REACT

SUCCINCTLY

BY **SAMER BUNA**

SUCCINCTLY EBOOK SERIES

Syncfusion®

React Succinctly

By

Samer Buna

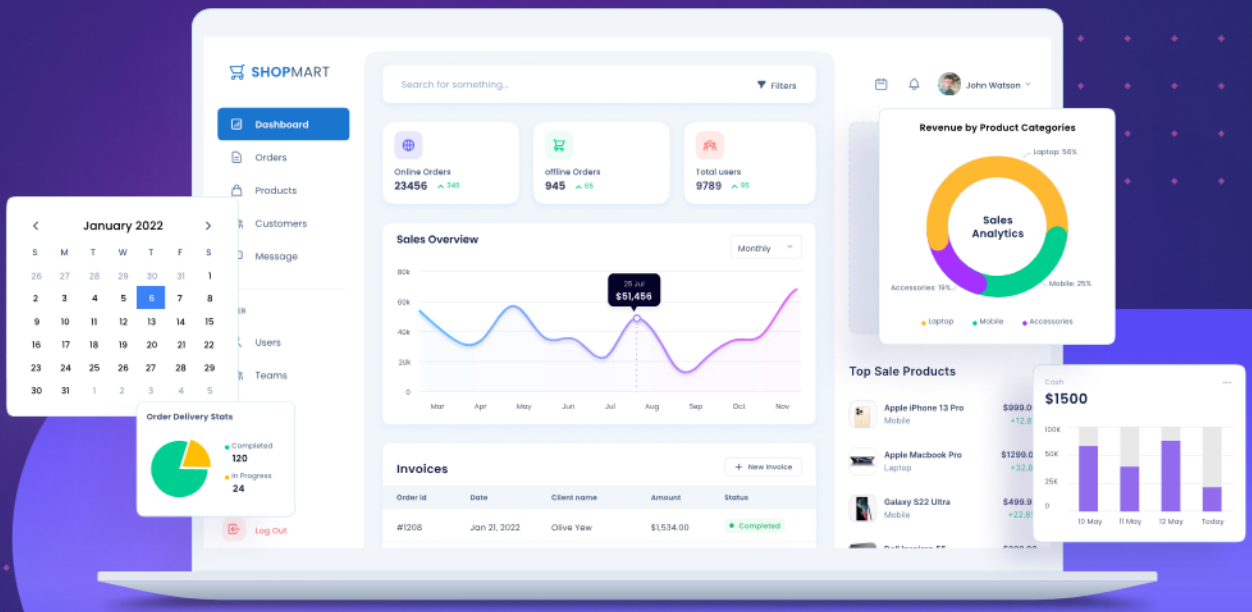
Foreword by Daniel Jebaraj

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA

Technical Reviewer: James McCaffrey
Copy Editor: Courtney Wright
Acquisitions Coordinator: Tres Watkins, content development manager, Syncfusion, Inc.
Proofreader: Graham High, senior content producer, Syncfusion, Inc.

THE WORLD'S BEST UI COMPONENT SUITE FOR BUILDING

POWERFUL APPS



GET YOUR FREE .NET AND JAVASCRIPT UI COMPONENTS

syncfusion.com/communitylicense

4.6 out of 5 stars

- 1,700+ components for mobile, web, and desktop platforms
- Support within 24 hours on all business days
- Uncompromising quality
- Hassle-free licensing
- 28000+ customers
- 20+ years in business

Trusted by the world's leading companies

Syncfusion

Table of Contents

The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Samer Buna has over 20 years of experience in software development, including web and mobile application development, API design, functional programming, optimization, system administration, database management, and scalability. He also has expertise in project management, agile practices, and teaching. He has worked in several industries including real estate, government, education, and publishing.

Samer has authored several technical books and online courses about JavaScript, Node, React, and more for publishers like Pluralsight, LinkedIn Learning, Manning, and others. Samer is passionate about everything JavaScript and he loves exploring new libraries. His favorite technical stack is PostgreSQL, GraphQL, Node, and React. You can follow his latest work at jsComplete.com. He's @samerbuna on Twitter and Medium.

Chapter 1 Introduction

You are going to be surprised how easy it is to learn the React API and start creating useful applications with it right away. This book is your first step. It will teach you the most important concepts of React, its main building blocks, and how to use it to build simple web applications.

This book will also teach you some of the popular, modern JavaScript concepts that are typically used with React. Learning and using modern JavaScript will make your React code better on many levels.

With modern JavaScript in your tool set, we'll go through one example application to understand the core theory of working with React components. After that, we'll put the knowledge into action and build a simple, timed memory challenge game where the player needs to recall the positions of cells in a grid. I built this one for my young children, and they love it.

Please note that this book will have a complete focus on the React library rather than any third-party packages that are usually used with React, like Redux React Router, or helper libraries like PropTypes. This does not mean that you should not use these libraries. It just means that for the duration of this book, your focus should be entirely on the React API. This focus will give you the understanding that's needed for you to correctly use packages designed for React.

Note: Whenever non-React JavaScript logic is needed throughout the book (for example, to sum or sample an array), I will provide a ready function for it. I will also provide starting HTML/CSS templates so that you do not get distracted with non-React needed steps.

You do need to learn more than just React to work with React, however. This is a good thing. React is a small library, and it is not the answer to everything. To name a few examples, you need to learn Node and many Node-based tools to run React applications, you need to learn how to work with data APIs to correctly make data available for React applications, and you need to learn many browsers' APIs like **`history`** and **`localStorage`** to correctly integrate React applications with what browsers can offer beyond the DOM.

During your learning process, the best thing you can possibly do is build stuff with your own hands. Do not copy and paste examples, and do not follow instructions blindly! Instead, mirror the instructions to build something else (ideally, something you care about). For example, after understanding the memory challenge game that we will build in this book, try to come up with different game ideas and implement them as well. After going through the memory challenge game example, try to improve it and add more features beyond what is presented in this book. Building new things with your own hands is the only truly helpful strategy when it comes to learning programming in general. React is no exception.

"You don't learn to walk by following rules. You learn by doing, and by falling over."

—Richard Branson

Is this book for you?

Although this book covers a little bit of modern JavaScript, it will not teach you the basics of the language. You need a good understanding of the JavaScript language before reading this book.

Nor will this book teach you anything about HTML/DOM and CSS. A good understanding of how these languages work is essential to getting good at React.

I like to be specific about the level of HTML and JavaScript that you need before learning React:

- If you do not know what the DOM is, or understand the concept of children and parent nodes, or how to define simple event handlers on DOM elements, you are probably not ready for React. React can be used without understanding the DOM API, but you will be a lot more productive in React if you know the basics. If you don't know anything about the DOM API, I strongly encourage you to read this introduction first.
- For JavaScript, if you do not know how to write `for` loops and `if` statements, or if you do not understand the concept of scopes and closures, you should probably learn these concepts in pure JavaScript before you dive into React. I have many resources for you about learning the JavaScript concepts you need to work with React applications. I have them all hosted here. Check them out first and make sure you understand them on their own before seeing them in action with React.

I am not saying you should not try to learn React if these concepts are new to you! I am just saying that these concepts will be needed when you work with React, and if you don't have a good understanding of them, you'll think that React is hard to learn. It is not. It just has some prerequisites that you might be lacking.

If you're coming to React with some previous knowledge of JavaScript but you have not used the modern features of the language that were added in the past few years, that is not a problem. In Chapter 3, I'll introduce you to features like arrow functions, destructuring, rest/spread operators, promises, and more.

You don't need to be an expert in JavaScript to write React applications, but you're likely to run into problems related to the JavaScript language syntax rather than the React API. I've put together a reference article about the common problems learners usually face when working with React, which you can read here. If you run into a problem while experimenting with React, make sure it's not one of the common problems listed in that article.

The jsComplete playground

You can follow along with all the examples in the book using the jsComplete React playground. This playground is a simple tool where you can test React (and JavaScript) code right in the browser, without any need to install, configure, import, or compile anything. I built this playground with React, and made it as simple as possible to remove any noise around learning the React API.

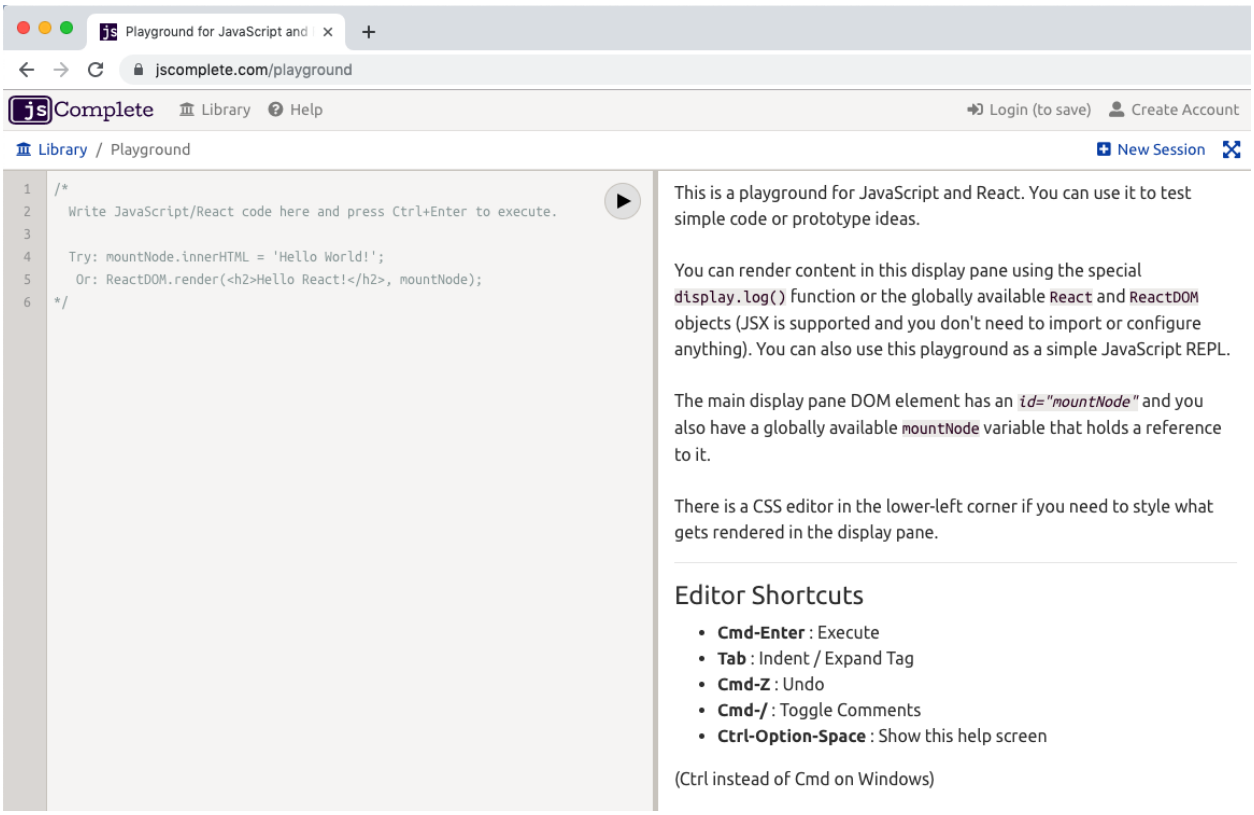


Figure 1: The jsComplete React playground interface

The playground has a two-column interface: an editor and a display. The latest version of React and its DOM renderer (which is named ReactDOM) are pre-loaded and available on the global scope. The editor understands JSX (the XML-like JavaScript syntax extension used with React) and the modern features in JavaScript. The display has a DOM element with an ID of **`mountNode`**. When you execute your JavaScript code, anything you put in the **`mountNode`** element shows up in the display. The display will also show any errors you encounter when you execute your code.

To execute the code at any time, press **Ctrl**+**Enter**. Try the following line, for example:

```
mountNode.innerHTML = 'Hello World!';
```

Or mount a simple React element with JSX:

```
ReactDOM.render(<h2>Hello React!</h2>, mountNode);
```

Note: Don't worry about this line yet. I'll explain everything about it in the next chapter.

Throughout the book, I'll be sharing saved code sessions using unique playground links so that you start from predefined snapshots. Use these links to work through the book's examples and compare your progress with the milestones completed in the book.

A playground tool like this one is a good start, but you'll eventually need to configure your own React environment. However, I think for the duration of this book you should keep your focus on learning the React API, and just use the playground. Once you're comfortable with React, you can use packages like **`create-react-app`** to generate a local environment, or you can configure your own. I wrote a guide on that topic, which you can read here, but don't go through it yet. Learn React first!

Note: Parts of this book were adapted from articles previously published on the jsComplete library and my Medium account. I am thankful to the many readers who provided early feedback on these articles and helped me make them better for future learners. Please do not hesitate in providing any further feedback about this book, good or bad! You can tweet me @samerbuna or find me at the jsComplete Help Slack channel.

Chapter 2 React: The Big Picture

React is defined as a JavaScript library for building user interfaces. Let's talk about the two different parts of this definition:

React is a JavaScript "library"

It is not exactly a "framework." It is not a complete solution, and you will often need to use more libraries with React to form any solution. React does not assume anything about the other parts in any solution.

Frameworks serve a great purpose, especially for young teams and startups. When working with a framework, many smart design decisions are already made for you, which gives you a clear path to focus on writing good application-level logic. However, frameworks come with some disadvantages. For experienced developers working on large codebases, these disadvantages are sometimes deal-breakers.

Frameworks are not flexible, although some claim to be. A framework usually wants you to code everything a certain way. If you try to deviate from that way, the framework usually ends up fighting you about it. Frameworks are also usually large and full of features. If you need to use only a small piece of them, you have to include the whole thing anyway. Admittedly, this point is in the process of changing today, but it is still not ideal. Some frameworks are going modular, which I think is great, but I am a big fan of the pure Unix philosophy:

> *"Write programs that do one thing and do it well. Write programs to work together."*
>
> *—Doug McIlroy*

React follows the Unix philosophy because it is a small library that focuses on just one thing, and on doing that thing extremely well. That "one thing" is the second part of React's definition: building user interfaces.

Building user interfaces

A user interface (UI) is anything we put in front of users to have them interact with a machine. UIs are everywhere, from the simple buttons on a microwave to the dashboard of a space shuttle. If the device we are trying to interface with can understand JavaScript, we can use React to describe a UI for it. Since web browsers understand JavaScript, we can use React to describe web UIs.

I like to use the word *describe* here because that is what we basically do with React—we just tell it what we want! React will then build the actual UI on our behalf in the web browser. Without React or similar libraries, we would need to manually build UIs with native web APIs and JavaScript, and that is not as easy.

When you hear the statement, "React is declarative," that is exactly what it means. We describe UIs with React and tell it what we want (not how to do it). React will take care of the "how" and translate our declarative descriptions (which we write in the React language) to actual UIs in the browser. React shares this simple declarative power with HTML itself, but with React we get to be declarative for HTML UIs that represent dynamic data, not just static data.

When React was released, there was a lot of buzz about its performance because it introduced the smart idea of a virtual DOM that can be used to reconcile the actual DOM (we'll talk about that in the next section). While React's performance is still one of the most important reasons why it is extremely popular today, I don't classify performance as the best thing about it. I think React was a game changer because it created a common language between developers and browsers that allows developers to declaratively describe UIs and manage transactions on their state, instead of transactions on their DOM elements. It's simply the language of user interface "outcomes." Instead of coming up with steps to describe transactions on interfaces, developers just describe the interfaces in terms of a "final" state (like a function). When transactions happen to that state, React takes care of updating the UIs in the DOM based on that (and it does it efficiently, as we'll see next).

If someone asked you to give one reason why React is worth learning, this outcomes-based UI language is the one. I call this language "the React language."

The React language

Say that we have a list of TODOs like this one:

Code Listing 1: The todos array

```
const todos: [
  { body: 'Learn React Fundamentals', done: true },
  { body: 'Build a TODOs App', done: false },
  { body: 'Build a Game', done: false },
];
```

This **`todos`** array is the starting state of your UI. You'll need to build a UI to display them and manage them. The UI might have a form to add new TODOs, a way for you to mark a TODO as done, and a way to remove all done TODOs. These are all features in standard TODOs. For example:

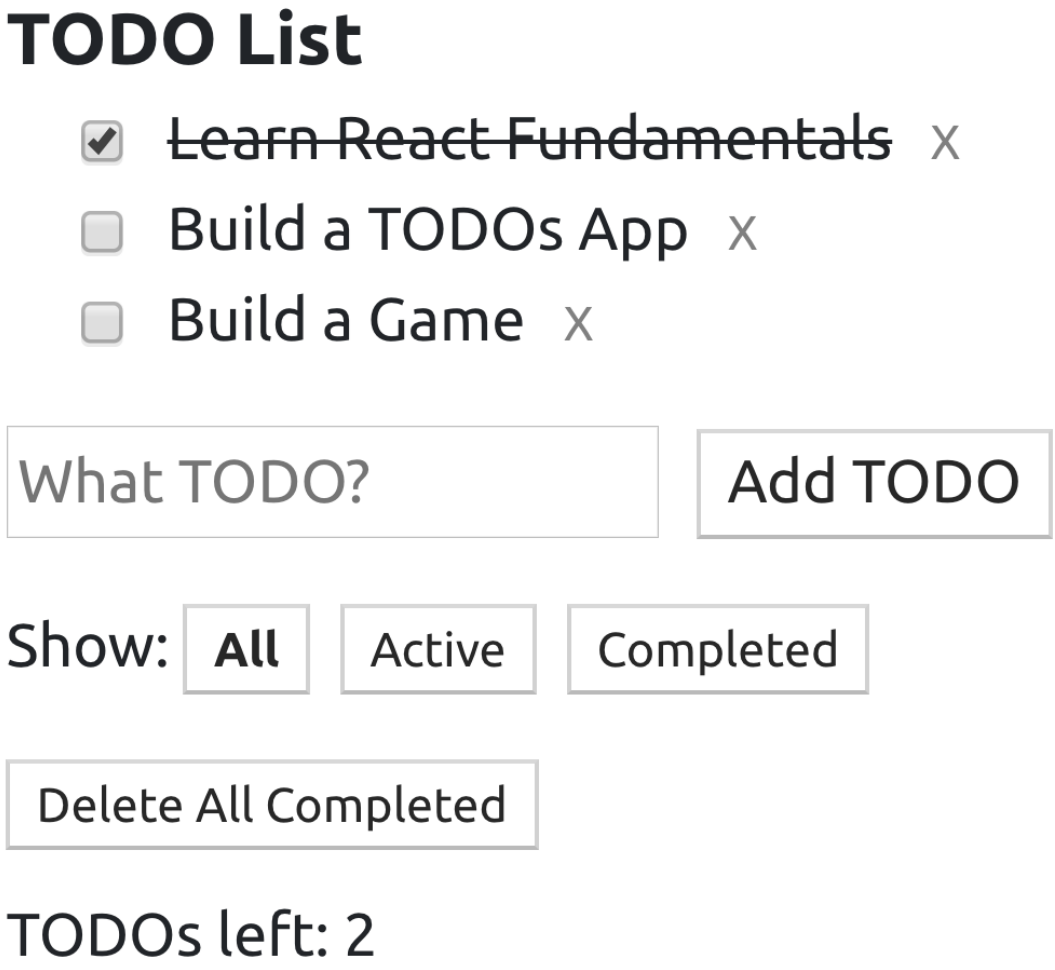


Figure 2: The UI for a TODOs example app

Each of these transactions will require the app to perform a DOM operation to create, insert, update, or delete DOM nodes. With React, you don't worry about all of these DOM operations. You don't worry about when they need to happen, and you don't worry about how to efficiently perform them.

With React, you just place the **`todos`** array in the "state" of your app, and then use the React language to "command" React to display that state a certain way in the UI:

Code Listing 2: The React language (don't worry about the syntax yet)

```
<header>TODO List</header>


<ul>
  {todos.map(todo =>
```

```
    <li>{todo.body}</li>
  )}
</ul>


// Other form elements
```

Note: The callback function for the `.map` method in Code Listing 2 is an inline arrow function that returns a React element. We mapped an array of JavaScript objects into an array of React elements. Both arrow functions and the map method are explained in the next chapter.

After coming up with this React language text, you get to focus on just doing transactions on that **`todos`** array! You add, remove, and update the items of that array, and React will reflect the changes you make on these items to the UI description that you wrote with the React language.

This mental model about modeling the UI based on the final state is much easier to understand and work with, especially when the views have lots of data transitions. For example, consider a view that tells you how many of your friends are online. That view's "state" will be just a single number that represents how many friends are currently online. It does not care that a moment ago three friends came online, then one of them disconnected, and then two more joined. It just knows that at this current moment, four friends are online.

React's tree reconciliation

Before React, when we needed to work with the browser's DOM API, we avoided traversing the DOM tree as much as possible, and there is a reason for that. Any operation on the DOM is done in the same single thread that's responsible for everything else that's happening in the browser, including reactions to user events like typing, scrolling, or resizing.

Any expensive operation on the DOM means a slow and janky experience for the user. It is extremely important that your applications do the absolute minimum operations and batch them where possible. React came up with a unique concept to help us do exactly that.

When we tell React to render a tree of elements in the browser, it first generates a *virtual* representation of that tree and keeps it around in memory for later. Then it'll proceed to perform the DOM operations that will make the tree show up in the browser.

When we tell React to update the tree of elements it previously rendered in the browser, it generates a new virtual representation of the updated tree. Now React has two versions of the tree in memory!

To render the updated tree in the browser, React does not discard what has already been rendered. Instead, it will compare the two virtual versions of the tree that it has in memory, compute the differences between them, figure out what sub-trees in the main tree need to be updated, and only update these sub-trees in the browser.

This process is what's known as the *tree reconciliation algorithm*, and it is what makes React a very efficient way to work with a browser's DOM tree. We'll see an example of it shortly.

Besides the declarative outcomes-based language and the efficient tree reconciliation, here are a few of the other reasons why I think React gained its massive popularity:

- Working with the DOM API is hard. React gives developers the ability to work with a "virtual" browser that is friendlier than the real browser. React basically acts like your agent who will do the communication with the DOM on your behalf.
- React is often given the "just JavaScript" label. This means it has a very small API to learn, and after that, your JavaScript skills are what make you a better React developer. This is an advantage over libraries with bigger APIs.
- Learning React pays off big-time for iOS and Android mobile applications as well. React Native allows you to use your React skills to build native mobile applications. You can even share some logic between your web, iOS, and Android applications.
- The React team at Facebook tests all improvements and new features that get introduced to React right there on facebook.com, which increases the trust in the library among the community. It's rare to see big and serious bugs in React releases because they only get released after thorough production testing at Facebook.

