



# React.js

Succinctly<sup>®</sup>

by Samer Buna

# React.js Succinctly

---

By  
Samer Buna

Foreword by Daniel Jebaraj



Copyright © 2016 by Syncfusion, Inc.  
2501 Aerial Center Parkway  
Suite 200  
Morrisville, NC 27560  
USA  
All rights reserved.

## **Important licensing information. Please read.**

This book is available for free download from [www.syncfusion.com](http://www.syncfusion.com) on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from [www.syncfusion.com](http://www.syncfusion.com).

This book is licensed for reading only if obtained from [www.syncfusion.com](http://www.syncfusion.com).

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.

**Technical Reviewer:** James McCaffrey

**Copy Editor:** Courtney Wright

**Acquisitions Coordinator:** Hillary Bowling, online marketing manager, Syncfusion, Inc.

**Proofreader:** Graham High, senior content producer, Syncfusion, Inc.

# Table of Contents

|  |           |
|--|-----------|
| <b>The Story behind the <i>Succinctly</i> Series of Books.....</b> | <b>6</b>  |
| <b>About the Author .....</b>                                      | <b>8</b>  |
| <b>Introduction .....</b>  | <b>9</b>  |
| <b>Chapter 1 What Is React?.....</b>                               | <b>11</b> |
| <b>Chapter 2 Why React?.....</b>                                   | <b>12</b> |
| Generating HTML .....  | 13        |
| Enhancing HTML.....  | 14        |
| React's way.....   | 15        |
| <b>Chapter 3 Declarative User Interfaces.....</b>                  | <b>19</b> |
| React's language.....  | 19        |
| To JSX or not to JSX .....   | 20        |
| <b>Chapter 4 React Components.....</b>                             | <b>22</b> |
| Readability .....  | 22        |
| Reusability.....   | 23        |
| Composability.....   | 24        |
| React's stateful components.....                                   | 25        |
| Creating React components.....                                     | 26        |
| <b>Chapter 5 Composability.....</b>                                | <b>33</b> |
| <b>Chapter 6 Reusability.....</b>                                  | <b>37</b> |
| Input validation .....   | 37        |
| Input default values .....   | 41        |
| Shared component behavior .....                                    | 42        |
| <b>Chapter 7 Working with User Input.....</b>                      | <b>45</b> |

|   |           |
|---|-----------|
| React's synthetic events .....                      | 45        |
| Working with DOM nodes in the browser .....         | 47        |
| Controlled components.....                          | 49        |
| <b>Chapter 8 Component Lifecycle.....</b>           | <b>51</b> |
| componentWillMount() .....                          | 56        |
| componentDidMount() .....                           | 57        |
| componentWillReceiveProps(nextProps) .....          | 61        |
| shouldComponentUpdate(nextProps, nextState).....    | 62        |
| componentWillUpdate(nextProps, nextState) .....     | 63        |
| componentDidUpdate(prevProps, prevState) .....      | 64        |
| componentWillUnmount() .....                        | 65        |
| <b>Chapter 9 Let's Build a Game with React.....</b> | <b>67</b> |
| The memory grid game.....                           | 67        |
| Implementation increments.....                      | 69        |
| Challenges .....                                    | 101       |

# The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President  
Syncfusion, Inc.

**S**taying 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](mailto: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

It was 1993 when my geeky uncle realized my passion for computers and set me up on a path to empower that passion. I am forever grateful to him.

I was just a kid back then, with a natural love for math and physics, but when I started learning to program (using Pascal at the time), I knew I found my life-long hobby!

I am not sure I can describe coding as my *hobby*; it is certainly a career that pays the bills. However, whenever I have some free time, I often just start coding for a side project, or learn that brand-new framework which smart person X tweeted about the other day.

I don't write in Pascal anymore; my favorite server-side language has been Ruby for over a decade, and for the past few years, I have been mainly coding in JavaScript. With the modern changes that are happening to JavaScript, I am starting to favor JavaScript over Ruby.

When React was first released, I was an Ember fan. I still favor Ember over Angular today, but after learning the simple and powerful React, I left the school of MVC frameworks for good. Today, I do my front-end work exclusively with React and vanilla JavaScript.

You can stalk me on Twitter [@samerbuna](https://twitter.com/samerbuna) or email me at [samer@agilelabs.com](mailto:samer@agilelabs.com). You can also find a list of all the books and courses I have authored on my LinkedIn profile at [linkedin.com/in/samerbuna](https://linkedin.com/in/samerbuna).

Special thanks to my friend Julia Hunt for her excellent contributions to this book. Her feedback was spot-on as always, and this book is much better because of her.

I hope that you enjoy this book and that it will teach you the great benefits of the React library.

Samer Buna



# Introduction

Since 2010 or so, Angular has been *the* MVC framework of choice for top developers. It's the most popular of all, and it offers the most impressive features out of the box. This all changed in 2015, and the change came from Facebook engineers this time. React started to claim a big portion of the market, with huge players adopting it: Netflix, Yahoo, and Airbnb, to name a few.

React is a huge revolutionary success; it's simple and effective, and it changed the way we design our views for good.

React is not an MVC framework—it's just a view library. The concepts introduced by React are what make it a big deal. Other frameworks out there, Angular included, have learned from what React does and copied it in parts or fully.

