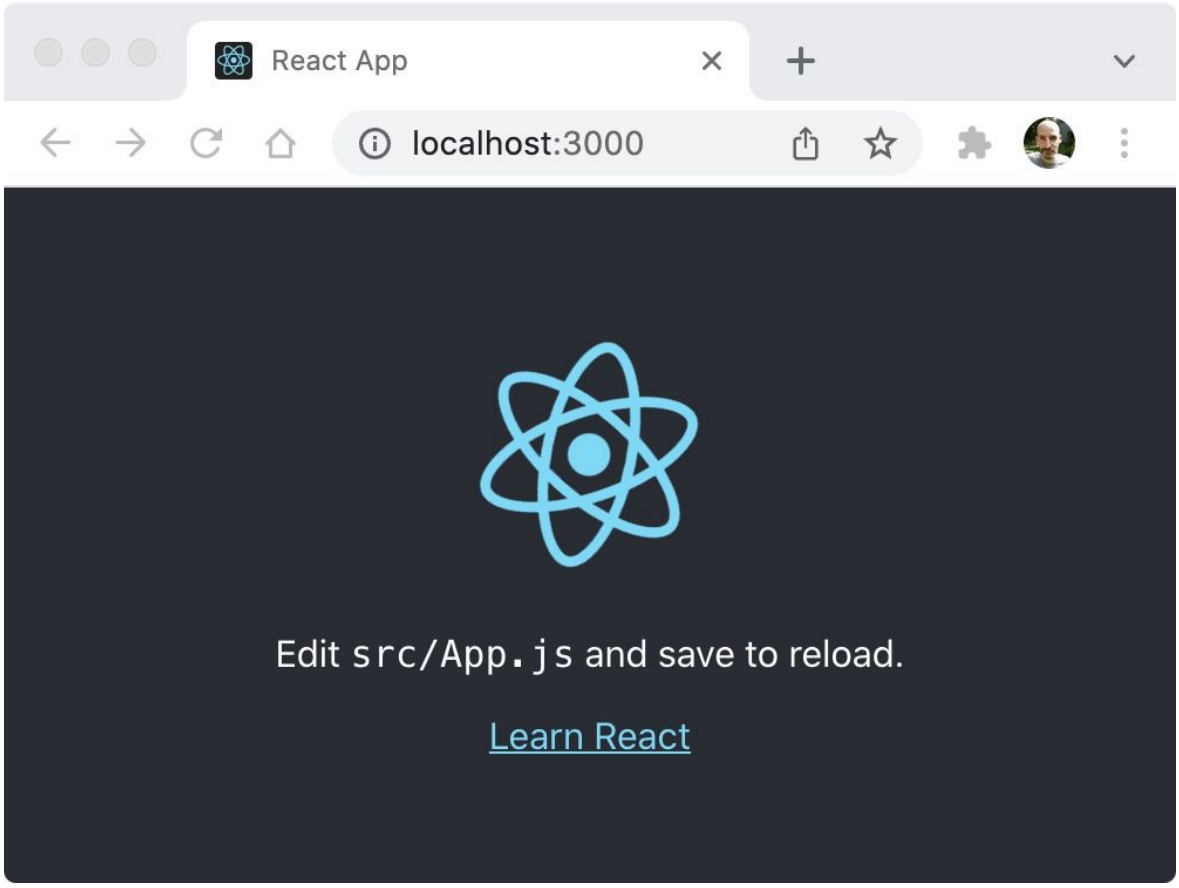


Figure 1 React starter application

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.

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: a CSS user interface library for web pages.
- react-bootstrap: a React component library wrapper for the `bootstrap` package.
- react-router-dom: a React component library that implements client-side routing.
- serve: 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.

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.

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.

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 1 public/index.html: Updated meta description tag

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

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, followed by this title.

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

Listing 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:

```
    <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.

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 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 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.

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.

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>,
);
```

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 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 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.

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.

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 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.

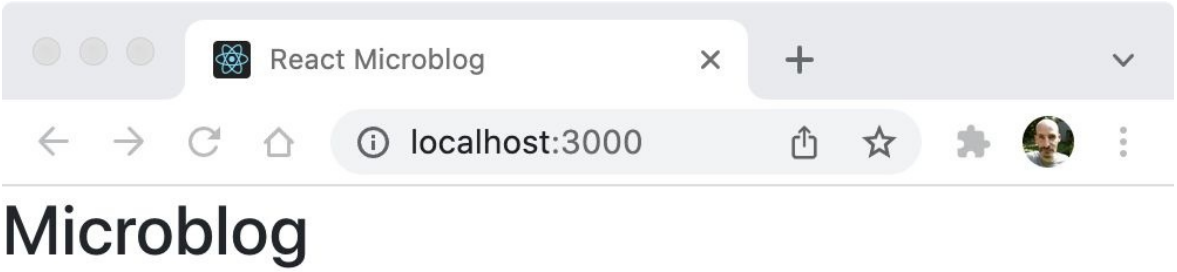


Figure 2 Basic application

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.

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.

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.

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 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 3 shows how the blog post looks on the browser.

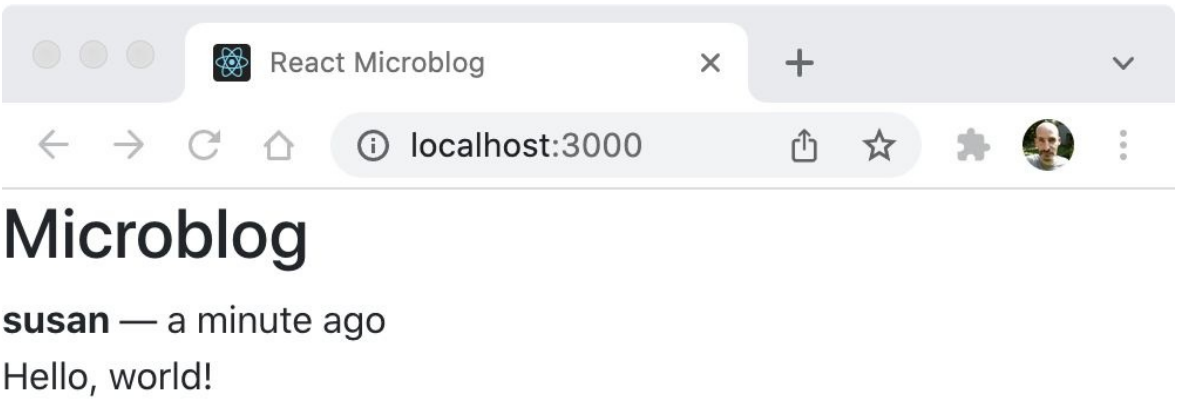


Figure 3 Render a blog post

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 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 4.

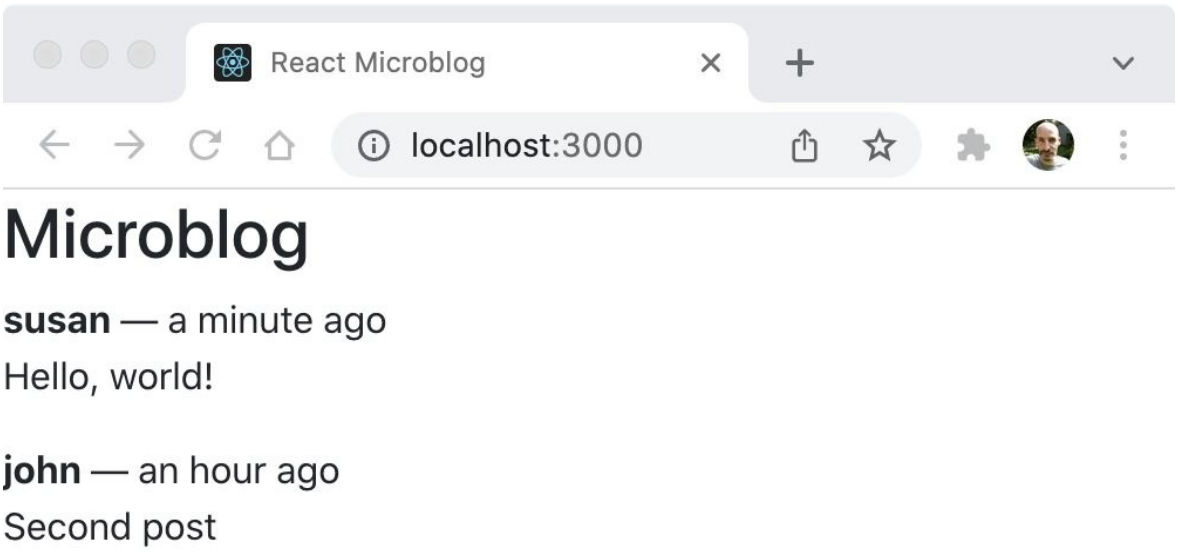


Figure 4 Render a list of blog posts

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 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 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>
        );
      })}
```

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>
      }
    </>
  );
}
```

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

Listing 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.timestam
              <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 5.

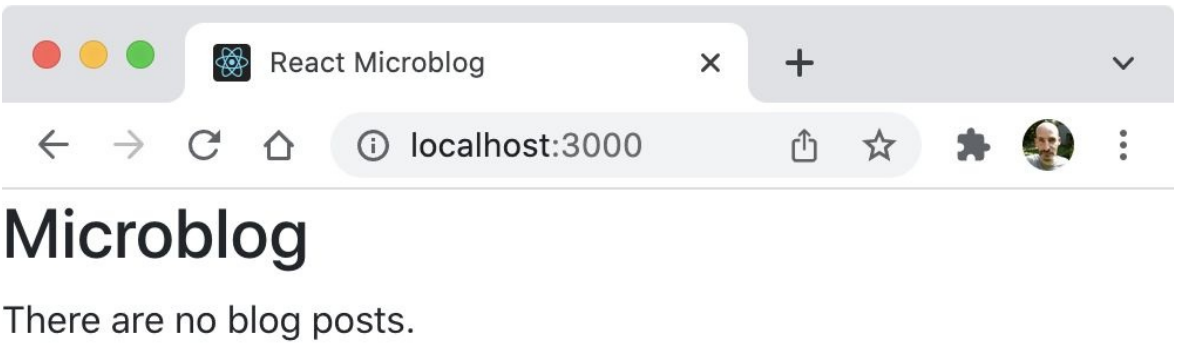


Figure 5 Render a warning that the post list is empty

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

Chapter Summary

- To start a new React project easily, use Create React App (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.

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.

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)
- React-Bootstrap
- Ant Design

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, 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.

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.

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 10 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 6 shows how the fluid container looks.

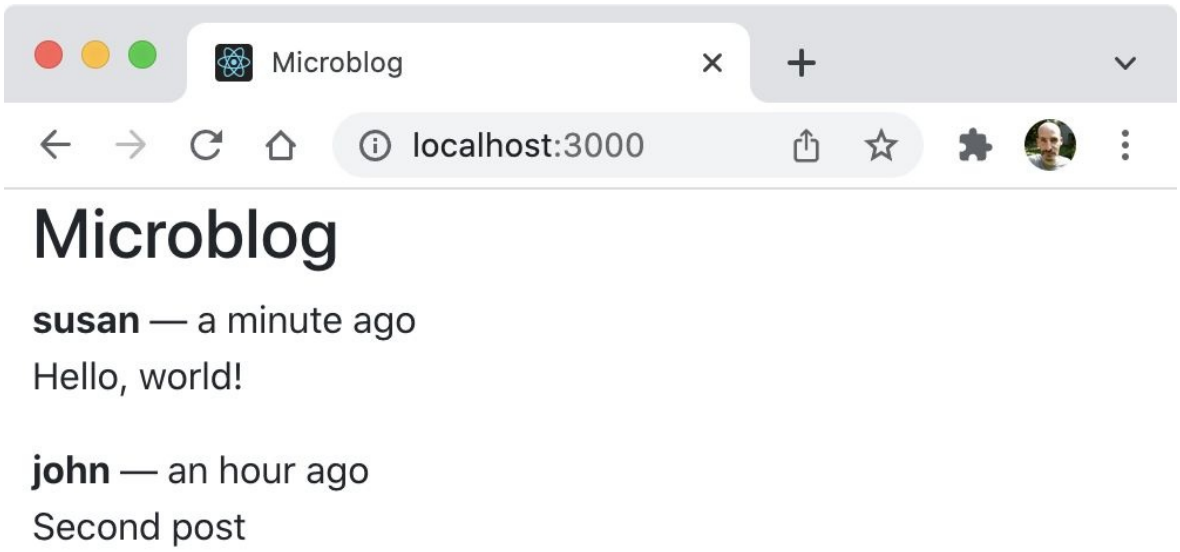


Figure 6 Fluid container from React-Bootstrap

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. Do not worry if the difference between fluid and non-fluid containers isn't very clear yet. In the next section you will be adding non-fluid containers into the layout, and then the difference will be more evident.

Adding a Header Component

The `<h1>` element with the application's name was a good first attempt at a header for the application, but of course this application needs something more polished, like a navigation bar that can eventually hold menu options.

To keep the application's source code well organized, it is a good idea to create a custom component for this header in a separate module.

The project as created by Create React App does not provide any guidance on how to structure the source code. I find it useful to put all the custom components of the application in subdirectory dedicated to them. Create this directory from your terminal:

```
mkdir src/components
```

React-Bootstrap comes with a `Navbar` component that is a perfect fit for the Microblog header. Its documentation has a number of examples, which makes it easy to find a design that fits the needs of the application.

Below you can see the `Header` component, which uses `Navbar` and `Container` from React-Bootstrap. Copy this code into *src/components/Header.js*.

Listing 11 *src/components/Header.js*: Header component

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

export default function Header() {
  return (
    <Navbar bg="light" sticky="top" className="Header">
      <Container>
        <Navbar.Brand>Microblog</Navbar.Brand>
      </Container>
    </Navbar>
  );
}
```

The component starts by importing any other components that are needed, in this case `Navbar` and `Container`. The component function is declared with `export default function`, because this component is going to be imported from *App.js* so that it can be included in the page.

The component's function has a `return` statement that returns a `Navbar` component, with options that were copied from one of the examples in the documentation. I have decided to use a light background, and I also thought it would be a good idea to make this bar "sticky", which means that when there is a

need to scroll, the bar will always stay visible at the top of the page. I gave this component a class name, so that I can then add custom styles for it.

As seen in the `Navbar` documentation examples, a `Container` component is defined as a child. A `Navbar.Brand` component with the application's name is the only thing inside the container for now. React-Bootstrap often groups related subcomponents and makes them attributes of the parent, so you will find many instances where a component uses the dot syntax, as in `Navbar.Brand`.

This header component shows how layout primitives from React-Bootstrap can be combined and embedded within each other. When this component is added to the page, there's going to be a fluid container as a root element, the navigation bar as its child, and a second container that is not fluid as the child of the bar. This is a good pattern for many applications, because it makes the navbar full width, but its contents are restricted to a narrower, more pleasant width that is chosen according to the screen or window size. The contents of the page will also be wrapped in a non-fluid container so that they align with contents of the header.

Now that the `Header` component is in the project, it can be imported and used in the `App` component in place of the `<h1>` heading. A second inner `Container` for the blog posts is added as well.

Listing 12 *src/App.js*: Add header component

```
import Container from 'react-bootstrap/Container';
import Header from './components/Header';

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

  return (
    <Container fluid className="App">
      <Header />
      <Container>
        {posts.length === 0 ?
          ...
        }
      </Container>
```

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

The `Header` component is imported from its source file, which is given as a relative path. Then this component is used instead of the `<h1>` element. Since the component has all the information that it needs to render itself to the page, there is no need to pass any arguments.

As mentioned above, there is now a `Container` component that wraps the loop that renders the fake blog posts. This is so that the blog posts also use non-fluid positioning, and are aligned with the contents of the header. Figure 7 shows how the page looks with these changes.

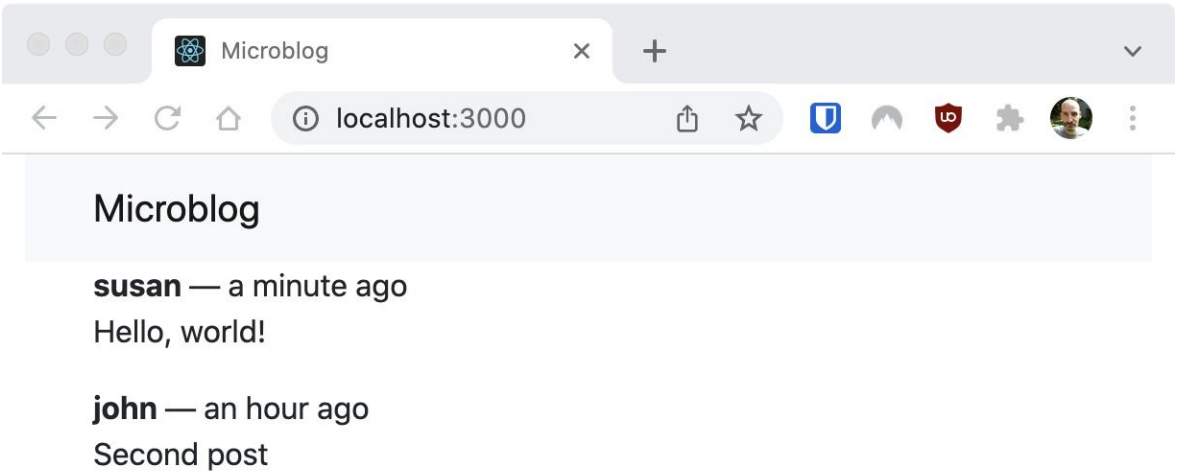


Figure 7 Header component

If you look at the header carefully, you will notice that it does not extend all the way to the left and right borders of the window, there is actually a small white margin on each side. Looking at the styles in the page with the browser's inspector I determined that the top-level `<div>`, which is rendered by the fluid `Container`, has non-zero padding. The solution to address this minor cosmetic annoyance is to override the padding for this component, which has the `App` class

name.

Another aspect of the styling of the header that I don't quite like is that there is no separation between the bottom of the header and the content area below it. A slightly darker border line there would make the separation more clearly visible.

The *src/index.css* file is where you will enter all the custom styles needed by the application. Open this file in your editor and replace all of its contents, which were added by Create React App and are not useful to this application, with the following:

Listing 13 *src/index.css*: Custom styles for App and Header components

```
.App {
  padding: 0;
}

.Header {
  border-bottom: 1px solid #ddd;
}
```

With these minor style overrides, the application looks much better, as you can see in Figure 8.

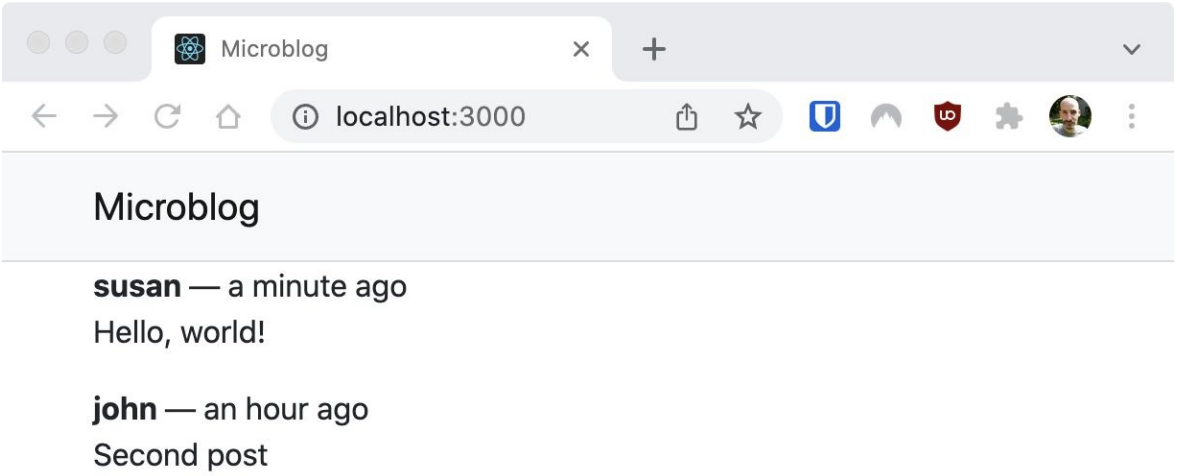


Figure 8 Styled Header component

Adding a Sidebar

Another common user interface component in many web applications is a sidebar. In Microblog, the sidebar will offer navigation links to switch between the "feed" page, which shows all the blog posts from followed users, and the "explore" page, which shows blog posts from all users.

Looking through the list of React-Bootstrap components, the same Navbar used for the `Header` component can work as a sidebar, with some small CSS customizations. And the Nav.Link component can be used for the navigation links.

Here is a first implementation of the sidebar, with placeholder links that will not work properly until page routing is implemented. Add this code to a *src/components/Sidebar.js* file.

Listing 14 *src/components/Sidebar.js*: A sidebar component

```
import Navbar from "react-bootstrap/Navbar";
```

```
import Nav from "react-bootstrap/Nav";

export default function Sidebar() {
  return (
    <Navbar sticky="top" className="flex-column Sidebar">
      <Nav.Item>
        <Nav.Link href="/">Feed</Nav.Link>
      </Nav.Item>
      <Nav.Item>
        <Nav.Link href="/explore">Explore</Nav.Link>
      </Nav.Item>
    </Navbar>
  );
}
```

The `sticky="top"` attribute of this `Navbar` component will keep the sidebar visible on the page as the user scrolls down. The flex-column class comes from the Bootstrap framework, and has the purpose of changing the direction of its children to vertical. The `Sidebar` class name is for the application to use when styling this component.

Now the sidebar needs to be added to the page, to the left of the content area. When needing to position two or more components side by side, the ideal layout tool is a horizontal stack. Below you can see the updated `App` component with a sidebar.

Listing 15 *src/App.js*: Add a sidebar

```
import Container from 'react-bootstrap/Container';
import Stack from 'react-bootstrap/Stack';
import Header from './components/Header';
import Sidebar from './components/Sidebar';

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

  return (
    <Container fluid className="App">
      <Header />
      <Container>
```

```
      <Stack direction="horizontal">
        <Sidebar />
        <Container>
          {posts.length === 0 ?
            ...  // <-- no changes to render loop
          }
        </Container>
      </Stack>
    </Container>
  </Container>
);
```

The `Stack` component was added with the `direction` attribute set to `horizontal`, which is necessary because the default for this component is to lay components out vertically. The stack has two children, the sidebar, and an inner `Container` with the fake blog posts. These two components will now appear side by side, as you see in Figure 9.

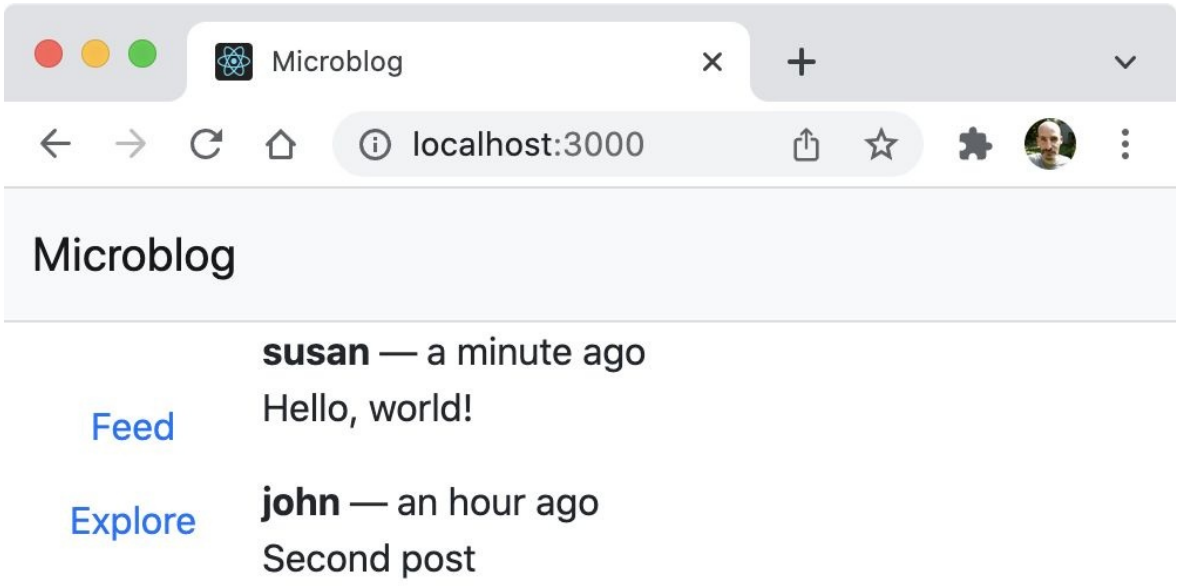


Figure 9 Sidebar

The sidebar needs some styling work to look its best. Add the following CSS definitions to *src/index.css*.

Listing 16 *src/index.css*: Sidebar styles

```
... // <-- no changes to existing styles

.Sidebar {
  width: 120px;
  margin: 5px;
  position: sticky;
  top: 62px;
  align-self: flex-start;
  align-items: start;
}

.Sidebar .nav-item {
  width: 100%;
}

.Sidebar a {
  color: #444;
}

.Sidebar a:hover {
  background-color: #eee;
}

.Sidebar a:visited {
  color: #444;
}
```

The definitions added to the `Sidebar` CSS class are a result of experimenting in the browser's debugging console. Here is a brief description of each rule:

- `width: 120px` sets the width of the sidebar to 120 pixels
- `margin 5px` adds a 5 pixel margin around the sidebar
- `position: sticky` attaches the sidebar to the left side of the browser, so that it stays there when the user scrolls the content
- `top: 62px` sets the correct vertical position for the sidebar with respect to the header
- `align-self: flex-start` aligns the sidebar with the top border of the

stack component
- `align-items: start` aligns the children components of the sidebar to the left

You may be wondering where does the `.nav-item` class come from in the above CSS definitions. This is a class defined by the Bootstrap library, and used by the `Nav.Link` component from React-Bootstrap. As mentioned above, opening the browser's developer console and looking at the rendered elements on the page is often the easiest way to find what classes are used and are potential targets for redefining the look of some elements. The CSS definition for `.nav-item` sets the width of the `<Nav.Item>` elements to 100%, which means that they will have the maximum width of the sidebar instead of the width of their text. This is done so that the hover style, which alters the background, shows a full bar regardless of the length of the text in the link.

The CSS definitions for `a`, `a:hover` and `a:visited` configure the colors for the links.

With the styling updates, the sidebar looks much better. If you hover the mouse pointer over an item, its background color changes to highlight it. See the current state of the sidebar in Figure 10.

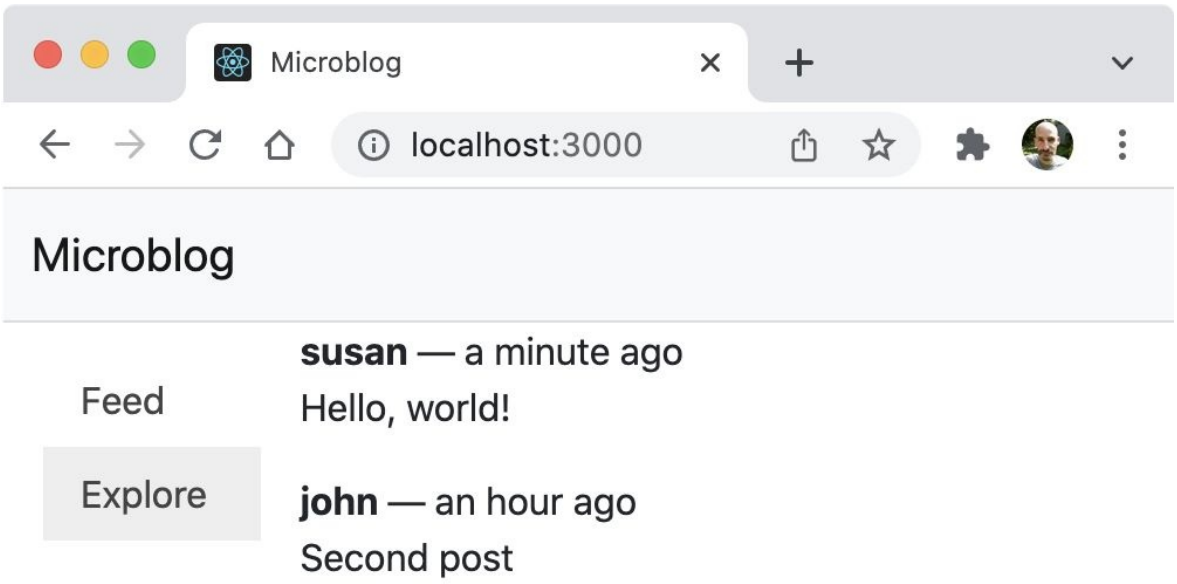


Figure 10 Styled sidebar

Building Reusable Components

A good strategy when building applications with React is to always try to partition the application into many components, each with only one purpose. The `Header` and `Sidebar` components from previous sections in this chapter are great models to follow.

The `App` component is an example of a component that is doing too much work on its own, as it is currently in charge of rendering the general layout of the application, and also the fake representation of what later is going to be the blog feed.

To prepare the application to support page navigation, it makes sense to refactor the `App` component so that the content area is rendered by a subcomponent that can be swapped out as the user navigates through different pages.

The following listing shows a new component called `Posts`, with the logic that renders the (currently fake) blog posts. Store this code in *src/components/Posts.js*.

Listing 17 src/components/Posts.js: Render a list of blog posts

```
export default function Posts() {
  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 (
    <>
      {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.timestam
              <br />
              {post.text}
            </p>
          );
        })
      }
    </>
  );
```

```
}
```

With this new component in the project, the `posts` fake blog post array in `App` can be removed, and the loop that renders it can be replaced with `<Posts />`. Here is the updated version of `App`:

Listing 18 src/App.js: Use the `Posts` component

```
import Container from 'react-bootstrap/Container';
import Stack from 'react-bootstrap/Stack';
import Header from './components/Header';
import Sidebar from './components/Sidebar';
import Posts from './components/Posts';

export default function App() {
  return (
    <Container fluid className="App">
      <Header />
      <Container>
        <Stack direction="horizontal">
          <Sidebar />
          <Container>
            <Posts />
          </Container>
        </Stack>
      </Container>
    </Container>
  );
}
```

Components with Props

The `App` component is still not very flexible. You can envision that once multiple pages are supported, the `<Posts />` component is going to be one of many possible options to include in the content area of the page, but with this structure a sidebar would always appear to the left of the content section. The problem is that for this project, the sidebar is only useful after the user logs in.

In this application there's going to be some situations in which the sidebar needs to be omitted, such as when rendering a login page. Since the goal is to keep `App` as simple as possible, this is another opportunity to move logic down into a new

subcomponent.

This new component, which I'm going to call `Body`, has to be very generic enough to be able to render the main page content with or without a sidebar. It is the first in this application that needs to accept input arguments, which in React are called *props*.

Before I show you how to write this component, take a look at a couple of examples of this component in use. Here is how the feed page of this application could render the list of blog posts, with the sidebar on the left:

```
<Body sidebar={true}>
  <Posts />
</Body>
```

Nice, right? To indicate whether the page needs to show a sidebar or not, a `sidebar` attribute is given with a boolean value. The contents of the page are given as children of the component, in this case just the `Posts` component. For a slightly more compact format, you can omit the `true` value for the `sidebar` prop, as it is the default:

```
<Body sidebar>
  <Posts />
</Body>
```

Implementing a login page using this same `Body` component could be done as follows:

```
<Body sidebar={false}>
  <h1>Login</h1>
  <form>
    ...
  </form>
</Body>
```

Or in a more compact way, you can omit the `sidebar` prop altogether, which would make it default to a falsy value:

```
<Body>
  <h1>Login</h1>
  <form>
    ...
```

```
  </form>
</Body>
```

This is extremely powerful, because the `Body` component becomes the absolute authority on how to format the body of the page, with or without sidebar. If you decide to change the layout of the application in some way, like maybe moving the sidebar to the right side, there is only one place in the entire application where the change needs to be made.

How does a component function access props that were passed as arguments and any subcomponents defined as children? React makes this very easy, because it passes an object with all these attributes as an argument when it calls the component function. The `Body()` component function can be declared as follows:

```
export default function Body(props) {
  // props.sidebar is the value of the sidebar attribute
  // props.children is the JSX component tree parented by this compo
}
```

The `props` object passed into the function includes keys for all the attributes that were given in the component declaration as props. And if the component was declared with children, then a `children` key is included as well.

In practice, you will find that most React developers use destructuring assignments (see Chapter 1) to receive props. The next example is functionally equivalent to the one above:

```
export default function Body({ sidebar, children }) {
  // sidebar is the value of the sidebar attribute
  // children is the JSX component tree parented by this component
}
```

The benefit of this syntax is that the component's function declaration explicitly names its input arguments.

Ready to implement your first non-trivial component? Here is the code for `Body`, which goes in *src/components/Body.js*.

Listing 19 src/components/Body.js: A body component

```
import Container from 'react-bootstrap/Container';
import Stack from 'react-bootstrap/Stack';
import Sidebar from './Sidebar';

export default function Body({ sidebar, children }) {
  return (
    <Container>
      <Stack direction="horizontal" className="Body">
        {sidebar && <Sidebar />}
        <Container className="Content">
          {children}
        </Container>
      </Stack>
    </Container>
  );
}
```

The JSX hierarchy is the same as it was in `App`. The parent `Container` is not fluid, and its purpose is to align the body of the page with the non-fluid container that exists in the header. The `Sidebar` component is now added inside a conditional, only when the `sidebar` prop has a truthy value. The second (or only, if `sidebar === false`) child is an inner `Container` with the children of the component, which represent the main content of the page. The `Stack` component is assigned a class name of `Body`, to help add styles as necessary. The inner `Container` is given the name `Content` for the same reason.

With the addition of `Body`, the `App` component can be simplified even more. The updated version is below.

Listing 20 src/App.js: Refactored application component

```
import Container from 'react-bootstrap/Container';
import Header from './components/Header';
import Body from './components/Body';
import Posts from './components/Posts';

export default function App() {
  return (
    <Container fluid className="App">
      <Header />
      <Body sidebar>
        <Posts />
```

```
      </Body>
    </Container>
  );
}
```

You are not going to see any changes to how the application looks in the browser, but this is a very robust and scalable refactor that is ready to be expanded to support multiple pages and the routing between them.

Chapter Summary

- To avoid reinventing the wheel, use a user interface component library such as React-Bootstrap.
- Add a top-level container component that provides sensible margins on all screen sizes.
- To keep your code better organized, create a subdirectory for application components.
- To maximize code reuse, do not add too much to a single component, and instead split the work across several components, each having a single purpose.
- Use props to create components that are reusable.

Routing and Page Navigation

React is a Single-Page Application (SPA) framework, which means that from the point of view of the browser, only one web page is ever downloaded. Once that page is active, all the application state changes will happen through JavaScript events, without the browser having to fetch new web pages from the server. How then, can the application support page navigation?

The React Router package, which you installed in Chapter 2, implements a complete page navigation system for SPAs. In this chapter you will learn how to create client-side routes and navigate between them.

Creating Page Components

The concept of a page, in the strict browser sense, does not really apply in an SPA, since SPAs only have one page. In an SPA, pages are just top-level application states that dictate how the application renders in the browser. As with traditional pages, each page in an SPA can be associated with paths such as */login* or */user/susan*, but these paths are managed by the client application and never reach the server.

The React-Router package keeps track of these page states, and automatically updates the address bar of the browser with the appropriate URL path. It also takes control of the Back and Forward buttons of the browser and makes them work as the user expects.

To help have a sane application structure, the top-level components that map to the logical pages of the application will be written in a separate directory, called *pages*. Create this directory now:

```
mkdir src/pages
```

The default page for this application is going to be the page that displays the post feed for the user. Let's move the `Body` component, which is currently in `App`, to a new `FeedPage` component, stored in *src/pages/FeedPage.js*.

Listing 21 *src/pages/FeedPage.js*: the Feed page

