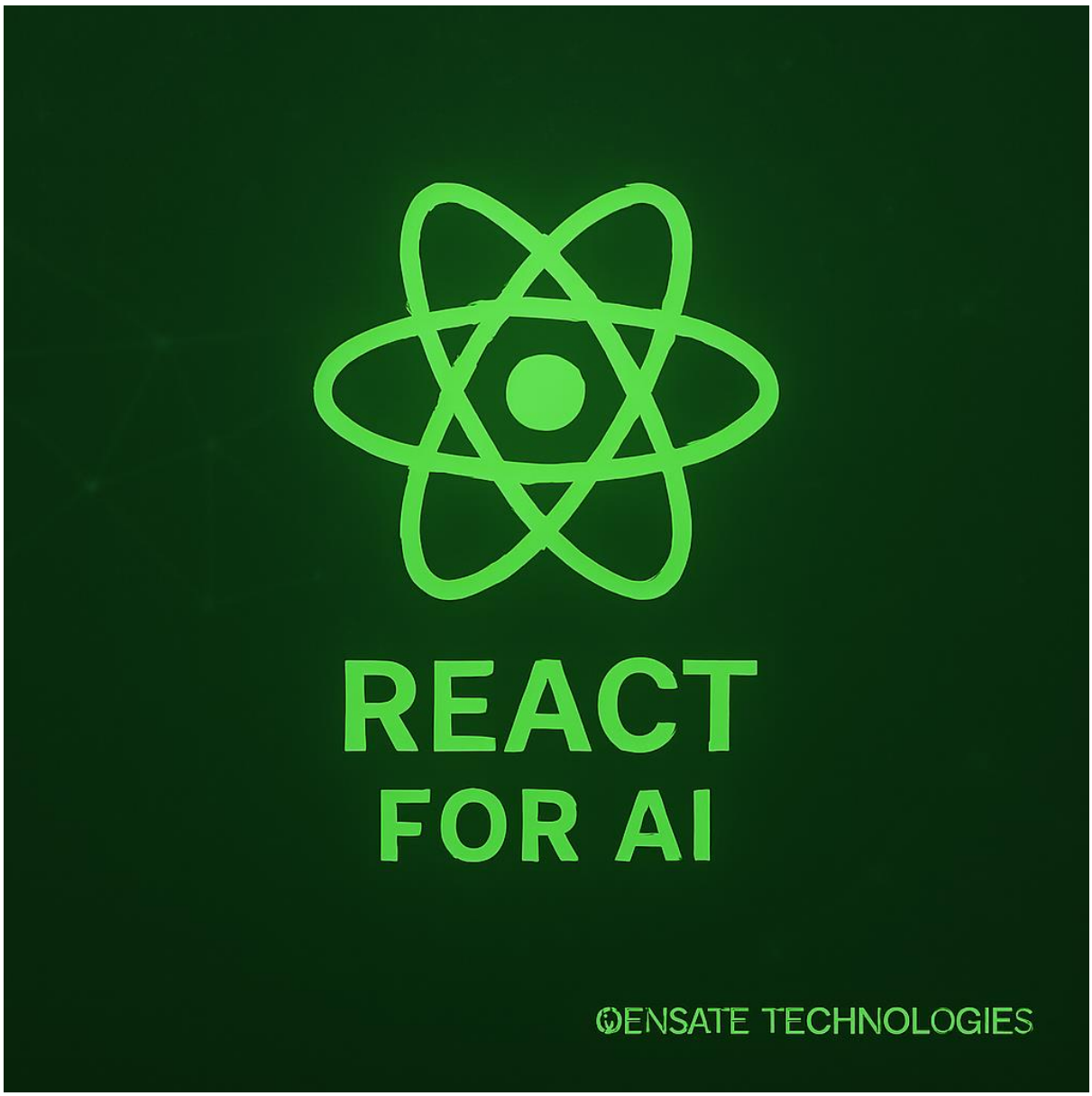


REACT

WEB DEVELOPMENT

SECTION -1

CHAPTER -1

React – What and Why

Learning Objectives

- 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) is one of the **most popular frontend JavaScript libraries**

The creators call it "The library for web and native user interfaces."

First, it's important to understand that React is a 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 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?

A library is 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 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.

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);
```

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?

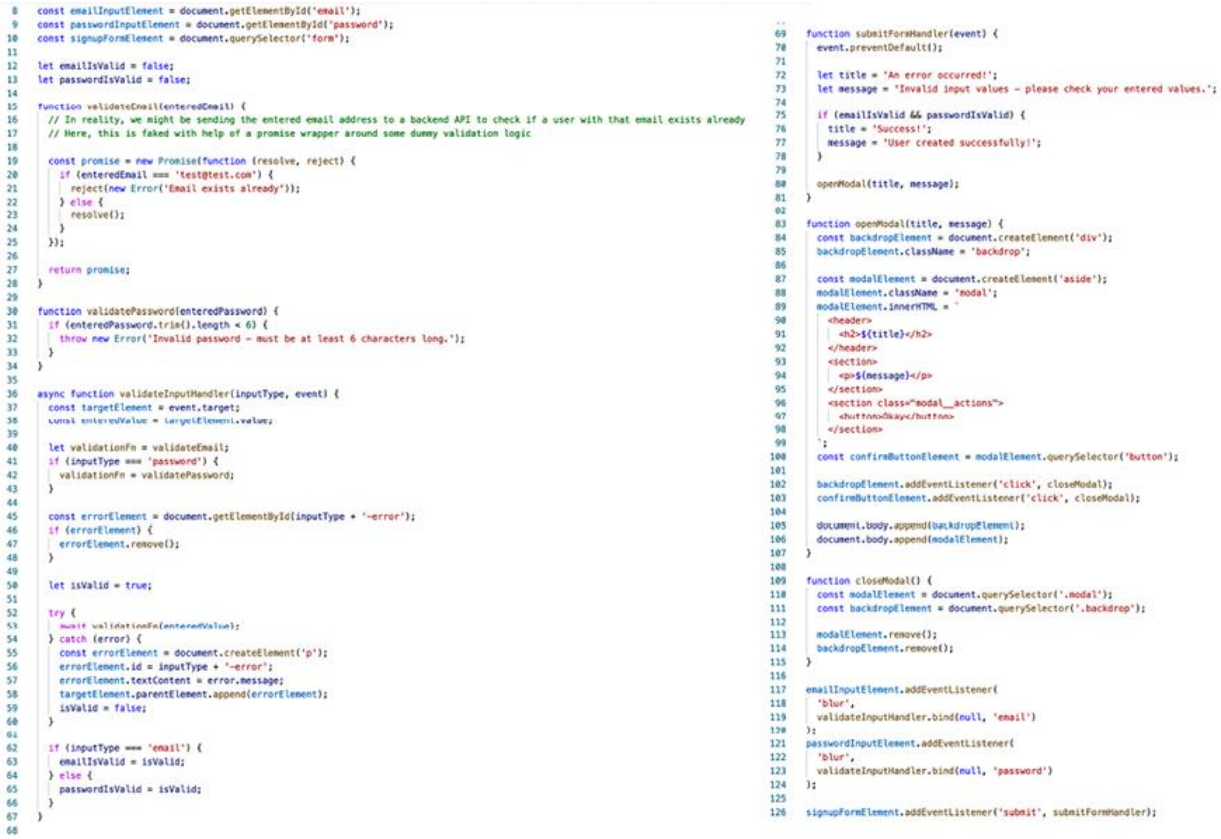
This shouldn't be too surprising because that is how most programming languages work: you define a series of steps that must be executed in order. It's an approach that makes a lot of sense because the order of code execution shouldn't be random or unpredictable.

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).

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.



```
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.

React and Declarative Code

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

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)

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

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

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)

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.

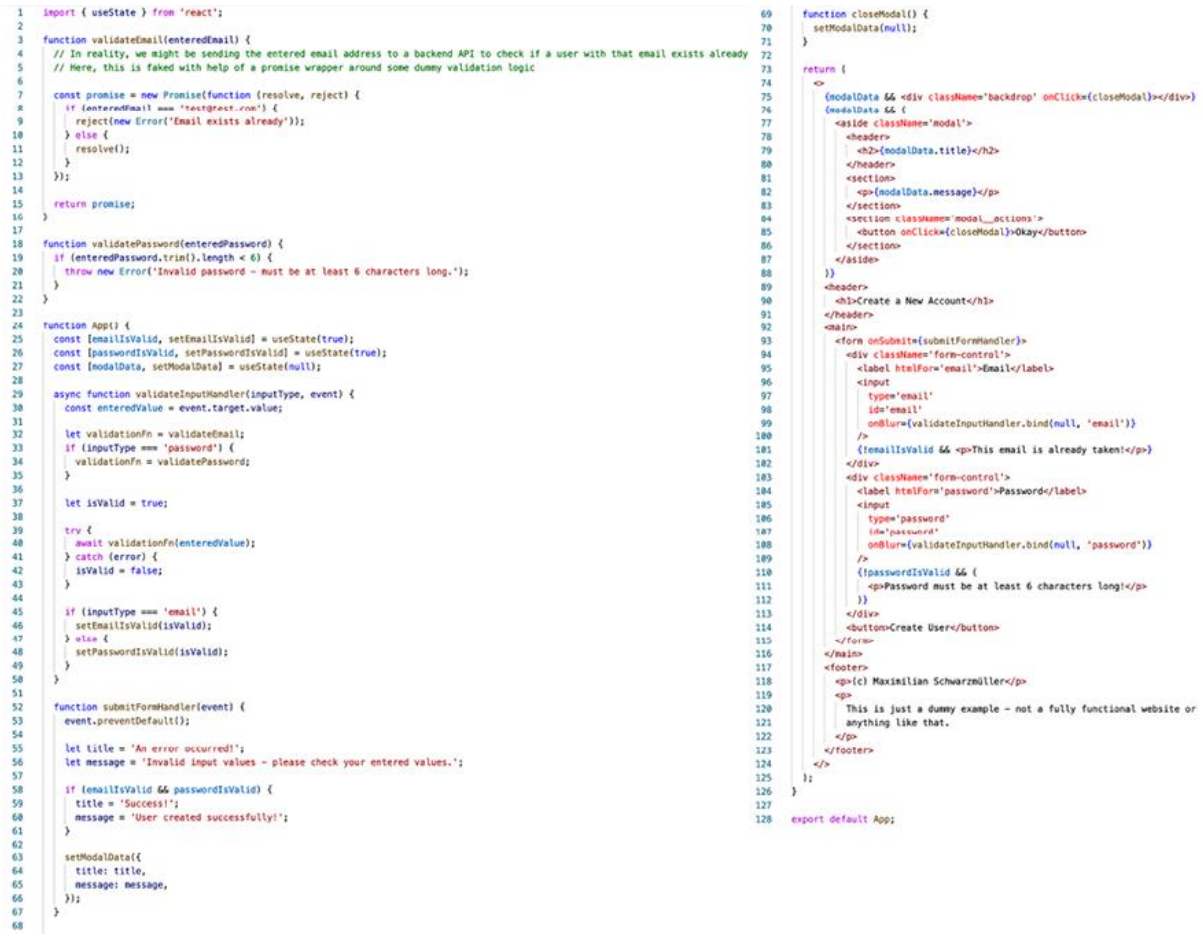
In the preceding example, outputText is some state managed by React. In the code, the updateTextHandler function is triggered upon a click, and the outputText state value is set to a new string value ('Text was changed!') with the help of the setOutputText function. The exact details of what's going 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 ({outputText}), 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.



```
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.

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

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.

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 , 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.

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**.

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.

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.

CHAPTER -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**

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.

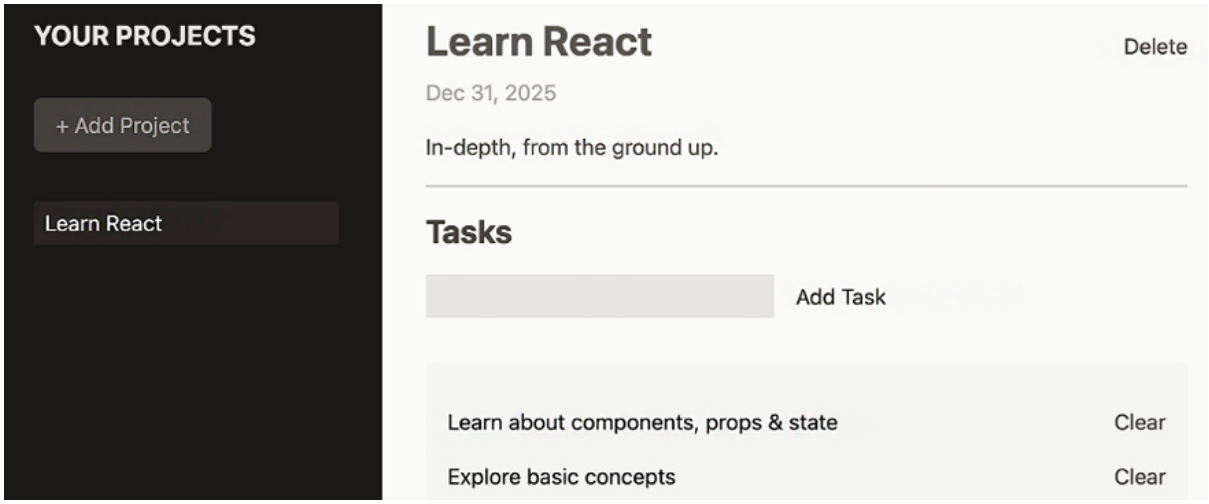


Figure 2.1: An example task management screen with sidebar and main area

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 course, 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. However, in late 2018, a new concept was introduced: **React Hooks**. This allows you to perform all operations and tasks with functional components.

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.

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 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.

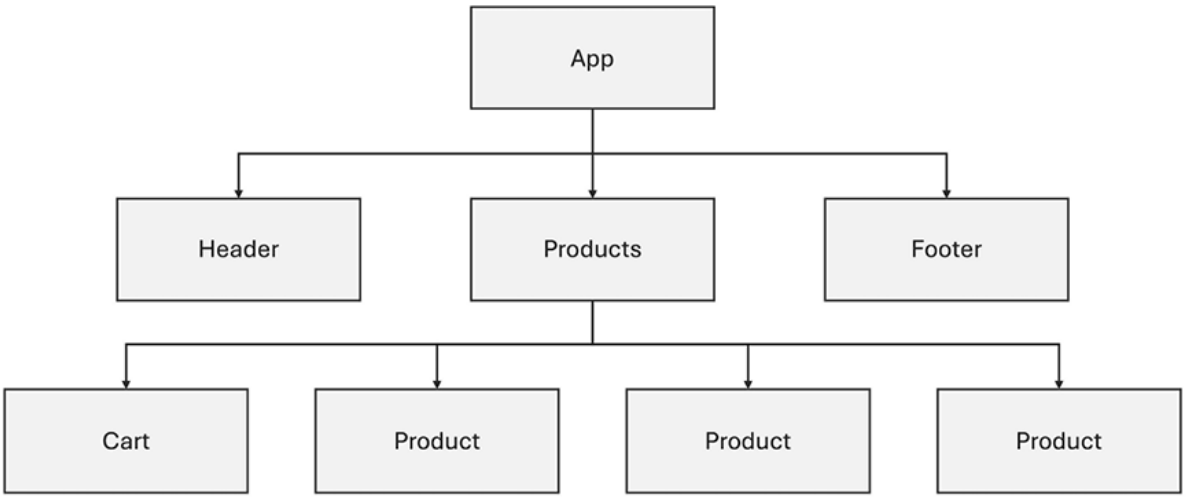


Figure 2.2: Nested React components form a component tree

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(…)).

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 , 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 .jsx file and it's not HTML markup. Instead, it's this special JSX code. It is important to keep this in mind.

Accordingly, these 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 consequently 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 course 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 hard rule 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 function as a 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 , 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 the folder name is up to you, but it should be somewhere inside the src/ folder. Using a folder named components/ is the standard though.

Whereas it is the standard to use **PascalCase** for the component functions, there is no general default regarding 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.jsx. Nonetheless, you will also encounter many React projects where components are stored in files that follow the **kebab-case** naming convention. (All lowercase and multiple words are combined into a single word via a dash). With this convention, component functions could be stored in files named submit-button.jsx, 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 it in basically every React project you explore, no matter whether you are using React for the browser or 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 run dev 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 modern React projects, it gets transformed to rather complex, unintuitive code that looks something like this:

```
function Ld() {
  return St.jsx('p', { children: 'Welcome to this book!' });
}
```

Of course, this code is not very developer-friendly. It's not the kind of code you would write. Instead, it's the code produced by Vite (i.e., by the underlying build process) for the browser to execute.

But you could, in theory, write code like this instead of using JSX—if, for some reason, you wanted to avoid writing JSX code. React has a built-in method you can use instead of JSX: you can use React's createElement(…) method.

Here's a concrete example, first in JSX:

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

Instead of using JSX, you could also write this component code like this:

```
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 the **virtual DOM**). 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.

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 anyway, 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 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 function calls behind the scenes.

JSX Elements Are Treated Like Regular JavaScript Values

Because JSX is just syntactical sugar that gets transformed, 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 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 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 (for instance, ones created via Vite), 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>. Since some React projects might not support the shorter <>...</> syntax, 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 by the build process 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.

JSX Elements Must Have a Closing Tag

Another important rule related to JSX elements is that they must always have a closing tag. Therefore, JSX elements 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.

Moving Beyond Static Content

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

But most websites, of course, need to output dynamic content that may change (e.g., due to user input). Similarly, you'll have a hard time finding lots of websites without any images.

Thus, as a React developer, it's important to know how to output dynamic content (and what "dynamic content" actually means) and how to display images in a React app.

Outputting Dynamic Content

React requires **state** concept 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 truly 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. However, it's worth noting that a few value types can't be used for outputting a value in JSX. For example, trying to output a JavaScript object in JSX will cause an error.

It's also worth noting that you're not limited to outputting dynamic content between element tags. Instead, you can also set dynamic values for attributes:

```
function App() {
  const userName = 'Max';
  return <input type="text" value={userName} />;
};
```

Rendering Images

Most websites do not just display plain text. Instead, you often need to render images as well.

Of course, when working with React, you can use the default <img /> element like in any other web project. But there are two important things to keep in mind when displaying images in React projects:

1. <img /> must be a self-closing tag.
2. When displaying local images stored inside of the src/ folder, you must **import** them into your .jsx files.