Your first React example

To see the practical benefit of the tree reconciliation process and the big difference it makes, let's work through a simple example focused on just that concept. Let's generate and update an HTML elements tree twice: once using the native web API, and then once using the React API (and its reconciliation work).

To keep this example simple, I will not use components or JSX (the JavaScript extension that's popularly used with React). I will also do the update operation inside a JavaScript interval timer. This is not how we write React applications, but let's focus on one concept at a time. We'll talk about components, JSX, and state managements in Chapter 4.

Start with this jsComplete playground session.

In that session, a simple HTML element is rendered to the display using two methods:

Code Listing 3: Method #1: Using the web DOM API directly

```
document.getElementById('mountNode').innerHTML = `
  <div>
    Hello HTML
```

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

Code Listing 4: Method #2: Using React's API

```
ReactDOM.render(
  React.createElement(
    'div',
    null,
    'Hello React',
  ),
  document.getElementById('mountNode2'),
);
```

The **`ReactDOM.render`** and **`React.createElement`** methods are the core API methods in a React application. In fact, a React web application cannot exist without using both of these methods. Let me briefly explain them.

ReactDOM.render

This is basically the entry point for a React application into the browser's DOM. It has two arguments:

- The first argument is WHAT to render to the browser. This is always a "React element."
- The second argument is WHERE to render that React element in the browser. This has to be a valid DOM node that exists in the statically rendered HTML. The previous example uses a special **`mountNode2`** element, which exists in the playground's display area (the first **`mountNode`** is used for the native version).

What exactly is a React element? It's a virtual element describing a DOM element. It's what the **`React.createElement`** API method returns.

React.createElement

Instead of working with strings to represent DOM elements (as in the native DOM example), in React we represent DOM elements with objects using calls to the **`React.createElement`** function. These objects are known as React elements.

The **`React.createElement`** function has the following arguments:

- The first argument is the HTML "tag" for the DOM element to represent, which is **`div`** in this example.
- The second argument is for any attributes (like **`id`**, **`href`**, or **`title`**) we want the DOM element to have. The simple **`div`** we're using has no attributes, so we used **`null`** in there.
- The third argument is the content of the DOM element. We've put a "Hello React" string in there. The third argument and all the optional arguments after it form the *children* list for the rendered element. An element can have 0 or more children. The **`div`** element in the example has a single text node child (so far).

Tip: `React.createElement` can also be used to create elements from React components. We will see examples of that in Chapter 4.

React elements are created in memory. To actually make a React element show up in the DOM, we use the **`ReactDOM.render`** method, which will do many things to figure out the most optimal way to reflect the state of a React element in the actual DOM tree in the browser.

When you execute the two methods in this code session, you'll see a "Hello HTML" box and a "Hello React" box:

Figure 3: The UI at jscomplete.com/playground/rs1.1

Nesting React elements

We have two nodes: one being controlled with the DOM API directly, and another being controlled with the React API (which in turn uses the DOM API). The only major difference between the ways we are building these two nodes in the browser is that in the HTML version, we used a string to represent the content, while in the React version, we used pure JavaScript calls and represented the content with an object instead of a string.

No matter how complicated the HTML user interface is going to get, when using React, every HTML element will be represented with a React element.

Let's add more HTML elements to this simple user interface. Let's add a text box to read input from the user. For the HTML version, you can just inject the new element's tag directly inside the template:

Code Listing 5

```
document.getElementById('mountNode').innerHTML = `
  <div>
    Hello HTML
    <input />
  </div>
`;
```

Tip: Throughout this book, when I make changes to code within related examples, these changes will be highlighted in a bold font in the code listing.

To do the same with React, you can add more arguments after the third argument for **`React.createElement`**. To match what's in the native DOM example so far, we can add a fourth argument that is another **`React.createElement`** call that renders an **`input`** element:

Code Listing 6

```
ReactDOM.render(
  React.createElement(
    "div",
    null,
    "Hello React ",
    React.createElement("input")
  ),
  document.getElementById('mountNode2'),
);
```

Let's also render the current time in both versions. Let's put it in a **`pre`** element (just to give it a monospace font in the playground). You can use **`new Date().toLocaleTimeString()`** to display a simple time string. Here's what you need to do for the native DOM version:

Code Listing 7

```
document.getElementById('mountNode1').innerHTML = `
  <div>
```

```
    Hello HTML
    <input />
    <pre>${new Date().toLocaleTimeString()}</p>
  </div>
`;
```

To do the same in React, we add a fifth argument to the top-level **`div`** element. This new fifth argument is another **`React.createElement`** call; this time using a **`pre`** tag with the **`new Date().toLocaleTimeString()`** call for content:

Code Listing 8

```
ReactDOM.render(
  React.createElement(
    "div",
    null,
    "Hello React ",
    React.createElement("input"),
    React.createElement(
      "pre",
      null,
      new Date().toLocaleTimeString()
    )
  ),
  document.getElementById('mountNode2'),
);
```

Both versions will now be rendering the exact same HTML in the browser:

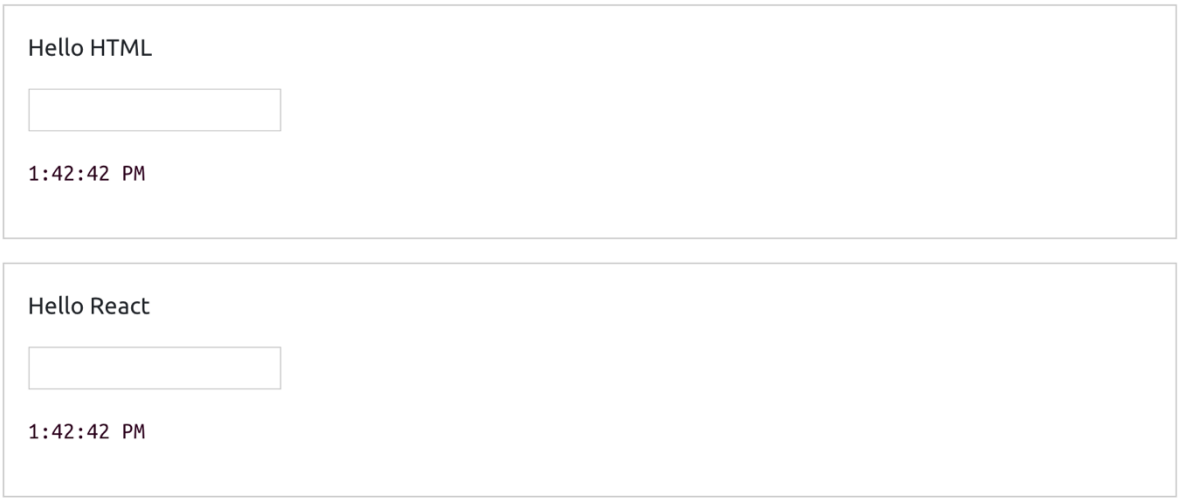


Figure 4: The UI after adding an input and a time string

As you're probably thinking by now, using React is a lot harder than the simple and familiar native way. What is it that React does so well that is worth giving up the familiar HTML and having to learn a new API to write what can be simply written in HTML?

The answer is not about rendering the first HTML view. It is about what we need to do to update any existing view in the DOM.

Updating React elements

Let's do an update operation on the DOM trees that we have so far. Let's simply make the time string tick every second.

We can easily repeat a JavaScript function call in a browser using the **`setInterval`** web timer API. Let's put all of our DOM manipulations for both versions into a function, name it **`render`**, and use it in a **`setInterval`** call to make it repeat every second.

> ***Tip: If you're not familiar with `setInterval`, the next chapter has more details and examples about it.***

Here is the full final code for this example:

Code Listing 9: Code available at jscomplete.com/playground/rs1.2

```
const render = () => {
  document.getElementById('mountNode').innerHTML = `
    <div>
      Hello HTML
```

```
      <input />
      <pre>${new Date().toLocaleTimeString()}</pre>
    </div>
  `;

  ReactDOM.render(
    React.createElement(
      'div',
      null,
      'Hello React',
      React.createElement('input', null),
      React.createElement('pre', null, new Date().toLocaleTimeString())
    ),
    document.getElementById('mountNode2')
  );
};

setInterval(render, 1000);
```

Check out the result of executing this code here, and notice how the time string is ticking every second in both versions. We are now updating our user interface in the DOM.

This is the moment when React will potentially blow your mind. If you try to type something in the text box of the native DOM version, you will not be able to. This is very much expected because we are basically throwing away the whole DOM node on every tick and regenerating it. However, if you try to type something in the text box that is rendered with React, you can certainly do so!

Although all the React rendering code is within the ticking timer, React is changing only the content of the **pre** element, not the whole DOM tree. This is why the text input box was not regenerated and we were able to type in it.

You can see the different ways we are updating the DOM visually if you inspect the two DOM nodes in a Chrome DevTools elements panel. The Chrome DevTools elements panel highlights any DOM elements that get updated. You will see how we are regenerating the entire **`mountNode`** element (which is the native version) with every tick, while React is smartly only regenerating the **`pre`** tag in the **`mountNode2`** element.

Figure 5: React updates only the pre element

This is React's smart *diffing* algorithm in action. It only updates in the main DOM tree what actually needs to be updated, while it keeps everything else the same. This diffing process is possible because of React's virtual DOM representation that it keeps around in memory. No matter how many times we regenerate our interface, React will take to the browser only the needed "partial" updates.

Not only is this method a lot more efficient, but it also removes a big layer of complexity in the way we think about updating user interfaces. Having React do all the computations about whether we should or should not update the DOM enables us to focus on thinking about our data (state) and the way to describe a user interface for it. We then manage the updates on the data state as needed, without worrying about the steps needed to reflect these updates in the actual user interface in the browser (because we know React will do exactly that, and it will do it in an efficient way).

Chapter 3 React and Modern JavaScript

React developers love the modern features in JavaScript and use them extensively in their projects. In this chapter, I'll go over the most popular features that are usually used with React. Most of these features are modern to JavaScript, but I'll also talk about some older ones that are related and important for React. This will NOT be a complete list of everything offered by the language, but rather the subset that I think will help you write better code for React.

Note: You can skip this chapter if you're already comfortable with JavaScript and its modern features like destructuring, rest/spread syntax, promises, etc.

Block scopes and the var/let/const keywords

A block scope is created with a pair of curly brackets. This happens every time you create an **`if`** statement, **`for`** statement, **`while`** statement, etc. The only exception is the curly brackets you use with functions. These create a function scope, not a block scope.

Code Listing 10: Block and function scopes

```
{
  // Block Scope
}

if (true) {
  // Block Scope
}

for (var i = 1; i <= 10; i++) {
  // Block Scope
}

function doSomething() {
  // Function Scope
}
```

Function scopes are created for each function (like **doSomething** in Code Listing 10). One difference they have from block scopes is obvious when using the **var** keyword. Variables defined with **var** inside a function scope are OK; they don't *leak* out of that scope. If you try to access them outside of the scope, you can't:

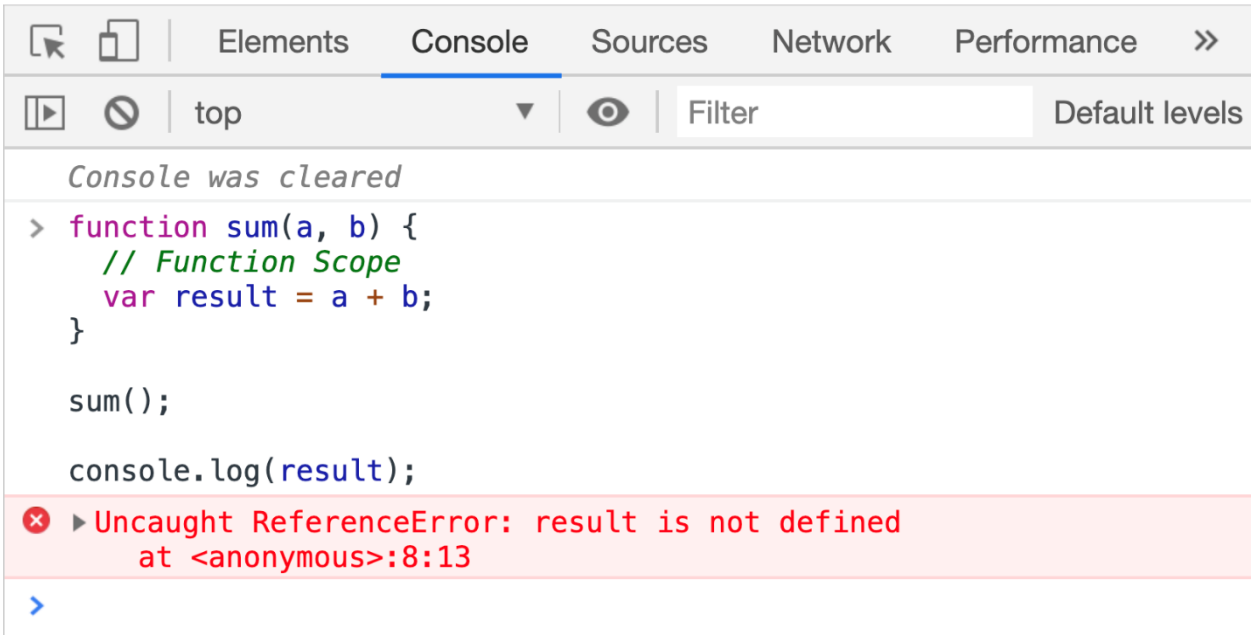


Figure 6: var in a function scope

However, when you define variables with **var** in a block scope, you can totally access them outside that scope afterward, which is a bit problematic. For example, in a standard for-loop statement, if the loop variable is defined with **var**, you can access that variable after the loop is done.

Figure 7: var in a block scope

This is why the recommended way to declare variables in modern JavaScript is by using the **let** keyword instead of the **var** keyword. When defining variables with **let**, we won't have this weird out-of-scope access problem.

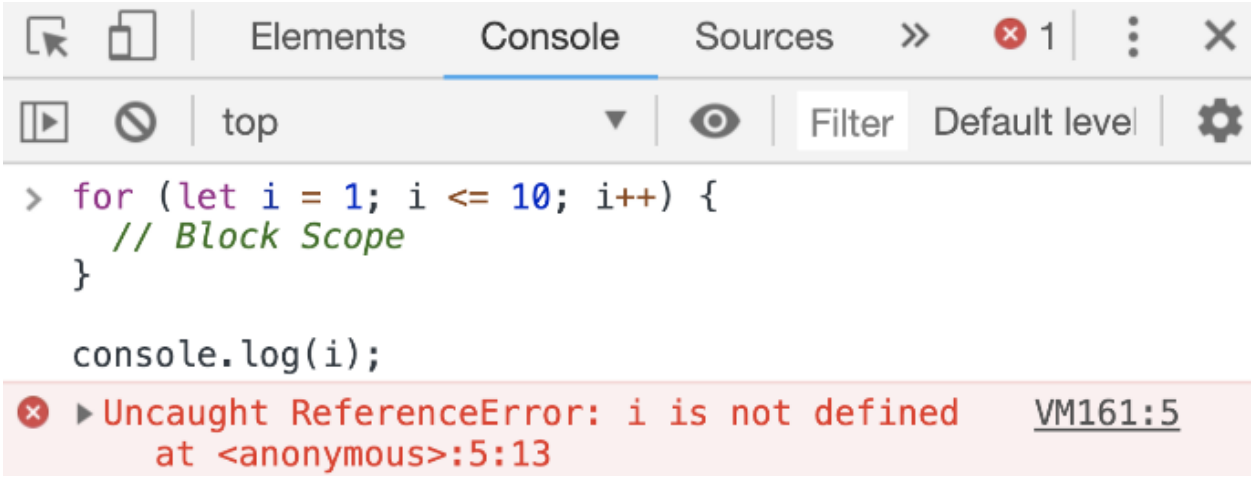


Figure 8: let in a block scope

However, you should use the **`let`** keyword only when the variable's value needs to be changed. This should not be a common thing in your code. For most other cases you should use the **`const`** keyword instead, so let me tell you about it.

In JavaScript, a variable is basically a label we put on a certain space in the computer's memory.

```
let V = { id: 42 }; // create a memory unit and label it as V
```

When you change the value of the variable **`V`** you are not really changing the content of the memory space that was initially associated with **`V`**. Instead, you're creating a new memory space and changing the **`V`** label to be associated with that new space.

Code Listing 11

```
// Discard current memory unit (and its current label)
// Create new memory unit and label it as V

V = []; // No errors
```

When you use **`const`** to define a variable, you are instructing the computer to not only label a space in memory, but to also never change that label. The label will be forever associated with its same space in memory.

Code Listing 12

```
// Create a memory unit and label it as V
// This label cannot be discarded or reused
```

```
const V = { id: 42 };


// Later in the program

V = []; // TypeError: Assignment to constant variable.
```

Note that the constant part here is just the label. The value of what's in the memory space can still change (if it's mutable). For example, objects in JavaScript are mutable, so for the **`V`** in Code Listing 12:

Code Listing 13

```
// You can do:

V.id = 37; // No errors


console.log(V.id); // 37


// But you still can't do:

V = { id: 37 }; // TypeError: Assignment to constant variable.
```

This applies to arrays too (because they are mutable as well).

Strings and integers are immutable in JavaScript, so the only way to change a string or integer value in JavaScript is to discard the current memory space and re-label another one. That's why if you have a numeric counter that you need to increment in your program, you would need to use **`let`**:

Code Listing 14: A use case for let

```
// Can't use const for this case:

let counter = 0;


counter = counter + 1; // Discard and re-label
```

Always use the **`const`** keyword to define variables. Only use the **`let`** keyword when you absolutely need it. Never use the **`var`** keyword.

Arrow functions and closures

Arrow functions are probably the most-used feature in modern JavaScript. Here's what they look like:

Code Listing 15: The arrow function syntax

```
const doSomething = () => {
  // Function Scope

};
```

This new "shorter" syntax to define functions is popular not only because it's shorter, but also because it behaves more predictably with closures. Arrow functions give access to their *defining* environment while regular functions give access to their *calling* environment. This access is possible through the special **`this`** keyword in a function's scope:

- The value of the **`this`** keyword inside a regular function depends on *how* the function was *called*.
- The value of the **`this`** keyword inside an arrow function depends on *where* the function was *defined***.**

Here is a code example to expand on the explanation. Try to figure out what will be printed in Output #1 through #4 (last four lines):

Code Listing 16: Code available at jscomplete.com/playground/arrow-functions

```
this.whoIsThis = 'TOP'; // Identify this scope


// 1) Defining
const fancyObj {
  whoIsThis: 'FANCY', // Identify this object
  regularF: function () {
    console.log('regularF', this.whoIsThis);
  },
  arrowF: () => {
    console.log('arrowF', this.whoIsThis);
  },
};
```

```
// 2) Calling

console.log('TOP-LEVEL', this.whoIsThis); // It's "TOP" here


fancyObj.regularF(); // Output #1 (Fancy)

fancyObj.arrowF();   // Output #2 (Top)


fancyObj.regularF.call({whoIsThis: 'FAKE'}); // Output #3 (Fake)

fancyObj.arrowF.call({whoIsThis: 'FAKE'});   // Output #4 (Top)
```

This example has a regular function (**regularF**) and an arrow function (**arrowF**) defined in the same environment and called by the same caller. Here's the explanation of the outputs in the last four lines:

- The **regular** function will always use its **this** to represent who called it. In the previous example, the caller of both functions was the **fancyObj** itself. That's why Output #1 was **FANCY**.
- The arrow function will always print the **this** scope that was available at the time it was defined. That's why Output #2 was **TOP**.
- The functions **.call**, **.apply**, and **.bind** can be used to change the calling environment. Their first argument becomes the new "caller." That's why Output #3 was **FAKE**.
- The arrow function does not care about the **.call** caller change. That's why Output #4 was **TOP**, and not the new **FAKE** caller.

One other cool thing about arrow functions is that if the function only has a single return line:

Code Listing 17

```
const square = (a) => {

  return a * a;

};
```

You can make it even more concise by removing the curly brackets and the **return** keyword altogether.

```
const square = (a) => a * a;
```

You can also remove the parentheses around the argument if the function receives a single argument:

```
const square = a => a * a;
```

This much shorter syntax is usually popular for functions that get passed to array methods like **`map`**, **`reduce`**, **`filter`**, and other functional programming methods:

```
console.log([1, 2, 3, 4].map(a => a * a));
```

Note: The `map`, `filter`, and `reduce` methods are explained in the last section of this chapter.

Note that if you want to use the one-line version of the arrow function to make a function that returns an object, you'll have to enclose the object in parentheses, because otherwise the curly brackets will actually be for the scope of the function.

```
// Wrong
const objMaker = () => { answer: 42 };


// Right
const objMaker = () => ({ answer: 42 });
```

This is actually one of the most common mistakes beginners make when working with libraries like React.

Arrow functions are short and more readable. They give access to their defining environments, making them ideal for cases when you need the function to be executed in a different environment than the one where it was defined (think timers or click event handlers).

The literal notations

You can create a JavaScript object in a few different ways, but the most common way is with an object literal (using curly brackets):

Code Listing 18: The object literal

```
const obj = {
  // key: value

};
```

This literal notation (also known as an initializer notation) is very common. We use it for objects, arrays, strings, numbers, and even things like regular expressions!

For arrays, the literal syntax uses a set of square brackets `[]`:

Code Listing 19: The array literal

```
const arr = [item0, item1, item2, ...];
```

For strings, you can use either single quotes or double quotes:

```
const greeting = "Hello World";

const answer = 'Forty Two';
```

These two ways to define string literals in JavaScript are equivalent. Modern JavaScript has a third way to define strings using the backtick character:

```
const html = `
  <div>
    ${Math.random()}
  </div>
`;
```

Paste that in your browser's console and see how it forms a multiline string that has a random value:

Figure 9: JavaScript template literal

Strings defined with the backtick character are called template strings because they can be used as a template with dynamic values. They support string interpolation. You can inject any JavaScript expression within the **${}** syntax.