This story, however, is not about a shiny new library from Facebook that is taking over. The story here is about MVC not being the best approach for big applications, and REST APIs, which have been the standard so far, not being the best solution for data communication between clients and servers.

Facebook engineers are challenging both the MVC and the REST standards. Here are their alternative proposals, in summary:

- Instead of MVC and data-binding, a front-end system should have a one-way data flow. The views should never change the models directly; they can only read from the models. For writing, the views trigger actions that eventually get dispatched to the models (the models are referred to as “Stores” in this flow). React fits perfectly in the way it *reacts* to the data changes in the stores. React components will be listening to those changes, and React will re-render the views efficiently when those changes happen.
- Instead of REST and having the logic of what data to expose on the server-side, and instead of managing different end-points for different needs (for example, one end-point to expose top posts, one to expose a post, and one to expose a post with comments included), let the clients ask the servers for exactly what they need. The servers will then parse the clients' questions, and respond with what they asked for (no over-fetching or under-fetching). This is the concept behind the GraphQL runtime and query language that Facebook engineers released in 2015.
- Instead of the standard recommendation of separating data and views, have the data requirement expressed in the view component itself. Since the view knows exactly which data elements it needs to render, these two shouldn't be separate at all. This is the concept behind the Relay.js framework that Facebook engineers released in 2015.

This book will not cover GraphQL or Relay, but an understanding of React itself is the first step toward understanding the rest of the architecture proposed by Facebook engineers.

## Why did I write this book?

I love big frameworks (and I cannot lie). They serve a great purpose, especially for young teams and startups. Lots of smart design decisions are already made for us, and there is a clear path to writing good code. This allows us to focus on our application's logic.

Frameworks, however, come with some disadvantages, and for big applications and experienced developers, those disadvantages are sometimes a deal breaker.

I'll name two of the relevant disadvantages of going with a framework:

- Frameworks are not flexible, although some will claim to be. Frameworks want us to code everything a certain way; if we try to deviate from that way, the framework usually ends up fighting us about that.
- Frameworks are big and full of features. If we need to use only a small piece of them, we have to include the whole thing anyway (hopefully this will change with HTTP2).

Some frameworks are going modular, which I think is great, but I am a big fan of the Unix philosophy:

*“Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.”* —Doug McIlroy

React is a small program that does one thing, and it does it really well. There is a lot of buzz around the performance of React's virtual DOM, but I don't think that's the greatest thing about React. The virtual DOM performance is a nice plus.

What React does really well is speak a common language between developers and browsers that allows developers to declaratively describe user interfaces, and to model the state of these interfaces and not the transactions on them. Basically, React taught us and the machines a new language that we both now understand: the language of UI outcomes.

This realization was the most important one I made about React, and I think it deserves a book focused on making it clear to everyone learning React.

## Who should read this book?

You need basic knowledge of JavaScript to survive this book. This book will not teach you JavaScript. If you're comfortable with JavaScript itself, but have never used a JavaScript framework or library before, this book is for you.

If you're learning React after using other JavaScript libraries, this book will also have an answer to the “Why?” question that's probably on your mind: Why bother learning something new?

# Chapter 1 What Is React?

React is a JavaScript library that can be used to describe *views* (for example, HTML elements) based on some *state* (which is often in the form of data). When you're working with React in a browser, React can mount the described views in the browser's DOM (Document Object Model) and automatically update what needs to be updated whenever the original state changes.

React is a small but powerful library, with the power being more in the concepts than in the implementation. Some of the concepts under which React operates are:

## **Reusable, composable, and stateful components**

In React, we build views using smaller components. We can reuse a single component in multiple places, with different states and properties, and components can contain other components. Every component in a React application has a private state that may change over time, and React will take care of updating the component's view when its state changes.

## **The nature of reactive updates**

React's name is the simple explanation for this concept. When the state of a component changes, those changes need to be reflected somewhere. For example, we need to regenerate the HTML views for the browser's Document Object Model (DOM) whenever their state changes. With React, we do not need to worry about how to reflect the state changes; React will simply react to the changes and automatically update the views when needed.

## **The virtual representation of views in memory**

With React, we write HTML using JavaScript. We rely on the power of JavaScript to generate HTML that depends on some data, rather than enhancing HTML to make it work with that data. Enhancing HTML is what other JavaScript frameworks usually do. For example, Angular extends HTML with features like loops, conditionals, and others.

If we are receiving just the data from the server (with AJAX), we need something more than HTML to work with it, so it's either using an enhanced HTML, or using the power of JavaScript itself to generate the HTML. Both approaches have advantages and disadvantages, and React embraces the latter one, with the argument that the advantages are stronger than the disadvantages.

Using JavaScript to render HTML allows React to have a virtual representation of HTML in memory (which is aptly named the virtual DOM), and React uses that to render the views virtually first. Every time a state changes and we have a new HTML tree that needs to be written back to the browser's DOM, instead of writing the whole tree, React will only write the difference between the new tree and the previous tree since it has both trees in memory. This process is known as *tree reconciliation*, and I think it's the best thing that's happened in web development since AJAX!

## Chapter 2 Why React?

HTML is a great language, and what makes it great, in my opinion, is how it allows us to be declarative when building our webpages. We basically tell the browsers what we want with HTML.