As explained above, in the JSX elements must have a closing tag section, you can't have void JSX elements, i.e., elements without any closing tag.

In addition, when outputting locally stored images (i.e., images stored in the project's src/ folder, not on some remote server), you typically don't set a string path to the image in your code.

You might be used to outputting images like this:

```
<img src="assets/images/wave.jpg">
```

But React projects (e.g., when created with Vite) do involve some kind of build process. In most projects, **the final project structure that will be deployed onto a server will look quite different from the project structure you work on during development**.

That being the case, if you store an image in the src/assets folder in a Vite-based React project, and you use that as a path (), the image will not load on the deployed website. It will not load there because the deployable folder structure will **not contain** a src/assets folder anymore.

Indeed, you can get an idea of the production-ready folder structure by running npm run build. This will build the project for deployment and produce a new dist folder in your project directory. It's the content of that dist folder that will be deployed onto some server. If you inspect that folder, you won't find a src folder in there.

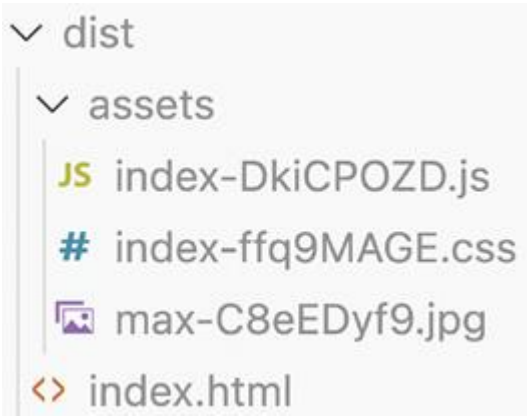


Figure 2.3: The dist folder contains a different structure

Put in other words: You can't tell the exact path of a locally stored image in advance. That's why you should import the image file into your .jsx file. As a result, you'll get a **string value** that will contain

the actual path (which will work in production). This value can then be set as a dynamic value for the src attribute of the <img /> element:

```
import myImage from './assets/my-image.png';

function App() {

  return <img src={myImage} />;

};
```

This might look strange at first, but it is code that will work in pretty much all React projects. Behind the scenes, this import gets analyzed by the underlying build process. The import statement then gets removed, and the image path is hardcoded into the production-ready output code (i.e., the code that's stored in the dist folder).

There is one **important exception though**: if you store an image file (or, actually, any asset) in the public/ folder of your project, you can directly reference its path.

For example, a demo.jpg image file stored in public/images/demo.jpg can be rendered and displayed like this:

```
function App() {

  return <img src="/images/demo.jpg" />;

};
```

This works because the contents of the public/ folder are simply copied into the dist/ folder. Unlike the src/ folder and its nested files, the public/ folder files skip the transpilation step.

Please note that the public folder name itself is not part of the paths referenced—it's src="/images/demo.jpg", not src="/public/images/demo.jpg".

Which approach should you use then? Store images in src/ or public/?

For most images, src/ is a sensible choice since the pre-processing step assigns a unique file name to each imported file. As a result, files can be cached more efficiently once the application is deployed.

Any files imported in the root index.html file, or files where the file name must never change (e.g., because it's also referenced by some other app, running on some other server) should typically go into the public/ folder.

Thus, in most cases, when outputting images that are stored locally in your project, you should store them in the src/ folder and then import them into your JSX files. When using images that are stored on some remote server, you would instead use the full image URL:

```
function App() {

  return <img src="https://some-server.com/my-image.jpg" />;

};
```

Bookmark

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?

As mentioned earlier, 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.jsx';

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 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 that 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 letters when being used in JSX code (typically, PascalCase naming is used)
- 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., {someText})
- Images can be rendered by referencing their paths (if stored remotely or in the public/ folder) or by importing the image files into JSX files and outputting them with the dynamic content syntax
- 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 upon this key knowledge and explains how you can make components reusable by making them configurable.

Before you continue, you can also practice what you have learned up to 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 questions below..

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.

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 in the previous chapter (i.e., create it via npm create vite@latest <your-project-name> and run npm run dev to start the development server) and edit the App.jsx 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 npm create vite@latest <project>..
2. Edit the App.jsx 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 can use the styles in the index.css file in the starting project snapshot to apply some styling.
3. Also, store an image in the src/assets folder and output it in the App component.

1. Create a new React project by running npm create vite@latest my-app. Choose "React" and "JavaScript".

You can replace my-app with any name of your choice, and you can run this command in any fitting place on your system (e.g., on your desktop).

cd into the newly created project (cd my-app) and install the required dependencies via npm install.

Start the development web server by running npm run dev inside the created project folder.

2. Open the project with any code editor of your choice—for example, with Visual Studio Code (https://code.visualstudio.com/).
3. Open the App.jsx file and replace the existing JSX code that is returned with JSX code that structures and contains the information about yourself that you want to output. Also import and output an image.

import maxImg from './assets/jobs.jpg';

```
function App() {
  return (
    <>
      <header>
        <img src={maxImg} alt="An image of Max" />
        <h1>Mr Max</h1>
        <p>Web developer, online course instructor & book author</p>
      </header>
      <main>
        <p>Right now, I am 35 years old and I live in Los Angeles.</p>
        <p>
          My full name is Maximilian and I am a web developer as
          well as top-rated, bestselling online course instructor.
        </p>
      </main>
    </>
  );
}

export default App;
```

Activity 2.2: Creating a React App to Log Your Goals for This Course

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 the main goals. In addition, you can add an extra header component that contains the main title for the web page.

The steps to complete this activity are as follows:

1. Create a new React project via npm create vite@latest <project
2. Inside the new project, create a components folder that 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).

Apply any style of your choice. You can also use the index.css file that's part of the starting project snapshot for inspiration.

You should get the following output in the end:

Figure 2.5: The final page output, showing a list of goals

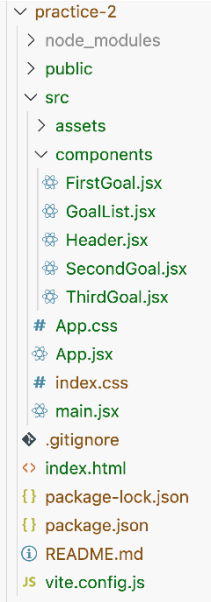
1. Create a new React project by running npm create vite@latest my-app. Choose "React" and "JavaScript".

You can replace my-app with any name of your choice, and you can run this command in any fitting place on your system (e.g., on your desktop). Start the development web server by running npm run dev inside the created project folder.

2. Create a new src/components folder in the project.

3. In the /src/components folder, create multiple component files—for example, FirstGoal.jsx, SecondGoal.jsx, ThirdGoal.jsx, GoalList.jsx and Header.jsx.

Your project folder should now look like this:



4. Edit the individual goal component files (FirstGoal.jsx, etc.) and define and export component functions inside of them.

Every component function should return a list item with any JSX markup of your choice and the goal title and text as main content.

Here's an example for the first goal:

```
function FirstGoal() {
 return (
  <li>
   <article>
    <h2>Teach React in a highly-understandable way</h2>
    <p>
     I want to ensure, that you get the most out of this book
     and you learn all about React!
```

```
      </p>
    </article>
   </li>
  );
}


export default FirstGoal;
```

5. In the GoalList.jsx file, define and export a GoalList component function and import the individual components. Thereafter, return JSX code that renders an unordered list (<ul>) with the custom goal components as list items:

```
import FirstGoal from './FirstGoal.jsx';
import SecondGoal from './SecondGoal.jsx';
import ThirdGoal from './ThirdGoal.jsx';


function GoalList() {
  return (
    <ul>
      <FirstGoal />
      <SecondGoal />
      <ThirdGoal />
    </ul>
  );
}
```

```
export default GoalList;
```

6. In the Header.jsx file, define and export a Header component and return some header JSX markup:

```
function Header() {
  return (
    <header>
      <h1>My Goals For This Book</h1>
```

```
    </header>
  );
}


export default Header;
```

7. Import the GoalList and Header components into the App.jsx file and replace the default JSX code with your own JSX code that renders these two components:

```
import GoalList from './components/GoalList.jsx';

import Header from './components/Header.jsx';


function App() {
  return (
    <>
      <Header />
      <main>
        <GoalList />
      </main>
    </>
  );
}


export default App;
```

CHAPTER-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 can 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).

Can Components Do More?

However, so far, 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 just configured 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 shown in the following snippet:

```
<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 and, therefore, need props and some components that are unique and might not need props.

The "how" part of the question 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 want 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.

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).

Consuming Props in a Component

How can you get access to the prop values passed into a component, when writing that component's inner code?

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 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 shown 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, these shouldn't be defined inside the GoalItem component anyway, as 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 is 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 and that, instead, React will call these functions on your behalf. And since React calls these functions, it can pass extra arguments into them 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. However, 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 (such as the title or id) 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 (
{props.title}).

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 that applies to 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 indeed 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 to 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, passing 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>
3. 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>`.
4. While you have no choice when working with regular HTML elements (you can't 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 the internal component code.

5. 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"`.
6. 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, 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 will absolutely also have components that are only used once in the entire app and still take advantage of props. As previously mentioned, it depends on the exact use case and component.

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:

```
<GoalItem title="A book" days={10} id="g1" />
// or
const goalData = {title: 'A book', days: 10, id: 'g1'};
< GoalItem data={ goalData } />
```

```
function GoalList() {
 const goalData = {title: 'A book', days: 10, id: 'g1'};
 return (

  <ul>
   <GoalItem  data={goalData}  >
    I want to ensure, that you get the most out of this book and you learn
    all about React!
```

```
  </GoalItem>
);
}
export default GoalList;
```

GoalList.jsx

```
function GoalItem(props) {//props will be an object containing data passed from GoalList
  // const { title, id } = props; // destructuring props
  return (
 <li>
     <article>
       <h2>{props.data.title}</h2>
       <h2>{props.data.days} days</h2>
       <h2>{props.data.id}</h2>
       <p>{props.children}</p>
     </article>
   </li>
 );
}
export default GoalItem;
```

GoalItem.jsx

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 variables or constants in a single step:

```
const user = {name: 'Max', age: 29};

const {name, age} = user; // <-- 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 GoalItem({title, days, id}) { // destructuring in action
  … // title, days, id are now available as variables inside this function
}
```

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

Let's look more one more example

Passing props to a component

Props are the information that you pass to a JSX tag. For example, className, src, alt, width, and height are some of the props you can pass to an <img>:

```
function Avatar() {
  return (
    <img
      className="avatar"
      src="https://i.imgur.com/1bX5QH6.jpg"
      alt="Lin Lanying"
      width={100}
      height={100}
    />
  );
}
export default function Profile() {
  return (
    <Avatar />
  );
}
```

The props you can pass to an <img> tag are predefined (ReactDOM conforms to the HTML standard). But you can pass any props to *your own* components, such as <Avatar>, to customize them. Here's how!

In this code, the Profile component isn't passing any props to its child component, Avatar

You can give Avatar some props in two steps.

Step 1: Pass props to the child component

First, pass some props to Avatar. For example, let's pass two props: person (an object), and size (a number):

```
export default function Profile() {
  return (
    <Avatar
      person={{ name: 'Lin Lanying', imageId: '1bX5QH6' }}
      size={100}
    />
  );
}
```

If double curly braces after person= confuse you, recall they're merely an object inside the JSX curlies.

Now you can read these props inside the Avatar component.

Step 2: Read props inside the child component

You can read these props by listing their names person, size separated by the commas inside ({ and }) directly after function Avatar. This lets you use them inside the Avatar code, like you would with a variable.

```
function Avatar({ person, size }) {
  // person and size are available here
}
```

Add some logic to Avatar that uses the person and size props for rendering, and you're done.

Now you can configure Avatar to render in many different ways with different props. Try tweaking the values!

Utils.jsx

```
export function getImageUrl(person, size = 's') {
  return (
    'https://i.imgur.com/' +
    person.imageId +
    size +
    '.jpg'
  );
}
```

```
import { getImageUrl } from './utils.js';
```

```
function Avatar({ person, size }) {
  return (
    <img
      className="avatar"
      src={getImageUrl(person)}
      alt={person.name}
      width={size}
      height={size}
    />
  );
}

export default function Profile() {
  return (
    <div>
```

```
    <Avatar
      size={100}
      person={{
        name: 'Katsuko Saruhashi',
        imageId: 'YfeOqp2'
      }}
    />
    <Avatar
      size={80}
      person={{
        name: 'Aklilu Lemma',
        imageId: 'OKS67lh'
      }}
    />
    <Avatar
      size={50}
      person={{
        name: 'Lin Lanying',
        imageId: '1bX5QH6'
      }}
    />
  </div>
 );
}
```

Props let you think about parent and child components independently. For example, you can change the person or the size props inside Profile without

having to think about how Avatar uses them. Similarly, you can change how the Avatar uses these props, without looking at the Profile.

You can think of props like “knobs” that you can adjust. They serve the same role as arguments serve for functions—in fact, props *are* the only argument to your component! React component functions accept a single argument, a props object:

```
function Avatar(props) {
  let person = props.person;
  let size = props.size;
  // ...
}
```

Usually you don’t need the whole props object itself, so you destructure it into individual props.

Note

Don’t miss the pair of { and } curlies inside of (and) when declaring props:

```
function Avatar({ person, size }) {
  // ...
}
```

This syntax is called “destructuring” and is equivalent to reading properties from a function parameter:

```
function Avatar(props) {
  let person = props.person;
  let size = props.size;
  // ...
}
```

Specifying a default value for a prop

If you want to give a prop a default value to fall back on when no value is specified, you can do it with the destructuring by putting = and the default value right after the parameter:

```
function Avatar({ person, size = 100 }) {
  // ...
}
```

Now, if <Avatar person={...} /> is rendered with no size prop, the size will be set to 100.

The default value is only used if the size prop is missing or if you pass size={undefined}. But if you pass size={null} or size={0}, the default value will **not** be used.

Spreading Props

Forwarding props with the JSX spread syntax

Sometimes, passing props gets very repetitive:

```
function Profile  { person, size,thickness, color, borderRadius }) {
  return (
    <div className="card">
<Avatar
        size={size}
        person={person}
        thickness={thickness}
        color={color}
        borderRadius={borderRadius}
      />
    </div>
  );
```

```
}
```

There's nothing wrong with repetitive code—it can be more legible. But at times you may value conciseness. Some components forward all of their props to their children, like how this Profile does with Avatar. Because they don't use any of their props directly, it can make sense to use a more concise **"spread"** syntax:

```
function Profile(props) {
  return (
    <div className="card">
      <Avatar {...props} />
    </div>
  );
}
```

This forwards all of Profile's props to the Avatar without listing each of their names.

Use spread syntax with restraint. If you're using it in every other component, something is wrong. Often, it indicates that you should split your components and pass children as JSX

Passing JSX as children

It is common to nest built-in browser tags:

```
<div>
  <img />
</div>
```

Sometimes you'll want to nest your own components the same way:

```
<Card>
  <Avatar />
</Card>
```

When you nest content inside a JSX tag, the parent component will receive that content in a prop called children. For example, the Card component below will receive a children prop set to <Avatar /> and render it in a wrapper div:

Avatar.jsx

```
import { getImageUrl } from './utils.js';

export default function Avatar({ person, size }) {
  return (
    <img
      className="avatar"
      src={getImageUrl(person)}
      alt={person.name}
      width={size}
      height={size}
    />
  );
}
```

Profile.jsx

```
import Avatar from './Avatar.js';

function Card({ children }) {
  return (
    <div className="card">
      {children}
    </div>
```

```
  );
}


export default function Profile() {
  return (
    <Card>
      <Avatar
        size={100}
        person={{
          name: 'Katsuko Saruhashi',
          imageId: 'YfeOqp2'
        }}
      />
    </Card>
  );
}
```

Try replacing the <Avatar> inside <Card> with some text to see how the Card component can wrap any nested content. It doesn't need to "know" what's being rendered inside of it. You will see this flexible pattern in many places.

You can think of a component with a children prop as having a "hole" that can be "filled in" by its parent components with arbitrary JSX. You will often use the children prop for visual wrappers: panels, grids, etc.

How props change over time

The Clock component below receives two props from its parent component: color and time. (The parent component's code is omitted because it uses state, which we won't dive into just yet.)

Clock.jsx

```
export default function Clock({ color, time }) {
  return (
    <h1 style={{ color: color }}>
      {time}
    </h1>
  );
}
```

Try changing the color in the select box below:

This example illustrates that **a component may receive different props over time.** Props are not always static! Here, the time prop changes every second, and the color prop changes when you select another color. Props reflect a component's data at any point in time, rather than only in the beginning.

However, props are immutable—a term from computer science meaning "unchangeable". When a component needs to change its props (for example, in response to a user interaction or new data), it will have to "ask" its parent component to pass it *different props*—a new object! Its old props will then be cast aside, and eventually the JavaScript engine will reclaim the memory taken by them.

Try out some challenges

Challenges are in the folder Challenges & Exercises\Props In Components

More on 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 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 use 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}>
```

An alternative, more common pattern uses yet another JavaScript feature: the **rest property**. That's a JavaScript pattern that allows you to group properties that have not been destructured into a new object (which then only contains those properties).

```
function Link({children, ...props}) {

  return <a {...props} target="_blank" rel="noopener
noreferrer">{children}</a>;

};
```

In this example, when destructuring props, only the children prop is destructured; the other ones are stored in a new object named props. The syntax is very similar to the spread operator syntax: you use three dots (...). But here, you use the operator in front of the property that should contain all remaining properties. Therefore, it's the place where you use that operator that defines what it does.

You can then use that rest property (props in the example) like any other object. In the example above, it's again used to spread its properties as props onto the <a> element.

Using this pattern allows you to use the Link component in a more natural way, where you don't have to create and use a separate configuration object:

<Link href="https://google.com">Can you google that for me?</Link>

These behaviors and patterns 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**.

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 11, Working with Complex States.

Bookmark

Summary and Key Takeaways

- Props are a key React concept that make 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.
- You can use JavaScript features like destructuring, the rest property, or the spread operator to write concise, flexible code.
- 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 at https://github.com/mschwarzmueller/book-react-key-concepts-e2/blob/03-components-props/exercises/questions-answers.md:

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

Bookmark

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 so far.

Activity 3.1: Creating an App to Output Your Goals for This Book

This activity builds upon Activity 2.2, Creating 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 that is accessible at the following

link: https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/02-components-jsx/activities/practice-2.

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 (via props).
4. Set the detailed text description for every goal between the goal component's opening and closing tags.