With template strings, you can also have multiple lines in the string, something that was not possible with the regular-quoted strings. You can also "tag" template strings with a function and have JavaScript execute that function before returning the string, which is a handy way to attach logic to strings. This tagging feature is used in the popular styled-components library (for React).

Tip: Backticks look very similar to single quotes. Make sure to train your eyes to spot template strings when they are used.

Expressions for React

In React, there is a syntax similar to the template literal syntax that you can use to dynamically insert a JavaScript expression into your React component's code. It looks like this:

Code Listing 20: JSX expressions

```
// Somewhere in a React component's return value


<div>
  {Math.random()}
</div>
```

This is NOT a JavaScript template literal. These curly brackets in React are how you can insert dynamic expressions in JSX. You don't use a **$** sign with them, although you can still use JavaScript template strings elsewhere in a React application (including anywhere within JSX curly brackets). This might be confusing, so here's an example that uses both JSX curly brackets and JavaScript template literal curly brackets in the same line:

Code Listing 21: JSX expression with JS template literals

```
<div>
  {`Random value is: ${Math.random()}`}
</div>
```

The bolded part is the JavaScript template literal, which is an expression. We're evaluating that expression within JSX curly brackets.

Destructuring arrays and objects

The destructuring syntax is simple, but it makes use of the same curly and square brackets you use with object or array literals, which makes it confusing sometimes. You need to inspect the context to know whether a set of curly brackets (**{}**) or square brackets (**[]**) are used as literal initializing or destructuring assignment.

Code Listing 22: Curly brackets multiuse

```
const PI = Math.PI;
console.log({ PI });
const fn = ({ PI }) => {}
```

In Code Listing 22, the first { **PI** } (in the second line) is an object literal that uses the **PI** constant defined in the first line. The second { **PI** } (in the last line) is a destructuring assignment that has nothing to do with the **PI** variable defined in the first line.

It can get a lot more confusing than that, but here is a simple general rule to identify what's what:

When brackets appear on the left-hand side of an assignment or within the parentheses used to define a function, they are most likely used for destructuring. There are exceptions to this rule, but these exceptions are rare.

Here's an example of destructuring:

Code Listing 23: Destructuring arrays and objects

```
// 1) Destructure array items

const [first, second,, fourth] = [10, 20, 30, 40];

// 2) Destructure object properties

const { PI, E, SQRT2 } = Math;
```

These are both destructuring, because the brackets are on the left-hand side of the assignment.

Destructuring simply extracts named items out of an array (using their position) or properties out of an object (using their names) and into local variables in the enclosing scope. The two lines in Code Listing 23 are equivalent to:

Code Listing 24: The equivalent of destructuring arrays and objects

```
// 1) assuming arr is [10, 20, 30, 40]

const first = arr[0];
const second = arr[1];
// third element skipped
```

```
const fourth = arr[3];

// 2)

const PI = Math.PI;
const E = Math.E;
const SQRT2 = Math.SQRT2;
```

This is useful when you need to use a few properties out of a bigger object. For example, here's a line to destructure the **`useState`** and **`useEffect`** Hook functions out of the React API.

```
const { useState, useEffect } = React;
```

After this line, you can use these React API objects directly:

Code Listing 25: Destructuring in React

```
const [state, setState] = useState();

useEffect(() => {
  // do something

});
```

Note how the two items in the **`useState`** function's return value (which is an array of exactly two items) were also destructured into two local variables. We'll see examples of **`useState`** (and **`useEffect`**) in the upcoming chapters.

When designing a function to receive objects and arrays as arguments, you can also use destructuring to extract named items or properties out of them and into local variables in the function's scope. Here's an example:

Code Listing 26: Destructuring arguments

```
const circle = {
  label: 'circleX',
  radius: 2,
};

const circleArea = ({ radius }, [precision = 2]) =>
  (Math.PI * radius * radius).toFixed(precision);

console.log(
  circleArea(circle, [5]) // 12.56637
);
```

The **circleArea** function is designed to receive an object in its first argument, and an array in its second argument. These arguments are not named, and not used directly in the function's scope. Instead, their properties and items are destructured and used in the function's scope. You can even give destructured elements default values (as it's done for the **precision** item).

In JavaScript, using destructuring with a single object as the argument of a function is an alternative to named arguments (which are available in other languages). It is less error-prone than relying on positional arguments.

The rest/spread syntax

Destructuring gets more interesting (and useful) when combined with the rest syntax and the spread syntax, which are both done using the three dots (**...**) syntax. However, they do different things.

The rest syntax is what you use with destructuring. The spread syntax is what you use in object/array literals.

Here's an example:

Code Listing 27: The rest syntax

```
const [first, ...restOfItems] = [10, 20, 30, 40];
```

The three dots here, because they are in a destructuring call, represent a rest syntax. We are asking JavaScript here to destructure only one item out of this array (the first one) and then create a new array under the name **restOfItems** to hold the rest of the items (after removing the first one).

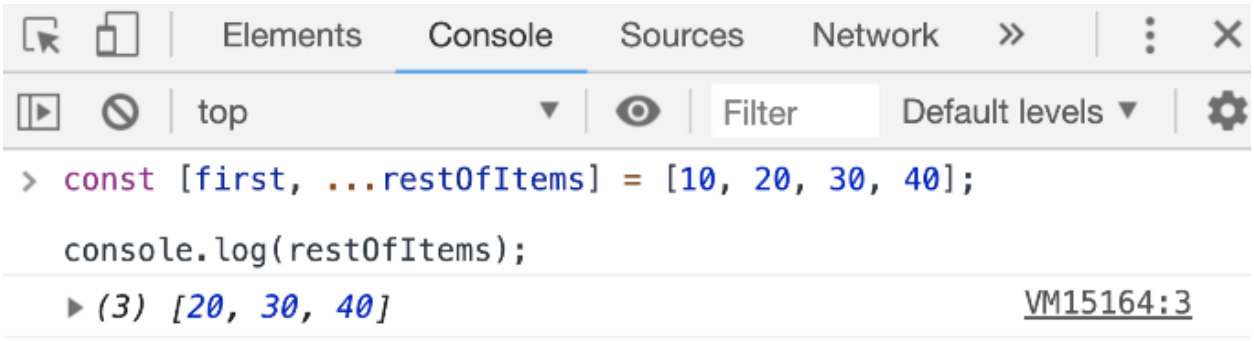


Figure 1: Using the rest syntax to extract an array

This is powerful for splitting the array, and it's even more powerful when working with objects to filter out certain properties from an object. For example, given this object:

Code Listing 28

```
const obj1 = {
```

```
  temp1: '001',
  temp2: '002',
  firstName: 'John',
  lastName: 'Doe',
  // many other properties
};
```

If you need to create a new object that has all the properties of **obj1** except for **temp1** and **temp2**, what would you do?

You can simply destructure **temp1** and **temp2** (and ignore them), and then use the rest syntax to capture the remaining properties into a new object:

Code Listing 29: The rest syntax in objects

```
const { temp1, temp2, ...obj2 } = obj1;
```

How cool is that?

The spread syntax uses the same three dots to *shallow-copy* an array or an object into a new array or an object. This is commonly used to merge partial data structures into existing ones. It replaces the need to use the **Object.assign** method.

Code Listing 30: The spread syntax in arrays and objects

```
const array2 = [newItem0, ...array1, newItem1, newItem2];

const object2 = {
  ...object2,
  newP1: 1,
  newP2: 2,
};
```

Tip: When using the spread syntax with objects, a property-name conflict will resolve to taking the value of the last property.

What is a shallow copy? Simply put, any nested arrays or objects will be shared between the copies. This is a similar story to memory spaces and their labels, except here labels are cloned and made to label the exact same memory spaces.

In React, the same three dots are used to spread an object of "props" for a component call. The JavaScript spread syntax was inspired by React (and others), but the usage of the three dots in React/JSX and in JavaScript is a little bit different. For example, given that a component **`X`** has access to an object like:

```
const engine = { href: "http://google.com", src: "google.png" };
```

That component can render another component **`Y`** and spread the properties of the **`engine`** object as props (attributes) for **`Y`**:

Code Listing 31: The spread syntax in JSX

```
<Y {...engine} />
```

This is equivalent to doing:

```
<Y href={engine.href} src={engine.src} />
```

Note that the curly brackets in Code Listing 31 are the JSX curly brackets.

Shorthand and dynamic properties

Here are a few things you can do with object literals in modern JavaScript:

Code Listing 32: Modern features in object literals

```
const mystery = 'answer';
const InverseOfPI = 1 / Math.PI;


const obj = {
  p1: 10,         // Plain old object property (don't abbreviate)

  f1() {},        // Define a shorthand function property

  InverseOfPI,    // Define a shorthand regular property

  f2: () => {},  // Define an arrow function property
```

```
  [mystery]: 42, // Define a dynamic property
};
```

Did you notice **`mystery`**? It's not an array or a destructuring thing—it's how you define a *dynamic property*.

Interview question: Given the code from Code Listing 32, what is the value of **`obj.mystery`**?

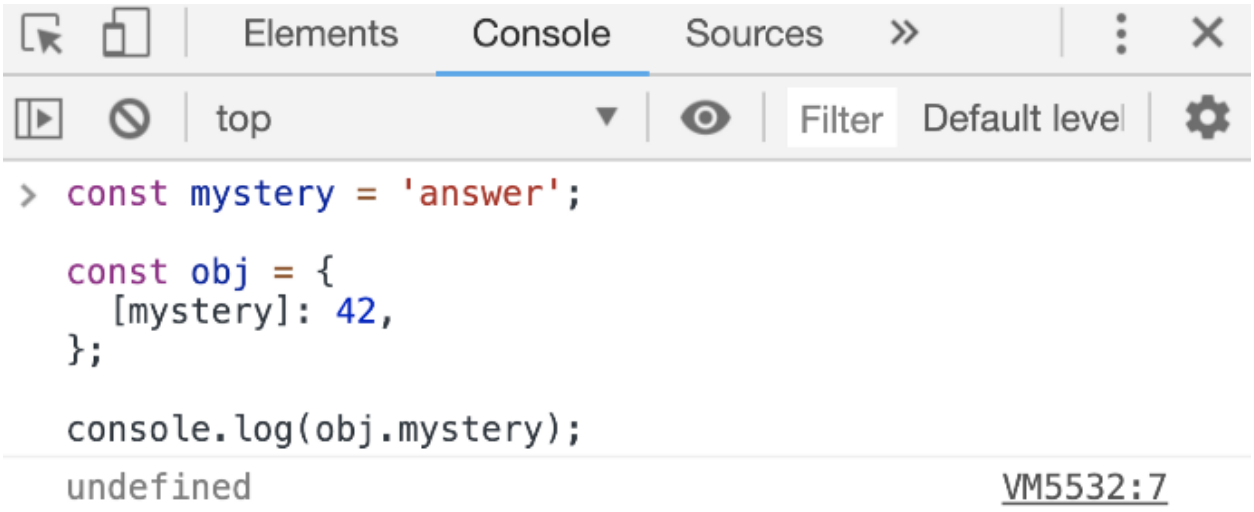


Figure 2: The answer is NOT 42

When you use the dynamic property syntax, JavaScript will first evaluate the expression inside **`[]`**, and whatever that expression evaluates to becomes the object's new property.

In this example, the **`obj`** object will have a property **`answer`** with the value of **`42`**.

Another widely popular feature of object literals is available to you when you need to define an object with property names to hold values that exist in the current scope with the exact same names. You can use the shorthand property name syntax for that. That's what we did for the **`InverseOfPI`** variable previously. That part of the object is equivalent to:

```
const obj = {
  InverseOfPI: InverseOfPI,
};
```

Objects are very popular in JavaScript. They are used to manage and communicate data, and using their modern literal features will often make your code a bit shorter and easier to read.

Promises and async/await

When you need to work with asynchronous operations, you usually have to deal with promise objects. A promise is an object that *might* deliver data at a later point in the program, or it might crash and deliver an error instead.

An example of an async function that returns a promise is the web **`fetch`** API that's natively available in some browsers.

Code Listing 33: The promise API

```
const fetchData = () => {
  fetch('https://api.github.com').then(resp => {
    resp.json().then(data => {
      console.log(data);
    });
  });
};
```

The **`fetchData`** function fetches information from the top-level GitHub API. Since **`fetch`** returns a promise, to consume that promise we do a **`.then`** call on the result of **`fetch`** and supply a *callback* function in there. The callback function will receive the raw response from the API. If you need to parse the data as JSON, you need to call the **`json()`** method on the raw response object. That **`json()`** method is also an asynchronous one, so it returns a promise as well. To get to the data, you need another **`.then`** call on the result of the **`json()`** method, and in the callback of that, you can access the parsed data.

As you can see, this syntax might get complicated with more nesting of asynchronous operations, or when you need to combine this with any looping logic. You can simplify the nesting from the previous example by making each promise callback return the promise object, but the whole **`.then`** syntax is a bit less readable than the modern way to consume promises in JavaScript, which is to use **`async`**/**`await`**:

Code Listing 34: Using async/await

```
const fetchData = async () => {
  const resp = await fetch('https://api.github.com');
  const data = await resp.json();
  console.log(data);
};
```

You just use **`await`** on the async call (the one that returns a promise), and that will give you back the response object directly. Then, you can use **`await`** on the **`json()`** method to access the parsed JSON data. To make **`await`** calls work, you just need to label the function as **`async`**.

The **`async`**/**`await`** syntax is just another way for you to consume promises (but without having to deal with **`.then`** calls). It's a bit simpler to read, but keep in mind that once you *await* on anything in a function, that function itself becomes asynchronous, and it will return a promise object (even if you don't return anything from it).

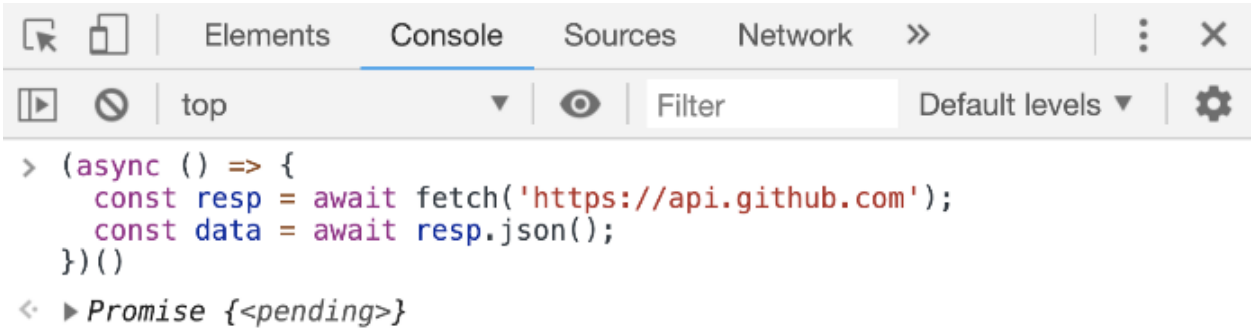


Figure 3: The IIFE had no return statement, but a promise was returned anyway

For error handling (when promises reject, for example), you can combine the **`async`**/**`await`** syntax with the plain-old **`try`**/**`catch`** statement (and you should do that all the time).

Modules import/export

Modern JavaScript introduced the **`import`** and **`export`** statements to provide a solution for "module dependency management," which is just a fancy term to describe JavaScript files that need each other.

A file **`X.js`** that needs to use a function from file **`Y.js`** can use the **`import`** statement to declare this dependency. The function in **`Y.js`** has to be *exported* first in order for any other files to import it. For that, you can use the **`export`** keyword:

Code Listing 35: Y.js

```
export const functionY() {


}
```

Now, any file can import this named **`functionY`** export. If **`X.js`** is on the same directory as **`Y.js`**, you can do the following:

Code Listing 36: X.js

```
import { functionY } from './Y';


// functionY();
```

The { **functionY** } syntax is not destructuring—it's importing a named export. You can also export without names using this syntax:

Code Listing 37: Y.js

```
export default function () {


}
```

When you import this default **Y** export, you can give it any name you want:

Code Listing 38: X.js

```
import function42 from './Y';


// function42();
```

Tip: While default exports have their advantages, named exports play much better with intelligent IDEs that offer autocomplete, discoverability, and other features. It is usually better to use named exports, especially when you're exporting many items in a module.

Map, filter, and reduce

These three array methods replace the need to use **for**/**while** loops in many cases. The value of using them over **for**/**while** loops is that they all return a value. They are expressions, and they can be embedded right into JSX curly brackets.

All of these methods work on an original array and receive a callback function as an argument. They invoke the callback function per item in the original array and do something with that callback's return value. The best way to understand them is through examples.

Here's an example of a **.map** that squares all numbers in an array of numbers:

Code Listing 39: The map method

```
[4, 2, 0].map(e => e * e);


// Result: [16, 4, 0]
```

The **map** method uses the return values of its callback function to construct a new array. The return value for each callback function invocation becomes the new value in the new constructed (mapped) array.

Here's an example of a **.filter** that filters an array of numbers reducing it to the set of even numbers only:

Code Listing 40: The filter method

```
[4, 7, 2, 5, 0, 11].filter(e => e%2 === 0)


// Result: [4, 2, 0]
```

The **filter** method uses the return values of its callback function to determine if the current item should remain in the new constructed (filtered) array. If the callback function returns **true**, the item remains.

Here's an example of **reduce** that will compute the sum of all numbers in an array:

Code Listing 41: The reduce method

```
[16, 4, 0].reduce((acc, curr) => acc + curr, 0);


// Result: 20
```

The **reduce** method uses a slightly different callback function. This one receives two arguments instead of one. Besides the regular current-item element (named **e** in all examples), this one also receives an *accumulator* value (named **acc** in the example). The initial value of **acc** is the second argument of reduce (**0** in the example).

The return value for each callback function invocation becomes the new value for the **acc** variable.

Here's what happens to reduce **[16, 4, 0]** into **20**:

Code Listing 42: The reduce method explanation

```
Initial value of acc = 0


First run: acc = 0, curr = 16
  New acc = 0 + 16 = 16


Second run: acc = 16, curr = 4
  New acc = 16 + 4 = 20


Third run: acc = 20, curr = 0
  New acc = 20 + 0 = 20


Final value of acc = 20
```

Because all of these functions are expressions that return values, we can chain them together:

Code Listing 43: Chaining map, filter, and reduce

```
[4, 7, 2, 5, 0, 11]
  .filter(e => e%2 === 0)
  .map(e => e * e)
  .reduce((acc, curr) => acc + curr, 0);


// Result: 20
```

This chain will take an array of numbers and compute the sum of the even numbers in that array after they are squared. You might think that doing three loops instead of one (which would manually include all the operations) is overkill, but this functional style has many advantages. This is beyond the scope of the book. However, whether you like them or not, expect to see them used often in React applications, especially the **`.map`** method, which we will use later.

Conditional expressions

Because you can only include expressions within the JSX curly brackets, you can't write an **`if`** statement in them. You can, however, use a ternary expression:

Code Listing 44: Using a ternary in JSX

```
<div>
  {condition ? valueX : valueY}
</div>
```

JSX will output either **valueX** or **valueY** based on **condition**. The values can be anything, including other UI elements rendered with JSX:

Code Listing 45: Using elements in ternaries in JSX

```
<div>
  {condition ? <input /> : <img />}
</div>
```

If the result of evaluating an expression inside JSX curly brackets is **true** or **false** (including **undefined** and **null**), React will completely ignore that expression. It will not be cast as strings: "true"/"false"/"null"/"undefined".

This **div** will have no content at all:

Code Listing 46: React ignores true/false in curly brackets

```
<div>
  {3 === 3}
</div>
```

This is intentional. It allows using a shorter syntax to put a value (or element) behind a condition by using the **&&** operator:

Code Listing 47: The short-circuit evaluation

```
<div>
  {condition && <input />}
</div>
```

If **condition** is true, the second operand will be returned. If it's false, React will ignore it. This means it will either render an input element or nothing at all. This JavaScript trick is known as the *short-circuit evaluation*.

Timeouts and intervals

Timeouts and intervals are part of a browser's API. They're not really part of the JavaScript language itself, but they're used with JavaScript functions like **`setTimeout`** and **`setInterval`**.

Both of these functions receive a "callback" function and a "delay" value. **`setTimeout`** will invoke its callback function *once* after its delay value, while **`setInterval`** will *repeatedly* invoke its callback function with its delay value between each invocation.

This code will print the **`Hello Timeout!`** message after three seconds:

Code Listing 48: setTimeout

```
setTimeout(() => {
  console.log('Hello Timeout!');
}, 3000);
```

The first argument is the callback function, and the second is the delay (in milliseconds). The code in the callback function (the bolded part) is the code that will be executed after three seconds.

This code will print the **`Hello Interval!`** message every three seconds, forever:

Code Listing 49: setInterval

```
setInterval(() => {
  console.log('Hello Interval!');
}, 3000);
```

A **`setInterval`** call will usually have an "exit" condition. Both **`setTimeout`** and **`setInterval`** return an **`id`** of the timer object they create, and that **`id`** value can be used to stop them. You can use a **`clearTimeout(id)`** call to stop a timeout object, and **`clearInterval(id)`** to stop an interval object.

This code will print the **`Hello Interval!`** message every three seconds, but only three times:

Code Listing 50: setInterval with an exit condition

```
let count = 0;
const intervalId = setInterval(() => {
  count = count + 1
  console.log('Hello Interval!');
```

```
  if (count === 3) {
    clearInterval(intervalId);
  }
}, 3000);
```

Timers in a React application are usually introduced within a "side effect" Hook function. We'll see an example of that in Chapter 5.

Chapter 4 React's Fundamental Concepts

The first and most important concept you need to learn about React is the concept of the *component*.

React is all about components

In React, we describe user interfaces using components that are reusable, composable, and stateful as well.

You define small components and then put them together to form bigger ones. All components small or big are reusable, even across different projects.

You can think of components as simple functions (in any programming language). We call functions with some input, and they give us some output. We can reuse functions as needed and compose bigger functions from smaller ones.

React components are exactly the same; their input is a set of "props," and their output is a description of a user interface. We can reuse a single component in multiple UIs, and components can contain other components. The basic form of a React component is actually a plain-old JavaScript function.

A React component can also have a private state to hold data that may change over the lifecycle of the component. This private state is an implicit part of the input that drives the component's output, and that's actually what gives React its name!

How did React get its name?