```
import Body from '../components/Body';
import Posts from '../components/Posts';

export default function FeedPage() {
  return (
    <Body sidebar>
      <Posts />
    </Body>
  );
}
```

Note how the import path for components now uses `../components/`. This is because the path is relative to the location of the importing source file, which is in *src/pages*.

The second most important page in this application is going to be the Explore page, which will display blog posts from all the users in the system, and is intended as the place where users can discover other users to follow. Add a placeholder for this page in the `ExplorePage` component:

Listing 22 *src/pages/ExplorePage.js*: a placeholder for the explore page

```
import Body from '../components/Body';

export default function ExplorePage() {
  return (
    <Body sidebar>
      <h1>Explore</h1>
      <p>TODO</p>
    </Body>
  );
}
```

Even though the application isn't ready to have a login page yet, also create a placeholder for the `LoginPage` component, so that you can later test navigation between three different pages.

Listing 23 *src/pages/LoginPage.js*: a placeholder for the login page

```
import Body from '../components/Body';

export default function LoginPage() {
  return (
    <Body>
      <h1>Login form</h1>
      <p>TODO</p>
    </Body>
  );
}
```

With the help of the React-Router package, the `App` component can now implement routing for these three pages. Below is the new version of `App`, using several new routing components.

Listing 24 *src/App.js*: Page routing

```
import Container from 'react-bootstrap/Container';
import { BrowserRouter, Routes, Route, Navigate } from 'react-rout
import Header from './components/Header';
import FeedPage from './pages/FeedPage';
import ExplorePage from './pages/ExplorePage';
import LoginPage from './pages/LoginPage';

export default function App() {
  return (
    <Container fluid className="App">
      <BrowserRouter>
        <Header />
        <Routes>
          <Route path="/" element={<FeedPage />} />
          <Route path="/explore" element={<ExplorePage />} />
          <Route path="/login" element={<LoginPage />} />
          <Route path="*" element={<Navigate to="/" />} />
        </Routes>
      </BrowserRouter>
    </Container>
  );
}
```

The React-Router library is called `react-router-dom`. There are four components provided by this library that are used above.

The BrowserRouter component adds routing support to the application. This component must be added very high in the component hierarchy, as it must be a parent to all the routing logic in the application. In terms of rendering, this component transparently renders its children without rendering anything itself, so it can be added conveniently as a parent near the top of the JSX tree.

Routes is a component that needs to be inserted in the place in the component tree where the contents need to change based on the current page. As an analogy, you can think of `Routes` as a routing equivalent to the `switch` statement in JavaScript, or a long chain or `if-then-else` statements.

Route is used to define a route inside the `Routes` component. The `path` attribute defines the path portion of the URL for the route, and the `element` attribute specifies what contents are associated with the route. Following the switch statement analogy, this component is equivalent to the `case` statement.

Navigate is a special component that allows to redirect from one route to another. The fourth `Route` component in the listing above has a path set to `*`, which works as a catch-all route for any URLs that are not matched by the routes declared above it. The `element` attribute in this route uses `Navigate` to redirect all these unknown URLs to the root URL.

With this version of the application, the address bar, Back and Forward buttons of your browser are connected to the application, and routing is functional. If you type *http://localhost:3000/login* in the address bar of your browser, the application will load and automatically render the login page placeholder added earlier. The same will occur if you use */explore* in the path portion of the URL.

The two links in the sidebar are standard browser links that are not connected to React-Router yet, so they trigger a full page reload. Ignoring the inefficiency this causes (which will soon be fixed), they should allow you to switch between the feed and explore pages.

After you've moved between the three pages a few times, try the Back and Forward buttons in your browser to see how React-Router makes the SPA behave like a normal multipage website.

Implementing Links

As noted above, the two links in the sidebar are working in the sense that they allow you to navigate between pages, but they are inefficient, because they are standard browser links that reload the entire React application every time they are clicked. For a single-page application, the routing between pages should be handled internally in JavaScript, without reaching the browser's own page navigation system.

The React-Router package provides the Link and NavLink components to aid in the generation of SPA-friendly links. The difference between them is that `Link` is just a regular link, while `NavLink` extends the behavior of a link with the ability to become "active" when its URL matches the current page, allowing the application to change the styling. In a navigation bar such as the `Sidebar` component, `NavLink` makes the most sense to use, because the currently active page can be highlighted.

The existing links in the sidebar use the Nav.Link component from React-Bootstrap. This component has a very similar name, except for the dot between `Nav` and `Link`, but these two components are completely different. In the current version of the sidebar, a link is created with the following syntax:

```
<Nav.Link href="/">Feed</Nav.Link>
```

How can this be combined with the `NavLink` component from React-Router to generate an SPA link? React-Bootstrap recognizes that many times its components need to be integrated with components from other libraries, so `Nav.Link` (as well as many others), have the `as` attribute to specify a different base component. This makes it possible to use a React-Bootstrap component as a wrapper to a component from another library such as React-Router.

Here is how the above link can be adapted to work with React-Router's `NavLink`, while still being compatible with Bootstrap:

```
<Nav.Link as={NavLink} to="/">Feed</Nav.Link>
```

Note how the `href` attribute of `Nav.Link` has been renamed to `to`, because now this component will render as React-Router's `NavLink`, which uses `to` instead of `href`. With this solution, the Bootstrap CSS classes associated with navigation links are preserved, but they are applied to the React-Router `NavLink` component.

Here is the React-Router version of `Sidebar`:

Listing 25 *src/components/Sidebar.js*: React-Router enabled sidebar

```
import Navbar from "react-bootstrap/Navbar";
import Nav from "react-bootstrap/Nav";
import { NavLink } from 'react-router-dom';

export default function Sidebar() {
  return (
    <Navbar sticky="top" className="flex-column Sidebar">
      <Nav.Item>
        <Nav.Link as={NavLink} to="/">Feed</Nav.Link>
      </Nav.Item>
      <Nav.Item>
        <Nav.Link as={NavLink} to="/explore">Explore</Nav.Link>
      </Nav.Item>
    </Navbar>
  );
}
```

The `NavLink` component recognizes when the current page URL maintained by React-Router matches its own link address, and in that case it considers the link "active". For an active links, it adds the `active` class name to its `<a>` element. This class name can be used in *src/index.css* to create new style for the active link. Add the following class definition at the bottom of *index.css*:

Listing 26 *src/index.css*: Active page style for navigation links

```
... // <-- no changes to existing styles

.Sidebar .nav-item .active {
  background-color: #def;
}
```

With this new CSS definition, you can navigate between the feed and explore pages, and whichever of the two links is active renders with a light blue background color, as shown in Figure 11.

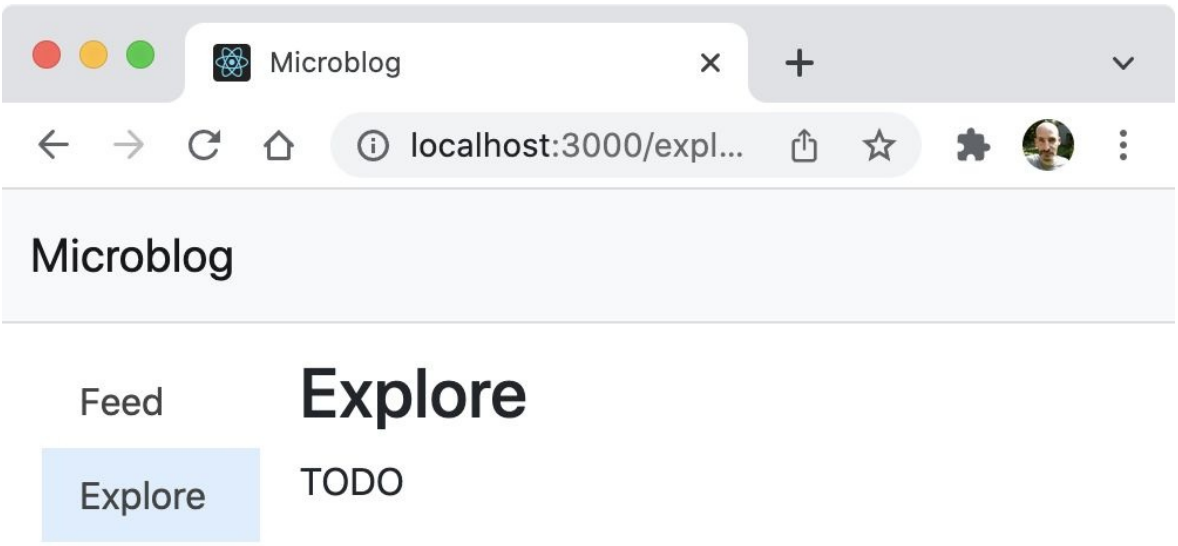


Figure 11 Navigation link styling

Pages with Dynamic Parameters

For most web applications, some pages need route URLs that have placeholders in sections of the path. Consider having a profile page for each user. The most convenient way to define the route for this page is to include the user ID or name in the path itself. In Microblog, the profile page for a given user is going to be */user/{username}*.

To define a route with a dynamic section, the `path` attribute of the `Route` component uses a special syntax with a colon prefix:

```
<Route path="/user/:username" element={<UserPage />} />
```

The `:` denotes that section of the path as a placeholder that matches any value. The component referenced by the `element` attribute or any of its children can use the useParams() function to access the dynamic parameters of the current URL

as an object.

This is the first time you encounter a function that starts with the word `use`. In React, "use" functions are called *hooks*. Hook functions are special in React because they provide access to application state. React includes a number of hooks, most of which you'll encounter in later chapters. Many libraries for React also provide their own hooks, such as the `useParams()` hook from React-Router. Applications can create custom hooks as well, and you will also learn about this later.

Let's add a simple version of the user profile page to the application, which for now will just show the username. This is going to be the `UserPage` component, stored in *src/pages/UserPage.js*.

Listing 27 src/pages/UserPage.js: a simple user profile page

```
import { useParams } from 'react-router-dom';
import Body from '../components/Body';

export default function UserPage() {
  const { username } = useParams();

  return (
    <Body sidebar>
      <h1>{username}</h1>
      <p>TODO</p>
    </Body>
  );
}
```

The listing below shows the updated `App` component with the user profile page.

Listing 28 src/App.js: User profile page with a dynamic parameter

```
import Container from 'react-bootstrap/Container';
import { BrowserRouter, Routes, Route, Navigate } from 'react-rout
import Header from './components/Header';
import FeedPage from './pages/FeedPage';
import ExplorePage from './pages/ExplorePage';
import UserPage from './pages/UserPage';
import LoginPage from './pages/LoginPage';
```

```
export default function App() {
  return (
    <Container fluid className="App">
      <BrowserRouter>
        <Header />
        <Routes>
          <Route path="/" element={<FeedPage />} />
          <Route path="/explore" element={<ExplorePage />} />
          <Route path="/user/:username" element={<UserPage />} />
          <Route path="/login" element={<LoginPage />} />
          <Route path="*" element={<Navigate to="/" />} />
        </Routes>
      </BrowserRouter>
    </Container>
  );
}
```

As with the login page, the new user profile page is currently not linked from any other part of the application, so the only way to view it is by typing a matching URL in the browser's address bar. Figure 12 shows the user page that corresponds to the *http://localhost:3000/user/susan* URL.

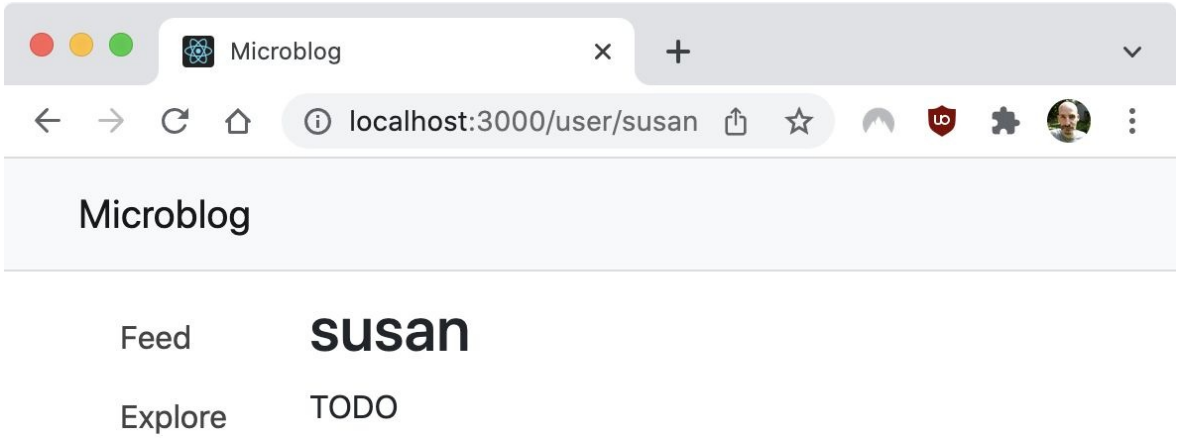


Figure 12 User profile page with a dynamic parameter

Chapter Summary

- Use React-Router to create client-side routes in your React application.
- Following on the idea of having nicely organized code, create a subdirectory for page-level components.
- Create each logical page of the application as a separate page-level component that can be defined as a route.

Connecting to a Back End

In this chapter you are going to learn how the React application can communicate with a back end application to request data. While doing this, you will learn how to use the two most important React hooks: `useState()` and `useEffect()`.

Running the Microblog API Back End

Starting in this chapter, you will need to run the project's companion back end application. This is an open source application called Microblog API that provides all the storage and authentication functionality needed by the React front end you are building.

Microblog API has a read me page that you can consult for detailed installation instructions, but the following sections cover the basic installation steps for three different installation methods.

Getting an Email Server

The Microblog API service needs access to an email server, to be used in a later chapter while implementing the reset password flow, which requires the server to send emails.

If you don't have an email service that you can use, my recommendation is that you open a SendGrid account. The free tier of this service includes 100 emails per day, which is more than enough for the needs of this project.

To configure the email service in Microblog API, you will need to provide values for the several configuration variables:

- `MAIL_SERVER`: The email server to use when sending emails.
- `MAIL_PORT`: The port in which the email server listens for connections.
- `MAIL_USE_TLS`: Set to any non-empty string to use an encrypted connection.
- `MAIL_USERNAME`: The username for the email sender's account.
- `MAIL_PASSWORD`: The password for the email sender's account.

- `MAIL_DEFAULT_SENDER`: The email address that appears in all emails sent by the application as the sender.

If you decide to use SendGrid, open an account, and then create an API key. The email settings for a SendGrid account are as follows:

```
MAIL_SERVER=smtp.sendgrid.net
MAIL_PORT=587
MAIL_USE_TLS=true
MAIL_USERNAME=apikey    # <-- this is the literal word "apikey"
MAIL_PASSWORD=          # <-- your SendGrid API key here
MAIL_DEFAULT_SENDER=    # <-- the sender email address you'd like to
```

As noted above in comments, the `MAIL_PASSWORD` variable needs to be set to your SendGrid API key. The `MAIL_DEFAULT_SENDER` can be set to any email address. This address is going to be in the `From` field of all emails sent by the service.

Installing the Back End

In this section, three methods of installation are described:

1. Run on your computer with Docker
2. Run on your computer with Python
3. Deploy to a free or paid Heroku account

Which is the best method? If you are familiar with Docker containers, then use option 1. If you are familiar with setting up and running Python applications on your computer, then option 2 should be relatively easy for you to implement, but if you don't want to complicate yourself with running a service on your computer, then Heroku might be the best option for you. You can review the following sections to learn what's involved in each of the methods if you need more information to decide.

Running the Back End on Docker

If you are interested in running the back end service as a Docker container, you need to have Docker and git installed.

Open a new terminal window and find a suitable parent directory that is outside the React project directory. Clone the Microblog API GitHub repository to

download the project to your computer:

```
git clone https://github.com/miguelgrinberg/microblog-api
cd microblog-api
```

Create a configuration file with the name *.env*. An example is shown below, assuming you are using a SendGrid account as email server. Make sure you enter the appropriate email server details that apply to you.

```
DISABLE_AUTH=true
MAIL_SERVER=smtp.sendgrid.net
MAIL_PORT=587
MAIL_USE_TLS=true
MAIL_USERNAME=apikey
MAIL_PASSWORD=                  # <-- your SendGrid API key
MAIL_DEFAULT_SENDER=            # <-- any email address
```

It is important that the `DISABLE_AUTH` variable is set to `true` at this time, to remove authentication. You will enable authentication support in a later chapter.

Start the back end with Docker Compose as follows:

```
docker-compose up -d
```

Once the service is up and running, run the next two commands to populate the database used by the service with some randomly generated data:

```
docker-compose run --rm microblog-api bash -c "flask fake users 10"
docker-compose run --rm microblog-api bash -c "flask fake posts 100"
```

To check that the service is running correctly, navigate to the *http://localhost:5000* URL on your web browser. This should open the live API documentation site.

On some computers, port 5000 might be in used by another service. If that is your situation, you can specify a different port by adding a `MICROBLOG_API_PORT` variable to the *.env* file. The following example configures the service to run on port 4000:

```
MICROBLOG_API_PORT=4000
```

Running the Back End with Python

If you are familiar with running Python applications, you can just install and run the application directly on your computer. For this you need to have a recent Python 3 interpreter and git installed.

Open a new terminal window, find a location outside the React project's directory, and clone the GitHub repository for Microblog API there:

```
git clone https://github.com/miguelgrinberg/microblog-api
cd microblog-api
```

As with Docker, you need to create a configuration file with the name *.env*. Below you can see an example configuration file that assumes you are using a SendGrid account as email server.

```
DISABLE_AUTH=true
MAIL_SERVER=smtp.sendgrid.net
MAIL_PORT=587
MAIL_USE_TLS=true
MAIL_USERNAME=apikey
MAIL_PASSWORD=                  # <-- your SendGrid API key
MAIL_DEFAULT_SENDER=            # <-- any email address
```

It is very important that the `DISABLE_AUTH` variable in this configuration file is set to `true` at this time, to allow this service to work without authentication. You will enable authentication later, when implementing user logins.

Create a Python virtual environment and install the project's dependencies in it:

```
python3 -m venv venv
source venv/bin/activate
pip install -r requirements.txt
```

Initialize the service's database and add some random content to it with the following commands, which must be executed while the virtual environment is activated:

```
flask db upgrade
flask fake users 10
flask fake posts 100
```

To start the service type the following command in your terminal:

```
flask run
```

With the service running, navigate to *http://localhost:5000* and make sure the live API documentation site opens.

If you need to run the service on a different port, add the `--port` option to the run command. For example, to use port 4000, the command is:

```
flask run --port=4000
```

Deploying the Back End to Heroku

If you want to deploy to the Heroku service, first make sure you have an account on this service. A free account is sufficient for the needs of this project.

The read me page for the Microblog API project has a "Deploy to Heroku" button. Click this button to configure the deployment.

In the "App name" field you have to enter a name for the deployed back end application. You will need to find a name that hasn't been used by any other Heroku customer, so it may take a few tries until you find a unique name that is accepted.

For the "Choose a region" dropdown you can pick a region that is closest to where you are located, or you can leave the default selection.

In the `DISABLE_AUTH` field, enter the word `true`. This is going to deploy the service with authentication disabled. Authentication is covered in a later chapter, for now you will need to have the service running without authentication.

The next set of fields are for the email settings. Above you can find the settings you have to use if you followed my recommendation of using a SendGrid account. If you use a different email provider, configure it as required by your service.

You can leave any additional fields set to their default values.

When you are done with the configuration, click the "Deploy app" button. The deployment should only take a minute or two. When the application is deployed, you should see all green check marks, as shown in Figure 13.

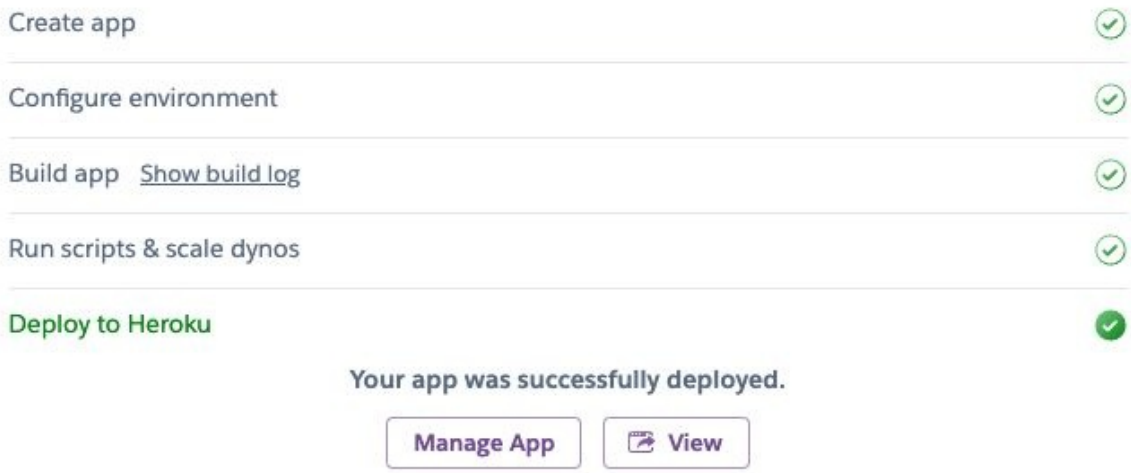


Figure 13 A completed Heroku deployment

The deployed application can be accessed at the URL *https://APPNAME.herokuapp.com*, where *APPNAME* is the application name that you entered in the configuration form. For example, if you deployed the application with the name "susan-microblog-api", then the base URL for the application is *https://susan-microblog-api/herokuapp.com*.

To make sure the application has been deployed successfully, you can open the base URL of your deployment in your browser. This should open the live documentation page for the service.

Using State Variables

The `Posts` component renders fake blog posts that were manually entered into the code early in the life of the project. This was useful because it made it possible to make progress with other parts of the project, without having to complicate things by involving the back end from the very start. Now it is finally time to remove the fake posts and render content returned by the server.

Doing this presents some challenges. For performance reasons, React requires that component functions render themselves to JSX quickly. In particular, it is not allowed for the render function to send a request out to the server asking for data and blocking the render while waiting for a response, as this would cause the whole application to freeze and appear unresponsive to the user.

How can the data be requested then? The short answer is that this is done as a *side effect*, but you will learn what this means in the next section. For now let's just say that the component has to schedule the request to run as a background task, so that the render process isn't held back.

The process of rendering data obtained from the server takes a few steps:

1. The component's render function is invoked. Within this function, the request to the server is scheduled as a background operation (called a side effect, in React jargon). The component function must return quickly and without blocking, so at this point it renders without any data, usually showing a spinner image or "loading" message.
2. The background task that was scheduled to request the data runs, and at some point a response from the server is received. The background task notifies React that some new data has arrived and is ready for rendering.
3. React calls the component's render function a second time, and the component re-renders itself with the data that was received.

After you think about this process, you will likely wonder how do the background data retrieval function, React and the rendering component communicate and coordinate to perform the three steps outlined above.

The React feature that makes this multistep render process possible is called a *state variable*.

The `useState()` hook function from React is used to create state variables. Hooks are special functions that have names that start with the word `use`. These functions can only be called from component functions or from other hooks. Calling a hook in any other context is not allowed and will be reported as an error when the application is built.

Here is how a `posts` state variable to hold a list of blog posts could be allocated inside the `Posts` component:

```
const [posts, setPosts] = useState();
```

If an argument is passed to `useState()`, then this becomes the initial value for the state variable. This can be a primitive value such as a string or a number, it can also be an array or an object, and it can also be set to `null` or `undefined`. If the argument is omitted, then the state variable is initialized with a value of

`undefined`.

The return value from the hook is an array with two elements, which in the above example are assigned to two constants using a destructuring assignment.

The first element of the returned array is the current value of the state variable. Going back to the three rendering steps above, when the component renders for the first time in step 1, this would be the initial value assigned to the state variable, which in this case is `undefined`.

The second element of the array is a *setter* function for the state variable. This function must be used to update the value of the state variable. This will be done in the background task when the data is received. Calling the setter function with a new value is what allows React to trigger the render that occurs in step 3 above.

When the component renders a second time in step 3, the `posts` constant returned by `useState()` is going to have the value that was passed to the setter function.

The first step in the migration to use real data in the `Posts` component is to replace the `posts` constant and the fake blog posts with a state variable of the same name.

Listing 29 src/components/Posts.js: Add a `posts` state variable

```
import { useState } from 'react';
import Spinner from 'react-bootstrap/Spinner';

export default function Posts() {
  const [posts, setPosts] = useState();

  // TODO: add a side effect function to request posts here

  return (
    <>
      {posts === undefined ?
        <Spinner animation="border" />
      :
        <>
          ... // <-- no changes to blog post JSX
```

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

The first line imports the `useState()` hook from the React library. The Spinner component imported in the second line comes with React-Bootstrap.

The state variable is created in the first line of the component function. The comment that follows is a placeholder for the side effect function, which will be added in the next section.

The `return` statement that looped through the blog posts has been expanded. At the top level there is a conditional that checks if the `posts` state variable is set to `undefined`, and in that case a spinner component is returned. This implements step 1 of the render process described earlier in this section, when the data from the server isn't available yet.

If `posts` has a value other than `undefined`, then it means that this is the second render, so in that case the previous loop logic is used to render the data obtained from the server.

Figure 14 shows how the application looks with these last changes.

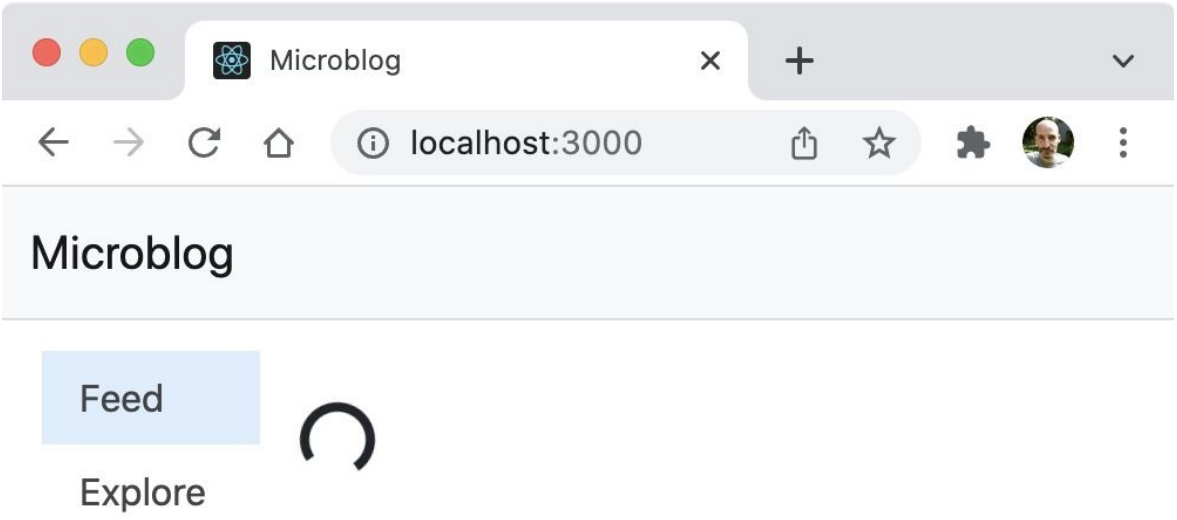


Figure 14 A spinner while the component's state variable is `undefined`

Side Effect Functions

In this section you are going to send a request to the back end to obtain remote data, which will be rendered to the page with the state variable logic added in the previous section. There is a bit of preparation required before the request can be made.

Configuring the Back End Root URL

Before the front end can make a request to the server, it needs to know what is the root URL of your back end. Depending on your back end installation method here is how you can determine this URL:

- If you are running the back end on a local Docker, then your root URL is *http://localhost:5000*. If the Docker host is remote, then change `localhost`

to the IP address or hostname of your Docker host. When using a custom port, change the `5000` to your chosen port number.

- If you are running the Python application directly on your computer, then your root URL is also *http://localhost:5000*. Once again, be sure to change the `5000` to the correct number if you are using a custom port.
- If you are running the back end on Heroku, then your root URL is *https://APPNAME.herokuapp.com*, where `APPNAME` is the application name that you selected when you deployed the service. If you can't remember the name, you can visit your Heroku dashboard to look it up.

To check that you have the correct root URL for your back end, type this URL in your web browser. If the URL is correct, the browser should automatically redirect to */docs* and show you the API documentation site. You can see how this looks in Figure 15.

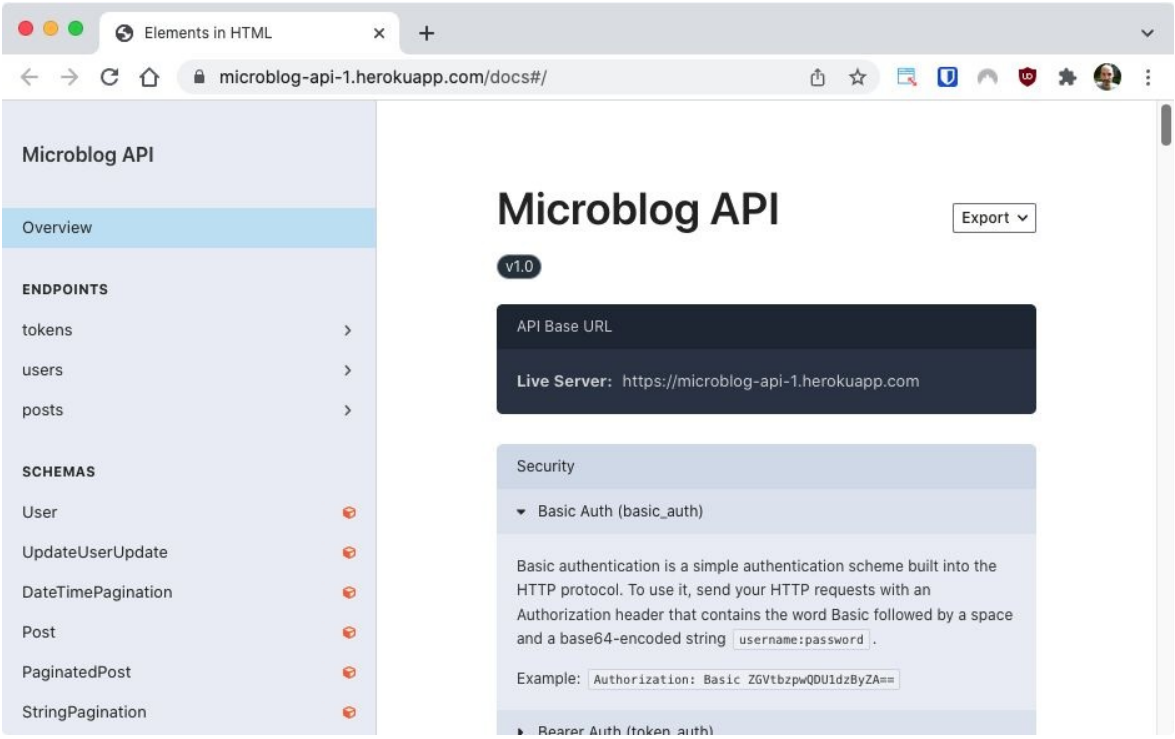


Figure 15 Microblog API documentation site

The most convenient way to incorporate the back end URL into the front end project is through an environment variable. Applications generated by Create React App have environment variable support, either directly from the shell or through environment files.

Create an environment file named *.env* (note the leading dot) in the top-level directory of your React project, and enter the following line in it, editing the URL to be the correct one for your back end installation:

```
REACT_APP_BASE_API_URL=http://localhost:5000
```

Do not include a trailing slash when you enter the back end URL.

Environment variables that are meant to be imported into the React application must have a name that starts with `REACT_APP_`. This ensures that any other variables that are in your environment, some of which can be sensitive, are not leaked to the front end by mistake.

Save the *.env* file with the new variable. Then stop the `npm start` process with Ctrl-C and restart it, so that the environment file is imported. The Create React App build will make the variable accessible anywhere in the application as `process.env.REACT_APP_BASE_API_URL`.

Using the API Documentation Site

Open the Microblog API documentation site once again, by typing its root URL in a browser. Find the "Endpoints" section in the left sidebar, and open the "posts" subsection to see all the endpoints related to blog posts. Click on the endpoint labeled "Retrieve the user's post feed".

The documentation page for the endpoint shows how to make this request. Because the back end is deployed without authentication support, you can ignore the security aspects for now. When running without authentication, all requests are automatically authenticated to the first user in the system.

From this page you can learn that the HTTP method for this request is `GET`, and that the URL path to attach after the root server URL is */api/feed*. There are some query parameters related to pagination, but they are all optional, so there is no need to worry about those for now.

The "Responses" section shows the structure of the data that is returned by the request, which has `data` and `pagination` top-level sections.

On the right side of the page there is a web form, where you can enter input arguments for the request and send a test request out to the server. The only input

argument this endpoint requires is the authentication token, but it is currently not checked, so you can click the "Send API Request" button with all the input fields blank and see an example response from this endpoint. This will allow you to familiarize yourself with a real response, which should include some randomly generated users and blog posts.

Sending a Request with `fetch()`

The fetch() function, available in all modern web browsers, is the simplest way to send HTTP requests to a server. Another popular HTTP client used in many React applications is axios. For this application `fetch()` will be used.

`GET` requests that don't require authentication or other input arguments can be sent just by calling `fetch()` with the URL of the target endpoint as an argument. For example:

```
const BASE_API_URL = process.env.REACT_APP_BASE_API_URL;
const response = await fetch(BASE_API_URL + '/api/feed');
```

The `fetch()` function uses promises, so it needs to be awaited when you are in a function declared as `async`.

The Response object returned by `fetch()` provides many attributes and methods to work with the HTTP response. A very useful attribute is `response.ok`, which is `true` when the request returned a success status code. For a JSON API, the `response.json()` method parses the data in the body of the response and returns it as a JavaScript object or array:

```
const results = await response.json();
if (response.ok) {
  console.log(results.data);
  console.log(results.pagination);
}
```

In this example, the `response.json()` method (which also returns a promise) is awaited, and then if the request was successful, the `data` and `pagination` keys of the JSON payload are printed to the console.

Creating a Side Effect Function

In React, side effect functions are created with the `useEffect()` hook function inside the component's render function. The first argument to `useEffect()` is a function with the code that needs to run in the background. The second argument is an array of dependencies that determine when the effect needs to run.

Understanding how to use the second argument to `useEffect()` is often difficult when learning React. A simple rule to remember, is that when this argument is set to an empty array, the side effect function runs once when the component is first rendered and never again.

A common mistake is to forget to include the second argument. This is interpreted by React as instructions to run the side effect function every single time the component renders, which is rarely necessary.

Later you will see that there are cases that require specific values for this array, but it is best to get used to set this argument to an empty array and then changing it only when necessary.

The listing below shows the updated `Posts` component, with the side effect function in place.

Listing 30 src/components/Posts.js: Load blog posts as a side effect

```
import { useState, useEffect } from 'react';
import Spinner from 'react-bootstrap/Spinner';

const BASE_API_URL = process.env.REACT_APP_BASE_API_URL;

export default function Posts() {
  const [posts, setPosts] = useState();

  useEffect(() => {
    (async () => {
      const response = await fetch(BASE_API_URL + '/api/feed');
      if (response.ok) {
        const results = await response.json();
        setPosts(results.data);
      }
      else {
        setPosts(null);
      }
```

```
    })();
  }, []);

  return (
    <>
      {posts === undefined ?
        <Spinner animation="border" />
      :
        <>
          {posts === null ?
             <p>Could not retrieve blog posts.</p>
          :
            <>
              ... // <-- no changes to blog post JSX
            </>
          }
        </>
      }
    </>
  );
}
```

The side effect function is defined after the `posts` state variable, so that it can access the `setPosts()` setter function to update the list of posts.

React requires that the function that is given as the argument to `useEffect()` is not async. A commonly used trick to enable the use of `async` and `await` in side effect functions is to create an inner async function and immediately call it. This pattern is commonly referred as an Immediately Invoked Function Expression (IIFE). To help you understand this, here is how this inner async function looks, isolated from the rest of the code:

```
(async () => {
  // await can be used here
})();
```

Note how the arrow function is defined within parenthesis, and the `()` at the end invokes it.

The logic in the side effect function uses `fetch()` to retrieve the user's post feed from the server. If the request succeeds, then the list of posts is set in the `posts` state variable through the `setPosts()` setter function.

When the state variable changes, React will trigger a new render of the component, and this time the loop section of the JSX will be used.

To make this component robust, it is also necessary to handle the case of the request failing. In the side effect function, the value of the `posts` state variable is set to `null` when `response.ok` is false.

As discussed above, the second argument to `useEffect()` is set to `[]`, to indicate that the function given in the first argument should only run when the initial render occurs. In this first render, the component will render itself with the spinner.

The JSX contents of this component have been expanded to also handle the case of `posts` being `null`, which is handled by rendering an error message. In a later chapter you will learn how to create global error alerts, which is better than having them in every component that uses the API.

Figure 16 shows actual blog posts returned by the server rendered on the page. If you are wondering why the content does not make any sense, remember that during installation, the back end's database was filled with randomly generated content.

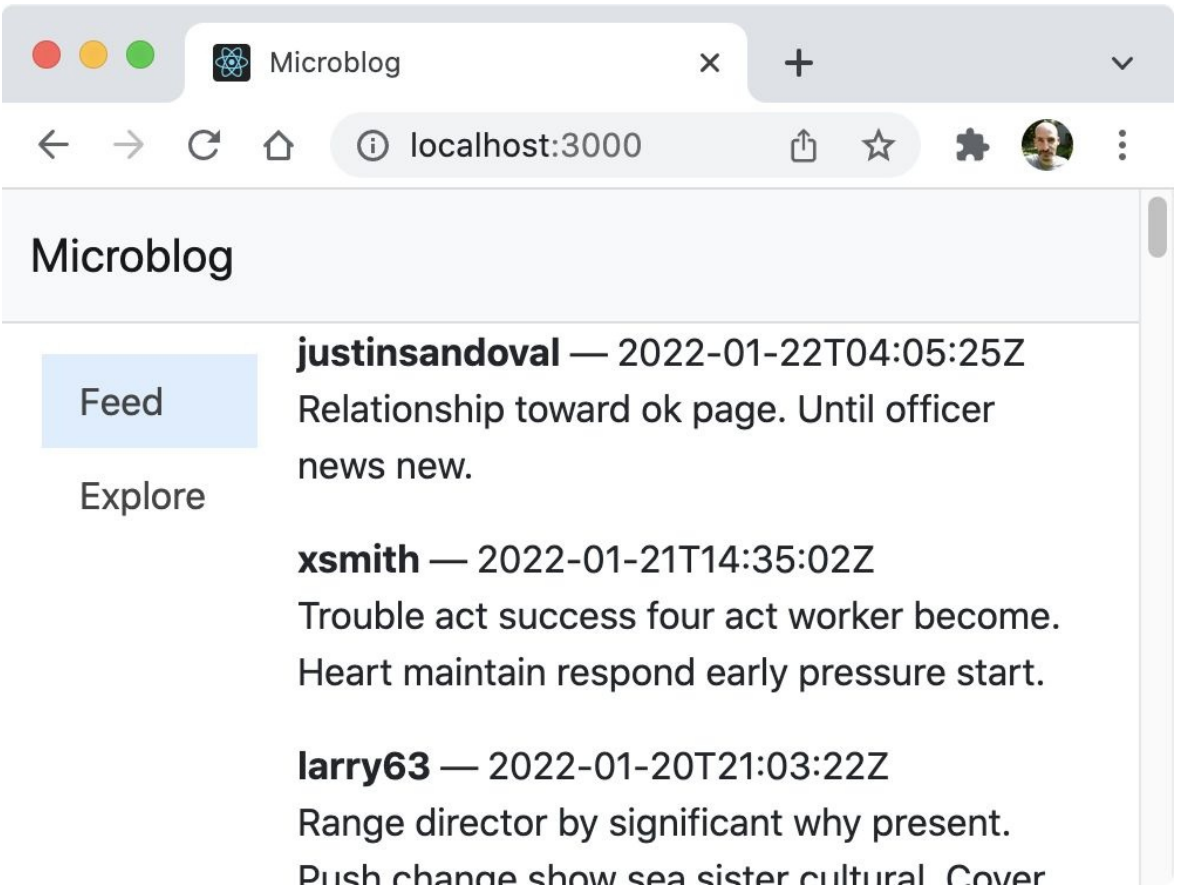


Figure 16 Blog posts rendered on the feed page

Rendering Blog Posts

Displaying more content makes it obvious that the render code for posts is extremely plain and needs an improvement. The code for the `Posts` component has grown as a result of adding the state variable and the side effect function, so this is a good opportunity to refactor it, by moving the JSX loop that renders a single blog post into a separate component, which can be called `Post`.

Here is `Posts` after the blog post rendering code was refactored:

Listing 31 *src/components/Posts.js*: Simplified `Posts` component

```
import { useState, useEffect } from 'react';
import Spinner from 'react-bootstrap/Spinner';
import Post from './Post';
```

```
const BASE_API_URL = process.env.REACT_APP_BASE_API_URL;

export default function Posts() {
  // <-- no changes to state variable and side effect function

  return (
    <>
      {posts === undefined ?
        <Spinner animation="border" />
      :
        <>
          {posts === null ?
             <p>Could not retrieve blog posts.</p>
          :
            <>
              {posts.length === 0 ?
                <p>There are no blog posts.</p>
              :
                posts.map(post => <Post key={post.id} post={post}
              }
            </>
          }
        </>
      }
    </>
  );
}
```

The loop in `Posts` is now a single line that references `Post` for each blog post. Something to remember when refactoring loops is that the required `key` attribute must always be in the source file that has the loop. React will not see it if it is moved into the child component.

The application is now broken, because `Posts` imports the `Post` component, which doesn't exist yet. The code for `Post`, with several render improvements, is shown below.

Listing 32 src/components/Post.js: Styled blog posts

```
import Stack from 'react-bootstrap/Stack';
import Image from 'react-bootstrap/Image';
import { Link } from 'react-router-dom';
```

```
export default function Post({ post }) {
  return (
    <Stack direction="horizontal" gap={3} className="Post">
      <Image src={post.author.avatar_url + '&s=48'}
             alt={post.author.username} roundedCircle />
      <div>
        <p>
          <Link to={'/user/' + post.author.username}>
            {post.author.username}
          </Link>
           &mdash; 
          {post.timestamp}:
        </p>
        <p>{post.text}</p>
      </div>
    </Stack>
  );
}
```

The new render code isn't very long, but there are a lot of changes from the previous version.

In the previous render logic, a post was rendered as a `<p>` element. This is not a good top-level element, because it makes it harder to also render the user's avatar image.

The new layout uses a horizontal `Stack` as the main component. The `gap` attribute of the stack adds a margin around each child. As with other top-level components, a class name of `Post` is added to facilitate CSS styling.

The first child in the stack is the Image component from React-Bootstrap, which is used to render the user's avatar image URL that is returned by the server. The server works with Gravatar URLs, which accept a `s` parameter in the query string to request a specific image size, set to 48 pixels in this instance. The `roundCircle` attribute makes the images round instead of square, which gives the avatar a nice touch. Microblog API returns Gravatar images that generate a geometric design for any email addresses that do not have a registered avatar. You can visit Gravatar to register an avatar for your email address.

The second component in this stack is the body of the post, which uses a `<div>` element as parent, with two `<p>` paragraphs inside for the post header and body respectively. The header uses a `Link` component from React-Router for the

author's username. The link points to the */user/{username}* user profile placeholder page created in Chapter 4.

Post Styling Improvements

The body of the post is rendered directly inside the second <p>. The resulting page needed a few minor CSS adjustments to look its best. Below are the CSS definitions added to *index.css*.

Listing 33 *src/index.css*: Styles for blog posts

```
... // <-- no changes to existing styles

.Content {
  margin-top: 10px;
}

.Post {
  align-items: start;
  padding-top: 5px;
  border-bottom: 1px solid #eee;
}

.Post:hover {
  background-color: #f8f8f8;
}

.Post a {
  color: #14c;
  text-decoration: none;
}

.Post a:visited {
  color: #14c;
}
```

In addition to some small spacing and alignment fixes, the CSS above changes the color and style of the username links, adds a border line between blog posts, and changes the background color of the post under the mouse pointer.

Displaying Relative Times

While the new `Post` component nicely formats blog posts on the page, the one part in which it is still lacking is in how the time of the post is presented. Microblog API returns all timestamps as strings that follow the ISO 8601 specification, which is a widely accepted format for machine-to-machine communication of dates and times. But this format is not the best for use by humans.

In this type of application, what works best is to show the time of a post in relative terms, such as "yesterday" or "3 hours ago". In this section you'll add a `TimeAgo` component that takes a timestamp in ISO 8601 format as a prop, and renders it to the page as a relative time. First update the `Post` component to use `TimeAgo`.

Listing 34 src/components/Post.js: Relative time for blog posts

```
import Stack from 'react-bootstrap/Stack';
import Image from 'react-bootstrap/Image';
import { Link } from 'react-router-dom';
import TimeAgo from './TimeAgo';

export default function Post({ post }) {
  return (
    <Stack direction="horizontal" gap={3} className="Post">
      <Image src={post.author.avatar_url + '&s=48'}
             alt={post.author.username} roundedCircle />
      <div>
        <p>
          <Link to={'/user/' + post.author.username}>
            {post.author.username}
          </Link>
           &mdash; 
          <TimeAgo isoDate={post.timestamp} />:
        </p>
        <p>{post.text}</p>
      </div>
    </Stack>
  );
}
```

With the above change the application is once again temporarily broken, this time due to the reference to a nonexistent `TimeAgo` component.

The following listing shows the general structure that will be used for this component, with placeholders for the actual logic.

Listing 35 src/components/TimeAgo.js: Render relative times

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

const secondsTable = [
  ['year', 60 * 60 * 24 * 365],
  ['month', 60 * 60 * 24 * 30],
  ['week', 60 * 60 * 24 * 7],
  ['day', 60 * 60 * 24],
  ['hour', 60 * 60],
  ['minute', 60],
];
const rtf = new Intl.RelativeTimeFormat(undefined, {numeric: 'autc

function getTimeAgo(date) {
  // TODO
}

export default function TimeAgo({ isoDate }) {
  // TODO
}
```

The component is going to use the `useState()` and `useEffect()` hooks, this time for a purpose that is not related to loading remote resources from the network.

The `secondsTable` array has the number of seconds in a year, a month, a week, a day, an hour, and a minute. These numbers are going to be useful in determining which of these units is the best to use in the relative time.

The `rtf` constant is an instance of the Intl.RelativeTimeFormat class, which is going to generate the actual text of the relative time. This class is not well known, but is available in all modern browsers.

The `getTimeAgo()` helper function accepts a `Date` object and finds the best relative units to use to render it. The function is defined outside the `TimeAgo` component because it is a standalone function that does not need to be different for each instantiation of the component, and it also does not need to change

when the component re-renders.

The implementation of the `getTimeAgo()` function is shown below.

Listing 36 src/components/TimeAgo.js: getTimeAgo() function

```
function getTimeAgo(date) {
  const seconds = Math.round((date.getTime() - new Date().getTime(
  const absSeconds = Math.abs(seconds);
  let bestUnit, bestTime, bestInterval;
  for (let [unit, unitSeconds] of secondsTable) {
    if (absSeconds >= unitSeconds) {
      bestUnit = unit;
      bestTime = Math.round(seconds / unitSeconds);
      bestInterval = unitSeconds / 2;
      break;
    }
  };
  if (!bestUnit) {
    bestUnit = 'second';
    bestTime = parseInt(seconds / 10) * 10;
    bestInterval = 10;
  }
  return [bestTime, bestUnit, bestInterval];
}
```

This function starts by calculating the number of seconds between the `date` argument and current time. For dates that are in the past, the result of this calculation is going to be negative, so for that reason the `absSeconds` constant stores the positive number of seconds.

A for-loop then iterates over the elements of `secondsTable`, to find the first unit (from largest to smallest) that is smaller than `absSeconds`, as this is the best relative unit to use. If a unit is found, then the function stores three related values to it:

- `bestUnit` is the unit that was determined to be the best to use.
- `bestTime` is the amount of time, in the selected units, rounded to the nearest integer.
- `bestInterval` is the interval at which the relative time needs to be updated. Since all amounts are rounded, there is no reason to update more often than

at half of the selected time unit.

If none of the units in `secondsTable` is selected, it means that the time that needs to be rendered is less than a minute old. In this case the units are set to `'second'`, but to avoid refreshing the time every second an interval of 10 seconds is used instead.

The return value of the function is an array with the amount of time, the unit, and the interval selected.

The implementation of the `TimeAgo` component is shown next.

Listing 37 src/components/TimeAgo.js: TimeAgo component

```
export default function TimeAgo({ isoDate }) {
  const date = new Date(Date.parse(isoDate));
  const [time, unit, interval] = getTimeAgo(date);
  const [, setUpdate] = useState(0);

  useEffect(() => {
    const timerId = setInterval(
      () => setUpdate(update => update + 1),
      interval * 1000
    );
    return () => clearInterval(timerId);
  }, [interval]);

  return (
    <span title={date.toString()}>{rtf.format(time, unit)}</span>
  );
}
```

Thanks to moving some logic to the `getTimeAgo()` helper function, this component is quite short, but there are several important React tricks hidden in this component that are worth discussing in detail.

The component first creates a `Date` object, by parsing the `isoDate` prop, which is the string representation of the post's date in ISO 8601 format. The `getTimeAgo()` function is called with this date to obtain the time, the units and the interval to use when rendering.

Let's ignore the state variable and the side effect function for a moment and look at the rendered JSX first. The component renders the timestamp as a `<span>` element. The text of the element is generated with the rtf.format() function, provided by the browser. This function takes the time and the units, and generates locale friendly text for that amount time. For example, when `time = -1` and `unit = 'day'`, this function might return the word `'yesterday'` for a browser configured for English.

The `<span>` element also has a `title` attribute, which renders the date with its default string representation. Browsers create a tooltip with the content of this attribute, which can be viewed by hovering the mouse over the text of the element.

You now know how the relative times can be rendered, but in addition to rendering them it would be nice if they automatically updated as time passes. The state variable and side effect function take care of this.

The usage of `useState()` in this component is different from the previous one, because only the setter function is stored. This usage solves a very specific need that this component has, which is to force itself to re-render even though none of the inputs ever change. React only re-renders components when their props or state variables change, so the only way to force a re-render is to create a dummy state variable that is not used anywhere, but is changed when a re-render is needed.

The `useEffect()` hook is used to create a side effect function, as before. The function creates an interval timer that runs at the interval returned in the third array element of the `getTimeAgo()` helper function.

The interval function calls the `setUpdate()` setter function of the state variable. Previously you've seen that state variable setter functions take the new value of the variable as an argument. An alternative form for the setter that is useful when the current value of the state variable is unknown or out of scope, is to pass a function. With this usage, React calls the function, passing the current value of the variable as an argument, and the function must return the updated value. Since this state variable is only needed to cause an update, any value that is different from the current one works. In this component the value of the state variable is incremented by one every time an update needs to be triggered.

This side effect function returns a function as a result, as opposite to the one in the `Posts` component, which does not return anything. Sometimes, side effect functions allocate resources, such as the interval timer here, and these resources need to be released when the component is removed from the page, to avoid resource leaks. When a side effect function returns a function, React calls this function to allow the component to perform any necessary clean up tasks. For this component, the side effect clean up function cancels the interval timer.

The second argument to `useEffect()` is also different in this side effect. You've seen that a side effect function that has an empty dependency array runs only during the first render of the component. Using the "10 minutes ago" example discussed above, this initial run of the side effect function would set up an interval timer than runs every 30 seconds. The interval timer is useful to keep the timestamp updated, but eventually one of these re-renders will change the units from minutes to hours, and at that point having an interval timer every 30 seconds makes no sense anymore, as it would be better to reduce the frequency to every 30 minutes. By adding `interval` to the dependency array of the side effect, the component is asking React to run the side effect function not only during the first render, but also during any renders in which the value of `interval` changes from its previous value.

Side effect dependencies are hard to think about. To help you understand the sequence of events in the life of this component, here is an example description of renders and side effect runs:

- Let's assume the initial render happens 10 minutes and 20 seconds after the time passed to `isoDate`. The component rounds this down and renders "10 minutes ago". Since this is the first render, the side effect function runs and starts an interval timer that forces a re-render every 30 seconds.
- 30 seconds later, the value of `isoDate` is now 10 minutes and 50 seconds ago. The component re-renders as "11 minutes ago". The interval is still 30 seconds, so the side effect does not run this time.
- Another 30 seconds pass, and `isoDate` is now 11 minutes and 20 seconds ago. The component renders "11 minutes ago" again and the side effect function does not run.
- The interval timer continues to run every 30 seconds, updating the rendered text as necessary, and without running the side effect function.
- After approximately 50 minutes of refreshes every 30 seconds, the component's re-render will render itself as "1 hour ago". The value of the

`interval` variable in this run of the component's render function is going to be 1800, which is equivalent to 30 minutes. In all previous renders, `interval` was 30, so this time the value has changed. Since this value is a dependency of the side effect function, React calls the cleanup function set during the initial render, which destroys the 30-second timer, and then launches the side effect function a second time. The side effect function now starts a new 30-minute interval timer.

- From now on the component re-renders every 30 minutes. If the application is left running long enough, eventually the interval will change to half a day, at which point the side effect function will run again to start an updated interval timer.

Figure 17 shows how blog posts look after all the improvements.

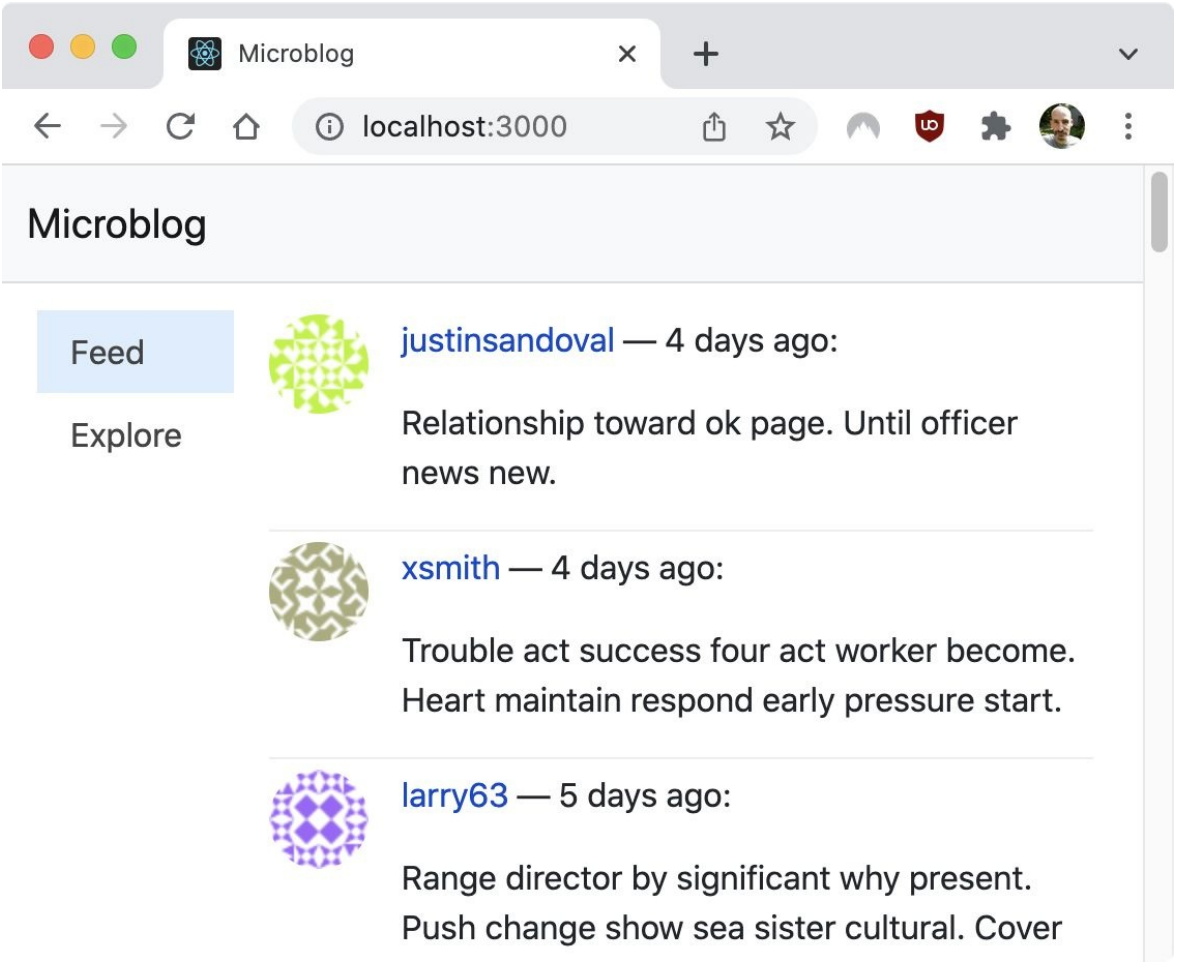


Figure 17 Blog posts rendered with improved styling

Chapter Summary

- Use environment variables that start with the `REACT_APP_` prefix in a *.env* file to provide configuration values such as the base URL of the back end.
- Render functions need to run and return quickly to prevent the application from becoming unresponsive.
- Use state variables to store data that needs to be retrieved asynchronously, or that can change throughout the life of the component.
- Use side effect functions to perform network or other asynchronous operations that update state variables. React will automatically re-render a component when any of its state variables change.
- For state variables associated with data loaded from the network, it is often useful to define specific values that indicate that the data is being retrieved, and also that the data retrieval has failed.
- A dummy write-only state variable can be used to force a component to re-render when none of its inputs have changed.
- A side effect function can return a cleanup function that React calls as needed to prevent memory and resource leaks.

Building an API Client

In Chapter 5, the `Posts` component was modified to make an API call that gets data from the server. The way it was done, however, does not scale very well, because as the application continues to grow, there's going to be other components that will also need to make their API calls, and having to repeat the API calling logic in several places will be difficult to manage, especially considering that these calls are going to become more complex once authentication and pagination are implemented.

In this chapter you are going to learn how to provide API access to any component that needs it, without having to duplicate code. As part of this effort, you will learn how to use React contexts and how to implement custom hook functions.

What is an API Client?

Having each component call `fetch()` directly when it needs to make an API call is far from ideal, because it leads to code duplication. A better solution is to have all the logic that deals with making API requests in a single place. For this project the code that makes API calls is going to be encapsulated in a JavaScript class.

Consider the following way in which a client class for this API might be used:

```
const api = new MicroblogApiClient();
const response = await api.get('/feed');
if (response.ok) {
  setPosts(response.body.data);
}
else {
  setPosts(null);
}
```

The `MicroblogApiClient` class will encapsulate all the knowledge about the API, including:

- The domain and port where the server is deployed
- The common portion of the path for all API endpoints (*/api*)
- How to pass arguments in the query string (which will be needed for pagination)
- How to catch and handle errors and exceptions
- How to parse the JSON in the response
- How to authenticate

A Simple Client Class for Microblog API

The listing below shows a first implementation of a Microblog API client class. Put this code in a file named *src/MicroblogApiClient.js*.

Listing 38 src/MicroblogApiClient.js: An API client class

```
const BASE_API_URL = process.env.REACT_APP_BASE_API_URL;

export default class MicroblogApiClient {
  constructor() {
    this.base_url =  BASE_API_URL + '/api';
  }

  async request(options) {
    let query = new URLSearchParams(options.query || {}).toString(
    if (query !== '') {
      query = '?' + query;
    }

    let response;
    try {
      response = await fetch(this.base_url + options.url + query,
        method: options.method,
        headers: {
          'Content-Type': 'application/json',
          ...options.headers,
        },
        body: options.body ? JSON.stringify(options.body) : null,
      });
    }
    catch (error) {
      response = {
```

```
        ok: false,
        status: 500,
        json: async () => { return {
          code: 500,
          message: 'The server is unresponsive',
          description: error.toString(),
        }; }
      };
    }

    return {
      ok: response.ok,
      status: response.status,
      body: response.status !== 204 ? await response.json() : null
    };
  }

  async get(url, query, options) {
    return this.request({method: 'GET', url, query, ...options});
  }

  async post(url, body, options) {
    return this.request({method: 'POST', url, body, ...options});
  }

  async put(url, body, options) {
    return this.request({method: 'PUT', url, body, ...options});
  }

  async delete(url, options) {
    return this.request({method: 'DELETE', url, ...options});
  }
}
```

The class constructor stores the common part of the URL that is used for all endpoints in the `base_url` attribute. This includes the base URL obtained from the `REACT_APP_BASE_API_URL` environment variable, and the */api* path prefix.

The actual request making logic, which is a slightly more elaborate version of the `fetch()` code used in the `Posts` component, is in the `request()` method. To make this class easier to use, there are shortcut methods for `get()`, `post()`, `put()` and `delete()` requests that invoke `request()` with appropriate settings.

Let's review the `request()` method in detail. This method takes all of its arguments from an `options` object. The `method` and `url` keys in this object are set by the `get()`, `post()`, `put()` and `delete()` helper methods, from its input arguments. The `body` key is set by `post()` and `put()`. Any additional options that the caller might need, such as custom headers or query string parameters, are accepted as a last argument on the four helper methods, and passed through.

The `request()` method starts by looking for the `query` key in the options, which allows the caller to specify query string arguments as an object, for convenience and readability. The `URLSearchParams` class available in the browser is used to render the object in the proper query string format.

When the server is down or unresponsive, the `fetch()` call raises an exception. For that reason, the `fetch()` call is made inside a `try`/`catch` block that handles this condition. This application handles fetch errors in the same way as if the server had returned a response with a 500 status code. The `catch` block builds a response object similar to that of fetch, but preloaded with the 500 status code response. The `body` in this error response is formatted in the same style as actual API errors returned by the Microblog API service.

The actual `fetch()` call is similar to the one in `Posts`, but generalized to work with the `method`, `url`, `query`, `body` and `headers` options that are passed in. Some common options are automatically added in this method. For example, a JSON content type, which is needed for `POST` and `PUT` requests that have a body, is automatically added. Also, the `body` argument is automatically rendered as a JSON object so that the caller doesn't have to worry about this detail.

The `ok`, `status` and `body` keys returned in the `fetch()` response are used to generate a simpler response object for the caller that has the JSON payload already decoded to an object. Responses from this API client will have three attributes:

- `ok`: a value of `true` or `false` that indicates the success or failure of the request.
- `status`: the numeric HTTP status code of the server response.
- `body`: an object with the payload returned in the body of the response, when available.

Sharing the API Client through a Context

The first way one may consider incorporating the `MicroblogApiClient` class into the application is to create an instance of the class in each component that needs API access. The problem with this approach is that it is inefficient when many components need to make API calls.

A better solution is to create a single instance that is shared among all the components that need to use the API. In Chapter 3 you learned that components can share data by passing props, but doing this would add a lot of boilerplate code as the API client instance would have to be passed down from high-level components down to low-level components through all the levels in between.

For cases when something needs to be shared with many components in different levels of the tree, React provides *contexts*.

A React context is created with the `createContext()` function from the React library:

```
import { createContext } from 'react';
const MyDataContext = createContext();
```

Once the context is created, it has to be inserted in the component hierarchy, high enough so that it is a parent to all the components that will use data shared through it. Here is an example of how this is done:

```
export default function MyApp() {
  return (
    <Container>
      <Header />
      <MyDataContext.Provider value={'data-to-share'}>
        <Sidebar />
        <Content />
      </MyDataContext.Provider>
    </Container>
  );
}
```

Here the `MyApp` component inserts a context into its JSX tree. This is done by using the `Provider` attribute of the context object that was created with the `createContext()` function.

In the above example, the `Sidebar` and `Content` components, along with all of

its children, will be able to gain access to the `value` prop set in the context provider component. The `Header` component will not be able to use the context, because it is not a child of the context provider element.

To access the value of a context, the child component can use the `useContext` hook as follows:

```
import { useContext } from 'react';
import { MyDataContext } from './MyDataContext';

export default function Sidebar() {
  const myData = useContext(MyDataContext);
  // ...
}
```

However, in practice, this leads to code that is not very clear to read, because the components that want to use the context need to import this strange context object only to be able to pass it to the `useContext()` hook.

An approach that is preferable is to create a *custom hook* function that returns the encapsulates the `useContext()` call. A custom hook is a function that starts with the word `use`. As previously discussed, hook functions are considered special by React. In particular, they are the only functions outside of component render functions that can call other hooks. Here is an example custom hook for the above context:

```
import { useContext } from 'react';

export function useMyData() {
  return useContext(MyDataContext);
}
```

The custom hook can be added in the same source file as the context object. Child components of the context then only need to import the hook function to access the context, leading to code that is more readable.

```
import { useMyData } from './MyDataContext';

export default function Sidebar() {
  const myData = useMyData();
  // ...
}
```

Ready to implement a context and a custom hook for the Microblog API client? Begin by creating a *src/contexts* subdirectory, where all the contexts of this application will be stored:

```
mkdir src/contexts
```

Add an `ApiProvider` component in *src/contexts/ApiProvider.js* that implements an API context, along with a `useApi()` custom hook.

Listing 39 src/contexts/ApiProvider.js: An API context

```
import { createContext, useContext } from 'react';
import MicroblogApiClient from '../MicroblogApiClient';

const ApiContext = createContext();

export default function ApiProvider({ children }) {
  const api = new MicroblogApiClient();

  return (
    <ApiContext.Provider value={api}>
      {children}
    </ApiContext.Provider>
  );
}

export function useApi() {
  return useContext(ApiContext);
}
```

First, notice how this source file has two exported functions. The `ApiProvider` component function is exported as the default symbol for the module, as with all other React components, but now there is also the custom hook `useApi()`, which has to be exported so that other components can use it.

The component renders the context's `ApiContext.Provider`, and puts its own children inside it. This effectively enables all the child components to access the context.

It is safe to assume that many of the application's components will need API access, so it makes sense to add this context high in the component hierarchy.

Below you can see how it is added in the `App` component.

Listing 40 *src/App.js*: Add the API context to the application

```
import Container from 'react-bootstrap/Container';
import { BrowserRouter, Routes, Route, Navigate } from 'react-rout
import ApiProvider from './contexts/ApiProvider';
import Header from './components/Header';
import FeedPage from './pages/FeedPage';
import ExplorePage from './pages/ExplorePage';
import UserPage from './pages/UserPage';
import LoginPage from './pages/LoginPage';

export default function App() {
  return (
    <Container fluid className="App">
      <BrowserRouter>
        <ApiProvider>
          <Header />
          <Routes>
            <Route path="/" element={<FeedPage />} />
            <Route path="/explore" element={<ExplorePage />} />
            <Route path="/user/:username" element={<UserPage />} /
            <Route path="/login" element={<LoginPage />} />
            <Route path="*" element={<Navigate to="/" />} />
          </Routes>
        </ApiProvider>
      </BrowserRouter>
    </Container>
  );
}
```

The context is added as the child of the `BrowserRouter` component from React-Router, with all the remaining components as its children. Since all the application components are children of this context, they are all able to use the `useApi()` hook to obtain access to the API client when they need to.

Now the `Posts` component can take advantage of the new hook to make its API call.

Listing 41 *src/components/Posts.js*: Using the `useApi()` hook

```
import { useState, useEffect } from 'react';
import Spinner from 'react-bootstrap/Spinner';
import { useApi } from '../contexts/ApiProvider';
import Post from './Post';

export default function Posts() {
  const [posts, setPosts] = useState(null);
  const api = useApi();

  useEffect(() => {
    (async () => {
      const response = await api.get('/feed');
      if (response.ok) {
        setPosts(response.body.data);
      }
      else {
        setPosts(null);
      }
    })();
  }, [api]);

  ... // <-- no changes in the rest of the function
}
```

Thanks to the `useApi()` hook function, the code in this component is now very clear, but there are a couple of interesting changes to pay attention to. First, note how the `useApi()` function is imported. Since this function was declared as a non-default export in *ApiProvider.js*, it has to be imported using a destructuring assignment.

The second argument to the `useEffect()` hook used to be an empty array, but is now `[api]`. As you already learned, this array is used to tell React what are the dependencies of the side effect function. With an empty array, the function only ran during the first render of the component. When dependencies are given, a change in a dependency will make React call the side effect function again, so that the component is always up-to-date.

The React build process analyzes your code, and one of the things it looks for is possible dependency omissions. If you forget to include `api` in the dependency array of the side effect function, the build generates a warning to alert you that this variable is used inside the function, which is a strong indicator that it should be treated as a dependency.

The User Profile Page

The `Post` component renders the author of each blog post as a link that points to the */user/{username}* page, defined within the React application. A placeholder for this page is implemented in the `UserPage` component. In this section this component will be enhanced to request user information from the API and render it to the page.

In Microblog API, you can get information about a user by sending a request to */api/users/{username}*. If you want to see what is the response for this endpoint, consult the API documentation by going to the root URL of your API service (for example, *http://localhost:5000*) and looking up the "Retrieve a user by username" endpoint.

The approach to construct the profile page is largely similar to that of the post feed, and it involves the following:

- Create a state variable for the data that needs to be requested from the API
- Obtain the API client object with the `useApi()` hook
- Define a side effect function that makes the API request and updates the state variable when the data is received
- Render a spinner while the state variable on the first render, then show the data from the state variable on the re-render

You can see the complete implementation of `UserPage` below.

Listing 42 src/pages/UserPage.js: User profile page

```
import { useState, useEffect } from 'react';
import Stack from 'react-bootstrap/Stack';
import Image from 'react-bootstrap/Image';
import Spinner from 'react-bootstrap/Spinner';
import { useParams } from 'react-router-dom';
import Body from '../components/Body';
import TimeAgo from '../components/TimeAgo';
import { useApi } from '../contexts/ApiProvider';

export default function UserPage() {
  const { username } = useParams();
  const [user, setUser] = useState();
```

```
  const api = useApi();

  useEffect(() => {
    (async () => {
      const response = await api.get('/users/' + username);
      setUser(response.ok ? response.body : null);
    })();
  }, [username, api]);

  return (
    <Body sidebar>
      {user === undefined ?
        <Spinner animation="border" />
      :
        <>
          {user === null ?
            <p>User not found.</p>
          :
            <Stack direction="horizontal" gap={4}>
              <Image src={user.avatar_url + '&s=128'} roundedCircl
              <div>
                <h1>{user.username}</h1>
                {user.about_me && <h5>{user.about_me}</h5>}
                <p>
                  Member since: <TimeAgo isoDate={user.first_seen}
                  <br />
                  Last seen: <TimeAgo isoDate={user.last_seen} />
                </p>
              </div>
            </Stack>
          }
        </>
      }
    </Body>
  );
}
```

As in the original placeholder page, the user to render is obtained from the URL, using the `useParams()` hook from React-Router. The `user` state variable is then defined to hold the user information, and the `useApi()` hook is used to gain access to the API client instance.

The side effect function makes the API request, and calls `setUser()` to update the state variable with the response. As in the `Posts` component, the state

variable starts with an `undefined` value, and once the request returns it is updated to the user information when the request succeeds, or to `null` if the request fails.

If you leave the dependency array for the side effect function empty, the React build warns that both `api` and `username` are dependencies, so both should be included, so that the component updates when they change. Making the `username` variable a dependency is extremely important, as it would cause the user information to be refreshed whenever the user changes.

The render code uses the `Body` component and includes the sidebar. Inside the body, if the `user` state variable is `undefined` a spinner is rendered to indicate that the data is being retrieved.

If the `user` state variable is `null`, which indicates a request error, an empty content area is shown. Recall that a centralized error handling solution for API errors will be built later.

When the `user` state variable is set to the user's information, a page for the user is built using many of the user attributes included in the response.

To create a nice layout for the profile page, a horizontal `Stack` component is used to create two sections for a large avatar (configured with a size of 128x128 pixels) on the left, and the username, user description and timestamps on the right. The `TimeAgo` component used for the blog post timestamps is also used here for the `member_since` and `last_seen` attributes.

With these changes, you can click on any username in the feed page to access their profile page. This is shown in Figure 18.