The final user interface might look like this:

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

Note

You can find a full example solution here: https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/03-components-props/activities/practice-1.

CHAPTER-4

WORKING WITH EVENTS AND STATE

Learning Objectives

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

- Add user event handlers (for example, for reacting to button clicks) to React apps
- Update the **user interface** (**UI**) via a concept called **state**
- Build real dynamic and interactive UIs (that is, so that they are not static anymore)

Introduction

In the previous chapters, you learned how to build UIs with the help of React **components**. You also learned about **props**—a concept and feature that enables React developers to build and reuse configurable components.

These are all important React features and building blocks, but with these features alone, you would only be able to build static React apps (that is, web apps that never change). You would not be able to **change or update the content on the screen** if you only had access to those features. You also would not be able to **react to any user events and update the UI in response to such events** (for instance, to show an overlay window upon a button click).

Put in other words, you would not be able to build real websites and web **applications** if you were limited to just components and props.

Therefore, in this chapter, a brand-new concept is introduced: **state**. State is a React feature that allows developers to update internal data and trigger a UI update based on such data adjustments. In addition, you will learn how to react to user events such as button clicks or text being entered into input fields.

EVENTS

Responding to Events

React lets you add *event handlers* to your JSX. Event handlers are your own functions that will be triggered in response to interactions like clicking, hovering, focusing form inputs, and so on.

You will learn

- Different ways to write an event handler
- How to pass event handling logic from a parent component
- How events propagate and how to stop them

Adding event handlers

To add an event handler, you will first define a function and then pass it as a prop to the appropriate JSX tag. For example, here is a button that doesn't do anything yet:

```
export default function Button() {
  return (
    <button>
      I don't do anything
    </button>
  );
}
```

You can make it show a message when a user clicks by following these three steps:

1. Declare a function called handleClick *inside* your Button component.
2. Implement the logic inside that function (use alert to show the message).

3. Add onClick={handleClick} to the <button> JSX.

```
export default function Button() {
  function handleClick() {
    alert('You clicked me!');
  }

  return (
    <button onClick={handleClick}>
      Click me
    </button>
  );
}
```

You defined the handleClick function and then passed it as a prop to <button>. handleClick is an **event handler.** Event handler functions:

- Are usually defined *inside* your components.
- Have names that start with handle, followed by the name of the event.

By convention, it is common to name event handlers as handle followed by the event name. You'll often see onClick={handleClick}, onMouseEnter={handleMouseEnter}, and so on.

Alternatively, you can define an event handler inline in the JSX:

```
<button onClick={function handleClick() {
  alert('You clicked me!');
}}>
```

Or, more concisely, using an arrow function:

```
<button onClick={() => {
  alert('You clicked me!');
}}>
```

All of these styles are equivalent. **Inline event handlers** are convenient for **short functions**.

Notes

Functions passed to event handlers must be passed, not called. For example:

passing a function (correct)	calling a function (incorrect)
<button onClick={handleClick}>	<button onClick={handleClick()}>

The difference is subtle. In the first example, the handleClick function is passed as an onClick event handler. This tells React to remember it and only call your function when the user clicks the button.

In the second example, the () at the end of handleClick() fires the function *immediately* during rendering, without any clicks. This is because JavaScript inside the JSX { and } executes right away.

When you write code inline, the same pitfall presents itself in a different way:

passing a function (correct)	calling a function (incorrect)
<button onClick={() => alert('...')}>	<button onClick={alert('...')}>

Passing inline code like this won't fire on click—it fires every time the component renders:

// This alert fires when the component renders, not when clicked!

<button onClick={alert('You clicked me!')}>

If you want to define your event handler inline, wrap it in an anonymous function like so:

<button onClick={() => alert('You clicked me!')}>

Rather than executing the code inside with every render, this creates a function to be called later.

In both cases, what you want to pass is a function:

- <button onClick={handleClick}> passes the handleClick function.
- <button onClick={() => alert('...')}> passes the () => alert('...') function.

Reading props in event handlers

Because event handlers are declared inside of a component, they have access to the component's props. Here is a button that, when clicked, shows an alert with its message prop:

This lets these two buttons show different messages. Try changing the messages passed to them.

Passing event handlers as props

Often you'll want the parent component to specify a child's event handler. Consider buttons: depending on where you're using a Button component, you might want to execute a different function—perhaps one plays a movie and another uploads an image.

To do this, pass a prop the component receives from its parent as the event handler like so:

```
function Button({ onClick, children }) {
  return (
    <button onClick={onClick}>
      {children}
    </button>
  );
}
function PlayButton({ movieName }) {
  function handlePlayClick() {
```

```
    alert(`Playing ${movieName}!`);
  }

  return (
    <Button onClick={handlePlayClick}>
      Play "{movieName}"
    </Button>
  );
}

function UploadButton() {
  return (
    <Button onClick={() => alert('Uploading!')}>
      Upload Image
    </Button>
  );
}

export default function Toolbar() {
  return (
    <div>
      <PlayButton movieName="Kiki's Delivery Service" />
      <UploadButton />
    </div>
  );
}
```

Here, the Toolbar component renders a PlayButton and an UploadButton:

- PlayButton passes handlePlayClick as the onClick prop to the Button inside.
- UploadButton passes () => alert('Uploading!') as the onClick prop to the Button inside.

Finally, your Button component accepts a prop called onClick. It passes that prop directly to the built-in browser <button> with onClick={onClick}. This tells React to call the passed function on click.

If you use a design system, it's common for components like buttons to contain styling but not specify behavior. Instead, components like PlayButton and UploadButton will pass event handlers down.

Naming event handler props

Built-in components like <button> and only support browser event names like onClick. However, when you're building your own components, you can name their event handler props any way that you like.

By convention, event handler props should start with on, followed by a capital letter.

For example, the Button component's onClick prop could have been called onSmash:

```
function Button({ onSmash, children }) {
  return (
    <button onClick={onSmash}>
      {children}
    </button>
  );
}


export default function App() {
```

```
  return (
    <div>
      <Button onSmash={() => alert('Playing!')}>
        Play Movie
      </Button>
      <Button onSmash={() => alert('Uploading!')}>
        Upload Image
      </Button>
    </div>
  );
}
```

In this example, <button onClick={onSmash}> shows that the browser <button> (lowercase) still needs a prop called onClick, but the prop name received by your custom Button component is up to you!

When your component supports multiple interactions, you might name event handler props for app-specific concepts. For example, this Toolbar component receives onPlayMovie and onUploadImage event handlers:

```
export default function App() {
  return (
    <Toolbar
      onPlayMovie={() => alert('Playing!')}
      onUploadImage={() => alert('Uploading!')}
    />
  );
}


function Toolbar({ onPlayMovie, onUploadImage }) {
```

```
  return (
    <div>
      <Button onClick={onPlayMovie}>
        Play Movie
      </Button>
      <Button onClick={onUploadImage}>
        Upload Image
      </Button>
    </div>
  );
}

function Button({ onClick, children }) {
  return (
    <button onClick={onClick}>
      {children}
    </button>
  );
}
```

Notice how the App component does not need to know *what* Toolbar will do with onPlayMovie or onUploadImage. That's an implementation detail of the Toolbar. Here, Toolbar passes them down as onClick handlers to its Buttons, but it could later also trigger them on a keyboard shortcut. Naming props after app-specific interactions like onPlayMovie gives you the flexibility to change how they're used later.

Note

Make sure that you use the appropriate HTML tags for your event handlers. For example, to handle clicks, use <button onClick={handleClick}> instead of . Using a real browser <button> enables built-in browser behaviors like keyboard navigation. If you don't like the default browser styling of a button and want to make it look more like a link or a different UI element, you can achieve it with CSS. Learn more about writing accessible markup.

Event propagation

Event handlers will also catch events from any children your component might have. We say that an event **"bubbles"** or **"propagates"** up the tree: it starts with where the event happened, and then goes up the tree.

This contains two buttons. Both the *and* each button have their own onClick handlers. Which handlers do you think will fire when you click a button?

```
export default function Toolbar() {
 return (
  <div className="Toolbar" onClick={() => {
   alert('You clicked on the toolbar!');
  }}>
   <button onClick={() => alert('Playing!')}>
    Play Movie
   </button>
   <button onClick={() => alert('Uploading!')}>
```

```
      Upload Image
    </button>
   </div>
  );
}
```

Stopping propagation

Event handlers receive an event object as their only argument. By convention, it's usually called e, which stands for "event". You can use this object to read information about the event.

That event object also lets you stop the propagation. If you want to prevent an event from reaching parent components, you need to call e.stopPropagation() like this Button component does:

```
function Button({ onClick, children }) {
  return (
    <button onClick={e => {
      e.stopPropagation();
      onClick();
    }}>
      {children}
    </button>
  );
}

export default function Toolbar() {
  return (
```

```
  <div className="Toolbar" onClick={() => {
    alert('You clicked on the toolbar!');
  }}>
    <Button onClick={() => alert('Playing!')}>
      Play Movie
    </Button>
    <Button onClick={() => alert('Uploading!')}>
      Upload Image
    </Button>
  </div>
 );
}
```

When you click on a button:

1. React calls the onClick handler passed to <button>.
2. That handler, defined in Button, does the following:
    - Calls e.stopPropagation(), preventing the event from bubbling further.
    - Calls the onClick function, which is a prop passed from the Toolbar component.
3. That function, defined in the Toolbar component, displays the button's own alert.
4. Since the propagation was stopped, the parent 's onClick handler does *not* run.

As a result of e.stopPropagation(), clicking on the buttons now only shows a single alert (from the <button>) rather than the two of them (from the <button> and the parent toolbar). Clicking a button is not the same thing

as clicking the surrounding toolbar, so stopping the propagation makes sense for this UI.

Passing handlers as alternative to propagation

Notice how this click handler runs a line of code *and then* calls the onClick prop passed by the parent:

```
function Button({ onClick, children }) {
  return (
    <button onClick={e => {
      e.stopPropagation();
      onClick();
    }}>
      {children}
    </button>
  );
}
```

You could add more code to this handler before calling the parent onClick event handler, too. This pattern provides an *alternative* to propagation. It lets the child component handle the event, while also letting the parent component specify some additional behavior. Unlike propagation, it's not automatic. But the benefit of this pattern is that you can clearly follow the whole chain of code that executes as a result of some event.

If you rely on propagation and it's difficult to trace which handlers execute and why, try this approach instead.

Preventing default behavior

Some browser events have default behavior associated with them. For example, a <form> **submit** event, which happens when a button inside of it is clicked, will reload the whole page by default:

```
export default function Signup() {
  return (
```

```
  <form onSubmit={() => alert('Submitting!')}>
    <input />
    <button>Send</button>
  </form>
 );
}
```

You can call e.preventDefault() on the event object to stop this from happening:

```
export default function Signup() {
 return (
  <form onSubmit={e => {
    e.preventDefault();
    alert('Submitting!');
  }}>
    <input />
    <button>Send</button>
  </form>
 );
}
```

Don’t confuse e.stopPropagation() and e.preventDefault(). They are both useful, but are unrelated:

- e.stopPropagation() stops the event handlers attached to the tags above from firing.
- e.preventDefault() prevents the default browser behavior for the few events that have it.

Recap

- You can handle events by passing a function as a prop to an element like <button>.
- Event handlers must be passed, **not called!** onClick={handleClick}, not onClick={handleClick()}.
- You can define an event handler function separately or inline.
- Event handlers are defined inside a component, so they can access props.
- You can declare an event handler in a parent and pass it as a prop to a child.
- You can define your own event handler props with application-specific names.
- Events propagate upwards. Call e.stopPropagation() on the first argument to prevent that.
- Events may have unwanted default browser behavior. Call e.preventDefault() to prevent that.
- Explicitly calling an event handler prop from a child handler is a good alternative to propagation.

STATE

What's the Problem?

As outlined previously, in the previous chapters, there is a problem with all React apps and sites you might be building: they're **static**. The UI can't change.

To understand this issue a bit better, take a look at a typical React component, as you are able to build it up to this point in the book:

```
function EmailInput() {
  return (
    <div>
      <input placeholder="Your email" type="email" />
      <p>The entered email address is invalid.</p>
    </div>
  );
};
```

This component might look strange though. Why is there a <p> element that informs the user about an incorrect email address?

Well, the goal might be to show that paragraph only if the user did enter an incorrect email address. That is to say, the web app should wait for the user to start typing and evaluate the user input once the user is done typing (that is, once the input loses focus). Then, the error message should be shown if the email address is considered invalid (for example, an empty input field or a missing @ symbol).

But at the moment, with the React skills picked up thus far, this is something you would not be able to build. Instead, the error message would always be shown since there is no way of changing it based on user events and dynamic conditions. In other words, this React app is a static app, not dynamic. The UI can't change.

Of course, changing UIs and dynamic web apps are things you might want to build. Almost every website that exists contains some dynamic UI elements and features. Therefore, that's the problem that will be solved in this chapter.

How Not to Solve the Problem

How could the component shown previously be made more dynamic?

The following is one solution you could come up with (spoiler, the code won't work, so you don't need to try running it):

```
function EmailInput() {
  return (
    <div>
      <input placeholder="Your email" type="email" />
      <p></p>
    </div>
  );
};
const input = document.querySelector('input');
const errorParagraph = document.querySelector('p');
function evaluateEmail(event) {
  const enteredEmail = event.target.value;
  if (enteredEmail.trim() === '' || !enteredEmail.includes('@')) {
    errorParagraph.textContent = ' The entered email address is invalid.';
  } else {
```

```
    errorParagraph.textContent = '';
  }
};
input.addEventListener('blur', evaluateEmail);
```

This code won't work, because you can't select React-rendered DOM elements from inside the same component file this way. This is just meant as a dummy example of how you could try to solve this. That being said, you could put the code below the component function some place where it does execute successfully (for example, into a setTimeout() callback that fires after a second, allowing the React app to render all elements onto the screen).

Put in the right place, this code will add the email validation behavior described earlier in this chapter. Upon the built-in blur event, the evaluateEmail function is triggered. This function receives the event object as an argument (automatically, by the browser), and therefore the evaluateEmail function is able to parse the entered value from that event object via event.target.value. The entered value can then be used in an if check to conditionally display or remove the error message.

Note

All the preceding code that deals with the blur event (such as addEventListener) and the event object, including the code in the if check, is standard JavaScript code. It is not specific to React in any way.

But what's wrong with this code if it would work in some places of the overall application code?

It's imperative code! That means you are writing down step-by-step instructions on what the browser should do. You are not declaring the desired end state; you are instead describing a way of getting there; and it's not using React.

Keep in mind that React is all about controlling the UI and that writing React code is about writing declarative code—instead of imperative code

You could achieve your goal by introducing this kind of code, but you would be working against React and its philosophy (React's philosophy being that you declare your desired end states and let React figure out how to get there). A clear indicator of this is the fact that you would be forced to find the right place for this kind of code in order for it to work.

his is not a philosophical problem, and it's not just some weird hard rule that you should follow. Instead, by working against React like this, you will make your life as a developer unnecessarily hard. You are neither using the tools React gives you nor letting React figure out how to achieve the desired (UI) state.

That does not just mean that you spend time on solving problems you wouldn't have to solve. It also means that you're passing up possible optimizations React might be able to perform under the hood. Your solution is very likely not just leading to more work (that is, more code) for you; it also might result in a buggy result that could also suffer from suboptimal performance.

The example shown previously is a simple one. Think about more complex websites and web apps, such as online shops, vacation rental websites, or web apps such as Google Docs. There, you might have dozens or hundreds of (dynamic) UI features and elements. Managing them all with a mixture of React code and standard vanilla JavaScript code will quickly become a nightmare.

A Better Incorrect Solution

The naïve approach discussed previously doesn't work well. It forces you to figure out how to make the code run correctly (for example, by wrapping parts of it in some setTimeout() call to defer execution) and leads to your code being scattered all over the place (that is, inside of React component functions, outside of those functions, and maybe also in totally unrelated files). How about a solution that embraces React, like this:

```
function EmailInput() {
```

```
let errorMessage = '';
function evaluateEmail(event) {
  const enteredEmail = event.target.value;
  if (enteredEmail.trim() === '' || !enteredEmail.includes('@')) {
    errorMessage = ' The entered email address is invalid.';
  } else {
    errorMessage = '';
  }
};
const input = document.querySelector('input');
input.addEventListener('blur', evaluateEmail);
return (
  <div>
    <input placeholder="Your email" type="email" />
    <p>{errorMessage}</p>
  </div>
);
};
```

This code again would not work (even though it's technically valid JavaScript code). Selecting JSX elements doesn't work like this. It doesn't work because document.querySelector('input') executes before anything is rendered to the DOM (when the component function is executed for the first time). Again, you would have to delay the execution of that code until the first render cycle is over (you would therefore be once again working against React).

But even though it still would not work, it's closer to the correct solution.

It's closer to the ideal implementation because it embraces React way more than the first attempted solution did. All the code is contained in the

component function to which it belongs. The error message is handled via an errorMessage variable that is output as part of the JSX code.

The idea behind this possible solution is that the React component that controls a certain UI feature or element is also responsible for its **state** and **events**. You might identify two important keywords of this chapter here!

This approach is definitely going in the right direction, but it still wouldn't work for two reasons:

- Selecting the JSX <input> element via document.querySelector('input') would fail.
- Even if the input could be selected, the UI would not update as expected.

These two problems will be solved next—finally leading to an implementation that embraces React and its features. The upcoming solution will avoid mixing React and non-React code. As you will see, the result will be easier code where you have to do less work (that is, write less code).

Improving the Solution by Properly Reacting to Events

Instead of mixing imperative JavaScript code such as document.querySelector('input') with React-specific code, you should fully embrace React and its features.

Since listening to events and triggering actions upon events is an extremely common requirement, React has a built-in solution. You can attach event listeners directly to the JSX elements to which they belong.

The preceding example would be rewritten like this:

```
function EmailInput() {
  let errorMessage = '';
  function evaluateEmail(event) {
    const enteredEmail = event.target.value;
    if (enteredEmail.trim() === '' || !enteredEmail.includes('@')) {
```

```
    errorMessage = 'The entered email address is invalid.';
  } else {
    errorMessage = '';
  }
};
return (
  <div>
    <input
      placeholder="Your email"
      type="email"
      onBlur={evaluateEmail} />
    <p>{errorMessage}</p>
  </div>
);
};
```

This code still will not update the UI, but at least the event is handled properly.