When the state of a React component (which is part of its input) changes, the user interface it represents (its output) changes as well. This change in the description of the UI has to be reflected in the device we are working with. In a browser, we need to update the HTML DOM tree. In a React application, we don't do that manually. React will simply react to the state changes and automatically (and efficiently) update the DOM when needed.

Creating components using functions

In its simplest form, a React component is a JavaScript function:

Code Listing 51: Code available at jscomplete.com/playground/rs2.1

```
function Button (props) {
```

```
  // Returns a DOM/React element here. For example:
  return <button type="submit">{props.label}</button>;
}

// To render a Button element in the browser
ReactDOM.render(<Button label="Save" />, mountNode);
```

Note how I wrote what looks like HTML in the returned output of the **`Button`** function in Code Listing 51. This is neither JavaScript nor HTML, and it is not even React. However, it is so popular that it became the default in React applications. It is called JSX, and it is a JavaScript extension that allows you to write JavaScript function calls in an XML-like syntax (the X in JSX refers to XML).

We've seen the **`ReactDOM.render`** method in Chapter 2; the only difference is that now we're feeding it JSX instead of **`React.createElement`** calls.

Go ahead and try to return any other HTML element inside the **`Button`** function in Code Listing 51, and see how they are all supported (for example, return an **`input`** element or a **`textarea`** element).

JSX is not HTML

JSX is not understood by browsers. If you try to execute the **`Button`** function in a regular browser console, it'll complain about the first character in the JSX part:

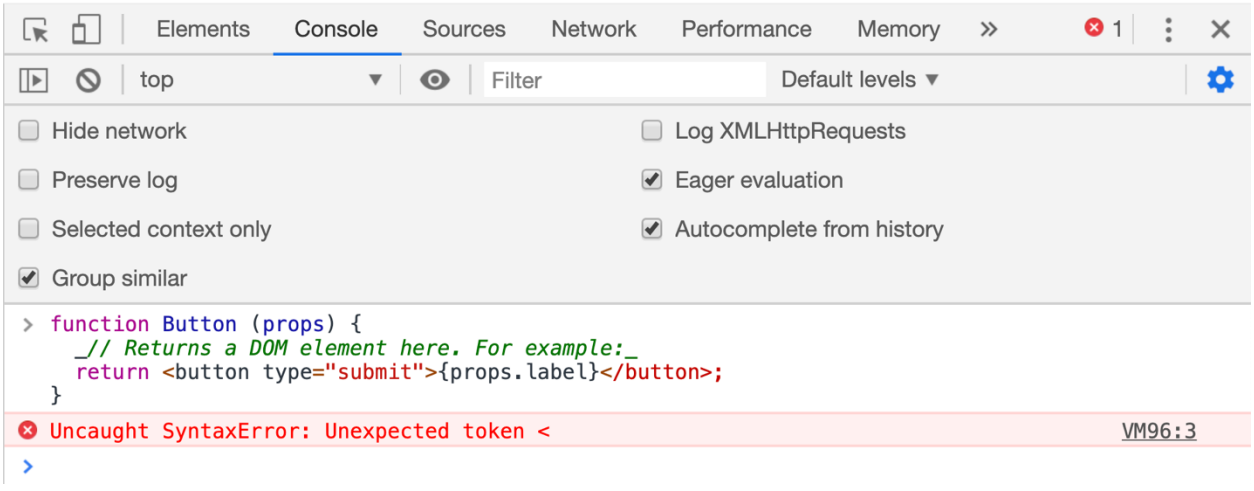


Figure 4: Browsers do not understand JSX

What browsers understand (given the React library is included) is the **`React.createElement`API** calls that we used in Chapter 2. The same **`Button`** example can be written without JSX as follows:

Code Listing 52: Code available at jscomplete.com/playground/rs2.2

```
function Button (props) {
  return React.createElement(
    "button",
    { type: "submit" },
    props.label
  );
}

ReactDOM.render(
  React.createElement(Button, { label: "Save"}),
  mountNode
);
```

You can use React like this—you can execute the **`Button`** function in a browser directly (after loading the React library), and things will work just fine. However, we like to see and work with HTML instead of dealing with function calls. When was the last time you built a website with just JavaScript and not used HTML? You can if you want to, but no one does that. That's why JSX exists.

JSX is basically a compromise. Instead of writing React components using the **`React.createElement`** syntax, we use a syntax very similar to HTML, and then use a compiler to translate it into **`React.createElement`** calls.

A compiler that translates one form of syntax into another is known as a *transpiler*. To translate JSX, we can use transpilers like Babel or TypeScript. For example, the jsComplete playground uses TypeScript to transpile any JSX you put into it. When you use create-react-app, the generated app will internally use Babel to transpile your JSX.

Tip: You can use babeljs.io/repl/ to see what any JSX syntax gets converted to for React, but JSX can also be used on its own. It is not a React-only thing.

So, a React component is a JavaScript function that returns a React element (usually with JSX). When JSX is used, the **`<tag></tag>`** syntax becomes a call to **`React.createElement("tag")`**. It's critically important for you to keep this in mind while building React components. You are not writing HTML— you are using a JavaScript extension to return function calls that create React elements (which are essentially JavaScript objects).

The name has to start with a capital letter

Note how I named the component **`Button`**. The first letter being a capital one is actually a requirement since we will be dealing with a mix of HTML elements and React elements. JSX will consider all names that start with a lowercase letter as names of HTML elements. This is important because HTML elements are passed as strings to **`React.createElement`** calls, while React elements need to be passed as variables:

```
<button></button>
,
<Button></Button>
```

```
"use strict";

React.createElement("button", null),
React.createElement(Button, null);
```

Figure 5: HTML tags vs. component names in JSX

Go ahead and try naming the React component **`button`** instead of **`Button`**, and see how ReactDOM will totally ignore the function and render a regular, empty HTML button.

Code Listing 53: Code available at jscomplete.com/playground/rs2.3

```
// Wrong:
const button = () => (
  <div>My Fancy Button</div>
);


// The following will render an HTML button
// (and ignore the fancy button function)
ReactDOM.render(<button />, mountNode);
```

The first argument is an object of "props"

Just like HTML elements can be assigned attributes like **`id`** or **`title`**, a React element can also receive a list of attributes when it gets rendered. The **`Button`** element in Code Listing 51 received a **`label`** attribute. In React, the list of attributes received by a React element is known as **`props`**. A React function component receives this list as its first argument. The list is passed as an object with keys representing the attribute names, and values representing the values assigned to them.

When using a function component, you don't have to name the object holding the list of attributes as **`props`**, but that is the standard practice. When using class components, which we will do in the following example, the same list of attributes is always presented with a special instance property named **`props`**.

> ***Note: Receiving props is optional. Some components will not have any props. However, a component's return value is not optional. A React component cannot return "undefined" (either explicitly or implicitly)—it has to return a value. It can return "null" to tell the renderer to ignore its output.***

I like to use object destructuring whenever I use component props (or state, really). For example, the **`Button`** component function can be written like this with props destructuring:

Code Listing 54: Destructuring component props

```
const Button = ({ label }) => (
  <button type="submit">{label}</button>
);
```

This approach has many benefits, but the most important one is to visually inspect what props are used in a component and make sure a component does not receive any extra props that are not needed.

> ***Note: Note how I used an arrow function in Code Listing 54 instead of a regular one. This is just a style preference for me personally. Some people prefer the regular function style, and there is nothing wrong with that. What's important is to be consistent with the style that you pick. I'll use arrow functions in this book's examples, but don't interpret that as a requirement.***

Expressions in JSX

In Chapter 3, we saw how you can include a JavaScript expression anywhere within the JSX syntax using a pair of curly brackets.

Code Listing 55: Code available at jscomplete.com/playground/rs2.4

```
const RandomValue = () => (
  <div>
```

```
    { Math.floor(Math.random() * 100) }
  </div>
);

ReactDOM.render(<RandomValue />, mountNode);
```

Only expressions can be included inside these curly brackets. For example, you cannot include a regular **`if`** statement, but a ternary expression is okay. Anything that returns a value is okay. You can always put any code in a function, make it return something, and call that function within the curly brackets. However, keep the logic you put in these curly brackets to a minimum. There are better places for long or complex logic, which we will see in the next chapter.

JavaScript variables are also expressions, so when the component receives a list of props you can use these props inside curly brackets. That's how we used **`{props.label}`** (and **`{label}`**) in the **`Button`** example.

JavaScript object literals are also expressions. Sometimes we use a JavaScript object inside curly brackets, which makes it look like double curly brackets: **`{{a:42}}`**. This is not a different syntax; it is just an object literal defined inside the regular JSX curly brackets.

For example, one use case for using an object literal in these curly brackets is to pass a CSS-style object to the special **`style`** attribute in React:

Code Listing 56: Code available at jscomplete.com/playground/rs2.5

```
const ErrorDisplay = ({ message }) => (
  <div style={ { color:'red', backgroundColor:'yellow' } }>
    {message}
  </div>
);

ReactDOM.render(
  <ErrorDisplay
    message="These aren't the droids you're looking for"
  />,
  mountNode
);
```

The **`style`** attribute is a special one in React. We use an object as its value, and that object defines the styles as if we are setting them through the JavaScript DOM API (camel-case property names and string values). React translates these style objects into inline CSS style attributes. This is generally not the best way to style a React component, but I find it extremely convenient to use when applying conditional styles to elements. For example, here is a component that will randomly output its text in either green or red about half the time:

Code Listing 57: Code available at jscomplete.com/playground/rs2.6

```
class ConditionalStyle extends React.Component {
```

```
  render() {
    return (
      <div style={{ color: Math.random() < 0.5 ? 'green': 'red' }}>
        How do you like this?
      </div>
    );
  }
}

ReactDOM.render(
  <ConditionalStyle />,
  mountNode,
);
```

The logic for this styling is right there in the component. I like that! This is easier to work with than conditionally using a class name and then tracking what that class name is doing in a CSS stylesheet.

JSX is not a template language

Some JavaScript libraries that deal with HTML provide a template language for it. You write your dynamic views with an "enhanced" HTML syntax that has loops and conditionals. These libraries will then use JavaScript to convert the templates into DOM operations. The DOM operations can then be used in the browser to generate the DOM tree described by the enhanced HTML.

React (and ReactDOM) eliminated that step. We do not send the browser a template at all in a React application. We send it a tree of objects described with the React API. React uses these objects to generate the DOM operations needed to display the desired HTML tree.

> ***Note: With an HTML template, the library parses your application as a string. A React application is parsed as a tree of objects.***

While JSX might look like a template language, it really isn't. It's just a JavaScript extension that allows us to represent React's tree of objects with a syntax that looks like an HTML template. Browsers don't have to deal with JSX at all, and React does not have to deal with it either—only the compiler deals with it. What we send to the browser is template-free and JSX-free code.

Take, for example, the **`todos`** array in Code Listing 1. If we're to display that array in a UI using a template language, we'll need to do something like:

Code Listing 58

```
<ul>
 <% FOR each todo in the list of todos %>
  <li><%= todo.body %></li>
 <% END FOR %>
```

```
</ul>
```

Note: The <% %> is one syntax to represent the dynamic enhanced parts. You might also see the {{ }} syntax. Some template languages use special attributes for their enhanced logic, and some template languages make use of whitespace indentation (off-side rule).

When changes happen to the **todos** array (and we need to update what's rendered in the DOM with a template language), we'll have to either re-render that template or compute where in the DOM tree we need to reflect the changes to the **todos** array.

In a React application, there is no template language at all. In Code Listing 2, we described our desired HTML tree for the **todos** array using JSX:

Code Listing 59

```
<ul>
  {todos.map(todo =>
    <li>{todo.body}</li>
  )}
</ul>
```

Which, before being used in the browser, gets translated to:

Code Listing 60

```
React.createElement(
  "ul",
  null,
  todos.map(todo =>
    React.createElement("li", null, todo.body)
  ),
);
```

React takes this tree of objects and makes it into a tree of DOM elements. From our point of view, we're done with this tree. We don't manage any transactions on it. We just manage transactions in the **todos** array itself.

Creating components using classes

React supports creating components through the JavaScript class syntax as well. Here is the same **Button** component example written with the class syntax:

Code Listing 61: Code available at jscomplete.com/playground/rs2.7

```
class Button extends React.Component {
  render() {
    return (
      <button>{this.props.label}</button>
    );
  }
}

// Use it (same syntax)

ReactDOM.render(<Button label="Save" />, mountNode);
```

In this syntax, you define a class that extends **`React.Component`**, which is one of the main classes in the React top-level API. A class-based React component has to at least define an instance method named **`render`**. This **`render`** method returns the element that represents the output of an object instantiated from the component. Every time we use the **`Button`** class-based component (by rendering a **`<Button … />`**), React will instantiate an object from this class-based component and use that object's representation to create a DOM element. It'll also associate the DOM-rendered element with the instance it created from the class.

Note how we used **`this.props.label`** inside the rendered JSX. Every component gets a special instance property named **`props`** that holds all values passed to that component's element when it was instantiated. Unlike function components, the **`render`** function in class-based components does not receive any arguments.

Functions vs. classes

Components created with functions used to be limited in React. The only way to make a component "stateful" was to use the class syntax. This has changed with the release of React Hooks, beginning with React version 16.8, which was released in early 2019. The React Hooks release introduced a new API to make a function component stateful (and give it many other features).

With this new API, most of what is usually done with React can be done with functions. The class-based syntax is only needed for advanced and rare cases.

In this book, I'll use the new Hooks-based API instead of the old class-based one. I believe the new API will slowly replace the old one, but that's not the only reason I want to encourage you to use it (exclusively if you can).

I've used both APIs in large applications, and I can tell you that the new API is far superior to the old one for many reasons, but here are the ones that I personally think are the most important:

- The new API has more constraints. Constraints are good because they basically force you to write less-buggy code.

- You don’t have to work with class "instances" and their implicit state. You work with functions that are refreshed on each render. The state is explicitly declared, and nothing is hidden. All of this basically means that you’ll encounter fewer surprises in your code.
- You can separate any "stateful" logic into self-contained composable and sharable units. This makes it easier to break complex components into smaller parts. It also makes testing components easier.
- You can consume any stateful logic in a declarative way, and without needing to use any hierarchical "nesting" in components trees.

While class-based components will continue to be part of React for the foreseeable future, as a newcomer to the ecosystem, it makes sense for you to start purely with just functions (and Hooks) and focus on learning the new API (unless you have to work with a codebase that already uses classes).

Benefits of components

The term "component" is used by many frameworks and libraries. We can even write web components natively using HTML5 features like custom elements and HTML imports. Components, whether we are working with them natively or through a library like React, have many advantages.

First, components make your code more readable and easier to work with. Consider this UI:

Code Listing 62: HTML-Based UI

```
<a href="http://facebook.com">
 <img src="facebook.png" />
</a>
```

What does this UI represent? If you speak HTML, you can parse it quickly here and say, “it’s a clickable image.” If we’re to convert this UI into a component, we can just name it **`ClickableImage`**.

```
<ClickableImage />
```

When things get more complex, this parsing of HTML becomes harder so components allow us to quickly understand what a UI represents using the language that we’re comfortable with. Here’s a bigger example:

```
<TweetBox>
  <TextAreaWithLimit limit="280" />
  <RemainingCharacters />
  <TweetButton />
</TweetBox>
```

Without looking at the actual HTML code, we know exactly what this UI represents. Furthermore, if we need to modify the output of the **`RemainingCharacters`** section, we know exactly where to go.

React components can also be reused in the same application, and across multiple applications. For example, here's a possible implementation of the **`ClickableImage`** component:

Code Listing 63: ClickableImage Render Function

```
const ClickableImage = ({ href, src }) => {
 return (
   <a href={href}>
     <img src={src} />
   </a>
 );
};
```

Having variables for both the **`href`** and the **`src`** props is what makes this component reusable. For example, to use this component we can render it with a set of props:

```
<ClickableImage href="http://google.com" src="google.png" />
```

And we can reuse it by using a different set of props:

```
<ClickableImage href="http://bing.com" src="bing.png" />
```

Tip: In functional programming, we have the concept of pure functions (which do not have any observable side effects). They are basically protected against any outside state; if we give them the same input, we'll always get the same output. If a React component does not depend on (or modify) anything outside of its definition (for example, if it does not use a global variable), we can label that component pure as well. Pure components have a better chance at being reused without any problems.

We create components to represent views. For **`ReactDOM`**, the React components we define will represent HTML DOM nodes. The **`ClickableImage`** component in Code Listing 63 was composed of two HTML elements.

We can think of HTML elements as built-in components in the browser. We can also use our own custom components to compose bigger ones. For example, let's write a component that displays a list of search engines.

Code Listing 64: SearchEngines Mockup

```
const SearchEngines = () => {
  return (
```

```
    <div className="search-engines">
      <ClickableImage href="http://google.com" src="google.png" />
      <ClickableImage href="http://bing.com" src="bing.png" />
    </div>
  );
};
```

Note how I used the **`ClickableImage`** component to compose the **`SearchEngines`** component!

We can also make the **`SearchEngines`** component reusable as well, by extracting its data into a variable and designing it to work with that variable.

For example, we can introduce a **`data`** array in a format like:

Code Listing 65: SearchEngines Data

```
const data = [
  { href: "http://google.com", src: "google.png" },
  { href: "http://bing.com", src: "bing.png" },
  { href: "http://yahoo.com", src: "yahoo.png" }
];
```

Then, to make **`<SearchEngines data={data} />`** work, we just map the **`data`** array from a list of objects into a list of **`ClickableImage`** components:

Code Listing 66: SearchEngines Render Function

```
const SearchEngines = ({ engines }) => {
  return (
    <List>
      {engines.map(engine => <ClickableImage {...engine} />)}
    </List>
  );
};

ReactDOM.render(
 <SearchEngines engines={data} />,
 document.getElementById("mountNode")
);
```

This **`SearchEngines`** component can now work with any list of search engines we give to it.

What exactly are Hooks?

A Hook in a React component is a call to a special function. All Hook functions begin with the word **use**. The most widely-used Hook function in React is the **useState** one. You can use it to give a component stateful elements.

To see an example of that, let's make the **Button** component (from Code Listing 51) respond to a click event. Let's maintain the number of times it gets clicked in a **count** variable and display the value of that variable as the label of the button it renders.

```
const Button = () => {
  let count = 0;

  return (
    <button>{count}</button>
  );
};

ReactDOM.render(<Button />, mountNode);
```

This **count** variable will be the state element that we need to introduce to the example. It's a piece of data that the UI will depend on (because we're displaying it), and it is a state element because it is going to change over time.

Tip: Every time you define a variable in your code, you will be introducing a state, and every time you change the value of that variable, you are mutating that state. Keep that in mind.

Before we can change the value of the count state, we need to learn about events.

Responding to user events

You can add an event handler with an **onEvent** property (to the **button** element in this case). This could be an **onClick**, **onMouseOver**, **onScroll**, or **onSubmit**, among others.

What we need here is an **onClick** event, and we just define it as an attribute on the target element. For example, to make the program log a message to the console every time the button is clicked, we can do something like:

Code Listing 67: Using onClick in React

```
const Button = () => {
  let count = 0;

  return (
    <button onClick={() => console.log('Button clicked')}>
```

```
      {count}
    </button>
  );
};

ReactDOM.render(<Button />, mountNode);
```

Unlike the DOM version of the **`onClick`** attribute (which uses a string), React's **`onClick`** attribute uses a function reference. You specify that inside curly brackets.

Code Listing 68

```
function func() {}

<button onClick={func} />
```

Note how we passed the **`func`** reference (name) as the **`onClick`** handler. We did not invoke **`func`** in there. React will invoke **`func`** when the button gets clicked.

In Code Listing 67, we inlined a function definition that when invoked will output a message to the console. Each time we click on the button, the **`onClick`** handler (the inline arrow function) will be invoked and we'll see that message.

Note: Note how the event name is in camel case. All DOM-related attributes (which are handled by React) need to be in camel case, and React will display an error if they are not. React also supports using custom HTML attributes, and those have to be in all-lowercase format.

Some DOM attributes in React are slightly different from what they do in the regular DOM API. An example of that is the `onChange` event. In a regular browser, it's usually fired when you click outside a form field (or tab out of it). In React, `onChange` fires whenever the value of a form field is changed (on every character added or removed).

Some attributes in React are named differently from their HTML equivalent. An example of that is the `className` attribute in React, which is equivalent to using the `class` attribute in HTML. You can see a complete list of the differences between React attributes and DOM attributes in the React docs.

Reading and updating state

To track state updates and trigger virtual DOM diffing and real DOM reconciliation, React needs to be aware of any changes that happen to any state elements that are used within components. To do this in an efficient way, React requires the use of special getters and setters for each state element you introduce in a component. This is where the **`useState`** Hook comes into play. It defines a state element and gives us a getter and setter for it.

Here's what we need for the **count** state element we're trying to implement:

```
const [count, setCount] = React.useState(0);
```

The **useState** function returns an array with two items. The first item is a value (getter), and the second item is a function (setter). I used array destructuring to give these items names. You can give them any names you want, but [**name, setName**] is the convention.

The first item "value" can be a string, number, array, or another type. In this case, we needed a number, and we needed to initialize that number with **0**. The argument to **React.useState** is used as the initial value of the state element.

The second item "function" will change the value of the state element (and trigger DOM processing if needed). Each time the **setCount** function is invoked, React will re-render the **Button** component, which will refresh all variables defined in the component (including the **count** value). The argument we pass to **setCount** becomes the new value for **count**.

To make the button increment its label, we need to invoke the **setCount** function within the **onClick** event and pass to it the current **count** value incremented by 1. Here's the full code of the label-incrementing button example:

Code Listing 69

```
const Button = () => {
  const [count, setCount] = useState(0);

  return (
    <button onClick={() => setCount(count + 1)}>
      {count}
    </button>
  );
};

ReactDOM.render(<Button />, mountNode);
```

Go ahead and test that. The button will now increment its label on each click.

Note how we did not implement any actions to change the UI itself. We implemented an action to change a JavaScript object (in memory). Our UI implementation was basically telling React that we want the label of the button to always reflect the value of the **count** object. Our code didn't do any DOM updates—React did.

Note also how I used the **const** keyword to define **count**, although it's a value that gets changed. Our code will not change that value. React will when it uses a fresh call of the **Button** function to render the UI of its new state. In that fresh call, the **useState** function will give us a new fresh **count** value.