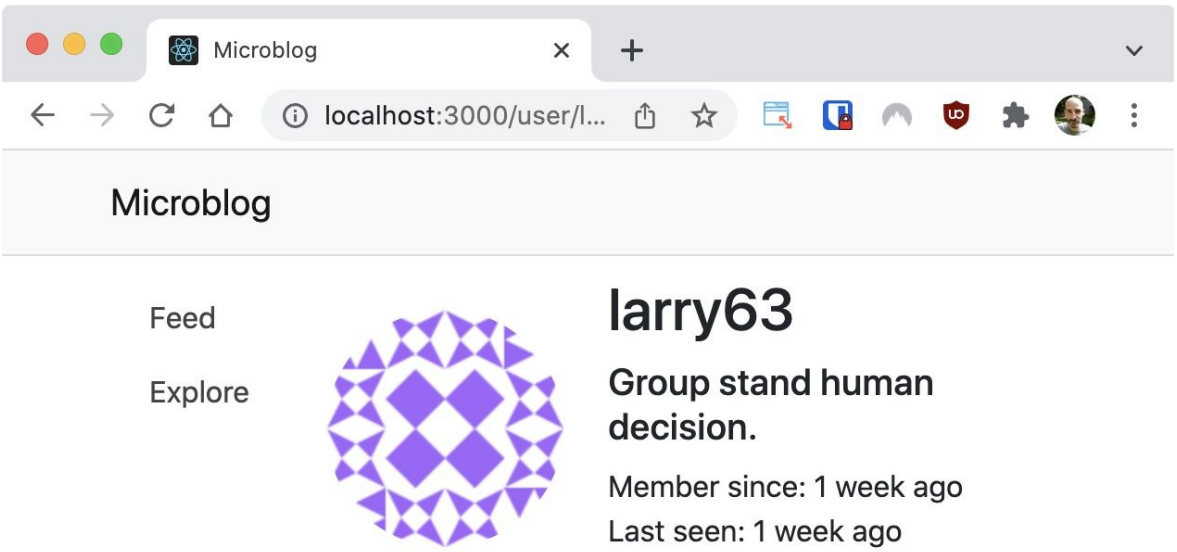


Figure 18 User profile page

Making Components Reusable Through Props

One small detail that would make the user profile page much more interesting is to include a list of the user's blog posts below the user details. This would be similar to the list rendered in the feed page, but restricted to only show the posts from the user being viewed. And the Explore page, which is still a placeholder, also needs to render a list of posts, this time all the posts in the system. So there are already three pages in the application that need to render lists of posts.

The `Posts` component already knows how to list a collection of blog posts, and it does it nicely, but it only renders the user's feed. To avoid building two nearly identical components that render the posts of a user in the user page, and all the posts in the explore page, the `Posts` component can be extended to render different lists of posts, depending on what the parent component wants.

The make this component more flexible, a `content` prop can be added to it. If this prop is not given, or if it is set to the string `feed`, then the user's feed is displayed as before. If the prop is set to `explore`, then all the available blog posts are rendered. Finally, if the `content` prop is set to any other value, it is assumed to be a user ID, and then the posts that are displayed are those of the user with

the requested ID.

See below the changes to the `Posts` component to implement this.

Listing 43 src/components/Posts.js: Configure post content to display

```
... // <-- no changes to imports

export default function Posts({ content }) {
  const [posts, setPosts] = useState(null);
  const api = useApi();

  let url;
  switch (content) {
    case 'feed':
    case undefined:
      url = '/feed';
      break;
    case 'explore':
      url = '/posts';
      break
    default:
      url = `/users/${content}/posts`;
      break;
  }

  useEffect(() => {
    (async () => {
      const response = await api.get(url);
      if (response.ok) {
        setPosts(response.body.data);
      }
      else {
        setPosts(null);
      }
    })();
  }, [api, url]);

  ... // <-- no changes in the rest of the function
}
```

In the previous version of this component, the URL for the API request was hardcoded to */feed*. Now a `switch` statement is used to determine what URL to

request based on the value of the `content` prop:

- for `feed`, or if `content` prop is not defined, the */feed* URL is requested as before
- for `explore`, the */posts* URL is requested
- for any other value, string interpolation is used to request the */users/{content}/posts* URL

All these URLs return lists of blog posts, and they all use the same response format. But each URL returns a different list. Remember that you can review the documentation for the endpoints in the documentation site provided by Microblog API.

The dependencies of the side effect function have been expanded one more time. Now `url` is also in the list, because it is a variable from outside that is used in the effect function, and consequently, whenever it changes the effect needs to run again to refresh the list of posts and keep it consistent with the state of the component.

The user profile page now can include a list of posts by the user being viewed. The changes to `UserPage` are below.

Listing 44 *src/pages/UserPage.js*: Show user's blog posts in profile page

```
... // <-- no changes to existing imports
import Posts from '../components/Posts';

export default function UserPage() {
  ... // <-- no changes in the main logic of the function

  return (
    <Body sidebar>
      {user === undefined ?
        <Spinner animation="border" />
      :
        <>
          {user === null ?
            <p>Could not retrieve blog posts.</p>
          :
            <>
              <Stack direction="horizontal" gap={4}>
```

```
                ... // <-- no changes to user details
              </Stack>
              <Posts content={user.id} />
            </>
          }
        </>
      }
    </Body>
  );
}
```

This change just wraps the `Stack` component in `<>` and `</>`, so that a `Posts` component can be added after it, while keeping a single parent in the JSX tree. The `Posts` component in this case has the `content` prop set to `user.id`, to request that only the posts from the user in question are rendered.

The updated user profile page is shown in Figure 19.

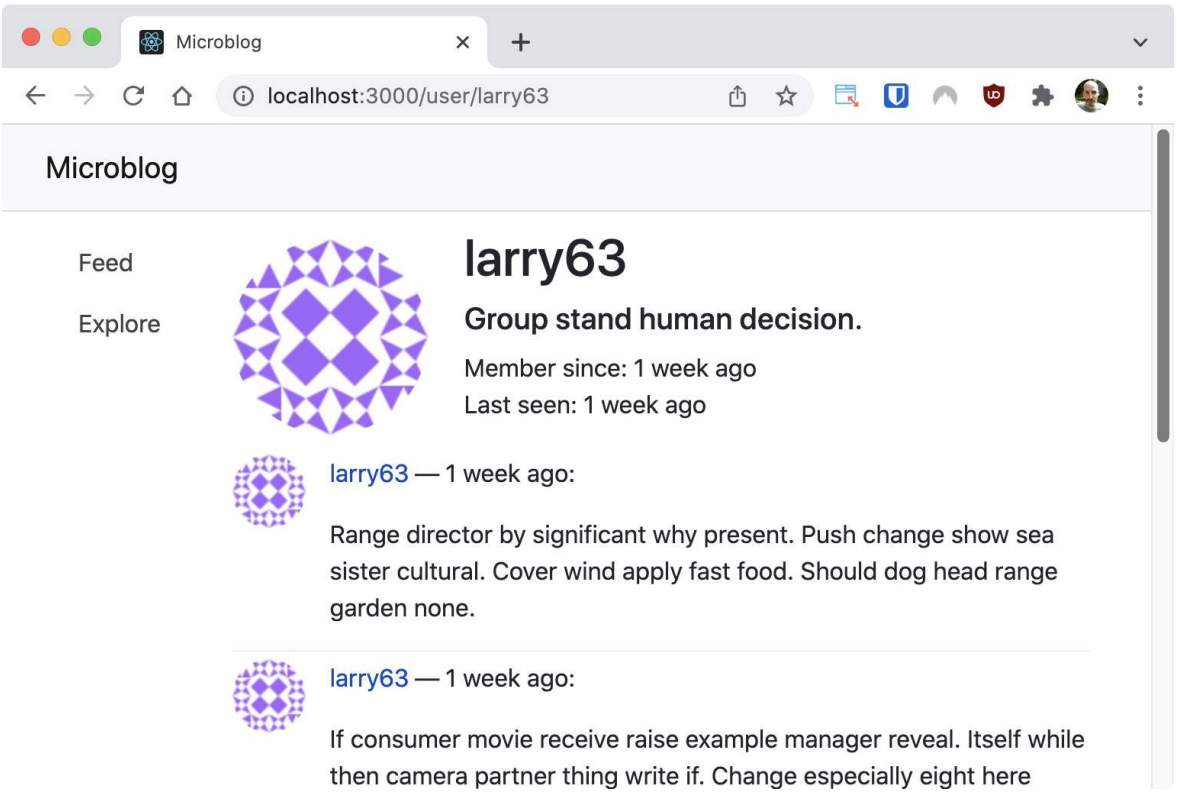


Figure 19 User profile page with blog posts

The `Posts` component is now also able to render posts in the Explore page, when the `content` prop is set to `explore`. The `ExplorePage` placeholder can be updated to display real content, as shown below.

Listing 45 src/pages/ExplorePage.js: the Explore page

```
import Body from '../components/Body';
import Posts from '../components/Posts';

export default function ExplorePage() {
  return (
    <Body sidebar>
      <Posts content="explore" />
    </Body>
  );
}
```

This is the perfect example of how good component reuse techniques help reduce code complexity. With just three lines of JSX code, a brand-new page

was added to the application!

Pagination

The lists of blog posts returned by Microblog API are *paginated*, which means that for a given request, the server returns the first few items, up to a configurable maximum (25 with default settings).

You've seen that the `data` attribute in the response payload contains the list of requested items. The server includes a second attribute in all the responses that include lists called `pagination`, which provides details about the portion of the complete list that was returned. Below you can see an example JSON payload, including pagination details:

```
{
  "data": [

  ],
  "pagination": {
    "count": 25,
    "limit": 25,
    "offset": 0,
    "total": 124
  }
}
```

Here `data` includes an array of 25 elements. This is reported in the `pagination` attribute under `count`. The `limit` key indicates what is the maximum page size that can be returned. The `offset` key reports the zero-based index of the first element returned, and `total` reports the size of the entire list.

Using these pagination details and a bit of math, it is possible to determine if there are more elements in the list following the ones that were returned. For the example above, the index of the last returned element is `offset + count - 1`, or 24. Since `total` is 124, the index of the last element of the list is 123, indicating that the user should be given the option to retrieve more elements if desired.

To let the user add for more elements, a "More" button is going to be displayed at the bottom of the list, when there are more elements available. Clicking the

button will trigger a new request to be issued for more items.

Let's begin by implementing a generic `More` component, which displays a button to load more items when appropriate.

Listing 46 src/components/More.js: A pagination button

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

export default function More({ pagination, loadNextPage }) {
  let thereAreMore = false;
  if (pagination) {
    const { offset, count, total } = pagination;
    thereAreMore = offset + count < total;
  }

  return (
    <div className="More">
      {thereAreMore &&
        <Button variant="outline-primary" onClick={loadNextPage}>
          More &raquo;
        </Button>
      }
    </div>
  );
}
```

The `More` component has two props. The `pagination` prop receives a pagination object extracted from a Microblog API response, including the four attributes discussed above. The `loadNextPage` prop is a handler that the parent component must provide, used to load the next page of items when the More button is clicked. Requiring the parent component to provide the logic to load the next page allows this component to be fully generic, instead of being tied to a specific part of the application.

The `thereAreMore` local variable uses the keys in the `pagination` prop to determine if there are more items to retrieve, and in that case, it displays the button. When `thereAreMore` has a `false` value, the button is not displayed, since there wouldn't be any items left to retrieve from the server.

The button is wrapped in a `<div>` element that is assigned the `More` class. This

class can be added to *index.css* to provide minor styling details to the component.

Listing 47 *src/index.css*: Pagination button styling

```
... // <-- no changes to existing styles

.More {
  margin-top: 10px;
  margin-bottom: 10px;
  text-align: right;
}
```

To support pagination in all the pages that show lists of posts, the `Posts` component can be expanded to use the `More` button, as shown below.

Listing 48 *src/components/Posts.js*: Post pagination

```
... // <-- no changes to existing imports
import More from './More';

export default function Posts({ content = 'feed' }) {
  const [posts, setPosts] = useState(null);
  const [pagination, setPagination] = useState();
  const api = useApi();

  ... // <-- no changes to how url is determined

  useEffect(() => {
    (async () => {
      const response = await api.get(url);
      if (response.ok) {
        setPosts(response.body.data);
        setPagination(response.body.pagination);
      }
      else {
        setPosts(null);
      }
    })();
  }, [api, url]);

  const loadNextPage = async () => {
```

```
    // TODO
  };

  return (
    <>
      {posts === undefined ?
        <Spinner animation="border" />
      :
        <>
          {posts === null ?
            <p>Could not retrieve blog posts.</p>
          :
            <>
              {posts.length === 0 ?
                <p>There are no blog posts.</p>
              :
                posts.map(post => <Post key={post.id} post={post}
              }
              <More pagination={pagination} loadNextPage={loadNext
            </>
          }
        </>
      }
    </>
  );
}
```

There are a few changes in this component:

- The `More` component is imported at the top.
- A new `pagination` state variable is included.
- When the response to the server request is received, the `posts` state variable is assigned the data, and the `pagination` state variable is assigned the pagination details.
- A `loadNextPage` function is defined as an inner function in the component, for now as a placeholder.
- The `More` component is added right after the list of blog posts is rendered. The `pagination` state variable and the `loadNextPage` function are passed as props.

With these changes, you should see the More button after the blog posts. If you don't see it, then the list of posts that you are looking at does not have additional

data. With the initial randomized data that is initialized with Microblog API, the Explore page should have enough posts to require pagination, but the Feed page may not. Figure 20 shows how the pagination button looks like.

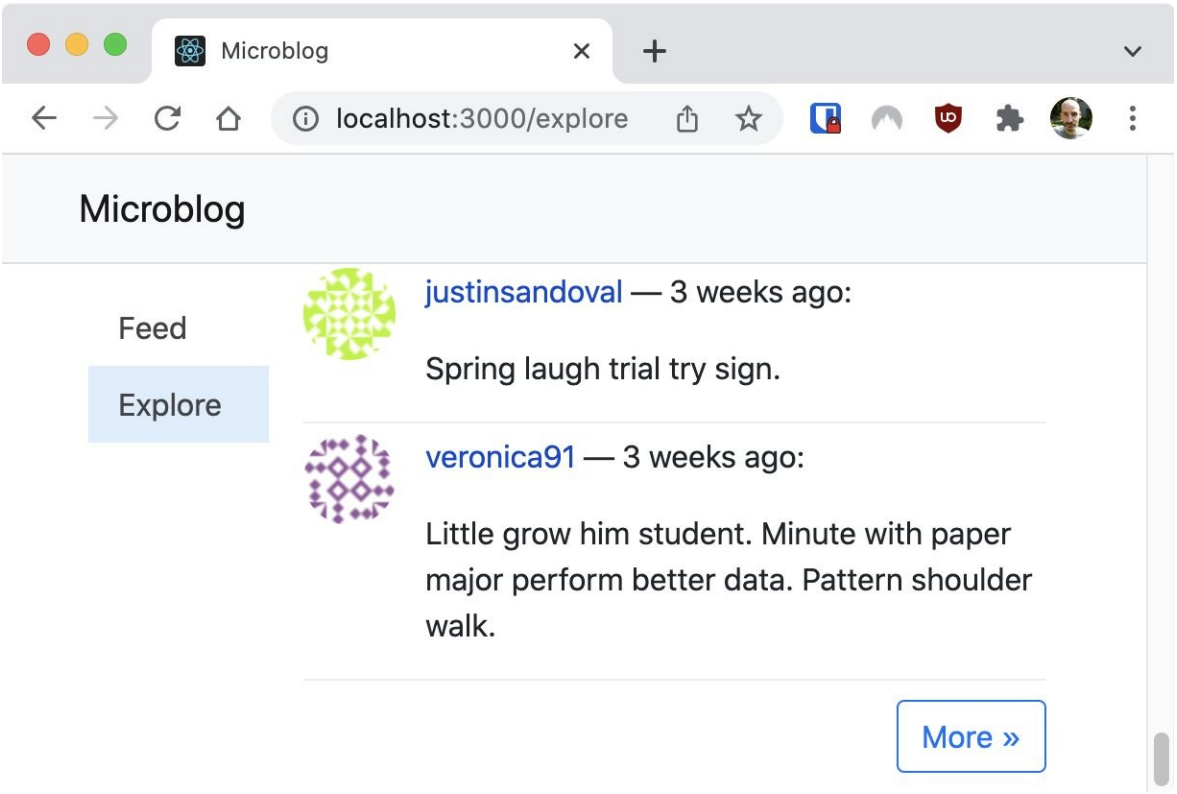


Figure 20 Pagination button

What's left to do now is to implement the logic in the `loadNextPage()` function. If you review the Microblog API documentation for any of the requests that return a list of blog posts, you'll notice that they all accept three optional query parameters, `limit`, `offset` and `after`.

The `limit` option is used to change the size of the returned page. This application does not need to do this, the client is happy to accept the default page size used by the server, so this option does not need to be used in this pagination implementation.

The `offset` option can be used to request items starting at a specific offset. For example, if the feed page is displaying the first 25 blog posts, passing `offset=25` to the same request URL would return blog posts 25 to 49 of that same list.

The `after` option can also be used to request additional items, but it works

differently than `offset`. For `after`, the argument needs to be the `timestamp` value of the last post being displayed, so for example, it could be `after=posts[posts.length - 1].timestamp`. The server then returns items starting right after the provided time.

Which of `offset` and `after` is better? This really depends on each particular case. One disadvantage of `offset` is that it only works well for lists that have new items added at the end, because items added in any other position would alter the indexes assigned to existing elements. The blog posts returned by Microblog API are sorted by their publication date, but in descending order, so new posts are always inserted at the start of the list, pushing older items down. On the other side, `after` works well only for lists that have a well-defined ordering.

Below you can see the implementation of the `loadNextPage()` function for the `Posts` component, using the `after` query parameter.

Listing 49 src/components/Posts.js: Load the next page of posts

```
const loadNextPage = async () => {
  const response = await api.get(url, {
    after: posts[posts.length - 1].timestamp
  });
  if (response.ok) {
    setPosts([...posts, ...response.body.data]);
    setPagination(response.body.pagination);
  }
};
```

The function makes a request on the same `url` as before, but this time the second argument for the `api.get()` call is added with the `after` query parameter.

When the response is received, the `posts` state variable is updated to a combined list of old and new blog posts, using the `...` spread operator. The `pagination` state variable is also updated with the new details received in the paginated request. The change to these two state variables will cause the `Posts` component to render again, which in turn will make the new blog posts appear at the bottom of the list.

After the new posts are displayed, the `More` component will remain at the

bottom, so the updated pagination state will be used again to determine if there are even more items waiting in the server, and in that case the button will continue to be visible. After the last batch of items are retrieved, the `More` component will not display the button anymore.

Chapter Summary

- Write the back end calling logic in a single place, for example in an API client class.
- To share a data item with a subcomponent, pass it down with a prop. To share a data item with many subcomponents across several levels, use a context.
- Use a custom hook function to make the code that uses a context more readable.
- Use props to make a component more generic, and facilitate its reuse.

Forms and Validation

A big part of most web applications is accepting, validating and processing user input. In this chapter you are going to learn how to perform these tasks with React by creating the user registration and login forms for Microblog.

Introduction to Forms with React-Bootstrap

The React-Bootstrap documentation has a section on forms that starts with a nice example of a login form. You can see how this example form looks in Figure 21.

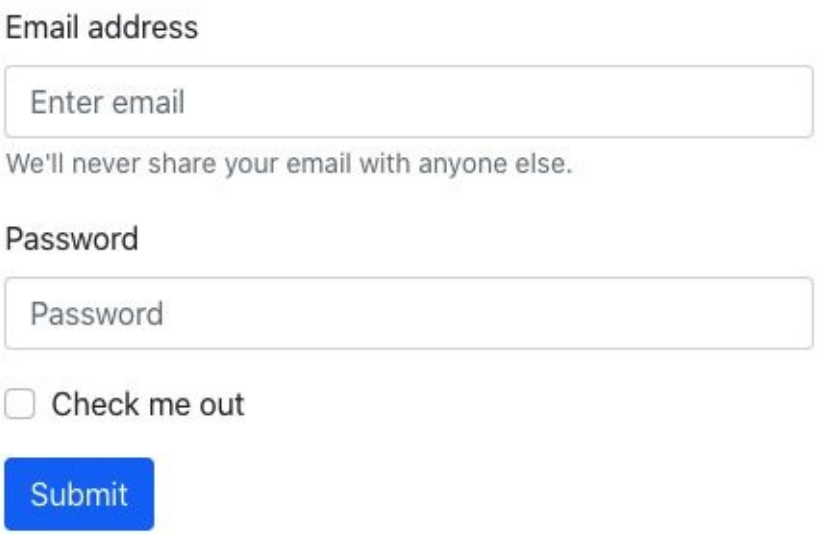


Figure 21 Example React-Bootstrap form

Below is a portion of the HTML source code that generates this form. For simplicity, I have only included the first field.

```
<Form>
  <Form.Group className="mb-3" controlId="formBasicEmail">
    <Form.Label>Email address</Form.Label>
    <Form.Control type="email" placeholder="Enter email" />
    <Form.Text className="text-muted">
      {"We'll never share your email with anyone else."}
    </Form.Text>
  </Form.Group>
```

```
  // ... more fields here

</Form>
```

The `Form` component from React-Bootstrap is used as the top-level parent of the form. Several subcomponents of `Form` are used to create the different parts of each field:

- `Form.Label` defines the label
- `Form.Control` defines the actual input field
- `Form.Text` defines a message that appears below the field

As you see, defining an input field involves a decent amount of boilerplate, and having to repeat all this for every input field in every form is not ideal. You can probably guess that a better idea is to create a reusable input field component.

A Reusable Form Input Field

Below is the definition of a new component called `InputField`.

Listing 50 *src/components/InputField.js*: A generic form input field

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

export default function InputField(
  { name, label, type, placeholder, error, fieldRef }
) {
  return (
    <Form.Group controlId={name} className="InputField">
      {label && <Form.Label>{label}</Form.Label>}
      <Form.Control
        type={type || 'text'}
        placeholder={placeholder}
        ref={fieldRef}
      />
      <Form.Text className="text-danger">{error}</Form.Text>
    </Form.Group>
  );
}
```

This component accepts several props, to allow for the most flexibility in rendering the field. The parent component must pass the field name, the label text, the field type (text, password, etc.), the placeholder text that appears inside the field when it is empty, and a validation error message, which will appear below the field.

The props are all included as JSX template expressions in the rendered input field. There are a few minor differences between this field and the example one from React-Bootstrap:

- The label is omitted if the `label` prop was not passed by the parent component
- The `type` prop is optional, and defaults to `text` when not given by the parent
- The error message uses a `text-danger` style from Bootstrap, which renders the text in red
- The `placeholder` and `error` props are optional, and will render as empty text when not provided by the parent

There is one more prop in this component called `fieldRef`, which is in turn passed to the `Form.Control` component as a `ref` prop. A reference provides a way for the application to interact with a rendered element. You will learn about React references in detail later in this chapter.

The top-level component in the input field was given the `InputField` class name to make it easier to customize how it looks on the page. Below is a small CSS addition to *index.css* that adds top and bottom margins to this component.

Listing 51 *src/index.css*: Styles for the input field

```
.InputField {
  margin-top: 15px;
  margin-bottom: 15px;
}
```

The Login Form

Using the generic input field defined above it is now possible to build the Login page in *src/pages/LoginPage.js*.

Listing 52 src/pages/LoginPage.js: Login page

```
import { useState } from 'react';
import Form from 'react-bootstrap/Form';
import Button from 'react-bootstrap/Button';
import Body from '../components/Body';
import InputField from '../components/InputField';

export default function LoginPage() {
  const [formErrors, setFormErrors] = useState({});

  const onSubmit = (ev) => {
    ev.preventDefault();
    console.log('handle form here');
  };

  return (
    <Body>
      <h1>Login</h1>
      <Form onSubmit={onSubmit}>
        <InputField
          name="username" label="Username or email address"
          error={formErrors.username} />
        <InputField
          name="password" label="Password" type="password"
          error={formErrors.password} />
        <Button variant="primary" type="submit">Login</Button>
      </Form>
    </Body>
  );
}
```

The page uses the `Body` component as its main wrapper, as all other pages. The login page will not offer navigation links, so for that reason the `sidebar` prop is not included.

The `Form` component has a `onSubmit` prop, which configures a handler function that will be invoked when the form is submitted. The handler starts by doing something very important: it disables the browser's own form submission logic, by calling the `ev.preventDefault()` method of the event object that was passed as an argument. This is necessary to prevent the browser from sending a network request with the form data.

This version of the component does not implement the form handling logic yet, it just logs a message to the browser's console to confirm that the event was received. The actual form submission code will be added in the following sections.

The form implements a standard two field login form, using the `InputField` component defined above. The `name`, `label` and `error` props are passed for both fields. For the first field `type` isn't passed, so the default type of `text` is used. The password field needs to have the `type` explicitly set, so that the characters are not visible. The `placeholder` prop is not used in any of the fields in this form.

The `error` props in both input fields is set to an attribute of a `formErrors` state variable that is defined inside the component. The state variable is initialized as an empty object, so the error messages for both fields are going to be initially set to `undefined`, which means they are not going to render any visible text. You will learn how to make validation errors visible later.

The submit button is defined directly with the Button component from React-Bootstrap.

Figure 22 shows how the login page looks.

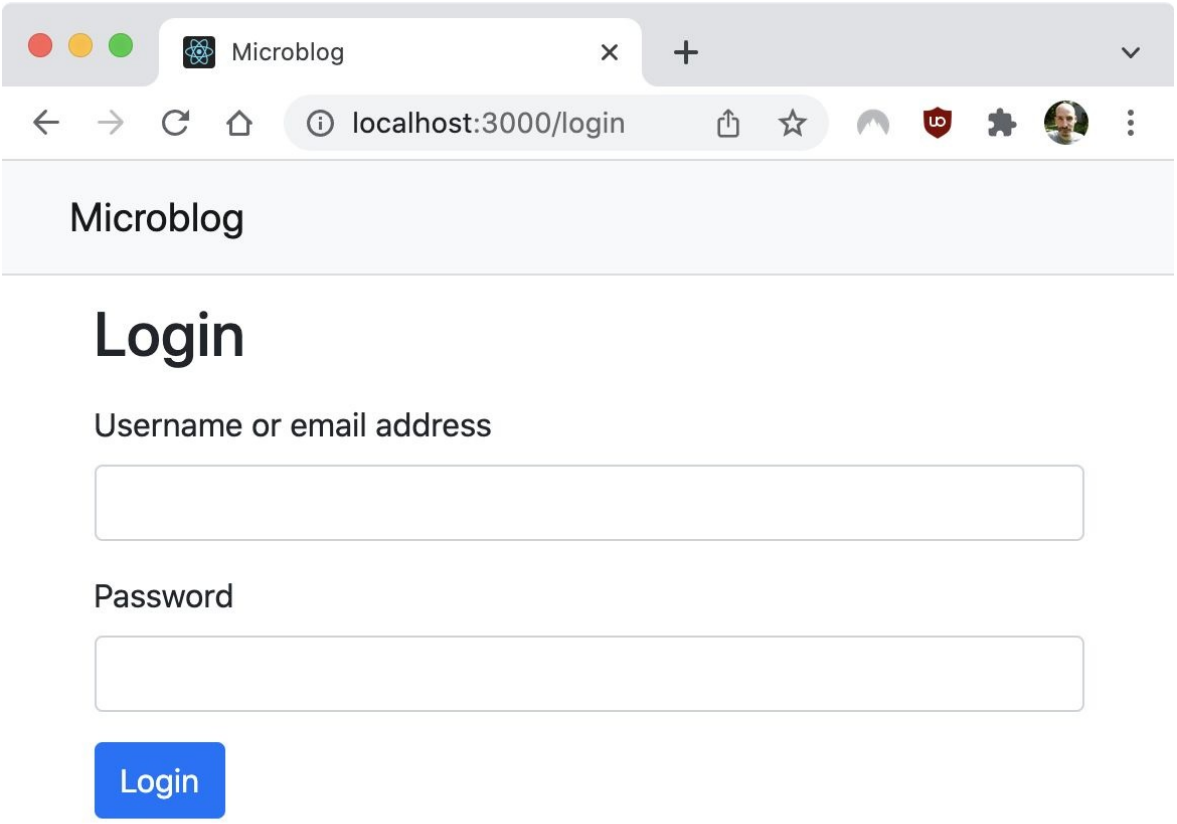


Figure 22 Login page

Controlled and Uncontrolled Components

An important part of implementing a form is to be able to read what the user enters in the form fields. There are two main techniques to handle user input in React, using controlled or uncontrolled components.

A *controlled* component is coded with event handlers that catch all changes to input fields. Every time a change handler triggers for a field, the updated contents of the field are copied to a React state variable. With this method, the values of the input fields for a form can be obtained from a state variable that acts as a mirror of the field.

An *uncontrolled* component, on the other side, does not have its value tracked by React. When the field's data is needed, DOM APIs are used to obtain it directly from the element.

An often cited disadvantage of the controlled method, especially for forms with large number of fields, is that every form field needs a state variable and one or more event handlers to capture all the changes the user can make to them, making them tedious to write. Uncontrolled components also have some boilerplate code required, but overall they need significantly less code.

For this project, the uncontrolled method will be used in all forms.

Accessing Components through DOM References

When working with vanilla JavaScript, the standard method to reference an element is to give it an `id` attribute, which then makes it possible to retrieve the element with the `document.getElementById()` function. In complex applications it is difficult to maintain unique `id` values for all the elements that need to be addressed on the page, and it is easy to inadvertently introduce duplicates.

React has a more elegant solution based on *references*. A reference eliminates the need to come up with a unique identifier for every element, a task that gets harder as the number of elements and page complexity grows.

A reference can be created with the `useRef()` hook inside a component's render function:

```
export default function MyForm() {
  const usernameField = useRef();
  ...
}
```

To associate this reference with an element rendered to the page, the `ref` attribute is added to the element when it is rendered.

```
export default function MyForm() {
  const usernameField = useRef();

  return (
    <form>
      <input type="text" ref={usernameField} />
    </form>
  );
}
```

The reference object has a `current` attribute that can be used in side effect functions and event handlers to access the actual DOM object associated with the component:

```
export default function MyForm() {
  const usernameField = useRef();

  const onSubmit = (ev) => {
    ev.preventDefault();
    alert('Your username is: ' + usernameField.current.value);
  };

  return (
    <form onSubmit={onSubmit}>
      <input type="text" ref={usernameField} />
    </form>
  );
}
```

Let's implement references for the two fields in the `LoginPage` component.

As you recall, the `InputField` component accepts a `fieldRef` prop. The parent component can use this prop to pass a reference object. This reference is assigned to the `ref` prop on the input field element. With this solution, the parent gets access the input field and can obtain its value when processing the form submission.

Why is the prop in the `InputField` component called `inputRef` and not also `ref`? The reason is that `ref` is an attribute name that React handles in a special way, similar to the `key` attribute, so these attributes cannot be used as prop names. If you feel strongly about passing references to a subcomponent using a prop named `ref`, React provides a forwarding ref option that makes it possible.

The following listing shows the `LoginPage` component updated to have references for the two input fields.

Listing 53 src/pages/LoginPage.js: References in the login page

```
import { useState, useEffect, useRef } from 'react';
import Form from 'react-bootstrap/Form';
import Button from 'react-bootstrap/Button';
```

```
import Body from '../components/Body';
import InputField from '../components/InputField';

export default function LoginPage() {
  const [formErrors, setFormErrors] = useState({});
  const usernameField = useRef();
  const passwordField = useRef();

  useEffect(() => {
    usernameField.current.focus();
  }, []);

  const onSubmit = (ev) => {
    ev.preventDefault();
    const username = usernameField.current.value;
    const password = passwordField.current.value;

    console.log(`You entered ${username}:${password}`);
  };

  return (
    <Body>
      <h1>Login</h1>
      <Form onSubmit={onSubmit}>
        <InputField
          name="username" label="Username or email address"
          error={formErrors.username} fieldRef={usernameField} />
        <InputField
          name="password" label="Password" type="password"
          error={formErrors.password} fieldRef={passwordField} />
        <Button variant="primary" type="submit">Login</Button>
      </Form>
    </Body>
  );
}
```

The `usernameField` and `passwordField` references are created at the start of the render function, and are passed as `fieldRef` props to the two `InputField` components, which in turn will set these references on the actual input elements of the form.

This new version of the `LoginPage` component has a new side effect function that shows a nice trick that takes advantage of the new references. The function

runs only the first time the login page is rendered (note that the dependency array is empty), and makes the first field in the form focused. This means that the user can start typing the username right away, without having to click on the field with the mouse first. The focus() method used here is part of the DOM API.

The `onSubmit` event handler for the form retrieves the values of the referenced input fields in the DOM, and for now, logs them to the console so that you can verify that everything is working when the form is submitted.

Client-Side Field Validation

When the user submits the form, the application must perform validation of all the form fields. It is important that all the data that is submitted to the server is validated there, because validation tasks performed in the client are easy to bypass by malicious users.

With the purpose of making applications more responsive, it is common for clients to perform complementary validation tasks in the client, with the goal to catch the most basic errors without having to make a trip to the server and back. For example, checks can be added to ensure that both the username and password fields are not empty before submitting the form. The server will check this again, but it is simple enough to check in the client as well.

The `formErrors` state variable added to the `LoginPage` component earlier is used to hold validation error messages for form fields. The listing below shows the additions to the `onSubmit` handler in this component that are necessary to validate that the username and password fields are not empty when the form is submitted.

Listing 54 src/pages/LoginPage.js: Validate input fields

```
const onSubmit = (ev) => {
  ev.preventDefault();
  const username = usernameField.current.value;
  const password = passwordField.current.value;

  const errors = {};
  if (!username) {
    errors.username = 'Username must not be empty.';
```

```
    }
    if (!password) {
      errors.password = 'Password must not be empty.';
    }
    setFormErrors(errors);
    if (Object.keys(errors).length > 0) {
      return;
    }

    // TODO: log the user in
  };
```

The `errors` constant is initialized to an empty object, and populated with `username` and/or `password` keys when any of these fields are found to be empty. The values inserted under these keys are the text of the error messages. These values are already passed as the `error` props on the `InputField` components.

After the `errors` object is populated, it is set as the updated value of the `formErrors` state variable. When the value of the state value changes, React will re-render the component, and during the re-render, the error messages will be displayed below each corresponding field. If there is at least one error, the function returns early, to prevent any actual actions defined later to execute when the form submission was deemed invalid.

After incorporating these changes, navigate to *http://localhost:3000/login* on your browser and try submitting the form with empty and non-empty fields to see how the client-side validation provides immediate feedback. Figure 23 shows the form after it was submitted with both fields empty.

