REACT
KEY CONCEPTS

Consolidate your knowledge of React's core features

Maximilian Schwarzmüller

packt

Table of Contents

PREFACE

ABOUT THE BOOK

As the most popular JavaScript library for building modern, interactive user interfaces, React is an in-demand framework that'll bring real value to your career or next project. But like any technology, learning React can be tricky, and finding the right teacher can make things a whole lot easier.

Maximilian Schwarzmüller is a bestselling instructor who has helped over two million students worldwide learn how to code, and his latest React video course (React—The Complete Guide) has over six hundred thousand students on Udemy.

Max has written this quick-start reference to help you get to grips with the world of React programming. Simple explanations, relevant examples, and a clear, concise approach make this fast-paced guide the ideal resource for busy developers.

This book distills the core concepts of React and draws together its key features with neat summaries, thus perfectly complementing other in-depth teaching resources. So, whether you've just finished Max's React video course and are looking for a handy reference tool, or you've been using a variety of other learning material and now need a single study guide to bring everything together, this is the ideal companion to support you through your next React projects. Plus, it's fully up to date for React 18, so you can be sure you're ready to go with the latest version.

ABOUT THE AUTHOR

Maximilian Schwarzmüller is a professional web developer and bestselling online course instructor. Having learned to build websites and web user interfaces the hard way with just HTML, CSS, and (old-school) JavaScript, he embraced modern frontend frameworks and libraries like Angular and React right from the start.

Having the perspective of a self-taught freelancer, Maximilian started teaching web development professionally in 2015. On Udemy, he is now one of the most popular and biggest online instructors, teaching more than 2mn students worldwide. Students can become developers by exploring more than 40 courses, most of those courses being bestsellers in their respective categories. In 2017, together with a friend, Maximilian also founded Academind to deliver even more and better courses to even more students. For example, Academind's "React – The Complete Guide" course is the bestselling React course on the Udemy platform, reaching more than 500,000 students.

Besides helping students from all over the world as an online instructor, Maximilian never stopped working as a web developer. He still loves exploring and mastering new technologies, building exciting digital products, and sharing his knowledge with fellow developers. He's driven by his passion for good code and engaging websites and apps. Beyond web development, Maximilian also works as a mobile app developer and cloud expert. He holds multiple AWS certifications, including the "AWS Certified Solutions Architect – Professional" certification.

Apart from his courses on Udemy, Maximilian also publishes free tutorial videos on Academind's YouTube channel (https://youtube.com/c/academind) and articles on academind.com. You can also follow him on Twitter (@maxedapps).

AUDIENCE

This book is designed for developers who already have some familiarity with React basics. It can be used as a standalone resource to consolidate understanding or as a companion guide to a more in-depth course. To get the most value from this book, it is advised that readers have some understanding of the fundamentals of JavaScript, HTML, and CSS.

PROSPECTIVE TABLE OF CONTENTS

Chapter 1, *React – What and Why*, will re-introduce the reader to React.js. Assuming that React.js is not brand-new to the reader, this chapter will clarify which problems React solves, which alternatives exist, how React generally works, and how React projects may be created.

Chapter 2, *Understanding React Components and JSX*, will explain the general structure of a React app (a tree of components) and how components are created and used in React apps.

Chapter 3, *Components and Props*, will ensure that readers are able to build reusable components by using a key concept called "props".

Chapter 4, *Working with Events and State*, will cover how to work with state in React components, which different options exist (single state vs multiple state slices) and how state changes can be performed and used for UI updates.

Chapter 5, *Rendering Lists and Conditional Content*, will explain how React apps can render lists of content (e.g. list of user posts) and conditional content (e.g. alert if incorrect values were entered into an input field).

Chapter 6, *Styling React Apps*, will clarify how React components can be styled and how styles can be applied dynamically or conditionally, touching on popular styling solutions like vanilla CSS, styled components, and CSS modules for scoped styles.

Chapter 7, *Portals and Refs,* will explain how direct DOM access and manipulation is facilitated via the "refs" feature which is built-into React. In addition, readers will learn how Portals may be used to optimize the rendered DOM element structure.

Chapter 8, *Handling Side Effects*, will discuss the **`useEffect`** hook, explaining how it works, how it can be configured for different use-cases and scenarios, and how side effects can be handled optimally with this React hook.

Chapter 9, *Behind the Scenes of React and Optimization Opportunities*, will take a look behind the scenes of React and dive into core topics like the virtual DOM, state update batching and key optimization techniques that help developers avoid unnecessary re-render cycles (and thus improve performance).

Chapter 10, *Working with Complex State*, will explain how the advanced React hook **`useReducer`** works, when and why you might want to use it and how it can be used in React components to manage more complex component state with it. In addition, React's Context API will be explored and discussed in-depth, allowing developers to manage app-wide state with ease.

Chapter 11, *Building Custom React Hooks*, will explain how developers can build their own, custom React hooks and what the advantage of doing so is.

Chapter 12, *Multipage Apps with React Router*, will explain what React Router is and how this extra library can be used to build multipage experiences in a React single-page-application.

Chapter 13, *Managing Data with React Router*, will dive deeper into React Router and explore how this package can also help with fetching and managing data.

Chapter 14, *Next Steps and Further Resources*, will further cover the core and "extended" React ecosystem and which resources may be helpful for next steps.

CONVENTIONS

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Store the paragraph element reference in a constant named **`paragraphElement`**."

Words that you see on the screen, for example, in menus or dialog boxes, also appear in the text like this: "In the header with the navigation bar you will find the following components: the navigation items (**`Login`** and **`Profile`**) and the **`Logout`** button."

A block of code is set as follows:

```
const buttonElement = document.querySelector('button');
const paragraphElement = document.querySelector('p');
function updateTextHandler() {
  paragraphElement.textContent = 'Text was changed!';
}
buttonElement.addEventListener('click', updateTextHandler);
```

New terms and important words are shown like this: "It is currently used by over 5% of the top 1,000 websites and compared to other popular frontend JavaScript frameworks like Angular, React is leading by a huge margin, when looking at key metrics like weekly package downloads via **npm** (**Node Package Manager**), which is a tool commonly used for downloading and managing JavaScript packages."

SETTING UP YOUR ENVIRONMENT

Before you can successfully install React.js on your system, you will need to ensure you have the following software installed:

Node.js and **npm** (included with your installation by default)

These are available for download at https://nodejs.org/en/.

The home page of this site should automatically provide you with the most recent installation options for your platform and system. For more options, select **`Other Downloads`** (the first of three links visible beneath each of your default options). This will open a new page through which you can explore all installation choices for all main platforms, as shown in the screenshot below:

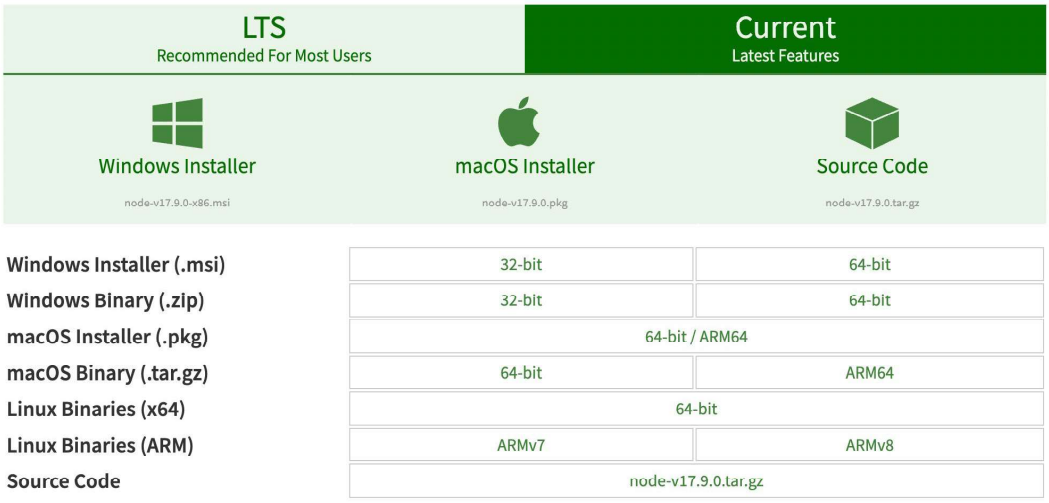


Figure 0.1: Download Node.js source code or pre-built installer

At the bottom of this page, you will find a bullet list of available resources should your system require specialised instructions, including guidance on Node.js installation via source code and node package manager.

Once you have downloaded Node.js through this website, find the **`.pkg`** file in your downloads folder. Double-click this file to open the **`Install Node.js`** pop-up window, then simply follow given instructions to complete your installation.

INSTALLING REACT.JS

React.js projects can be created in various ways, including custom-built project setups that incorporate webpack, babel and other tools. The easiest and recommended way is the usage of the **`create-react-app`** command though. This book uses this method. The creation of a react app will be covered in *Chapter 1*, *React.js – What and Why*, but you may refer to this section for step-by-step instructions on this task.

> **NOTE**
>
> For further guidance regarding the installation and setup of your React.js environment, resources are available at the following: https://reactjs.org/docs/getting-started.html

Perform the following steps to install React.js on your system:

1. Open your terminal (Powershell/command prompt for Windows; bash for Linux).
2. Use the make directory command to create a new project folder with a name of your choosing (e.g. **`mkdir react-projects`**) and navigate to that directory using the change directory command (e.g. **`cd react-projects`**).
3. Enter the following command prompt to create a new project directory within this folder:

```
npx create-react-app my-app
```

4. Grant permission when prompted to install all required files and folders needed for basic React setup. This may take several minutes.
5. Once completed, navigate to your new directory using the cd command:

```
cd my-app
```

6. Open a terminal window in this new project directory and run the following command to start a Node.js development server and lauch a new browser to preview your app locally:

```
npm start
```

7. This should open a new browser window automatically, but if it does not, open your browser manually type **`http://localhost:3000`** in the address/location bar to navigate to **`localhost:3000`**, as shown in the screenshot below:

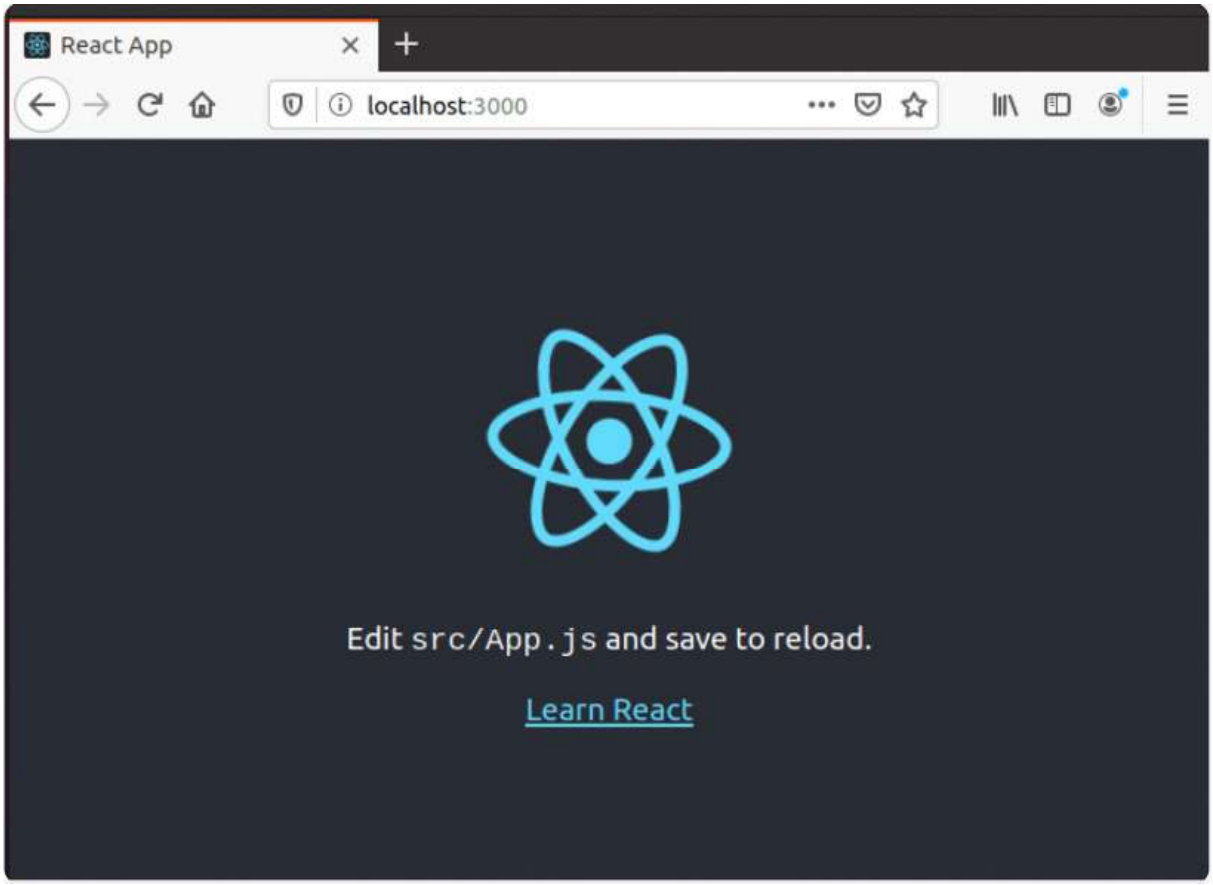


Figure 0.2: Access React App in Your Browser

8. When you are ready to stop development for the time being, use **`Ctrl`** + **`C`** in the same terminal as in step 6 to quit running your server. To relaunch it, simply run the **`npm start`** command in that terminal once again. Keep the process started by **`npm start`** up and running while developing, as it will automatically update the website loaded on **`localhost:3000`** with any changes you make.