For example, when we tell the browser to render a paragraph of text in a page, we don't care how the browser is actually going to do that. We don't have to worry about what steps the browser needs to do to make our text look like a paragraph, such as how many pixels of white space should be around it, how many words to display per line, or what to do when the screen is resized. We just want a paragraph, and with HTML we have the power to command the browser to make it happen.

But HTML by itself is not enough. We don't have loops, if statements, or variables in HTML, which means we can't write dynamic HTML based on data. HTML works great for static content, but when we have data in the equation, we need something else.

The simple truth is data is *always* in the equation. We write HTML to represent some data, even when we use HTML for static content. When we write static content, we are manually writing HTML by hand to represent some data, and that's okay for small sites where the data is not frequently changed.

The word “change” is key here. When the data we represent with HTML is changed, we need to manually update the HTML to reflect that change. For example, when that small business for which you built a website gets a new phone number, it'll be time to drop into an HTML editor, make that change, and then upload the new HTML files to the server.

If the nature of the change is more frequent though, for example, if the business now wants to feature a different product on the home page every hour, you will not agree (I hope) to make that manual change every hour. Imagine if Facebook was all static content, and every time you update your status someone at Facebook needs to edit an HTML file and upload it to the server.

We realized a long time ago that HTML is not enough, and we have since been trying to enhance HTML to make it support dynamic content.

Let's assume that we want to represent a list of products in HTML. For every product, we have a name and a price. Here's some sample data:

*Code Listing 1: Sample Products Data*

```
{
  "products": [
    { "name": "React.js Essentials", "price": 2999 },
    { "name": "Pro React", "price": 2856 },
    { "name": "Learning React Native", "price": 2199 }
  ]
}
```

A small business mentality would reason that we have only these three products, and their prices rarely change. Manually writing some static HTML for these three products would be okay.

But a list of products is dynamic—some products will be added, some will be removed, and the prices will change seasonally. Manually writing HTML will not scale, so we need to come up with HTML that always represents the last *state* of the products' data.

## Generating HTML

We can use database tables (or collections) to represent data in a database. For our products example, we can use a products table.

*Code Listing 2: Products Table*

```
create table products (  
  name text,  
  price integer  
)
```

The rows of this table each represent a single product. We can then give the client write access on the table to allow them to add, remove, and update their list of products whenever they want.

To come up with HTML to represent these products, we can read the last state of the data from this table and process it in some programming language—like Java—to make the program concatenate together HTML strings based on the data.

*Code Listing 3: HTML Strings*

```
html = "<html><body><ul>";  
  
// Loop over rows in table, and for every row:  
html += "<li>" + name + " - " + price + "</li>";  
  
// When the loop is done:  
html += "</ul></body></html>";
```

We can now write the value of that `html` variable to an HTML file and ship it. Every time the customer updates their data table, we can regenerate and ship the new HTML.

This worked great, and it gave us a lot of power to work with dynamic data, but something was wrong; it didn't feel right.

*We wanted to write HTML, not concatenate strings.*

## Enhancing HTML

A genius mind came up with an idea to create a new language that is somewhere between HTML and programming languages like Java. A language where we can still write normal HTML tags, but also use special tags that will act like loops and conditionals when we need them. JSP and PHP are examples of this.

Here's how we can represent our product data using an enhanced HTML language:

*Code Listing 4: HTML+ (Pseudocode)*

```
<html><body><ul>
  <% FOR each product in the list of products %>
    <li><%= the product's name %> - <%= the product's price %></li>
  <% END FOR %>
</ul></body></html>
```

This is clearly a bit more pleasant to work with than concatenating strings. The compiler will take care of doing the string concatenations for us, and we will get HTML that always reflects our exact data.

This has been the standard way to create dynamic websites for years. In fact, until recently, Facebook did exactly that to show you the list of updates from your friends.

But just like everything else in life, browsers evolved and became smarter. JavaScript became a standard language that is supported in all browsers, and someone invented AJAX, which allowed us to ask servers questions in the background of a webpage.

We realized that we could be more efficient in the way we represent our data. Instead of preparing the HTML server-side and shipping it ready to the client, we can send just the data to the client, and we'll have the fancy, smart browsers prepare the HTML themselves. JavaScript can do that.

Since we didn't want to concatenate strings, we needed something like JSP and PHP, but something that would work with the JavaScript engine in the browser.

Frameworks like Angular, Ember, and a handful of others were developed to make this process easier.

Just like server-side enhanced HTML languages, with Angular we do:

*Code Listing 5: Angular Loop*

```
<html><body><ul>
  <li *ngFor="#product of products">
    {{product.name}} - {{product.price}}
  </li>
</ul></body></html>
```

Now we're being more efficient over the wire, and we are generating dynamic HTML without having to deal with concatenating strings.

This worked great, but as we used it to build big applications, a few issues came up:

- Performance: Since we are no longer hitting the refresh button every time we click a link, our use of memory should now be managed more carefully.
- Browsers are great for rendering an HTML tree of elements dynamically, but they are still slow, especially when we need to update the tree. Every time an update is needed, there are tree traversals that need to happen, and when we reach the node and update it, the browser needs to redraw the user screen, which is an expensive process.
- In some cases, we were not modeling the state of our data directly with user interfaces, and instead we were writing code that expressed the steps and the transitions needed to update those interfaces. This is not ideal.