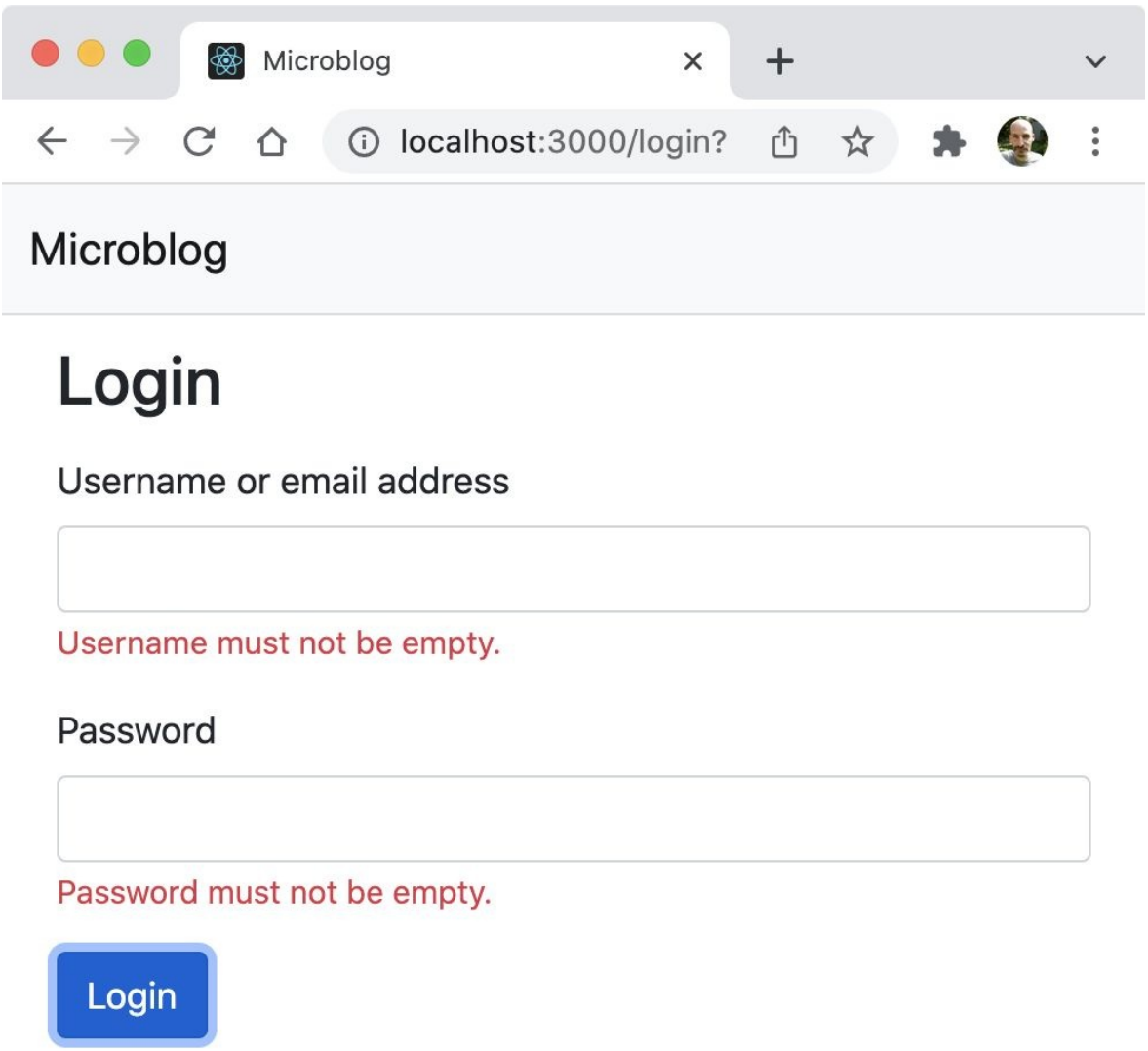


Figure 23 Form validation errors

The User Registration Form

You will learn how to complete the implementation of the login form in the next chapter, when the topic of user authentication is discussed. In preparation for that work, in this section you'll implement another important form, the one dedicated to registering new users into the system.

The new form will be in a `RegistrationPage` component, with the same structure as `LoginPage`. You can see it below.

Listing 55 src/pages/RegistrationPage.js: A user registration page

```
import { useState, useEffect, useRef } from 'react';
import Form from 'react-bootstrap/Form';
import Button from 'react-bootstrap/Button';
import Body from '../components/Body';
import InputField from '../components/InputField';

export default function RegistrationPage() {
  const [formErrors, setFormErrors] = useState({});
  const usernameField = useRef();
  const emailField = useRef();
  const passwordField = useRef();
  const password2Field = useRef();

  useEffect(() => {
    usernameField.current.focus();
  }, []);

  const onSubmit = async (event) => {
    // TODO
  };

  return (
    <Body>
      <h1>Register</h1>
      <Form onSubmit={onSubmit}>
        <InputField
          name="username" label="Username"
          error={formErrors.username} fieldRef={usernameField} />
        <InputField
          name="email" label="Email address"
          error={formErrors.email} fieldRef={emailField} />
        <InputField
          name="password" label="Password" type="password"
          error={formErrors.password} fieldRef={passwordField} />
        <InputField
          name="password2" label="Password again" type="password"
          error={formErrors.password2} fieldRef={password2Field} /
        <Button variant="primary" type="submit">Register</Button>
      </Form>
    </Body>
  );
}
```

The registration form has four fields for username, email, password and password confirmation. These fields all have a reference assigned to them. A `formErrors` state variable is added on this form as well, to keep track of validation error messages. The first field of the form is given the focus with a side effect function, as in `LoginPage`.

This page needs to be associated with the */register* route in *App.js*.

Listing 56 *src/App.js*: Registration page route

```
... // <-- no changes to existing imports
import RegistrationPage from './pages/RegistrationPage';

export default function App() {
  return (
    <Container fluid className="App">
      <BrowserRouter>
        <ApiProvider>
          <Header />
          <Routes>
            <Route path="/" element={<FeedPage />} />
            <Route path="/explore" element={<ExplorePage />} />
            <Route path="/user/:username" element={<UserPage />} /
            <Route path="/login" element={<LoginPage />} />
            <Route path="/register" element={<RegistrationPage />}
            <Route path="*" element={<Navigate to="/" />} />
          </Routes>
        </ApiProvider>
      </BrowserRouter>
    </Container>
  );
}
```

The login page can include a link to the new registration page below the login form's submit button.

Listing 57 *src/pages/LoginPage.js*: Link to user registration page

```
... // <-- no changes to existing imports
import { Link } from 'react-router-dom';
```

```
export default function LoginPage() {
  ... // <-- no changes in the body of the function

  return (
    <Body>
      <h1>Login</h1>
      ... // <-- no changes to the form
      <hr />
      <p>Don&apos;t have an account? <Link to="/register">Register
    </Body>
  );
}
```

Form Submission and Server-Side Field Validation

Registering a user with Microblog API is a straightforward operation that only requires sending a `POST` request to */api/users* with the new user's username, email and chosen password. This request can be made through the `MicroblogApiClient` instance, which can be accessed in this component through the `useApi()` custom hook.

The listing below shows the changes to the registration page to support the form submission.

Listing 58 src/pages/RegistrationPage.js: Registration form submission

```
... // <-- no changes to existing imports
import { useNavigate } from 'react-router-dom';
import { useApi } from '../contexts/ApiProvider';

export default function RegistrationPage() {
  ... // <-- no changes to state variables and references
  const navigate = useNavigate();
  const api = useApi();

  const onSubmit = async (event) => {
    event.preventDefault();
    if (passwordField.current.value !== password2Field.current.val
      setFormErrors({password2: "Passwords don't match"});
    }
    else {
```

```
      const data = await api.post('/users', {
        username: usernameField.current.value,
        email: emailField.current.value,
        password: passwordField.current.value
      });
      if (!data.ok) {
        setFormErrors(data.body.errors.json);
      }
      else {
        setFormErrors({});
        navigate('/login');
      }
    }
  };

  ... // <-- no changes to returned JSX
}
```

In the body of the component function, the `api` instance is imported with the custom `useApi()` hook function. In addition to that, the `navigate()` function from the React-Router package is included, through the `useNavigate()` hook. This function is used to automatically redirect the user to the */login* route after a successful user registration.

The validation of this form has two passes. First, a client-side validation pass is done in the `onSubmit()` handler function. For this form, a check is done to ensure that the `password` and `password2` fields are equal. If an error is found, the `formErrors` state variable is updated with an error message and the form submission is suspended.

The client-side validation could be expanded to also check that none of the fields are empty, that the email address is syntactically correct, and many other checks. But given that the back end performs all these checks already, an argument can be made that it is best to avoid repetition.

If the client-side validation of the two password fields does not detect any errors, then the `api` instance is used to send the `POST` request to the server. The first argument to `api.post()` is the last part of the endpoint URL (after */api*), and the second argument is an object with the request body, which is sent to the server in JSON format.

If the request fails due to validation, the Microblog API back end returns an

error response. The body of the response includes detailed information about the error, and in particular, the `errors.json` object contains validation error messages for each field. This is actually very convenient, because the `formErrors` state variable uses the same format. To display the validation errors from the server, this object is set directly on the state variable. Figure 24 shows an example of server-side validation errors displayed on the form.

If the request succeeds, the `formErrors` state variable is cleared of any previous errors, and then the `navigate()` function from React-Router is used to issue a redirect to the login page.

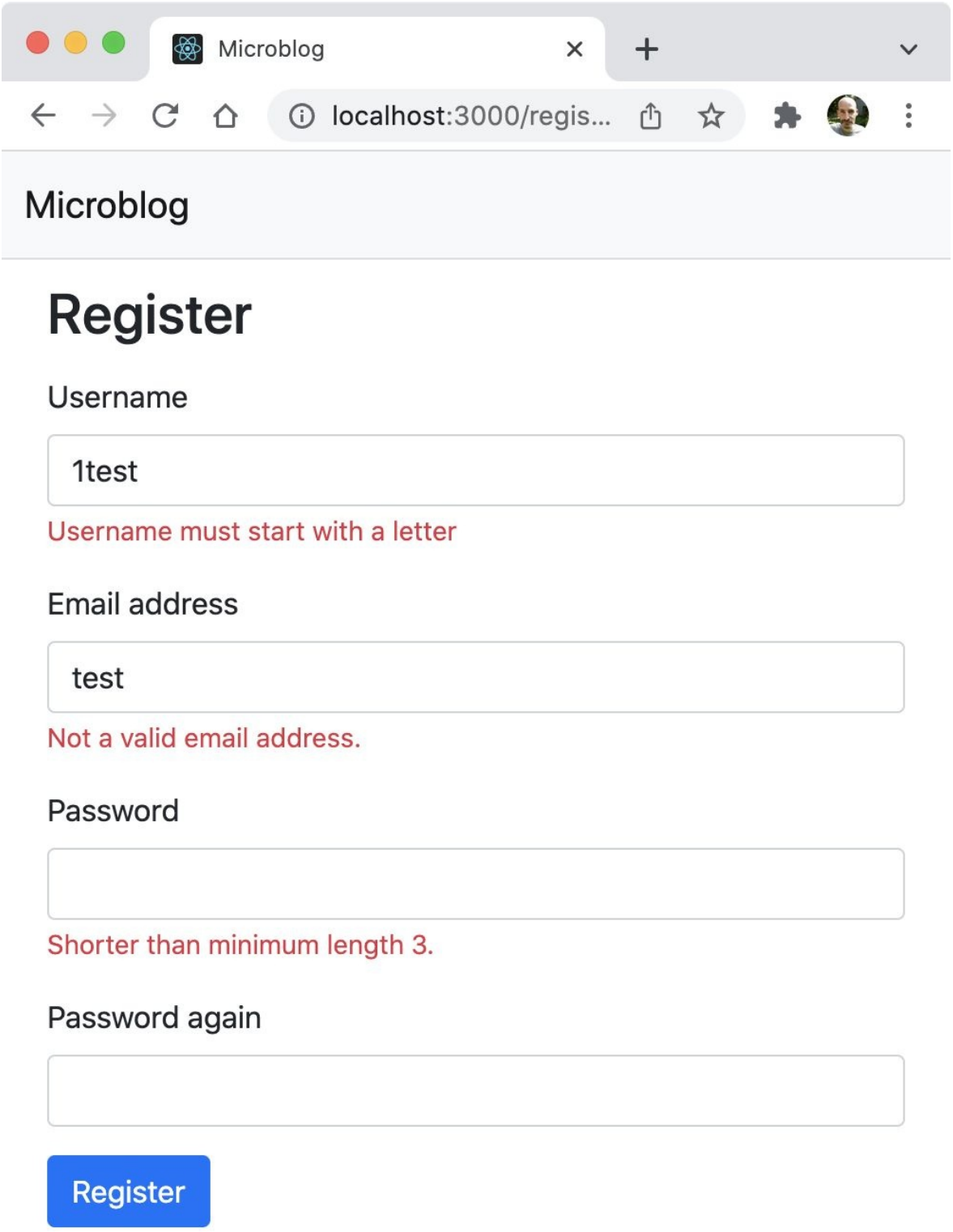


Figure 24 Sever-side validation errors in the registration form

The user registration functionality is now fully functional. You can register users

by navigating to *http://localhost:3000/register* in your browser and submitting the registration form.

Flashing Messages to the User

If you tried to register a user, you must have noticed that after the user registration is processed, you are unceremoniously redirected to the login page, without any indication of success or failure.

A standard user interface pattern in web applications is to indicate the status of an operation by "flashing" a message to the user. A flashed message often appears at the top of the page, styled to call attention to it. In React-Bootstrap, the Alert component can create this type of user interface component.

Implementing message flashing in a reusable way presents an interesting challenge. To flash a message, a component needs to share the message to be flashed with the component that renders the alert message, which will very likely be in a different part of the component tree.

Before going into implementation specifics, let's think about a nice design for flashing a message. Below you can see how an example `MyForm` component might flash a message after processing a form:

```
export default function MyForm() {
  const { flash } = useFlash();

  const onSubmit = (ev) => {
    ev.preventDefault();
    ... // form processing here
    flash('Your registration has been successful', 'success');
  };

  return ( ... );
  }
}
```

This is a really nice pattern, because any component that needs to flash a message can just get the `flash()` function from the `useFlash()` hook, without having to worry about how or where the alert is going to render in the page.

But how do you implement something like this? React contexts are actually powerful enough to make a solution like this work. The component hierarchy that supports message flashing for the example `MyForm` component above might look like this:

```
<FlashProvider>
  <Alert />
  <MyForm />
</FlashProvider>
```

And this is starting to look more familiar. The `FlashProvider` component can share a context that includes a `flash()` function, which child components such as the `MyForm` of the example above can use to set the alert message. The context will also need to share the text and style of the message, so that the component in charge of rendering the alert, also a child of `FlashProvider`, has the information it needs to do it.

The actual implementation has an additional complication omitted above. On a traditional, server-rendered application, a flashed message goes away as soon as the user navigates to a new page. Since real page navigation does not exist in a React application, an alert may end up being displayed for a long time, so a mechanism to hide an alert after a reasonable amount of time needs to be included in this solution.

The listing below shows the complete implementation of `FlashProvider`, with support for automatically hiding the alert after a specified number of seconds.

Listing 59 src/contexts/FlashProvider.js: A Flash context

```
import { createContext, useContext, useState } from 'react';

export const FlashContext = createContext();
let flashTimer;

export default function FlashProvider({ children }) {
  const [flashMessage, setFlashMessage] = useState({});
  const [visible, setVisible] = useState(false);

  const flash = (message, type, duration = 10) => {
    if (flashTimer) {
      clearTimeout(flashTimer);
```

```
      flashTimer = undefined;
    }
    setFlashMessage({message, type});
    setVisible(true);
    if (duration) {
      flashTimer = setTimeout(hideFlash, duration * 1000);
    }
  };

  const hideFlash = () => {
    setVisible(false);
  };

  return (
    <FlashContext.Provider value={{flash, hideFlash, flashMessage,
      {children}
    </FlashContext.Provider>
  );
}

export function useFlash() {
  return useContext(FlashContext).flash;
}
```

The overall structure of this component is similar to that of `ApiProvider`. The `FlashContext` object is created in the global scope, and the `FlashContext.Provider` component is then rendered as a wrapper to the component's children. The value shared by this context is an object with four elements:

- `flash` is the function that components can use to flash a message to the page.
- `hideFlash` is a function that updates the visible state of the flash message to `false`.
- `flashMessage` is an object with `message` and `type` properties, defining the alert to display. The `type` property can be any of the styling options supported by Bootstrap's alerts, for example: `success`, `danger`, `info`, etc.
- `visible` is the current visible state of the alert.

The `value={{ ... }}` prop of the `FlashContext.Provider` component has a syntax that may look strange. In React, when a prop needs to be assigned a value that is an object, two sets of braces are required. The outer pair of braces is what

tells the JSX parser that the value of the prop is given as a JavaScript expression. The inner pair of braces are the object braces. The elements of the object, which are normally provided in `key: value` format, are in this case given as simple variables, using the object property shorthand, which takes the key and the value from the same variable.

The component has two state variables. The `flashMessage` state variable holds the message and the type of the current alert. The `visible` variable is a boolean that keeps track of when the alert is displayed.

The `flashTimer` global variable is going to manage a JavaScript timer instance that is created each time an alert is displayed, with the purpose of automatically hiding the alert after enough time has passed. You may wonder why this variable is declared as a global variable and not a state variable inside the component. State variables have the unique feature that they cause any components that use them to re-render when they change. The timer used by this component is an internal implementation value that has no connections to anything that is visible on the page that might need to re-render, so for that reason a global variable is used.

The `flash()` function is defined with three arguments `message`, `type` and `duration`. The function starts by checking if there is an active flash timer. If there is a timer, that means that there is currently an alert on display that is going to be replaced. In that case the timer needs to be canceled, since a new timer will be created for the new alert.

The function then updates the `flashMessage` state with the provided `message` and `type` arguments, sets the `visible` state to `true`, and finally creates a timer that calls the `hideFlash()` function after the number of seconds given in `duration` have passed. The `duration` argument defaults to 10 seconds, and the caller can pass 0 to skip the timer creation and display an alert that remains visible until the user closes it manually.

The `hideFlash()` function just sets the `visible` state variable to `false`, to indicate that the alert should now be hidden.

The JSX returned by this component just creates the context provider with all the children components inside.

As in previous contexts, a `useFlash()` custom hook function is defined to return

the `flash` function. This is a convenience function that all components can use to easily flash a message, without having to deal with the context directly.

The flash context needs to be placed high enough in the component tree that all the components that may need to flash messages, plus the alert component that renders these messages are all children. As before, it is a good idea to do this in the `App` component, where all the application wide behaviors are defined.

Listing 60 *src/App.js*: The flash context

```
... // <-- no changes to existing imports
import FlashProvider from './contexts/FlashProvider';

export default function App() {
  return (
    <Container fluid className="App">
      <BrowserRouter>
        <FlashProvider>
          <ApiProvider>
            <Header />
            <Routes>
              ... // <-- no changes to routes
            </Routes>
          </ApiProvider>
        </FlashProvider>
      </BrowserRouter>
    </Container>
  );
}
```

The `FlashProvider` component is inserted as a parent of `ApiProvider`, which was the top-most application-specific component. This would enable `ApiProvider` and all of its children to work with flashed messages.

The FlashMessage Component

The second part of this solution is the component that displays the flashed messages. This is the task of the `FlashMessage` component. Below is the implementation of this component.

Listing 61 src/components/FlashMessage.js: Display a flashed message

```
import { useContext } from 'react';
import Alert from 'react-bootstrap/Alert';
import Collapse from 'react-bootstrap/Collapse';
import { FlashContext } from '../contexts/FlashProvider';

export default function FlashMessage() {
  const { flashMessage, visible, hideFlash } = useContext(FlashCor

  return (
    <Collapse in={visible}>
      <div>
        <Alert variant={flashMessage.type || 'info'} dismissible
          onClose={hideFlash}>
          {flashMessage.message}
        </Alert>
      </div>
    </Collapse>
  );
}
```

To be able to render alerts, this component needs access to the data shared by the `FlashProvider` component. The `useFlash()` hook function only provides access to the `flash()` function, which components can use to display an alert. Because this component needs access to the remaining elements of `FlashContext`, it accesses the context with the `useContext` hook. There is no great benefit in adding custom hook functions for these attributes of the flash context, since their use is limited to this component.

This component takes advantage not only of the Alert component of React-Bootstrap, but also Collapse, which adds a nice sliding animation when the alert is shown or hidden. The `in` prop of `Collapse` determines if the components needs to be shown or hidden, so it is directly assigned the value of the `visible` state variable that was obtained from the flash context.

The `Collapse` documentation indicates that to have a smooth animation it is often necessary to wrap the collapsible elements in a `<div>`, so this is done here to wrap `Alert`. The alert itself uses the attributes of `flashMessage` to configure the styling and the message.

To make these alerts more friendly, the `dismissible` prop is added, so that there is a close button on the alert that the user can click to immediately dismiss it, without having to wait for the timer to do it. The `onClose` prop is the handler for the close action, which is directly sent to the same `hideFlash` function that the timer uses.

The `FlashMessage` can be added to the `Body` component, so that it is available in all the pages of the application, above the content area:

Listing 62 *src/components/Body.js*: Show a flashed message in the page

```
... // <-- no changes to existing imports
import FlashMessage from './FlashMessage';

export default function Body({ sidebar, children }) {
  return (
    <Container>
      <Stack direction="horizontal" className="Body">
        {sidebar && <Sidebar />}
        <Container className="Content">
          <FlashMessage />
          {children}
        </Container>
      </Stack>
    </Container>
  );
}
```

The `FlashMessage` component is inserted as the first child of the `Content` container, so that it appears on top of any page content. When the sidebar is enabled, the alert only covers the extent of the content portion.

The `RegistrationPage` component can now retrieve the `flash()` function from the `useFlash()` hook, and display a success message after registration.

Listing 63 *src/pages/RegistrationPage.js*: Flash a message after registration

```
... // <-- no changes to existing imports
import { useFlash } from '../contexts/FlashProvider';

```

```
export default function RegistrationPage() {
  ... // <-- no changes to state variables, references and other h
  const flash = useFlash();

  ... // <-- no changes to side effect function

  const onSubmit = async (event) => {
    event.preventDefault();
    if (passwordField.current.value !== password2Field.current.val
      ... // <-- no changes to client-side validation
    }
    else {
      ... <-- no changes
      if (!data.ok) {
        ... // <-- no changes
      }
      else {
        setFormErrors({});
        flash('You have successfully registered!', 'success');
        navigate('/login');
      }
    }
  };

  ... // <-- no changes to returned JSX
}
```

Figure 25 shows how the flashed message looks after a user is registered.

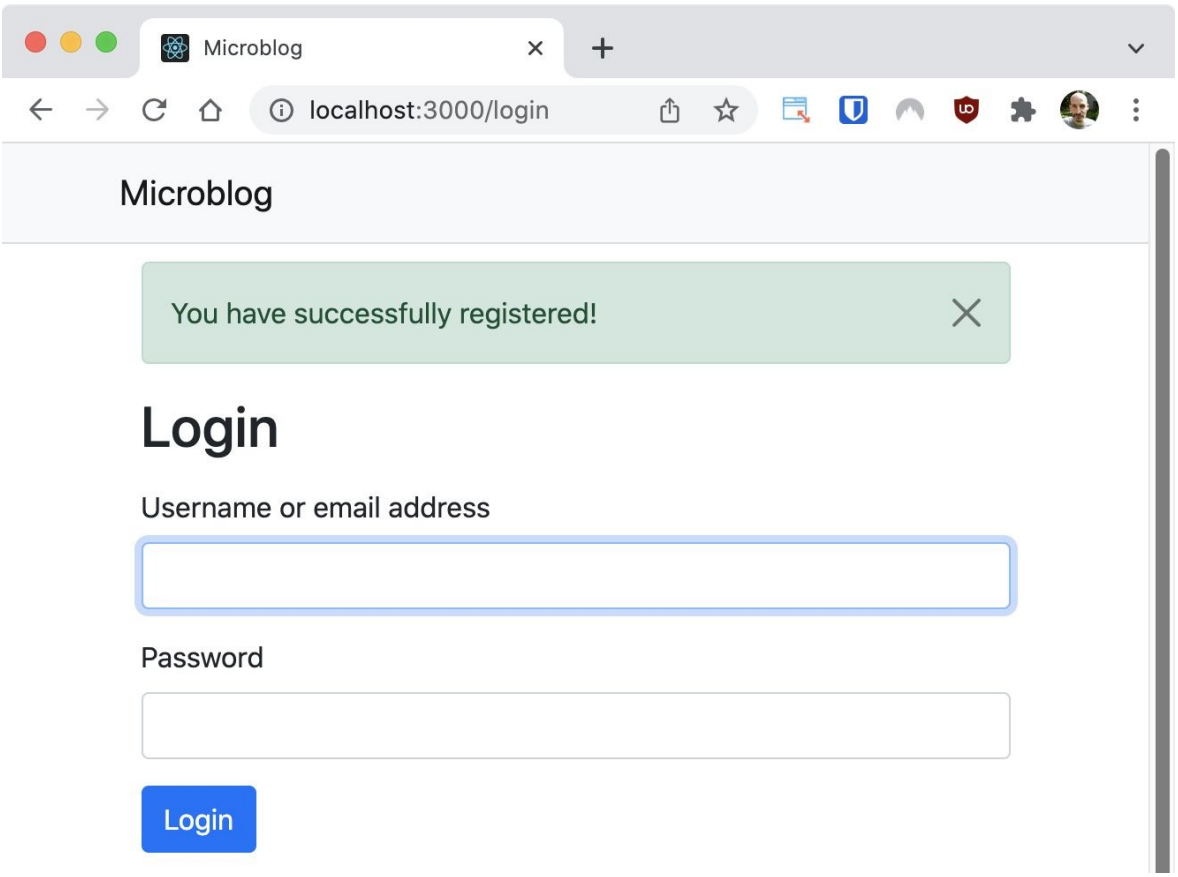


Figure 25 Flashed message

Chapter Summary

- Use a React reference to access DOM elements in side effect or event handler functions.
- Use DOM APIs to obtain or set the value of an uncontrolled input component through a reference.
- Use client-side validation for forms as a complement, but not as a replacement for server-side validation.
- A React context is not only useful when a parent needs to share data with its children. It can also be used to enable children components to pass information between themselves with the parent as intermediary.

Authentication

Up to this point, you have been using the Microblog API back end with an option to bypass authentication. This enabled the project to grow without having to deal with the highly complex matter of user authentication up front. In this chapter you will finally learn how to do this.

Enabling Back End Authentication

Back in Chapter 5 you installed the Microblog API back end, and as part of the setup instructions, you added the `DISABLE_AUTH=true` configuration option.

You now need to change this option to `DISABLE_AUTH=false`. If you deployed Microblog API locally, either in a Docker container or as a Python process, change this setting in the *.env* file located in the top-level directory of the Microblog API project, and then stop and restart your back end for the change to take effect.

If you deployed Microblog API on Heroku, visit your application dashboard and select your deployed application to reconfigure it. From the application page select "Settings", scroll down to "Config vars" and click the "Reveal Config Vars" button to edit the value of `DISABLE_AUTH`. Heroku automatically restarts your application after the configuration is changed.

Keep in mind that as soon as you activate authentication on the back end, your React application will lose the ability to communicate with it, as many requests will not return a 401 status code and deny access. For requests that render a spinner while data is loaded, the spinner will stay visible indefinitely. All these issues are going to be fixed as the authentication is implemented in this chapter.

Authentication in the API Client

The most important benefit of having a dedicated API client class is that all aspects of working with the API are implemented in a single place. Because authentication support will be added inside the class, none of the application components that make API calls will need to change.

The Microblog API authentication flow is as follows:

- The client sends a `POST` request to */api/tokens*, passing the username and password in a standard basic authentication header.
- If the credentials are correct, the status code in the response is 200, and the body includes the `access_token` attribute. To enable token security best practices, the access token has a short life, and a refresh token that can be used to renew it is returned in a secure and HTTP-only cookie.
- To make authenticated API calls, the client must include a standard Bearer authentication header with a valid access token in all requests.
- If an API call is made with an access token that is incorrect or expired, a response with status code 401 is returned.

While in general terms the solution presented in this chapter can be adapted to fit your own projects, you will need to customize the specific authentication flow according to the requirements of your server.

Passing the Access Token in API Calls

The `fetch()` call in the `request()` method needs to be expanded to include an `Authorization` header with the `Bearer` scheme.

Listing 64 src/MicroblogApiClient.js: Include bearer token header

```
      response = await fetch(this.base_url + options.url + query,
        method: options.method,
        headers: {
          'Content-Type': 'application/json',
          'Authorization': 'Bearer ' + localStorage.getItem('acces
          ...options.headers,
        },
        body: options.body ? JSON.stringify(options.body) : null,
      });
}
```

The access token is retrieved from the browser's local storage. You will see later that when an access token is returned by the back end, it is immediately written to this storage. If there is no previously stored access token in the local storage, then `null` will be returned.

You may be wondering why this code isn't checking that the access token is not `null` before including it in the request. The assumption is that if the client object doesn't have an access token, then a different `Authorization` header will be included in the `options` object passed by the caller, which overrides the `Authorization` header defined here.

Storing the token in local storage makes it possible for the application to "remember" the authenticated user when the user refreshes the browser page, when the site is opened in multiple tabs, and when the browser is closed and reopened, which is a behavior that most users expect from a web application. However, you should keep in mind that storing sensitive information in local storage presents a risk if your application is vulnerable to cross-site scripting (XSS) attacks.

An alternative implementation that avoids the risk of getting tokens compromised in an XSS attack is to use an instance variable in the `MicroblogApiClient` class to store the access token instead of local storage. While this solution increases security, the user experience would be severely degraded, as users would be required to authenticate every time a session with the application is started, even after a page refresh.

Cross-Site Scripting Attacks

You've learned in the previous section that storing sensitive information in the browser's local storage is sometimes considered a security risk, yet this is exactly what this project is doing. So this is a good time to discuss these risks in detail.

An XSS attack involves the attacker figuring out a way to insert malicious JavaScript into an application running on the user's browser. There are two basic ways in which this can happen:

- The attacker finds a way to break into the server that hosts the application's JavaScript files and makes modifications to them so that hacked versions with malicious code are served to clients.
- The attacker tricks the application running in the browser into rendering a `<script>` tag with malicious JavaScript code.

If you are using a third-party service such as Heroku to serve your React application, then the first scenario is the responsibility of your service provider.

If you host your React application yourself, then you must use standard server hardening techniques such as passwordless logins, use of a firewall, closing any unnecessary network ports, etc.

React provides decent protection against the second attack vector, as long as you render all the content in your application through JSX. Protection against XSS attacks in React consists in applying escaping to all the text that is included in JSX contents returned by components. This escaping is always applied, there is no need to enable this protection.

To keep your application well protected, it is extremely important to avoid the temptation to bypass JSX and render contents to the page directly through DOM APIs, as this would not have any protection against XSS attacks.

When all these security concerns are addressed, the risk of a React application being the victim of an XSS attack is extremely low. Even though it would be unlikely for an access token to be compromised, it is considered a good practice to use short expirations on these tokens, so that if an attacker manages to steal a token by some unknown attack method, the damage that can be done with it is limited. Microblog API provides access tokens that last only 15 minutes in the default configuration. The tokens can only be renewed with a refresh token that is stored in a secure cookie, inaccessible from the browser's JavaScript environment.

Logging In and Out

How is the access token obtained? To receive a token, the client must send a `POST` request to */api/tokens*, with the username and password entered by the user on the login page.

Instead of sending this request directly from the `LoginPage` component, the `MicroblogApiClient` class can implement a `login()` method with the required logic that generates the basic authentication header.

Listing 65 src/MicroblogApiClient.js: Login method

```
async login(username, password) {
  const response = await this.post('/tokens', null, {
    headers: {
```

```
        Authorization:  'Basic ' + btoa(username + ":" + password)
      }
    });
    if (!response.ok) {
      return response.status === 401 ? 'fail' : 'error';
    }
    localStorage.setItem('accessToken', response.body.access_toker
    return 'ok';
  }
```

Here you can see how the `Authorization` header is passed as an option in the third argument to `api.post()`, to override the default bearer token header added in the previous section. The second argument is used for the body of the request, which is set to `null` because in this particular case there is no body required.

In case you are curious, the `Authorization` header used in this method follows the Basic Authentication format defined by the HTTP specification, which requires encoding the username and password as a base64 string, done here by the `btoa()` function available in JavaScript.

There are three possible outcomes for this function. The two most obvious ones indicate authentication success or failure. A third less likely case occurs when the authentication request fails due to an unexpected issue, and not because the credentials are invalid. The return value of this function can be `ok`, `fail` or `error` to represent these three cases. The error case occurs when the authentication request returns a failure status code other than 401.

If the server returns a successful response, then the access token is written to the browser's local storage, so that it is used in all subsequent requests.

The user should be given the option to log out of the application when desired. To log a user out, the access token should be removed. A helper `logout()` method in the `MicroblogApiClient` class performs this action.

Listing 66 src/MicroblogApiClient.js: Logout method

```
  async logout() {
    await this.delete('/tokens');
    localStorage.removeItem('accessToken');
  }
```

The `logout()` method makes a request to the token revocation endpoint of Microblog API, which ensures that the access token cannot be used again. It also removes the token from local storage so that the React application completely forgets about it.

For convenience, an `isAuthenticated()` method is also added to the client class. The application can use it to check if there is an authenticated user or not.

Listing 67 src/MicroblogApiClient.js: isAuthenticated method

```
  isAuthenticated() {
    return localStorage.getItem('accessToken') !== null;
  }
```

A User Context and Hook

An important function of the authentication system is to provide information about the logged-in user to any components that need it. The preferred way to share the user with the application's components is with a context and hook combination.

The `UserContext` will share an object with four attributes:

- `user`: the currently logged-in user, or `null` if the user is not logged in. A value of `undefined` indicates that the user is being retrieved from the server.
- `setUser`: a setter for the user.
- `login`: a helper function to log the user in with the username and password provided.
- `logout`: a helper function to log the user out.

The complete implementation of the user context and its associated hook function is shown below.

Listing 68 src/contexts/UserProvider.js: User context and hook

```
import { createContext, useContext, useState, useEffect } from 're
import { useApi } from './ApiProvider';

const UserContext = createContext();
```

```
export default function UserProvider({ children }) {
  const [user, setUser] = useState();
  const api = useApi();

  useEffect(() => {
    (async () => {
      if (api.isAuthenticated()) {
        const response = await api.get('/me');
        setUser(response.ok ? response.body : null);
      }
      else {
        setUser(null);
      }
    })();
  }, [api]);

  const login = async (username, password) => {
    const result = await api.login(username, password);
    if (result === 'ok') {
      const response = await api.get('/me');
      setUser(response.ok ? response.body : null);
      return response.ok;
    }
    return result;
  };

  const logout = async () => {
    await api.logout();
    setUser(null);
  };

  return (
    <UserContext.Provider value={{ user, setUser, login, logout }}
      {children}
    </UserContext.Provider>
  );
}

export function useUser() {
  return useContext(UserContext);
}
```

The `UserProvider` component defines a `user` state variable that will contain the

details of the authenticated user. The state variable is initialized with a value of `undefined`, which is used while the user is being retrieved from the server. The value will change to the user details returned by the server as soon as they are available. The value will change to `null` if there is no authenticated user.

A side effect function is used to try to resolve the value of the `user` state variable when the component is rendered for the first time. When the `api` instance is authenticated, the function loads the user's details by calling the */api/me* endpoint of Microblog API. The state variable remains set to the initial `undefined` value until the user information is returned, so any components that need to render user information can display a spinner or similar UI component while waiting.

If the `api` instance does not have an access token, then the `user` state variable is set to `null` to indicate to the rest of the application that there is no user logged in.

The `login()` helper function accepts username and password arguments. This function will be called by the `LoginPage` component to log the user in. The function starts by invoking the method of the same name in the API client. If the authentication attempt succeeds, a request to retrieve user information is issued, and the `user` state variable is updated accordingly, in a very similar way to how it was done in the side effect function.

The `logout()` helper function logs the user out in the `api` client instance, and then updates the `user` state variable to `null`, so that all components that use the context know that there is no authenticated user anymore.

The `UserContext.Provider` component sets the value of the context as an object with the `user`, `setUser`, `login` and `logout` keys.

As with all previous contexts, a `useUser()` companion hook function is defined to make it more convenient for components to access the elements of the context. For this hook, the entire object shared in the context is returned. The components accessing the context can use a destructuring assignment to obtain the attributes that they need.

The user context needs to be added to the application. Since this context depends on having access to the API client, it needs to be a child of it. Below is the updated `App` component.

Listing 69 *src/App.js*: Add user context

```
... // <-- no changes to existing imports
import UserProvider from './contexts/UserProvider';

export default function App() {
  return (
    <Container fluid className="App">
      <BrowserRouter>
        <FlashProvider>
          <ApiProvider>
            <UserProvider>
              <Header />
              <Routes>
                ... // <-- no changes to routes
              </Routes>
            </UserProvider>
          </ApiProvider>
        </FlashProvider>
      </BrowserRouter>
    </Container>
  );
}
```

Implementing Private Routes

When designing the authentication subsystem of the application, you have to decide what is the subset of the application routes that are going to be available only to users that have previously authenticated. Let's call these the *private* routes of the application.

When a client that is not authenticated attempts to access a private route, the best course of action is to redirect the user to the login page, and once the user submits the authentication form, redirect back to the route that was initially attempted.

The `PrivateRoute` component shown below can be used as a parent for any components that are only allowed to render when there is an authenticated user.