The onBlur prop was added to the built-in input element. This prop is made available by React, just as all these base HTML elements (such as <input> and <p>) are made available as components by React. In fact, all these built-in HTML components come with their standard HTML attributes as React props (plus some extra props, such as the onBlur event handling prop).

React exposes all standard events that can be connected to DOM elements as onXYZ props (where XYZ is the event name, such as blur or click, starting with a capital character). You can react to the blur event by adding the onBlur prop. You could listen to a click event via the onClick prop. You get the idea.

These props require values to fulfill their job. To be precise, they need a pointer to the function that should be executed when the event occurs. In the

preceding example, the onBlur prop receives a pointer to the evaluateEmail function as a value.

Note

There's a subtle difference between evaluateEmail and evaluateEmail(). The first is a pointer to the function; the second actually executes the function (and yields the return value, if any).

By using these event props, the preceding example code will now finally execute without throwing any errors. You could verify this by adding a console.log('Hello'); statement inside the evaluateEmail function. This will display the 'Hello' text in the console of your browser developer tools, whenever the input loses focus:

```
function EmailInput() {
 let errorMessage = '';
 function evaluateEmail(event) {
  console.log('Hello');
  const enteredEmail = event.target.value;
  if (enteredEmail.trim() === '' || !enteredEmail.includes('@')) {
   errorMessage = 'The entered email address is invalid.';
  } else {
   errorMessage = '';
  }
 };
 return (
  <div>
   <input
    placeholder="Your email"
    type="email"
```

```
      onBlur={evaluateEmail} />
    <p>{errorMessage}</p>
  </div>
 );
};
```

Updating State Correctly

By now, you understand how to correctly set up event listeners and execute functions upon certain events. What's missing is a feature that forces React to update the visible UI on the screen and the content that is displayed to the app users.

That's where React's **state** concept comes into play. Like props, state is a key concept of React, but whereas props are about receiving external data inside a component, state is about managing and updating **internal data**. Most importantly, whenever state is updated, React goes ahead and updates the parts of the UI that are affected by the state change.

Here's how state is used in React (of course, the code will then be explained in detail afterward):

```
import { useState } from 'react';
function EmailInput() {
 const [errorMessage, setErrorMessage] = useState('');
 function evaluateEmail(event) {
  const enteredEmail = event.target.value;
  if (enteredEmail.trim() === '' || !enteredEmail.includes('@')) {
   setErrorMessage('The entered email address is invalid.');
  } else {
```

```
    setErrorMessage('');
  }
 };
 return (
  <div>
   <input
    placeholder="Your email"
    type="email"
    onBlur={evaluateEmail} />
   <p>{errorMessage}</p>
  </div>
 );
};
```

Compared to the example code discussed earlier in this chapter, this code doesn't look much different. But there is a key difference: the usage of the **useState()** Hook.

Hooks are another key concept of React. These are special functions that can only be used inside of React components

Hooks add special features and behaviors to the React components in which they are used. For example, the useState() Hook allows a component (and therefore, implicitly React) to set and manage some state that is tied to this component. React provides various built-in Hooks, and they are not all focused on state management.

The useState() Hook is an extremely important and commonly used Hook as it enables you to manage data inside a component, which, when updated, tells React to update the UI accordingly.

That is the core idea behind state management and this state concept: state is data, which, when changed, should force React to re-evaluate a component and update the UI if needed.

Using Hooks, such as useState(), is pretty straightforward: you import them from 'react' and you then call them like a function inside your component function. You call them like a function because, as mentioned, React Hooks are functions—just special functions (from React's perspective).

A Closer Look at useState()

How exactly does the useState() Hook work and what does it do internally?

By calling useState() inside a component function, you register some data with React. It's a bit like defining a variable or constant in vanilla JavaScript. But there is something special: *React will track the registered value internally, and whenever you update it, React will re-evaluate the component function in which the state was registered.*

React does this by checking whether the data used in the component changed. Most importantly, React validates whether the UI needs to change because of changed data (for example, because a value is output inside the JSX code). If React determines that the UI needs to change, it goes ahead and updates the real DOM in the places where an update is needed (for example, changing some text that's displayed on the screen). If no update is needed, React ends the component re-evaluation without updating the DOM.

The entire process starts with calling useState() inside a component. This creates a state value (which will be stored and managed by React) and ties it to a specific component. An initial state value is registered by simply passing it as a parameter value to useState(). In the preceding example, an empty string ('') is registered as a first value:

```
const [errorMessage, setErrorMessage] = useState('');
```

As you can see in the example, useState() does not just accept a parameter value. It also returns a value: an array with exactly two elements.

The preceding example uses **array destructuring**, which is a standard JavaScript feature that allows developers to retrieve values from an array and immediately assign them to variables or constants. In the example, the two elements that make up the array returned by useState() are pulled out of that array and stored in two constants (errorMessage and setErrorMessage). You don't have to use array destructuring when working with React or useState(), though.

Quick Reference

A sample destructuring with vanilla javascript

Destructuring comes in handy when a function returns an array:

```
<!DOCTYPE html>
<html>

<body>

<script>
function calculate(a, b) {
  const add = a + b;
  const subtract = a - b;
  const multiply = a * b;
  const divide = a / b;

  return [add, subtract, multiply, divide];
}

const [add, subtract, multiply, divide] = calculate(4, 7);

document.write("<p>Sum: " + add + "</p>");
```

```
document.write("<p>Difference " + subtract + "</p>");
document.write("<p>Product: " + multiply + "</p>");
document.write("<p>Quotient " + divide + "</p>");
</script>

</body>
</html>
```

Coming to useState

You could also write the code like this instead:

```
const stateData = useState('');
const errorMessage = stateData[0];
const setErrorMessage = stateData[1];
```

This works absolutely fine, but when using array destructuring, the code stays a bit more concise. That's why you typically see the syntax using array destructuring when browsing React apps and examples. You also don't have to use constants; variables (via let) would be fine as well. As you will see throughout this chapter, though, the variables won't be reassigned, so using constants makes sense (but it is not required in any way).

Note

If array destructuring or the difference between variables and constants sounds brand new to you, it's strongly recommended that you refresh your JavaScript basics before progressing with this book. As always, MDN provides great resources for that (see http://packt.link/3B8Ct for array destructuring, https://packt.link/hGjqL for information on the let variable, and https://packt.link/TdPPS for guidance on the use of const).

As mentioned before, useState() returns an array with **exactly two elements**. **It will always be exactly two elements—and always exactly the same kind of elements**. The first element is always the **current state value**, and the second element is a **function** that you can call to set the state to a new value.

But how do these two values (the state value and the state-updating function) work together? What does React do with them internally? How are these two array elements used (by React) to update the UI?

A Look Under the Hood of React

React manages the state values for you, in some internal storage that you, the developer, can't directly access. Since you often do need access to a state value (for instance, some entered email address, as in the preceding example), React provides a way of reading state values: the first element in the array returned by useState(). The first element of the returned array holds the current state value. You can therefore use this element in any place where you need to work with the state value (for example, in the JSX code to output it there).

In addition, you often also need to update the state—for example, because a user entered a new email address. Since you don't manage the state value yourself, React gives you a function that you can call to inform React about the new state value. That's the second element in the returned array.

In the example shown before, you call setErrorMessage('Error!') to set the errorMessage state value to a new string ('Error!').

But why is this managed like this? Why not just use a standard JavaScript variable that you can assign and reassign as needed?

Because React must be informed whenever there's a state that impacts the UI changes. Otherwise, the visible UI doesn't change at all, even in cases where it should. React does not track regular variables and changes to their values, so they have no influence on the state of the UI.

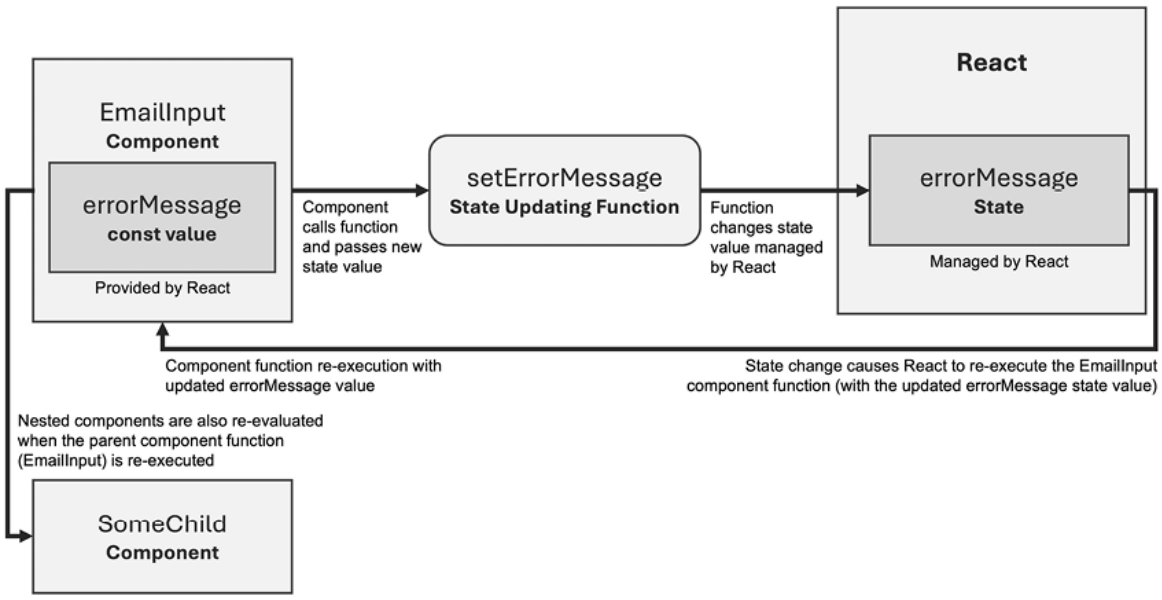
The state-updating function exposed by React (that second array element returned by useState()) does trigger some internal UI-updating effect though.

This state-updating function does more than set a new value; it also informs React that a state value changed and that the UI might therefore be in need of an update.

So, whenever you call setErrorMessage('Error!'), React does not just update the value that it stores internally; it also checks the UI and updates it when needed. UI updates can involve anything from simple text changes up to the complete removal and addition of various DOM elements. Anything is possible there!

React determines the new target UI by rerunning (also called re-evaluating) any component functions that are affected by a state change. That includes the component function that executed the useState() function that returned the state-updating function that was called. But it also includes any child components, since an update in a parent component could lead to new state data that's also used by some child components (the state value could be passed to child components via props).

If you need a visual of how all this fits together, consider the following diagram:



It's important to understand and keep in mind that React will re-execute (re-evaluate) a component function if a state-updating function is called in the component function or some parent component function. This also explains why the state value returned by useState() (that is, the first array element) can

be a constant, even though you can assign new values by calling the state-updating function (the second array element). Since the entire component function is re-executed, useState() is also called again (because all the component function code is executed again) and hence a new array with two new elements is returned by React. The first array element is still the current state value.

However, as the component function was called because of a state update, the current state value is now the updated value.

This can be a bit tricky to wrap your head around, but it is how React works internally. In the end, it's just about component functions being called multiple times by React, just as any JavaScript function can be called multiple times.

Naming Conventions

The useState() Hook is typically used in combination with array destructuring, like this:

```
const [enteredEmail, setEnteredEmail] = useState('');
```

But when using array destructuring, the names of the variables or constants (enteredEmail and setEnteredEmail, in this case) are up to you, the developer. Therefore, a valid question is how you should name these variables or constants. Fortunately, there is a clear convention when it comes to React and useState(), and these variable or constant names.

The **first element** (that is, the current state value) should be named such that it describes what the state value is all about. Examples would be enteredEmail, userEmail, providedEmail, just email, or similar names. You should avoid generic names such as a or value or misleading names such as setValue (which sounds like it is a function—but it isn't).

The **second element** (that is, the state-updating function) should be named such that it becomes clear that it is a function and that it does what it does. Examples would be setEnteredEmail or setEmail. In general, the convention for this function is to name it setXYZ, where XYZ is the name you chose for the first

element, the current state value variable. (Note, though, that you start with an uppercase character, as in setEnteredEmail, not setenteredEmail.)

Allowed State Value Types

Managing entered email addresses (or user input in general) is indeed a common use case and example for working with state. However, you're not limited to this scenario and value type.

In the case of entered user input, you will often deal with string values such as email addresses, passwords, blog posts, or similar values. But any valid JavaScript value type can be managed with the help of useState(). You could, for example, manage the total sum of multiple shopping cart items—that is, a number—or a Boolean value (for example, "Did a user confirm the terms of use?").

Besides managing primitive value types, you can also store and update reference data types such as objects and arrays.

React gives you the flexibility of managing all these value types as state. You can even switch the value type at runtime (just as you can in vanilla JavaScript). It is absolutely fine to store a number as the initial state value and update it to a string at a later point in time.

Just as with vanilla JavaScript, you should, of course, ensure that your program deals with this behavior appropriately, though there's nothing technically wrong with switching types.

More on State

Reacting to input with state

With React, you won't modify the UI from code directly. For example, you won't write commands like "disable the button", "enable the button", "show the success message", etc. Instead, you will describe the UI you want to see for the different visual states of your component ("initial state", "typing state", "success state"), and then trigger the state changes in response to user input. This is similar to how designers think about UI.

Here is a quiz form built using React. Note how it uses the status state variable to determine whether to enable or disable the submit button, and whether to show the success message instead.

```
import { useState } from 'react';
export default function Form() {
  const [answer, setAnswer] = useState('');
  const [error, setError] = useState(null);
  const [status, setStatus] = useState('typing');

  if (status === 'success') {
    return <h1>That's right!</h1>
  }

  async function handleSubmit(e) {
    e.preventDefault();
    setStatus('submitting');
    try {
      await submitForm(answer);
      setStatus('success');
    } catch (err) {
```

```
    setStatus('typing');
    setError(err);
  }
}

function handleTextareaChange(e) {
  setAnswer(e.target.value);
}

return (
  <>
    <h2>City quiz</h2>
    <p>
      In which city is there a billboard that turns air into drinkable water?
    </p>
    <form onSubmit={handleSubmit}>
      <textarea
        value={answer}
        onChange={handleTextareaChange}
        disabled={status === 'submitting'}
      />
      <br />
      <button disabled={
        answer.length === 0 ||
        status === 'submitting'
      }>
```

```
        Submit
      </button>
      {error !== null &&
        <p className="Error">
          {error.message}
        </p>
      }
    </form>
  </>
 );
}

function submitForm(answer) {
 // Pretend it's hitting the network.
 return new Promise((resolve, reject) => {
  setTimeout(() => {
    let shouldError = answer.toLowerCase() !== 'lima'
    if (shouldError) {
      reject(new Error('Good guess but a wrong answer. Try again!'));
    } else {
      resolve();
    }
  }, 1500);
 });
}
```

Choosing the state structure

Structuring state well can make a difference between a component that is pleasant to modify and debug, and one that is a constant source of bugs. The most important principle is that state shouldn't contain redundant or duplicated information. If there's unnecessary state, it's easy to forget to update it, and introduce bugs!

For example, this form has a **redundant** fullName state variable:

```
import { useState } from 'react';
export default function Form() {
  const [firstName, setFirstName] = useState('');
  const [lastName, setLastName] = useState('');
  const [fullName, setFullName] = useState('');

  function handleFirstNameChange(e) {
    setFirstName(e.target.value);
    setFullName(e.target.value + ' ' + lastName);//redundant call
  }

  function handleLastNameChange(e) {
    setLastName(e.target.value);
    setFullName(firstName + ' ' + e.target.value); );//redundant call

  }

  return (
```

```
  <>
    <h2>Let's check you in</h2>
    <label>
      First name:{' '}
      <input
        value={firstName}
        onChange={handleFirstNameChange}
      />
    </label>
    <label>
      Last name:{' '}
      <input
        value={lastName}
        onChange={handleLastNameChange}
      />
    </label>
    <p>
      Your ticket will be issued to: <b>{fullName}</b>
    </p>
  </>
 );
}
```

Sharing state between components

Sometimes, you want the state of two components to always change together. To do it, remove state from both of them, move it to their closest common

parent, and then pass it down to them via props. This is known as "lifting state up", and it's one of the most common things you will do writing React code.

In this example, only one panel should be active at a time. To achieve this, instead of keeping the active state inside each individual panel, the parent component holds the state and specifies the props for its children.

Lifting state up by example

In this example, a parent Accordion component renders two separate Panels:

- Accordion
  - Panel
  - Panel

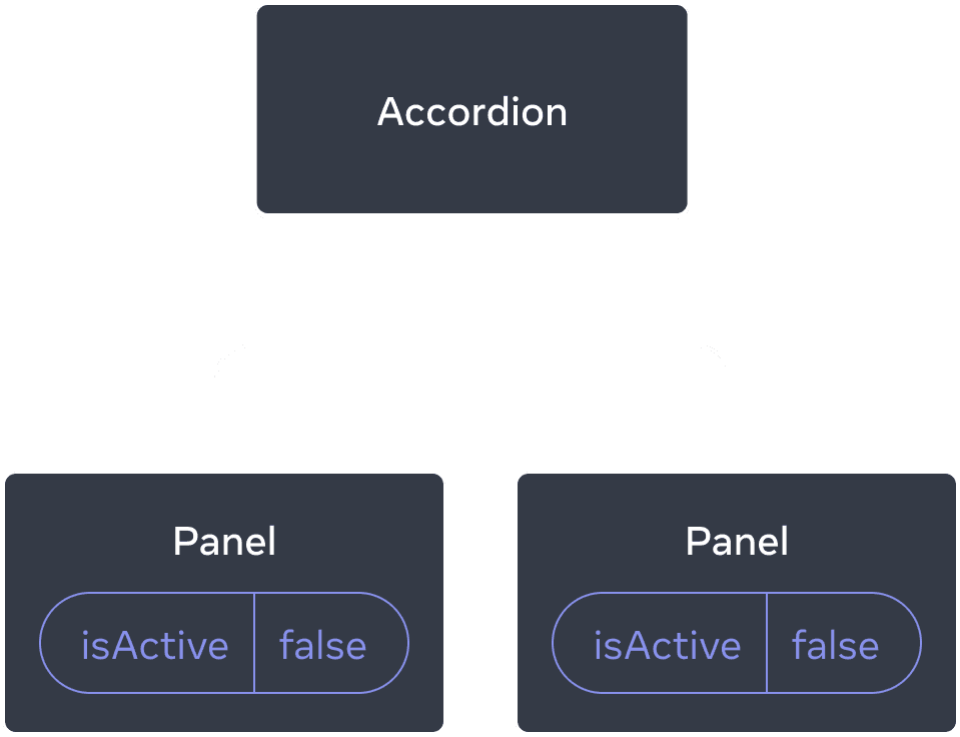
Each Panel component has a boolean isActive state that determines whether its content is visible.