1

REACT – WHAT AND WHY

LEARNING OBJECTIVES

By the end of this chapter, you will be able to do the following:

- Describe what React is and why you would use it
- Compare React to web projects built with just JavaScript
- Create new React projects

INTRODUCTION

React.js (or just **React**, as it's also called and as it'll be referred to for the majority of this book) is one of the **most popular frontend JavaScript libraries—**maybe even the most popular one, according to a 2021 Stack Overflow developer survey. It is currently used by over 5% of the top 1,000 websites and compared to other popular frontend JavaScript frameworks like Angular, React is leading by a huge margin, when looking at key metrics like weekly package downloads via **npm** (**Node Package Manager**), which is a tool commonly used for downloading and managing JavaScript packages.

Though it is certainly possible to write good React code without fully understanding how React works and why you're using it, you should always aim to understand the tools you're working with as well as the reasons for picking a certain tool in the first place.

Therefore, before considering anything about its core concepts and ideas or reviewing example code, you first need to understand **what** React actually is and why it exists. This will help you understand how React works internally and why it offers the features it does.

If you already know why you're using React, why solutions like React in general are being used instead of **vanilla JavaScript** (i.e. JavaScript without any frameworks or libraries, more on this in the next section), and what the idea behind React and its syntax is, you may of course skip this section and jump ahead to the more practice-oriented chapters later in this book.

But if you only *think* that you know it and are not 100% certain, you should definitely follow along with this chapter first.

WHAT IS REACT?

React is a JavaScript library, and if you take a look at the official webpage (Official React website and documentation: https://reactjs.org), you learn that it's actually a *"JavaScript library for building user interfaces"*.

But what does this mean?

It should be clear what JavaScript is and why you use JavaScript in the browser (React is mostly a browser-side JavaScript library). JavaScript allows you to add interactivity to your website since, with JavaScript, you can react to user events and manipulate the page after it was loaded. This is extremely valuable as it allows you to build highly interactive web user interfaces.

But what is a "library" and how does React help with "building user interfaces"?

While you can have philosophical discussions about what a library is (and how it differs from a framework), the pragmatic definition of a library is that it's a collection of functionalities that you can use in your code to achieve results that would normally require more code and work from your side. Libraries help you write better and shorter code and enable you to implement certain features more quickly. In addition, since you can focus on your "core business logic", you not only move faster but are also likely to produce better code since you don't have to reinvent the wheel for problems that have been solved before by others.

React is such a library—one that focuses on providing functionalities that help you create interactive and reactive user interfaces. Indeed, React deals with more than web interfaces (i.e. websites loaded in browsers). You can also build native mobile devices with React and React Native, which is another library that utilizes React under the hood. No matter which platform you're targeting though, creating interactive user interfaces with just JavaScript can quickly become very complex and overwhelming.

> **NOTE**
>
> The focus of this book is on React in general, and for simplicity, the focus lies on React for websites. With projects like React Native, you can also use React to build user interfaces for native mobile apps. The React concepts covered in this book still apply, no matter which target platform is chosen. But examples will focus on React for web browsers.

THE PROBLEM WITH "VANILLA JAVASCRIPT"

Vanilla JavaScript is a term commonly used in web development for referring to "JavaScript without any frameworks or libraries". That means, you write all the JavaScript on your own, without falling back to any libraries or frameworks that would provide extra utility functionalities. When working with vanilla JavaScript, you especially don't use major frontend frameworks or libraries like React or Angular.

Using vanilla JavaScript generally has the advantage, that visitors of a website have to download less JavaScript code (as major frameworks and libraries typically are quite sizeable and can quickly add 50+ kb of extra JavaScript code that has to be downloaded).

The downside of relying on vanilla JavaScript is, that you, as the developer, must implement all functionalities from the ground up on your own. This can be error-prone and highly time consuming. Therefore, especially more complex user interfaces and websites can quickly become very hard to manage with vanilla JavaScript.

React simplifies the creation and management of such user interfaces by moving from an imperative to a declarative approach. Though this is a nice sentence, it can be hard to grasp if you haven't worked with React or similar frameworks before. To understand it and the idea behind "imperative vs declarative approaches" and why you might want to use React instead of just vanilla JavaScript, it's helpful to take a step back and evaluate how vanilla JavaScript works.

Here's a short code snippet that shows how you could handle the following user interface actions with vanilla JavaScript:

1. Add an event listener to a button to listen for "click" events.
2. Replace the text of a paragraph with new text once a click on the button occurs.

```
const buttonElement = document.querySelector('button');
const paragraphElement = document.querySelector('p');

function updateTextHandler() {
  paragraphElement.textContent = 'Text was changed!';
}

buttonElement.addEventListener('click', updateTextHandler);
```

This example is deliberately kept simple, so it's probably not looking too bad or overwhelming. It's just a basic example to show how code is generally written with vanilla JavaScript (a more complex example will be discussed below). But even though this example is very straightforward and easy to digest, working with vanilla JavaScript will quickly reach its limits for feature-rich user interfaces and the code to handle various user interactions therefore also becomes more complex. Code can quickly grow significantly, and therefore maintaining it can become a challenge.

In the preceding example, code is written with vanilla JavaScript and, therefore, imperatively. This means that you write instruction after instruction, and you describe every step that needs to be taken in detail.

The code shown above could be translated to these more human-readable instructions:

1. Look for an **HTML** element of type **`button`** to obtain a reference to the first button on the page.
2. Create a constant (i.e., a data container) named **`buttonElement`** that holds that button reference.
3. Repeat step 1 but get a reference to the first element that is of type of **`p`**.
4. Store the paragraph element reference in a constant named **`paragraphElement`**.
5. Add an event listener to the **`buttonElement`** that listens for **`click`** events and triggers the **`updateTextHandler`** function whenever such a **`click`** event occurs.
6. Inside the **`updateTextHandler`** function, use the **`paragraphElement`** to set its **`textContent`** to **`"Text was changed!"`**.

Do you see how every step that needs to be taken is clearly defined and written out in the code?

This shouldn't be too surprising because that is how most programming languages work: you define a series of steps that must be executed in order. It's an approach that makes a lot of sense because the order of code execution shouldn't be random or unpredictable.

But when working with user interfaces, this imperative approach is not ideal. Indeed, it can quickly become cumbersome because, as a developer, you have to add a lot of instructions that despite adding little value, cannot simply be omitted. You need to write all the DOM (**Document Object Model**) instructions that allow your code to interact with elements, add elements, manipulate elements. etc.

Your core business logic (e.g., deriving and defining the actual text that should be set after a click) therefore often makes up only a small chunk of the overall code. When controlling and manipulating web user interfaces with JavaScript, a huge chunk (often the majority) of your code is frequently made up of DOM instructions, event listeners, HTML element operations, and UI state management.

Therefore, you end up describing all the steps that are required to interact with the UI technically **and** all the steps that are required to derive the output data (i.e., the desired final state of the UI).

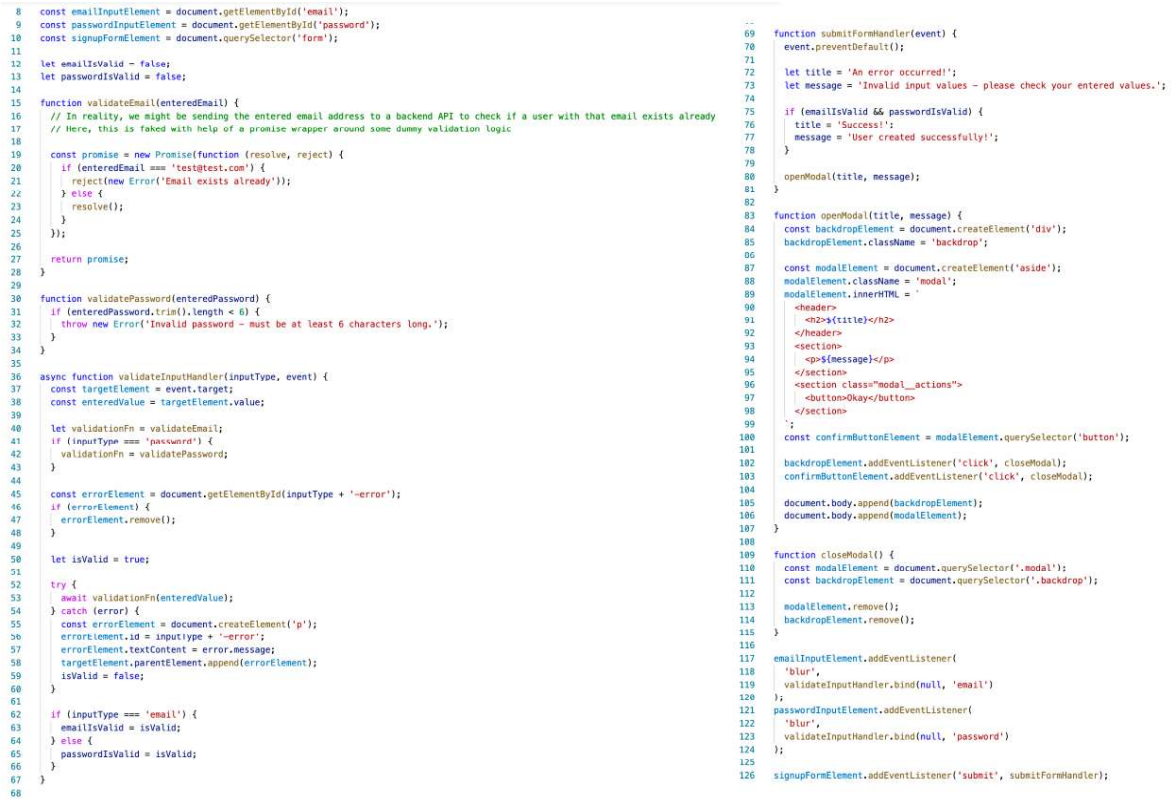
> **NOTE**
>
> This book assumes that you are familiar with the DOM (Document Object Model). In a nutshell, the DOM is the "bridge" between your JavaScript code and the HTML code of the website with which you want to interact. Via the built-in DOM **API**, JavaScript is able to create, insert, manipulate, delete, and read HTML elements and their content.
>
> You can learn more about the DOM in this article: https://academind.com/tutorials/what-is-the-dom.

Modern web user interfaces are often quite complex, with lots of interactivity going on behind the scenes. Your website might need to listen for user input in an input field, send that entered data to a server to validate it, output a validation feedback message on the screen, and show an error overlay modal if incorrect data is submitted.

This is not a complex example in general, but the vanilla JavaScript code for implementing such a scenario can be overwhelming. You end up with lots of DOM selection, insertion, and manipulation operations, as well as multiple lines of code that do nothing but manage event listeners. And keeping the DOM updated, without introducing bugs or errors, can be a nightmare since you must ensure that you update the right DOM element with the right value at the right time. Below, you will find a screenshot of some example code for the described use-case.

> **NOTE**
>
> The full, working, code can be found on GitHub at https://packt.link/tLSLU.