## React's way

The engineers at Facebook, realizing the problems of all other JavaScript frameworks, wanted something better, so they went back to the drawing board. A few months later, they came up with what eventually became React.

They had a couple simple rules to guide their genius solution:

- Enhancements to HTML are noise—let's just concatenate strings in JavaScript itself, and yes, sacrifice some readability for performance.
- The DOM is a bottleneck; we should avoid it as much as possible.

HTML views are hierarchical. An HTML document is a big tree of nodes. With the power and flexibility of JavaScript, Facebook engineers realized that if HTML nodes were represented as JavaScript objects, we would have a new world of possibilities for working with them.

Having the power and flexibility of JavaScript, Facebook engineers realized that if HTML nodes were represented as JavaScript objects, we would have a new world of possibilities to work with them.

That led to their first major decision: *Represent every HTML element with a JavaScript object.*



**Note:** If you want to test the React code samples in this book, the easiest way to get a React environment for testing is to use one of the web development online playground applications. My favorite is [JS Bin](#); [here's a React-ready bin](#) that you can fork to start testing.

React's API today has the `createElement` function, which can be used to do exactly that.

Code Listing 6: Void Element

```
// Instead of:


// We do:
React.createElement("img", { src: "logo.png" })
```

The `createElement` function returns a JavaScript object. To inspect that object, go to the [main React](#) website and paste the `createElement` line in the developer tools JavaScript console:

```
> React.createElement("img", { src: "logo.png" })
< ▶ Object {$$typeof: Symbol(react.element), type: "img", key: null, ref: null, props: Object...}
> |
```

Figure 1: `createElement` in JS Console

Notice the `type` property for that object, and take a look at the `props` property as well.

`<img />`, however, is a void (self-closing) element; let's take a look at another example for a normal element.

Code Listing 7: Normal Element

```
// Instead of:
<a href="/">Home</a>

// We do:
React.createElement("a", { href: "/" }, "Home");
```

To represent a parent-child relation, we can use the arguments of the `React.createElement` function. We can pass one or more React objects, starting from the third argument of `React.createElement`.

For example, to make the logo image clickable:

Code Listing 8: Parent-Child Relation

```
// Instead of:
<a href="/">
  
</a>

// We do:
React.createElement("a", { href: "/" },
```



```
React.createElement("img", { src: "logo.png" })
);
```

If we inspect this last object in the console, we'll notice how the **props** attribute for the `<a>` element has a **children** attribute, and that in turn holds the object for the `<img />` element. This is how we represent a tree with React.

Of course, under the hood, React will eventually generate HTML strings from these JavaScript objects and concatenate them together.

There are big benefits to writing HTML with JavaScript:

- We have the power of JavaScript itself to work with data. Our product example becomes:

*Code Listing 9: The Power of JavaScript*

```
React.createElement("ul", {},
  ...products.map(product =>
    React.createElement("li", {}, `${product.name} - ${product.price}`)
  )
);
```

- It's all JavaScript, so we don't mix HTML with new tags or anything. We also don't have to learn a new language, and instead learn how to use the few JavaScript functions that React defines.
- We'll always have a representation of our HTML in memory, separate from the one in the browser. This means that we can optimize our DOM operations by using a smart diffing algorithm on the structures we have in memory. This is only relevant when things change. This concept is what we call *the virtual DOM* in React.

## The virtual DOM

After rendering HTML for the product data example that we started with, we'll have three products with their names and prices displayed in the browser. With an **Ajax** call, we've determined that the price for product #2 has changed, and we now have a fourth product to display.

React has the original user interface state for three products stored in memory (because we wrote it in JavaScript first). Before sending anything to the browser, React will compute the new state, which now has four products.

With both states available in memory now, React can compute the difference between the new state and the original state, and it can conclude that it needs to add a node #4 and modify node #2. All other nodes will remain unchanged.

Having the original state in memory represents a big performance benefit. Other frameworks that don't have an original state in memory have two options:

- Generate the new state's UI and replace the whole tree in the browser.
- Try to change nodes #2 and #4, but rely on what is in the DOM. This results in making a bunch of READ operations in the DOM, and processing the tree to figure out where node #2 begins and where node #3 ends. These DOM operations are usually expensive.

React's virtual representation of the browser's DOM in memory (known as the virtual DOM) is clearly a better alternative. In fact, it's so good that most other JavaScript frameworks today have changed their algorithms to do exactly what React does for updating the DOM.

React's power is more than a virtual DOM with a smart diffing algorithm. Views in React have private state, so our product list view can have its original three products object represented in that private state, and when the new data comes in, we just update the private state. React triggers a DOM reload whenever the private state of the view changes. It basically hits a special refresh button in the browser.

From the point of view of a user inspecting the state, the view has been completely refreshed, and the new view represents the new state. But under the hood, React is doing this refresh operation smartly by using the diff it computed via the virtual DOM, and only applying that diff to the browser's tree. As developers, however, we don't have to worry about that; we just declare our state changes and let React take care of the steps needed to get those changes efficiently reflected in the browser's DOM.

