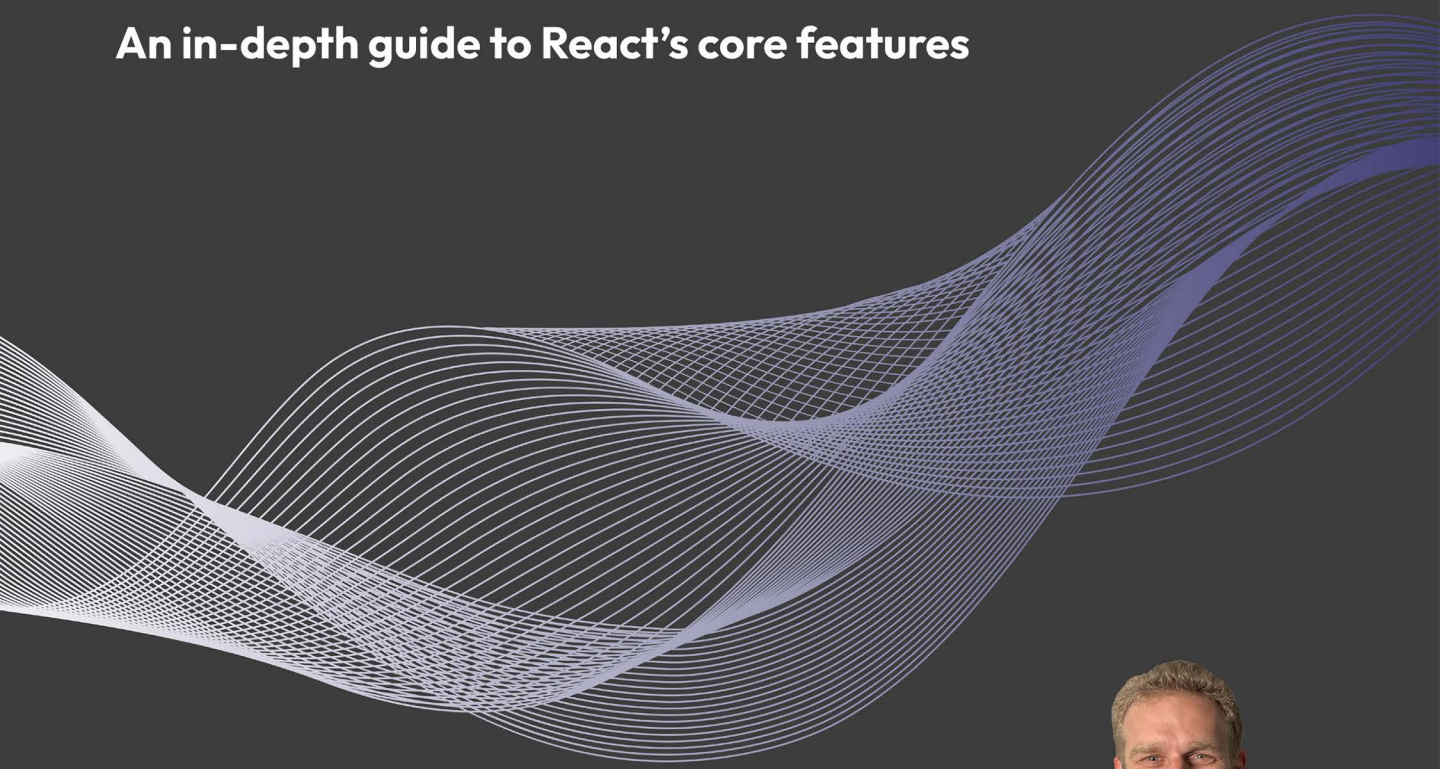
EXPERT INSIGHT

React Key Concepts

An in-depth guide to React's core features



Second Edition

Maximilian Schwarzmüller

packt

React Key Concepts

Second Edition

An in-depth guide to React's core features

Maximilian Schwarzmüller

‹packt›

React Key Concepts

Second Edition

Publishing Product Manager: Lucy Wan
Acquisition Editor – Peer Reviews: Jane Dsouza
Project Editor: Janice Gonsalves
Senior Development Editor: Elliot Dallow
Copy Editor: Safis Editing
Technical Editor: Tejas Mhasvekar
Proofreader: Safis Editing
Indexer: Pratik Shirodkar
Presentation Designer: Ajay Patule
Developer Relations Marketing Executive: Priyadarshini Sharma

First published: December 2022
Second edition: December 2024

Production reference: 1231224

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK.