If you take a look at the JavaScript code in the screenshot (or in the linked repository), you will probably be able to imagine how a more complex user interface is likely to look.



```
const emailInputElement = document.getElementById('email');
const passwordInputElement = document.getElementById('password');
const signupFormElement = document.querySelector('form');

let emailIsValid = false;
let passwordIsValid = false;

function validateEmail(enteredEmail) {
  // In reality, we might be sending the entered email address to a backend API to check if a user with that email exists already
  // Here, this is faked with help of a promise wrapper around some dummy validation logic

  const promise = new Promise(function (resolve, reject) {
    if (enteredEmail === 'test@test.com') {
      reject(new Error('Email exists already'));
    } else {
      resolve();
    }
  });

  return promise;
}

function validatePassword(enteredPassword) {
  if (enteredPassword.trim().length < 6) {
    throw new Error('Invalid password - must be at least 6 characters long.');
  }
}

async function validateInputHandler(inputType, event) {
  const targetElement = event.target;
  const enteredValue = targetElement.value;

  let validationFn = validateEmail;
  if (inputType === 'password') {
    validationFn = validatePassword;
  }

  const errorElement = document.getElementById(inputType + '-error');
  if (errorElement) {
    errorElement.remove();
  }

  let isValid = true;

  try {
    await validationFn(enteredValue);
  } catch (error) {
    const errorElement = document.createElement('p');
    errorElement.id = inputType + '-error';
    errorElement.textContent = error.message;
    targetElement.parentElement.append(errorElement);
    isValid = false;
  }

  if (inputType === 'email') {
    emailIsValid = isValid;
  } else {
    passwordIsValid = isValid;
  }
}

function submitFormHandler(event) {
  event.preventDefault();

  let title = 'An error occurred!';
  let message = 'Invalid input values - please check your entered values.';

  if (emailIsValid && passwordIsValid) {
    title = 'Success!';
    message = 'User created successfully!';
  }

  openModal(title, message);
}

function openModal(title, message) {
  const backdropElement = document.createElement('div');
  backdropElement.className = 'backdrop';

  const modalElement = document.createElement('aside');
  modalElement.className = 'modal';
  modalElement.innerHTML = `
    <header>
      <h2>${title}</h2>
    </header>
    <section>
      <p>${message}</p>
    </section>
    <section class="modal__actions">
      <button>Okay</button>
    </section>
  `;
  const confirmButtonElement = modalElement.querySelector('button');

  backdropElement.addEventListener('click', closeModal);
  confirmButtonElement.addEventListener('click', closeModal);

  document.body.append(backdropElement);
  document.body.append(modalElement);
}

function closeModal() {
  const modalElement = document.querySelector('.modal');
  const backdropElement = document.querySelector('.backdrop');

  modalElement.remove();
  backdropElement.remove();
}

emailInputElement.addEventListener(
  'blur',
  validateInputHandler.bind(null, 'email')
);
passwordInputElement.addEventListener(
  'blur',
  validateInputHandler.bind(null, 'password')
);

signupFormElement.addEventListener('submit', submitFormHandler);
```

Figure 1.1. An example JavaScript code file that contains over 100 lines of code for a fairly trivial user interface

This example JavaScript file already contains roughly 110 lines of code. Even after minifying ("minifying" means that code is shortened automatically, e.g. by replacing long variable names with shorter ones and removing redundant whitespace; in this case via https://javascript-minifier.com/) it and splitting the code across multiple lines thereafter (to count the raw lines of code), it still has around 80 lines of code. That's a full 80 lines of code for a simple user interface with only basic functionality. The actual business logic (i.e., input validation, determining if and when overlays should be shown, and defining the output text) only makes up a small fraction of the overall codebase—around 20 to 30 lines of code, in this case (around 20 after minifying).

That's roughly 75% of code spent on pure DOM interaction, DOM state management, and similar boilerplate tasks.

As you can see by these examples and numbers, controlling all the UI elements and their different states (e.g., whether an info box is visible or not) is a challenging task and trying to create such interfaces with just JavaScript often leads to complex code that might even contain errors.

That's why the imperative approach, wherein you must define and write down every single step, has its limits in situations like this. This is the reason why React provides utility functionalities that allow you to write code differently: with a declarative approach.

> **NOTE**
>
> This is not a scientific paper, and the preceding example is not meant to act as an exact scientific study. Depending on how you count lines and which kind of code you consider to be "core business logic", you will end up with higher or lower percentage values. The key message doesn't change though: Lots of code (in this case most of it) deals with the DOM and DOM manipulation—not with the actual logic that defines your website and its key features.

REACT AND DECLARATIVE CODE

Coming back to the first, simple, code snippet from above, here's that same code snippet, this time using React:

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

function App() {
  const [outputText, setOutputText] = useState('Initial text');

  function updateTextHandler() {
    setOutputText('Text was changed!');
  }

  return (
    <>
      <button onClick={updateTextHandler}>Click to change text</button>
      <p>{outputText}</p>
```

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

This snippet performs the same operations as the first did with just vanilla JavaScript:

1. Add an event listener to a button to listen for **`click`** events (now with some React-specific syntax: **`onClick={...})`**.
2. Replace the text of a paragraph with new text once the click on the button occurred.

Nonetheless, this code looks totally different—like a mixture of JavaScript and HTML. And indeed, React uses a syntax extension called **JSX** (i.e., JavaScript with embedded XML). For the moment, it's enough to understand that this JSX code will work because of a **pre-processing** step that's part of the build workflow of every React project.

Pre-processing means that certain tools, which are part of React projects, analyze and transform the code before its deployed. This allows for development-only syntax like JSX which would not work in the browser and is therefore transformed to regular JavaScript before deployment. (You'll get a thorough introduction into JSX in *Chapter 2*, *Understanding React Components and JSX*.)

In addition, the snippet shown above contains a React specific feature: State. State will be discussed in greater detail later in the book (*Chapter 4*, *Working with Events and State* will focus on handling events and state with React). For the moment, you can think of this state as a variable that, when changed, will trigger React to update the user interface in the browser.

What you see in the preceding example is the "declarative approach" used by React: You write your JavaScript logic (e.g., functions that should eventually be executed), and you combine that logic with the HTML code that should trigger it or that is affected by it. You don't write the instructions for selecting certain DOM elements or changing the text content of some DOM elements. Instead, with React and JSX, you focus on your JavaScript business logic and define the desired HTML output that should eventually be reached. This output can and typically will contain dynamic values that are derived inside of your main JavaScript code.

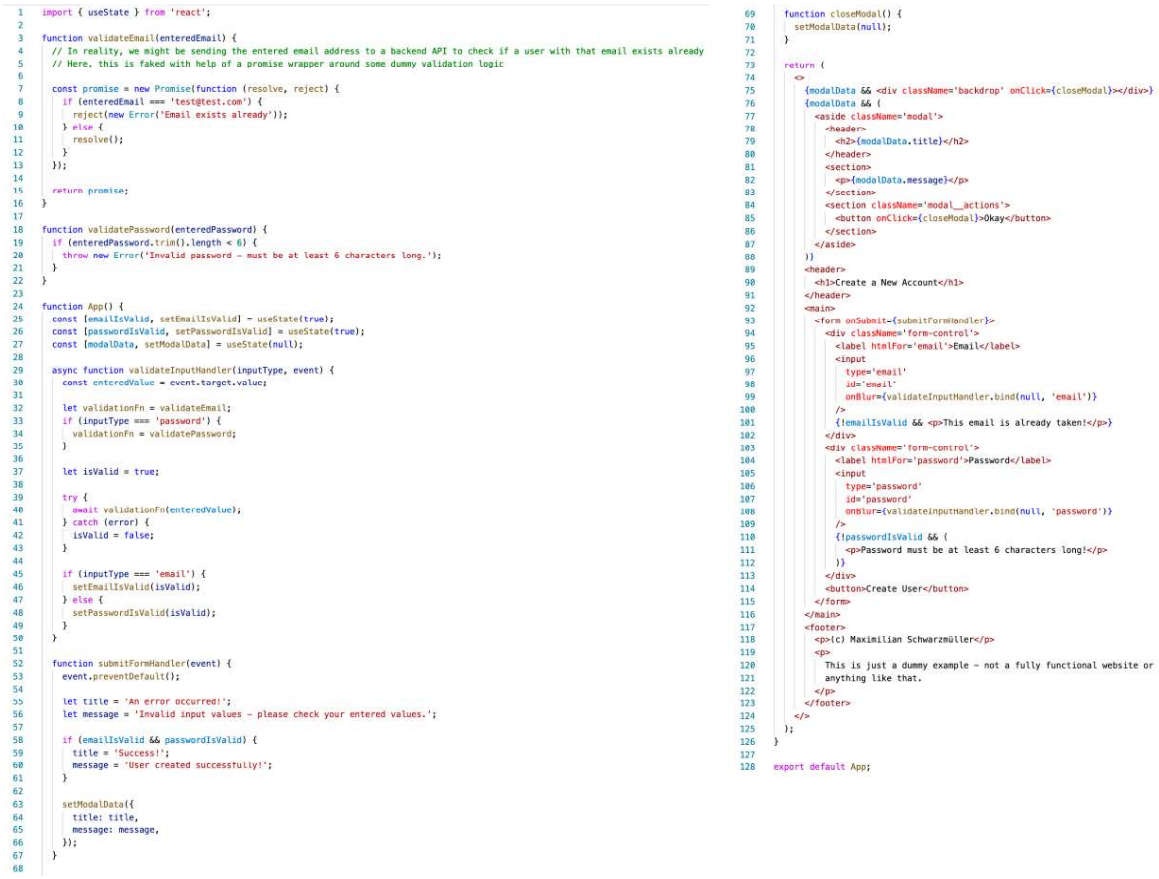
In the preceding example, **`outputText`** is some state managed by React. In the code, the **`updateTextHandler`** function is triggered upon a click, and the **`outputText`** state value is set to a new string value (**`'Text was changed!'`**) with help of the **`setOutputText`** function. The exact details of what's going on here will be explored in *Chapter 4*.

The general idea, though, is that the state value is changed and, since it's being referenced in the last paragraph (**`<p>{outputText}</p>`**), React outputs the current state value in that place in the actual DOM (and therefore on the actual web page). React will keep the paragraph updated, and therefore, whenever **`outputText`** changes, React will select this paragraph element again and update its **`textContent`** automatically.

This is the declarative approach in action. As a developer, you don't need to worry about the technical details (for example, selecting the paragraph, updating its **`textContent`**). Instead, you will hand this work off to React. You will only need to focus on the desired end state(s) where the goal simply is to output the current value of **`outputText`** in a specific place (i.e., in the second paragraph in this case) on the page. It's React's job of doing the "*behind the scenes*" work of getting to that result.

It turns out that this code snippet isn't shorter than the vanilla JavaScript one; indeed, it's actually even a bit longer. But that's only the case because this first snippet was deliberately kept simple and concise. In such cases, React actually adds a bit of overhead code. If that were your entire user interface, using React indeed wouldn't make too much sense. Again, this snippet was chosen because it allows us to see the differences at a glance. Things change if you take a look at the more complex vanilla JavaScript example from before) and compare that to its React alternative.

> **NOTE**
>
> Referenced code can be found on GitHub at http://packt.link/tLSLU and https://packt.link/YkpRa, respectively.



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

function validateEmail(enteredEmail) {
  // In reality, we might be sending the entered email address to a backend API to check if a user with that email exists already
  // Here, this is faked with help of a promise wrapper around some dummy validation logic

  const promise = new Promise(function (resolve, reject) {
    if (enteredEmail === 'test@test.com') {
      reject(new Error('Email exists already'));
    } else {
      resolve();
    }
  });

  return promise;
}

function validatePassword(enteredPassword) {
  if (enteredPassword.trim().length < 6) {
    throw new Error('Invalid password - must be at least 6 characters long.');
  }
}

function App() {
  const [emailIsValid, setEmailIsValid] = useState(true);
  const [passwordIsValid, setPasswordIsValid] = useState(true);
  const [modalData, setModalData] = useState(null);

  async function validateInputHandler(inputType, event) {
    const enteredValue = event.target.value;

    let validationFn = validateEmail;
    if (inputType === 'password') {
      validationFn = validatePassword;
    }

    let isValid = true;

    try {
      await validationFn(enteredValue);
    } catch (error) {
      isValid = false;
    }

    if (inputType === 'email') {
      setEmailIsValid(isValid);
    } else {
      setPasswordIsValid(isValid);
    }
  }

  function submitFormHandler(event) {
    event.preventDefault();

    let title = 'An error occurred!';
    let message = 'Invalid input values - please check your entered values.';

    if (emailIsValid && passwordIsValid) {
      title = 'Success!';
      message = 'User created successfully!';
    }

    setModalData({
      title: title,
      message: message,
    });
  }

  function closeModal() {
    setModalData(null);
  }

  return (
    <>
      {modalData && <div className='backdrop' onClick={closeModal}></div>}
      {modalData && (
        <aside className='modal'>
          <header>
            <h2>{modalData.title}</h2>
          </header>
          <section>
            <p>{modalData.message}</p>
          </section>
          <section className='modal__actions'>
            <button onClick={closeModal}>Okay</button>
          </section>
        </aside>
      )}
      <header>
        <h1>Create a New Account</h1>
      </header>
      <main>
        <form onSubmit={submitFormHandler}>
          <div className='form-control'>
            <label htmlFor='email'>Email</label>
            <input
              type='email'
              id='email'
              onBlur={validateInputHandler.bind(null, 'email')}
            />
            {!emailIsValid && <p>This email is already taken!</p>}
          </div>
          <div className='form-control'>
            <label htmlFor='password'>Password</label>
            <input
              type='password'
              id='password'
              onBlur={validateInputHandler.bind(null, 'password')}
            />
            {!passwordIsValid && (
              <p>Password must be at least 6 characters long!</p>
            )}
          </div>
          <button>Create User</button>
        </form>
      </main>
      <footer>
        <p>(c) Maximilian Schwarzmüller</p>
        <p>
          This is just a dummy example - not a fully functional website or
          anything like that.
        </p>
      </footer>
    </>
  );
}

export default App;
```

Figure 1.2. The code snippet from before, now implemented via React.

It's still not short because all the JSX code (i.e., the HTML output) is included in the JavaScript file. If you ignore pretty much the entire right side of that screenshot (since HTML was not part of the vanilla JavaScript files either), the React code gets much more concise. But, most importantly, if you take a closer look at all the React code (also in the first, shorter snippet), you will notice that there are absolutely no operations that would select DOM elements, create or insert DOM elements, or edit DOM elements.

And this is the core idea of React. You don't write down all the individual steps and instructions; instead, you focus on the "big picture" and the desired end state(s) of your page content. With React, you can merge your JavaScript and markup code without having to deal with the low-level instructions of interacting with the DOM like selecting elements via **`document.getElementById()`** or similar operations.

Using this declarative approach, instead of the imperative approach with vanilla JavaScript, allows you, the developer, to focus on your core business logic and the different states of your HTML code. You don't need to define all the individual steps that have to be taken (like "adding an event listener", "selecting a paragraph", etc.), and this simplifies the development of complex user interfaces tremendously.