This is so important to understand that it's worth repeating: With React, we model the state of our user interfaces, and not the transitions on them. If there are changes, we just model the new state.

React has effectively taught both the browsers and us developers to speak a new language: the language of user interface outcomes. We can now communicate with our browsers in this higher-level language that allows us to express our user interfaces in a declarative way.

Using the React language for our products example, we're basically saying, "Browser, I have three products for you." Then a minute later, we come back and say, "Browser, I now have four products for you." We don't have to worry about the changes in the prices or names, since this is not part of the language; it's a transition, and we just model the new state: four products. In fact, even the number of products does not matter because of the reactive update nature of React, which will take care of re-rendering the views when that number changes. Our command to the browser ends up being, "Browser, we have a list of products for you, and changes might happen to that list."

# Chapter 3 Declarative User Interfaces

## React's language

With React's new language of outcomes, which browsers now understand, we can build declarative user interfaces (like we used to do with pure HTML), but now we can work with data as well.

For our products example, we just command the browser to display a list of products; we don't even worry about how many products are actually on that list.

Our command to the browser is this: Display this list of products as an unordered list; for every product, display its name, then display its price.

After that, we're done. When the data we have for the list of products changes, we don't need to do anything to our UI to get it updated with the new data. Our command to the browser does not need to be changed.

In other words, if we have transitions on the data, we don't have to worry about the user interface—we just manage the new state of our data.

Data transitions include operations like the following:

- One product was removed from the catalog.
- Three more products were added to the catalog.
- A product name has changed.
- Five products' prices have changed.
- The order of the products in the catalog has changed.

Of course, we have to change our views if the *structure* of our data changes. For example, when we add a **Boolean** flag to every product to track whether the product is in stock or not, and we want to display the out-of-stock products in gray, this would be a structure transition, not just a data transition. Once we account for the new structure for the first time though, updating that new flag becomes just a data transition that we don't have to worry about in the UI.

This mental model about modeling the final state is much easier to understand and work with, especially when our views have lots of data transitions.

For example, consider the view that tells you how many of your friends are online. The view state will be just one single number of how many of friends are currently online. This view does not care that a moment ago, three friends came online, then one of them disconnected, then two more joined. It just knows that at this current moment, four friends are online.

## To JSX or not to JSX

Writing HTML in JavaScript gives us a lot of power and advantages over using templates and display logic. But working with HTML directly has great advantages that we would sacrifice in this case. HTML is familiar and concise, and if you work with designers, they would certainly hate to work with our JavaScript-created elements.

JSX is the optional compromise. It is a simple HTML-like JavaScript syntax extension used to create React elements. You can think of JSX as an enhancement of JavaScript to allow for syntax that looks like HTML. We can use it in the return statement of a component's render function.

JSX is labeled "HTML-like" because it can't be exactly HTML. Some element attributes have to be used the way the DOM API defines them; **class** and **for** are both examples of this. To use them in React, we use **className** and **htmlFor** instead.

The following example is an email input field interface.

*Code Listing 10: JSX*

```
// The desired HTML:
<form>
  <label for="email">Email:</label>
  <input type="email" id="email" class="form-control" />
</form>

// How to represent it with React:
React.createElement(
  "form",
  null,
  React.createElement(
    "label",
    { htmlFor: "email" },
    "Email:"
  ),
  React.createElement(
    "input",
    { type: "email", id: "email", className: "form-control" }
  )
);

// How to represent it with JSX:
<form>
  <label htmlFor="email">Email:</label>
  <input type="email" id="email" className="form-control" />
</form>
```

It's important to understand that JSX is completely optional, and not required to use React. We can write React in plain JavaScript; we don't need the JSX extension.

JSX is actually not shipped with React at all; we will need third-party tools to make our project work with JSX. The Facebook-recommended tool is Babel.

Writing components render functions in JSX instead of pure JavaScript has the advantage of being concise and familiar, and that's especially true for designers. Balanced opening and closing tags are much easier to read and parse with the human eye than JavaScript functions and object literals.

# Chapter 4 React Components

User interfaces are defined as components in React. The term *component* is used by many other frameworks. We can also write web components natively using HTML5 features like custom elements and HTML imports.

Components have many advantages, and whether we are working with them natively using HTML5, or using a library like React, we get the following great benefits.

## Readability

Consider this UI:

*Code Listing 11: HTML-Based UI*

```
<a href="http://facebook.com">
  
</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, maybe **ClickableImage** is a good name for it.

*Code Listing 12: Component-Based UI*

```
<ClickableImage />
```

When things get more complex, this parsing of HTML becomes harder, so components allow us to know quickly what the UI represents using the language that we’re comfortable with (English in this case).

Here’s a bigger example:

*Code Listing 13: Component-Based UI*

```
<TweetBox>
  <TextAreaWithLimit limit={140} />
  <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 remaining characters section, we know exactly where to go.

## Reusability

Think of components as functions. This analogy is actually very close to the truth, especially in React. Functions take input, they do something (possibly on that input), and then give us back an output.

In functional programming, we also have *pure* functions, which are basically protected against any outside state; if we give them the same input, we'll always get the same output.

In React, a component is modeled after a function. Every component has private properties, which act like the input to that function, and we have a virtual DOM output. If the component does not depend on anything outside of its definition (for example, if it does not use a global variable), then we label that component pure as well.