Tip: The `useState` function is available globally in the playground. This is just an alias to `React.useState`. In your code, you can use named imports to have `useState` available directly in the scope of a module. In this book's examples, I'll use the `useState` (and other React `use`* functions) directly for brevity. All examples will work in the playground, but remember to name-import these Hook functions when you start using them in a different environment.

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

You'll need a few more examples to appreciate this power. So, let's add some more features to this basic example. Let's make the UI show many buttons, and make them increment a single count value.

Working with multiple components

Let's split the **`Button`** component that we have so far into two components:

- Keep a **`Button`** component to represent a button element, but with a static label.
- Add a new **`Display`** component to display the count's value.

The new **`Display`** component will be a purely presentational one, with no state or interactions of its own. That's normal. Not every React component has to have stateful Hooks or be interactive.

Code Listing 70

```
const Display = (props) => (
  <pre>COUNT VALUE HERE...</pre>
);
```

The responsibility of the **`Display`** component is to simply display a value that it will receive as a prop. For example, the fact that a **`pre`** element was used to host the value is part of that responsibility. Other components in this application have no say about that!

Rendering sibling components

We now have two elements to render: **`Button`** and **`Display`**. We can't render them directly next to each other like this:

```
// This will not work


ReactDOM.render(<Button /><Display />, mountNode);
```

Adjacent elements can't be rendered like this in React because each of them gets translated into a function call when JSX is converted. You have a few options to fix this problem.

First, you can pass an array of elements to **`ReactDOM.render`** and insert into that array as many React elements as you wish.

Code Listing 71: Option #1

```
ReactDOM.render([<Button />, <Display />], mountNode);
```

This is usually a good solution when all the elements you're rendering are coming from a dynamic source. It's not ideal for the case we're doing here.

Another option is to make the sibling React elements the children of another React element. For example, we can just enclose them in a **`div`** element.

Code Listing 72: Option #2

```
ReactDOM.render(
  <div>
    <Button />
    <Display />
  </div>,
  mountNode
);
```

React API supports this nesting. In fact, React has a special object if you need to enclose multiple adjacent elements like this without introducing a new DOM parent node. You can use **`React.Fragment`**:

Code Listing 73: Option #3

```
ReactDOM.render(
  <React.Fragment>
    <Button />
    <Display />
  </React.Fragment>,
  mountNode
);
```

This case is so common in React that the JSX extension has a shortcut for it. Instead of typing **`React.Fragment`**, you can just use an empty tag, **`<>`**.

Code Listing 74: Option #3+

```
ReactDOM.render(
  <>
    <Button />
```

```
    <Display />
  </>,
  mountNode
);
```

The empty tag will get transpiled into the **`React.Fragment`** syntax. I'll use this syntax to continue with the example.

However, you should always try to make the first argument to **`ReactDOM.render`** a single component call instead of the nested tree that we just did. This is essentially a code quality preference. It forces you into thinking about your components' hierarchy, names, and relations. Let's do that next.

The top-level component

Let's introduce a top-level component to host both the **`Button`** and **`Display`** components. The question now is: what should we name this new parent component?

Note: Believe it or not, naming your components and their state/props elements is a very hard task that will affect the way these components work and perform. The right names will force you into the right design decisions. Take some time and think about every new name you introduce in your React apps.

Since this new parent component will host a **`Display`** with a **`Button`** that increments the displayed count, we can think of it as the count value manager. Let's name it **`CountManager`**.

```
const CountManager = () => {
  return (
    <>
      <Button />
      <Display />
    </>
  );
};

ReactDOM.render(<CountManager />, mountNode);
```

Since we're going to display the count's value in the new **`Display`** component, we no longer need to show the count's value as the label of the button. Instead, we can change the label to something like **`+1`**.

Code Listing 75

```
const Button = () => {
  return (
    <button onClick={() => console.log('TODO: Increment counter')}>
```

```
      +1
    </button>
  );
};
```

Note how I've also removed the state element from the **Button** component because we can't have it there anymore. With the new requirement, both the **Button** and **Display** components need access to the **count** state element. The **Display** component will display it, and the **Button** component will update it. When a component needs to access a state element that's owned by its sibling component, one solution is to "lift" that state element one level up and define it inside its parent component, which, in this case, is the **CountManager** component that we just introduced.

By moving the state to **CountManager**, we can now "flow" data from parent to child using component props. That's what we should do to display the count value in the **Display** component:

Code Listing 76

```
const Display = ({ content }) => (
  <pre>{content}</pre>
);

const CountManager = () => {
  const [count, setCount] = useState(0);

  return (
    <>
      <Button />
      <Display content={count} />
    </>
  );
};

ReactDOM.render(<CountManager />, mountNode);
```

Note how in **CountManager** I used the exact same **useState** line that was in the **Button** component. We are lifting the same state element. Note also how when I flowed the **count** value down to the **Display** component via a prop, I used a different name for it (**content**). That's normal. You don't have to use the exact same name. In fact, in some cases, introducing new generic names are better for children components because they make them more reusable. The **Display** component could be reused to display other numeric values besides **count**.

Parent components can also flow down behavior to their children, which is what we need to do next.

Since the **count** state element is now in the **CountManager** component, we need a function on that level to handle updating it. Let's name this function **incrementCounter**. The logic for this function is actually the exact same logic we had before in the **handleClick** function in the **Button** component. The new **incrementCounter** function is going to update the **CountManager** component **count** state to increment the value using the previous value:

Code Listing 77

```
const CountManager = () => {
  // ....

  const incrementCounter = () => setCount(count + 1);

  // ...
}
```

The **onClick** handler in the **Button** component has to change now. We want it to execute the **incrementCounter** function that's in the **CountManager** component, but a component can only access its own functions. So, to make the **Button** component able to invoke its parent's **incrementCounter** function, we can pass a reference to **incrementCounter** to the **Button** component as a prop. Yes, props can hold functions as well, not just data. Functions are just objects in JavaScript, and just like objects, you can pass them around.

We can name this new prop anything. I'll name it **clickAction** and pass it a value of **incrementCounter**, which is the reference to the function we defined in the **CountManager** component. We can use this new passed-down behavior directly as the **onClick** handler value. It will be a prop for the **Button** component:

Code Listing 78

```
const Button = ({ clickAction }) => {
  return (
    <button onClick={clickAction}>
      +1
    </button>
  );
};

// ...


const CountManager = () => {
  // ...


  return (
    <div>
      <Button clickAction={incrementCounter} />
```

```
      <Display content={count} />
    </div>
  );
}
```

Something very powerful is happening here. This **clickAction** property allowed the **Button** component to invoke the **CountManager** component's **incrementCounter** function. It's like when we click that button, the **Button** component reaches out to the **CountManager** component and says, "Hey parent, go ahead and invoke that increment counter behavior now."

In reality, the **CountManager** component is the one in control here, and the **Button** component is just following generic rules. If you analyze the code as it is now, you'll realize how the **Button** component has no clue about what happens when it gets clicked. It just follows the rules defined by the parent and invokes a generic **clickAction**. The parent controls what goes into that generic behavior. That's an example of the concept of responsibility isolation. Each component here has certain responsibilities, and they get to focus on that.

Look at the **Display** component for another example. From its point of view, the count value is not a state; it is just a prop that the **CountManager** component is passing to it. The **Display** component will always display that prop. This is also a separation of responsibilities.

As the designer of these components, you get to choose their level of responsibilities. For example, if we want to, we can make the responsibility of displaying the count value part of the **CountManager** component itself and not use a new **Display** component for that, but I like it this way. The **CountManager** component has the responsibility of managing the count state. That's an important design decision that we made, and it's one you're going to have to make a lot in a React application: where to define the state?

The practice I follow is to define a state element in a shared parent node that's as close as possible to all the children who need to access that state element. For a small application like this one, that usually means the top-level component itself. In bigger applications, a sub-tree might have its own state "branch." In the next chapter, we'll see an example of the value of having some state managed in a sub-tree rather than defining all state elements on the top level.

Tip: The top-level component is usually the one used to manage shared application state and actions because it's a parent to all other components. However, be careful about this design because updating a state element on the top-level component means that the whole tree of components will be re-rendered (in memory), which can affect performance.

Here's the full code for this example so far:

Code Listing 79: Code available at jscomplete.com/playground/rs2.8

```
const Button = ({ clickAction }) => {
  return (
```

```
    <button onClick={clickAction}>
      +1
    </button>
  );
};

const Display = ({ content }) => (
  <pre>{content}</pre>
);

const CountManager = () => {
  const [count, setCount] = useState(0);

  const incrementCounter = () =>
    setCount(count + 1);

  return (
    <div>
      <Button clickAction={incrementCounter} />
      <Display content={count} />
    </div>
  );
}
```

Making components reusable

Components are all about reusability. Let's make the **`Button`** component reusable by changing it so that it can increment the global count with any value, not just 1.

Let's start by adding more **`Button`** elements in the **`CountManager`** component so that we can test this new feature:

Code Listing 80: Adding more Button elements

```
const CountManager = () => {
  // ..

  return (
    <>
      <Button clickAction={this.incrementCounter} /> {/* +1 */}
      <Button clickAction={this.incrementCounter} /> {/* +5 */}
      <Button clickAction={this.incrementCounter} /> {/* +10 */}
      <Display count={this.state.count} />
    </>
  );
};
```

All the **`Button`** elements rendered in Code Listing 80 will have a **`+1`** label, and they will increment the count with 1. We want to make them display different labels that are specific to each button and make them perform a different action based on a value that is specific to each one. Remember that you can pass any value to a React element as a prop.

Here's the UI I have in mind after clicking each button once:

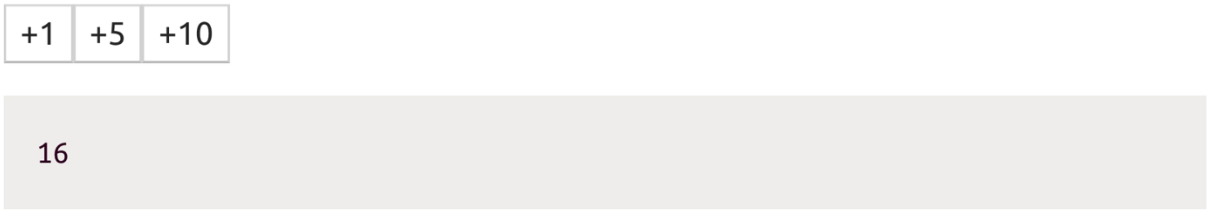


Figure 6: The count value started with 0. We added 1, then 5, and then 10 to get to 16

Tip: Before we go through this exercise, take some time and think about it and try to implement it yourself. It is mostly straightforward. Hint: you'll need to introduce one new prop for `Button`. Give it a shot and come back when you are ready to compare your solution with mine.

Adding new props

The first thing we need do is make the **`+1`** label in the **`Button`** component a custom one.

To make something customizable in a React component, we introduce a new prop (which the parent component can control) and make the component use its value. For our example, we can make the **`Button`** component receive the amount to increment (**`1`**, **`5`**, **`10`**) as a new prop. I'll name this prop **`clickValue`**. We can change the render method in **`CountManager`** to pass the values we want to test to this new **`clickValue`** prop.

Code Listing 81: The clickValue prop

```
return (
  <>
    <Button clickAction={incrementCounter} clickValue={1} />
    <Button clickAction={incrementCounter} clickValue={5} />
    <Button clickAction={incrementCounter} clickValue={10} />
    <Display content={count} />
  </>
);
```

Note a couple of things about this code so far. First, I did not name the new property with anything related to **`count`**. The **`Button`** component does not need to be aware of the meaning of its click event. It just needs to pass this **`clickValue`** along when its click event is triggered. For example, naming this new property **`countValue`** would not be the best choice because we would read the code to understand that a **`Button`** element is related to a count. This makes the **`Button`** component less reusable. For example, if I want to use the same **`Button`** component to append a letter to a string, its code would be confusing.

Also note that I used curly brackets to pass the values of the new **`clickValue`** property (**`clickValue={5}`**). I did not use strings there (**`clickValue="5"`**). Since I have a mathematical operation to do with these values (every time a button is clicked), I need these values to be numbers. If I pass them as strings, I would have to do some string-to-number conversion when the add operation is to be executed.

Customizing behaviors

The other thing we need to make generic in the **`CountManager`** component is the **`incrementCounter`** action function. It cannot have a hardcoded **`count + 1`** operation as it does now. Similar to what we did for the **`Button`** component, to make a function generic we make it receive an argument and use that argument's value. For example:

Code Listing 82

```
incrementCounter = (incrementValue) =>
  setCount(count + incrementValue+);
```

Now all we need to do is make the **`Button`** component use its **`clickValue`** prop as its label, and make it invoke its **`onClick`** action with its **`clickValue`** as an argument.

Code Listing 83

```
const Button = ({ clickValue, clickAction }) => {
  return (
    <button onClick={() => clickAction(clickValue)}>
      +{clickValue}
    </button>
  );
};
```

Note how I had to wrap the **`onClick`** prop with an inline arrow function in order to bind it to the button's **`clickValue`**. The JavaScript closure for this new arrow function will take care of that.

The three buttons should now increment the shared **`count`** state with their three different click values. You can see the code for this example at jscomplete.com/playground/rs2.9.

Accepting input from the user

Imagine we need to count the characters a user types in a text area, just like Twitter's tweet form. With each character the user types, we need to update the UI with the new character count.

Here's a component that displays a **`textarea`** input element with a placeholder **`div`** for the character count:

Code Listing 84: Code available at jscomplete.com/playground/rs2.10

```
const CharacterCounter = () => {
  return (
    <div>
      <textarea cols={80} rows={10} />
      <div>Count: X</div>
    </div>
  );
};

ReactDOM.render(<CharacterCounter />, mountNode);
```

To update the count as the user types in the **`textarea`**, we need to customize the event that fires when the user types. That event in React is **`onChange`**. Note that it is different than the DOM's **`onchange`** event, which is not fired when the user types in a **`textarea`** element. React has a few improvements to the way events are fired.

We'll also need to use a state element for the count of characters and fire its updater function within the **`onChange`** event.

In the new **`onChange`** event handler that we need to come up with, we'll need access to the text that was typed in the **`textarea`** element. We'll need to read it somehow because React by default is not aware of it. As the user types, the rendered UI changes through the browser's own state management. We did not instruct React to change the UI based on the value of the **`textarea`** element.

We can read the value using two main methods. First, we can read it by using the DOM API itself directly. We'll need to "select" the element with a DOM selection API, and once we do that we can read its value using an **`element.value`** call. To select the element, we can simply give it an ID and use the **`document.getElementById`** DOM API to select it.

Because React renders the **`textarea`** element, we can actually do the element selection through React itself. React has a special **`ref`** attribute that we can assign to each DOM element and use to access it later.

We can also access the element through the **`onChange`** event's target object itself. Each event exposes its target, and in the case of an **`onChange`** event on a **`textarea`**, the target is the **`textarea`** element.

That means all we need to do is:

Code Listing 85: Code available at jscomplete.com/playground/rs2.11

```
const CharacterCounter = () => {
  const [count, setCount] = useState(0);

  const handleChange = (event) => {
    const element = event.target;
    setCount(element.value.length);
  };

  return (
    <div>
      <textarea cols={80} rows={10} onChange={handleChange} />
      <div>Count: {count}</div>
    </div>
  );
};
```

This is the simplest solution, and it actually works fine. What's not ideal about this solution is that we're mixing concerns. The **handleChange** event has the side effect of calling the **setCount** function and computing the length of the text. This is really not the concern of an event handler.

The reason we needed to mix these concerns is that React is not aware of what is being typed. It's a DOM change, not a React change.

We can make it a React change by overriding the value of **textarea** and updating it through React as a state change. In the **onChange** handler, instead of counting the characters, we just set the value of what has been typed on the state of the component. Then the concern of what to do with that value becomes part of the React UI render logic. Here's a version of the solution that uses this strategy:

Code Listing 86: Code available at jscomplete.com/playground/rs2.12

```
const CharacterCounter = () => {
  const [inputValue, setInputValue] = useState('');

  const handleChange = (event) => {
    const element = event.target;
    setInputValue(element.value);
  };

  return (
    <div>
      <textarea cols={80} rows={10} value={inputValue}
onChange={handleChange} />
      <div>Count: {inputValue.length}</div>
```

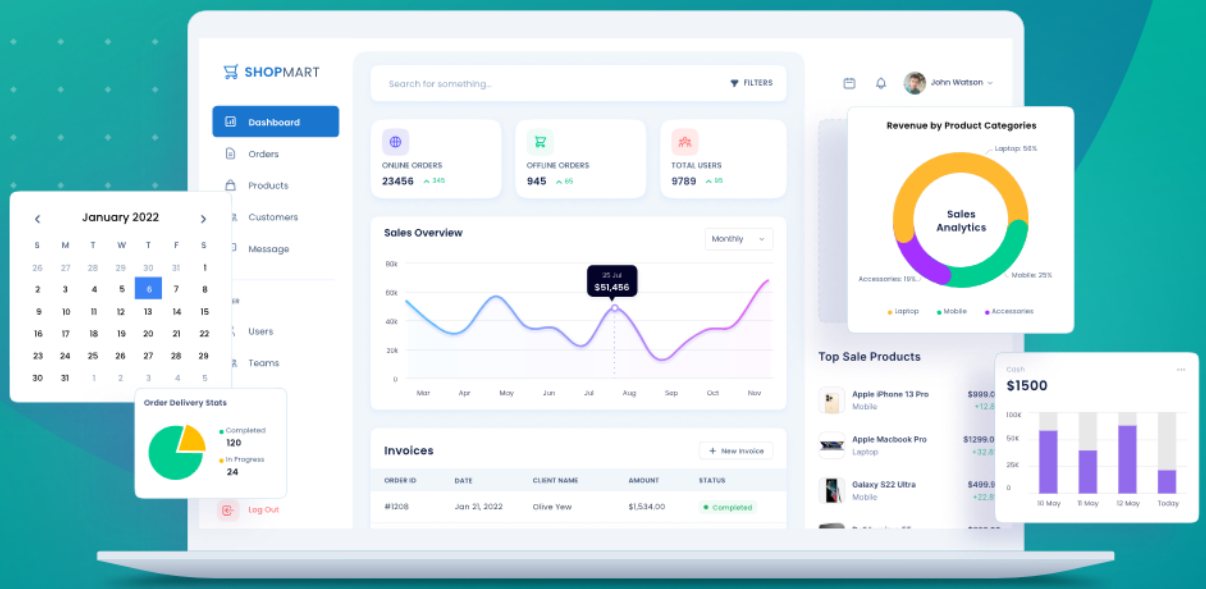
```
    </div>
  );
};
```

Although this is a bit more code, it has clear separation of concerns. React is now aware of the input element state. It controls it. This pattern is known as the *controlled component pattern* in React.

This version is also easier to extend. If we're to compute the number of words as the user types, this becomes another UI computed value. No need to add anything else on the state.

The complete React UI components library for building mobile and web Apps

4.6 out of 5 stars



GET YOUR FREE REACT UI COMPONENTS

syncfusion.com/communitylicense

- 65+ React UI components that are lightweight and user-friendly.
- Responsive and touch friendly on all devices.
- Stunning built-in themes available with UI customization.
- One of the best React components libraries in the market.
- Expect new features and widgets frequently.
- A wide range of product demos, documentation, and video tutorials.

Trusted by the world's leading companies

Syncfusion

Chapter 5 Let's Build a Game

In the previous chapter, we learned a few basic concepts about React components, including how to define them, make them receive props, render them in the browser, make them interactive, and make them reusable. However, we have been using abstract examples so far. Let's build something a little bit more interesting (and fun).

I love memory games, so let's build a simple one in this chapter. I picked a grid-based memory challenge game where the player is challenged to recall the locations of cells on a grid. Here's what the UI will look like when we're done:

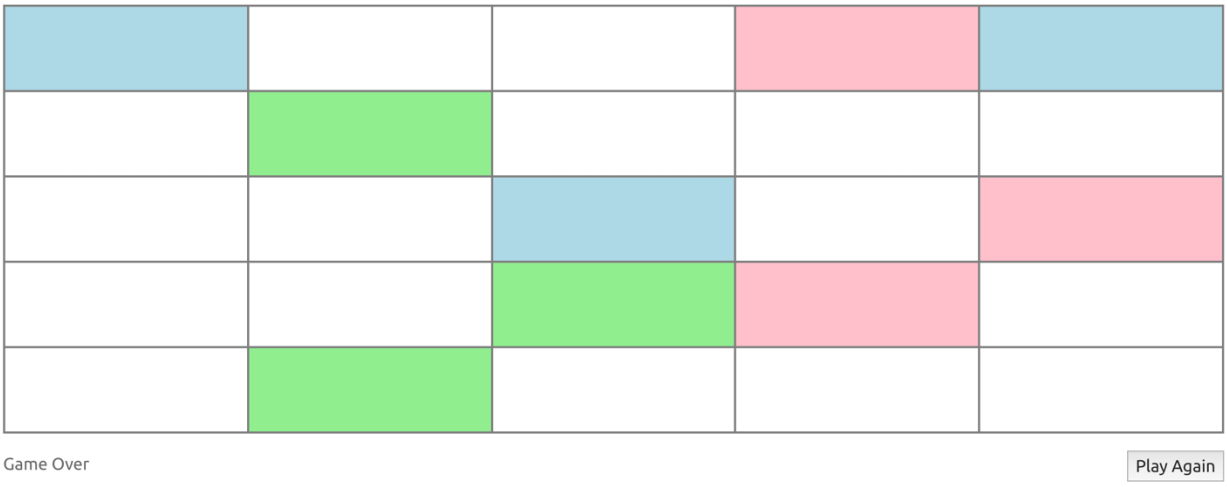


Figure 7: *The memory challenge game*

Here's how this game works:

- The first UI will show an empty grid (no colors) and a **Start Game** button. Clicking that button starts the game. This means the game will have different statuses, and we need to manage them with a React state element (because we need the UI to change).
- When the game starts, a few cells on the grid will be highlighted with a light blue background for three seconds, and then the grid resets back to no colors. The player is then given 10 seconds to recall the locations of the blue cells to win. This will require the use of timers in the code and that'll teach us about side-effect Hooks in React.
- During the 10-second playtime, the player can make up to two wrong picks. The game will be over at three wrong picks, or if the 10-second timer runs out. A wrong pick is shown in the UI with a pink background, while a correct one is shown with a light green background. We'll need to figure out what minimum information we need to place on React's state to satisfy all these UI changes.
- When the game is over (either with a win or lose status), a Play Again button shows up, and that button will reset everything and start a new game. This will teach us how to properly reset a stateful component that has side effects.