ISBN 978-1-83620-227-1

www.packt.com

Contributors

About the Author

Maximilian Schwarzmüller is a professional web developer and bestselling online course instructor. On Udemy, he is one of the most popular and biggest online instructors, teaching more than 3 million students worldwide. Students can become developers by exploring his more than 40 courses, most of them bestsellers in their respective categories.

Besides helping students from all over the world, Maximilian 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.

I may be the author of this book but planning, polishing, and publishing this book was really a group effort.

Most of all, I'm thankful for all the support from my wife Anna-Maria. You're the love of my life!

I also want to thank my publisher, Packt: Thank you Bridget, Megan, Elliot, Janice, Lucy, Tejas, and everyone else who was involved!

About the Reviewers

Cihan Yakar has over twenty years of experience in software development. He specializes in fullstack development and machine learning, creating applications with .NET and Node.js. An enthusiastic learner and knowledge sharer, Cihan often speaks at user group meetings. He is the founder of Bitsody Software and Defne Software. He was also a technical reviewer of *The TypeScript Workshop*. To discover more about his professional journey, feel free to connect with him on LinkedIn. When not working, Cihan enjoys spending time with his family and indulging in his passion for all things Star Trek.

Slava Knyazev has been writing software since his early teenage years and is always seeking to find ways to improve his mastery of the craft. He has worked for well-known names, including theScore, Amazon Web Services, and Airbnb. When he isn't writing code, he dives into technical topics on his blog, *Building Better Software Slower*.

Eric Harvey is a consultant for Enwise Webtech LLC, focused on EdTech and secure systems integrations. He has worked in technology since 1998, his roles have included: applications engineer, web developer, manager of learning systems at a major university, and solutions engineer. In 2005, he founded a web development and hosting services company. Outside of work he is an avid collector of board games and vintage computers, and plays mandolin in a local Celtic string band.

I would like to thank my kids – Amber, Nate, and Rylan – and my wife, Meredith, for being understanding, patient, and always loving.

Join Us on Discord

Read this book alongside other users, AI experts, and the author himself.

Ask questions, provide solutions to other readers, chat with the author via Ask Me Anything sessions, and much more.

Scan the QR code or visit the link to join the community.

`https://packt.link/ReactKeyConcepts2e`

Table of Contents

Chapter 16: React Server Components & Server Actions 429

Chapter 17: Understanding React Suspense & The use() Hook 467

Preface

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 three million students worldwide learn how to code, and his latest React video course (*React—The Complete Guide*) has over eight hundred thousand students on Udemy.

Max has written this in-depth 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 19, so you can be sure you're ready to go with the latest version.

Who This Book Is For

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 recommended that you have some understanding of the fundamentals of JavaScript, HTML, and CSS.

What This Book Covers

Chapter 1, *React – What and Why*, will re-introduce you to React.js. Assuming that React.js is not brand-new to you, 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 you 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., lists of user posts) and conditional content (e.g., alert if incorrect values are 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, Tailwind 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 that is built-into React. In addition, you 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, *Handling User Input & Forms with Form Actions*, will explore how React simplifies the process of handling forms by allowing you to define client-side form actions that are triggered upon submission.

Chapter 10, *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 you avoid unnecessary re-render cycles (and thus improve performance).

Chapter 11, *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 you to manage app-wide state with ease.

Chapter 12, *Building Custom React Hooks*, will build up on the previous chapters and explore how you can build your own, custom React hooks and what the advantage of doing so is.

Chapter 13, *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 14, *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 15, *Server-side Rendering & Building Fullstack Apps with Next.js*, will help you understand the concept of **server-side rendering** (**SSR**) and help you use your React knowledge with the popular Next.js framework to build applications that span across both the front and backend.

Chapter 16, *React Server Components & Server Actions*, will build upon the idea of building fullstack React apps and explain how you may render components and handle form submissions on the server side.

Chapter 17, *Understanding React Suspense & The use() Hook*, will explain how React helps you provide better user experiences by showing fallback content while data is being fetched.

Chapter 18, *Next Steps and Further Resources*, will cover the core and extended React ecosystem and which resources may be helpful for next steps.

This book also comes with the following downloadable supplementary content:

- A cheatsheet accompanying every chapter of the book
- A video in which author Maximilian gives you his recommendations for next steps after finishing this book
- A video in which author Maximilian shares his thoughts about the future of React

Instructions for claiming this content are available at the end of the *Preface*.