Listing 70 *src/components/PrivateRoute.js*: Private route component

```
import { useLocation, Navigate } from 'react-router-dom';
import { useUser } from '../contexts/UserProvider';

export default function PrivateRoute({ children }) {
  const { user } = useUser();
  const location = useLocation();

  if (user === undefined) {
    return null;
  }
  else if (user) {
    return children;
  }
  else {
    const url = location.pathname + location.search + location.has
    return <Navigate to="/login" state={{next: url}} />
  }
}
```

The component's render function uses the `useUser()` hook from the previous section to retrieve the `user` state variable. If this variable is `undefined`, it means that the application isn't ready to provide user information yet. In this situation, this component cannot do anything, so it returns `null` to not render anything. This isn't a problem because the `undefined` value for the user is temporary. As soon as the `user` state variable in the user context resolves to `null` or to the user's details, this component will re-render.

When the `user` is set to a truthy value, it means that there is an authenticated user, and in that case this component renders its children.

The interesting case is when `user` is `null`, indicating that the user is not authenticated. In this situation, the children of this component cannot be rendered, because they require a user to be logged in. The component renders a `Navigate` component from React-Router, which sends the user to the login page is rendered instead.

As mentioned earlier, the intention is to redirect the user back to the original page after authentication is completed, so it is necessary to store the current URL so that it can be used later. To determine what is the current URL, React-Router provides a `useLocation()` hook, which returns a location object, very much like the browser's own document.location. From this object, the `pathname`, `search` and `hash` attributes are concatenated to form a single URL string to save.

The `Navigate` component supports a `state` prop in which the application can store any custom data to be preserved in the `location` object. This is the perfect place to write the URL that the user needs to be sent back to after authentication. The URL is stored in the `state` prop in an object, under a `next` key.

Public Routes

The login and user registration routes have the interesting property that they have no purpose after the user logs in. To prevent a logged-in user from navigating to these routes, it is a good idea to also create a `PublicRoute` component that only allows its children to render when the user is not logged in, or else it redirects the user to the root URL of the application.

Below is the complete implementation of `PublicRoute`, following a style that is very similar to that of `PrivateRoute`.

Listing 71 *src/components/PublicRoute.js*: Public route component

```
import { Navigate } from 'react-router-dom';
import { useUser } from '../contexts/UserProvider';

export default function PublicRoute({ children }) {
  const { user } = useUser();

  if (user === undefined) {
    return null;
  }
  else if (user) {
    return <Navigate to="/" />
  }
  else {
    return children;
  }
}
```

Routing Public and Private Pages

How are the `PrivateRoute` and `PublicRoute` components defined in the previous sections used? There are a couple of different options, but these

components act as wrappers of the route components.

A simple method to use these components is to add them directly in the `Route` definitions, wrapping the intended page components. For example:

```
<Route path="/explore" element={
  <PrivateRoute><ExplorePage /></PrivateRoute>
} />
```

Given that most pages in this application are going to be private, it can get noisy to have lots of `PrivateRoute` wrappers. A less verbose alternative is to separate the public and private routes and then use a single `PrivateRoute` wrapper for the entire group of private routes.

Below are the changes to `App` to apply the route wrappers:

Listing 72 *src/App.js*: Routing of public and private routes

```
... // <-- no changes to existing imports
import PrivateRoute from './components/PrivateRoute';
import PublicRoute from './components/PublicRoute';

export default function App() {
  return (
    ... // <-- no changes to outer components

    <Routes>
      <Route path="/login" element={
        <PublicRoute><LoginPage /></PublicRoute>
      } />
      <Route path="/register" element={
        <PublicRoute><RegistrationPage /></PublicRoute>
      } />
      <Route path="*" element={
        <PrivateRoute>
          <Routes>
            <Route path="/" element={<FeedPage />} />
            <Route path="/explore" element={<ExplorePage />} />
            <Route path="/user/:username" element={<UserPage />} /
            <Route path="*" element={<Navigate to="/" />} />
          </Routes>
        </PrivateRoute>
```

```
      } />
    </Routes>

    ... // <-- no changes to outer components
  );
}
```

With this solution, only the public routes of the application are defined as top-level routes. The two public routes that map to the login and registration pages need to be disabled once the user logs in, so both are wrapped individually with `PublicRoute`.

If none of the public routes are a match, then a catch-all `*` route is defined with an inner `<Routes>` component that includes the remaining routes, which are all private and are wrapped with a single `PrivateRoute` component.

Hooking Up the Login Form

Most of the low-level pieces of the authentication solution are now in place, so the `LoginPage` component can now be completed, so that it actually performs the authentication procedure.

Listing 73 src/pages/LoginPage.js: Log users in

```
import { useState, useEffect, useRef } from 'react';
import { Link, useNavigate, useLocation } from 'react-router-dom';
import Form from 'react-bootstrap/Form';
import Button from 'react-bootstrap/Button';
import Body from '../components/Body';
import InputField from '../components/InputField';
import { useUser } from '../contexts/UserProvider';
import { useFlash } from '../contexts/FlashProvider';

export default function LoginPage() {
  ... // <-- no changes to existing state and references
  const { login } = useUser();
  const flash = useFlash();
  const navigate = useNavigate();
  const location = useLocation();

  ... // <-- no changes to side effect function
```

```
  const onSubmit = async (ev) => {
    ... // <-- no changes to existing submit logic

    const result = await login(username, password);
    if (result === 'fail') {
      flash('Invalid username or password', 'danger');
    }
    else if (result === 'ok') {
      let next = '/';
      if (location.state && location.state.next) {
        next = location.state.next;
      }
      navigate(next);
    }
  };

  ... // <-- no changes to returned JSX
}
```

From the four hooks used in this function, only `useNavigate()` is new. This hook is from React-Router, and provides access to a `navigate()` function that is similar to the `Navigate` component, but in function form.

The new logic to log the user in is added at the end of the `onSubmit()` function, after all the validation checks have passed. Note that this function was converted to `async`, so that `await` can be used.

To initiate the authentication, the `login()` function from the user context is called with the `username` and `password` values entered by the user in the form. The `login()` function is asynchronous, so it is awaited. The return value is the string `ok` when the user is successfully authenticated, `fail` when the authentication fails, or `error` when an unexpected error prevented authentication to be carried out.

In case of an authentication failure, an error message is flashed, and the user remains on the login page to try again.

If the authentication call succeeds, then the API client now has an access token and the user context has the user details to share with other components, so the user can be redirected to any of the private routes of the application.

Normally the redirect is to the root URL, which is the user's feed page, but if `location.state` has a `next` attribute, then the redirect goes to this route, which is the one that was saved by the `PrivateRoute` component when the user attempted to access the page without being logged in.

The third possibility occurs when the authentication call fails due to an unexpected reason. This case is not handled by this component and will instead be handled later with an application-wide error handler.

The application now has a working login procedure. You should be able to register a new user (if you haven't done that already), and then log in. Once you are logged in, your feed page is going to be empty, since you aren't following any users yet. You can navigate to the Explore page to confirm that blog posts from other users are displayed.

As discussed above, access tokens returned by Microblog API are valid for 15 minutes, after which they expire. At this time the `MicroblogApiClient` class does not have the ability to "refresh" an expired access token, so the application may appear to stop working if you use it long enough for the access token to expire. Refresh support for tokens will be added soon, but in the meantime, if you reach the token expiration time you will need to refresh the page to trigger a new login.

User Information in the Header

Application components can now add logic to render themselves differently when a user is logged in versus when not. This makes it possible to add a common user interface component found in many websites: a dropdown in the top-right corner with account related options.

Below is the new version of `Header` that includes a dropdown with options to access the user's profile page and to logout of the application.

Listing 74 src/components/Header.js: Show a user account dropdown

```
import Navbar from 'react-bootstrap/Navbar';
import Container from 'react-bootstrap/Container';
import Nav from 'react-bootstrap/Nav';
import NavDropdown from 'react-bootstrap/NavDropdown';
```

```
import Image from 'react-bootstrap/Image';
import Spinner from 'react-bootstrap/Spinner';
import { NavLink } from 'react-router-dom';
import { useUser } from '../contexts/UserProvider';

export default function Header() {
  const { user, logout } = useUser();

  return (
    <Navbar bg="light" sticky="top" className="Header">
      <Container>
        <Navbar.Brand>Microblog</Navbar.Brand>
        <Nav>
          {user === undefined ?
            <Spinner animation="border" />
          :
            <>
              {user !== null &&
                <div className="justify-content-end">
                  <NavDropdown title={
                    <Image src={user.avatar_url + '&s=32'} rounded
                  } align="end">
                    <NavDropdown.Item as={NavLink} to={'/user/' +
                      Profile
                    </NavDropdown.Item>
                    <NavDropdown.Divider />
                    <NavDropdown.Item onClick={logout}>
                      Logout
                    </NavDropdown.Item>
                  </NavDropdown>
                </div>
              }
            </>
          }
        </Nav>
      </Container>
    </Navbar>
  );
}
```

The component gets the currently logged-in user from the `useUser()` hook. Right after the `Navbar.Brand` component, a conditional JSX expression is inserted. If the current user is `undefined`, meaning that the application is still trying to figure out who the user is, a `Spinner` component is rendered on the

right side of the navigation bar.

When the `user` is not `undefined` or `null`, a dropdown is created using Nav components from React-Bootstrap. The case of `user` being `null` is handled by not rendering anything in this part of the navigation bar. An alternative that might work for many applications is to render a login link or button, but given that this application automatically redirects to the login form, this does not seem necessary.

The dropdown is created with the avatar image of the logged-in user, which is available in `user.avatar_url`. As with avatar images displayed in blog posts, the image is provided by the Gravatar service for the email registered in the account. For emails that do not have an avatar image registered with the service, Gravatar returns a unique geometric design. You can associate an image with your email address by visiting the Gravatar site.

The "Profile" option is created using the `NavDropdown.Item` component from React-Bootstrap, customized to be a `NavLink` component from React-Router. Recall that this technique was also used to create React-Router compatible links in the sidebar. The `to` prop of the link is dynamically set to the */api/user/{username}* URL for the logged-in user.

The logout option in the dropdown has an `onClick` handler that invokes the `logout()` function obtained from the `useUser()` hook, which logs the user out and then redirects to the main route of the application.

The `NavDropdown.Divider` component creates a divider line between the two options. In case you want to learn more about creating great looking sidebars, the design of this dropdown is based on the examples of navigation bars with dropdowns in the NavDropdown documentation.

To ensure that the options in this dropdown have a look that is consistent with the links in the sidebar, the foreground and background color definitions added earlier for the sidebar can be extended to also apply to the `.dropdown-item.active` class.

Listing 75 src/index.css: Styling of dropdown options

```
... // <-- no changes to other styles
```

```
.Sidebar a, .dropdown-item, .dropdown-item.active {
  color: #444;
}

.Sidebar .nav-item .active, .dropdown-item.active {
  background-color: #def;
}
```

Figure 26 shows the account dropdown.

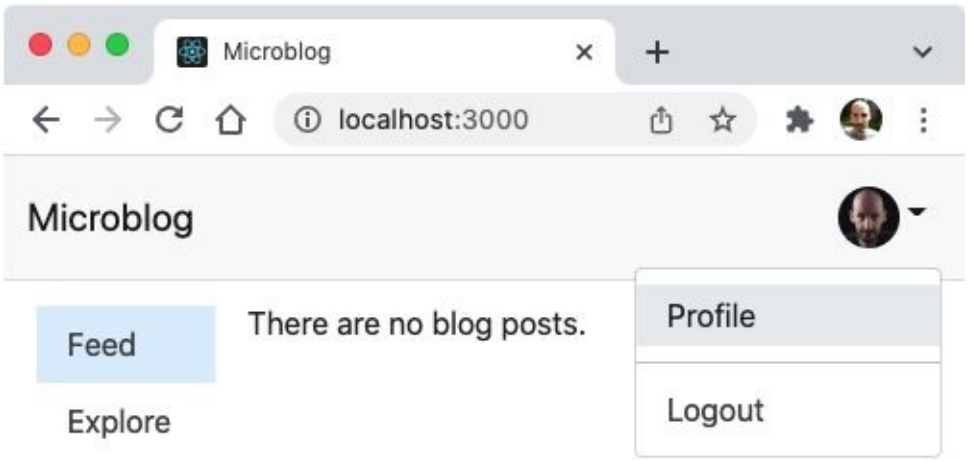


Figure 26 User account dropdown

Handling Refresh Tokens

If the client receives a 401 status code while using a previously valid access token, then this indicates that the token has expired and needs to be "refreshed".

As mentioned before, using short-lived tokens is a well accepted security practice in APIs, designed to limit the potential damage that can be caused if an access token is compromised.

When the access token expires, the client can request a new access token by sending a `PUT` request to */api/tokens*, passing the expired access token in the body of the request. In addition to the expired access token, the client must provide a refresh token through a cookie that the server set during the initial authentication flow. The cookie that stores the refresh token is secured so that it cannot be stolen via JavaScript based attacks.

One of the great benefits of having a dedicated API client class is that the refresh logic does not need to be known outside this class. The rest of the application can send requests normally, and whenever a token refresh is necessary, the process can be handled internally by the class without affecting the rest of the application. Using pseudocode, the request sending logic can be expanded to support transparent token refreshes as follows:

- Send the request
- If the response is not 401
  - Return response to caller
- Else
  - Refresh access token
  - Send original request again with new access token
  - Return response to caller

To prevent the code in the `request()` method of the API client from getting too complicated, the refresh logic can be implemented in a separate method. Start by renaming `request()` to `requestInternal()`, and creating a new `request()` wrapper method that calls it.

Listing 76 src/MicroblogApiClient.js: Request wrapper method

```
export default class MicroblogApiClient {
  async request(options) {
    let response = await this.requestInternal(options);
    return response;
  }

  async requestInternal(options) {
    ... // <-- original request() code here
  }

  ... // <-- no changes to other methods
```

```
}
```

The token refresh requests that are sent to Microblog API must include the refresh token cookie that the API set when the user first authenticated. The `fetch()` function used to make the requests does not send cookies by default, so it needs to be told to do it for this request by adding the `credentials` option. The change to add this option in the `requestInternal()` method is shown in the listing below.

Listing 77 src/MicroblogApiClient.js: Include cookies in requests

```
      response = await fetch(this.base_url + options.url + query,
        method: options.method,
        headers: {
          'Content-Type': 'application/json',
          'Authorization': 'Bearer ' + localStorage.getItem('acces
          ...options.headers,
        },
        credentials: options.url === '/tokens' ? 'include' : 'omit
        body: options.body ? JSON.stringify(options.body) : null,
      });
}
```

The `credentials` option is set to `include` only when the URL is that of the refresh token endpoint. In all other cases it is best to avoid sending unnecessary cookies by setting this option to `omit`.

Now the new `request()` wrapper method can be expanded to refresh the token and retry the original request when necessary.

Listing 78 src/MicroblogApiClient.js: Refresh token logic

```
export default class MicroblogApiClient {
  async request(options) {
    let response = await this.requestInternal(options);
    if (response.status === 401 && options.url !== '/tokens') {
      const refreshResponse = await this.put('/tokens', {
        access_token: localStorage.getItem('accessToken'),
      });
      if (refreshResponse.ok) {
        localStorage.setItem('accessToken', refreshResponse.body.a
```

```
        response = await this.requestInternal(options);
      }
    }
    return response;
  }

  ... // <-- no changes to other methods
}
```

The refresh logic is only used when the original request comes back with a 401 status code, and the URL is not the one for the refresh token endpoint already. Checking the URL is useful to avoid a possible endless loop if the refresh token endpoint also fails with a 401 status code, maybe due to an expired or missing refresh token.

To refresh the token, a `PUT` request to */api/tokens* is sent with the expired access token in the body. The cookie previously set during login will be included, per the `credentials` option added in `requestInternal()`.

If the token refresh endpoint succeeds, then a new access token is returned. This token is written to local storage, replacing the now expired one. For added security, Microblog API will also send a new refresh token and invalidate the one that was just used, but this is in a secure cookie that is handled automatically by the browser.

Once the client is updated with the new access token, the original request can be retried, and this time the response is directly returned, regardless of its status code.

With these changes, a logged-in user should be able to stay on the system for as long as it wants, regardless of access token expirations.

Chapter Summary

- The logged-in user can be shared with the application through a context and custom hook.
- For common route behaviors such as redirecting to a login form or restricting access to logged-in users, create a wrapper component.
- Abstract the authentication logic in the methods of an API client, so that its complexity does not bleed into other parts of the application.
- Never render contents directly to the page with DOM APIs, as this introduces a risk of XSS attacks.

Application Features

By now you have learned most of the React concepts you need to complete this application. This chapter is dedicated to building the remaining features of the application, with the goal of solidifying the knowledge you acquired in previous chapters.

Submitting Blog Posts

Writing a post is arguably the most important feature of this application. The good news is that with the concepts you have learned in previous chapters and the Microblog API documentation, this feature does not present any new challenges.

In terms of the API, a new blog post must be submitted to the server with a `POST` request to */api/posts*. The only field that needs to be included is `text`, with the text of the post. The author and the timestamp for the new post are derived by the server from the access token and current time respectively.

The `Write` component, which implements a form to write blog posts, is shown below.

Listing 79 src/components/Write.js: Blog post write form

```
import { useState, useEffect, useRef } from 'react';
import Stack from "react-bootstrap/Stack";
import Image from "react-bootstrap/Image";
import Form from 'react-bootstrap/Form';
import InputField from './InputField';
import { useApi } from '../contexts/ApiProvider';
import { useUser } from '../contexts/UserProvider';

export default function Write({ showPost }) {
  const [formErrors, setFormErrors] = useState({});
  const textField = useRef();
  const api = useApi();
  const { user } = useUser();
```

```
  useEffect(() => {
    textField.current.focus();
  }, []);

  const onSubmit = async (ev) => {
    ev.preventDefault();
    const response = await api.post("/posts", {
      text: textField.current.value
    });
    if (response.ok) {
      showPost(response.body);
      textField.current.value = '';
    }
    else {
      if (response.body.errors) {
        setFormErrors(response.body.errors.json);
      }
    }
  };

  return (
    <Stack direction="horizontal" gap={3} className="Write">
      <Image
        src={ user.avatar_url + '&s=64' }
        roundedCircle
      />
      <Form onSubmit={onSubmit}>
        <InputField
          name="text" placeholder="What's on your mind?"
          error={formErrors.text} fieldRef={textField} />
      </Form>
    </Stack>
  );
}
```

This component uses a horizontal `Stack` with two slots as the main structure. Similarly to how blog posts are rendered, the user's avatar is shown on the left. The right side of the stack includes a form with a single input field named `text`. This field uses a placeholder text instead of a label, so that the prompt appears inside the field when it is empty. The handling of form errors, autofocusing of the field and submission are done in the same way as in the login and registration forms (see Chapter 7 for details).

The only aspect of this form that is different from previous ones is the `showPost` prop. When a user writes a blog post, the application needs to immediately add the post to the feed at the top position. Since this component does not know anything about modifying the displayed feed, it accepts a callback function provided by the parent component to perform this action. As far as the `Write` component is concerned, all that needs to be done after successfully creating a blog post is to call the function passed in `showPost` with the new post object (which Microblog API returns in the body of the response) as an argument.

The `Stack` component was given a `Write` class name so that it is possible to customize the styling of the form. The styles for this form are shown in the next listing.

Listing 80 *src/index.css*: Styling of the write form

```
... // <-- no changes to existing styles

.Write {
  margin-bottom: 10px;
  padding: 30px 0px 40px 0px;
  border-bottom: 1px solid #eee;
}

.Write form {
  width: 100%;
}
```

The `Write` component is rendered by the `Posts` component, above the list of posts. Because `Posts` is used for all the post lists in the application, an option needs to be added for the parent component to indicate if the blog post write form needs to appear in the page or not. The changes to `Posts` are shown below.

Listing 81 *src/components/Posts.js*: Write form above posts

```
... // <-- no changes to existing imports
import Write from './Write';

export default function Posts({ content, write }) {
  ... // <-- no changes to existing logic
```

```
  const showPost = (newPost) => {
    setPosts([newPost, ...posts]);
  };

  return (
    <>
      {write && <Write showPost={showPost} />}
      ... // <-- no changes to existing JSX
    </>
  );

}
```

The component adds a `write` prop. If this prop has a truthy value, then the JSX for the `Write` component is rendered above the post list.

The `showPost` prop required by the `Write` component is assigned a handler function of the same name. In the function, the `posts` state variable is updated to contain the new post as the first post, with all the other posts after it. The spread operator is used to generate the updated post list.

To complete this feature, the `FeedPage` component needs to render `Posts` with its `write` prop set to `true`. The new version of this component is shown in the next listing.

Listing 82 src/pages/FeedPage.js: Feed page with write form

```
import Body from '../components/Body';
import Posts from '../components/Posts';

export default function FeedPage() {
  return (
    <Body sidebar>
      <Posts write={true} />
    </Body>
  );
}
```

You should now be able to enter blog posts into the system from the feed page. Figure 27 shows how this page looks after a first post was entered into the system.

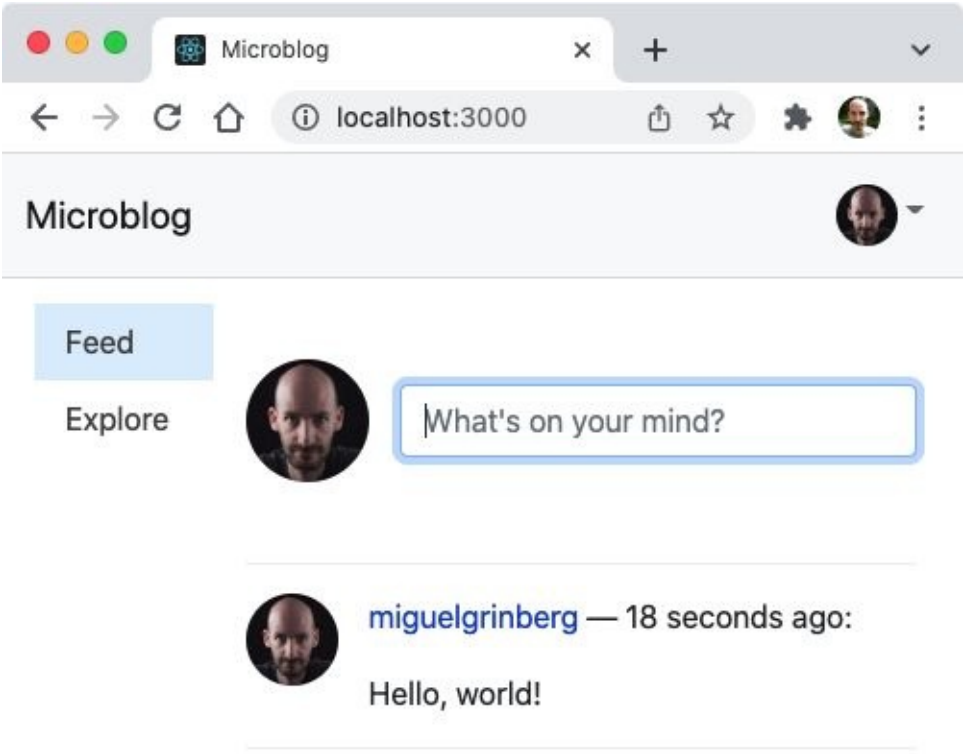


Figure 27 Feed page with write form

User Page Actions

The user page, which currently displays information about a user, can be made more useful by providing some actions, depending on the user being viewed:

- If the page is the profile of the logged-in user, then provide an edit option to change user details.
- If the page is for a user that the logged-in user is not currently following, then provide a follow option.
- If the page is for a user that the logged-in user is already following, then provide an unfollow option.

The complication to implement this is in determining which of the three cases above applies, given a logged-in user and a user being viewed. The straightforward case is when these two users are the same, in which case the "Edit" option should be presented.

To determine if the logged-in user is following a given user or not, a `GET` request

must be sent to */api/me/following/{user_id}*. If the response comes back with status code 204, then the user is already followed. If the user isn't being followed, the response will have a 404 status code.

The listing that follows shows the changes that need to be made to the `UserPage` component to display a button to trigger the correct action out of the three possibilities discussed above.

Listing 83 src/pages/UserPage.js: Action buttons in user page

```
... // <-- no changes to existing imports
import Button from 'react-bootstrap/Button';
import { useNavigate } from 'react-router-dom';
import { useUser } from '../contexts/UserProvider';
import { useFlash } from '../contexts/FlashProvider';

export default function UserPage() {
  ... // <-- no changes to existing state, references and custom h
  const [isFollower, setIsFollower] = useState();
  const { user: loggedInUser } = useUser();
  const flash = useFlash();
  const navigate = useNavigate();

  useEffect(() => {
    (async () => {
      const response = await api.get('/users/' + username);
      if (response.ok) {
        setUser(response.body);
        if (response.body.username !== loggedInUser.username) {
          const follower = await api.get(
            '/me/following/' + response.body.id);
          if (follower.status === 204) {
            setIsFollower(true);
          }
          else if (follower.status === 404) {
            setIsFollower(false);
          }
        }
        else {
          setIsFollower(null);
        }
      }
      else {
```

```
      setUser(null);
    }
  })();
}, [username, api, loggedInUser]);

const edit = () => {
  // TODO
};

const follow = async () => {
  // TODO
};

const unfollow = async () => {
  // TODO
};

return (
  <Body sidebar>
    {user === undefined ?
      <Spinner animation="border" />
    :
      <>
        {user === null ?
          <p>User not found.</p>
        :
          <>
            <Stack direction="horizontal" gap={4}>
              <Image src={user.avatar_url + '&s=128'} roundedCir
              <div>
                ... // <-- no changes to user details

                {isFollower === null &&
                  <Button variant="primary" onClick={edit}>
                    Edit
                  </Button>
                }
                {isFollower === false &&
                  <Button variant="primary" onClick={follow}>
                    Follow
                  </Button>
                }
                {isFollower === true &&
                  <Button variant="primary" onClick={unfollow}>
                    Unfollow
```

```
                    </Button>
                  }
                </div>
              </Stack>
              <Posts content={user.id} />
            </>
          }
        </>
      }
    </Body>
  );
}
```

With these changes, the page has a new state variable `isFollower` that will be `true` if the logged-in user is following the user being viewed, or `false` if it isn't. A value of `null` will be used to indicate that the logged-in user and the viewed user are the same, and as always, `undefined` is the initial value of the state variable, indicating that the side effect function hasn't determined which of the three possible states is the correct state to use.

The `useUser()` hook is used to retrieve the logged-in user. This presents a name collision, since the component already has a state variable called `user` that represents the user being viewed. To disambiguate, the destructuring assignment used with the hook renames the logged-in user to `loggedInUser`.

The side effect function obtains the viewed user and stores it in the `user` state variable, and then needs to determine the value of the `isFollower` state variable, which determines which of the three possible operations apply. If the viewed user is different from the logged-in user, a second request is sent to the back end to obtain the follow relationship between the two users, which is used to update the `isFollower` state variable.

The buttons that present the actions to the user are going to have `onClick` handlers `edit()`, `follow()` and `unfollow()` respectively, all placeholders for now.

In the JSX section, the `isFollower` state variable is used to determine which of the three buttons to display. In the first render of the component, `isFollower` is going to be `undefined`, so none of the buttons will render. Once the side effect function completes and the variable resolves to one of `true`, `false` or `null`, the correct button will display.

Editing User Information

When users view their own profile page, they'll see an "Edit" button. The handler for this button in the `UserPage` component will redirect to a new route that displays a form with user information fields.

Listing 84 *src/pages/UserPage.js*: Edit user handler

```
  const edit = () => {
    navigate('/edit');
  };
```

The routing in the `App` component needs to be updated to support a new route, which will be handled by a component called `EditUserPage`.

Listing 85 *src/App.js*: Edit user route

```
// add this import at the top
import EditUserPage from './pages/EditUserPage';

export default function App() {
  ... // <-- no changes to logic in this function

  return (
    ...

    // add this route in the private routes section, above the "*'
    <Route path="/edit" element={<EditUserPage />} />

    ...
  );
}
```

The complete implementation of the `EditUserPage` component is shown below.

Listing 86 *src/pages/EditUserPage.js*: Edit user form

```
import { useState, useEffect, useRef } from 'react';
import Form from 'react-bootstrap/Form';
import Button from 'react-bootstrap/Button';
```

```
import { useNavigate } from 'react-router-dom';
import Body from '../components/Body';
import InputField from '../components/InputField';
import { useApi } from '../contexts/ApiProvider';
import { useUser } from '../contexts/UserProvider';
import { useFlash } from '../contexts/FlashProvider';

export default function EditUserPage() {
  const [formErrors, setFormErrors] = useState({});
  const usernameField = useRef();
  const emailField = useRef();
  const aboutMeField = useRef();
  const api = useApi();
  const { user, setUser } = useUser();
  const flash = useFlash();
  const navigate = useNavigate();

  useEffect(() => {
    usernameField.current.value = user.username;
    emailField.current.value = user.email;
    aboutMeField.current.value = user.about_me;
    usernameField.current.focus();
  }, [user]);

  const onSubmit = async (event) => {
    event.preventDefault();
    const response = await api.put('/me', {
      username: usernameField.current.value,
      email: emailField.current.value,
      about_me: aboutMeField.current.value,
    });
    if (response.ok) {
      setFormErrors({});
      setUser(response.body);
      flash('Your profile has been updated.', 'success');
      navigate('/user/' + response.body.username);
    }
    else {
      setFormErrors(response.body.errors.json);
    }
  };

  return (
    <Body sidebar={true}>
      <Form onSubmit={onSubmit}>
```

```
        <InputField
          name="username" label="Username"
          error={formErrors.username} fieldRef={usernameField} />
        <InputField
          name="email" label="Email"
          error={formErrors.email} fieldRef={emailField} />
        <InputField
          name="aboutMe" label="About Me"
          error={formErrors.about_me} fieldRef={aboutMeField} />
        <Button variant="primary" type="submit">Save</Button>
      </Form>
    </Body>
  );
}
```

This component renders a form with three fields for the user to change the username, email address or "about me" information. The fields are all instances of the `InputField` component, and are provisioned with references as seen in previous forms.

Other forms had a side effect function that put the focus on the first field of the form. In this component, the side effect function does that as well, but it also pre-populates the three input fields with their current values, obtained from the `user` object returned by the `useUser()` hook. Assigning values to form fields is done through the DOM, using the reference objects assigned to the input fields.

The form submission handler sends a `PUT` request to */api/me* in Microblog API, with the data obtained from the form fields through the references. If the request succeeds, the `setUser()` function returned by the `useUser()` hook is called to update the user object in the `UserProvider` component, a success message is flashed, and a redirect back to the user page is issued.

As with previous forms, in case of an error, the validation messages from the server are loaded in the `formErrors` state variable, which in turn will make the errors visible below each field.

The JSX code for this form renders the form with the three fields and a submit button, as done in previous forms. Figure 28 shows how the edit user form looks.

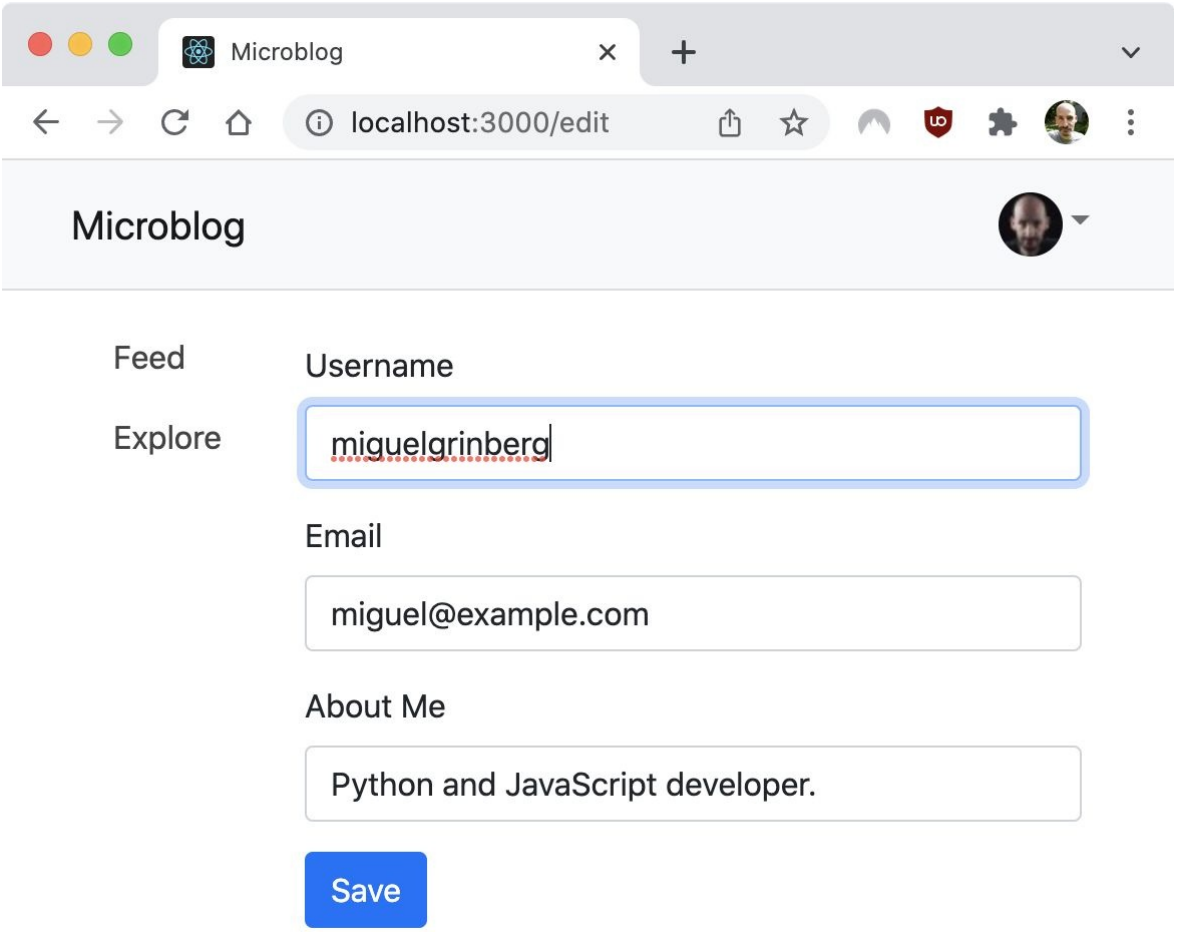


Figure 28 Edit user form

Following and Unfollowing Users

The changes required to follow and unfollow users are surprisingly short. To follow user, a `POST` request to */api/me/following/{user_id}* needs to be sent. To unfollow, a `DELETE` request to the same URL is used. The two handlers for the `UserPage` component are shown below.

Listing 87 src/pages/UserPage.js: Follow and unfollow handlers

```
const follow = async () => {
  const response = await api.post('/me/following/' + user.id);
  if (response.ok) {
    flash(
      <>
```

```
          You are now following <b>{user.username}</b>.
        </>, 'success'
      );
      setIsFollower(true);
    }
  };

  const unfollow = async () => {
    const response = await api.delete('/me/following/' + user.id);
    if (response.ok) {
      flash(
        <>
          You have unfollowed <b>{user.username}</b>.
        </>, 'success'
      );
      setIsFollower(false);
    }
  };
```

The handlers send the appropriate request to the back end. On success, they flash a confirmation message to the user, and then update the `isFollower` state variable, so that the "Follow" button becomes "Unfollow" and vice versa.

The first arguments to the `flash()` function in these two handlers is unusual. Instead of them being plain strings, JSX fragments are used. This makes it possible to use a bold font for the username. Using strings with HTML elements would not work, because as a measure to prevent cross-site scripting (XSS) attacks React escapes all text that is rendered.

With these changes, you can go to the Explore page, click on one or more usernames, and follow them. The blog posts from the users you follow will appear on your feed page along with your own posts.

Changing the Password

An important security feature in all applications is to let users change their passwords. The password is an attribute of the user, so like other attributes, it is changed by sending a `PUT` request to */api/me*. But unlike the other attributes, the password is special in that the user must provide both the old and the new passwords in the body of the request for the change to be accepted by the server, and for that reason it is best to handle it separately.

The change password option can be added to the account dropdown that appears on the right side of the `Header` component. Add this option between the divider and the logout menu option.