Note: Familiarize yourself with the mechanics of playing the game before you proceed. You can play the final version here.

Let's go through this game one step at a time. The key strategy is to find small increments and focus on them, rather than getting overwhelmed with the whole picture. Don't shoot for perfection from the first round. Have something that works and then iterate, improve, and optimize.

"Make it work. Make it right. Make it fast."

—Kent Beck

Initial markup and style

I like to start a React app like this one with some initial markup and any styles that I can come up with for that markup. I do that using a single React component. That's usually a good starting point that'll make it easier to think about the structure of the app components and which elements will need to have a dynamic aspect to them.

The initial template for this game will render a simple empty grid of white cells, a message, and a button. Here's the HTML and CSS that we'll start with. This will render a 3 × 3 grid:

Code Listing 87: The HTML | jscomplete.com/playground/rs3.1

```
const Game = () => {
  return (
    <div className="game">
      <div className="grid">
        <div className="cell" style={{ width: '33.3%' }} />
        <div className="cell" style={{ width: '33.3%' }} />
        <div className="cell" style={{ width: '33.3%' }} />

        <div className="cell" style={{ width: '33.3%' }} />
        <div className="cell" style={{ width: '33.3%' }} />
        <div className="cell" style={{ width: '33.3%' }} />

        <div className="cell" style={{ width: '33.3%' }} />
        <div className="cell" style={{ width: '33.3%' }} />
        <div className="cell" style={{ width: '33.3%' }} />
      </div>
      <div className="message">Game Message Here...</div>
      <div className="button">
        <button>Start Game</button>
      </div>
    </div>
```

```
  );
};

ReactDOM.render(<Game />, mountNode);
```

Code Listing 88: The CSS | jscomplete.com/playground/rs3.1

```
*, *:before, *:after {
  box-sizing: border-box;
}
.game {
  max-width: 600px; margin: 5% auto;
}
.grid {
  border: thin solid gray; line-height: 0;
}
.cell {
  border: thin solid gray; display: inline-block;
  height: 80px; max-height: 15vh;
}
.message {
  float: left; color: #666; margin-top: 1rem;
}
.button {
  float: right; margin-top: 1rem;
}
```

A few things to note about this code:

- The template HTML was rendered with a single, top-level React component (which I named **`Game`**).
- I intentionally kept the list of cells flat to simplify the code. I used the power of styling to display the list of flat **`divs`** as a grid. The alternative is to use a list of lists and convert that into rows and columns, which would have complicated the code a bit. I love how CSS can sometimes greatly simplify UI logic. If something can be adequately done with CSS, you should consider that.
- I used React's special **`style`** prop for the width of each cell. This width value will later be driven based on a dynamic size for the grid, and using React's **`style`** object will make that change easier.

We'll also need a few utility math functions for this game. To keep the focus on React code in this example, I'll provide these utility functions for you here. Their implementation is not really important in the context of this React example, but it's always fun to read their code and try to understand what they do (but let's do that when we start using them).

Code Listing 89: The math science utility functions | jscomplete.com/playground/rs3.1

```
// Math science

const utils = {
  /*

    Create an array based on a numeric size property.

    Example: createArray(5) => [0, 1, 2, 3, 4]

  */

  createArray: size => Array.from({ length: size }, (_, i) => i),

  /*

    Pick random elements from origArray up to sampleSize

    And use them to form a new array.

    Example: sampleArray([9, 12, 4, 7, 5], 3) => [12, 7, 5]

  */

  sampleArray: (origArray, sampleSize) => {
    const copy = origArray.slice(0);
    const sample = [];
    for (let i = 0; i < sampleSize && i < copy.length; i++) {
      const index = Math.floor(Math.random() * copy.length);
      sample.push(copy.splice(index, 1)[0]);
    }
    return sample;
  },

  /*

    Given a srcArray and a crossArray, Count how many elements

    in srcArray exist or do not exist in crossArray.

    Returns an array with the two counts.

    Example: arrayCrossCounts([0, 1, 2, 3, 4], [1, 3, 5]) => [2, 3]

  */

  arrayCrossCounts: (srcArray, crossArray) => {
    let includeCount = 0;
    let excludeCount = 0;
    srcLoop: for (let s = 0; s < srcArray.length; s++) {
      for (let c = 0; c < crossArray.length; c++) {
        if (crossArray[c] === srcArray[s]) {
          includeCount += 1;
```

```
          continue srcLoop;
        }
      }
      excludeCount += 1;
    }
    return [includeCount, excludeCount];
  },
};
```

I'll explain these utility functions as we use them.

Note: The code for all the HTML, CSS, and utility functions is available here.

Extracting components

Once we reach a good state for the initial markup and styles, thinking about components is the natural next step.

There are many reasons to extract a section of the code into a component. Here are a couple in the context of our game:

- We extract components to make part of the tree define data requirements or behavior on its own and take responsibility of it. In this game, all grid cells will be clickable to count toward a guess, and their click events should trigger a UI update. This is a good indicator that we should create a component to represent a single grid cell. I'll name it **`Cell`**.
- We extract components to make the code more readable. A component with too much JSX is harder to reason with and maintain. To demonstrate an example of this, I'll make the footer part (which holds the game message and a button) into a component. I'll name it **`Footer`**.

We extract components for other reasons too. For example, when there is duplication of sub-trees, or when we need to make parts of the full tree reusable in general. This is a simple game with no shared sub-trees, so we don't need to extract any components for these reasons.

There is another very important reason to extract a component, and that's to split complex logic into smaller, related units. This is usually done once the code crosses the boundaries of the level of complexity that's acceptable to you. At that point, it'll be easier for you to decide what related units should be extracted on their own. We'll see an example of that soon.

Extracting a component is simply taking part of the tree as is, making it the return value of the new component (for example, **`Cell`**), and then using the new component in the exact place where the extracted part used to be. To use the new **`Cell`** component, you would include a call to **`<Cell />`**.

Here's the code we started with after I extracted **`Cell`** and **`Footer`** out of **`Game`**:

Code Listing 90: Code available at jscomplete.com/playground/rs3.2

```
const Cell = () => {
  return (
    <div className="cell" style={{ width: '33.3%' }} />
  );
}

const Footer = () => {
  return (
    <>
      <div className="message">Game Message Here...</div>
      <div className="button">
        <button>Start Game</button>
      </div>
    </>
  );
};

const Game = () => {
  return (
    <div className="game">
      <div className="grid">
        <Cell />
        <Cell />
        <Cell />

        <Cell />
        <Cell />
        <Cell />

        <Cell />
        <Cell />
        <Cell />
      </div>
      <Footer />
    </div>
  );
};
```

The represented DOM tree did not change—only its arrangement in React's memory did. Note how I needed to use a **`React.Fragment`** (**`<></>`**) around the two **`div`**s in **`Footer`** because they're now abstracted into a new sub-tree represented with a single React element. We didn't need that before because they were part of the main tree.

In the new **`Cell`** component, we can now define the generic shared logic for both displaying a grid cell and for the behavior of each grid cell in this game.

You might want to extract more components, maybe a **`Row`** or **`GameMessage`** component. While adding components like these might enhance the readability of the code, I am going to keep the example simple and start with just these three components.

Tip: You might feel that this game is a bit challenging to think about all at once. It is! It has many states and many interactions, but don't let that distract you. What you need is to focus your thinking into finding your next simple "increment." Find the tiniest possible change that you can validate, just like "extracting part of the initial markup into a component." That was a good, small increment that we just validated. We didn't worry about anything else in the game while focusing on that extracting increment. Now you need to think about the next increment and focus on just that. The smaller the increments you choose, the better.

An increment could be anything. There is no right answer. You could focus on defining a click handler on `Cell` and make sure it fires correctly, or you could think about the first three-second timer that'll start ticking when the** Start Game **button is clicked. However, some increments will depend on others. Try to identify the increments that you're ready for. For example, a great next increment for us at this point is to get rid of the manually repeated `Cell` lines and drive that through a dynamic value.

Note: The playground session for the code so far can be found here.

Making the grid dynamic

Imagine needing to pass a prop value to each **`Cell`** element. Right now, we would need to do that in nine places. It'll be a lot better if we generated this list of grid cells with a dynamic loop. This will enhance the readability and maintainability of the code.

In general, while you're making progress in your code, you should keep an eye on values that can be made dynamic. For example, unless the requirement is to have a fixed grid that is always 3 **×** 3, we should introduce a **`gridSize`** property somewhere and use it to generate a **`gridSize * gridSize`** grid of **`Cell`** components. Don't hard-code the number nine (or 16 or 25) into the loop code. Hard-coded values in your code are usually a "smell" (bad code). Even if you don't need things to be dynamic, just slapping a label on a hard-coded number makes the code a bit more readable. This is especially important if you need to use the same number (or string, really) in many places.

Let's test things out with a **`gridSize`** of **`5`**. Instead of defining this as a variable in the **`Game`** component, I'll make it into a "prop" that the top-level component receives. This just an option that's available to you when you need it, but in our case it delivers an extra value! Besides not having the number **`5`** hard-coded (within the loop code), we can also render the **`Game`** component with a different grid size just by changing its props.

Code Listing 91

```
// In the ReactDOM.render call


<Game gridSize={5} />
```

Tip: Note again how I used `{5}` and not `"5"` because the `Game` component should receive this prop as a number value, not as a string value.

Can you think of more primitive values that we should not hard-code in the code? How about the number of cells to highlight as challenge cells? The maximum wrong attempts allowed? The number of seconds to highlight the challenge cells? The number of seconds that'll cause the game to time out? All of these should really be made into variables in the code. I'll make them all as props to the **`Game`** component.

Code Listing 92

```
// In the ReactDOM.render call


<Game
  gridSize={5}
  challengeSize={6}
  challengeSeconds={3}
  playSeconds={10}
  maxWrongAttempts={3}
/>
```

We can use a simple **`for`** loop to generate a list of **`Cell`** components based on the new **`gridSize`** prop and give every cell a unique ID based on the loop index. However, the **declarative** way to do this is to create an array of unique IDs first, and then map that into an array of **`Cell`** components.

The value of doing so is really beyond just being declarative. Having an array of all cell IDs will simplify any calculations that we need. For example, picking a list of random challenge cells becomes a matter of sampling this generated array.

We need an array with a length of **`gridSize * gridSize`**. There are many ways to do this in JavaScript, but my favorite is to use the **`Array.from`** method. This method can be used to initialize a generated array using a callback function. In the provided **`utils`** object, the **`createArray`** function uses this method to generate an array of any size and have its elements initialized as the index value for their positions.

We can use this **`utils.createArray`** function to create the grid cell IDs array. I will name that array **`cellIds`**. This is a fixed array for each game, which means we don't need to make it part of the game state:

Code Listing 93

```
const Game = ({ gridSize }) => {
  const cellIds = utils.createArray(gridSize * gridSize);

  // ...

};
```

Note: I'll be using the comment // … to mean "omitted code."

Note how I destructured the **`gridSize`** prop and used it. We'll need to destructure the other props as well, but we'll do that when we need them.

With this array, we can now use the **`Array.map`** method to convert the array of numbers into an array of **`Cell`** elements.

Code Listing 94

```
const Game = ({ gridSize }) => {
  // ...

      <div className="grid">
        {cellIds.map(cellId =>
          <Cell
            key={cellId}
          />
        )}
      </div>

  // ...
};
```

Note: I used the values in the `cellIds` array as keys for `Cell`. This not the same as using the loop index as the key. These values are unique IDs in this context. We're also not re-ordering or deleting any of these cells, so these sequential numbers are sufficient for their elements' identities.

With all the changes up to this point, the app will render a grid of **`5 × 5`**, but we are still using a fixed width of **`33.3%`** for each cell. We need to compute the width we need for each grid size: **`100/gridSize`**. We can pass that value as a prop to the **`Cell`** component.

We could do something like:

Code Listing 95

```
<Cell
  key={cellId}
  width={100/gridSize}
/>
```

However, this new **`width`** property has a small problem. Think about it. How can we do this in a better way?

The **`100/gridSize`** computation, while fast, is still a fixed value for each rendered game. Instead of doing it 25 times (for a **`gridSize`** of **`5`**), we can do it only once before rendering all the **`Cell`** elements and just pass the fixed value to all of them:

Code Listing 96

```
const Game = ({ gridSize }) => {
  // ..

  const cellWidth = 100 / gridSize;

  // ...

      <div className="grid">
        {cellIds.map(cellId =>
          <Cell
            key={cellId}
            width={cellWidth}
          />
        )}

  // ...
};
```

Tip: I don't classify the `cellWidth` change here as a premature optimization. I classify it as an obvious one. It will probably not improve the performance of the code by much, but it is the type of change that we can make with 100 percent confidence that it's better. I also think it makes the code more readable. One rule of thumb to follow here is to try and extract any computation done for a value passed to a prop into its own variable or function call.

We can now use the new **`width`** prop in the **`Cell`** component instead of the previously hard-coded **`33.3%`**:

Code Listing 97

```
const Cell = ({ width }) => {
  return (
    <div className="cell" style={{ width: `${width}%` }} />
  );
};
```

All the changes we made so far are part of the core concept in React that the UI is a function of data and state. However, our UI functions so far have been using only fixed-value data elements (like **`gridSize`**). We have not placed anything on the state yet; we will do that shortly. We will also add more fixed-value data elements. Sometimes, it's challenging to figure out if a data element you're considering should be a state one or a fixed-value one. Let's talk about that next.

Note: The playground session for the code so far can be found here.

What's a good next increment now? Let's think about what we need to place on the state of this game.

Designing data and state elements

When the user clicks the **Start Game** button, the game UI will move into a different status. A number of random cells will need to be marked as challenge cells. The UI will need to be updated to display these challenge cells differently. The game "status" is something that all three components need. The **`Cell`** component needs it as part of the logic to determine what color to use. The **`Footer`** component needs it to determine what message and button to show. The **`Game`** component needs it to determine what timers to start and stop, among a few other things.

This means we need to use a state element to represent this game status in the **`Game`** component itself, because it's the common parent to the other two components. I'll name this state element **`gameStatus`**, and it will have five different values. Let's give each one a label:

- A game starts with a **`gameStatus`** of **`NEW`**, which is before the **Start Game** button is clicked.
- When the **Start Game** button is clicked, the **`gameStatus`** becomes **`CHALLENGE`**. This is when we highlight the blue challenge cells.
- After a delay of **`challengeSeconds`**, the **`gameStatus`** becomes **`PLAYING`**. This is when the player can actually recall the challenge cells (which should not be highlighted at that point).
- If the player wins the game, the **`gameStatus`** becomes **`WON`**. If the player loses the game, the **`gameStatus`** becomes **`LOST`**.

We can place this **`gameStatus`** variable on the **`Game`** component's state with a **`useState`** call. The initial value is **`NEW`**:

Code Listing 98

```
const Game = ({ gridSize }) => {
  const [gameStatus, setGameStatus] = useState('NEW');

  // ...

}
```

When the user clicks the **Start Game** button, we can use **`setGameStatus`** to change the **`gameStatus`** variable to **`CHALLENGE`**. However, since we moved the button into a new **`Footer`** component, and its behavior needs to change a state element on the **`Game`** component, we need to create a function in the **`Game`** component for this purpose and flow it down to the **`Footer`** component. I'll just use an inline function:

Code Listing 99

```
const Game = ({ gridSize }) => {
  // ...


      <Footer
        startGame={() => setGameStatus('CHALLENGE')}
      />

  // ...

};


const Footer = ({ startGame }) => {
  // ...


      <div
        className="button"
        onClick={startGame}
      >
        <button>Start Game</button>
      </div>

  // ...

};
```

At this point, we'll need to highlight the challenge cells. However, before we do that, let's think a little bit more about the **`gameStatus`** values we used.

Using ENUMs

Using strings as labels creates a small problem. If you make a typo in these strings somewhere in the code, that typo will go undetected until you realize that things are not working (in the UI probably) and try to find that typo manually. Many developers can tell countless stories of the time they wasted on a silly typo.

A better way of doing this is to introduce an ENUM holding the specific values for **`gameStatus`**. This way, if you use an invalid ENUM value, the problem will be a lot easier to find (especially if your editor uses a type checker like TypeScript).

Unfortunately, JavaScript does not have an ENUM data structure, but we can use a simple object to represent one:

Code Listing 100

```
const GameStatus = {
  NEW: 'NEW',
  CHALLENGE: 'CHALLENGE',
  PLAYING: 'PLAYING',
  WON: 'WON',
  LOST: 'LOST',
};

// ...


const Game = ({ gridSize }) => {
  const [gameStatus, setGameStatus] = useState(GameStatus.NEW);

  // ...


      <Footer
        startGame={() => setGameStatus(GameStatus.CHALLENGE)}
      />


  // ...

};
```

The values can be anything. You can use numbers or symbols, but simple strings there are okay too.

Similarly, we should introduce an ENUM for the different cell statuses. Let's give each a label as well:

- A cell starts with a status of **`NORMAL`**. This is when it appears white.
- If the cell is a challenge cell and the game is in the **`CHALLENGE`** phase, the cell will have a status of **`HIGHLIGHT`**. This is when it's highlighted in blue.

- When the player chooses a cell, its status will become either **`CORRECT`** or **`WRONG`** based on whether it's a challenge cell or not. A **`CORRECT`** cell will be highlighted with a light-green color, and a **`WRONG`** one with pink.

Since these cell statuses directly drive the color values, I'll make their ENUM value the colors themselves:

Code Listing 101

```
const CellStatus = {
  NORMAL: 'white',
  HIGHLIGHT: 'lightblue',
  CORRECT: 'lightgreen',
  WRONG: 'pink',
};
```

This way, in the **`Cell`** component, given that we compute the cell status correctly, we can just use its value as the **`backgroundColor`** value:

Code Listing 102

```
const Cell = ({ width }) => {
  let cellStatus = CellStatus.NORMAL;

  // Compute the cell status here...


  // ...


    <div
      className="cell"
      style={{ width: `${width}%`, backgroundColor: cellStatus }}
    />

  // ...

};
```

The last ENUM we need for this game is for the game messages. Should these messages be placed on the state? Not really. These messages will be directly derived from the **`gameStatus`** value as well. We can actually include them in the **`GameStatus`** ENUM itself, but I'll just give them their own ENUM for simplicity.

For each **`gameStatus`** value, let's give the player a helpful hint or let them know if they won or lost:

Code Listing 103

```
const Messages = {
```

```
  NEW: 'You will have a few seconds to memorize the blue random cells',
  CHALLENGE: 'Remember these blue cells now',
  PLAYING: 'Which cells were blue?',
  WON: 'Victory!',
  LOST: 'Game Over',
};
```

Now the content of the message **div** can simply be: **Messages[gameStatus]**.

Because we moved the message **div** into the **Footer** component, we'll need to flow the **gameStatus** variable from **Game** into **Footer** to be able to look up the game message in **Footer**. We could alternatively look up the game message in the **Game** component and flow that to the **Footer** component, but I think it's cleaner to have that logic in the **Footer** component. Besides, the rendering of the **Footer** button will also depend on **gameStatus**.

Code Listing 104

```
const Footer = ({ gameStatus, startGame }) => {
  // ...


      <div className="message">{Messages[gameStatus]}</div>


  // ..

};

const Game = ({ gridSize }) => {
  // ..


      <Footer
        gameStatus={gameStatus}
        startGame={() => setGameStatus(GameStatus.CHALLENGE)}
      />

  // ...

};
```

Note: Note that if the `Messages` ENUM needs to be fetched from an API (for example, to use a different language), then we need to place it on the state or delay the render process until we have it.

Identifying computed values

Let's now think about how to highlight the challenge cells. When the game starts, we need to randomly pick **`challengeSize`** elements from the grid array that has all of the cells' IDs (**`cellIds`**). I'll put these in a **`challengeCellIds`** array.

To compute this array, we can use the provided **`utils`** function **`sampleArray`**, which takes an **`origArray`** and a **`sampleSize`**, and returns a new array with **`sampleSize`** elements randomly picked from **`origArray`**.

The important question now is: should we place this **`challengeCellIds`** array on the state of the **`Game`** component? To answer this question, think about whether or not this **`challengeCellIds`** array will change during a game session.

It will not—it is another a fixed value that does not need to be part of the game state. Deciding on the optimal elements that need to be placed on a component state is one of the most important skills of a React developer.

The **`challengeCellIds`** array can be a variable in the **`Game`** component:

Code Listing 105

```
const Game = ({ gridSize, challengeSize }) => {
  const [gameStatus, setGameStatus] = useState(GameStatus.NEW);

  const cellIds = utils.createArray(gridSize * gridSize);
  const cellWidth = 100 / gridSize;
  const challengeCellIds = utils.sampleArray(cellIds, challengeSize);


  // ...

};
```

Tip: Always place the state element first, and then introduce any "computed" ones after that.

The **`Game`** component is now aware of which cells it should highlight when the **`gameStatus`** becomes **`CHALLENGE`**.

However, since **challengeCellIds** (and **cellIds**) are computed from props and do not depend on the state of the **Game** component, co-locating them with the stateful elements in the **Game** component is a problem.

Note: Interview question: Explain the problem with co-locating the `challengeCellIds` array with the stateful elements in the `Game` component.

To see the problem I am talking about, log the value of **challengeCellIds** after the line that defines it:

Code Listing 106

```
const Game = ({ gridSize, challengeSize }) => {
  // ...

  const challengeCellIds = utils.sampleArray(cellIds, challengeSize);
  console.log(challengeCellIds);

  // ...
```

Try to render the **Game** component a few times. On each render, we get a different set of **challengeCellIds**. That's great—the **sampleArray** function is working as expected.

Now, render a game session and click **Start Game**. What's your observation?

The **challengeCellIds** array was regenerated when we clicked that button.

Figure 8: This should not happen

That's not good: this **challengeCellIds** should be a fixed value during a single game session. Once it's generated, it should not change. It should only be re-generated when we need a new game session (with the Play Again button).

Why is this happening? Didn't we define **challengeCellIds** using the **const** keyword? How is it changing when we click the button?