Staying Up to Date with This Book

This edition of this book was written when React 19 was released, though most of the core concepts explained throughout this book have beern around since React 18 or even before that. Thus, the vast majority of the features covered in this book can be considered extremely stable and unlikely to change in the near future.

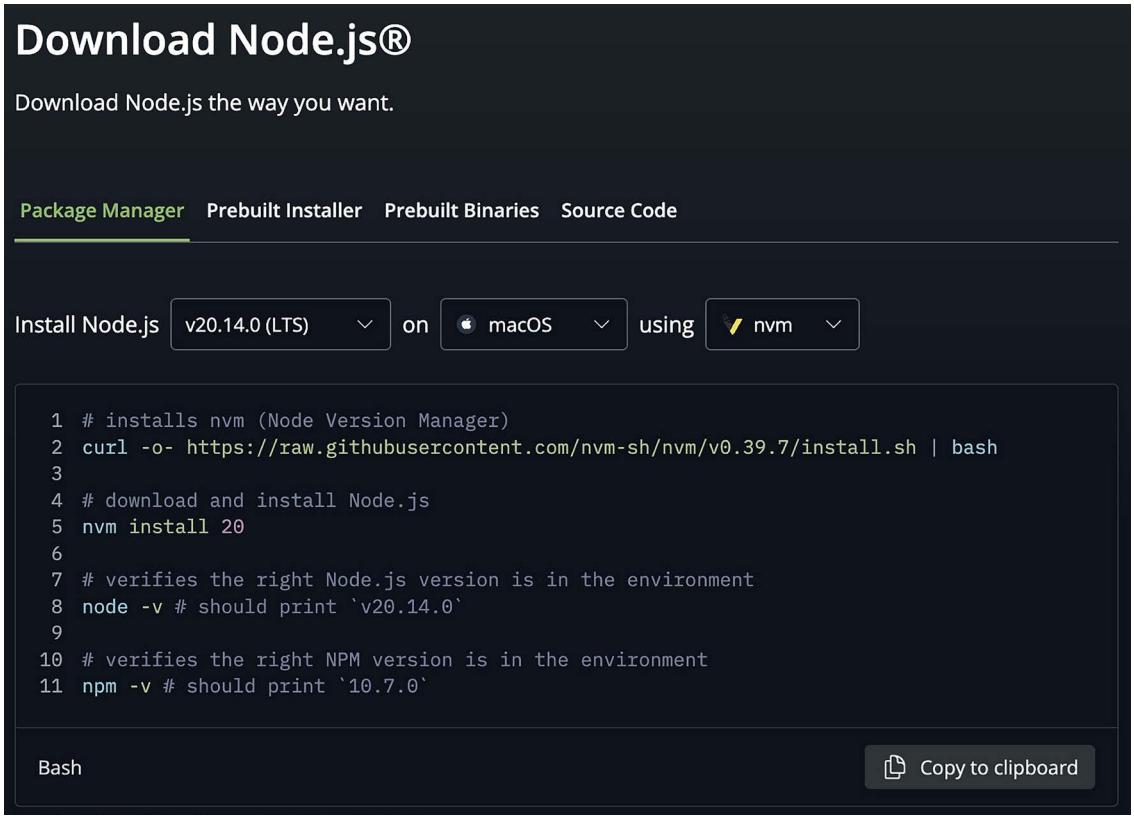
But the book will also cover some relatively new React features, like server components or server actions. Whilst breaking changes are also unlikely for those concepts, a document has been created on GitHub to track any corrections or deviations you should be aware of when reading this book: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/blob/main/CHANGELOG.md`.

Following Along with the Book

Before you can successfully create and run React.js projects on your system, you will need to ensure you have **Node.js** and **npm** (included with your installation by default) installed.

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 **Downloads** in the site navigation bar. This will open a new page through which you can explore all installation choices for all main platforms, as shown in the screenshot below:



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 recommended way for this book is the usage of the Vite tool though. This tool and the process of creating a React app will be covered in *Chapter 1*, *React – What and Why*, but you may refer to this section for step-by-step instructions on this task.

Perform the following steps to create a React.js project 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:

```
npm create vite@latest my-app
```

After running this command, choose **React** and **JavaScript** when prompted for input.