All React components can be reused in the same application and across multiple applications. Pure components, however, have a better chance at being reused without any problems.

For an example, let's implement our **ClickableImage** component.

*Code Listing 14: ClickableImage Render Function*

```
var ClickableImage = function(props) {  
  return (  
    <a href={props.href}>  
      <img src={props.src} />  
    </a>  
  );  
};  
  
ReactDOM.render(  
  <ClickableImage href="http://google.com" src="http://goo.gl/Q1B7w1" />,  
  document.getElementById("react")  
);
```

Having variables for both the **href** and the **src** properties is what makes this component reusable.

Note how we defined the component as an actual function. We can create React components in multiple ways. The simplest way is to use a normal JavaScript function that receives the component's **props** as an argument.

The function's output is the virtual HTML view, which this component represents.

Don't worry about the syntax now—just focus on the concepts. To reuse this component, we could do something like render **ClickableImage** with a Google logo:

```
props = { href: "http://google.com", src: "google.png" }
```

We can simply reuse the same component with different props:

```
props = { href: "http://bing.com", src: "bing.png" }
```

The **src** properties should be replaced with actual images, as we did in the render function of the previous example.

## Composability

We create components to represent views. For ReactDOM, the React components we define will represent HTML DOM nodes.

The **ClickableImage** component in the last example 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 15: SearchEngines Mockup*

```
var SearchEngines = function(props) {  
  return (  
    <div className="search-engines">  
      <ClickableImage href="http://google.com" src="google.png" />  
      <ClickableImage href="http://bing.com" src="bing.png" />  
    </div>  
  );  
}
```

If, for example, we have the data in this format:

*Code Listing 16: SearchEngines Data*

```
var 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 to a list of **ClickableImage** components:

*Code Listing 17: SearchEngines Render Function*

```
var SearchEngines = function(props) {  
  return (  
    <List>  
      {props.data.map(engine => <ClickableImage {...engine} />)}  
    </List>  
  );  
}
```



```

    );
  };

  ...

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

```

The three dots in `...engine` mean spread the attribute of an engine as flat properties for **ClickableImage**, which is equivalent to doing:

```
href={engine.href} src={engine.src}
```

This **SearchEngines** component is now reusable. It can work with any list of search engines we give to it. We also used the **ClickableImage** component to *compose* the **SearchEngines** component.

## React's stateful components

Everything changes eventually. In React, a component manages its changes using a *state* object. In addition to the data we pass to components as props, React components can also have a private state, which can change over time.

For example, a timer component can store its current timer value in its state.

*Code Listing 18: Timer Component*

```
<Timer initialSeconds={42} />
```

This timer will start at 42 seconds and count down to 0, decrementing its state every second. At second 0, its private state will be 42; at second 1, the private state will be 41; and at second 42, the private state will be 0.

With React, we just make the timer component display its private state.

*Code Listing 19: Timer Render Function*

```

function() {
  return (
    <div>
      {this.state.counter}
    </div>
  )
}

```

```
}
```

Every second, we decrement the counter state.

*Code Listing 20: Changing the State (Pseudocode)*

```
Every second:  
  state.counter = state.counter - 1  
  if state.counter reaches 0  
    stop the timer
```

Here's the great news: React components recognize the private state changes. When changes happen, React automatically hits the Refresh button for us and re-renders the UI of a component. This is where React gets its name—it will react to the state changes, and reflect them in the UI.

## Creating React components

It's time to officially learn how to create React components.

Let's define our **ClickableImage** component, which understands two input properties, **href** and **src**. Once we have this **ClickableImage** component ready, we can mount it in the browser using the **ReactDOM** render function.

*Code Listing 21: Rendering a React Component to the DOM*

```
ReactDOM.render(  
  <ClickableImage href="google.com", src="google.com" />,  
  document.getElementById("react")  
);
```



**Note:** *ReactDOM is a library that's maintained separately and can be used with React to work with a browser's DOM. To be able to use it, you need to include its CDN entry or import it in your project. This [JSBin template](#) has a working example of a component mounted with ReactDOM.*

The first argument to the **ReactDOM.render** method is the React element that needs to be rendered, and the second argument is where to render that element in the browser. In this case, we're rendering it to the HTML node with **id="react"**.

There are three main ways to define a React component:

- Stateless function components
- `React.createClass`

- `React.Component`

## Stateless function components

Since components are modeled after functions, we can use a vanilla JavaScript function to write pure components:

*Code Listing 22: Stateless Function Component*

```
var ClickableImage = function(props) {  
  return (  
    <a href={props.href}>  
      <img src={props.src} />  
    </a>  
  );  
};
```

When we use a function component, we're not creating an instance from a component *class*; rather, the function itself represents what would be the render method in a regular component definition. If we design our application in a functional and declarative way, most of our components could just be simple stateless function components.

Stateless function components can't have any internal state, they don't expose any lifecycle methods, and we can't attach any refs to them. If we need any of these features (which I will explain in later chapters), we will need a class-based component definition.

With the new ES2015 arrow function syntax, the **ClickableImage** component can be defined more concisely with:

*Code Listing 23: Stateless Function Component with Arrow Function*