Listing 88 src/components/Header.js: Change password menu option

```
<NavDropdown.Item as={NavLink} to="/password">
  Change Password
</NavDropdown.Item>
```

The */password* client route needs to be added in the `App` component. As with other similar routes, it will be mapped to a new component in the *pages* directory called `ChangePasswordPage`.

Listing 89 src/App.js: Change password route

```
// add this import at the top
import ChangePasswordPage from './pages/ChangePasswordPage';

export default function App() {
  ... // <-- no changes to logic in this function

  return (
    ...

    // add this route in the private routes section, above the "*"
    <Route path="/password" element={<ChangePasswordPage />} />

    ...
  );
}
```

Below is the `ChangePasswordPage` component.

Listing 90 src/pages/ChangePasswordPage.js: Change password form

```
import { useState, useEffect, useRef } from 'react';
import Form from 'react-bootstrap/Form';
import Button from 'react-bootstrap/Button';
import { useNavigate } from 'react-router-dom';
import Body from '../components/Body';
```

```
import InputField from '../components/InputField';
import { useApi } from '../contexts/ApiProvider';
import { useFlash } from '../contexts/FlashProvider';

export default function EditUserPage() {
  const [formErrors, setFormErrors] = useState({});
  const oldPasswordField = useRef();
  const passwordField = useRef();
  const password2Field = useRef();
  const navigate = useNavigate();
  const api = useApi();
  const flash = useFlash();

  useEffect(() => {
    oldPasswordField.current.focus();
  }, []);

  const onSubmit = async (event) => {
    event.preventDefault();
    if (passwordField.current.value !== password2Field.current.val
        setFormErrors({password2: "New passwords don't match"});
    }
    else {
      const response = await api.put('/me', {
        old_password: oldPasswordField.current.value,
        password: passwordField.current.value
      });
      if (response.ok) {
        setFormErrors({});
        flash('Your password has been updated.', 'success');
        navigate('/me');
      }
      else {
        setFormErrors(response.body.errors.json);
      }
    }
  };

  return (
    <Body sidebar>
      <h1>Change Your Password</h1>
      <Form onSubmit={onSubmit}>
        <InputField
          name="oldPassword" label="Old Password" type="password"
          error={formErrors.old_password} fieldRef={oldPasswordFie
```

```
        <InputField
          name="password" label="New Password" type="password"
          error={formErrors.password} fieldRef={passwordField} />
        <InputField
          name="password2" label="New Password Again" type="passwo
          error={formErrors.password2} fieldRef={password2Field} /
        <Button variant="primary" type="submit">Change Password</B
      </Form>
    </Body>
  );
}
```

This form uses similar logic to previous forms. It combines client-side validation for the two password fields, with server-side validation, which ensures the old password is correct, and the new password is not empty.

After a successful response from the `PUT` request to */api/me*, the user is redirected to user profile page, with a flashed message that indicates the success of the change.

Figure 29 shows the change password page.

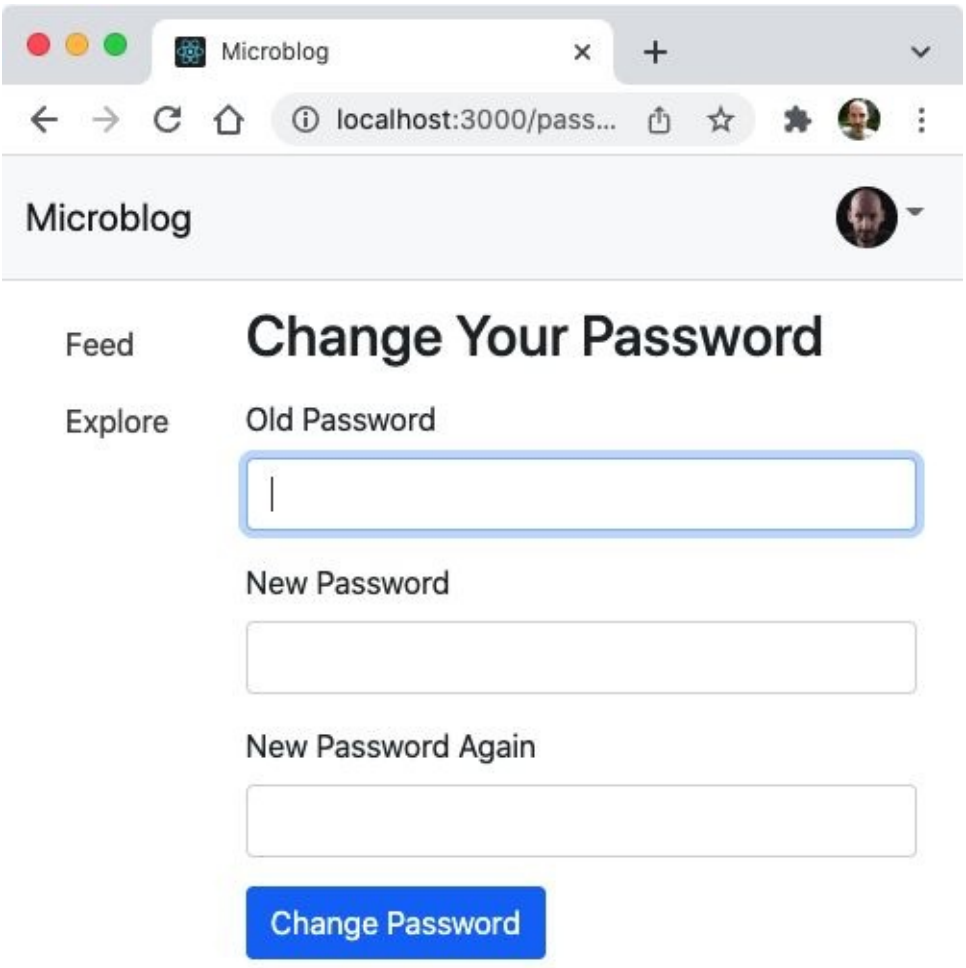


Figure 29 Change password page

Password Resets

The last feature of the application that will be added in this chapter is the option for users to request a password reset when they forget their account password. This feature is implemented in two parts:

- First, the user must click a "Forgot Password" link in the login page to access the password reset request page. On this page, the user must enter their email address. If the email address is valid, the server will send an email with a password reset link.
- The second part of the reset process occurs when the user clicks the password reset link received by email. The link contains a token that needs to be submitted to the server for verification, along with a new password. If

the token is valid, then the password is updated.

The implementation of this feature requires two new routes, which will be */reset-request* and */reset*. These are public routes that need to be wrapped with the `PublicRoute` component to ensure that a logged-in user does not have access to them.

Listing 91 *src/App.js*: Password reset routing updates

```
// add these imports at the top
import ResetRequestPage from './pages/ResetRequestPage';
import ResetPage from './pages/ResetPage';

export default function App() {
  ... // <-- no changes to logic in this function

  return (
    ...

    // add these routes in the public routes section
    <Route path="/reset-request" element={
      <PublicRoute><ResetRequestPage /></PublicRoute>
    } />
    <Route path="/reset" element={
      <PublicRoute><ResetPage /></PublicRoute>
    } />

    ...
  );
}
```

As before, the application will temporarily break because the modules imported above do not exist yet.

Requesting a Password Reset

The login page needs a link to a new page in which the user can request a password reset.

Listing 92 *src/pages/LoginPage.js*: Reset password link

```
  // add this above the registration link
  <p>Forgot your password? You can <Link to="/reset-request">reset i
```

The complete implementation of the `ResetRequestPage` component is shown below.

```
caption: "*src/pages/ResetRequestPage.js*: Reset request form"
name: 09resetrequestpagejs
---
import { useState, useEffect, useRef } from 'react';
import Form from 'react-bootstrap/Form';
import Button from 'react-bootstrap/Button';
import Body from '../components/Body';
import InputField from '../components/InputField';
import { useApi } from '../contexts/ApiProvider';
import { useFlash } from '../contexts/FlashProvider';

export default function ResetRequestPage() {
  const [formErrors, setFormErrors] = useState({});
  const emailField = useRef();
  const api = useApi();
  const flash = useFlash();

  useEffect(() => {
    emailField.current.focus();
  }, []);

  const onSubmit = async (event) => {
    event.preventDefault();
    const response = await api.post('/tokens/reset', {
      email: emailField.current.value,
    });
    if (!response.ok) {
      setFormErrors(response.body.errors.json);
    }
    else {
      emailField.current.value = '';
      setFormErrors({});
      flash(
        'You will receive an email with instructions ' +
        'to reset your password.', 'info'
      );
    }
  };
```

```
  return (
    <Body>
      <h1>Reset Your Password</h1>
      <Form onSubmit={onSubmit}>
        <InputField
          name="email" label="Email Address"
          error={formErrors.email} fieldRef={emailField} />
        <Button variant="primary" type="submit">Reset Password</Butt
      </Form>
    </Body>
  );
}
```

The reset request form has a single field, in which the user must enter the email address for the account to reset. The email is submitted in a `POST` request to the */api/tokens/reset* URL, and as a result of this request the user should receive an email with a reset link.

Note that a valid email server must be configured in Microblog API for reset emails to be sent out.

Resetting the Password

The email reset links that Microblog API sends out to users have a query string parameter called `token`. The value of this token, along with the new chosen password for the account must be sent on a `PUT` request to the */api/tokens/reset* endpoint. If the server determines that the reset token is valid, then it updates the password on the account and returns a response with a 200 status code.

Implementing this part of the process represents a departure from anything else that was done in this application, because in this case the user will be starting a new instance of the application by clicking on the link that is in the email. The URL in the email point to the URL where the React application is running, appended with the */reset* path so that the correct client route is invoked.

Microblog API has a configuration variable called `PASSWORD_RESET_URL` that defines what is the URL that needs to be sent in reset emails. By default, the URL is *http://localhost:3000/reset*, which will work for a development version of the React front end running on your computer.

The `ResetPage` component shown below extracts the token from the query string, presents a form that asks the user for the account password, and finally submits both to the server.

Listing 93 src/pages/ResetPage.js: Reset password

```
import { useState, useEffect, useRef } from 'react';
import Form from 'react-bootstrap/Form';
import Button from 'react-bootstrap/Button';
import { useNavigate, useLocation } from 'react-router-dom';
import Body from '../components/Body';
import InputField from '../components/InputField';
import { useApi } from '../contexts/ApiProvider';
import { useFlash } from '../contexts/FlashProvider';

export default function EditUserPage() {
  const [formErrors, setFormErrors] = useState({});
  const passwordField = useRef();
  const password2Field = useRef();
  const navigate = useNavigate();
  const { search } = useLocation();
  const api = useApi();
  const flash = useFlash();
  const token = new URLSearchParams(search).get('token');

  useEffect(() => {
    if (!token) {
      navigate('/');
    }
    else {
      passwordField.current.focus();
    }
  }, [token, navigate]);

  const onSubmit = async (event) => {
    event.preventDefault();
    if (passwordField.current.value !== password2Field.current.val
        setFormErrors({password2: "New passwords don't match"});
    }
    else {
      const response = await api.put('/tokens/reset', {
        token,
        new_password: passwordField.current.value
```

```
      });
      if (response.ok) {
        setFormErrors({});
        flash('Your password has been reset.', 'success');
        navigate('/login');
      }
      else {
        if (response.body.errors.json.new_password) {
          setFormErrors(response.body.errors.json);
        }
        else {
          flash('Password could not be reset. Please try again.',
          navigate('/reset-request');
        }
      }
    }
  };

  return (
    <Body>
      <h1>Reset Your Password</h1>
      <Form onSubmit={onSubmit}>
        <InputField
          name="password" label="New Password" type="password"
          error={formErrors.password} fieldRef={passwordField} />
        <InputField
          name="password2" label="New Password Again" type="passwo
          error={formErrors.password2} fieldRef={password2Field} /
        <Button variant="primary" type="submit">Reset Password</Bu
      </Form>
    </Body>
  );
}
```

The `token` constant is obtained from the query string included in the URL, which can be obtained from the `location` object provided by React-Router as `search`. The `URLSearchParams` class provided by the browser is used to decode the query string into an object.

The form implemented in this component has two fields for the user to enter the new password twice. Client-side validation ensures that these two fields have the same value.

The side effect function has a small difference. Invoking this route without a `token` query parameter makes no sense, so as a security check the function redirects to the home page if it finds that there is no token.

The function references `token` and `navigate()`, which are defined outside the side effect function. For that reason the React build generated warnings and suggested that these should be added to the side effect function as dependencies.

The submission request includes the token and the password in the body of the request, as required by Microblog API. If the request is successful, the user is redirected to the */login* route with a flashed message. If the request fails there are two possibilities. When the reason for the failure is a validation error in the password field, then the `formErrors` state is updated to show the error to the user. For any error responses that do not include validation issues on the password field, a redirect is used to send the user back to the password reset request page to retry. This could happen if the user attempts to use a reset link after the token has expired.

To test the password reset flow, first make sure you are not logged in. From the login page, click on the reset link and enter your email. Click on the link that you receive a few seconds later and enter a new password for your account. If you do not receive an email from Microblog API, then your email server settings are probably incorrect, so go back to Chapter 5 to review them.

Chapter Summary

- A convenient pattern to create reusable child components is for parents to create callback functions with customized behaviors and pass them to the child component as props.
- If you don't like some aspects of how a third-party component works, you may be able to create a wrapper component that overrides and improves the original component.
- You can pre-populate form fields in a side effect function using DOM APIs and field references.
- To prevent XSS attacks, React escapes all rendered variables. If you need to store HTML in a variable intended for rendering, use a JSX expression instead of a plain string.
- Application routes are proper links that can be bookmarked or even shared in emails or other communication channels.

Memoization

An important part of React that you haven't seen yet is memoization, a technique that helps prevent unnecessary renders and improve the overall performance of the application. In this chapter you are going to learn how the React rendering algorithm works, and how you can make it work efficiently for your application.

The React Rendering Algorithm

Before looking at how to optimize rendering in React applications, let's review what happens while the application renders.

During the initial load, React renders the top component of the application, often called `App`. It does this by invoking its render function. Because this is the initial render, React also renders all the children components referenced in the JSX returned by `App`.

However, the component render functions represent half of the picture. In React, renders are carried out in two phases. First, output of the render functions is applied to the *virtual DOM*, which is an internal copy of the actual DOM from the browser. Once the components are rendered on the virtual DOM, the second phase invokes an efficient algorithm to merge the virtual DOM to the real DOM, and this is what makes the components visible on the page. On the initial render, the entire contents of the virtual DOM have to be copied over to the real DOM, but once the application is running and the real DOM is populated, the merge process only copies the elements that are different, making updates more efficient.

Once the two phases of the first render complete and the application is visible on the page, React runs the side effect functions that were started by many of the components. These functions do some work, and will eventually update some of the state variables that are held by components.

Every time a state variable changes, React determines which of the components that are on the page have a dependency on this variable. These affected components are re-rendered, to give them a chance to refresh their state. These

renders will also be made in two phases, first the render results are applied to the virtual DOM, and then the virtual DOM is efficiently merged to the real DOM.

This cycle can sometimes continue, because the renders that were started after state variables changed may launch new side effect functions, which will in turn change more state variables that will trigger even more re-renders. But the application will eventually settle, with all the components and state variables updated, and all the side effect functions finished. When the application reaches this state, React has no more work to do.

But at this point the user may click a button, or a link. Event handlers attached to these UI elements will be invoked by the browser, and these will very likely end up changing some state variables, starting another chain of renders.

As you can see, the rendering algorithm is very optimized to only update the parts of the component tree that depend on state changes. While the optimizations React implements are very useful, they don't always prevent performance problems.

Unnecessary Renders

Sometimes a component renders even though none of its inputs or state variables have changed. When that happens, the render function still runs, and its results are applied to the virtual DOM. Presumably the result of the render is going to be identical to the previous render, since no inputs have changed. When the render reaches the second phase, React will find nothing to merge from virtual to real DOM.

It is good that React is able to identify these unnecessary renders and avoid updating the real DOM when they happen. But it is also bad that React sometimes does not realize that a component that is in the queue for rendering hasn't changed, because if it did, it could avoid rendering it altogether.

Unfortunately unnecessary renders are more common that most developers think. When a component renders, it returns a new JSX subtree. Any child components that are included in the returned JSX are forced to re-render even if they don't change, simply because they generated anew during the parent's render.

As a general rule, you can say that unless measures are taken to prevent it, each time a component renders, any child components included in the JSX returned have to render too.

In many cases the cost of these unnecessary renders is negligible, but this isn't always the case. Consider the `Posts` component, which has 25 instances of the `Post` component. If the user clicks the "More" button to obtain the 25 posts, all 50 posts will re-render.

Memoizing Components

Given the same inputs, the vast majority of components will return the same output. So you could make the argument that it is unnecessary to re-render a component when neither its props nor its state variables have changed.

This is the idea behind the `memo()` function provided by React. The function wraps any component to give it additional logic that can decide to skip the render when the inputs did not change.

This can be used to optimize the rendering of the `Posts` component when pagination is used to add more children to it. To prevent these unnecessary renders, the `Post` component can be memoized by adding the `memo()` function to it as a wrapper.

Listing 94 src/components/Post.js: Memoize the component

```
import { memo } from 'react';
... // <-- no changes to existing imports

export default memo(function Post({ post }) {
  ... // <-- no changes to function body
});
```

This change transparently replaces the original `Post` component with a version of it that is optimized to skip the render when it is invoked with the same inputs as the previous time.

Should all components be memoized? This is a difficult question to answer. It is easy to visualize the performance improvements for components such as `Post`,

which are often re-rendered by React, just because the parent renders.

It is hard to say that it is beneficial to memoize components that do not render as often, or that generally render with different inputs, because the logic added by `memo()` also has a cost. If you are interested in finding out if applying `memo()` to a component makes your application render faster, a specialized tool such as the profiler included in the React Developer Tools plugin for Chrome or Firefox should be used to take accurate measurements.

Render Loops

Another common problem related to the React rendering algorithm happens when endless render cycles are inadvertently introduced across many components. These occur when application re-renders parts of its tree indefinitely due to cyclical dependencies. An endless render cycle will cause high CPU usage and poor application performance, so it should be avoided. The application in its current state does not have any render loops, but it is actually easy to inadvertently introduce one.

A feature that would be nice to have in Microblog is a global error handler that can show an error alert to the user when the server is unresponsive or offline. To implement this feature, the `MicroblogApiClient` class constructor can be expanded to accept an `onError` argument that the caller can use to pass a custom error handler function.

Listing 95 src/MicroblogApiClient.js: custom error handler

```
export default class MicroblogApiClient {
  constructor(onError) {
    this.onError = onError;
    this.base_url =  BASE_API_URL + '/api';
  }

  async request(options) {
    let response = await this.requestInternal(options);
    if (response.status === 401 && options.url !== '/tokens') {
      ... // <-- no changes to retry logic
    }
    if (response.status >= 500 && this.onError) {
      this.onError(response);
```

```
    }
    return response;
  }

  ... // <-- no changes to the rest of the class
}
```

If the caller passes an error handler, then it is called when the response to a request has a 500 or higher status code, which according to the HTTP specification are associated with server failures.

With the error handling support incorporated into the API client, the `ApiProvider` component can create a handler that flashes an error message, so that the user knows there is a problem. The updates to the `ApiProvider` component are shown below.

Listing 96 src/contexts/ApiProvider.js: Error handling

```
import { createContext, useContext } from 'react';
import MicroblogApiClient from '../MicroblogApiClient';
import { useFlash } from './FlashProvider';

export const ApiContext = createContext();

export default function ApiProvider({ children }) {
  const flash = useFlash();

  const onError = () => {
    flash('An unexpected error has occurred. Please try again.', '
  };

  const api = new MicroblogApiClient(onError);

  // <-- no changes to the returned JSX
}

... // <-- no changes to the hook function
```

While this implementation appears to be correct at first sight, the reality is that it has introduced a render loop. It is really difficult to spot it, but see if you can identify where the loop is.

The following list is a step by step account of what would happen when the server returns a 500 or higher response after a user action that required an API call:

1. The `MicroblogApiClient` instance owned by the `ApiProvider` component receives an error response from the back end, and invokes the `onError()` function, also owned by `ApiProvider`.
2. The `onError()` function in `ApiProvider` invokes the `flash()` function to display an error alert.
3. The `flash()` function, owned by the `FlashProvider` component, calls the `setFlashMessage()` function to update the `flashMessage` state.
4. Because the `flashMessage` state variable was changed, React decides that the `FlashProvider` component, which owns it, must be re-rendered.
5. The `FlashProvider` render function creates new `flash()` and `hideFlash()` functions (because these are defined locally inside the render function) and updates the value of the `FlashContext` with them.
6. After these last changes, the `ApiProvider` component depends on the `flash()` function. Because of this dependency, React re-renders `ApiProvider`.
7. The `ApiProvider` render function creates a new instance of the `MicroblogApiClient` class. A new `onError()` handler function is created as well, and configured into it.
8. The `UserProvider` component uses the `useApi()` hook to get the `MicroblogApiClient` instance. Because this instance changed, React decides `UserProvider` needs to re-render.
9. The `UserProvider` has a side effect function that depends on the `api` instance. Since this instance changed, React re-runs the side effect.
10. The side effect function in `UserProvider` attempts to load the currently logged-in user by sending a request to the */me* endpoint in the server. A malfunction in the API is what started this render cycle, so the most likely outcome for another call to the API is to also end in error, and this would make this entire render process repeat from the beginning. The cycle will continue until the API stops returning errors.

How can you prevent loops such as this one? You can study the source code for all the components involved, and you may still not see a straightforward way to break the loop. These components just have a circular chain of relationships, so the first solution you may consider is to eliminate some dependencies, at the cost of removing features. For example, if the `onError()` handler didn't use the

`flash()` function, then this issue wouldn't case a render loop.

Luckily there is another way to look at the problem, without having to compromise on the features. If you review the sequence of actions carefully, you may see that step 5 is when things started to get out of control.

As a result of the first API error, the `onError()` handler calls the `flash()` function, so the `FlashProvider` needs to re-render to publish the alert message to the `FlashContext`.

However, an unintended consequence of this re-render is that the `flash()` and `hideFlash()` functions stored in the context are replaced with brand-new versions. These new functions are identical to the original ones, but they are new copies, because the functions are declared in the local scope of the component's render function. The new `flash()` function is what makes React decide that `ApiProvider` needs to re-render in step 6. In other words, if there was a way to prevent the `flash()` function from changing, then `ApiProvider` and any other components that use it would not become outdated and would not need to re-render.

Similarly, when `ApiProvider` re-renders, it creates a new `api` object, which is added to the `ApiContext` in step 7. This change is what makes React decide that `UserProvider` needs to re-render in step 8, to prevent it from having a stale `api` value. But the `api` object is an instance of the `MicroblogApiClient` class which is not affected by any of the other changes. If the `api` object could be kept the same when `ApiProvider` re-renders, then other components that depend on it would not need to re-render just so that they get the new object, which is 100% equivalent to the old one.

The solution that often allows these circular dependencies to not loop forever is to ensure that functions and objects created in the local context of a render function are not updated unnecessarily. In other words: if a function or object created in a component's render function is not directly affected by the change that triggered the render, then it should not change. This can also be handled by memoization, but at a more granular scale than the `memo()` function discussed above.

Memoizing Functions and Objects

As discussed above, each time a component renders, new instances of any functions and objects allocated inside its render function are created. Sometimes it is necessary to update these items, but many times using new instances is unnecessary. React provides the `useCallback()` and `useMemo()` hooks to memoize functions and other values.

To memoize a function, it needs to be wrapped with the `useCallback()` hook function. This function takes the function to memoize as the first argument, and a dependency list similar to the one used in the `useEffect()` callback as second argument. As long as the dependencies declared in the second argument don't change, the function will not change.

You can see how to memoize `flash()` and `hideFlash()` below.

Listing 97 src/contexts/FlashProvider.js: Memoize `flash()` and `hideFlash()`

```
import { createContext, useContext, useState, useCallback } from '

export const FlashContext = createContext();
let flashTimer;

export default function FlashProvider({ children }) {
  const [flashMessage, setFlashMessage] = useState({});
  const [visible, setVisible] = useState(false);

  const hideFlash = useCallback(() => {
    ... // <-- no changes in the function body
  }, []);

  const flash = useCallback((message, type, duration = 10) => {
    ... // <-- no changes in the function body
  }, [hideFlash]);

  // <-- no changes to the returned JSX
}

... // <-- no changes to the hook function
```

The `hideFlash()` was moved above `flash()`, because `flash()` calls it, which means that it needs to be listed as a dependency. Remember that the React development build detects these missing dependencies and warns you about any

missing ones, so always check the warning in the terminal where you run the development web server.

The next listing shows the memoizing changes for `ApiProvider`.

Listing 98 src/contexts/ApiProvider.js: Memoize `onError` and `api`

```
import { createContext, useContext, useCallback, useMemo } from 'r
import MicroblogApiClient from '../MicroblogApiClient';
import { useFlash } from './FlashProvider';

export const ApiContext = createContext();

export default function ApiProvider({ children }) {
  const flash = useFlash();

  const onError = useCallback(() => {
    flash('An unexpected error has occurred. Please try again late
  }, [flash]);

  const api = useMemo(() => new MicroblogApiClient(onError), [onEr

  // <-- no changes to the returned JSX
}

... // <-- no changes to the hook function
```

The `onError()` function is memoized just like before, with the dependency list passed in the second argument to `useCallback()` set to the `flash()` function, which is used inside the body of the function.

The `api` object cannot be memoized with `useCallback()`, which only works with functions. The `useMemo()` hook is a more generic version of `useCallback()` that can be used to memoize values of any type. The first argument to this hook is a function that returns the value to memoize. The second argument is the dependency list, same as in `useCallback()`. For `api`, the only dependency is the `onError()` function, defined right above.

The `UserProvider` component was also part of the render loop, but now that `api` is memoized, this component will only re-render when its `user` state variable changes, which only happens when the user logs in or out. Even though the

render loop is addressed with the above changes, it makes sense to also memoize `UserProvider`, since it has two functions shared in its context that should not change unless strictly necessary to avoid children components that use them to re-render unnecessarily.

Listing 99 src/contexts/UserProvider.js: Memoize `login()` and `logout()`

```
import { createContext, useContext, useState, useEffect, useCallba
import { useApi } from './ApiProvider';

export const UserContext = createContext();

export default function UserProvider({ children }) {
  const [user, setUser] = useState();
  const api = useApi();

  ... // <-- no changes to side effect function

  const login = useCallback(async (username, password) => {
    ... // <-- no changes in the function body
  }, [api]);

  const logout = useCallback(async () => {
    ... // <-- no changes in the function body
  }, [api]);

  // <-- no changes to the returned JSX
}

... // <-- no changes to the hook function
```

If you are running the Microblog API service locally and would like to try the global error handler, you can stop the service by pressing Ctrl-C in the terminal session it is running, and then the next operation you trigger in the application that requires access to the server is going to show the error alert. Once you restart the API you will be able to continue using the application.

Chapter Summary

- Optimize components that render many times with the same inputs by memoizing them with the `memo()` function from React.
- Memoizing functions and objects defined in high-level components such as those sharing contexts with `useCallback()` and `useMemo()`, so that children components that depend on them do not render unnecessarily.

Automated Testing

Up to now, all the testing done on the application you've built was manual. Manual testing as you develop your application is useful, but as the application grows the required testing effort grows as well, until it becomes so time-consuming that the only way to keep up is to test less, or to find a way to automate some testing work.

In this chapter you are going to learn some techniques to create automated tests for your React application.

The Purpose of Automated Testing

When asked about the purpose of automated testing, most people say that it is the same as testing manually, which is to find bugs. This is, in general terms, correct, but what many don't know is that automated testing is far superior to manual testing at finding a specific type of bugs called *regressions*.

Regressions are bugs that occur in code that previously worked correctly. They are usually caused by additions or modifications that affect existing code in ways that the developer fails to recognize. The larger the codebase, the easier it is for regressions to be introduced, as the relationships and dependencies between different parts of the application become more complex difficult to track.

In short, when you develop a new feature of your application, you are going to test it manually and ensure that it works correctly. This is the best time for you to spend a little more time creating automated tests that can ensure that this feature continues to work as designed in the future, as more changes and features are added.

Testing React Applications with Jest

Applications created with Create React App come with testing support built it, using the Jest testing framework.

A common naming convention for test files is to use a *.test.js* suffix. For

example, tests that apply to code in *src/App.js* are written in a *src/App.test.js* file. In fact, the project already has a *src/App.test.js* file with an example test that was added by Create React App.

The example test in *src/App.test.js* was designed to work with the starter application. With all the changes that went into the application this test is now failing.

To start the test suite, you can run the following command from a terminal:

```
npm test
```

As with `npm start`, the test command runs in the background and watches your code changes. As files change, related tests execute.

When the test suite is started, Jest is going to try to determine what were the most recent changes that were made to the application, and then run the tests for the affected files. Jest may decide that *App.js* needs to be tested, and then you will see a long description of this test's failure, followed by a summary such as the following:

```
Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
Snapshots:   0 total
Time:        1.996 s, estimated 2 s
Ran all test suites.

Watch Usage: Press w to show more.
```

Jest may also decide that none of the changes you made recently to the project affect *App.js*, and in that case it will not run any tests and just start to watch for changes with the following message:

```
No tests found related to files changed since last commit.
Press `a` to run all tests, or run Jest with `--watchAll`.

Watch Usage
 â€º Press a to run all tests.
 â€º Press f to run only failed tests.
 â€º Press q to quit watch mode.
 â€º Press p to filter by a filename regex pattern.
 â€º Press t to filter by a test name regex pattern.
```

```
 â€º Press Enter to trigger a test run.
```

If you get this output, you press "a" to instruct Jest to run the test suite. I recommend that you do this, so that you become familiar with how Jest shows test failures.

Renders, Queries and Assertions

The first task is to make adjustments to the test in *src/App.test.js* so that it is correct for the application in its current state. Open this file to have a look at the test code:

```
import { render, screen } from '@testing-library/react';
import App from './App';

test('renders learn react link', () => {
  render(<App />);
  const linkElement = screen.getByText(/learn react/i);
  expect(linkElement).toBeInTheDocument();
});
```

To write a test, you called the `test()` function, which is globally installed by the Jest framework. The first argument to this function is a written description of the purpose of the test, and the second argument is a function that contains the test logic.

There are several libraries and tools that make it possible to write tests. The test above shows three of them:

- The `render()` function from the React Testing Library allows the test to render a component. Rendering in this case does not happen in the browser, but in an emulated environment that is similar to it.
- The `screen` object, also exposed by the same library, allows the test to query the contents of the page that was rendered by `render()`.
- The `expect()` function, another global from Jest, allows the test to create *assertions*, which are the checks that determine if the test passed of failed.

In the test, the `render()` function is used to render the `App` component. Then `screen.getByText()` is used to find an element in the render page that has the given text. Finally, `expect()` is used to ensure that this element exists in the

page.

Let's rewrite this test so that it checks that the "Microblog" heading in the top-left corner of the page is rendered. Here is the updated version of *src/App.test.js* with this test:

Listing 100 *src/App.test.js*: First application test

```
import { render, screen } from '@testing-library/react';
import App from './App';

test('renders brand element', () => {
  render(<App />);

  const element = screen.getByText(/Microblog/);

  expect(element).toBeInTheDocument();
  expect(element).toHaveClass('navbar-brand');
});
```

If you have the `npm test` command running, as soon as you save the new version of *src/App.test.js* the test will run again, and this time it will pass.

The new version of the test still renders `App`, but then it runs a query to locate an element in the page with the text "Microblog", which should return the title element in the heading.

In this test there are two assertions. First the test ensures that the element was found in the page. As an additional check, it makes sure that the element has the class `navbar-brand`, which comes from Bootstrap.

The majority of tests in a React application are going to have this same structure. First, a component will be rendered, then one or more queries will retrieve elements of interest from the rendered contents, and finally, one or more assertions will ensure that these elements have the expected attributes.

Using Queries

The React Testing Library provides the `screen` object to perform queries on content previously rendered with the `render()` function.

When looking for an element, there are three groups of queries:

- `getBy...()` returns the matching element, or raises an error if there are no matching elements, or more than one matching element.
- `queryBy...()` returns the matching element, returns `null` if there are no matching elements, or raises an error if there are more than one matching elements.
- `findBy...()` waits for up to one second (by default) for a matching element to appear in the rendered page. An error is raised if no matching element or multiple elements appear during the waiting period. This method is asynchronous and returns a promise.

Sometimes it is useful to look for multiple matching elements. There is a similar set of query functions that support this:

- `getAllBy...()` returns an array with the matching elements, or raises an error if there are no matches.
- `queryAllBy...()` returns an array with the matching elements, or an empty array if there are no matches.
- `FindAllBy...()` waits for one or more matching elements to appear and returns them as an array. An error is raised if no matching elements appear. This method is asynchronous and returns a promise.

For each of these six query options, you can choose from a selection of query options. Above you've seen how to locate an element by its text with `getByText()`. Other options are `getByTitle()`, `getByAltText()`, `getByRole()` and more that you can see in the documentation.

Using Assertions

Once you have an element of interest, you have to create assertions that ensure that this element has the expected structure. This is done with the expect() function from Jest, which is combined with a matcher function. Below is a straightforward assertion that checks that a variable `result` is `null`:

```
expect(result).toBeNull();
```

The structure of an assertion is always the same. The value or expression that is the target of the check is passed as an argument to the `expect()` function. Then a

matcher method such as `toBeNull()` above is called on the result.

Jest provides an extensive list of matcher methods, but none of this are specific to React or web development. The jest-dom package, which is also installed by Create React App, extends Jest with a range of custom matcher methods that apply to DOM elements, such as the `toBeInTheDocument()` and `toHaveClass()` methods used in the test above.

Testing Individual Components

The test above rendered the complete application, starting from the `App` component. In many cases it is preferable to render a single component or a small subset.

Below you can see a test for the `Post` component.

Listing 101 *src/components/Post.test.js*: Test `Post` component

```
import { render, screen } from '@testing-library/react';
import { BrowserRouter } from 'react-router-dom';
import Post from './Post';

test('it renders all the components of the post', () => {
  const post = {
    text: 'hello',
    author: {username: 'susan', avatar_url: 'https://example.com/a
    timestamp: '2022-01-01T00:00:00.000Z',
  };

  render(
    <BrowserRouter>
      <Post post={post} />
    </BrowserRouter>
  );

  const message = screen.getByText('hello');
  const authorLink = screen.getByText('susan');
  const avatar = screen.getByAltText('susan');
  const timestamp = screen.getByText(/.* ago$/);

  expect(message).toBeInTheDocument();
```

```
  expect(authorLink).toBeInTheDocument();
  expect(authorLink).toHaveAttribute('href', '/user/susan');
  expect(avatar).toBeInTheDocument();
  expect(avatar).toHaveAttribute('src', 'https://example.com/avata
  expect(timestamp).toBeInTheDocument();
  expect(timestamp).toHaveAttribute(
    'title', 'Sat Jan 01 2022 00:00:00 GMT+0000 (Greenwich Mean Ti
});
```

This test starts by create a fake blog post, with a similar structure as the post that are returned by Microblog API.

In the render section, an instance of the `Post` component receiving the fake blog post is rendered. Because the `Post` component renders a link that navigates to the author's page, it is necessary to have the support of React Router, so instead of rendering the `Post` component alone, it is wrapped in a `BrowserRouter` component.

When rendering isolated components, it is often necessary to create a minimal environment that is similar to that of the real application. In this case, the component needed routing support, so a router component was added. In other cases a component might need to have access to a specific context, to the corresponding provider component must be included in the render. A good approach to render the smallest possible tree is to start by rendering the target component and look at the errors the render produces to determine what wrapper components are needed.

In the query section of the test, several elements of the rendered post are retrieved. Sometimes it may not be clear what is the best method to query for an element. A good approach is to look at the HTML code that the component renders in the real application from your browser's developer console to find if there is a particular text, alt text or maybe title that can be used to query for the element. If you can't find anything that you can use, you can add a data-testid attribute to query for in the test.