Press the Show button for both panels:

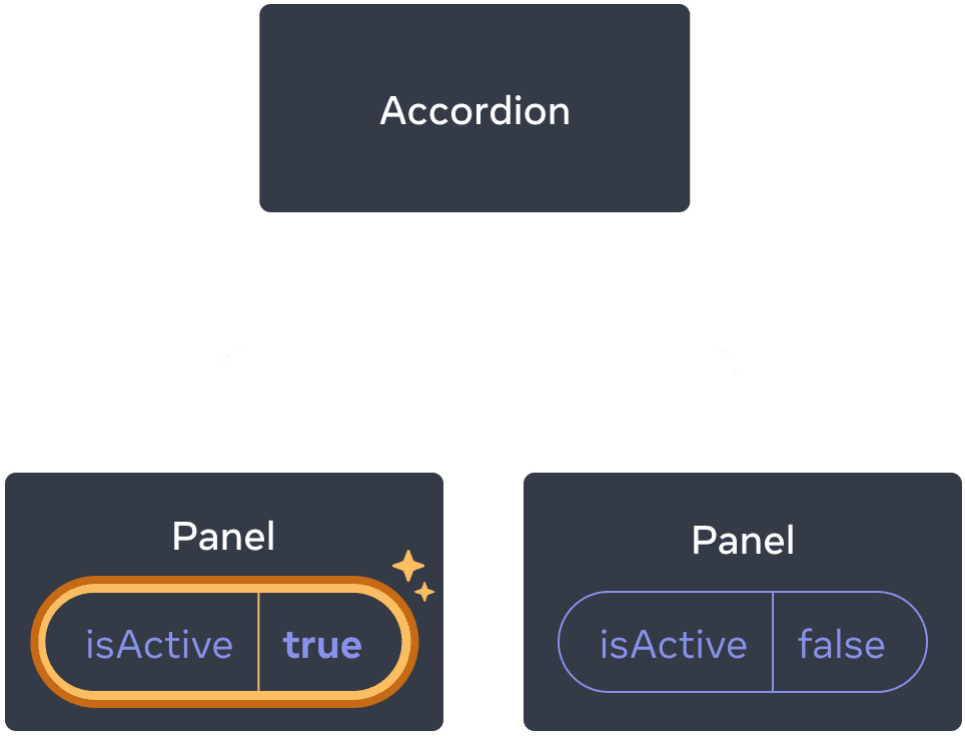
```
import { useState } from 'react';


function Panel({ title, children }) {
 const [isActive, setIsActive] = useState(false);
 return (
  <section className="panel">
   <h3>{title}</h3>
   {isActive ? (
    <p>{children}</p>
   ) : (
    <button onClick={() => setIsActive(true)}>
     Show
    </button>
```

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


export default function Accordion() {
  return (
    <>
      <h2>Almaty, Kazakhstan</h2>
      <Panel title="About">
        With a population of about 2 million, Almaty is Kazakhstan's largest city. From 1929 to 1997, it was its capital city.
      </Panel>
      <Panel title="Etymology">
        The name comes from <span lang="kk-KZ">алма</span>, the Kazakh word for "apple" and is often translated as "full of apples". In fact, the region surrounding Almaty is thought to be the ancestral home of the apple, and the wild <i lang="la">Malus sieversii</i> is considered a likely candidate for the ancestor of the modern domestic apple.
      </Panel>
    </>
  );
}
```

Notice how pressing one panel's button does not affect the other panel—they are independent.



Initially, each Panel's isActive state is false, so they both appear collapsed



Clicking either Panel's button will only update that Panel's isActive state alone

But now let's say you want to change it so that only one panel is expanded at any given time. With that design, expanding the second panel should collapse the first one. How would you do that?

To coordinate these two panels, you need to "lift their state up" to a parent component in three steps:

1. **Remove** state from the child components.
2. **Pass** hardcoded data from the common parent.
3. **Add** state to the common parent and pass it down together with the event handlers.

This will allow the Accordion component to coordinate both Panels and only expand one at a time.

Step 1: Remove state from the child components

You will give control of the Panel's isActive to its parent component. This means that the parent component will pass isActive to Panel as a prop instead. Start by **removing this line** from the Panel component:

const [isActive, setIsActive] = useState(false);

And instead, add isActive to the Panel's list of props:

function Panel({ title, children, isActive }) {

Now the Panel's parent component can *control* isActive by passing it down as a prop. Conversely, the Panel component now has *no control* over the value of isActive—it's now up to the parent component!

Step 2: Pass hardcoded data from the common parent

To lift state up, you must locate the closest common parent component of *both* of the child components that you want to coordinate:

- Accordion *(closest common parent)*
    - Panel
    - Panel

In this example, it's the Accordion component. Since it's above both panels and can control their props, it will become the "source of truth" for which panel is currently active. Make the Accordion component pass a hardcoded value of isActive (for example, true) to both panels:

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

```
export default function Accordion() {
  return (
    <>
      <h2>Almaty, Kazakhstan</h2>
      <Panel title="About" isActive={true}>
        With a population of about 2 million, Almaty is Kazakhstan's largest city. From 1929 to 1997, it was its capital city.
      </Panel>
      <Panel title="Etymology" isActive={true}>
        The name comes from <span lang="kk-KZ">алма</span>, the Kazakh word for "apple" and is often translated as "full of apples". In fact, the region surrounding Almaty is thought to be the ancestral home of the apple, and the wild <i lang="la">Malus sieversii</i> is considered a likely candidate for the ancestor of the modern domestic apple.
      </Panel>
    </>
  );
}

function Panel({ title, children, isActive }) {
  return (
    <section className="panel">
      <h3>{title}</h3>
      {isActive ? (
        <p>{children}</p>
      ) : (
```

```
      <button onClick={() => setIsActive(true)}>
        Show
      </button>
    )}
  </section>
 );
}
```

Step 3: Add state to the common parent

Lifting state up often changes the nature of what you're storing as state.

In this case, only one panel should be active at a time. This means that the Accordion common parent component needs to keep track of *which* panel is the active one. Instead of a boolean value, it could use a number as the index of the active Panel for the state variable:

const [activeIndex, setActiveIndex] = useState(0);

When the activeIndex is 0, the first panel is active, and when it's 1, it's the second one.

Clicking the "Show" button in either Panel needs to change the active index in Accordion. A Panel can't set the activeIndex state directly because it's defined inside the Accordion. The Accordion component needs to *explicitly allow* the Panel component to change its state by passing an event handler down as a prop:

```
<>
 <Panel
  isActive={activeIndex === 0}
  onShow={() => setActiveIndex(0)}
 >
```

```
  ...
 </Panel>
 <Panel
  isActive={activeIndex === 1}
  onShow={() => setActiveIndex(1)}
 >
  ...
 </Panel>
</>
```

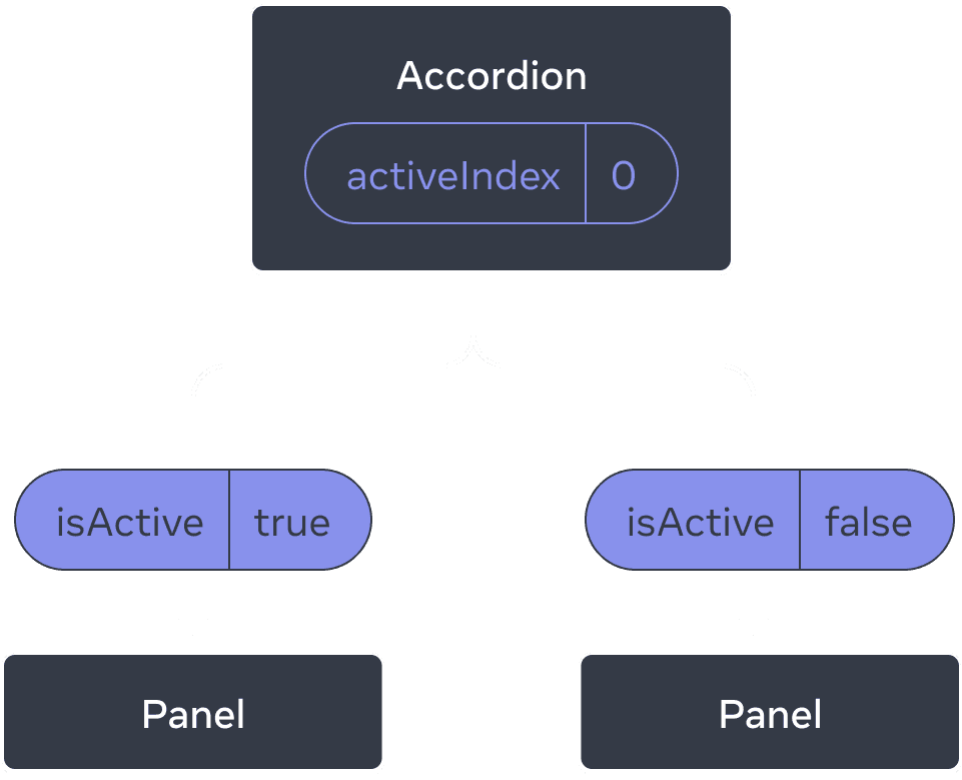
The <button> inside the Panel will now use the onShow prop as its click event handler:

```
import { useState } from 'react';
export default function Accordion() {
 const [activeIndex, setActiveIndex] = useState(0);
 return (
  <>
   <h2>Almaty, Kazakhstan</h2>
   <Panel
    title="About"
    isActive={activeIndex === 0}
    onShow={() => setActiveIndex(0)}
   >
```

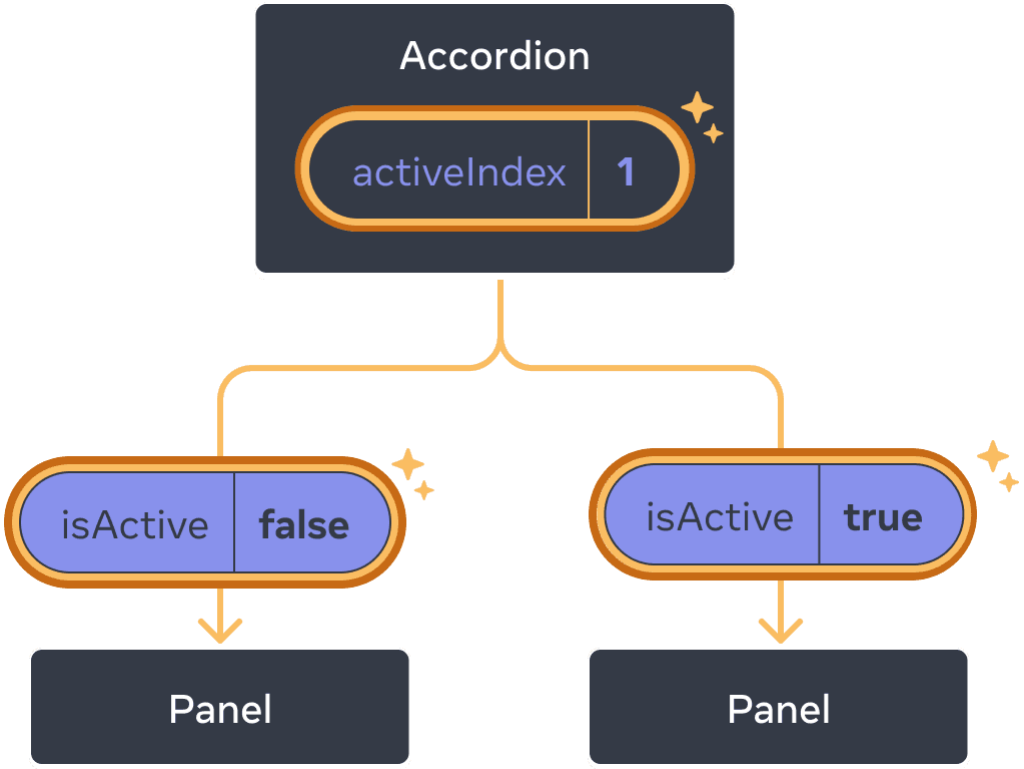
```
      With a population of about 2 million, Almaty is Kazakhstan's largest city. From 1929 to 1997, it was its capital city.
    </Panel>
    <Panel
      title="Etymology"
      isActive={activeIndex === 1}
      onShow={() => setActiveIndex(1)}
    >
      The name comes from <span lang="kk-KZ">алма</span>, the Kazakh word for "apple" and is often translated as "full of apples". In fact, the region surrounding Almaty is thought to be the ancestral home of the apple, and the wild <i lang="la">Malus sieversii</i> is considered a likely candidate for the ancestor of the modern domestic apple.
    </Panel>
  </>
 );
}

function Panel({
 title,
 children,
 isActive,
 onShow
}) {
 return (
  <section className="panel">
   <h3>{title}</h3>
```

```
    {isActive ? (
      <p>{children}</p>
    ) : (
      <button onClick={onShow}>
        Show
      </button>
    )}
  </section>
 );
}
```

This completes lifting state up! Moving state into the common parent component allowed you to coordinate the two panels. Using the active index instead of two “is shown” flags ensured that only one panel is active at a given time. And passing down the event handler to the child allowed the child to change the parent’s state.



Initially, Accordion's activeIndex is 0, so the first Panel receives isActive = true



When Accordion's activeIndex state changes to 1, the second Panel receives isActive = true instead

A single source of truth for each state

In a React application, many components will have their own state. Some state may "live" close to the leaf components (components at the bottom of the tree) like inputs. Other state may "live" closer to the top of the app. For example, even client-side routing libraries are usually implemented by storing the current route in the React state, and passing it down by props!

For each unique piece of state, you will choose the component that "owns" it. This principle is also known as having a "single source of truth". It doesn't mean that all state lives in one place—but that for *each* piece of state, there is a *specific* component that holds that piece of information. Instead of duplicating shared state between components, *lift it up* to their common shared parent, and *pass it down* to the children that need it.

Your app will change as you work on it. It is common that you will move state down or back up while you're still figuring out where each piece of the state "lives". This is all part of the process!

Recap

- When you want to coordinate two components, move their state to their common parent.
- Then pass the information down through props from their common parent.
- Finally, pass the event handlers down so that the children can change the parent's state.
- It's useful to consider components as "controlled" (driven by props) or "uncontrolled" (driven by state).

Try out some challenges

Challenge 1 of 2:

Synced inputs

These two inputs are independent. Make them stay in sync: editing one input should update the other input with the same text, and vice versa.

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


export default function SyncedInputs() {
  return (
    <>
      <Input label="First input" />
      <Input label="Second input" />
    </>
  );
}
```

```
function Input({ label }) {
  const [text, setText] = useState('');

  function handleChange(e) {
    setText(e.target.value);
  }

  return (
    <label>
      {label}
      {' '}
      <input
        value={text}
        onChange={handleChange}
      />
    </label>
  );
}
```

PRESERVING AND RESETTING STATE

When you re-render a component, React needs to decide which parts of the tree to keep (and update), and which parts to discard or re-create from scratch. In most cases, React's automatic behavior works well enough. By default, React preserves the parts of the tree that "match up" with the previously rendered component tree.

However, sometimes this is not what you want. In this chat app, typing a message and then switching the recipient does not reset the input. This can make the user accidentally send a message to the wrong person:

React lets you override the default behavior, and *force* a component to reset its state by passing it a different key, like <Chat key={email} />. This tells React that if the recipient is different, it should be considered a *different* Chat component that needs to be re-created from scratch with the new data (and UI like inputs). Now switching between the recipients resets the input field—even though you render the same component.

```
import { useState } from 'react';
import Chat from './Chat.js';
import ContactList from './ContactList.js';

export default function Messenger() {
  const [to, setTo] = useState(contacts[0]);
  return (
    <div>
      <ContactList
        contacts={contacts}
        selectedContact={to}
        onSelect={contact => setTo(contact)}
      />
      <Chat contact={to} />
    </div>
  )
}
```

```
const contacts = [
  { name: 'Taylor', email: 'taylor@mail.com' },
  { name: 'Alice', email: 'alice@mail.com' },
  { name: 'Bob', email: 'bob@mail.com' }
];
```

ContactList.jsx

```
export default function ContactList({
  selectedContact,
  contacts,
  onSelect
}) {
  return (
    <section className="contact-list">
      <ul>
        {contacts.map(contact =>
          <li key={contact.email}>
            <button onClick={() => {
              onSelect(contact);
            }}>
              {contact.name}
            </button>
          </li>
        )}
      </ul>
```