4. Once completed, navigate to your new directory using the `cd` command:

```
cd my-app
```

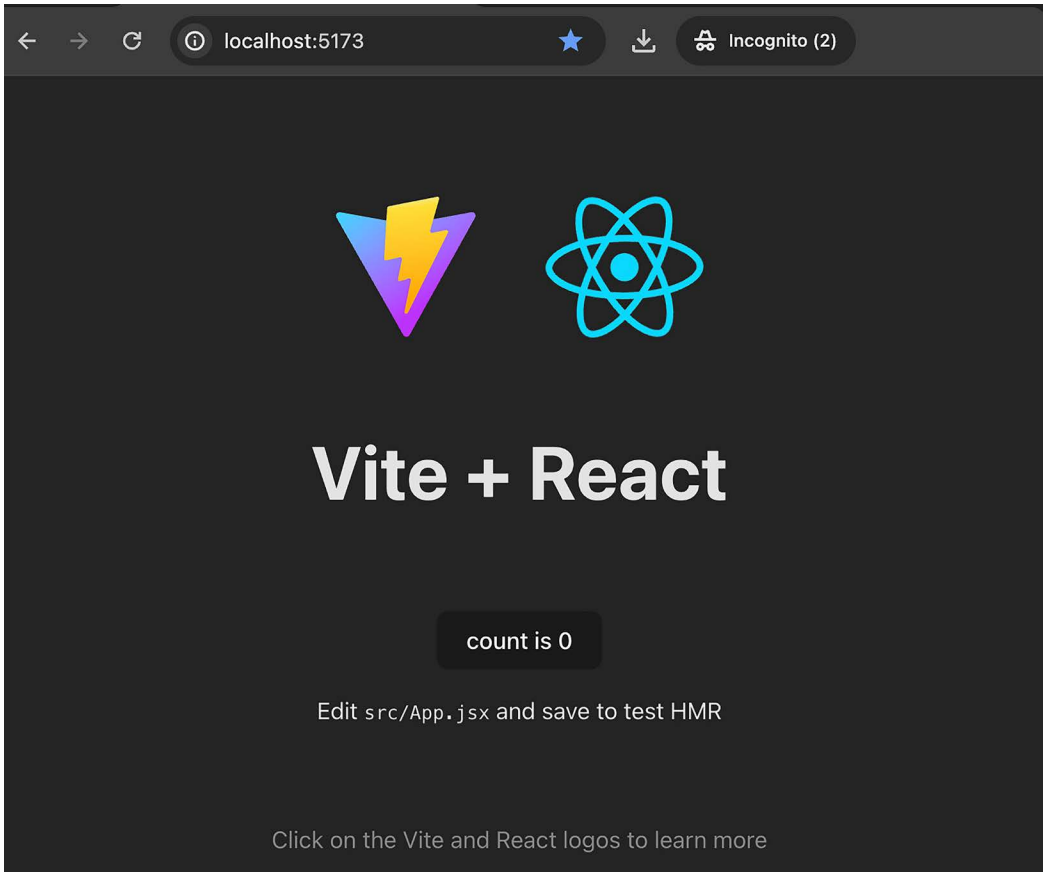
5. Open a terminal window in this new project directory and run the following command to install all required dependencies:

```
npm install
```

6. Once this command is completed, in the same terminal, run the following command to start a Node.js development server:

```
npm run dev
```

7. This command outputs a server address you can visit to preview the React application. By default, the address is `http://localhost:5173`. Type that address in the address/location bar to navigate to `localhost:5173`, as shown in the screenshot below:



8. When you are ready to stop development for the time being, use *Ctrl + C* in the same terminal as in *Step 5* to quit running your server. To relaunch it, simply run the `npm run dev` command in that terminal once again. Keep the process started by `npm run dev` up and running while developing, as it will automatically update the website loaded on `localhost:5173` with any changes you make.

Download the Example Code Files

The code bundle for the book is hosted on GitHub at `https://github.com/mschwarzmueller/book-react-key-concepts-e2`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the Color Images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://packt.link/gbp/9781836202271`.

Conventions Used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: "Once the root entry point has been defined, a method called `render()` can be called on the `root` object created via `createRoot()`."

A block of code is set as follows:

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

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

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

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

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

import classes from './Error.module.css';

function Error({ message }) {
  console.log('<Error /> component function is executed.');
  if (!message) {
    return null;
```

```
  }

  return <p className={classes.error}>{message}</p>;
}

export default memo(Error);
```

Any command-line input or output is written as follows:

```
npm create vite@latest my-react-project
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: "React simplifies the creation and management of such UIs by moving from an **imperative** to a **declarative** approach."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in Touch

Feedback from our readers is always welcome.

General feedback: Email `feedback@packtpub.com` and mention the book's title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit `http://www.packtpub.com/submit-errata`, click **Submit Errata**, and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packtpub.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `http://authors.packtpub.com`.