> **NOTE**
>
> It is worth emphasizing that React is not a great solution if you're working on a very simple user interface. If you can solve a problem with a few lines of vanilla JavaScript code, there is probably no strong reason to integrate React into the project.

Looking at React code for the first time, it can look very unfamiliar and strange. It's not what you're used to from JavaScript. Still, it is JavaScript—just enhanced with this JSX feature and various React-specific functionalities (like State). It may be less confusing if you remember that you typically define your user interface (i.e., your content and its structure) with HTML. You don't write step-by-step instructions there either, but rather create a nested tree structure with HTML tags. You express your content, the meaning of different elements, and the hierarchy of your user interface by using different HTML elements and by nesting HTML tags.

If you keep this in mind, the "traditional" (vanilla JavaScript) approach of manipulating the UI should seem rather odd. Why would you start defining low-level instructions like *"insert a paragraph element below this button and set its text to <some text>"* if you don't do that in HTML at all? React in the end brings back that HTML syntax, which is far more convenient when it comes to defining content and structure. With React, you can write dynamic JavaScript code side-by-side with the UI code (i.e., the HTML code) that is affected by it or related to it.

HOW REACT MANIPULATES THE DOM

As mentioned earlier, when writing React code, you typically write it as shown above: You blend HTML with JavaScript code by using the JSX syntax extension.

It is worth pointing out that JSX code does not run like this in browsers. It instead needs to be pre-processed before deployment. The JSX code must be transformed to regular JavaScript code before being served to browsers. The next chapter will take a closer look at JSX and what it's transformed to. For the moment, though, simply keep in mind that JSX code must be transformed.

Nonetheless, it is worth knowing that the code to which JSX will be transformed will also not contain any DOM instructions. Instead, the transformed code will execute various utility methods and functions that are built-into React (in other words, those that are provided by the React package that needs to be added to every React project). Internally, React creates a virtual DOM-like tree structure that reflects the current state of the user interface. This book takes a closer look at this abstract, virtual DOM and how React works in *Chapter 9*, *Behind the Scenes of React and Optimization Opportunities*. Therefore, React (the library) splits its core logic across two main packages:

- The main **`react`** package
- And the **`react-dom`** package

The main react package is a third-party JavaScript library that needs to be imported into a project to use React's features (like JSX or state) there. It's this package that creates this virtual DOM and derives the current UI state. But you also need the **`react-dom`** package in your project if you want to manipulate the DOM with React.

The **`react-dom`** package, specifically the **`react-dom/client`** part of that package, acts as a "translation bridge" between your React code, the internally generated virtual DOM, and the browser with its actual DOM that needs to be updated. It's the **`react-dom`** package that will produce the actual DOM instructions that will select, update, delete, and create DOM elements.

This split exists because you can also use React with other target environments. A very popular and well-known alternative to the DOM (i.e., to the browser) would be React Native, which allows developers to build native mobile apps with help of React. With React Native, you also include the react package into your project, but in place of **`react-dom`**, you would use the **`react-native`** package. In this book, "React" refers to both the react package and the "bridge" packages (like **`react-dom)`**.

> **NOTE**
>
> As mentioned earlier, this book focuses on React itself. The concepts explained in this book, therefore, will apply to both web browsers and websites as well as mobile devices. Nonetheless, all examples will focus on the web and react-DOM since that avoids introducing extra complexity.

INTRODUCING SINGLE PAGE APPLICATIONS

React can be used to simplify the creation of complex user interfaces, and there are two main ways of doing that:

1. Manage parts of a website (e.g., a chat box in the bottom left corner).
2. Manage the entire page and all user interaction that occurs on it.

Both approaches are viable, but the more popular and common scenario is the second one: Using React to manage the entire web page, instead of just parts of it. This approach is more popular because most websites that have complex user interfaces, have not just one complex element but multiple elements on their pages. Complexity would actually increase if you were to start using React for some website parts without using it for other areas of the site. For this reason, it's very common to manage the entire website with React.

This doesn't even stop after using React on one specific page of the site. Indeed, React can be used to handle URL path changes and update the parts of the page that need to be updated in order to reflect the new page that should be loaded. This functionality is called "routing" and third-party packages like **`react-router-dom`** (see *Chapter 12, Multipage Apps with React Router*), which integrate with React, allow you to create a website wherein the entire user interfaces is controlled via React.

A website that does not just use React for parts of its pages but instead for all subpages and for routing is called a **`"Single Page Application" (SPA)`** because it consists of only one HTML file (typically named **`index.html`**) which is used to initially load the React JavaScript code. Thereafter, the React library and your React code take over and control the actual user interface. This means that the entire user interface is created and managed by JavaScript via React and your React code.

CREATING A REACT PROJECT

To work with React, the first step is the creation of a React project. This can be done in multiple ways, but the most straightforward and easiest way is to use the **`create-react-app`** utility command line tool. This is a tool maintained by (parts of) the React team, and you can install it as a **Node.js** package via the **Node Package Manager** (**npm**). Once installed, this tool can be used to create a project folder that comes with React pre-installed, as well as some other tools, such as the **Jest** package for automated testing.

You need a project setup like this because you typically use features like JSX which wouldn't work in the browser without prior code transformation. Therefore, as mentioned earlier, a pre-processing step is required, and the React project created via **`create-react-app`** includes such a step as part of the code build workflow.

To create a project with **`create-react-app`**, you must have Node.js installed—preferably the latest (or latest **LTS**) version. You can get the official Node.js installer for all operating systems from https://nodejs.org/. Once you have installed Node.js, you will also gain access to the built-in **`npm`** and **`npx`** commands, which you can use to utilize the **`create-react-app`** package to create a new project.

You can run the following command inside of your command prompt (Windows), bash (Linux), or terminal (macOS) program. Just make sure that you navigated (via **`cd`**) into the folder in which you want to create your new project.

```
npx create-react-app my-react-project
```

This command will create a new subfolder with a basic React project setup (i.e., with various files and folders) in the place where you ran it. You should run it in some path on your system where you have full read and write access and where you're not conflicting with any system or other project files.

The exact project structure (that is, the file names and folder names) may vary over time, but generally, every new React project contains a couple of key files and folders:

- A **`src/`** folder that contains the main source code files for the project:
- An **`index.js`** file which is the main entry script file that will be executed first
- An **`App.js`** file which contains the root component of the application (you'll learn more about components in the next chapter)
- Various styling (**`*.css`**) files that are imported by the JavaScript files
- Other files, like code files for automated tests
- A **`public/`** folder which contains static files that will be part of the final website:
- This folder may contain static images like favicons
- The folder also contains an **`index.html`** file which is the single HTML page of this website

- **`package.json`** and **`package-lock.json`** are files that manage third-party dependencies of your project
- Production dependencies like **`react`** or **`react-dom`**
- Development dependencies like **`jest`** for automated tests

> **NOTE**
>
> **`package.json`** is the file in which you actually manage packages and their versions. **`package-lock.json`** is created automatically (by **`Node.js`**). It locks in exact dependency and sub-dependency versions, whereas **`package.json`** only specifies version ranges. You can learn more about these files and package versions on https://docs.npmjs.com/.

- The **`node_modules`** folder holds the actual third-party package code of the packages that are listed in the **`package.json`** file. This **`node_modules`** folder can be deleted since you can recreate it by running **`npm`** install inside of the project folder

Most of the React code will be written in the **`App.js`** file or custom components that will be added to the project. This book will explore components in the next chapter.

> **NOTE**
>
> The **`node_modules`** folder can become very big since it contains all projects dependencies and dependencies of dependencies. Therefore, it's typically not included if projects are shared with other developers or pushed to GitHub. The **`package.json`** file is all you need. By running **`npm install`**, the **`node_modules`** folder will be recreated locally.

Once the project is created, you can start writing your code. To preview your code on a live website locally on your system, you can **`run npm`** start inside of the project folder. This will start a built-in development server that pre-processes, builds, and hosts your React-powered SPA. This process should normally open the preview page in a new browser tab automatically. If that doesn't happen, you can manually open a new tab and navigate to **`localhost:3000`** there (unless you see a different address as output in the window where you executed **`npm start`**, in which case, use the address that's shown after you ran **`npm start`**).

The preview website that opens up will automatically reload and reflect code changes whenever you save changes to your code.

When you're done with development for the day, you can quit the running development server process via **`CTRL + C`** (in the command prompt or terminal window where you started it via **`npm start`**). To continue development and get back that live preview website, you can always restart it by running **`n`**pm start (inside of the project folder) again.

SUMMARY AND KEY TAKEAWAYS

- React is a library, though it's actually a combination of two main packages: **`react`** and **`react-dom`**.
- Though it is possible to build non-trivial user interfaces without React, simply using vanilla JavaScript to do so can be cumbersome, error-prone, and hard to maintain.
- React simplifies the creation of complex user interfaces by providing a declarative way to define the desired end state(s) of the UI.
- **Declarative** means that you define the target user interface content and structure, combined with different states (e.g., *"is a modal open or closed?"*), and you leave it up to React to figure out the appropriate DOM instructions.
- The react package itself derives UI states and manages a virtual DOM. It's "bridges" like **`react-dom`** or **`react-native`** that translate this virtual DOM into actual UI (DOM) instructions.
- With React, you can build Single Page Applications (SPAs), meaning that React is used to control the entire user interface on all pages as well as the routing between pages.
- React projects can be created with help of the **`create-react-app`** package, which provides a readily configured project folder and a live preview development server.

WHAT'S NEXT?

At this point, you should have a basic understanding of what React is and why you might consider using it, especially for building non-trivial user interfaces. You learned how to create new React projects with create-react-app, and you are now ready to dive deeper into React and the actual key features it offers.

In the next chapter, you will learn about a concept called **components** which are the fundamental building blocks of React apps. You will learn how components are used to compose user interfaces and why those components are needed in the first place. The next chapter will also dive deeper into JSX and explore how it is transformed to regular JavaScript code and which kind of code you could write alternatively to JSX.

TEST YOUR KNOWLEDGE!

Test your knowledge about the concepts covered in this chapter by answering the below questions. You can then compare your answers to example answers that can be found here: https://packt.link/ENPda.

1. What is React?
2. Which advantage does React offer over vanilla JavaScript projects?
3. What's the difference between imperative and declarative code?
4. What is a Single-Page-Application (SPA)?
5. How can you create new React projects and why do you need such a more complex project setup?

2

UNDERSTANDING REACT COMPONENTS AND JSX

LEARNING OBJECTIVES

By the end of this chapter, you will be able to do the following:

- Define what exactly **components** are
- Build and use components effectively
- Utilize common naming conventions and code patterns
- Describe the relation between components and JSX
- Write JSX code and understand why it's used
- Write React components without using JSX code
- Write your first React apps

INTRODUCTION

In the previous section, you learned about React in general, what it is and why you could consider using it for building user interfaces. You also learned how to create React projects with the help of **`npx create-react-app`**.

In this chapter, you will learn about one of the most important React concepts and building blocks: React as above, components. You will learn that components are reusable building blocks which are used to build user interfaces. In addition, JSX code will be discussed in greater detail so that you will be able to use the concept of components and JSX to build your own, first, basic React apps.

WHAT ARE COMPONENTS?

A key concept of React is the usage of so-called components. **Components** are reusable building blocks which are combined to compose the final user interface. For example, a basic website could be made up of a header that includes a navigation bar and a main section that includes an authentication form.

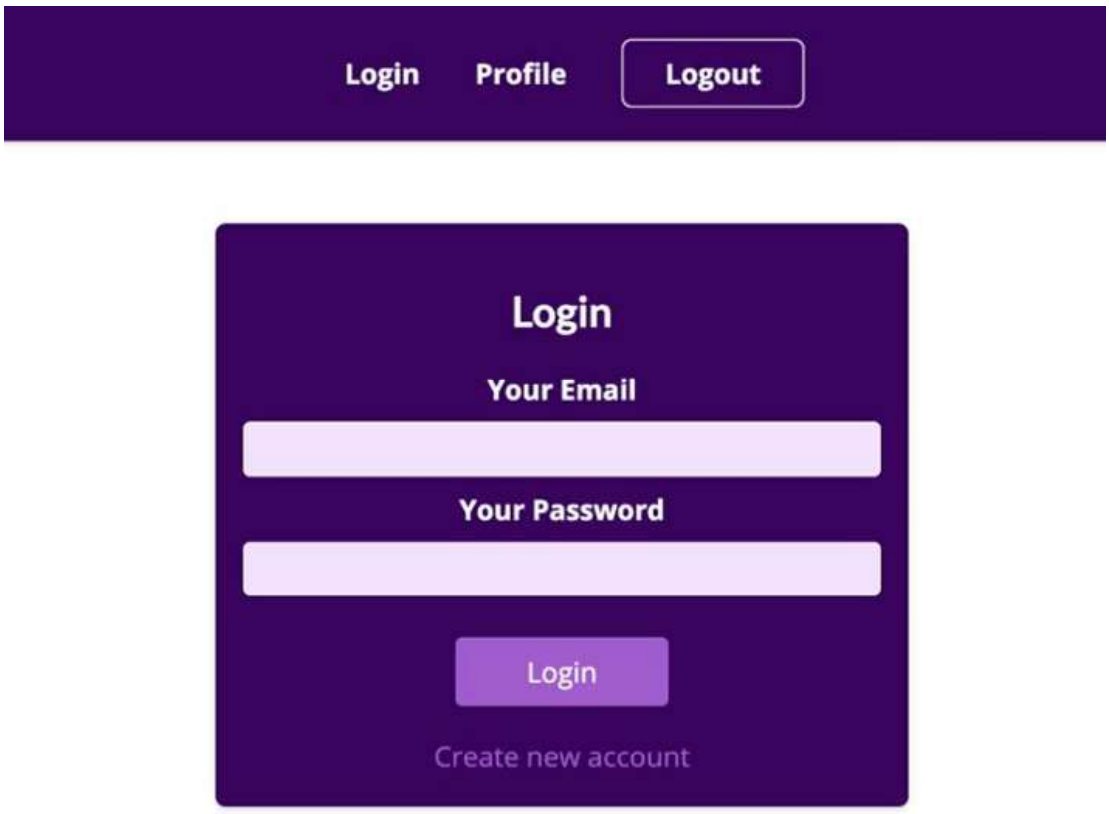


Figure 2.1 An example authentication screen with navigation bar.