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

Chat.jsx

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


export default function Chat({ contact }) {
 const [text, setText] = useState('');
 return (
  <section className="chat">
   <textarea
    value={text}
    placeholder={'Chat to ' + contact.name}
    onChange={e => setText(e.target.value)}
   />
   <br />
   <button>Send to {contact.email}</button>
  </section>
 );
}
```

State Continued…

https://react.dev/learn/preserving-and-resetting-state#state-is-tied-to-a-position-in-the-tree

State is isolated between components. React keeps track of which state belongs to which component based on their place in the UI tree. You can control when to preserve state and when to reset it between re-renders.

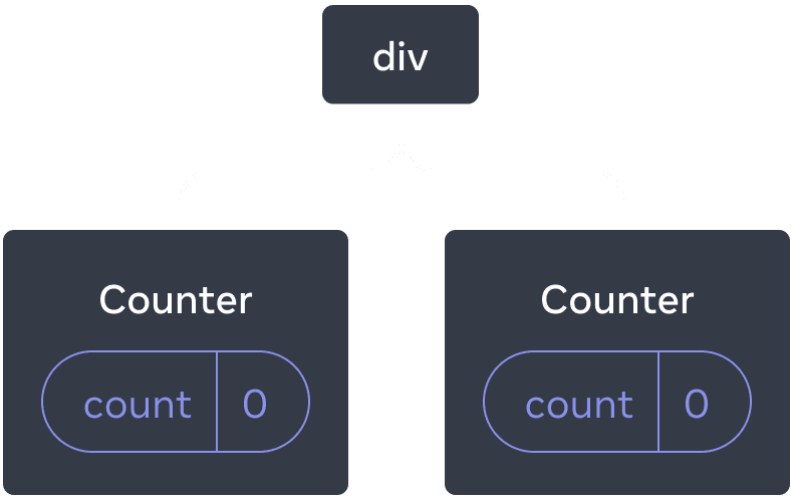
State is tied to a position in the render tree

React builds render trees for the component structure in your UI.

When you give a component state, you might think the state "lives" inside the component. But the state is actually held inside React. React associates each piece of state it's holding with the correct component by where that component sits in the render tree.

Here, there is only one <Counter /> JSX tag, but it's rendered at two different positions:

Here's how these look as a tree:



React tree

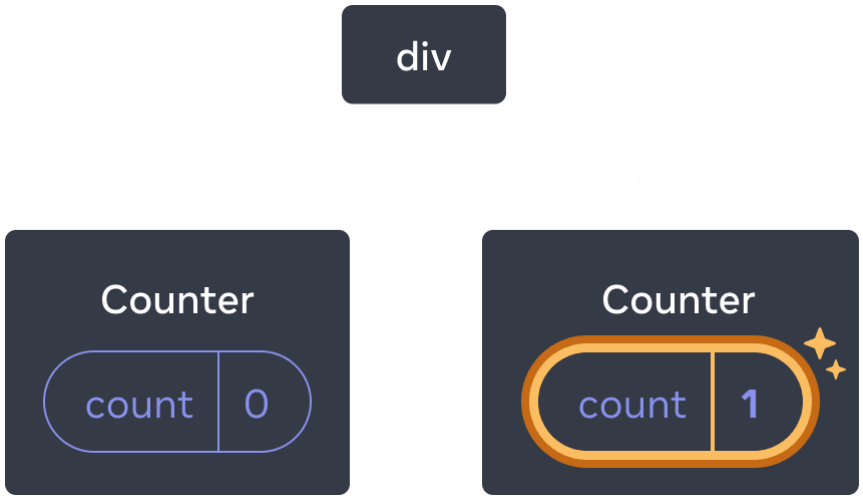
These are two separate counters because each is rendered at its own position in the tree. You don't usually have to think about these positions to use React, but it can be useful to understand how it works.

In React, each component on the screen has fully isolated state. For example, if you render two Counter components side by side, each of them will get its own, independent, score and hover states.

Try clicking both counters and notice they don’t affect each other:

```
const [showB, setShowB] = useState(true);
 return (
  <div>
   <Counter />
   {showB && <Counter />}
   <label>
    <input
     type="checkbox"
     checked={showB}
     onChange={e => {
      setShowB(e.target.checked)
     }}
    />
    Render the second counter
   </label>
  </div>
 );
```

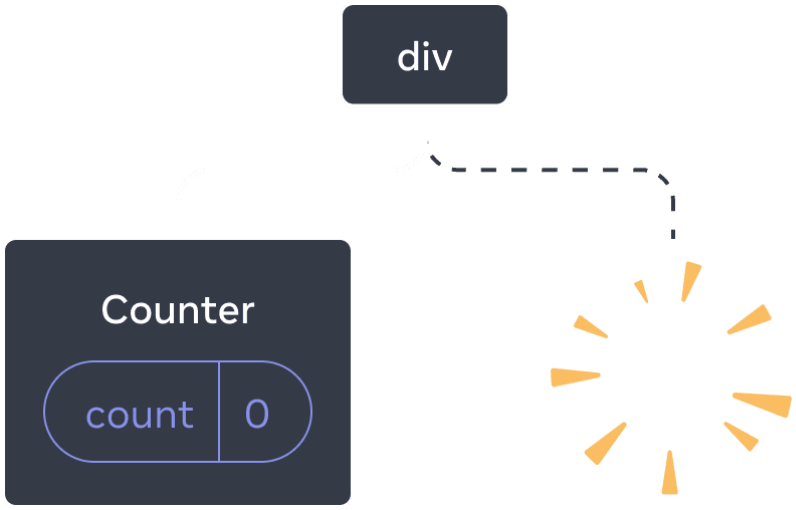
As you can see, when one counter is updated, only the state for that component is updated:



Updating state

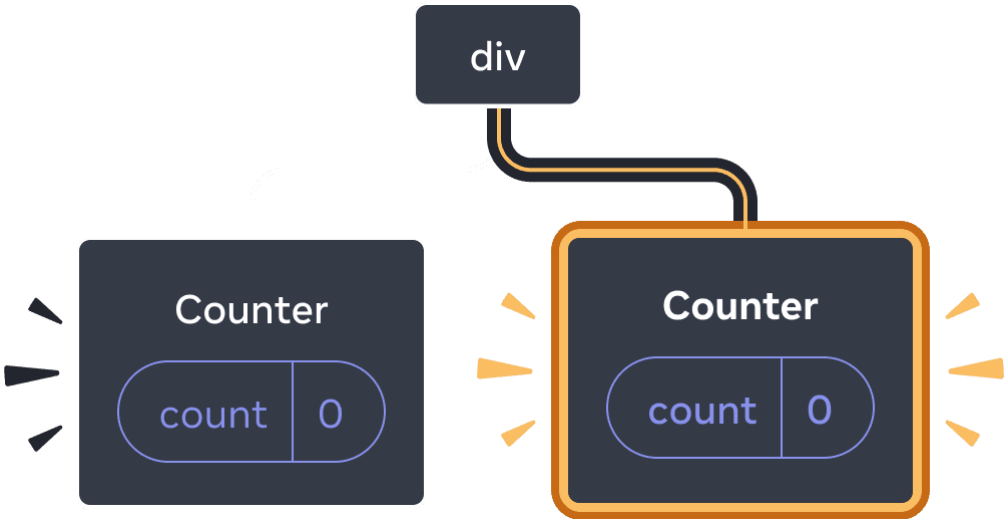
React will keep the state around for as long as you render the same component at the same position in the tree. To see this, increment both counters, then remove the second component by unchecking “Render the second counter” checkbox, and then add it back by ticking it again:

Notice how the moment you stop rendering the second counter, its state disappears completely. That’s because when React removes a component, it destroys its state.



Deleting a component

When you tick “Render the second counter”, a second Counter and its state are initialized from scratch (score = 0) and added to the DOM.



Adding a component

React preserves a component's state for as long as it's being rendered at its position in the UI tree. If it gets removed, or a different component gets rendered at the same position, React discards its state.

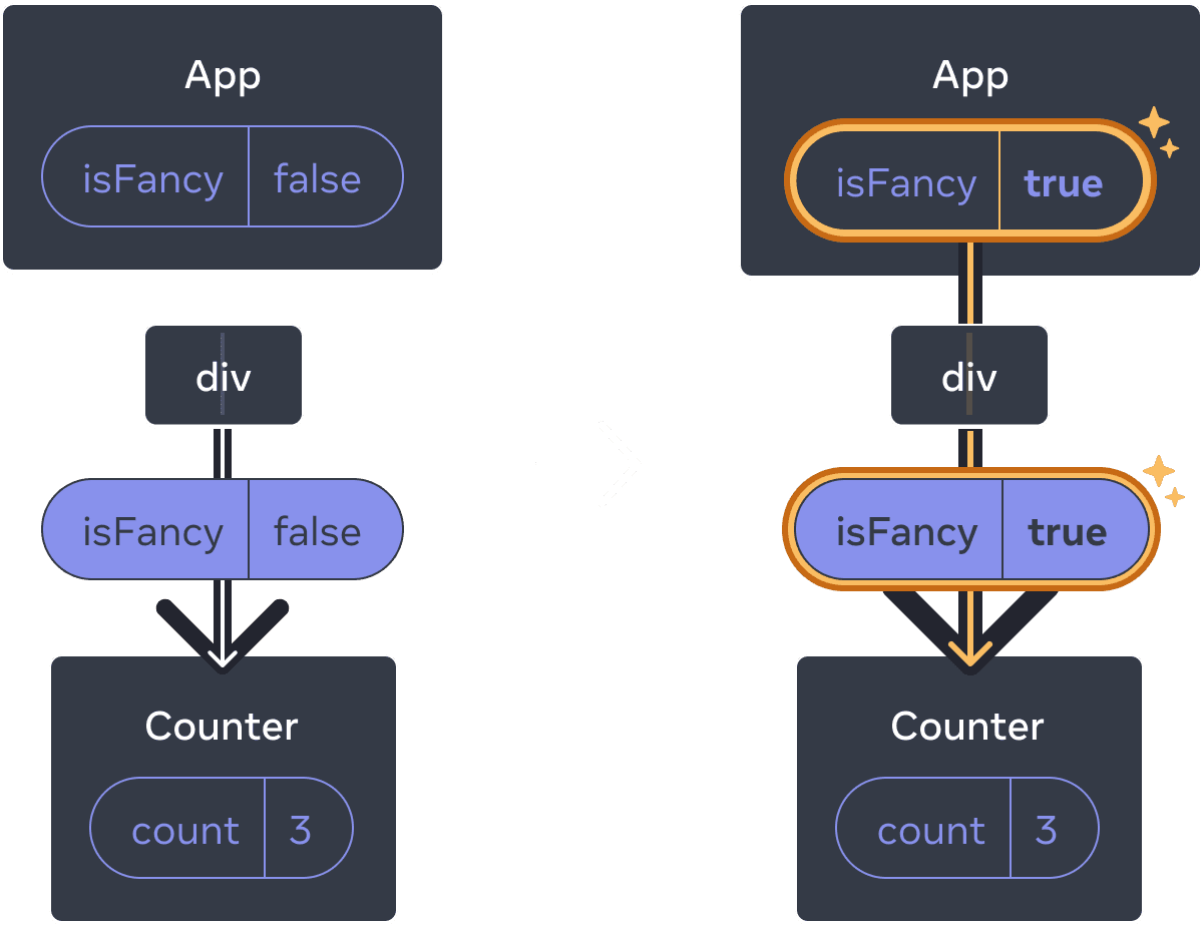
Same component at the same position preserves state

In this example, there are two different <Counter /> tags:

```
function Counter({ isFancy }) {
  const [score, setScore] = useState(0);
  const [hover, setHover] = useState(false);

  let className = 'counter';
  if (hover) {
    className += ' hover';
  }
  if (isFancy) {
    className += ' fancy';
  }
```

```
  return (
    <div
      className={className}
      onPointerEnter={() => setHover(true)}
      onPointerLeave={() => setHover(false)}
    >
      <h1>{score}</h1>
      <button onClick={() => setScore(score + 1)}>
        Add one
      </button>
    </div>
  );
}
```

When you tick or clear the checkbox, the counter state does not get reset. Whether isFancy is true or false, you always have a <Counter /> as the first child of the div returned from the root App component:



Updating the App state does not reset the Counter because Counter stays in the same position

It's the same component at the same position, so from React's perspective, it's the same counter.

Remember that **it's the position in the UI tree—not in the JSX markup—that matters to React!** This component has two return clauses with different <Counter /> JSX tags inside and outside the if:

```
const [isFancy, setIsFancy] = useState(false);
 if (isFancy) {
  return (
   <div>
    <Counter isFancy={true} />
```

```
    <label>
     <input
      type="checkbox"
      checked={isFancy}
      onChange={e => {
       setIsFancy(e.target.checked)
      }}
     />
     Use fancy styling
    </label>
   </div>
  );
}
return (
 <div>
  <Counter isFancy={false} />
  <label>
   <input
    type="checkbox"
    checked={isFancy}
    onChange={e => {
     setIsFancy(e.target.checked)
    }}
   />
   Use fancy styling
  </label>
```

```
  </div>
 );
```

You might expect the state to reset when you tick checkbox, but it doesn't! This is because **both of these <Counter /> tags are rendered at the same position.** React doesn't know where you place the conditions in your function. All it "sees" is the tree you return.

In both cases, the App component returns a with <Counter /> as a first child. To React, these two counters have the same "address": the first child of the first child of the root. This is how React matches them up between the previous and next renders, regardless of how you structure your logic.

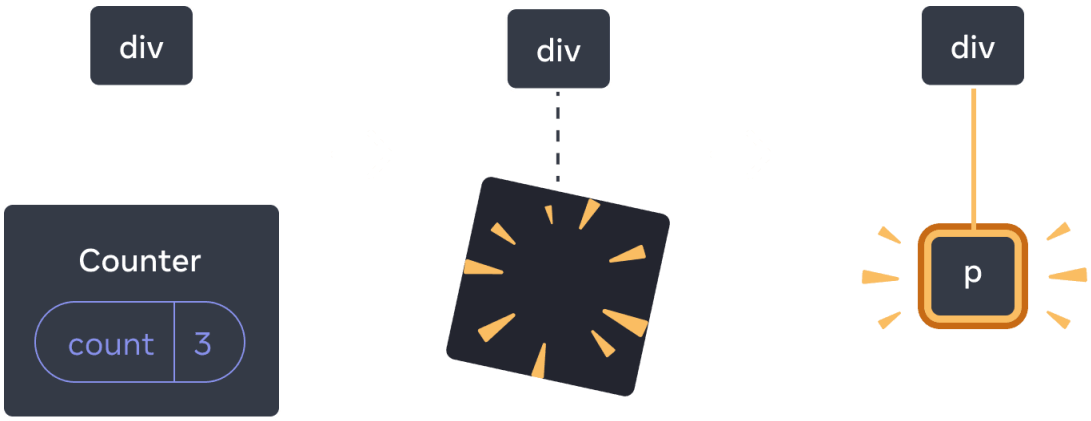
Different components at the same position reset state

In this example, ticking the checkbox will replace <Counter> with a <p>:

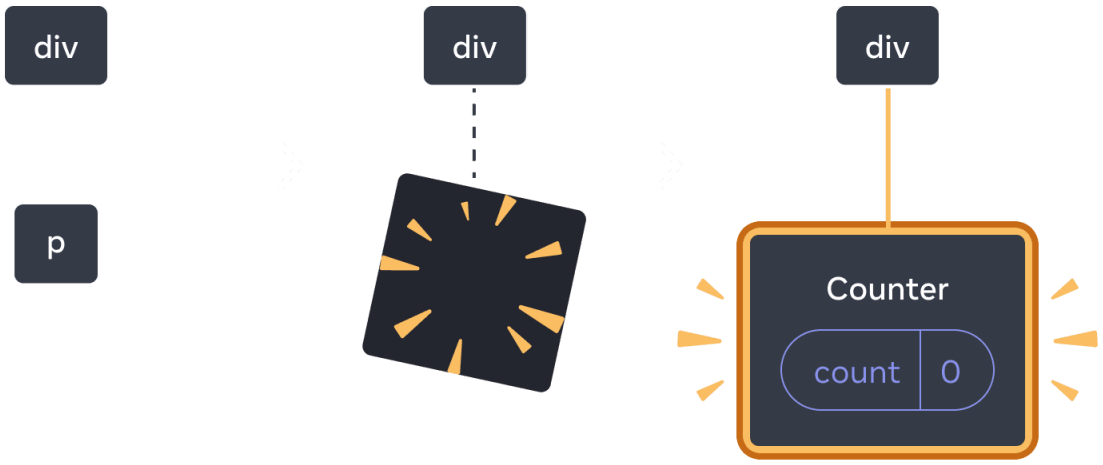
```
function Counter() {
  const [score, setScore] = useState(0);
  const [hover, setHover] = useState(false);

  let className = 'counter';
  if (hover) {
    className += ' hover';
  }

  return (
    <div
      className={className}
      onPointerEnter={() => setHover(true)}
      onPointerLeave={() => setHover(false)}
```

```
  >
    <h1>{score}</h1>
    <button onClick={() => setScore(score + 1)}>
      Add one
    </button>
  </div>
 );
}
```

Here, you switch between *different* component types at the same position. Initially, the first child of the contained a Counter. But when you swapped in a p, React removed the Counter from the UI tree and destroyed its state.



When Counter changes to p, the Counter is deleted and the p is added



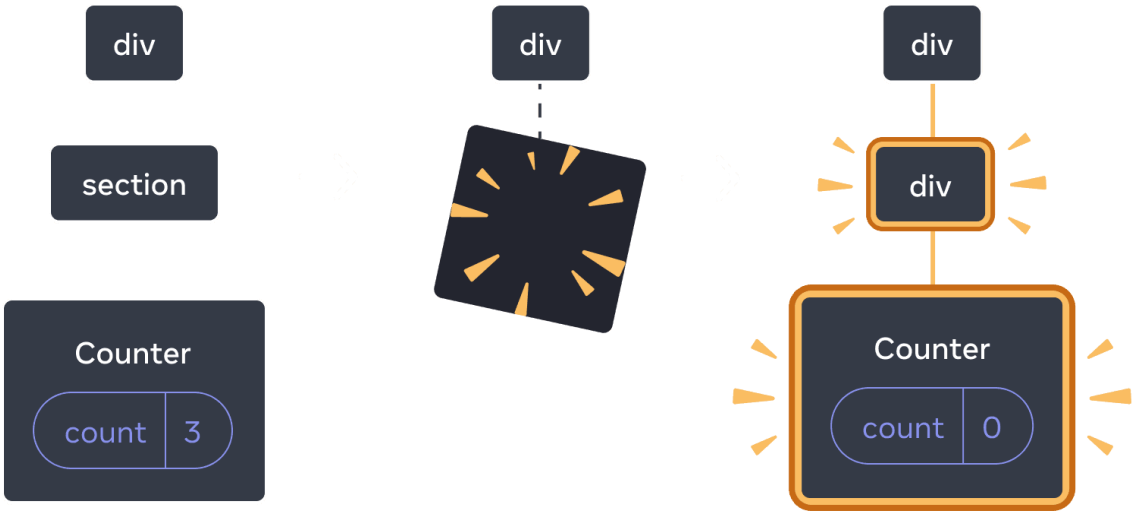
When switching back, the p is deleted and the Counter is added

Also, **when you render a different component in the same position, it resets the state of its entire subtree.** To see how this works, increment the counter and then tick the checkbox:

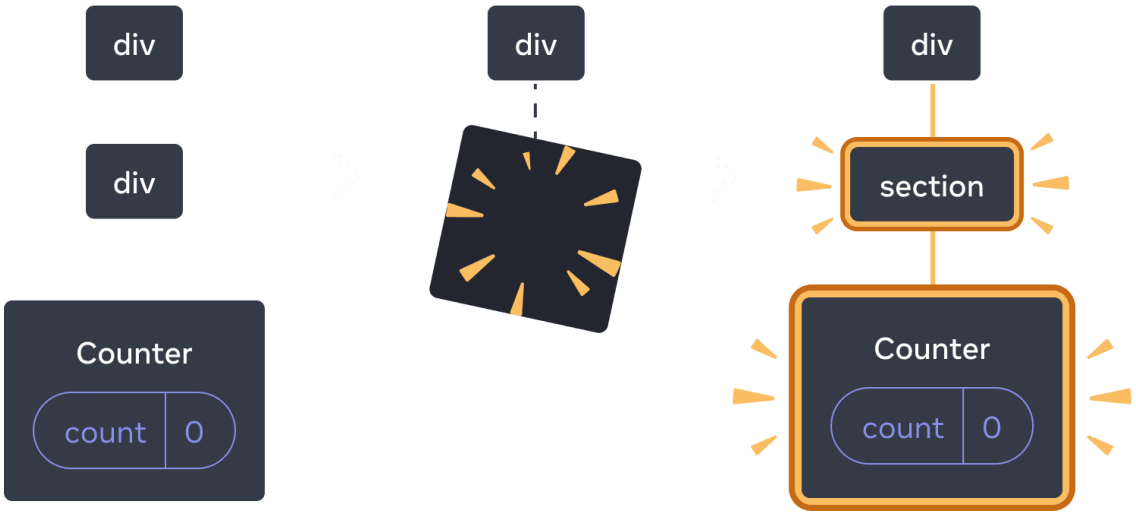
```
const [isFancy, setIsFancy] = useState(false);
 return (
  <div>
   {isFancy ? (
    <div>
     <Counter isFancy={true} />
    </div>
   ) : (
    <section>
     <Counter isFancy={false} />
    </section>
   )}
   <label>
    <input
     type="checkbox"
     checked={isFancy}
```