Share Your Thoughts

Once you've read *React Key Concepts, Second Edition*, we'd love to hear your thoughts! Please `click here to go straight to the Amazon review page` for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download the Free PDF and Supplementary Content

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

Additionally, with this book you get access to supplementary/bonus content for you to learn more. You can use this to add on to your learning journey on top of what you have in the book.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily.

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below:

`https://packt.link/supplementary-content-9781836202271`

2. Submit your proof of purchase.
3. Submit your book code. You can find the code on page no. 169 of the book.
4. That's it! We'll send your free PDF, supplementary content, and other benefits to your email directly

Description of Supplementary Content

This book comes with the following bonus material (claimable via the mechanism described above):

- A cheatsheet accompanying every chapter of the book
- A video in which author Maximilian gives you his recommendations for next steps after finishing this book
- A video in which author Maximilian shares his thoughts about the future of React

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
- Explain the difference between imperative and declarative code
- Differentiate between **single-page applications** (**SPAs**) and multi-page apps
- 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 2023 Stack Overflow developer survey. It is currently used by over 5% of the top 1,000 websites and compared to other popular frontend JavaScript libraries and frameworks like Angular, React is leading by a huge margin, when looking at key metrics like weekly package downloads via **npm**, 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'll likely be able to learn advanced concepts quicker and avoid errors when trying 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 read this chapter first.

What is React?

React is a JavaScript library, and if you take a look at the official web page (the official React website and documentation are available at this link: `https://react.dev/`), you learn that the creators call it *"The library for web and native user interfaces."*

But what does this mean?

First, it's important to understand that React is a JavaScript library. As a reader of this book, you know what JavaScript is and why you use JavaScript in the browser. JavaScript allows you to add interactivity to your website since, with JavaScript, you can react to user events and manipulate the page after it is loaded. This is extremely valuable as it allows you to build highly interactive web **user interfaces** (**UIs**).

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

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 can help you write more concise and possibly also less error-prone code and enable you to implement certain features more quickly.

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

The Problem with "Vanilla JavaScript"

Vanilla JavaScript is a term commonly used in web development to refer 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 UIs and websites can quickly become very hard to manage with vanilla JavaScript.

React simplifies the creation and management of such UIs 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, the idea behind "imperative versus 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.

Let's look at a short code snippet that shows how you could handle the following UI 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 later). But even though this example is straightforward to digest, working with vanilla JavaScript will quickly reach its limits for feature-rich UIs and the code to handle various user interactions accordingly also becomes more complex. Code can quickly grow significantly, so maintaining it can become a challenge.

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

The code shown previously could be translated into these more human-readable instructions:

1. Look for an **HTML** element of the `button` type 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.

However, when working with UIs, 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 **Document Object Model** (**DOM**) instructions that allow your code to interact with elements, add elements, manipulate elements, and so on.

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 UIs 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.

As a result, 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. 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 UIs 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.

The button-clicking example 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. Also, 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. Here, 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://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/01-what-is-react/examples/example-1/vanilla-javascript`.

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 UI 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 UI

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://www.toptal.com/developers/javascript-minifier`) 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 UI with only basic functionality. The actual business logic (i.e., input validation, determining whether and when overlays should be shown, and defining the output text) only makes up a small fraction of the overall code base – around 20 to 30 lines of code, in this case (around 20 after minifying).

That's roughly 75% of the 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 earlier, 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 a new text once the click on the button occurs.

Nonetheless, this code looks totally different – like a mixture of JavaScript and HTML. Indeed, React uses a syntax extension called **JSX** (i.e., JavaScript extended to include XML-like syntax). For the moment, it's enough to understand that this JSX code will work because of a **pre-processing** (or **transpilation**) 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 it is deployed. This allows for development-only syntax like JSX, which would not work in the browser and is for that reason transformed to regular JavaScript before deployment. (You'll get a thorough introduction to JSX in *Chapter 2*, *Understanding React Components and JSX*.)