If you look at this example page, you might be able to identify various building blocks (i.e., components). Some of these components are even reused.

In the header with the navigation bar you will find the following components:

- The navigation items (**`Login`** and **`Profile`**)
- The **`Logout`** button

Below this, the main section displays the following:

- The container that contains the authentication form
- The input elements
- The confirmation button
- A link to switch to the **`New Account`** page

Please note that some components are nested inside other components—i.e., components are also made up of other components. That's a key feature of React and similar libraries.

WHY COMPONENTS?

No matter which web page you look at, they are all made up of building blocks like this. It's not a React-specific concept or idea. Indeed, HTML itself "thinks" in components if you take a closer look. You have elements like **`<img>`**, **`<header>`**, **`<nav>`**, etc. And you combine these elements to describe and structure your website content.

But React **embraces** this idea of breaking a web page into reusable building blocks because this is an approach that allows developers to work on small, manageable chunks of code. It's easier and more maintainable than working on a single, huge HTML (or React code) file.

That's why other libraries—both frontend libraries like React or Angular as well as backend libraries and templating engines like **EJS** (**Embedded JavaScript templates**)—also embrace components (though the names might differ, you also find *"partials"* or *"includes"* as common names).

> **NOTE**
>
> EJS is a popular templating engine for JavaScript. It's especially popular for backend web development with NodeJS.

When working with React, it's especially important to keep your code manageable and work with small, reusable components because React components are not just collections of HTML code. Instead, a React component also encapsulates JavaScript logic and often also CSS styling. For complex user interfaces, the combination of markup (JSX), logic (JavaScript) and styling (CSS) could quickly lead to large chunks of code, thus making it difficult to maintain that code. Think of a large HTML file that also includes JavaScript and CSS code. Working in such a code file wouldn't be a lot of fun.

To make a long story short, when working in a React project, you will work with lots of components. You will split your code into small, manageable building blocks and then combine these components to form the overall user interface. It's a key feature of React.

> **NOTE**
>
> When working with React, you should embrace this idea of working with components. But technically, they're optional. You could, theoretically, build very complex web pages with one single component alone. It would not be much fun, and it would not be practical, but it would technically be possible without any issues.

THE ANATOMY OF A COMPONENT

Components are important. But what exactly does a React component look like? How do you write React components on your own?

Here's an example component:

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

function SubmitButton() {
  const [isSubmitted, setIsSubmitted] = useState(false);

  function submitHandler() {
    setIsSubmitted(true);
  };

  return (
    <button onClick={submitHandler}>
      { isSubmitted ? 'Loading…' : 'Submit' }
```

```
    </button>
  );
};

export default SubmitButton;
```

Typically, you would store a code snippet like this in a separate file (e.g., a file named **`SubmitButton.js`**, stored inside a **`/components`** folder which in turn resides in the **`/src`** folder of your React project) and import it into other component files that need this component. For example, the following component imports the component defined above and uses it in its **`return`** statement to output the **`SubmitButton`** component:

```
import SubmitButton from './submit-button';

function AuthForm() {
  return (
    <form>
      <input type="text" />
      <SubmitButton />
    </form>
  );
};

export default AuthForm;
```

The import statements you see in these examples are standard JavaScript import statements with one extra twist: the file extension (**`.js`** in this case) can be omitted in most React projects (like the one created via **`npx create-react-app`**). **`import`** and **`export`** are standard JavaScript keywords that help with splitting related code across multiple files. Things like variables, constants, classes, or functions can be exported via **`export`** or **`export default`** so that they can then be used in other files after **`import`**ing them there.

> **NOTE**
>
> If the concept of splitting code into multiple files and using **`import`** and **`export`** is brand-new to you, you might want to dive into more basic JavaScript resources on this topic first. For example, MDN has an excellent article that explains the fundamentals, which you can find at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules.

Of course, the components shown in these examples are highly simplified and also contain features that you haven't learned about yet (e.g. **`useState()`**). But the general idea of having standalone building blocks that can be combined should be clear.

When working with React, there are two alternative ways to define components:

Class-based components (or "**class components**"): components defined via the **`class`** keyword

Functional components (or "**function components**"): components that are defined via regular JavaScript functions

In all the examples covered in this book thus far, components were built as JavaScript functions. As a React developer, you have to use one of these two approaches as React expects components to be functions or classes.

> **NOTE**
>
> Until late 2018, you had to use class-based components for certain kinds of tasks—specifically for components that use state internally. (State will be covered later in the book). However, in late 2018, a new concept was introduced: **React Hooks**. This allows you to perform all operations and tasks with functional components. Consequently, class-based components are on their way out and not covered in this book.

In the examples above, there are a couple of other noteworthy things:

- The component functions carry capitalized names (e.g., **`SubmitButton`**)
- Inside the component functions, other "inner" functions can be defined (e.g., **`submitHandler`**)
- The component functions return *HTML-like* code (JSX code)
- Features like **`useState()`** can be used inside the component functions
- The component functions are exported (via **`export default`**)
- Certain features (like **`useState`** or the custom component, **`SubmitButton`**) are imported via the **`import`** keyword

The following sections will take a closer look at these different concepts that make up components and their code.

WHAT EXACTLY ARE COMPONENT FUNCTIONS?

In React, components are functions (or classes, but as mentioned above, those aren't relevant anymore).

A function is a regular JavaScript construct, not a React-specific concept. This is important to note. React is a JavaScript library and therefore **uses JavaScript features** (like functions); React is **not a brand-new programming language**.

When working with React, regular JavaScript functions can be used to encapsulate HTML (or, to be more precise, JSX) code and JavaScript logic that belongs to that markup code. However, it depends on the code you write in a function, whether it qualifies to be treated as a React component or not. For example, in the code snippets above, the **`submitHandler`** function is also a regular JavaScript function, but it's not a React component. The following example shows another regular JavaScript function that doesn't qualify as a React component:

```
function calculate(a, b) {
  return {sum: a + b};
};
```

Indeed, a function will be treated as a component and can therefore be used like a HTML element in JSX code if it returns a **renderable** value (typically JSX code). This is very important. You can only use a function as a React component in JSX code if it is a function that returns something that can be rendered by React. The returned value technically doesn't have to be JSX code, but in most cases, it will be. You will see an example for non-JSX code being returned later in the book, in *Chapter 7*, *Portals and Refs*.

In the code snippet where functions named **`SubmitButton`** and **`AuthForm`** were defined, those two functions qualified as React components because they both returned JSX code (which is code that can be rendered by React, making it renderable). Once a function qualifies as a React component, it can be used like a HTML element inside of JSX code, just as **`<SubmitButton />`** was used like a (self-closing) HTML element.

When working with vanilla JavaScript, you of course typically call functions to execute them. With functional components, that's different. React calls these functions on your behalf, and therefore, as a developer, you use them like HTML elements inside of this JSX code.

> **NOTE**
>
> When referring to renderable values, it is worth noting that by far the most common value type being returned or used is indeed JSX code—i.e., markup defined via JSX. This should make sense because, with JSX, you can define the HTML-like structure of your content and user interface.
>
> But besides JSX markup, there are a couple of other key values that also qualify as renderable and therefore could be returned by custom components (instead of JSX code). Most notably, you can also return strings or numbers as well as arrays that hold JSX elements or strings or numbers.

WHAT DOES REACT DO WITH ALL THESE COMPONENTS?

If you follow the trail of all components and their **`import + export`** statements to the top, you will find a **`root.render(...)`** instruction in the main entry script of the React project. Typically, this main entry script can be found in the index.js file, located in the project's **`src/`** folder. This **`render()`** method, which is provided by the React library (to be precise, by the **`react-dom`** package), takes a snippet of JSX code and interprets and executes it for you.

The complete snippet you find in the root entry file (**`index.js`**) typically looks like this:

```
import React from 'react';
import ReactDOM from 'react-dom/client';

import './index.css';
import App from './App';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<App />);
```

The exact code you find in your new React project might look slightly different.

It may, for instance, include an extra **<StrictMode>** element that's wrapped around **<App>**. **<StrictMode>** turns on extra checks that can help catch subtle bugs in your React code. But it can also lead to confusing behavior and unexpected error messages, especially when experimenting with React or learning React. As this book is primarily interested in the coverage of React core features and key concepts, **<StrictMode>** will not be covered.

To follow along smoothly then, cleaning up a newly created **index.js** file to look like the code snippet above is a good idea.

The **createRoot()** method instructs React to create a new **entry point** which will be used to inject the generated user interface into the actual HTML document that will be served to website visitors. The argument passed to **createRoot()** therefore is a pointer to a DOM element that can be found in **index.html**—the single page that will be served to website visitors.

In many cases, **document.getElementById('root')** is used as an argument. This built-in vanilla JavaScript method yields a reference to a DOM element that is already part of the **index.html** document. Therefore, as a developer, you must ensure that such an element with the provided **id** attribute value (**root**, in this example) exists in the HTML file into which the React app script is loaded. In a default React project created via **npx create-react-app**, this will be the case. You can find a ** ** element in the **index.html** file in the **public/** folder.

This **index.html** file is a relatively empty file which only acts as a shell for the React app. React just needs an entry point (defined via **createRoot()**) which will be used to attach the generated user interface to the displayed website. The HTML file and its content therefore do not directly define the website content. Instead, the file just serves as a starting point for the React application, allowing React to then take over and control the actual user interface.

Once the root entry point has been defined, a method called **render()** can be called on the root object created via **createRoot()**:

```
root.render(<App />);
```

This **render()** method tells React which content (i.e., which React component) should be injected into that root entry point. In most React apps, this is a component called **App**. React will then generate appropriate DOM-manipulating instructions to reflect the markup defined via JSX in the **App** component on the actual webpage.

This **App** component is a component function that is imported from some other file. In a default React project, the **App** component function is defined and exported in an **App.js** file which is also located in the **src/** folder.

This component, which is handed to **`render()`** (**`<App />`**, typically), is also called the **root component** of the React app. It's the main component that is rendered to the DOM. All other components are nested in the JSX code of that **`App`** component or the JSX code of even more nested descendent components. You can think of all these components building up a tree of components which is evaluated by React and translated into actual DOM-manipulating instructions.

> **NOTE**
>
> As mentioned in the previous chapter, React can be used on various platforms. With the **`react-native`** package, it could be used to build native mobile apps for iOS and Android. The **`react-dom`** package which provides the **`createRoot()`** method (and therefore implicitly the **`render()`** method) is focused on the browser. It provides the "bridge" between React's capabilities and the browser instructions that are required to bring the UI (described via JSX and React components) to life in the browser. If you would build for different platforms, replacements for **`ReactDOM.createRoot()`** and **`render()`** are required (and of course such alternatives do exist).

Either way, no matter whether you use a component function like an HTML element inside of JSX code of other components or use it like an HTML element that's passed as an argument to the **`render()`** method, React takes care of interpreting and executing the component function on your behalf.

Of course, this is not a new concept. In JavaScript, functions are first-class objects, which means that you can pass functions as arguments to other functions. This is basically what happens here, just with the extra twist of using this JSX syntax which is not a default JavaScript feature.

React executes these component functions for you and translates the returned JSX code into DOM instructions. To be precise, React traverses the returned JSX code and dives into any other custom components that might be used in that JSX code until it ends up with JSX code that is only made up of native, built-in HTML elements (technically, it's not really HTML, but that will be discussed later in this chapter).

Take these two components as an example:

```
function Greeting() {
  return <p>Welcome to this book!</p>;
};

function App() {
  return (
    <div>
      <h2>Hello World!</h2>
      <Greeting />
    </div>
  );
};

const root = ReactDOM.createRoot(document.getElementById('app'));
root.render(<App />);
```

The **`App`** component uses the **`Greeting`** component inside its JSX code. React will traverse the entire JSX markup structure and derive this final JSX code:

```
root.render((
  <div>
    <h2>Hello World!</h2>
    <p>Welcome to this book!</p>
  </div>
), document.getElementById('app'));
```

And this code would instruct React and ReactDOM to perform the following DOM operations:

- Create a **`<div>`** element
- Inside that **`<div>`**, create two child elements: **`<h2>`** and **`<p>`**
- Set the text content of the **`<h2>`** element to **`'Hello World!'`**
- Set the text content of the **`<p>`** element to **`'Welcome to this book!'`**
- Insert the **`<div>`** with its children into the already-existing DOM element which has the **`id 'app'`**

This is a bit simplified, but you can think of React handling components and JSX code as described above.

> **NOTE**
>
> React doesn't actually work with JSX code internally. It's just easier to use as a developer. Later, in this chapter, you will learn what JSX code gets transformed to and how the actual code, with which React works, looks like.

BUILT-IN COMPONENTS

As shown in the earlier examples, you can create your own, custom components by creating functions that return JSX code. And indeed, that's one of the main things you will do all the time as a React developer: you create component functions. Lots of component functions.

But ultimately, if you would merge all JSX code into just one big snippet of JSX code, as shown in the last example above, you would end up with a chunk of JSX code that includes only standard HTML elements like **`<div>`**, **`<h2>`**, **`<p>`**, and so on.

When using React, you don't create brand-new HTML elements that the browser would be able to display and handle. Instead, you create components that **only work inside the React environment**. Before they reach the browser, they have been evaluated by React and "translated" into DOM-manipulating JavaScript instructions (like **`document.append(...)`**).

But keep in mind that all this JSX code is a feature that's not part of the JavaScript language itself. It's basically **syntactical sugar** (i.e., a simplification regarding the code syntax) provided by the React library and the project setup you're using to write React code. Therefore, elements like **`<div>`**, when used in JSX code, also **aren't normal HTML elements** because you **don't write HTML code**. It might look like that, but it's inside a **`.js`** file and it's not HTML markup. Instead, it's this special JSX code. It is important to keep this in mind.

Therefore, these **`<div>`** and **`<h2>`** elements you see in all these examples are also just React components in the end. But they are not components built by you, but instead provided by React (or to be precise, by ReactDOM).

When working with React, you therefore always end up with these primitives—these built-in component functions that are later translated to browser instructions that generate and append or remove normal DOM elements. The idea behind building custom components is to group these elements together such that you end up with reusable building blocks that can be used to build the overall UI. But, in the end, this UI is made up of regular HTML elements.

> **NOTE**
>
> Depending on your level of frontend web development knowledge, you might have heard about a web feature called **Web Components**. The idea behind this feature is that you can indeed build brand-new HTML elements with vanilla JavaScript.
>
> As mentioned, React does not pick up this feature; you don't build new custom HTML elements with React.

NAMING CONVENTIONS

All component functions that you can find in this book carry names like **`SubmitButton`**, **`AuthForm`**, or **`Greeting`**.

You can generally name your React functions however you want—at least in the file where you are defining them. But it is a common convention to use the **PascalCase** naming convention, wherein the first character is uppercase and multiple words are grouped into one single word (**`SubmitButton`** instead of **`Submit Button`**), where every "subword" then starts with another uppercase character.

In the place where you define your component function, it is only a naming convention, not a hard rule. However, it **is** a requirement in the place where you **use** the component functions—i.e., in the JSX code where you embed your own custom components.

You can't use your own custom component like this:

```
<greeting />
```

React forces you to use an uppercase starting character for your own custom component names, when using them in JSX code. This rule exists to give React a clear and easy way of telling custom components apart from built-in components like **`<div>`** etc. React only needs to look at the starting character to determine whether it's a built-in element or a custom component.

Besides the names of the actual component functions, it is also important to understand file naming conventions. Custom components are typically stored in separate files that live inside a **`src/components/`** folder. However, this is not a hard rule. The exact placement as well as folder name is up to you, but it should be somewhere inside the **`src/`** folder. Using a folder named **`components/`** is the standard though.

Where it is the standard to use **`PascalCase`** for the component functions, there is no general default regarding the file names. Some developers prefer **`PascalCase`** for file names as well; and, indeed, in brand-new React projects, created as described in this book, the **`App`** component can be found inside a file named **`App.js`**. Nonetheless, you will also encounter many React projects where components are stored in files that follow the **`kebap-case`** naming convention. (All-lowercase, multiple words are combined into a single word via a dash.) With this convention, component functions could be stored in files named **`submit-button.js`**, for example.

Ultimately, it is up to you (and your team) which file naming convention you want to follow. In this book, **`PascalCase`** will be used for file names.

JSX VS HTML VS VANILLA JAVASCRIPT

As mentioned above, React projects typically contain lots of JSX code. Most custom components will return JSX code snippets. You can see this in all the examples shared thus far, and you will see in basically every React project you will explore, no matter whether you are using React for the browser or for other platforms like **`react-native`**.

But what exactly is this JSX code? How is it different from HTML? And how is it related to vanilla JavaScript?

JSX is a feature that's not part of vanilla JavaScript. What can be confusing, though, is that it's also not directly part of the React library.

Instead, JSX is syntactical sugar that is provided by the build workflow that's part of the overall React project. When you start the development web server via **`npm start`** or build the React app for production (i.e., for deployment) via **`npm run build`**, you kick off a process that transforms this JSX code back to regular JavaScript instructions. As a developer, you don't see those final instructions but React, the library, actually receives and evaluates them.

So, what does the JSX code get transformed to?

In the end, all JSX snippets get transformed into calls to the **`React.createElement(...)`** method.

Here's a concrete example:

```
function Greeting() {
  return <p>Hello World!</p>;
};
```

The JSX code returned by this component would be translated into the following vanilla JavaScript code:

```
function Greeting() {
  return React.createElement('p', {}, 'Hello World!');
};
```

`createElement()` is a method built into the React library. It instructs React to create a paragraph element with **`'Hello World!'`** as child content (i.e., as inner, nested content). This paragraph element is then created internally first (via a concept called **virtual DOM**, which will be discussed later in the book, in *Chapter 9*, *Behind The Scenes Of React and Optimization Opportunities*). Thereafter, once all elements for all JSX elements have been created, the virtual DOM is translated into real DOM-manipulating instructions that are executed by the browser.

> **NOTE**
>
> It has been mentioned before that React (in the browser) is actually a combination of two packages: **`react`** and **`react-dom`**.
>
> With the introduction of **`React.createElement(...)`**, it's now easier to explain how these two packages work together: React creates this virtual DOM internally and then passes it to the **`react-dom`** package. This package then generates the actual DOM-manipulating instructions that must be executed in order to update the web page such that the desired user interface is displayed there.
>
> As mentioned, this will be covered in greater detail in *Chapter 9*.

The middle parameter value (**`{}`**, in the example) is a JavaScript object that may contain extra configuration for the element that is to be created.

Here's an example where this middle argument becomes important:

```
function Advertisement() {
  return <a href="https://my-website.com">Visit my website</a>;
};
```

This would be transformed to the following:

```
function Advertisement() {
  return React.createElement(
    'a',
    { href: ' https://my-website.com ' },
    'Visit my website'
  );
};
```

The last argument that's passed to **`React.createElement(...)`** is the child content of the element—i.e., the content that should be between the element's opening and closing tags. For nested JSX elements, nested **`React.createElement(...)`** calls would be produced:

```
function Alert() {
  return (
    <div>
      <h2>This is an alert!</h2>
    </div>
  );
};
```

This would be transformed like this:

```
function Alert() {
  return React.createElement(
    'div', {}, React.createElement('h2', {}, 'This is an alert!')
  );
};
```

USING REACT WITHOUT JSX

Since all JSX code gets transformed to these native JavaScript method calls anyways, you can actually build React apps and user interfaces with React without using JSX.

You can skip JSX entirely if you want to. Instead of writing JSX code in your components and all the places where JSX is expected, you can simply call **`React.createElement(...)`**.

For example, the following two snippets will produce exactly the same user interface in the browser:

```
function App() {
  return (
    <p>Please visit my <a href="https://my-blog-site.com">Blog</a></p>
  );
};
```

The preceding snippet will ultimately be the same as the following:

```
function App() {
  return React.createElement(
    'p',
    {},
    [
      'Please visit my ',
      React.createElement(
        'a',
        { href: 'https://my-blog-site.com' },
        'Blog'
      )
    ]
  );
};
```

Of course, it's a different question whether you would want to do this. As you can see in this example, it's way more cumbersome to rely on **`React.createElement(...)`** only. You end up writing a lot more code and deeply nested element structures will lead to code that can become almost impossible to read.

That's why, typically, React developers do use JSX. It's a great feature that makes building user interfaces with React way more enjoyable. But it is important to understand that it's neither HTML nor a vanilla JavaScript feature, but that it instead is some syntactical sugar that gets transformed to these **`React.createElement(...)`** calls behind the scenes.

JSX ELEMENTS ARE TREATED LIKE REGULAR JAVASCRIPT VALUES!

Because JSX is just syntactical sugar that gets transformed to **`React.createElement()`** calls, there are a couple of noteworthy concepts and rules you should be aware of:

- JSX elements are just **regular JavaScript values** (functions, to be precise) in the end
- The same rules that apply to all JavaScript values also apply to JSX elements
- As a result, in a place, where only one value is expected (e.g., after the **`return`** keyword), you must only have one JSX element

This code would cause an error:

```
function App() {
  return (
    <p>Hello World!</p>
    <p>Let's learn React!</p>
  );
};
```

The code might look valid at first, but it's actually incorrect. In this example, you would return two values instead of just one. That is not allowed in JavaScript.

For example, the following non-React code would also be invalid:

```
function calculate(a, b) {
  return (
    a + b
    a - b
  );
};
```

You can't return more than one value. No matter how you write it.

Of course, you can return an array or an object though. For example, this code would be valid:

```
function calculate(a, b) {
  return [
    a + b,
    a - b
  ];
};
```

It would be valid because you only return one value: an array. This array than contains multiple values as arrays typically do. That would be fine and the same would be the case if you used JSX code:

```
function App() {
  return [
    <p>Hello World!</p>,
    <p>Let's learn React!</p>
  ];
};
```

This kind of code would be allowed since you are returning one array with two elements inside of it. The two elements are JSX elements in this case, but as mentioned earlier, JSX elements are just regular JavaScript values. Thus, you can use them anywhere, where values would be expected.

When working with JSX, you won't see this array approach too often though—simply because it can become annoying to remember wrapping JSX elements via square brackets. It also looks less like HTML which kind of defeats the purpose and core idea behind JSX (it was invented to allow developers to write HTML code inside of JavaScript files).

Instead, if sibling elements are required, as in these examples, a special kind of wrapping component is used: a React **fragment**. That's a built-in component that serves the purpose of allowing you to return or define sibling JSX elements.

```
function App() {
  return (
    <>
      <p>Hello World!</p>
      <p>Let's learn React!</p>
    </>
  );
};
```

This special **<>**...**</>** element is available in most modern React projects (like the one that is created via **`npx create-react-app`**), and you can think of it wrapping your JSX elements with an array behind the scenes. Alternatively, you can also use **`<React.Fragment>`**...**`</React.Fragment>`**. This built-in component is always available.

The parentheses (**()**) that are wrapped around the JSX code in all these examples are required to allow for nice multiline formatting. Technically, you could put all your JSX code into one single line, but that would be pretty unreadable. In order to split the JSX elements across multiple lines, just as you typically do with regular HTML code in **.html** files, you need those parentheses; they tell JavaScript where the returned value starts and ends.

Since JSX elements are regular JavaScript values (after being translated to **React.createElement(...)** calls, at least), you can also use JSX elements in all the places where values can be used.

Thus far, that has been the case for all these **return** statements, but you can also store JSX elements in variables or pass them as arguments to other functions.

```
function App() {
  const content = <p>Stored in a variable!</p>; // this works!
  return content;
};
```

This will be important once you dive into slightly more advanced concepts like conditional or repeated content—something that will be covered later in the book, of course.

JSX ELEMENTS MUST BE SELF-CLOSING

Another important rule related to JSX elements is that they must be self-closing if there is no content between the opening and closing tags.

```
function App() {
  return <img src="some-image.png" />;
};
```

In regular HTML, you would not need that forward backslash at the end. Instead, regular HTML supports void elements (i.e., **<img src="...">**). You can add that forward slash there as well, but it's not mandatory.

When working with JSX, these forward slashes are mandatory, if your element doesn't contain any child content.

OUTPUTTING DYNAMIC CONTENT

Thus far, in all these examples, the content that was returned was static. It was content like **`<p>Hello World!</p>`**—which of course is content that never changes. It will always output a paragraph that says, **`'Hello World!'`**.

At this point in the book, you don't yet have any tools to make the content more dynamic. To be precise, React requires that state concept (which will be covered in a later chapter) to change the content that is displayed (e.g. upon user input or some other event).

Nonetheless, since this chapter is about JSX, it is worth diving into the syntax for outputting dynamic content, even though it's not yet dynamic.