The assertions of this test ensure that the four elements queried appear in the document. For the author link, the `href` attribute is also checked to be correct. Likewise, for the avatar image the `src` attribute is checked. The timestamps rendered by the `TimeAgo` component have a `title` attributes that can be included in assertions. When trying to decide what assertions to write for a component, it

is also a good idea to look at the HTML of the rendered component in your browser's developer console.

Using Fake Timers

Are you ready for something more advanced? In the next test, a small `Test` component is created, with the only purpose of flashing a message. The test then verifies that the alert component appears on the rendered page.

Listing 102 src/contexts/FlashProvider.test.js: Test message flashing

```
import { render, screen } from '@testing-library/react';
import { useEffect } from 'react';
import FlashProvider from './FlashProvider';
import { useFlash } from './FlashProvider';
import FlashMessage from '../components/FlashMessage';

beforeEach(() => {
  jest.useFakeTimers();
});

afterEach(() => {
  jest.useRealTimers()
});

test('flashes a message', async () => {
  const Test = () => {
    const flash = useFlash();
    useEffect(() => {
      flash('foo', 'danger');
    }, []);
    return null;
  };

  render(
    <FlashProvider>
      <FlashMessage />
      <Test />
    </FlashProvider>
  );

  const alert = screen.getByRole('alert');
```

```
  expect(alert).toHaveTextContent('foo');
  expect(alert).toHaveClass('alert-danger');
});
```

There are a number of new concepts in this test. First, since the flashing mechanisms in the application use timers, it is a good idea to use mocked timers that can run time faster. The `beforeEach()` and `afterEach()` functions from Jest take functions that are executed before and after each test in the file. The before function calls `jest.useFakeTimers()` to switch to simulated timers, and the after function resets the default timers from JavaScript.

The test begins by creating the `Test` component, which launches a side effect function and then returns `null` to render itself as an empty component. The side effect function calls the `flash()` function, returned by the `useFlash()` hook.

The `render()` call in this test creates a small application with the `FlashProvider` component at the root. This is necessary so that the `FlashContext` is available. This component has two children: the `FlashMessage` component, which renders the alerts, and the `Test` component create specifically for this test.

An important implementation detail of the `render()` function is that it not only renders the component tree, but also waits for side effects functions to run and update the application state until the application settles and there is nothing else to update. When `render()` returns, the alert should already be visible.

The `Alert` component from React-Bootstrap can be obtained by its role. The assertions ensure that the text and the alert class are correct.

Testing for Follow-Up State Changes

The test above ensures that alerts are displayed, but the flash message system in this application has some complexity that the test isn't reaching. In particular, alerts are supposed to close on their own after 10 seconds. How can that be tested, ideally without having to wait those 10 seconds?

The complication with testing the automatic closing of the alert is that the alert isn't ever removed or hidden. The `FlashMessage` component wraps the alert with a `Collapse` component from React-Bootstrap. This component uses CSS transitions to shrink the alert until it reaches a height of 0 pixels instead of hiding

it.

There are several options to work around this problem, but most would require the test to have knowledge of the inner workings of the `Collapse` component, which is not ideal because the test could break if React-Bootstrap is upgraded to a new version. A surprisingly simple solution is to add a custom data attribute to the `Alert` component indicating the visible state of the alert. This adds a tiny amount of overhead, but gives the test something to assert on.

In the listing below, the alert rendered by the `FlashMessage` component includes a `data-visible` attribute that exposes the `visible` state variable from the `FlashContext`.

Listing 103 src/components/FlashMessage.js: Add testing helper

```
... // <-- no changes to imports

export default function FlashMessage() {
  const { flashMessage, visible, hideFlash } = useContext(FlashCor

  return (
    <Collapse in={visible}>
      <div>
        <Alert variant={flashMessage.type || 'info'} dismissible
          onClose={hideFlash} data-visible={visible}>
          {flashMessage.message}
        </Alert>
      </div>
    </Collapse>
  );
}
```

With the `data-visible` element in place, the flash message test can be expanded to check the message visibility.

Listing 104 src/contexts/FlashProvider.test.js: Test alert visibility

```
import { render, screen, act } from '@testing-library/react';
... // <-- no other changes to imports

... // <-- no changes to beforeEach() and afterEach()
```

```
test('flashes a message', () => {
  ... // <-- no changes to Test component or render() call

  const alert = screen.getByRole('alert');

  expect(alert).toHaveTextContent('foo');
  expect(alert).toHaveClass('alert-danger');
  expect(alert).toHaveAttribute('data-visible', 'true');

  act(() => jest.runAllTimers());
  expect(alert).toHaveAttribute('data-visible', 'false');
});
```

The `data-visible` attribute is checked to be `true` when the alert is rendered. After that, the test needs to wait 10 seconds for the timer to go off and collapse the alert, but having the test really wait that long is impractical. When working with Jest's fake timers, the `jest.runAllTimers()` function advances the time until the next timer, without having to wait for the actual time to pass.

As a result of the timer firing, some state variables in React will change, and that will require some re-renders, which in turn might launch new side effect functions that might require even more renders. The `render()` function is designed to wait for this asynchronous activity until all state variables, side effects and renders settle, but calling `jest.runAllTimers()` on its own would not provide the same kind of safety wait.

The React Testing Library provides the `act()` function to perform this type of waiting. Instead of calling `jest.runAllTimers()` directly, `act()` is called with a function that performs this action. The `act()` function will call the function passed as an argument, and then wait for the React application to settle down.

Once the timer has been advanced, a final assertion ensures that the `data-visible` attribute is now `false`, which confirms that the alert has been collapsed.

Mocking API Calls

Your tests should run in complete isolation, without requiring the availability of external services such as databases or API back ends. When a component that makes calls to a back end needs to be tested, any API calls that it makes should

be *mocked*.

You have seen an example of mocking in the previous section, where a JavaScript timer was replaced with a fake version that runs much faster. In addition to providing fake timers, Jest provides an extensive set of functions for mocking functions or entire modules.

Ready to learn how to mock? The authentication subsystem is a core piece of the application that can benefit from automated testing. This is an excellent use case to learn how to work with mocks.

Since the authentication logic is mostly located in the `UserProvider` component, the tests will be in *src/contexts/UserProvider.test.js*. Below you can see the includes that will be needed, and a `beforeEach()` and `afterEach()` pair of functions that initialize and restore the test environment.

Listing 105 src/contexts/UserProvider.test.js: Create a mock for `fetch()`

```
import { render, screen } from '@testing-library/react';
import userEvent from '@testing-library/user-event'
import { useState, useEffect } from 'react';
import FlashProvider from './FlashProvider';
import ApiProvider from './ApiProvider';
import UserProvider from './UserProvider';
import { useUser } from './UserProvider';

const realFetch = global.fetch;

beforeEach(() => {
  global.fetch = jest.fn();
});

afterEach(() => {
  global.fetch = realFetch;
  localStorage.clear();
});
```

The mocking in this case is done in the `beforeEach()` handler function, so that it applies to all the test in the file. Since the purpose of mocking is to prevent network access from actually reaching a remote server, the `fetch()` function is the target to mock. Before this function is modified, it is a good idea to save the

original in a global variable, so that it can be restored after the tests are done. Accessing the `fetch()` function as `global.fetch` is done to make the intention of mocking a global function more clear.

The `jest.fn()` function creates a general purpose fake function that can be used to replace any function in the application. Functions created with `jest.fn()` have the interesting property that they record calls made to them, so these calls can then be checked in the test assertions. Jest mocks can be programmed to return specific values, as needed by each test, as you will see next.

To keep things tidy, the `afterEach()` handler reverts the mock back to the original function. Since the `MicroblogApiClient` class stores access tokens in local storage, it is also a good practice to clear any data the might have been stored during a test when the test is done.

Testing a Valid Login

The next listing shows the first test, which logs a user in. This code goes right after the `afterEach()` handler.

Listing 106 src/contexts/UserProvider.test.js: Login test

```
test('logs user in', async () => {
  const urls = [];

  global.fetch
    .mockImplementationOnce(url => {
      urls.push(url);
      return {
        status: 200,
        ok: true,
        json: () => Promise.resolve({access_token: '123'}),
      };
    })
    .mockImplementationOnce(url => {
      urls.push(url);
      return {
        status: 200,
        ok: true,
        json: () => Promise.resolve({username: 'susan'}),
      };
```

```
    });

  const Test = () => {
    const { login, user } = useUser();
    useEffect(() => {
      (async () => await login('username', 'password'))();
    }, []);
    return user ? <p>{user.username}</p> : null;
  };

  render(
    <FlashProvider>
      <ApiProvider>
        <UserProvider>
          <Test />
        </UserProvider>
      </ApiProvider>
    </FlashProvider>
  );

  const element = await screen.findByText('susan');
  expect(element).toBeInTheDocument();
  expect(global.fetch).toHaveBeenCalledTimes(2);
  expect(urls).toHaveLength(2);
  expect(urls[0]).toMatch(/^http.*\/api\/tokens$/);
  expect(urls[1]).toMatch(/^http.*\/api\/me$/);
});
```

The new and most interesting part of this test is at the start of the function. The `urls` array is going to be used to collect all the URLs that the application calls `fetch()` on. These calls are now going to be redirected to the mock function, so the test has full control of what happens in these calls.

The two `mockImplementationOnce()` calls that are made on the `global.fetch` function, which is now a mock function, provide alternative functions that are going to execute each time the application makes a call. The two implementations capture the `url` parameter that was sent by the application and add it to the `urls` array, so that they can be asserted later. Then they return a response that is similar to what the real `fetch()` function returns.

The first API call that the application makes goes to the */api/tokens* endpoint to request an access token. The first fake function registered with the mock returns a success response with status code 200 and a made up `123` token.

Once the API client receives an API token, the `UserProvider` component is going to make a request to the */api/me* endpoint to retrieve the information about the user. This call is handled by the second mocked response, which returns a fictitious user "susan".

Note that `mockImplementationOnce()` is not the only way to configure mocks, and in fact there is an extensive list of methods that can be used to model how the mocked function will behave during the test.

The mock function is now ready to be used. The `Test` component created by the test calls the `login()` function obtained from the `useUser()` hook in a side effect function, and renders the username of the logged-in user.

For the render portion of this test, the `FlashProvider`, `ApiProvider` and `UserProvider` components are added as wrappers to `Test`, so that all the hooks and contexts required during the test are available.

In the assertions section the `screen.findByText()` method is used to wait for the username to be rendered after the effect function completes and the fake user is logged in. Note that the `findBy...()` set of functions are asynchronous, so they need to be awaited. The test function was created as an async function so that `await` can be used here.

After ensuring that the username was rendered, the number of calls made to the `global.fetch()` mock is checked. And then, the `urls` array is also checked, to make sure the correct URLs were requested by the application. A regular expression is used to check the URLs, to avoid discrepancies in the domain and port portions of the URL, which are irrelevant to this test.

Testing an Invalid Login

The following test is similar to the previous one, but it models the case of a user providing invalid credentials.

Listing 107 src/contexts/UserProvider.test.js: Invalid login test

```
test('logs user in with bad credentials', async () => {
  const urls = [];
```

```
  global.fetch
    .mockImplementationOnce(url => {
      urls.push(url);
      return {
        status: 401,
        ok: false,
        json: () => Promise.resolve({}),
      };
    });

  const Test = () => {
    const [result, setResult] = useState();
    const { login, user } = useUser();
    useEffect(() => {
      (async () => {
        setResult(await login('username', 'password'));
      })();
    }, []);
    return <>{result}</>;
  };

  render(
    <FlashProvider>
      <ApiProvider>
        <UserProvider>
          <Test />
        </UserProvider>
      </ApiProvider>
    </FlashProvider>
  );

  const element = await screen.findByText('fail');
  expect(element).toBeInTheDocument();
  expect(global.fetch).toHaveBeenCalledTimes(1);
  expect(urls).toHaveLength(1);
  expect(urls[0]).toMatch(/^http.*\/api\/tokens$/);
});
```

This test programs the `global.fetch()` mock to return a 401 response in its first call, which is what would happen if a user entered an invalid username or password in the login form.

The `Test` component this time renders the return value of the `login()` function returned by the `useUser()` hook. If you recall, this function returns the string

`'ok'` when the login was successful, `'fail'` when the login was invalid, and `'error'` when an unexpected error occurred during the request.

The assertions ensure that "fail" is rendered to the page, in addition to similar checks on the mock and `urls` array.

Testing Logouts

The last authentication test ensures that users can log out of the application.

Listing 108 *src/contexts/UserProvider.test.js*: Invalid login test

```
test('logs user out', async () => {
  global.fetch
    .mockImplementationOnce(url => {
      return {
        status: 200,
        ok: true,
        json: () => Promise.resolve({username: 'susan'}),
      };
    })
    .mockImplementationOnce((url) => {
      return {
        status: 204,
        ok: true,
        json: () => Promise.resolve({}),
      };
    });

  localStorage.setItem('accessToken', '123');

  const Test = () => {
    const { user, logout } = useUser();
    if (user) {
      return (
        <>
          <p>{user.username}</p>
          <button onClick={logout}>logout</button>
        </>
      );
    }
    else if (user === null) {
```

```
      return <p>logged out</p>;
    }
    else {
      return null;
    }
  };

  render(
    <FlashProvider>
      <ApiProvider>
        <UserProvider>
          <Test />
        </UserProvider>
      </ApiProvider>
    </FlashProvider>
  );

  const element = await screen.findByText('susan');
  const button = await screen.findByRole('button');
  expect(element).toBeInTheDocument();
  expect(button).toBeInTheDocument();

  userEvent.click(button);
  const element2 = await screen.findByText('logged out');
  expect(element2).toBeInTheDocument();
  expect(localStorage.getItem('accessToken')).toBeNull();
});
```

This test presents a new challenge, because to be able to test that a user can log out, the user must be first logged in. Instead of repeating a login procedure as in previous tests, this time the test installs a fake access token in local storage, which will make the application believe that it is being started on a browser on which the user is logged in already.

The mocked `fetch()` function for this test includes two calls. The first mocks the response to the */api/me* request issued by the `UserProvider` component. A second mocked is response is included for the token revocation request issued during logout.

The `Test` component used in this test is more complex than before. The component renders a page with the username of the logged-in user and a button to log out. When the button is clicked, the component re-renders with just the text "logged out".

The first group of assertions ensure that both the username and the button are rendered to the page, after the `UserProvider` component gets a chance to load the fake user from the mocked `fetch()` call.

The next step is to simulate a user clicking the button to log out. The React Testing Library includes user-event, a companion library that is specialized in generating fake events. In this case a `click` event is simulated on the `button` element.

Once the click is triggered, a `screen.findByText()` call is issued to retrieve the element with the "logged out" text. Note that this is issued with the asynchronous `findBy...()`, to wait for React to run all asynchronous operations and update the page.

Chapter Summary

- The main purpose of writing automated tests is to ensure that new code does not fail in the future.
- Applications bootstrapped with Create React App integrate the Jest testing framework and the React Testing Library.
- Tests have access to a `render()` function to render the application, a subset, or an individual component.
- Correction functioning is confirmed by querying the results of a render for elements of interest, and asserting that these elements have the expected structure.
- Jest allows the tests to mock timers, remote services and other external entities required by the application, so that the test runs in an isolated, controlled and reproducible environment.

Production Builds

You have an application that you have been using in your own computer during development. How do you put this application in front of your users? In this chapter you are going to learn how to work with production builds of your application.

Development vs. Production Builds

The version of the application that you've been running is called a *development build*. The main goal of a development build is make it easy for the developer to test and debug the application while changes are constantly made.

An important detail to keep in mind is that development builds sacrifice performance and file size in favor of debuggability. Many functions provided by React and related libraries include additional logging or instrumentation, with the purpose of helping the developer detect and fix issues.

A *production build* is a highly optimized version of the application, both in terms of performance and size, intended to be deployed on a production server and used by real users.

Generating React Production Builds

A production build of the application can be generated at any time with the following command:

```
npm run build
```

The command will run for a minute or two, producing output similar to the following:

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

File sizes after gzip:

```

```
  85.78 kB (-6.46 kB)  build/static/js/main.52a6162f.js
  24.57 kB             build/static/css/main.d11fbe84.css
  1.78 kB (-96 B)      build/static/js/787.49411143.chunk.js

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

The build folder is ready to be deployed.
You may serve it with a static server:

  npm install -g serve
  serve -s build

Find out more about deployment here:

  https://cra.link/deployment
```

The command creates the production build of the application in the *build* subdirectory of the project. The build process performs the same code checks as the development build and will print any warnings found in your code.

The `serve` package recommended in the output of the build command was installed locally in Chapter 2. To test the production build on your computer, you can start a web server as follows:

```
npx serve -s build
```

Note that for this command to work, you have to stop the development web server, which also runs on port 3000.

Structure of a Production Built Application

After the production build completes, the *build* subdirectory contains the files that should be deployed to the root of your production web server. Some files you'll find in this directory are:

- *index.html*: The entry point of the application.
- *favicon.ico*: The application icon, displayed in browser tabs.
- *manifest.json*: The web application manifest, which provides instructions on how the application can be installed on the user's device.
- Additional icon files of different sizes for use in mobile devices.

The *build* directory also has a subdirectory of its own named *static*. This is where the application files are located.

Inside *static*, there are two subdirectories called *js* and *css*, which hold the JavaScript and CSS files respectively. The application code is stored in files that start with a `main.` prefix. Depending on the application and the third-party packages used, a number of additional JavaScript and CSS files can be included in the build.

For each *.js* and *.css* file, React also includes corresponding *.js.map* and *.css.map*. Map files make it possible to perform some debugging tasks on the production build. As a side effect of that they may also allow your users to inspect a fairly readable version of your application's source code, so depending on your particular case you may decide not to copy the map files to the production server.

All the files inside the *static* subdirectory have hashes in their filenames, which are unique for each build. For this reason, and with the goal of improving performance, the contents of this directory can be configured to have a long caching controls if the web server provides this option.

For applications that are very large, the main JavaScript bundle may end up being a fairly large file that can adversely affect the application start up time. React supports a technique called code splitting that can be used to move less frequently used components to separate bundle files that are loaded on demand when the components are first rendered.

Deploying the Application

To deploy a React application you have to publish all the files that are inside the *build* subdirectory (except maybe the *.map* files) to the root directory on a production web server, preserving the directory structure. There are many options for deployment of React applications, some of which are discussed in the following sections.

An important deployment detail that needs to be observed, is that the web server must serve *index.html* whenever a non-existent path is requested. This is necessary to support client-side routing. If a user bookmarked a specific client route and attempts to start the application from it, a standard static file web

server will return a 404 response, since the requested resource is not known in the server. To let React-Router handle this URL in the client, the *index.html* page should be served. For the `serve` command, the `-s` option achieves this.

The `serve` command

If you have access to a server connected to the Internet, then the simplest option is actually to use the `serve` command, as shown above. This web server can be configured to listen on a different port other than the default 3000, if desired. It also has options to add TLS encryption through an SSL certificate.

Static Site Hosting Services

There is a great variety of services that offer static file serving, many at no cost. Here are some of them:

- Netlify
- Cloudflare
- GitHub Pages
- Digital Ocean's App Platform
- Heroku
- AWS
- and many more.

For many of these options, the site can be deployed with a single terminal command, which is very convenient because you can incorporate it as a custom `npm` command in the `scripts` section of the *package.json* file as shown in the following example:

```
{
  ...
  "scripts": {
    ...
    "deploy": "npm run build && [your deploy command here]"
  },
  ...
}
```

This would allow you to deploy your application with `npm`:

```
npm run deploy
```

Back End Static Files

In addition to the production build of the React application, you will need to deploy your back end. The method of deployment for the back end will vary depending on the language and framework used, so it is outside the scope of this book to cover this aspect of the deployment.

Aside from specific back end deployment methods, it is very common in back end frameworks to provide an option to serve static files alongside its endpoints. If your back end provides this option, you can configure the static directory for your back end framework to be the *build* directory of your React application, and then both front and back ends will be served by the back end framework's web server.

Reverse Proxy

Many back end services use a reverse proxy such as Nginx as the public facing web server. For this type of installation, the reverse proxy can be configured to serve the static files of the React application's build and to proxy to the back end for paths that have a predefined prefix such as */api*.

The following nginx configuration is an example of this technique. Here the React application is assumed to be in */home/ubuntu/react-microblog*, and the back end server is listening on port 5000 in the same host.

```
server {
    listen       80;
    server_name  localhost;

    root   /home/ubuntu/react-microblog/build;
    index  index.html;

    location / {
        try_files $uri $uri/ /index.html;
        add_header Cache-Control "no-cache";
    }

    location /static {
        expires 1y;
```

```
        add_header Cache-Control "public";
    }

    location /api {
        proxy_pass http://localhost:5000;
    }
}
```

This configuration sets the web root to the *build* directory of the React project. It configures URL paths starting with */api* to be proxied over to *http://localhost:5000*, while all others are served as static files.

The `try_files` directive defines */index.html* as its final argument, so that Nginx serves the React application for any paths that have no matching file on disk. This allows bookmarks to specific routes such as */explore* or */user/susan* in the client application to work, instead of returning a 404 error.

The caching recommendations in the Create React App documentation are cache the contents of the */static* directory for one year, while serving all other files with caching disabled. This is implemented with the `add_header` and `expires` directives in the above configuration.

Production Deployment with Docker

Just to give you an idea of what effort goes into creating a production deployment of the React application plus its back end, in this section you'll learn how to create a complete Docker deployment.

Dockerizing the React Application

While it is possible to create a Docker container that uses the `serve` command discussed above to serve the React application, a better option is to create a more elaborate deployment based on Nginx, which as you've seen above, can be configured to serve static files, while at the same time act as a proxy to a back end.

Nginx has an official container image on DockerHub that is perfect to use as a base image. The "How to use this image" section of its documentation, shows an example *Dockerfile* for a derived image that incorporates a custom build

directory. Using this as a guide, a *Dockerfile* for the React front end can be written as follows:

Listing 109 DockerFile: A simple Dockerfile for the React application

```
FROM nginx
COPY build/ /usr/share/nginx/html/
```

This *Dockerfile* uses the official Nginx image as a base, and adds the React production build files in the appropriate directory where Nginx looks for static files to serve.

Before building the container image, make sure you have an updated production build:

```
npm run build
```

Now, assuming you have Docker installed, you can build a container image for the application with the following command:

```
docker build -t react-microblog .
```

Once the image is built, you can launch a container based on this image with the `docker run` command:

```
docker run -p 8080:80 --name microblog -d react-microblog
```

The `-p` option tells Docker to map port 80 on the container, which is the port on which Nginx listens for requests, to port 8080 on the host computer. The `--name` option gives the container a friendly name, and the `-d` option runs the container in the background, giving you back control of the terminal prompt. The `react-microblog` at the end of the command is the name of the image to launch.

When this command executes, Docker will print the ID of the launched container to the terminal. You can now open *http://localhost:8080* on your browser to access the React application running as a Docker container. Note that at this point the back end configuration is the same that you've used during development, so the same back end service will be used. Reusing a back end between development and production is not a good solution for a real-world project, but it means that until a separate production back end is started you can

test the container by logging in with the same account you've been using before.

When you are done testing, you can stop and delete the container with this command:

```
docker rm -f microblog
```

The `-f` option tells Docker to force the removal of the container. This is necessary when removing a container that is running. The `microblog` name in this command is a reference to the value given for `--name` option in the `docker run` command. Alternatively, you can reference a container by its ID, which was printed by the `docker run` command.

Using a Custom Nginx Configuration

The official Nginx container image comes with its own default configuration, sufficient when all you need to do is serve some static files. The `server` section of this default configuration is stored in */etc/nginx/conf.d/default.conf* inside the Nginx container image. To have a specialized configuration, this file can be replaced during the build of the derived image.

The following server configuration is based on the Nginx example configuration shown earlier in this chapter. Store it in a *nginx.default.conf* file in the React project.

Listing 110 nginx.default.conf: Custom Nginx configuration

```
server {
    listen       80;
    server_name  localhost;

    root   /usr/share/nginx/html;
    index  index.html;

    location / {
        try_files $uri $uri/ /index.html;
        add_header Cache-Control "no-cache";
    }

    location /static {
        expires 1y;
```

```
        add_header Cache-Control "public";
    }

    location /api {
        proxy_pass http://api:5000;
    }
}
```

This version of the configuration uses the most appropriate settings for the production build files of the React application. An */api* location is also added to proxy requests to a back end server, which is assumed to be running on a host named `api`, on port 5000. In the next section, you'll add a second container for the back end, and this container will be associated with the `api` hostname.

This custom Nginx configuration can be added to the *Dockerfile*, replacing the stock configuration provided with the Nginx container image. Below you can see the updated *Dockerfile* for the application.

Listing 111 *DockerFile*: Custom Nginx configuration in the Dockerfile

```
FROM nginx
COPY build/ /usr/share/nginx/html/
COPY nginx.default.conf /etc/nginx/conf.d/default.conf
```

A Docker-Compose Configuration for Front and Back Ends

The new Nginx configuration serves the React production build files, and forwards all requests with a path starting with */api* to a back end running at *http://api:5000*. While it is possible to manually start a second container for the back end, and then set up a private network that connects the two containers, this type of complex set up with multiple networked containers is better created using Docker Compose.

A Docker Compose configuration is written in a file named *docker-compose.yml*, and it describes a collection of containers running services that need to communicate with each other. The general structure of this file follows this pattern:

```
version: '3'
services:
```

```
  frontend:
    # <-- configuration of the frontend container
  api:
    # <-- configuration of the api container
# <-- other options
```

The `frontend` and `api` names in this example are arbitrary names given to the containers that are going to be part of the deployment. Docker Compose sets these names as hostnames in each of the containers. So for example, the `frontend` container will be able to proxy requests to the `api` container using `api` as hostname. And while this serves no purpose in this deployment, the `api` container can also reference the `frontend` container by its name.

Below you can see a complete Docker Compose configuration that includes a container for the React front end and another one for the Microblog API back end. The back end is given a volume to use as storage for its database.

Listing 112 *docker-compose.yml*: A Docker Compose configuration

```
version: '3'
services:
  frontend:
    build: .
    image: react-microblog
    ports:
      - "8080:80"
    restart: always
  api:
    build: ../microblog-api
    image: microblog-api
    volumes:
      - type: volume
        source: data
        target: /data
    env_file: .env.api
    environment:
      DATABASE_URL: sqlite:////data/db.sqlite
    restart: always
volumes:
  data:
```

The `frontend` container includes options that are similar to those used in the

`docker run` command shown above. The `build` option configures the directory where a *Dockerfile* for the container image is located, which for this container is the current directory. The `image` option sets the name of the container image that will be built. The `ports` option configures the mapping of network ports between host and container. The `restart` option specifies that the container should be automatically restarted if it ever dies.

The definition of the `api` container is based on the docker-compose.yml file from the Microblog API project. This `build` option assumes that the repository for the Microblog API project is a sibling of the current directory. The `image` option sets the name of the API container image. The `volumes` section creates a mount for a data volume, where the database file will be stored. The benefit of putting the database in a mounted volume over having it in the container's own file system is that the container image can be updated without affecting the data. The `env_file` option configures a *.env.api* file with environment variables for the container (you will create this file soon). The `environment` section defines the `DATABASE_URL` environment variable explicitly, using the */data* path where the volume was configured to be mounted. The `restart` option ensures that the container is restarted if it is ever interrupted, as with the `frontend` container. At the bottom there is a `volumes` top-level section that creates the volume named `data`, defined as a mount by the `api` container.

The `api` container does not have a `ports` section. Port mapping is only necessary for services that need to be public, as this is what ties a container port to an actual network port on the host computer. This is required in the front end container because that's where the public facing web server (Nginx) runs. The back end container does not need to be accessible to the outside world, so it does not need port mapping. The network that Docker Compose creates will allow the two containers to communicate privately.

Configuring the Docker Production Deployment

In previous chapters you have provided configuration options for the front and back ends in *.env* files. Using a single *.env* for configuration is often insufficient, because it does not allow options to change between development and production builds. Consider the `REACT_APP_BASE_API_URL` variable, which configures the base URL of the back end inside the React application. This variable needs to have a different value then the one currently in use when a production deployment is created.

The build support that was added to the project by Create React App has a set of rules for importing environment variables not only from *.env* but also from other files. In particular, any variables defined in files *.env.development*, *.env.production* and *.env.test* override those in *.env* for their specific environment.

To be able to redefine `REACT_APP_BASE_API_URL`, create a *.env.production* file in the root of your React project, and enter the following variable in it:

Listing 113 *.env.production*: Production configuration

```
REACT_APP_BASE_API_URL=
```

Why is this variable empty for a production build? Remember than Nginx is going to be the point of contact for both the front and back ends. The Nginx configuration will determine that a request is for the back end when the URL path starts with */api*, but to the left of the URL path, URLs for front and back end are going to be the same. By setting the base URL to empty the front end is going to send requests that start with the path, which means that the browser will use the same base URL that was used to load the front end application.

The back end container also needs its configuration. You already have a valid configuration for it, that you've been using since Chapter 5. For this deployment, a copy of this configuration needs to be written in the React project, with name *.env.api*, as referenced in the *docker-compose.yml* file. An example configuration file for the API is shown below:

Listing 114 *.env.api*: Back end configuration

```
MAIL_SERVER=smtp.sendgrid.net
MAIL_PORT=587
MAIL_USE_TLS=true
MAIL_USERNAME=apikey
MAIL_PASSWORD=xxxxxxxxxxxxxxxxx
MAIL_DEFAULT_SENDER=donotreply@microblog.example.com
```

This example configuration file uses SendGrid as email server. The `MAIL_PASSWORD` variable is shown with a placeholder value that you need to replace with your own SendGrid API key. If you've used a different email

provider, you can keep using it for your production deployment.

Building the Docker Production Images

Before you build your production container images, make sure that you are in the *react-microblog* directory, where you have created the *docker-compose.yml*, *Dockerfile*, *.env.production* and *.env.api* files. The *docker-compose.yml* includes a relative path that points to the back end project directory, assuming it is a sibling directory. If you don't have the back end project cloned on your computer, clone it now with the following commands:

```
cd ..
git clone https://github.com/miguelgrinberg/microblog-api
cd react-microblog
```

Note how the `cd` commands are used to ensure that the Microblog API project is installed outside the React project, and under the same parent directory.

From the *react-microblog* directory, you can now build the front and back end container images with these two commands:

```
npm run build
docker-compose build
```

The first command creates a production build of the React application. This needs to be done before the Docker build, so that when Docker builds the container image for the front end, it copies the up-to-date production build files that are built with the production configuration you recently added.

The second command tells Docker Compose to go over all the containers declared in the `services` section of the *docker-compose.yml* file and build the container images for them.

Once the build is complete, you should be able to see them listed when you run the `docker images` command:

```
$ docker images
REPOSITORY                    TAG         IMAGE ID       CREATED
microblog-api                 latest      93e6d0414110   3 minutes
react-microblog               latest      d9f09e24b646   2 minutes
nginx                         latest      12766a6745ee   2 weeks ag
```

```
python                          3.10-slim   9328aba7e797   4 months a
```

In addition to the two images that were just built, Docker has copies of the images on which these two are based. You may see additional images in the list as well, depending on what other projects you use Docker for.

Starting the Production Deployment

After you have built your images, you can start a deployment using the `docker-compose up` command as follows:

```
$ docker-compose up -d
[+] Running 3/3
 - Network react-microblog_default       Created
 - Container react-microblog-api-1       Started
 - Container react-microblog-frontend-1  Started
```

The `-d` option runs the deployment as a background process, returning the prompt so that you can continue working on the terminal session. The output of the command shows that a private network has been created, and two containers were started.

The *docker-compose.yml* file defines a mapping between port 80 in the front end container and port 8080 in the host computer, so you now have the production version of the application listening on port 8080. You can check that the application is running by navigating to *http://localhost:8080*. If you want to connect from another device, replace `localhost` with the IP address or hostname of the computer running Docker.

This is a completely separate deployment that does not share data with the development back end that you've been using, so you will need to create a new account. The database on this deployment is also completely empty, so you will not see posts from other users until more accounts are created, and they start contributing content.

What happens next? This is pretty much it. If you follow the deployment steps on your production host, when you reach this point you have a fully working application. While this is outside the scope of this article, the Nginx configuration can be further expanded to use a proper domain name that is associated with an SSL certificate.

But you were probably testing the deployment on your development computer, so you need a way to remove these containers when you are done testing. The `docker-compose down` command achieves that:

```
$ docker-compose down
[+] Running 3/3
 - Container react-microblog-api-1       Removed
 - Container react-microblog-frontend-1  Removed
 - Network react-microblog_default       Removed
```

Simplifying the Deployment Process

As you've seen in the previous sections, creating a production deployment involves a few actions:

- Run `npm run build` to generate an updated production build of the React project
- Run `docker-compose build` to generate updated container images for front and back ends.
- Run `docker-compose up -d` to start or restart the production containers.

Remembering these steps can be hard, especially if you don't need to deploy updates very often. To avoid errors in the process, it is best to create a wrapper script that performs all these steps.

You've seen earlier that it is possible to create a custom command for `npm`, so that technique can be applied here. In the *package.json* file, locate the `scripts` section and add an entry for a `deploy` command:

Listing 115 package.json: Custom deploy command

```
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "deploy": "npm run build && docker-compose build && docker-com
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
```

With this new command in place, you can start or update a deployment with a

single command:

```
npm run deploy
```

Chapter Summary

- Development builds have instrumentation that makes debugging easier, at the cost of reduced performance.
- Production builds are optimized for performance, but are more difficult to debug.
- Use the `npm run build` command to generate a production build.
- Quickly test a production build with `npx serve -s build`.
- To deploy your project to production, the contents of the *build* subdirectory must be served on a static file web server.
- Nginx is an ideal production web server to combine serving the React application files and proxying requests to a back end.

Next Steps

Congratulations on completing this course! In this short section I want to give you some pointers on how to continue on your learning path.

If you are interested in learning more about front end development with React:

- The React documentation should be a main destination, especially to stay up to date with changes and improvements introduced in new releases.
- Now that you are familiar with React, you can explore other related projects, such as:
  - Next.js, a framework that extends React by providing server-side rendering and other benefits.
  - Gatsby, a framework that is optimized for creating static (or mostly static) sites.
  - React Native, a framework for building React applications that run natively on Android and iOS devices.

If you are interested in learning about back end development, the field is full of options. The beauty of front and back end using standard communication protocols such as HTTP, WebSocket or GraphQL is that the back end can be written in pretty much any programming language or web framework.

My personal experience is with back ends developed using the Python language

and the Flask web framework. If you enjoyed working with the Microblog API back end and would like to learn more about it, here are some pointers:

- You can learn by inspecting the Microblog API source code, which is open source and available on GitHub.
- My Flask Mega-Tutorial has a chapter on designing APIs.
- The Flask Web Development book, published by O'Reilly Media, also covers APIs.
- My personal blog has a variety of articles covering various aspects of web development with Python and Flask.

If Python and/or Flask are not your thing, you should be able to find resources on writing APIs and services in your favorite programming language and web framework.

I would like to thank you for allowing me to share my experience writing React applications with you. I hope that this tutorial has given you the tools that you need to embark on your own front end projects with confidence.

Best of luck with your front end development journey!

Index

H

I

J

M

N

P

R

S

T

Semicolons

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

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

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

Trailing Commas

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

const myObject = {
```

```
  name: 'susan',
  age: 20,
};
```

Imports and Exports

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

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

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

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

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

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

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

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

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

Variables and Constants

```
let a;
```

```
let a = 1;
```

```
const c = 3;

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

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

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

Equality and Inequality Comparisons

```
let a = 1;

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

String Interpolation

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

For-Of Loops

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

Arrow Functions

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

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

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

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

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

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

square(2);  // 4
```

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

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

Promises

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

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

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

Async and Await

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

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

```
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}`);
  }
}
```

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

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

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

Spread Operator

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

```
console.log(Math.min(...a));  // 2
```

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

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

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

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

Object Property Shorthand

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

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

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

Destructuring Assignments

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

```
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]
```

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

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

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

Classes

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

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

JSX

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

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

```
const myTable = (
  <table>
    <tr>
      <th>Name</th>
      <th>Age</th>
    </tr>
    <tr>
      <td>Susan</td>
      <td>20</td>
    </tr>
    <tr>
      <td>John</td>
      <td>45</td>
    </tr>
  </table>
);
```