```
var ClickableImage = props => (  
  <a href={props.href}>  
    <img src={props.src} />  
  </a>  
);
```

## React.createClass

React has an official API to define a stateful component. Our simple **ClickableImage** component would be:

*Code Listing 24: **React.createClass** Syntax*

```
var ClickableImage = React.createClass({  
  render: function() {  
    return (  

```

```

    <a href={this.props.href}>
      <img src={this.props.src} />
    </a>
  );
}
});

```

The **createClass** function takes a single argument, a JavaScript configuration object. That object requires one property, the **render** property, which is where we define the component's function that describes its UI.

Note how with **createClass**, we don't pass the **props** object to the **render** call. Instead, elements created from this component class can access their properties using **this.props** within the **render** function.

The **this** keyword references the *instance* of the component that we mounted in the DOM (using **ReactDOM.render**). Every time we mount a **<ClickableImage />** element, we're creating an instance of the **ClickableImage** component class.

In object-oriented programming terms, **ClickableImage** is the class, and **<ClickableImage />** is the object instantiated from that class.

With **createClass**, we can use the private state of the component object, and we can invoke custom behavior in its lifecycle methods. To demonstrate both concepts, let's implement a **Timer** component.

First, here's how we're going to use this **Timer** component:

*Code Listing 25: Rendering the Timer Component*

```

ReactDOM.render(
  <Timer initialSeconds={42} />,
  document.getElementById("react")
);

```

The simple definition of this component, before considering the private state or the tick operation, is:

*Code Listing 26: Timer Component **render()** Function*

```

var Timer = React.createClass({
  render: function() {
    return (
      <div>{this.state.counter}</div>
    );
  }
});

```

The **counter** variable is part of the private state within a component instance. The private state object can be accessed using **this.state**.

Our **Timer** component has the property **initialSeconds**, which is where the counter should start. Properties of a component instance can be accessed using **this.props**, so if we need to read the value that we're passing to the **initialSeconds** property, we use **this.props.initialSeconds**.

The state of a React component can be initialized using a **getInitialState** function in the **createClass** definition object. Anything we return from the **getInitialState** function will be used as the initial private state for a component instance.

We want the initial state for our counter variable to start as **this.props.initialSeconds**, so we do the following:

*Code Listing 27: Timer Component Initial State*

```
var Timer = React.createClass({
  getInitialState: function() {
    return {
      counter: this.props.initialSeconds
    };
  },
  render: function() {
    return (
      <div>{this.state.counter}</div>
    );
  }
});
```

Let's now define the “tick” operation. We can use a vanilla **setInterval** function to tick every second (1000 milliseconds). Inside the interval function, we need to change our component state and decrement the counter.

The ticking operation should start after the component gets rendered to the DOM so that we're sure there is a **div** in the DOM whose content we can now control. For that, we need a lifecycle method.

Lifecycle methods act like hooks for us to define custom behavior at certain points in the lifecycle of a React component instance. The one we need here is **componentDidMount()**, and it allows us to define a custom behavior right after the component is mounted in the DOM.

*Code Listing 28: Timer Interval in **componentDidMount()***

```
var Timer = React.createClass({
  getInitialState: function() {
    return { counter: this.props.initialSeconds };
  },
  componentDidMount: function() {
```

```

    var component = this;
    setInterval(function() {
      component.setState({
        counter: component.state.counter - 1
      });
    }, 1000);
  },
  render: function() {
    return <div>{this.state.counter}</div>;
  }
});

```

There are a couple of things to notice here:

- We had to use a closure around the **setInterval** so that we have access to the **this** keyword in there. It's an old-school JavaScript trick (which we don't need anymore with the new ES2015 arrow function syntax).
- To change the state of the component, we used the **setState** function. In React, we should never mutate the state directly as a variable. All changes to the state should be done using **setState**.

This **Timer** component is ready, except that the timer will not stop, and it will keep going to the negative side. We can use the **clearTimeout** function to stop the timer. Go ahead and try to do that for our component, and come back to see the following full solution.

*Code Listing 29: Timer Component Full Definition*

```

var Timer = React.createClass({
  getInitialState: function() {
    return { counter: this.props.initialSeconds };
  },
  componentDidMount: function() {
    var component = this, currentCounter;
    component.timerId = setInterval(function() {
      currentCounter = component.state.counter;
      if (currentCounter === 1) {
        clearInterval(component.timerId);
      }
      component.setState({ counter: currentCounter - 1 });
    }, 1000);
  },
  render: function() {
    return <div>{this.state.counter}</div>;
  }
});

ReactDOM.render(

```

```
<Timer initialSeconds={42} />,
document.getElementById("react")
);
```

## React.Component

ES2015 was finalized in 2015, and with it, we can now use the class syntax. A class is syntax sugar for JavaScript's constructor functions, and classes can inherit from each other using the **extends** keyword.

*Code Listing 30: ES2015 Class Syntax*

```
class Student extends Person { }
```

With this line, we define a new **Student** class that inherits from a **Person** class.

The React API has a class that we can extend to define a React component. Our **ClickableImage** definition becomes:

*Code Listing 31: React.Component Syntax*