Wrapping stateful components

Keep in mind that each time you invoke a **`useState`** updater function, which is what we're doing in the **`onClick`** handler of the button using the **`setGameStatus`** updater function, React will re-render the whole component that asked for that state. In this case, it is the **`Game`** component.

Re-rendering means React will call the **`Game`** function again, discard all variables that were defined in the previous render, and create new variables for the new render. It'll create a new **`challengeCellIds`** variable (and new **`cellIds`** and **`cellWidth`** variables).

These three variables should not be re-created. They should be "immutable" once a game "instance" is rendered. One way of doing that is to make them into global variables. However, that means when we're ready to implement the Play Again button, all new games will use the exact same challenge cells. That's not good either.

We want a game session's fixed values (**`cellIds`**, **`challengeCellIds`**, and **`cellWidth`**) to be global to that session, but not to the whole app. This means we can't have the **`Game`** component as a top-level one. We'll need a "game generator" component that renders (and later re-renders) a game session. That new component can be responsible for the app-global variables that are needed for each top-level render.

Tip: We can also implement the `challengeCellIds` using a `React.useRef` Hook. This Hook gives us an element that'll remember its value between component renders. However, wrapping the stateful component with another one is a much simpler way.

The problem starts with the fact that I am mixing many responsibilities in the one component that I named **`Game`**. Naming matters here—a lot! What exactly did I mean by **`Game`**? Is it referring to the game in general, the app that renders many game sessions, or the single playable game session?

The solution starts with better names. This app should have a **`GameSession`** component and a **`GameGenerator`** component that's responsible for generating and rendering a game session. When the **`GameGenerator`** component is rendered, it can compute the global values needed to run a **`GameSession`**. The **`GameSession`** component needs to manage its state independently of the state of the **`GameGenerator`** component.

Go ahead and try that on your own first. Rename **`Game`** to **`GameSession`** and keep the **`console.log`** line in there for testing. Create a **`GameGenerator`** component and move the computing of all three fixed values into it. Then pass them as props to the **`GameSession`** component. Test that the **`challengeCellIds`** array does not change when you click the **Start Game** button.

Here are the changes we need to make:

Code Listing 107

```
const GameSession = ({
  cellIds,
```

```
  challengeCellIds,
  cellWidth,
  challengeSize,
  challengeSeconds,
  playSeconds,
  maxWrongAttempts,
}) => {
  // ...

  // Remove lines for cellIds, cellWidth, and challengeCellIds

  console.log(challengeCellIds);

  // ...

};

const GameGenerator = () => {
  const gridSize = 5;
  const challengeSize = 6;
  const cellIds = utils.createArray(gridSize * gridSize);
  const cellWidth = 100 / gridSize;
  const challengeCellIds = utils.sampleArray(cellIds, challengeSize);

  return (
    <GameSession
      cellIds={cellIds}
      challengeCellIds={challengeCellIds}
      cellWidth={cellWidth}
      challengeSize={challengeSize}
      challengeSeconds={3}
      playSeconds={10}
      maxWrongAttempts={3}
    />
  );
};
ReactDOM.render(<GameGenerator />, mountNode);
```

Note a few things about these changes:

- I moved the fixed values of **gridSize** and **challengeSize** into the **GameGenerator** component (instead of having them as props for **GameSession**). This was just to simplify the new code, but it'll also be a first step for you to implement some of the bonus challenges at the end of the chapter.
- The three variables we moved out of the **GameSession** component and into the **GameGenerator** component are now passed to **GameSession** as props using the same names. Nothing else needs to change in the **GameSession** component's render logic.

- The **GameGenerator** component has no state of its own (yet). You don't need to place the state of your app on the top-level component. In this particular case, keeping the state in the **GameSession** component (which is now a sub-tree) allowed us to "cache" the app-global variables in the parent tree.
- Since the **gridSize** is no longer needed in the **GameSession** component, we don't need to keep it as a prop on it. All the other props I kept are needed; we're just not using them all yet. For example, the **challengeSize** is needed to determine a **GameStatus.WON** state, and the **maxWrongAttempts** is needed to determine a **GameStatus.LOST** state. The "seconds" variables are needed to manage the timers. I destructured all these props in the **GameSession** component.

Now when you click the **Start Game** button, the **challengeCellIds** array will be exactly the same because it's just a prop that was not changed. React re-renders the **GameSession** component with its exact same props when its state changes.

```
Console was cleared
▶ (6) [3, 24, 15, 20, 0, 23]
▶ (6) [3, 24, 15, 20, 0, 23]
>
```

Figure 9: Re-rendering Game is now okay

Note: The playground session for the code so far can be found here.

Determining what to make stateful

To be able to compute a cell status, we need to design the structure to identify a cell as a correct or wrong pick. Should we have **correctCellIds** and **wrongCellIds** arrays managed on the state of the **GameSession** component?

We could, but a good practice here is to minimize the elements you make stateful in a component. If something can be computed, don't place it on the state. Take a moment and think about what structure can be the minimum here to enable us to compute a correct or wrong pick.

These statuses will appear when the player starts clicking on cells to pick them. With each pick, something has to change on the state to make React re-render the chosen cell and enable us to recompute its status, which will be either **CellStatus.CORRECT** or **CellStatus.WRONG**.

However, we don't need to manage a structure for correct and wrong picks because both of them can be computed from **challengeCellIds**. Just knowing that a cell was picked allows us to determine if that pick was correct or wrong. All we need to place on the state is the fact that the cell was picked!

Let's use an array to keep track of **pickedCellIds** on the state of the **GameSession** component. This array starts as empty:

Code Listing 108

```
const GameSession = ({ gridSize, challengeSize }) => {
  const [gameStatus, setGameStatus] = useState(GameStatus.NEW);
  const [pickedCellIds, setPickedCellIds] = useState([]);

  // ...

};
```

We will later design the click behavior on each cell so that it adds the clicked cell's ID to this **`pickedCellIds`** array.

What about winning or losing the game? Do we need to place anything else on the state to make the UI render differently for these states? The answer is no.

Both of these cases are also computable. With a **`challengeSize`** of **`6`** and a **`maxWrongAttempts`** of **`3`**, if we have six correctly guessed cells after every guess, the game is won. If we have three wrong attempts, the game is lost. We actually do not even need to change the **`gameStatus`** to **`WON`** or **`LOST`** because these two statuses are computable. However, computing these statuses requires some array math, and placing them on the state is a form of simply caching this computation.

Tip: Ideally, caching the won/lost `gameStatus` computation is better done with a React `ref` object (using the `React.useRef` Hook). However, that's a bit of an advanced optimization. I will store these values on the `gameStatus` state for simplicity and consistency. Having part of `gameStatus` on the state and the other part on a `ref` object might be confusing.

What about the game timers? Do we need to place anything on the state for them? The answer depends on what we want to display in the UI. If all we need is a timer ticking in the background that drives nothing in the UI to be changed (on every tick), then we do not need to place anything on the state.

However, if we'd like to show a "countdown" value in the UI as the timer is ticking, then that timer needs to update a state element. Let's do that for the **`playSeconds`** timer. Let's show the countdown from **`playSeconds`** to 0 in the UI. We'll need a state element to hold these countdown values. I'll name it **`countdown`**. Its initial value is the **`playSeconds`** prop:

Code Listing 109

```
const GameSession = ({
  cellIds,
  challengeCellIds,
  cellWidth,
  challengeSize,
  challengeSeconds,
  playSeconds,
  maxWrongAttempts,
```

```
}) => {
  const [gameStatus, setGameStatus] = useState(GameStatus.NEW);
  const [pickedCellIds, setPickedCellIds] = useState([]);
  const [countdown, setCountdown] = useState(playSeconds);

  // ...

};
```

I think we have all the data and states we need for a game session. This is the easy part though—the next few steps are where the real power of React comes in handy. We can complete the design of our UIs as functions of data and state, and then only worry about changing the state afterwards, eliminating the need to do any manual DOM operations.

Note: The playground session for the code so far can be found here.

Completing the UI as a function of state

Now that we have identified and designed all the static and dynamic data elements this game needs, we can continue the task of implementing the UI as a function of these elements. Everything rendered in the UI is a direct map of a certain snapshot of the data and state elements we designed.

This step requires implementing a few computed properties. For example, we need to compute each **`cell`** status based on the **`gameStatus`** value, the **`challengeCellIds`** value, and the **`pickedCellIds`** value. Each **`cell`** status depends on all three variables.

Practically adopting the minimum props concept

To keep the code simple, I'll compute each **`cell`** status in the **`Cell`** component itself. Does this mean we need to flow **`gameStatus`**, **`challengeCellIds`**, and **`pickedCellIds`** from the **`GameSession`** component to the **`Cell`** component?

Code Listing 110

```
// In the GameSession component


        {cellIds.map(id => (
          <Cell
            key={id}
            width={cellWidth}
            gameStatus={gameStatus}
            challengeCellIds={challengeCellIds}
            pickedCellIds={pickedCellIds}
```

```
          />
        ))}
```

No. Don't do that.

The **`Cell`** component does not need to know about all **`challengeCellIds`** or all **`pickedCellIds`**. It only needs to know whether it is a challenge cell or a picked cell. What we need are flags like **`isPicked`** and **`isChallenge`**, and we can use a simple **`Array.includes`** to compute them:

Code Listing 111

```
// In the GameSession component


        {cellIds.map(id => (
          <Cell
            key={id}
            width={cellWidth}
            gameStatus={gameStatus}
            isChallenge={challengeCellIds.includes(id)}
            isPicked={pickedCellIds.includes(id)}
          />
        ))}
```

I'll refer to this pattern as *props minimization*. It is mostly a readability pattern, but it is also tied to the responsibility and maintainability of each component. For example, if we later decided to use a different data structure to hold **`challengeCellIds`**, ideally the code inside the **`Cell`** component should not be affected.

Another way to think about this pattern is to ask yourself this question about each prop you pass to a component: Does this component need to be re-rendered when the value of this prop changes?

For example, think about the **`pickedCellIds`** prop we first considered passing to each **`Cell`**. This array will be changed on each click of a cell. Since we're re-rendering a gird of cells (25 cells, for example), the props-minimization question becomes: do all 25 grid cells need to be re-rendered every time we click on a cell?

The answer is no, only one cell needs to be re-rendered: the one cell that will change its **`isPicked`** value from **`false`** to **`true`**.

With every prop you pass to a child component comes great responsibility!

Tip: React will re-render all components in a sub-tree when the parent component re-renders. So, although we minimized the information by using `isPicked` instead of `pickedCellIds`, React will still re-render all 25 cells on each click (and each timer tick later on). However, the props minimization allows us to optimize the frequency

with which that component re-renders itself by "memoizing" it. We can wrap a component with a `React.memo` call to memoize it. Memoizing comes with a small penalty because it requires comparing the props each time a component re-renders, but that penalty is usually less of an issue than actually re-rendering a component that didn't need to be re-rendered, especially when there are some significant computations in these components.

What about **`gameStatus`**? Is that a minimum prop? Not really. When that value changes from **`NEW`** to **`CHALLENGE`**, we only need to re-render a few cells (to highlight them); we don't need to re-render all cells.

We could make this better by not passing the **`gameStatus`** to **`Cell`**, and instead passing something like **`shouldBeHighlighted`**. However, I think it's cleaner to just pass the **`gameStatus`** itself down to the **`Cell`** component. It's really a core primitive value in each cell computation, and using it directly will make the code more readable. Besides, if we go the other way, more computational logic will need to exist in the **`GameSession`** component. I think all the logic about computing the status of a cell should be inside the **`Cell`** component. This is just my preference.

The practice I can get behind here is to try to only flow the minimum props that a component needs to re-render itself, but don't sacrifice readability and responsibility management to be 100 percent exact about that.

I think that's a good starting point. Let's now compute the **`cellStatus`** value based on these three values.

Computing values before rendering

We have the three variables that are directly responsible for the value of each cell status. We can come up with a rough description of the conditions around these variables that are responsible for what status a cell should have. It often helps to start with the pseudocode natural description of the computation we're about to make. Here's what I came up with based on the three points we analyzed when we came up with the **`cellStatus`** ENUM.

Code Listing 112

```
The cell-status starts as NORMAL

If the game-status is NEW:
  Do nothing. All cells should be normal

if the game-status is not NEW:

  if the cell is picked
    The cell-status should be either CORRECT or WRONG
    based on whether it is a challenge cell or not

  if the cell is not picked
    If it's a challenge cell and the game-status is CHALLENGE:
```

```
      The cell-status is HIGHLIGHT
```

Technically, the "if the cell is picked" condition should also account for what game status we want the correct or wrong cell statuses to appear for. Do we want the green/pink highlights to stay when the game is won or lost? I think so. It is nice to have the players see the original challenge cells that they failed to guess. This is why I did not include a condition about game status in the is-picked branch. The only other game status value there is **`CHALLENGE`**, and we know for a fact that when the game has that status, no cell has been picked yet.

Here's one way to translate the pseudocode in the **`Cell`** component:

Code Listing 113

```
const Cell = ({ width, gameStatus, isChallenge, isPicked }) => {
  let cellStatus = CellStatus.NORMAL;
  if (gameStatus !== GameStatus.NEW) {
    if (isPicked) {
      cellStatus = isChallenge ? CellStatus.CORRECT : CellStatus.WRONG;
    } else if (
      isChallenge &&
      (gameStatus === GameStatus.CHALLENGE || gameStatus ===
GameStatus.LOST)
    ) {
      cellStatus = CellStatus.HIGHLIGHT;
    }
  }
  return (
    <div
      className="cell"
      style={{ width: ${width}%, backgroundColor: cellStatus }}
    />
  );
};
```

Note how I added the **`LOST gameStatus`** case to show the cell as highlighted. There is no need to add the **`WON`** status there because all cells will be correctly picked in that case.

We can partially test this change by clicking the **Start Game** button. That click changes **`gameStatus`** to **`CHALLENGE`**, and the UI will show the six random blue cells. Render the UI a few times and click the **Start Game** button to verify that the random blue cells are different on each game render.

How can we test the rest of this logic? We have not implemented the "pick-cell" UI behavior yet. We don't have a way to win or lose the game yet. We can use mock state values.

Using mock state values

The state usually starts with empty values (like the empty **`pickedCellIds`** array). It is hard to design the UI without testing using actual values. Mock values come to the rescue.

For example, we can start the **`pickedCellIds`** with some mock values and start the **`gameStatus`** with a value of **`GameStatus.PLAYING`**:

Code Listing 114: Using mock state values

```
// In the Game component


const [gameStatus, setGameStatus] = useState(GameStatus.PLAYING);
const [pickedCellIds, setPickedCellIds] = useState([0, 1, 2, 22, 23,
24]);
```

With these temporary changes in the initial values, and based on the randomly assigned challenge cells, the grid should show the first and last three cells as either pink or green. Print the **`challengeCellIds`** array to make sure the right cells are displayed green or pink:

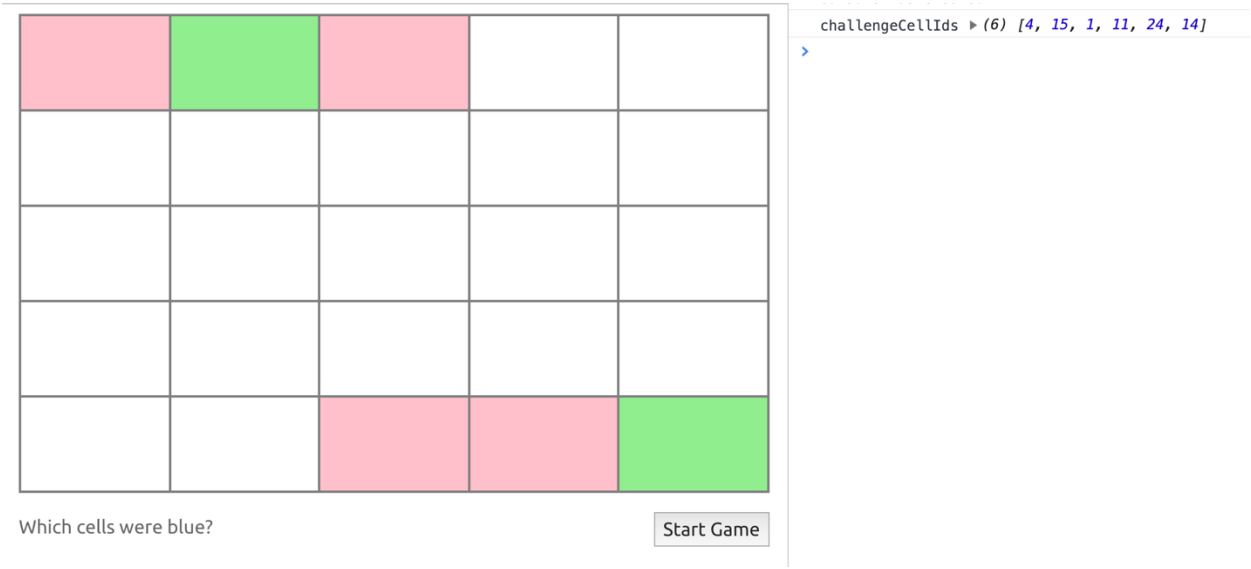


Figure 19: Testing with mock initial values in the state

Note how cells **#1** and **#24** were green because they happened to be part of both the **`pickedCellIds`** and **`challengeCellIds`** when I took the screenshot. The other picked cells were pink because they were wrong picks for this mock state.

> ***Tip: Change the `gameStatus` mock value to `WON/LOST` to verify that all the logic we have for `cellStatus` is working as expected.***

Using this strategy, we do not have to worry about behavior and user interactions (yet). We can focus on just having the UI designed as functions of data and (mock) state.

For example, another thing that needs to change in the UI based on the different states is the appearance of the button in the **`Footer`** component. The **Start Game** button should only show up when **`gameStatus`** is **`NEW`**. When the player wins or loses the game, a Play Again button should show up instead. While the player is playing the game, let's just show the **`countdown`** value in the button area. A few **`if`** statements in the **`Footer`** component will do the trick:

Code Listing 115: The UI for the button area

```
const Footer = ({ gameStatus, countdown, startGame }) => {
  const buttonAreaContent = () => {
    if (gameStatus === GameStatus.NEW) {
      return <button onClick={startGame}>Start Game</button>;
    }
    if (
      gameStatus === GameStatus.CHALLENGE ||
      gameStatus === GameStatus.PLAYING
    ) {
      return countdown;
    }
    return <button onClick={() => {/* TODO */}}>Play Again</button>;
  };
  return (
    <>
      <div className="message">{Messages[gameStatus]}</div>
      <div className="button">{buttonAreaContent()}</div>
    </>
  );
};

// Then in GameSession, pass the newly-used countdown value


      <Footer
        gameStatus={gameStatus}
        countdown={countdown}
        startGame={() => setGameStatus(GameStatus.CHALLENGE)}
      />
```

Note a few things about what I did:

- I used a function to compute the content of the button area. This approach has a few advantages over just inlining the code in the component function itself (like what we did for **`cellStatus`** in the **`Cell`** component). This is really a style preference, but the use of early returns and the isolation of that logic into a single unit (that, for example, is testable on its own) are among the reasons I like this approach better. You could also extract that part itself into its own component. A component is just a function, after all.
- I rendered the **Play Again** button, but we have not implemented its behavior yet. I left a TODO comment in there instead. If you need to inline comments in JSX, that's how you do it.

- I omitted the conditions to render the Play Again button because they're all that's remaining (after the first two checks that cover three statuses). However, this means if another **`GameStatus`** value is introduced, this code will stop behaving right for all the cases. Shall we fix that?

I think compromises like these are okay for prototyping, but you should get in the habit of trying to cover all the cases and think a tiny bit forward about the extendability of your code. I think I'd like the **`buttonAreaContent`** function better with a **`switch`** statement that does that:

Code Listing 116

```
const Footer = ({ gameStatus, countdown, startGame }) => {
  const buttonAreaContent = () => {
    switch(gameStatus) {
      case GameStatus.NEW:
        return <button onClick={startGame}>Start Game</button>;
      case GameStatus.CHALLENGE:
        // fall-through
      case GameStatus.PLAYING:
        return countdown;
      case GameStatus.WON:
        // fall-through
      case GameStatus.LOST:
        return <button onClick={() => {/* TODO */}}>Play Again</button>;
    }
  };
  // ...

};
```

You can test this new UI logic by just changing the initial value of **`gameStatus`** in the **`GameSession`** component.

Now all we need to do is implement the user interactions and have them change the state—then we'll be done! React will always reflect the current values in data/state in the UI using the functions we already defined.

Note: The playground session for the code so far can be found here.

Implementing behaviors to change the state

Using a timeout side effect

We implemented the first step of the **Start Game** button already: it changes the **`gameStatus`** value into **`CHALLENGE`**. What we need now is to change that value into **`PLAYING`** after three seconds (or whatever **`challengeSeconds`** value we end up using).

This is where we can use a timer object. The source of the behaviors that change the state is not always directly from the user (click or other events). Sometimes the behavior comes from a timer in the background.

We can start this timer in the same function that's invoked when the **Start Game** button is clicked, the one where we inlined the call to **`setGameStatus`**. However, React has a different place for this logic. We can use a side-effect Hook (through the **`React.useEffect`** function).

Starting the timer is a side effect for changing the **`gameStatus`** value to **`CHALLENGE`**. Every time the **`gameStatus`** value changes to **`CHALLENGE`**, we need to start that timer. Every time the **`gameStatus`** is no longer **`CHALLENGE`**, we need to clear that timer if it's still running.

The **`React.useEffect`** Hook function is designed to implement that exact logic. It takes a callback function and a list of dependencies:

Code Listing 117: The useEffect Hook

```
useEffect(() => {
  // do something when dep1 or dep2 changes
}, [dep1, dep2])
```

During any rendering of the component, if the values of the dependencies are different from the last time **`useEffect`** was invoked, it'll invoke its callback function. We can start the timer inside a side-effect Hook if we can determine that the **`gameStatus`** has changed to **`CHALLENGE`**. To do that, we can make **`gameStatus`** a dependency for our timer side-effect Hook.

Furthermore, if the component is re-rendered or removed from the DOM, a side effect can invoke a cleanup function. That cleanup function can be returned from the **`useEffect`** callback function. We should clean any timers we start in a side effect's callback function in that side effect's cleanup function.