```
        onChange={e => {
          setIsFancy(e.target.checked)
        }}
      />
      Use fancy styling
    </label>
  </div>
);
```

The counter state gets reset when you click the checkbox. Although you render a Counter, the first child of the div changes from a section to a div. When the child section was removed from the DOM, the whole tree below it (including the Counter and its state) was destroyed as well.



When section changes to div, the section is deleted and the new div is added



When switching back, the div is deleted and the new section is added

As a rule of thumb, **if you want to preserve the state between re-renders, the structure of your tree needs to "match up"** from one render to another. If the structure is different, the state gets destroyed because React destroys state when it removes a component from the tree.

Pitfall

This is why you should not nest component function definitions.

Here, the MyTextField component function is defined *inside* MyComponent:

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

export default function MyComponent() {
 const [counter, setCounter] = useState(0);

 function MyTextField() {
  const [text, setText] = useState('');

  return (
   <input
    value={text}
```

```
      onChange={e => setText(e.target.value)}
    />
  );
}

return (
  <>
    <MyTextField />
    <button onClick={() => {
      setCounter(counter + 1)
    }}>Clicked {counter} times</button>
  </>
);
}
```

Every time you click the button, the input state disappears! This is because a *different* MyTextField function is created for every render of MyComponent. You're rendering a *different* component in the same position, so React resets all state below. This leads to bugs and performance problems. To avoid this problem, **always declare component functions at the top level, and don't nest their definitions.**

Resetting state at the same position

By default, React preserves state of a component while it stays at the same position. Usually, this is exactly what you want, so it makes sense as the default behavior. But sometimes, you may want to reset a component's state. Consider this app that lets two players keep track of their scores during each turn:

https://react.dev/learn/preserving-and-resetting-state#state-is-tied-to-a-position-in-the-tree

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

```
export default function Scoreboard() {
  const [isPlayerA, setIsPlayerA] = useState(true);
  return (
    <div>
      {isPlayerA ? (
        <Counter person="Taylor" />
      ) : (
        <Counter person="Sarah" />
      )}
      <button onClick={() => {
        setIsPlayerA(!isPlayerA);
      }}>
        Next player!
      </button>
    </div>
  );
}

function Counter({ person }) {
  const [score, setScore] = useState(0);
  const [hover, setHover] = useState(false);

  let className = 'counter';
  if (hover) {
    className += ' hover';
```

```
  }


  return (
    <div
      className={className}
      onPointerEnter={() => setHover(true)}
      onPointerLeave={() => setHover(false)}
    >
      <h1>{person}'s score: {score}</h1>
      <button onClick={() => setScore(score + 1)}>
        Add one
      </button>
    </div>
  );
}
```

Currently, when you change the player, the score is preserved. The two Counters appear in the same position, so React sees them as *the same* Counter whose person prop has changed.

But conceptually, in this app they should be two separate counters. They might appear in the same place in the UI, but one is a counter for Taylor, and another is a counter for Sarah.

There are two ways to reset state when switching between them:

1. Render components in different positions
2. Give each component an explicit identity with key

REDUCER

Extracting state logic into a reducer

Components with many state updates spread across many event handlers can get overwhelming. For these cases, you can consolidate all the state update logic outside your component in a single function, called "reducer".

You will learn

- What reducer functions are
- How to switch from useState to useReducer
- When to choose a reducer
- Tips for writing effective reducers

Consolidate state logic with a reducer

As your components grow in complexity, it can get harder to see at a glance all the different ways in which a component's state gets updated. For example, the TaskApp component below holds an array of tasks in state and uses three different event handlers to add, remove, and edit tasks:

```
import { useState } from 'react';
import AddTask from './AddTask.js';
import TaskList from './TaskList.js';

export default function TaskApp() {
  const [tasks, setTasks] = useState(initialTasks);

  function handleAddTask(text) {
    setTasks([
      ...tasks,
      {
```

```
    id: nextId++,
    text: text,
    done: false,
  },
 ]);
}

function handleChangeTask(task) {
 setTasks(
  tasks.map((t) => {
   if (t.id === task.id) {
    return task;
   } else {
    return t;
   }
  })
 );
}

function handleDeleteTask(taskId) {
 setTasks(tasks.filter((t) => t.id !== taskId));
}

return (
 <>
  <h1>Prague itinerary</h1>
```

```
      <AddTask onAddTask={handleAddTask} />
      <TaskList
        tasks={tasks}
        onChangeTask={handleChangeTask}
        onDeleteTask={handleDeleteTask}
      />
    </>
  );
}

let nextId = 3;
const initialTasks = [
  {id: 0, text: 'Visit Kafka Museum', done: true},
  {id: 1, text: 'Watch a puppet show', done: false},
  {id: 2, text: 'Lennon Wall pic', done: false},
];
```

Each of its event handlers calls setTasks in order to update the state. As this component grows, so does the amount of state logic sprinkled throughout it. To reduce this complexity and keep all your logic in one easy-to-access place, you can move that state logic into a single function outside your component, **called a "reducer".**

Reducers are a different way to handle state. You can migrate from useState to useReducer in three steps:

1. **Move** from setting state to dispatching actions.
2. **Write** a reducer function.

3. **Use** the reducer from your component.

Step 1: Move from setting state to dispatching actions

Your event handlers currently specify *what to do* by setting state:

```
function handleAddTask(text) {
  setTasks([
    ...tasks,
    {
      id: nextId++,
      text: text,
      done: false,
    },
  ]);
}

function handleChangeTask(task) {
  setTasks(
    tasks.map((t) => {
      if (t.id === task.id) {
        return task;
      } else {
        return t;
      }
    })
  );
}
```

```
function handleDeleteTask(taskId) {
  setTasks(tasks.filter((t) => t.id !== taskId));
}
```

Remove all the state setting logic. What you are left with are three event handlers:

- handleAddTask(text) is called when the user presses “Add”.
- handleChangeTask(task) is called when the user toggles a task or presses “Save”.
- handleDeleteTask(taskId) is called when the user presses “Delete”.

Managing state with reducers is slightly different from directly setting state. Instead of telling React “what to do” by setting state, you specify “what the user just did” by dispatching “actions” from your event handlers. (The state update logic will live elsewhere!) So instead of “setting tasks” via an event handler, you’re dispatching an “added/changed/deleted a task” action. This is more descriptive of the user’s intent.

```
function handleAddTask(text) {
  dispatch({
    type: 'added',
    id: nextId++,
    text: text,
  });
}

function handleChangeTask(task) {
  dispatch({
    type: 'changed',
    task: task,
  });
```

```
}


function handleDeleteTask(taskId) {
  dispatch({
    type: 'deleted',
    id: taskId,
  });
}
```

The object you pass to dispatch is called an “action”:

```
function handleDeleteTask(taskId) {
  dispatch(
    // "action" object:
    {
      type: 'deleted',
      id: taskId,
    }
  );
}
```

It is a regular JavaScript object. You decide what to put in it, but generally it should contain the minimal information about *what happened*. (You will add the dispatch function itself in a later step.)

Note

An action object can have any shape.

By convention, it is common to give it a string type that describes what happened, and pass any additional information in other fields. The type is

```
specific to a component, so in this example either 'added' or 'added_task'
would be fine. Choose a name that says what happened!
dispatch({
  // specific to component
  type: 'what_happened',
  // other fields go here
});
```

Step 2: Write a reducer function

A reducer function is where you will put your state logic. It takes two arguments, the current state and the action object, and it returns the next state:

```
function yourReducer(state, action) {
  // return next state for React to set
}
```

React will set the state to what you return from the reducer.

To move your state setting logic from your event handlers to a reducer function in this example, you will:

1. Declare the current state (tasks) as the first argument.
2. Declare the action object as the second argument.
3. Return the *next* state from the reducer (which React will set the state to).

Here is all the state setting logic migrated to a reducer function:

```
function tasksReducer(tasks, action) {
  if (action.type === 'added') {
    return [
      ...tasks,
      {
```

```
      id: action.id,
      text: action.text,
      done: false,
    },
  ];
 } else if (action.type === 'changed') {
  return tasks.map((t) => {
   if (t.id === action.task.id) {
    return action.task;
   } else {
    return t;
   }
  });
 } else if (action.type === 'deleted') {
  return tasks.filter((t) => t.id !== action.id);
 } else {
  throw Error('Unknown action: ' + action.type);
 }
}
```

Because the reducer function takes state (tasks) as an argument, you can **declare it outside of your component.** This decreases the indentation level and can make your code easier to read.

Step 3: Use the reducer from your component

Finally, you need to hook up the tasksReducer to your component. Import the useReducer Hook from React:

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

Then you can replace useState:

```
const [tasks, setTasks] = useState(initialTasks);
```

with useReducer like so:

```
const [tasks, dispatch] = useReducer(tasksReducer, initialTasks);
```

The useReducer Hook is similar to useState—you must pass it an initial state and it returns a stateful value and a way to set state (in this case, the dispatch function). But it's a little different.

The useReducer Hook takes two arguments:

1. A reducer function
2. An initial state

And it returns:

1. A stateful value
2. A dispatch function (to "dispatch" user actions to the reducer)

Now it's fully wired up! Here, the reducer is declared at the bottom of the component file:

```
import { useReducer } from 'react';
import AddTask from './AddTask.js';
import TaskList from './TaskList.js';

export default function TaskApp() {
  const [tasks, dispatch] = useReducer(
    tasksReducer,
    initialTasks
  );

  function handleAddTask(text) {
    dispatch({
```

```
    type: 'added',
    id: nextId++,
    text: text,
  });
}

function handleChangeTask(task) {
  dispatch({
    type: 'changed',
    task: task
  });
}

function handleDeleteTask(taskId) {
  dispatch({
    type: 'deleted',
    id: taskId
  });
}

return (
  <>
    <h1>Prague itinerary</h1>
    <AddTask
      onAddTask={handleAddTask}
    />
```

```
      <TaskList
        tasks={tasks}
        onChangeTask={handleChangeTask}
        onDeleteTask={handleDeleteTask}
      />
    </>
  );
}

function tasksReducer(tasks, action) {
  switch (action.type) {
    case 'added': {
      return [...tasks, {
        id: action.id,
        text: action.text,
        done: false
      }];
    }
    case 'changed': {
      return tasks.map(t => {
        if (t.id === action.task.id) {
          return action.task;
        } else {
          return t;
        }
      });
```

```
    }
    case 'deleted': {
      return tasks.filter(t => t.id !== action.id);
    }
    default: {
      throw Error('Unknown action: ' + action.type);
    }
  }
}

let nextId = 3;
const initialTasks = [
  { id: 0, text: 'Visit Kafka Museum', done: true },
  { id: 1, text: 'Watch a puppet show', done: false },
  { id: 2, text: 'Lennon Wall pic', done: false }
];
```

Component logic can be easier to read when you separate concerns like this. Now the event handlers only specify *what happened* by dispatching actions, and the reducer function determines *how the state updates* in response to them.

Comparing useState and useReducer

Reducers are not without downsides! Here's a few ways you can compare them:

- **Code size:** Generally, with useState you have to write less code upfront. With useReducer, you have to write both a reducer function *and* dispatch actions. However, useReducer can help cut down on the code if many event handlers modify state in a similar way.
- **Readability:** useState is very easy to read when the state updates are simple. When they get more complex, they can bloat your component's code and make it difficult to scan. In this case, useReducer lets you cleanly separate the *how* of update logic from the *what happened* of event handlers.
- **Debugging:** When you have a bug with useState, it can be difficult to tell *where* the state was set incorrectly, and *why*. With useReducer, you can add a console log into your reducer to see every state update, and *why* it happened (due to which action). If each action is correct, you'll know that the mistake is in the reducer logic itself. However, you have to step through more code than with useState.
- **Testing:** A reducer is a pure function that doesn't depend on your component. This means that you can export and test it separately in isolation. While generally it's best to test components in a more realistic environment, for complex state update logic it can be useful to assert that your reducer returns a particular state for a particular initial state and action.
- **Personal preference:** Some people like reducers, others don't. That's okay. It's a matter of preference. You can always convert between useState and useReducer back and forth: they are equivalent!

Use reducer if you often encounter bugs due to incorrect state updates in some component, and want to introduce more structure to its code. You don't have to use reducers for everything: feel free to mix and match! You can even useState and useReducer in the same component.

PASSING DATA DEEPLY WITH CONTEXT

Usually, you will pass information from a parent component to a child component via props. But passing props can become verbose and inconvenient if you have to pass them through many components in the middle, or if many components in your app need the same information. *Context* lets the parent component make some information available to any component in the tree below it—no matter how deep—without passing it explicitly through props.

You will learn

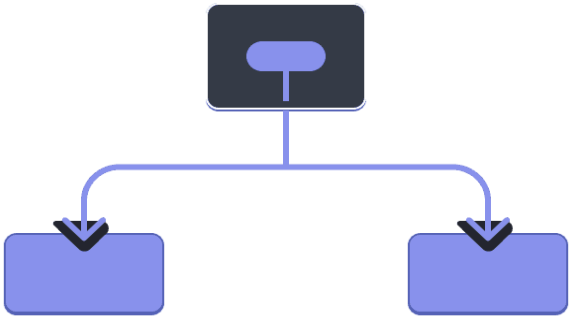
- What "prop drilling" is
- How to replace repetitive prop passing with context
- Common use cases for context
- Common alternatives to context

The problem with passing props

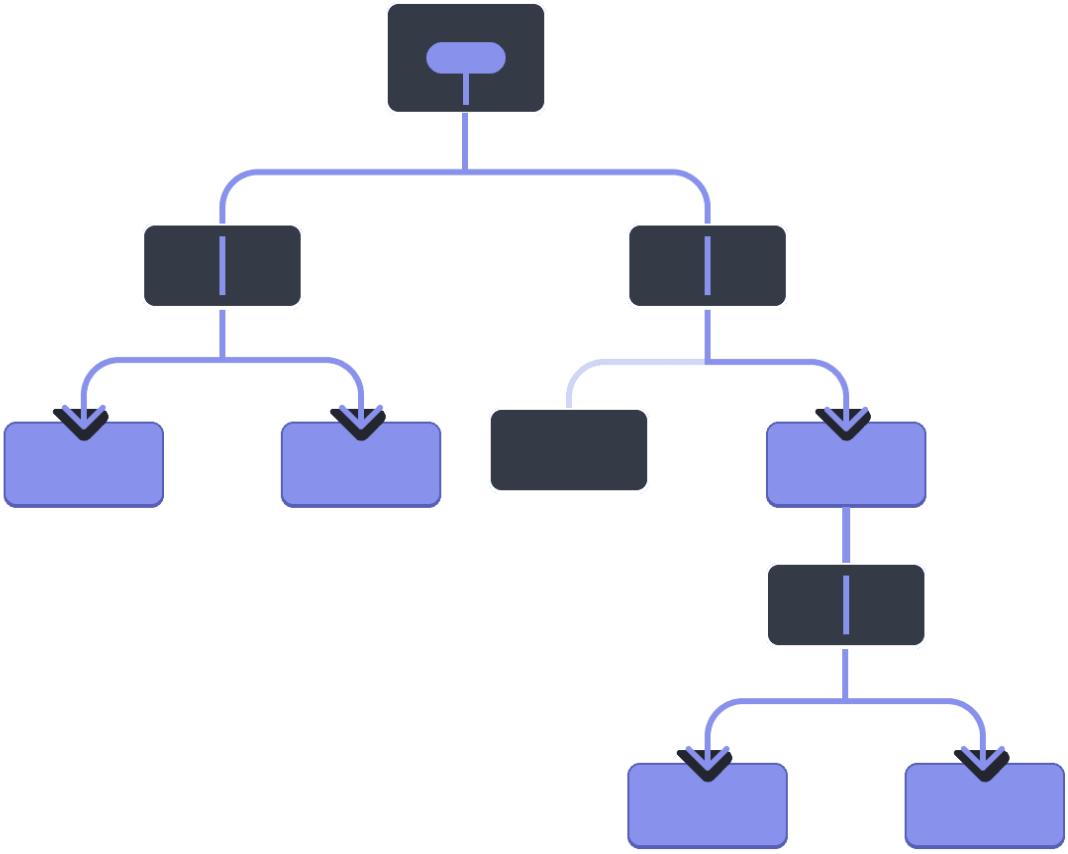
Passing props is a great way to explicitly pipe data through your UI tree to the components that use it.

But passing props can become verbose and inconvenient when you need to pass some prop deeply through the tree, or if many components need the same prop. The nearest common ancestor could be far removed from the components that need data, and lifting state up that high can lead to a situation called "prop drilling".

Lifting state up



Prop drilling



Wouldn't it be great if there were a way to "teleport" data to the components in the tree that need it without passing props? With React's context feature, there is!