In addition, the snippet shown previously 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 states with React). For the moment, you can think of this state as a variable that, when changed, will trigger React to update the UI 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 the 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 hence, 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 and updating its `textContent`). Instead, you will hand this work off to React. You will only need to focus on the desired end states 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 to do 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 UI, 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 `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/01-what-is-react/examples/example-1/vanilla-javascript` and `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/01-what-is-react/examples/example-1/reactjs`, respectively.

```jsx
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 is 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. However, 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.

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 states 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 UIs tremendously.

Note

It is worth emphasizing that React is not a great solution if you're working on a very simple UI. 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 UI (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 UI by using different HTML elements and 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 previously: 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 into regular JavaScript code before being served to browsers. The next chapter will take a closer look at JSX and what it's transformed into. 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 need to be added to every React project). Internally, React creates a virtual DOM-like tree structure that reflects the current state of the UI. This book takes a closer look at this abstract, virtual DOM, and how React works in *Chapter 10, Behind the Scenes of React and Optimization Opportunities*. That's why React (the library) splits its core logic across two main packages:

- The main `react` package
- 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 the help of React. With React Native, you also include the react package in 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 SPAs

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

- Manage parts of a website (e.g., a chat box in the bottom left corner).
- Manage the entire page and all user interactions that occur 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 UIs have not just one, but multiple complex 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 13*, *Multipage Apps with React Router*), which integrate with React, allow you to create a website wherein the entire UI 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 often built as a SPA because it's common to create React projects that contain 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 UI. This means that the entire UI is created and managed by JavaScript via React and your React code.

That being said, it's also becoming more and more popular to build full-stack React apps, where frontend and backend code are merged. Modern React frameworks like **Next.js** simplify the process of building such web apps. Whilst the core concepts are the same, no matter which kind of application is built, this book will explore full-stack React app development in greater detail in *Chapter 15*, *Server-side Rendering & Building Fullstack Apps with Next.js*, *Chapter 16*, *React Server Components and Server Actions* and *Chapter 17*, *Understanding React Suspense and the use() Hook*.

Ultimately, this book prepares you for working with React on all kinds of React projects since the core building blocks and key concepts are always the same.

Creating a React Project with Vite

To work with React, the first step is the creation of a React project. The official documentation recommends using a framework like Next.js. But while this might make sense for complex web applications, it's overwhelming for getting started with React and for exploring React concepts. Next.js and other frameworks introduce their own concepts and syntax. As a result, learning React can quickly become frustrating since it can be difficult to tell React features apart from framework features. In addition, not all React apps need to be built as full-stack web apps – consequently, using a framework like Next.js might add unnecessary complexity.

That's why Vite-based React projects have emerged as a popular alternative. **Vite** is an open-source development and build tool that can be used to create and run web development projects based on all kinds of libraries and frameworks – React is just one of the many options.

Vite creates projects that come with a built-in, preconfigured build process that, in the case of React projects, takes care of the JSX code transpilation. It also provides a development web server that runs locally on your system and allows you to preview the React app while you're working on it.

You need a project setup like this because React projects typically use features like JSX, which wouldn't work in the browser without prior code transformation. Hence, as mentioned earlier, a pre-processing step is required.

To create a project with Vite, 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` command, which you can use to utilize the Vite package to create a new React project.

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

```
npm create vite@latest my-react-project
```

Once executed, this command will prompt you to choose a framework or library you want to use for this new project. You should choose React and then JavaScript.

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.

It's worth noting that the project creation command does not install any required dependencies such as the React library packages. For that reason, you must navigate into the created folder in your system terminal or command prompt (via `cd my-react-project`) and install these packages by running the following command:

```
npm install
```

Once the installation finishes successfully, the project setup process is complete.

To view the created React application, you can start a development server on your machine via this command:

```
npm run dev
```

This invokes a script provided by Vite, which will spin up a locally running web server that pre-processes, builds, and hosts your React-powered SPA – by default on `localhost:5173`. Therefore, while working on the code, you typically have this development server up and running as it allows you to preview and test code changes.

Best of all, this local development server will automatically update the website whenever you save any code changes, hence allowing you to preview your changes almost instantly.

You can quit this server whenever you're done for the day by pressing *Ctrl* + *C* in the terminal or command prompt where you executed `npm run dev`.

Whenever you're ready to start working on the project again, you can restart the server via `npm run dev`.

Note

In case you encounter any issues with creating a React project, you can also download and use the following starting project: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/01-what-is-react/react-starting-project`. It's a project created via Vite, which can be used in the same way as if it were created with the preceding command.

When using this starting project (or, in fact, any GitHub-hosted code snapshot belonging to this book), you need to run `npm install` in the project folder first, before running `npm run dev`.

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

- A `src/` folder, which contains the main source code files for the project:
    - A `main.jsx` file, which is the main entry script file that will be executed first

    - An `App.jsx` file, which contains the root component of the application (you'll learn more about components in the next chapter)
    - Various styling (`*.css`) files, which are imported by the JavaScript files
    - An `assets/` folder that can be used to store images or other assets that should be used in your React code
- A `public/` folder, which contains static files that will be part of the final website (e.g., a favicon)
- An `index.html` file, which is the single HTML page of this website
- `package.json` and `package-lock.json` are files that list and define the third-party dependencies of your project:
    - Production dependencies like `react` or `react-dom`
    - Development dependencies like `eslint` for automated code quality checks
- Other project configuration files (e.g., `.gitignore` for managing Git file tracking)
- A `node_modules` folder, which contains the actual code of the installed third-party packages

It's worth noting that `App.jsx` and `main.jsx` use `.jsx` as a file extension, not `.js`. This is a file extension that's enforced by Vite for files that do not just contain standard JavaScript but also JSX code. When working on a Vite project, most of your project files will consequently use `.jsx` as an extension.

Almost all of the React-specific code will be written in the `App.jsx` file or custom component files that will be added to the project. We will explore components in the next chapter.

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 at `https://docs.npmjs.com/`.

The code of the project's dependencies is stored in the `node_modules` folder. This folder can become very big since it contains the code of all installed packages and their dependencies. For that reason, 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.

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 UIs without React, simply using vanilla JavaScript to do so can be cumbersome, error prone, and hard to maintain.
- React simplifies the creation of complex UIs by providing a declarative way to define the desired end states of the UI.

- **Declarative** means that you define the target UI 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 is a "bridge," like `react-dom` or `react-native`, that translates this virtual DOM into actual UI (DOM) instructions.
- With React, you can build SPAs, meaning that React is used to control the entire UI on all pages as well as the routing between pages.
- You can also use React, in combination with frameworks like Next.js, to build full-stack web applications where server- and client-side code are connected.
- React projects can be created with the help of the Vite 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 UIs. You learned how to create new React projects with Vite, 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 UIs 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 into 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 following questions. You can then compare your answers to example answers that can be found here: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/blob/01-what-is-react/exercises/questions-answers.md`.

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 complex project setup?

Join Us on Discord

Read this book alongside other users, AI experts, and the author himself.

Ask questions, provide solutions to other readers, chat with the author via Ask Me Anything sessions, and much more.

Scan the QR code or visit the link to join the community.

`https://packt.link/ReactKeyConcepts2e`

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 relationship 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 chapter, you learned about React in general, what it is, and why you should consider using it for building user interfaces. You also learned how to create React projects with the help of Vite, by running `npm create vite@latest <your-project-name>`.