Here's what we need to change the **`gameStatus`** into **`PLAYING`** after three seconds using a side-effect Hook:

Code Listing 118: Using a timer in a side-effect Hook

```
// In the GameSession component


  useEffect(() => {
    let timerId;
    if (gameStatus === GameStatus.CHALLENGE) {
      timerId = setTimeout(
        () => setGameStatus(GameStatus.PLAYING),
        1000 * challengeSeconds
      );
    }
    return () => clearTimeout(timerId);
  }, [gameStatus, challengeSeconds]);
```

Because **gameStatus** is a dependency, this **useEffect** Hook will invoke its callback function every time that value changes. When it changes into **CHALLENGE**, the **if** statement will become true and we'll start a timer. In the cleanup function, we clear the timer.

For the sake of completeness, I also included the **challengeSeconds** as a dependency, although that value will not change during a single game. You should always include all the variables a side-effect Hook is using in its list of dependencies. You can even automate that in your editor using React ESLint tools like the eslint-plugin-react-hooks package.

Now the game will go into **PLAYING** mode three seconds after clicking the **Start Game** button, and the highlighting of challenge cells should go away. Test that.

When **gameStatus** changes into **PLAYING**, we need to start another timer. This time we need a timer that ticks 10 times repeatedly and modifies the **countdown** state element. This is where we can use a **setInterval** call. We'll need to put this new logic in a side-effect Hook as well.

Because it'll also depend on **gameStatus**, we can use the same Hook for both timers. They just have different branches that we can do through different **if** statements. Don't forget that we need an exit condition for this new branch; we can check the **countdown** value to determine if the interval is to be stopped:

Here's a way to do that:

Code Listing 119: Using an interval timer in a side-effect Hook

```
useEffect(() => {
  let timerId;

  // ...


  if (gameStatus === GameStatus.PLAYING) {
    timerId = setInterval(() => {
      if (countdown === 1) {
        clearTimeout(timerId);
        setGameStatus(GameStatus.LOST);
      }
      setCountdown(countdown - 1);
    }, 1000);
  }
  return () => clearTimeout(timerId);
}, [gameStatus, challengeSeconds]);
```

This will not work—the interval timer will continue to run. Test it and try to figure out why.

Because the code depended on the value of **countdown** and we have not included it as a dependency, the callback function will have a "stale" closure when the **countdown** value changes. That's why you should always include all the dependencies. Adding **countdown** to the list of dependencies will solve the problem, but it's not ideal.

By adding the **countdown** as a dependency, the Hook callback function (and its cleanup one) will be invoked every time the **countdown** value changes. That means every second, we will clear the interval timer and start a new one. We don't even need an interval; we can replace **setInterval** with **setTimeout** and things will still work because of the recursive nature of repeated updates. A **setTimeout** changes the **countdown** value, React re-renders, the Hook's callback function is invoked again, a **setTimeout** changes the **countdown** value, and so on.

The question is: How can we use a **setInterval** that updates the state then?

You'll need the state-update logic to not depend on the current value of the state. We needed the **countdown** dependency in the first place because we're decrementing the current value of that state element. However, all React state updater functions support their own callback form. You can pass a function to any updater function (like **setCountdown**), and that function will be called with the current value of the state as its argument. Whatever that function returns becomes the new value for that state element.

For this example, we just need to move the logic that depends on **countdown** into the **setCountdown** callback function, and we can keep the **setInterval** use that way:

Code Listing 120: Using a callback function in a state updater function

```
useEffect(() => {
  let timerId;

  // ...


  if (gameStatus === GameStatus.PLAYING) {
    timerId = setInterval(() => {
      setCountdown(countdown => {
        if (countdown === 1) {
          clearTimeout(timerId);
          setGameStatus(GameStatus.LOST);
        }
        return countdown - 1;
      });
    }, 1000);
  }
  return () => clearTimeout(timerId);
}, [challengeSeconds, gameStatus]);
```

This version does not depend on the value of **countdown** at all. React will make the current value of **countdown** available to **setCountdown** when it invokes it. This will work fine.

This callback version of React state updater functions is much better because it allows you to optimize any code that updates the state. I'd go as far as saying you should always use the callback version if your state-update logic depends on the current value of the state (like the decrement logic we had).

Tip: Whenever you create a timeout or interval timers in a React component, do not forget to clear them when they're no longer needed (for example, when the component gets removed from the DOM). You can use the `useEffect` cleanup function or the `componentWillUnmount` lifecycle method in class components.

Note: The playground session for the code so far can be found here.

Implementing computations that depend on a state update

The core user interaction in this game, and the one that'll be responsible for the remaining computations we need to make, is the pick-cell behavior.

When a player clicks a cell two things are affected:

- We will always need to change the **`pickedCellIds`** array and add the clicked cell to it.
- We might need to change the **`gameStatus`** value (when **`maxWrongAttempts`** is reached).

Because these states are managed in the **`GameSession`** component, we need to place the pick-cell behavior on that level.

Let's start by defining an empty **`pickCell`** function in **`GameSession`** and pass that function to **`Cell`** so that we can make the **`Cell`** component click handler. We'll wire it to receive the ID of the cell being clicked:

Code Listing 121

```
// In the GameSession component:


  const pickCell = cellId => {
  };

  // ...


  <Cell
    key={cellId}
    width={cellWidth}
    gameStatus={gameStatus}
    isChallenge={challengeCellIds.includes(cellId)}
    isPicked={pickedCellIds.includes(cellId)}
    onClick={() => pickCell(cellId)}
    />

// Then in the Cell component:
```

```
const Cell = ({
  width,
  gameStatus,
  isChallenge,
  isPicked,
  onClick,
}) => {

  // ...

  return (
    <div
      className="cell"
      style={{ width: ${width}%, backgroundColor: cellStatus }}
      onClick={onClick}
    />
  );
  }, [challengeSeconds, gameStatus]);
```

The first change this new function is going to make is straightforward. We can use array destructuring to append the new clicked **`cellId`** to the **`pickedCellIds`** array:

Code Listing 122

```
// In the GameSession component:

  const pickCell = cellId => {
    if (gameStatus === GameStatus.PLAYING) {
      setPickedCellIds(pickedCellIds => {
        if (pickedCellIds.includes(cellId)) {
          return pickedCellIds;
        }
        return [...pickedCellIds, cellId];
      });
    }
  };
```

Note a few things about this code:

- I made it check the **`gameStatus`** first. A player can only pick a cell when the game mode is **`PLAYING`**.
- I used the callback version of the **`setPickedCellIds`** function, although I did not need to. Because I am using the current value of **`pickedCellIds`** (to append to it), it's always a good practice to use the callback version rather than relying on the **`pickedCellIds`** value exposed through function closures.

- I added a check to make sure we do not pick a cell twice. This is important for the logic to determine if a game is **`WON`** or **`LOST`**.

With this code, you can now partially play the game. You can pick cells and see them change to green or pink, but you cannot win or lose the game through picking (the game will be over after 10 seconds through the timer code).

Where can we implement the win/lose game status logic? We can put it in the **`pickCell`** function itself! After all, we need to check if the game is **`WON`** or **`LOST`** after each pick. There is a problem though: this logic will depend on the new value for **`pickedCellIds`** after each pick. We need to do it after the **`setPickedCellIds`** call. All React state updater functions are asynchronous. We can't do anything after them directly.

We could work around this problem by using the new **`pickedCellIds`** array before we use it with **`setPickedCellIds`**. Here's a version of the code to do that:

Code Listing 123: Using the "state to be" in computations

```
// In the GameSession component:

  const pickCell = cellId => {
    if (gameStatus === GameStatus.PLAYING) {
      setPickedCellIds(pickedCellIds => {
        if (pickedCellIds.includes(cellId)) {
          return pickedCellIds;
        }
        const newPickedCellIds = [...pickedCellIds, cellId];
        const [correctPicks, wrongPicks] = utils.arrayCrossCounts(
          newPickedCellIds,
          challengeCellIds
        );
        if (correctPicks === challengeSize) {
          setGameStatus(GameStatus.WON);
        }
        if (wrongPicks === maxWrongAttempts) {
          setGameStatus(GameStatus.LOST);
        }
        return newPickedCellIds;
      });
    }
  };
```

The math logic to compute the new **`gameStatus`** values is not really important, but I am basically using the **`utils.arrayCrossCounts`** array to count how many **`correctPicks`** and **`wrongPicks`** the player has made so far (including the current pick), and then compare these numbers with **`challengeSize`**/**`maxWrongAttempts`** to determine the new **`gameStatus`** value.

By inserting this logic right before the return value, I get to use the value of "the state to be." Go ahead and try to play now—you can win and lose the game.

What's wrong with this code though? Nothing, right? Let's ship it!

Using side effects to separate concerns

While the code in Code Listing 123 works just fine, putting the logic to update **`gameStatus`** in **`pickCell`** itself makes the function less readable and harder to reason with. The **`pickCell`** function should just pick a cell!

Where are we supposed to put the win/lose logic then?

The fact that we need to invoke the win/lose logic is a side effect to updating the **`pickedCellIds`** array. Every time we update the **`pickedCellIds`** array, we need to invoke this side effect. Although it is really "internal" to React and has nothing to do with anything external to it, we can still use a side-effect Hook just to separate these two different concerns.

Revert **`pickCell`** to its first version and move the win/lose logic to its own new **`useEffect`** Hook:

Code Listing 124: Using a side-effect Hook to compute a new state

```
// In the GameSession component:


  React.useEffect(() => {
    const [correctPicks, wrongPicks] = utils.arrayCrossCounts(
      pickedCellIds,
      challengeCellIds
    );
    if (correctPicks === challengeSize) {
      setGameStatus(GameStatus.WON);
    }
    if (wrongPicks === maxWrongAttempts) {
      setGameStatus(GameStatus.LOST);
    }
  }, [pickedCellIds]);
```

This is a bit cleaner. There is no mix of concerns or confusing new versus old picked cells concepts. Each time the **`pickedCellIds`** array is different, the **`useEffect`** callback function will be invoked because we wired **`pickedCellIds`** as a dependency for it. There is no need to do any cleanup for this side-effect Hook because it isn't really a full side effect that depends on something external to React. It is only used to manage internal concerns.

Note: The playground session for the code so far can be found here.

Resetting a React component

The final behavior we need to implement for this game is the **Play Again** button. To reset a game session, we can simply re-initialize the state elements with their first initial values. For example, we could have a **`resetGame`** function like this:

Code Listing 125: Resetting a component state

```
// In the GameSession component:


  const resetGame = () => {
    setGameStatus(GameStatus.NEW);
    setPickedCellIds([]);
    setCountdown(playSeconds);
  };


  // ..


  <Footer
    gameStatus={gameStatus}
    countdown={countdown}
    startGame={() => setGameStatus(GameStatus.CHALLENGE)}
    resetGame={resetGame}
  />


// In the Footer component:


const Footer = ({ gameStatus, countdown, startGame, resetGame }) => {
  const buttonAreaContent = () => {
    switch(gameStatus) {
      // ...
      case GameStatus.WON:
        // fall-through
      case GameStatus.LOST:
        return <button onClick={resetGame}>Play Again</button>;
    }
  };
  // ...

};
```

This method will partially work. It'll even reset the timers because they are cleared in the **`useEffect`** cleanup function that gets invoked before the new game is rendered. However, this method has two problems. Can you identify them?

The first problem is a small one. This method introduces a bit of a code duplication problem. If we're to add more elements to the state of this component (for example, to display the number of current wrong attempts), then we'll have to remember to reset this new state element in **`resetGame`**. This is probably not a big deal for a small app like this one.

The second problem is a bigger one. I saved this solution here for you to test it. Play a few times and try to identify the problem.

Between game resets, the challenge cells remain the same. We're only resetting the state of the **`GameSession`** component. We're not resetting anything in the **`GameGenerator`** component which is responsible for the static data elements that the **`GameSession`** component uses. In addition to resetting the state of the **`GameSession`** component, we need the **`GameGenerator`** to re-render itself, compute a new set of random challenge cells, and render a fresh version of the **`GameSession`** component.

Changing a component's identity

React offers a trick to combine both actions needed to reset a game session. We can re-render the **`GameGenerator`** component (by changing its state) and use a different identity value for the **`GameSession`** component through the special "key" prop. That'll make React render a brand new **`GameSession`** instance (using initial state values).

When we use the **`key`** attribute for a component, React uses it as the unique identity of that component. Say we rendered a component with **`key="X"`**, and then later re-rendered the exact same component (same props and state) but with **`key="Y"`**. React will assume that these two are different elements. It'll completely remove the **`X`** component from the DOM (which is known as "unmounting"), and then it'll mount a new version of it as **`Y`**, although nothing else has changed in **`X`** besides its key.

This can be used to reset any stateful component. We just give each game instance a key and change that key to reset it. We'll need to do that in the **`GameGenerator`** component and since we need that component to re-render, we can use a stateful element for the value of this new key prop.

To make this change, remove the **`resetGame`** function from the **`GameSession`** component and instead make it receive a prop named **`resetGame`**. The **`GameGenerator`** component will now be in charge of resetting a game:

Code Listing 126

```
const Game = ({
  cellIds,
  challengeCellIds,
  cellWidth,
  challengeSize,
```

```
  challengeSeconds,
  playSeconds,
  maxWrongAttempts,
  autoStart,
  resetGame,
}) => {

  // Remove the resetGame function


};
```

Next, make the following changes to the **GameGenerator** component:

- Define a **gameId** state variable and pass it as the key for React to identify a **GameSession** instance.
- Pass **resetGame** to the **GameSession** component as a function that'll change the **gameId** state variable.

Code Listing 127: Changing a component identity

```
const GameGenerator = () => {
  const [gameId, setGameId] = useState(1);

  // ..


  return (
    <GameSession
      key={gameId}
      cellIds={cellIds}
      challengeCellIds={challengeCellIds}
      cellWidth={cellWidth}
      challengeSize={challengeSize}
      challengeSeconds={3}
      playSeconds={10}
      maxWrongAttempts={3}
      resetGame={() => setGameId(gameId => gameId + 1)}
    />
  );
};
```

That's it—now the game will reset properly. All computed data elements in **GameGenerator** will be re-computed with the **gameId** state change. All props passed to the **GameSession** component will have a new value. The state of the **GameSession** component will be re-initialized because it's now a brand-new element in the main tree (because of the new **key** value).

It would be nice, however, to auto-start the second game session (and all sessions after that) instead of having the player click the **Start Game** button again after clicking **Play Again**.

Try to do that on your own first.

Controlling state initial value with a prop

The **GameGenerator** component is the one that knows if a game was the first one, or if it was generated through a **resetGame** action. In fact, that condition is simply **gameId > 1**. However, the **GameSession** component is the one responsible for starting the game (through its **gameStatus** value). One way to solve this issue is by passing an **autoStart** prop to **GameSession** and using it to initialize the **gameStatus** variable as either **NEW** or **CHALLENGE** right away:

Code Listing 128: Controlling state initial value with a prop

```
const Game = ({
  cellIds,
  challengeCellIds,
  cellWidth,
  challengeSize,
  challengeSeconds,
  playSeconds,
  maxWrongAttempts,
  autoStart,
  resetGame,
}) => {
  const [gameStatus, setGameStatus] = useState(
    autoStart ? GameStatus.CHALLENGE : GameStatus.NEW
  );

  // ...

};

const GameGenerator = () => {
  const [gameId, setGameId] = useState(1);

  // ..


  return (
    <Game
      key={gameId}
      cellIds={cellIds}
      challengeCellIds={challengeCellIds}
      cellWidth={cellWidth}
      challengeSize={challengeSize}
      challengeSeconds={3}
      playSeconds={10}
      maxWrongAttempts={3}
      autoStart={gameId > 1}
```

```
      resetGame={() => setGameId(gameId => gameId + 1)}
    />
  );
};
```

The "Play Again" action will now render a brand-new game, and it will directly be in its **`CHALLENGE`** mode.

This game is now feature-complete. However, before we conclude, there is one more trick that I'd like you to be aware of. One of the best things about React Hooks is that we can extract related logic into a custom function. Let's do that.

Note: The playground session for the code so far can be found here.

Using custom Hooks

The logic to have a **`gameId`** that starts with **`1`**, an **`autoStart`** game flag, and a **`resetGame`** function that increments **`gameId`** is all related. Right now, it's within the **`GameGenerator`** component mixed with the other logic about **`cellIds`** and **`challengeCellIds`**.

If we decide later to change the values of **`gameId`** to be string-based instead of an incrementing number, we'll have to modify the **`GameGenerator`** component. Ideally, this logic should be extracted into its own unit, and the **`GameGenerator`** can just use that unit. This is exactly like extracting a piece of code into a function and invoking that function, but this particular piece of code is stateful. The **`useState`** call Hooks into React internals to associate a component with a state element.

Luckily, React supports extracting stateful code into a function. You can extract any Hook calls including **`useState`** and **`useEffect`** into a function (which is known as a custom Hook) and React will continue to associate the Hook calls with the component that invoked the custom Hook function.

Name your custom Hook function **`useXYZ`**. This way, linting tools can treat it as yet another React Hook and provide helpful hints and warnings about it. This is not a regular function—it's a function that contains stateful logic for a component.

I'll name this new custom Hook function **`useGameId`**. You can make it receive any arguments you want, and make it return any data in any shape or form. It does not have to be similar to **`useState`** (or other React Hooks), and we don't need to pass any argument to it in this case. I'll also make it return the three elements related to the game ID concept: the ID itself, a way to tell if this is the first ID, and a way to generate a new ID:

Code Listing 129: Using a custom Hook function

```
const useGameId = () => {
  const [gameId, setGameId] = useState(1);
```

```
  return {
    gameId,
    isNewGame: gameId === 1,
    renewGame: () => setGameId(gameId => gameId + 1),
  };
};

const GameGenerator = () => {
  const { gameId, isNewGame, renewGame } = useGameId();

  // ...


  return (
    <Game
      key={gameId}
      cellIds={cellIds}
      challengeCellIds={challengeCellIds}
      cellWidth={cellWidth}
      challengeSize={challengeSize}
      challengeSeconds={3}
      playSeconds={10}
      maxWrongAttempts={3}
      autoStart={!isNewGame}
      resetGame={renewGame}
    />
  );
};
```

Note how I opted to use different names for elements of the custom Hook. This custom Hook can be reusable across components. Different game apps like this one can use the same **`useGameId`** custom Hook function to provide identity for their components, offer a way to do something for a new or subsequent game, and provide a method to renew the **`gameId`** identity.

Tip: Identify other areas in the code where a custom Hook can improve the separation of concerns in the code. Create and use a custom Hook and make sure things still work after you do.

Note: The playground session for the final code can be found here.

Bonus challenges

Let's say you took this version to the client and they loved it, but of course they want more. Here are two major features that I'll leave you with as a bonus challenge on this game:

- **Track scores**: A perfect score of 3 happens when the player guesses all correct cells without any wrong attempts. If they make one wrong attempt the score is 2, and with two wrong attempts, the score is 1. Make the score time-aware. If the player finishes the game in the first five seconds, double their score. Display the total score in the UI somewhere. When the player wins the game again, add the new score to the total score.
- **Make it harder**: When the player clicks the **Play Again** button, make the grid bigger: **6 × 6**, then **7 × 7**, and so on. Also, increment the challenge size with each new game.

Chapter 6 What's Next

I hope you've enjoyed the exercises in this book. You can now build simple apps with React, and that's exactly what you need to do next. Before you build anything else, you should configure a local environment and configure your editor to work for a React project. I cover that in detail here.

Once you have an easy way to create a local environment and good tools in your editor to make your experience a more pleasant one, you should build a few small apps with the knowledge you gained in this book. Focus on the simple core concepts you gained here and build cool things. Build other games! See this page for some ideas. Build the infamous TODOs app. Build that web app idea your friends and relatives keep telling you about. Build something you care for.

Don't jump into the advanced areas yet—you should master the fundamentals first. Most of what needs to be done in simple React applications can be done with what you learned in this book. You'll need to learn many other things eventually. Here's a list of some of the important concepts and tools that you should learn after getting comfortable with the fundamentals:

- Learn the rest of React Hooks and understand the cases they cover. Most importantly, learn about **`useReducer`**, **`useRef`**, **`useContext`**, and **`useCallback`**.
- Learn about testing React components. One of the popular testing frameworks out there for React is Jest, and it's what I use in all my React projects. It's great! Give it a spin.
- Learn how to use React applications to server-side render static HTML as the starting state for the same React application that'll run in the browser.
- Learn how to associate React components with URLs and how to integrate them into the browser's history API.
- Learn how to profile and analyze the performance of React components.
- Learn about the lifecycle methods in React class-based components and the error boundaries that you can use with them. Learn the abstractions and patterns that are usually used with them, like higher-order components and the "render-props" method.
- Learn about dynamic props validation with tools like PropTypes, which will give better error messages in the console if you (or other members on your team) use a certain component incorrectly. You should also explore static analysis tools like TypeScript or Flow. You'll probably find them hard to use and hate them at first, but give them a fair try. After a few weeks, you'll see the HUGE advantage they bring to the table.
- Learn about React Cache and Suspense. It's still brand new as I am finishing this book, but it will be the standard way for a React component to work with remote data (and fetch it with Ajax).

These topics are beyond the scope of this beginner-level book. I cover them in other books and courses. Check out my *React Beyond the Basics* book here, and other advanced-level resources about React here.

After getting comfortable with React for the web, take a look at React Native. The same skills that you gain for React will enable you to build native iOS and Android applications. You'll be up and running in no time.

You should also take a look at GraphQL. It's nothing short of a game changer! If you're planning to build a data API to work with your React applications, do not start until you read a book about GraphQL, which I happen to have written. You can find *GraphQL In Action* here.

Finally, you'll need to learn Node.js. Your React application will most likely end up hosted as a Node application, and you'll need tools like Webpack and Babel to work with React (these are Node packages). If you ever decide to server-side-render your React application, using a Node web server is the easiest way to do that. Learn as much about Node as you possibly can—you will not regret it. Check out my *Node Beyond the Basics* book here.

I also have a few video courses about React, GraphQL, and Node that are available at Pluralsight, LinkedIn Learning, and jsComplete. You can find the links to all them here.

Please reach out with questions or feedback. Tweet me @samerbuna or find me at the jsComplete help Slack channel.

Thanks for reading. Now get busy!