Context: an alternative to passing props

Context lets a parent component provide data to the entire tree below it. There are many uses for context. Here is one example. Consider this Heading component that accepts a level for its size:

Heading.jsx

```
export default function Heading({ level, children }) {
  switch (level) {
    case 1:
      return <h1>{children}</h1>;
    case 2:
      return <h2>{children}</h2>;
    case 3:
      return <h3>{children}</h3>;
    case 4:
      return <h4>{children}</h4>;
    case 5:
      return <h5>{children}</h5>;
    case 6:
      return <h6>{children}</h6>;
    default:
      throw Error('Unknown level: ' + level);
  }
}
```

Section.jsx

```
export default function Section({ children }) {
  return (
    <section className="section">
      {children}
    </section>
  );
```

```
}
```

ContextPage.jsx

```
import Heading from './Heading.js';
import Section from './Section.js';

export default function Page() {
  return (
    <Section>
      <Heading level={1}>Title</Heading>
      <Heading level={2}>Heading</Heading>
      <Heading level={3}>Sub-heading</Heading>
      <Heading level={4}>Sub-sub-heading</Heading>
      <Heading level={5}>Sub-sub-sub-heading</Heading>
      <Heading level={6}>Sub-sub-sub-sub-heading</Heading>
    </Section>
  );
}
```

Let’s say you want multiple headings within the same Section to always have the same size:

```
import Heading from './Heading.js';
import Section from './Section.js';
```

```
export default function ContextPage() {
 return (
  <Section>
   <Heading level={1}>Title</Heading>
   <Section>
    <Heading level={2}>Heading</Heading>
    <Heading level={2}>Heading</Heading>
    <Heading level={2}>Heading</Heading>
    <Section>
     <Heading level={3}>Sub-heading</Heading>
     <Heading level={3}>Sub-heading</Heading>
     <Heading level={3}>Sub-heading</Heading>
     <Section>
      <Heading level={4}>Sub-sub-heading</Heading>
      <Heading level={4}>Sub-sub-heading</Heading>
      <Heading level={4}>Sub-sub-heading</Heading>
     </Section>
    </Section>
   </Section>
  </Section>
 );
}
```

Currently, you pass the level prop to each <Heading> separately:

```
<Section>
 <Heading level={3}>About</Heading>
```

```
  <Heading level={3}>Photos</Heading>
  <Heading level={3}>Videos</Heading>
</Section>
```

It would be nice if you could pass the level prop to the <Section> component instead and remove it from the <Heading>. This way you could enforce that all headings in the same section have the same size:

```
<Section level={3}>
  <Heading>About</Heading>
  <Heading>Photos</Heading>
  <Heading>Videos</Heading>
</Section>
```

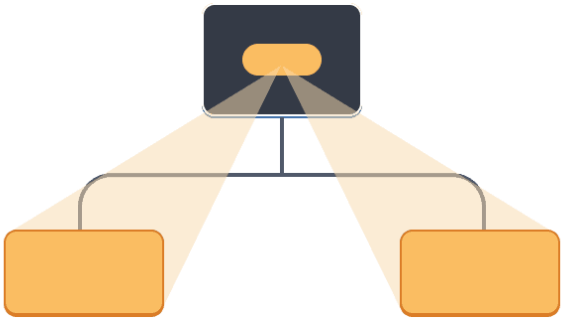
But how can the <Heading> component know the level of its closest <Section>? **That would require some way for a child to "ask" for data from somewhere above in the tree.**

You can't do it with props alone. This is where context comes into play. You will do it in three steps:

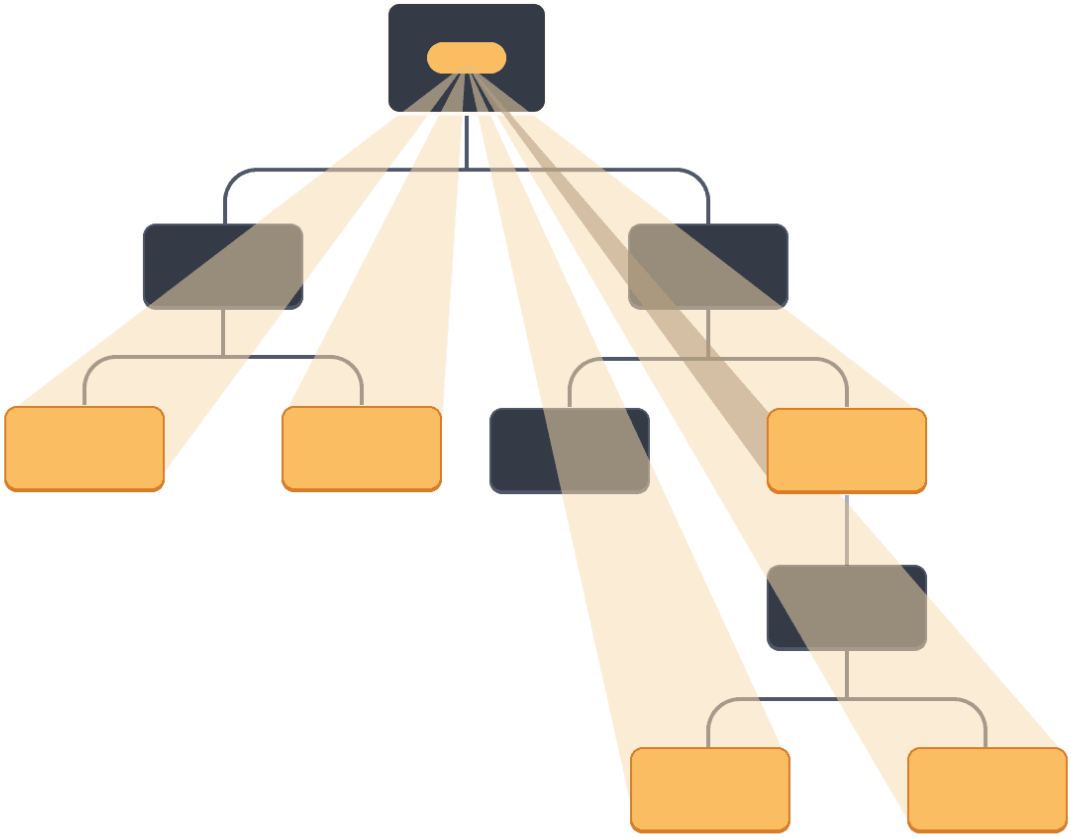
1. **Create** a context. (You can call it LevelContext, since it's for the heading level.)
2. **Use** that context from the component that needs the data. (Heading will use LevelContext.)
3. **Provide** that context from the component that specifies the data. (Section will provide LevelContext.)

Context lets a parent—even a distant one!—provide some data to the entire tree inside of it.

Using context in close children



Using context in distant children



Step 1: Create the context

First, you need to create the context. You'll need to **export it from a file** so that your components can use it:

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

export const LevelContext = createContext(1);
```

The only argument to createContext is the *default* value. Here, 1 refers to the biggest heading level, but you could pass any kind of value (even an object). You will see the significance of the default value in the next step.

Step 2: Use the context

Import the useContext Hook from React and your context:

```
import { useContext } from 'react';
import { LevelContext } from './LevelContext.js';
```

Currently, the Heading component reads level from props:

```
export default function Heading({ level, children }) {
  // ...
}
```

Instead, remove the level prop and read the value from the context you just imported, LevelContext:

```
export default function Heading({ children }) {
  const level = useContext(LevelContext);
  // ...
}
```

useContext is a Hook. Just like useState and useReducer, you can only call a Hook immediately inside a React component (not inside loops or conditions). **useContext tells React that the Heading component wants to read the LevelContext.**

Now that the Heading component doesn't have a level prop, you don't need to pass the level prop to Heading in your JSX like this anymore:

```
<Section>
  <Heading level={4} >Sub-sub-heading</Heading>
  <Heading level={4}>Sub-sub-heading</Heading>
  <Heading level={4}>Sub-sub-heading</Heading>
```

```
</Section>
```

Update the JSX so that it's the Section that receives it instead:

```
<Section level={4}>
  <Heading>Sub-sub-heading</Heading>
  <Heading>Sub-sub-heading</Heading>
  <Heading>Sub-sub-heading</Heading>
</Section>
```

As a reminder, this is the markup that you were trying to get working:

```
LevelContext.jsx
import { createContext } from 'react';
export const LevelContext = createContext(1);
```

```
Heading.jsx
import { useContext } from 'react';
import { LevelContext } from './LevelContext.js';

export default function Heading({ children }) {
  const level = useContext(LevelContext);
  switch (level) {
    case 1:
      return <h1>{children}</h1>;
    case 2:
      return <h2>{children}</h2>;
    case 3:
      return <h3>{children}</h3>;
```

```
    case 4:
      return <h4>{children}</h4>;
    case 5:
      return <h5>{children}</h5>;
    case 6:
      return <h6>{children}</h6>;
    default:
      throw Error('Unknown level: ' + level);
  }
}
```

Section.jsx

```
export default function Section({ children }) {
  return (
    <section className="section">
      {children}
    </section>
  );
}
```

ContextPage.jsx

```
import Heading from './Heading.js';
import Section from './Section.js';

export default function ContextPage () {
  return (
    <Section level={1}>
```

```
    <Heading>Title</Heading>
    <Section level={2}>
      <Heading>Heading</Heading>
      <Heading>Heading</Heading>
      <Heading>Heading</Heading>
      <Section level={3}>
        <Heading>Sub-heading</Heading>
        <Heading>Sub-heading</Heading>
        <Heading>Sub-heading</Heading>
        <Section level={4}>
          <Heading>Sub-sub-heading</Heading>
          <Heading>Sub-sub-heading</Heading>
          <Heading>Sub-sub-heading</Heading>
        </Section>
      </Section>
    </Section>
  </Section>
 );
}
```

Notice this example doesn't quite work, yet! All the headings have the same size because **even though you're *using* the context, you have not *provided* it yet.** React doesn't know where to get it!

If you don't provide the context, React will use the default value you've specified in the previous step. In this example, you specified 1 as the argument to createContext, so useContext(LevelContext) returns 1, setting all those

headings to <h1>. Let's fix this problem by having each Section provide its own context.

Step 3: Provide the context

The Section component currently renders its children:

```
export default function Section({ children }) {
  return (
    <section className="section">
      {children}
    </section>
  );
}
```

Wrap them with a context provider to provide the LevelContext to them:

```
import { LevelContext } from './LevelContext.js';
export default function Section({ level, children }) {
  return (
    <section className="section">
      <LevelContext value={level}>
        {children}
      </LevelContext>
    </section>
  );
}
```

This tells React: "if any component inside this <Section> asks for LevelContext, give them this level." The component will use the value of the nearest <LevelContext> in the UI tree above it.

It's the same result as the original code, but you did not need to pass the level prop to each Heading component! Instead, it "figures out" its heading level by asking the closest Section above:

1. You pass a level prop to the <Section>.
2. Section wraps its children into <LevelContext value={level}>.
3. Heading asks the closest value of LevelContext above with useContext(LevelContext).

Using and providing context from the same component

Currently, you still have to specify each section's level manually:

```
export default function Page() {
 return (
  <Section level={1}>
   ...
   <Section level={2}>
    ...
    <Section level={3}>
     ...
```

Since context lets you read information from a component above, each Section could read the level from the Section above, and pass level + 1 down automatically. Here is how you could do it:

LevelContext.jsx

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

export const LevelContext = createContext(0); //set default to 0 when level +1
```

```
import { useContext } from 'react';
import { LevelContext } from './LevelContext.js';
export default function Section({ children }) {
  const level = useContext(LevelContext);
  return (
    <section className="section">
      <LevelContext value={level + 1}>
        {children}
      </LevelContext>
    </section>
  );
}
```

Heading.jsx

```
import { useContext } from 'react';
import { LevelContext } from './LevelContext.jsx';
export default function Heading({  children }) {
  const level = useContext(LevelContext);

  switch (level) {
//case 0 added since in in LevelContext default value changed to 0 .so level+1
//will be <h1>
    case 0:
      throw Error('Heading must be inside a Section!');
    case 1:
      return <h1>{children}</h1>;
    case 2:
```

```
      return <h2>{children}</h2>;
    case 3:
      return <h3>{children}</h3>;
    case 4:
      return <h4>{children}</h4>;
    case 5:
      return <h5>{children}</h5>;
    case 6:
      return <h6>{children}</h6>;
    default:
      throw Error('Unknown level: ' + level);
  }
}
```

With this change, you don't need to pass the level prop *either* to the <Section> or to the <Heading>:

Now both Heading and Section read the LevelContext to figure out how "deep" they are. And the Section wraps its children into the LevelContext to specify that anything inside of it is at a "deeper" level.

Note

This example uses heading levels because they show visually how nested components can override context. But context is useful for many other use cases too. You can pass down any information needed by the entire subtree: the current color theme, the currently logged in user, and so on.

Context passes through intermediate components

You can insert as many components as you like between the component that provides context and the one that uses it. This includes both built-in components like and components you might build yourself.

In this example, the same Post component (with a dashed border) is rendered at two different nesting levels. Notice that the <Heading> inside of it gets its level automatically from the closest <Section>:

LevelContext.jsx

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

export const LevelContext = createContext(0);
```

Section.jsx

```
import { useContext } from 'react';
import { LevelContext } from './LevelContext.jsx';
export default function Section({ children, isFancy }) {
  const level = useContext(LevelContext);
  return (
    <section className={
      'section ' +
      (isFancy ? 'fancy' : '')
    }>
      <LevelContext value={level + 1}>
        {children}
      </LevelContext>
    </section>
  );
}
```

Heading.jsx

```
import { useContext } from 'react';
import { LevelContext } from './LevelContext.jsx';
export default function Heading({  children }) {
  const level = useContext(LevelContext);

  switch (level) {
    case 0:
     throw Error('Heading must be inside a Section!');
   case 1:
     return <h1>{children}</h1>;
   case 2:
     return <h2>{children}</h2>;
   case 3:
     return <h3>{children}</h3>;
   case 4:
     return <h4>{children}</h4>;
   case 5:
     return <h5>{children}</h5>;
   case 6:
     return <h6>{children}</h6>;
   default:
     throw Error('Unknown level: ' + level);
  }
}
```

ProfilePage.jsx

```
import Heading from './Heading.jsx';
import Section from './Section.jsx';
export default function ProfilePage() {
  return (
    <Section>
      <Heading>My Profile</Heading>
      <Post
        title="Hello traveller!"
        body="Read about my adventures."
      />
      <AllPosts />
    </Section>
  );
}

function AllPosts() {
  return (
    <Section>
      <Heading>Posts</Heading>
      <RecentPosts />
    </Section>
  );
}

function RecentPosts() {
```

```
 return (
  <Section>
   <Heading>Recent Posts</Heading>
   <Post
    title="Flavors of Lisbon"
    body="...those pastéis de nata!"
   />
   <Post
    title="Buenos Aires in the rhythm of tango"
    body="I loved it!"
   />
  </Section>
 );
}

function Post({ title, body }) {
 return (
  <Section isFancy={true}>
   <Heading>
    {title}
   </Heading>
   <p><i>{body}</i></p>
  </Section>
 );
}
```

You didn't do anything special for this to work. A Section specifies the context for the tree inside it, so you can insert a <Heading> anywhere, and it will have the correct size

Context lets you write components that "adapt to their surroundings" and display themselves differently depending on *where* (or, in other words, *in which context*) they are being rendered.

How context works might remind you of CSS property inheritance. In CSS, you can specify color: blue for a , and any DOM node inside of it, no matter how deep, will inherit that color unless some other DOM node in the middle overrides it with color: green. Similarly, in React, the only way to override some context coming from above is to wrap children into a context provider with a different value.

In CSS, different properties like color and background-color don't override each other. You can set all 's color to red without impacting background-color. Similarly, **different React contexts don't override each other.** Each context that you make with createContext() is completely separate from other ones, and ties together components using and providing *that particular* context. One component may use or provide many different contexts without a problem.

In CSS, different properties like color and background-color don't override each other. You can set all 's color to red without impacting background-color. Similarly, **different React contexts don't override each other.** Each context that you make with createContext() is completely separate from other ones, and ties together components using and providing *that particular* context. One component may use or provide many different contexts without a problem.

Before you use context

Context is very tempting to use! However, this also means it's too easy to overuse it. **Just because you need to pass some props several levels deep doesn't mean you should put that information into context.**

Here's a few alternatives you should consider before using context:

1. **Start by passing props.** If your components are not trivial, it's not unusual to pass a dozen props down through a dozen components. It may feel like a slog, but it makes it very clear which components use which data! The person maintaining your code will be glad you've made the data flow explicit with props.
2. **Extract components and pass JSX as children to them.** If you pass some data through many layers of intermediate components that don't use that data (and only pass it further down), this often means that you forgot to extract some components along the way. For example, maybe you pass data props like posts to visual components that don't use them directly, like <Layout posts={posts} />. Instead, make Layout take children as a prop, and render <Layout><Posts posts={posts} /></Layout>. This reduces the number of layers between the component specifying the data and the one that needs it.

If neither of these approaches works well for you, consider context.

Use cases for context

- **Theming:** If your app lets the user change its appearance (e.g. dark mode), you can put a context provider at the top of your app, and use that context in components that need to adjust their visual look.
- **Current account:** Many components might need to know the currently logged in user. Putting it in context makes it convenient to read it anywhere in the tree. Some apps also let you operate multiple accounts at the same time (e.g. to leave a comment as a different user). In those cases, it can be convenient to wrap a part of the UI into a nested provider with a different current account value.
- **Routing:** Most routing solutions use context internally to hold the current route. This is how every link "knows" whether it's active or not. If you build your own router, you might want to do it too.
- **Managing state:** As your app grows, you might end up with a lot of state closer to the top of your app. Many distant components below may want to change it. It is common to use a reducer together with context to manage complex state and pass it down to distant components without too much hassle.

Context is not limited to static values. If you pass a different value on the next render, React will update all the components reading it below! This is why context is often used in combination with state.

In general, if some information is needed by distant components in different parts of the tree, it's a good indication that context will help you.

Recap

- Context lets a component provide some information to the entire tree below it.
- To pass context:
    1. Create and export it with export const MyContext = createContext(defaultValue).
    2. Pass it to the useContext(MyContext) Hook to read it in any child component, no matter how deep.
    3. Wrap children into <MyContext value={...}> to provide it from a parent.
- Context passes through any components in the middle.
- Context lets you write components that "adapt to their surroundings".
- Before you use context, try passing props or passing JSX as children.

SCALING UP WITH REDUCER AND CONTEXT