In this chapter, you will learn about one of the most important React concepts and building blocks. You will learn that components are reusable building blocks that 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 that are combined to compose the final user interface. For example, a basic website could be made up of a sidebar that includes navigation items and a main section that includes elements for adding and viewing tasks.

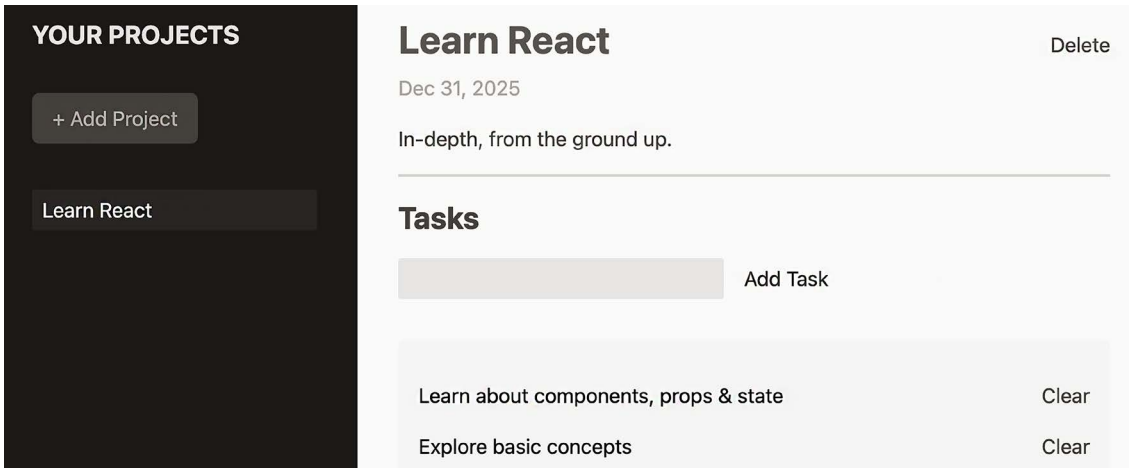


Figure 2.1: An example task management screen with sidebar and main area

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:

- The sidebar and its navigation items
- The main page area
- In the main area, the header with the title and due date
- A form for adding tasks
- A list of tasks

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 it 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 Node.js.

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 on 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 handleSubmit() {
    setIsSubmitted(true);
  };

  return (
```

```
    <button onClick={handleSubmit}>
      { 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.jsx`, 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. `.jsx` is used as an extension since the file contains JSX code. Vite enforces the usage of `.jsx` as a file extension if you're writing JSX code – storing such code in `.js` files is not allowed in Vite projects (even though it might work in other React project setups).

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.jsx';

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. Theoretically, in Vite-based projects, you could omit the file extension (`.jsx` in this case) in the `import` statement. However, it might be a good idea to include the extension since that's in line with standard JavaScript. When importing from third-party packages (like `useState` from the `react` package), no file extension is added though – you just use the package name. `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 importing 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()`). However, 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, components are 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 in *Chapter 4*, *Working with Events and State*). However, in late 2018, a new concept was introduced: **React Hooks**. This allows you to perform all operations and tasks with functional components. Consequently, while still supported by React, class-based components are on their way out and are 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., `handleSubmit`, typically written in **camelCase**)
- 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 consequently **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 `handleSubmit` 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 an 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 of non-JSX code being returned 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 an 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 for that reason, 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` and `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 `main.jsx` 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 (`main.jsx`) typically looks like this:

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

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

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 used.

While omitted here, strict mode will be covered in *Chapter 10*, *Behind the Scenes of React and Optimization Opportunities*. If you want to learn more about it right now, you can delve into the official documentation: `https://react.dev/reference/react/StrictMode`. Just be aware that some of the effects triggered by strict mode will be easier to understand after you've read more of this book.

To follow along smoothly then, cleaning up a newly created `main.jsx` 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. Hence, 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 `npm create vite@latest`, this will be the case. You can find a `<div id="root">` element in the `index.html` file in the root project folder.

This `index.html` file is a relatively empty file that 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, as a result, 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 web page.

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.jsx` 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 that is evaluated by React and translated into actual DOM-manipulating instructions.

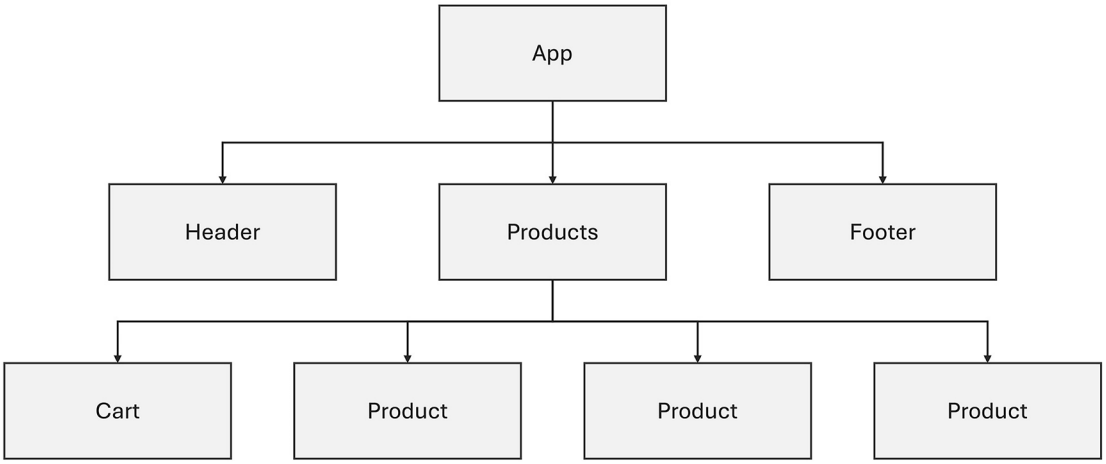


Figure 2.2: Nested React components form a component tree

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 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('root'));
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('root'));
```

This code will instruct React and ReactDOM to perform the following DOM operations:

1. Create a element
2. Inside that , create two child elements: <h2> and <p>
3. Set the text content of the <h2> element to 'Hello World!'
4. Set the text content of the <p> element to 'Welcome to this book!'
5. Insert the , with its children, into the already-existing DOM element, which has the ID 'root'

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 into and what the actual code that React works with 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: create component functions – lots of component functions.

But, ultimately, if you were to merge all JSX code into just one big snippet of JSX code, as shown in the last example, you would end up with a chunk of JSX code that includes only standard HTML elements like , <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(…)).