```
function App() {
  const userName = 'Max';
  return <p>Hi, my name is {userName}!</p>;
};
```

This example technically still produces static output since **`userName`** never changes; but you can already see the syntax for outputting dynamic content as part of the JSX code. You use opening and closing curly braces (**`{...}`**) with a JavaScript expression (like the name of a variable or constant, as is the case here) between those braces.

You can put any valid JavaScript expression between those curly braces. For example, you can also call a function (e.g. **`{getMyName()}`**) or do simple inline calculations (e.g. **`{1 + 1}`**).

You can't add complex statements like loops or **`if`**-statements between those curly braces though. Again, standard JavaScript rules apply. You output a (potentially) dynamic value, and therefore, anything that produces a single value is allowed in that place.

WHEN SHOULD YOU SPLIT COMPONENTS?

As you work with React and learn more and more about it, and as you dive into more challenging React projects, you will most likely come up with one very common question: "*When should I split a single React component into multiple, separate components?*".

Because, as mentioned earlier in this chapter, React is all about components, and it is therefore very common to have dozens, hundreds or even thousands of React components in a single React project.

When it comes to splitting a single React component into multiple smaller components, there is no hard rule you must follow. As mentioned earlier, you could put all your UI code into one single, large component. Alternatively, you could create a separate custom component for every single HTML element and piece of content that you have in your UI. Both approaches are probably not that great. Instead, a good rule of thumb is to create a separate React component for every **data entity** that can be identified.

For example, if you're outputting a "to do" list, you could identify two main entities: the individual to-do item and the overall list. In this case, it could make sense to create two separate components instead of writing one bigger component.

The advantage of splitting your code into multiple components is that the individual components stay manageable because there's less code per component and component file.

However, when it comes to splitting components into multiple components, a new problem arises: how do you make your components reusable and configurable?

```
import Todo from './todo';

function TodoList() {
  return (
    <ul>
      <Todo />
      <Todo />
    </ul>
  );
};
```

In this example, all "to-dos" would be the same because we use the same **`<Todo />`** component which can't be configured. You might want to make it configurable by either adding custom attributes (**`<Todo text="Learn React!" />`**) or by passing content between the opening and closing tags (**`<Todo>Learn React!</Todo>`**).

And, of course, React supports this. In the next course chapter, you will learn about a key concept called **props** which allows you to make your components configurable like this.

SUMMARY AND KEY TAKEAWAYS

- React embraces **components**: reusable building blocks that are combined to define the final user interface
- Components must return **renderable** content, typically JSX code which defines the HTML code that should be produced in the end
- React provides a lot of built-in components: besides special components like **<>**...**</>** you get components for all standard HTML elements
- To allow React to tell custom components apart from built-in components, custom component names have to start with capital characters, when being used inside of JSX code (typically, **`PascalCase`** naming is used therefore)
- JSX is neither HTML nor a standard JavaScript feature, instead it's **syntactical sugar** provided by build workflows that are part of all React projects
- You could replace JSX code with **`React.createElement(...)`** calls; but since this leads to significantly more unreadable code, it's typically avoided.
- When using JSX elements, you must not have sibling elements in places where single values are expected (e.g., directly after the **`return`** keyword)
- JSX elements must always be self-closing, if there is no content between the opening and closing tags
- Dynamic content can be output via curly braces (e.g., **`<p>{someText}</p>`**)
- In most React projects, you split your UI code across dozens or hundreds of components which are then exported and imported in order to be combined again

WHAT'S NEXT?

In this chapter, you learned a lot about components and JSX. The next chapter builds up on this key knowledge and explains how you can make components reusable by making them configurable.

Before you continue, you can also practice what you learned up this point by going through the questions and exercises below.

TEST YOUR KNOWLEDGE!

Test your knowledge about the concepts covered in this chapter by answering the below questions. You can then compare your answers to example answers that can be found here: https://packt.link/iSHGL.

1. What's the idea behind using components?
2. How can you create a React component?
3. What turns a regular function into a React component function?
4. Which core rules should you keep in mind regarding JSX elements?
5. How is JSX code handled by React and ReactDOM?

APPLY WHAT YOU LEARNED

With this and the previous chapter, you have all the knowledge you need to create a React project and populate it with some first, basic components.

Below, you'll find your first two activities for this book:

ACTIVITY 2.1: CREATING A REACT APP TO PRESENT YOURSELF

Suppose you are creating your personal portfolio page, and as part of that page, you want to output some basic information about yourself (e.g., your name or age). You could use React and build a React component that outputs this kind of information, as outlined in the following activity.

The aim is to create a React app as you learned it in the previous chapter (i.e., create it via **`npx create-react-app`**, run **`npm start`** to start the development server) and edit the **`App.js`** file such that you output some basic information about yourself. You could, for example output your full name, address, job title or other kinds of information. In the end, it is up to you what content you want to output and which HTML elements you choose.

The idea behind this first exercise is that you practice project creation and working with JSX code.

The steps are as follows:

1. Create a new React project via **`npx create-react-app`**.
2. Edit the App.js file in the **`/src folder`** of the created project and return JSX code with any HTML elements of your choice to output basic information about yourself.

You should get output like this in the end:

Figure 2.2: The final activity result—some user information being output on the screen.

> **NOTE**
>
> The solution to this activity can be found via this link.

ACTIVITY 2.2: CREATING A REACT APP TO LOG YOUR GOALS FOR THIS BOOK

Suppose you are adding a new section to your portfolio site, where you plan to track your learning progress. As part of this page, you plan to define and output your main goals for this book (e.g., *"Learn about key React features"*, *"Do all the exercises"* etc.).

The aim of this activity is to create another new React project in which you add **multiple new components**. Each goal will be represented by a separate component, and all these goal components will be grouped together into another component that lists all main goals. In addition, you can add an extra header component that contains the main title for the webpage.

The steps to complete this activity are as follows:

1. Create a new React project via **`npx create-react-app`**.
2. Inside the new project, create a **`components`** folder which contains multiple component files (for the individual goals as well as for the list of goals and the page header).
3. Inside the different component files, define and export multiple component functions (**`FirstGoal`**, **`SecondGoal`**, **`ThirdGoal`**, etc.) for the different goals (one component per file).
4. Also, define one component for the overall list of goals (**`GoalList`**) and another component for the page header (**`Header`**).
5. In the individual goal components, return JSX code with the goal text and a fitting HTML element structure to hold this content.

6. In the **`GoalList`** component, import and output the individual goal components.
7. Import and output the **`GoalList`** and **`Header`** components in the root App component (replace the existing JSX code).

You should get the following output in the end:

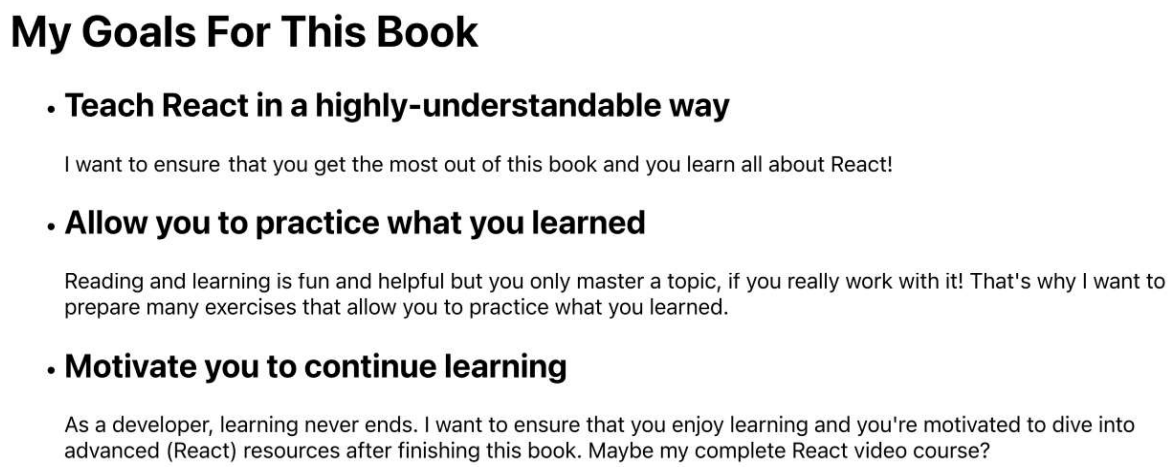


Figure 2.3: The final page output, showing a list of goals.

> **NOTE**
>
> The solution to this activity can be found via this link.

3

COMPONENTS AND PROPS

LEARNING OBJECTIVES

By the end of this chapter, you will be able to do the following:

- Build reusable React components
- Utilize a concept called **Props** to make components configurable
- Build flexible user interfaces by combining components with props

INTRODUCTION

In the previous chapter, you learned about the key building block of any React-based user interface: **components**. You learned why components matter, how they are used, and how you may build components yourself.

You also learned about JSX, which is the HTML-like markup that's typically returned by component functions. It's this markup that defines what should be rendered on the final web page (in other words, which HTML markup should end up on the final web page that is being served to visitors).

NOT THERE YET

But up to this point, those components haven't been too useful. While you could use them to split your web page content into smaller building blocks, the actual reusability of these components was pretty limited. For example, every course goal that you might have as part of an overall course goal list would go into its own component (if you decided to split your web page content into multiple components in the first place).

If you think about it, this isn't too helpful: it would be much better if different list items could share one common component and you would just configure that one component with different content or attributes—just like how HTML works.

When writing plain HTML code and describing content with it, you use reusable HTML elements and configure them with different content or attributes. For example, you have one **<a>** HTML element, but thanks to the **`href`** attribute and the element child content, you can build an endless amount of different anchor elements that point at different resources, as in the following snippets:

```
<a href="https://google.com">Use Google</a>
<a href="https://academind.com">Browse Free Tutorials</a>
```

These two elements use the exact same HTML element (**<a>**) but lead to totally different links that would end up on the web page (pointing to two totally different websites).

To fully unlock the potential of React components, it would therefore be very useful if you could configure them just like regular HTML elements. And it turns out that you can do exactly that—with another key React concept called **props**.

USING PROPS IN COMPONENTS

How do you use props in your components? And when do you need them?

The second question will be answered in greater detail a little bit later. For the moment, it's enough to know that you typically will have some components that are reusable that therefore need props and some components that are unique that might not need props.

The "how" part is the more important part at this point.

And this part can be split into two complementary problems:

1. Passing props to components
2. Consuming props in a component

PASSING PROPS TO COMPONENTS

How would you expect props and component configurability to work if you were to design React from the ground up?

Of course, there would be a broad variety of possible solutions, but there is one great role model that can be considered: HTML. As mentioned above, when working with HTML, you pass content and configuration either between element tags or via attributes.

And fortunately, React components work just like HTML elements when it comes to configuring them. Props are simply passed as attributes (to your component) or as child data between component tags, and you can also mix both approaches:

- **`<Product id="abc1" price="12.99" />`**
- **`<FancyLink target="https://some-website.com">Click me</FancyLink>`**

For this reason, configuring components is quite straightforward—at least, if you look at them from the consumer's angle (in other words, at how you use them in JSX).

But what about defining component functions? How are props handled in those functions?

CONSUMING PROPS IN A COMPONENT

Imagine you're building a **`GoalItem`** component that is responsible for outputting a single goal item (for example, a course goal or project goal) that will be part of an overall goals list.

The parent component JSX markup could look like this:

```
<ul>
  <GoalItem />
  <GoalItem />
  <GoalItem />
</ul>
```

Inside **`GoalItem`**, the goal (no pun intended) would be to accept different goal titles so that the same component (**`GoalItem`**) can be used to output these different titles as part of the final list that's displayed to website visitors. Maybe the component should also accept another piece of data (for example, a unique ID that is used internally).

That's how the **`GoalItem`** component could be used in JSX, as in the following example:

```
<ul>
  <GoalItem id="g1" title="Finish the book!" />
  <GoalItem id="g2" title="Learn all about React!" />
</ul>
```

Inside the **`GoalItem`** component function, the plan would probably be to output dynamic content (in other words, the data received via props) like this:

```
function GoalItem() {
  return <li>{title} (ID: {id})</li>;
}
```

But this component function would not work. It has a problem: **`title`** and **`id`** are never defined inside that component function. This code would therefore cause an error because you're using a variable that wasn't defined.

Of course, it shouldn't be defined inside the **`GoalItem`** component, though, because the idea was to make the **`GoalItem`** component reusable and receive different **`title`** and **`id`** values *from outside the component* (i.e., from the component that renders the list of **`<GoalItem>`** components).

React provides a solution for this problem: a special parameter value that will be passed into every component function automatically by React. This is a special parameter that contains the extra configuration data that is set on the component in JSX code, called the **`props`** parameter.

The preceding component function could (and should) be rewritten like this:

```
function GoalItem(props) {
  return <li>{props.title} (ID: {props.id})</li>;
}
```

The name of the parameter (**`props`**) is up to you, but using **`props`** as a name is a convention because the overall concept is called **props**.

To understand this concept, it is important to keep in mind that these component functions are not called by you somewhere else in your code, but that instead, React will call these functions on your behalf. And since React calls these functions, it can pass extra arguments into the functions when calling them.

This **`props`** argument is indeed such an extra argument. React will pass it into every component function, irrespective of whether you defined it as an extra parameter in the component function definition. Though, if you didn't define that **`props`** parameter in a component function, you, of course, won't be able to work with the props data in that component.

This automatically provided **`props`** argument will always contain an object (because React passes an object as a value for this argument), and the properties of this object will be the "attributes" you added to your component inside the JSX code where the component is used.

That's why in this **`GoalItem`** component example, custom data can be passed via attributes (**`<GoalItem id="g1" ... />`**) and consumed via the **`props`** object and its properties (**`<li>{props.title}</li>`**).

COMPONENTS, PROPS, AND REUSABILITY

Thanks to this props concept, components become *actually* reusable, instead of just being *theoretically* reusable.

Outputting three **<GoalItem>** components without any extra configuration could only render the same goal three times since the goal text (and any other data you might need) would have to be hardcoded into the component function.

By using props as described above, the same component can be used multiple times with different configurations. That allows you to define some general markup structure and logic once (in the component function) but then use it as often as needed with different configurations.

And if that sounds familiar, that is indeed exactly the same idea as for regular JavaScript (or any other programming language) functions. You define logic once, and you can then call it multiple times with different inputs to receive different results. It's the same for components—at least when embracing this props concept.

THE SPECIAL "CHILDREN" PROP

It was mentioned before that React passes this **props** object automatically into component functions. That is the case, and, as described, this object contains all the attributes you set on the component (in JSX) as properties.

But React does not just package your attributes into this object; it also adds another extra property into the props object: the special **children** property (a built-in property whose name is fixed, meaning you can't change it).

The **children** property holds a very important piece of data: the content you might have provided between the component's opening and closing tags.

Thus far, in the examples shown above, the components were mostly self-closing. **<GoalItem id="..." title="..." />** holds no content between the component tags. All the data is passed into the component via attributes.

There is nothing wrong with this approach. You can configure your components with attributes only. But for some pieces of data and some components, it might make more sense and be more logical to actually stick to regular HTML conventions and pass that data between the component tags instead. And the **GoalItem** component is actually a great example.

Which approach looks more intuitive?

1. **<GoalItem id="g1" title="Learn React" />**
2. **<GoalItem id="g1">Learn React</GoalItem>**

You might determine that the second option looks a bit more intuitive and in line with regular HTML because, there, you would also configure a normal list item like this: **`<li id="li1">Some list item</li>`**.

While you have no choice when working with regular HTML elements (you can't just add a **`goal`** attribute to a **`<li>`** just because you want to), you do have a choice when working with React and your own components. It simply depends on how you consume props inside the component function. Both approaches can work, depending on that internal component code.

Still, you might want to pass certain pieces of data between component tags, and the special **`children`** property allows you to do just that. It contains any content you define between the component opening and closing tags. Therefore, in the case of example 2 (in the list above), **`children`** would contain the string **`"Learn React"`**.

In your component function, you can work with the **`children`** value just as you work with any other prop value:

```
function GoalItem(props) {
  return <li>{props.children} (ID: {props.id})</li>;
}
```

WHICH COMPONENTS NEED PROPS?

It was mentioned before already, but it is extremely important: **Props are optional!**

React will always pass **prop** data into your components, but you don't have to work with that **`prop`** parameter. You don't even have to define it in your component function if you don't plan on working with it.

There is no hard rule that would define which components need **props** and which don't. It comes with experience and simply depends on the role of a component.

You might have a general **`Header`** component that displays a static header (with a logo, title, and so on), and such a component probably needs no external configuration (in other words, no "attributes" or other kinds of data passed into it). It could be self-contained, with all the required values hardcoded into the component.

But you will also often build and use components like the **`GoalItem`** component (in other words, components that do need external data to be useful). Whenever a component is used more than once in your React app, there is a high chance that it will utilize props. However, the opposite is not necessarily true. While you will have one-time components that don't use props, you absolutely will also have components that are only used once in the entire app and still take advantage of props. As mentioned, it depends on the exact use case and component.

Throughout this book, you will see plenty of examples and exercises that will help you gain a deeper understanding of how to build components and use props.

HOW TO DEAL WITH MULTIPLE PROPS

As shown in the preceding examples, you are not limited to only one prop per component. Indeed, you can pass and use as many props as your component needs—no matter if that's 1 or 100 (or more) props.

Once you do create components with more than just two or three props, a new question might come up: do you have to add all those props individually (in other words, as separate attributes)? Or can you pass fewer attributes that contain grouped data, such as arrays or objects?

And indeed, you can. React allows you to pass arrays and objects as prop values as well. In fact, any valid JavaScript value can be passed as a prop value!

This allows you to decide whether you want to have a component with 20 individual props ("attributes") or just one "big" prop. Here's an example of where the same component is configured in two different ways:

```
<Product title="A book" price={29.99} id="p1" />
// or
const productData = {title: 'A book', price: 29.99, id: 'p1'}
<Product data={productData} />
```

Of course, the component must also be adapted internally (in other words, in the component function) to expect either individual or grouped props. But since you're the developer, that is, of course, your choice.

Inside the component function, you can also make your life easier.

There is nothing wrong with accessing prop values via **`props.XYZ`**, but if you have a component that receives multiple props, repeating **`props.XYZ`** over and over again could become cumbersome and make the code a bit harder to read.

You can use a default JavaScript feature to improve readability: **object destructuring**.

Object destructuring allows you to extract values from an object and assign those values to new variables or constants in a single step:

```
const user = {name: 'Max', age: 29};
const {name, age} = user; // <-- this is object destructuring in action
console.log(name); // outputs 'Max'
```

You can therefore use this syntax to extract all prop values and assign them to equally named variables directly at the start of your component function:

```
function Product({title, price, id}) { // destructuring in action
  … // title, price, id are now available as variables inside this
function
}
```

You don't have to use this syntax, but it can make your life easier.

> **NOTE**
>
> When using object destructuring to extract prop values, it is worth noting that this only works if **single-value** props were passed to the component. You can also extract grouped props, but there you only have one property to destructure: the attribute that contains the grouped data.
>
> For more information on object destructuring, MDN is a great place to dive deeper. You can access this at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment.

SPREADING PROPS

Imagine you're building a custom component that should act as a "wrapper" around some other component— a built-in component, perhaps.

For instance, you could be building a custom **`Link`** component that should return a standard **`<a>`** element with some custom styling (covered in *Chapter 6, Styling React Apps*) or logic added:

```
function Link({children}) {
  return <a target="_blank" rel="noopener noreferrer">{children}</a>
};
```

This very simple example component returns a pre-configured **<a>** element. This custom **Link** component configures the anchor element such that new pages are always opened in a new tab. In place of the standard **<a>** element, you could be using this **Link** component in your React app to get that behavior out of the box for all your links.

But this custom component suffers from a problem: it's a wrapper around a core element, but by creating your own component, you remove the configurability of that core element. If you were to use this **Link** component in your app, how would you set the **href** prop to configure the link destination?

You might try the following:

```
<Link href="https://some-site.com">Click here</Link>
```

However, this example code wouldn't work because **Link** doesn't accept or use a **href** prop.

Of course, you could adjust the **Link** component function such that a href prop is used:

```
function Link({children, href}) {
  return <a href={href} target="_blank" rel="noopener
noreferrer">{children}</a>
};
```

But what if you also wanted to ensure that the **download** prop could be added if needed?

Well, it's true that you can always accept more and more props (and pass them on to the **<a>** element inside your component), but this reduces the reusability and maintainability of your custom component.

A better solution is to use the standard JavaScript spread operator (i.e., the **...** operator) and React's support for that operator when working with components.

For example, the following component code is valid:

```
function Link({children, config}) {
  return <a {...config} target="_blank" rel="noopener
noreferrer">{children}</a>
};
```

In this example, **config** is expected to be a JavaScript object (i.e., a collection of key-value pairs). The spread operator (**...**), when used in JSX code on a JSX element, converts that object into multiple props.

Consider this example **`config`** value:

```
const config = { href: 'https://some-site.com', download: true };
```

In this case, when spreading it on **`<a>`**, (i.e., **`<a {...config}>`**), the result would be the same as if you had written this code:

```
<a href="https://some-site.com" download={true}>
```

The behavior and pattern can be used to build reusable components that should still maintain the configurability of the core element they may be wrapping. This helps you avoid long lists of pre-defined, accepted props and improves the reusability of components.

PROP CHAINS / PROP DRILLING

There is one last phenomenon that is worth noting when learning about props: "prop drilling" or "prop chains".

This isn't an official term, but it's a problem every React developer will encounter at some point. It occurs when you build a slightly more complex React app that contains multiple layers of nested components that need to send data to each other.

For example, assume that you have a **`NavItem`** component that should output a navigation link. Inside that component, you might have another nested component, **`AnimatedLink`**, that outputs the actual link (maybe with some nice animation styling).

The **`NavItem`** component could look like this:

```
function NavItem(props) {
  return <div><AnimatedLink target={props.target} text="Some text" /></
div>;
}
```

And **`AnimatedLink`** could be defined like this:

```
function AnimatedLink(props) {
  return <a href={props.target}>{props.text}</a>;
}
```

In this example, the **`target`** prop is passed through the **`NavItem`** component to the **`AnimatedLink`** component. The **`NavItem`** component must accept the **`target`** prop because it must be passed on to **`AnimatedLink`**.

That's what prop drilling / prop chains is all about: you forward a prop from a component that doesn't really need it to another component that *does* need it.

Having some prop drilling in your app isn't necessarily bad and you can definitely accept it. But if you should end up with longer chains of props (in other words, multiple **pass-through components**), you can use a solution that will be discussed in *Chapter 10, Working with Complex States*.

SUMMARY AND KEY TAKEAWAYS

- Props are a key React concept that makes components configurable and therefore reusable.
- Props are automatically collected and passed into component functions by React.
- You decide (on a per-component basis) whether you want to use the props data (an object) or not.
- Props are passed into components like attributes or, via the special **`children`** prop, between the opening and closing tags.
- Since you are writing the code, it's up to you how you want to pass data via props. Between the tags or as attributes? A single grouped attribute or many single-value attributes? It's up to you.

WHAT'S NEXT?

Props allow you to make components configurable and reusable. Still, they are rather static. Data and therefore the UI output doesn't change. You can't react to user events like button clicks.

But the true power of React only becomes visible once you do add events (and reactions to them).

In the next chapter, you will learn how you can add event listeners when working with React and you will learn how you can react (no pun intended) to events and change the (invisible and visible) state of your application.

TEST YOUR KNOWLEDGE!

Test your knowledge regarding the concepts covered in this chapter by answering the following questions. You can then compare your answers to the example answers that can be found here: https://packt.link/ekAg6.

1. Which "problem" do props solve?
2. How are props passed into components?
3. How are props consumed inside a component function?
4. Which options exist for passing (multiple) props into components?

APPLY WHAT YOU LEARNED

With this and the previous chapters, you now have enough basic knowledge to build truly reusable components.

Below, you will find an activity that allows you to apply all the knowledge, including the new props knowledge, you have acquired up to this point:

ACTIVITY 3.1: CREATING AN APP TO OUTPUT YOUR GOALS FOR THIS BOOK

This activity builds upon *Activity 2.2, "Create a React app to log your goals for this book"* from the previous chapter. If you followed along there, you can use your existing code and enhance it by adding props. Alternatively, you can also use the solution provided as a starting point accessible through the following link: https://packt.link/8tvm6.

The aim of this activity is to build reusable **`GoalItem`** components that can be configured via props. Every **`GoalItem`** component should receive and output a goal title and a short description text with extra information about the goal.

The steps are as follows:

1. Complete the second activity from the previous chapter.
2. Replace the hardcoded goal item components with a new configurable component.
3. Output multiple goal components with different titles and descriptions (configured via props).

The final user interface could look like this:

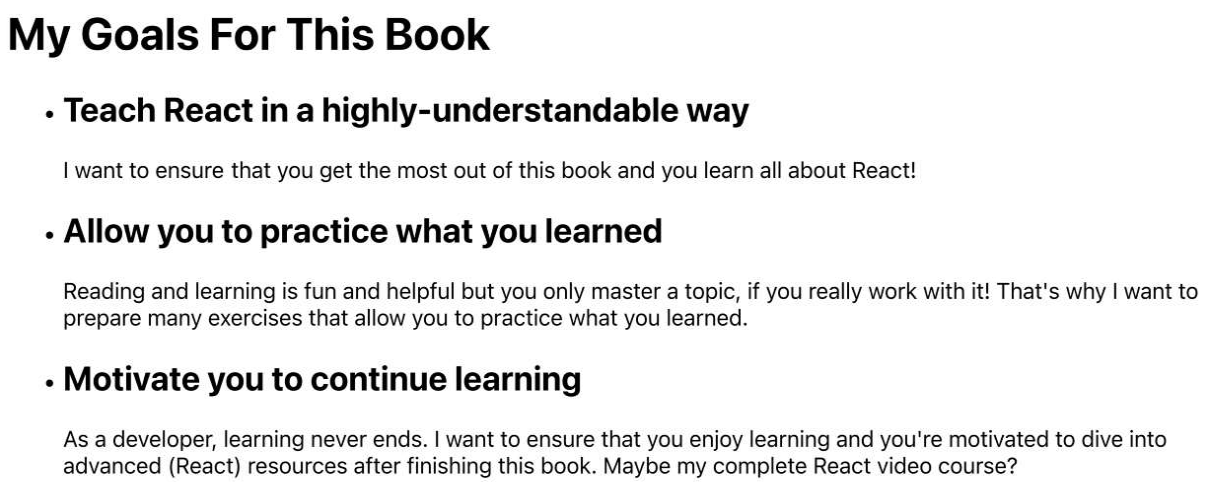


Figure 3.1: The final result: Multiple goals output below each other

> **NOTE**
>
> The solution to this activity can be found via this link.