```
class ClickableImage extends React.Component {
  render() {
    return (
      <a href={this.props.href}>
        <img src={this.props.src} />
      </a>
    );
  }
}
```

Within the class definition, the **render** function is basically the same, except that we used a new ES2015 syntax to define it. The word **function** can be completely avoided in ES2015.

Let's look at our **Timer** example using the ES2015 syntax. Try to identify the differences.

*Code Listing 32: Timer Component Using Class Syntax*

```
class Timer extends React.Component {
  constructor(props) {
    super(props);
    this.state = { counter: this.props.initialSeconds };
  }
  componentDidMount() {
    let currentCounter;
    this.timerId = setInterval(() => {
```

```

        currentCounter = this.state.counter;
        if (currentCounter === 1) {
            clearInterval(this.timerId);
        }
        this.setState({ counter: currentCounter - 1 });
    }, 1000);
}
render() {
    return (
        <div>{this.state.counter}</div>
    );
}
}
}

```

Here are the differences explained:

- Instead of **getInitialState**, we now use the ES2015 class constructor function, and just assign a value to **this.state** in the constructor. We need to invoke the **super()** call there as well to fire the **React.Component** constructor call.
- In the **setInterval**, we used an arrow function, i.e. **() => { }**. With an arrow function, we don't need to do the closure trick we did before because arrow functions have a lexically bound **"this"** that captures the enclosing context by default.
- All the functions are defined using the new function property syntax. For example, **componentDidMount() { ... }**.

## Component classes, elements, and instances

Sometimes you'll find these terms mixed up in guides and tutorials. It's important to understand that we have three different things here:

- What's usually referred to as "Component" is the class. The blueprint. The global definition. In the **Timer** example, the variable **Timer** itself is the component class.
- **<Timer />** on the other hand, is a React element that we constructed from the **Timer** component class. This is a stateless, immutable virtual DOM object.
- When a React element is mounted in the browser's DOM, it becomes a component instance, which is stateful. The result of a **ReactDOM.render** call is a component instance.



# Chapter 5 Composability

Several React components can be combined to produce another React component. This is one of the best features of React. It's a simple concept with great advantages.

Composability enables abstraction and allows us to understand code without having to care about all the details all the time. If a **Profile** component is composed of a **ProfilePicture** component and a **ContactInformation** component, we'll have a pretty good idea about what's going on there without looking at the details of each component.

Composability also enables a more uniform behavior. If we have a **ContactInformation** component, whenever we need to display a person's contact information, whether that person is a guest, client, vendor, or employee of your business, we can uniformly use the **ContactInformation** component. This also means less repetition of code in any UI that represents a contact information section.

Every time we reuse a component to write another, we are cashing out our original time investment that we put into creating the original one.

The most important benefit of composability, however, is that it allows us to separate the different concerns of our applications with great flexibility.

Let me explain composability with an example. Let's try to describe Twitter's account page with components.

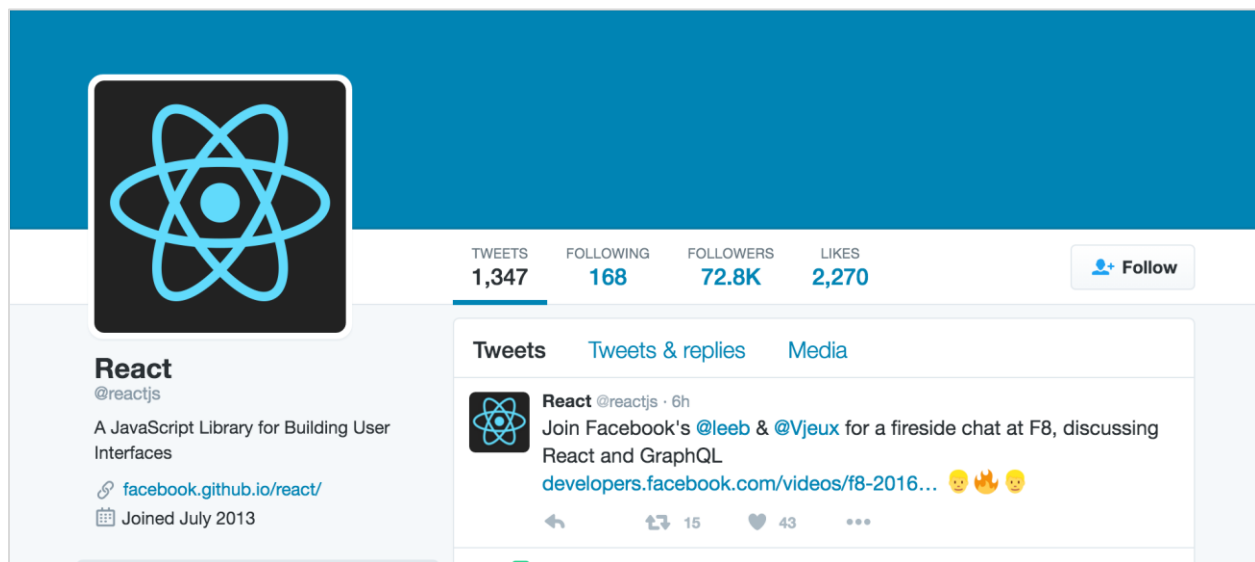


Figure 2: Twitter's User Profile Page

A simplified component list for this page could be something like the following (omitting the details of every component for brevity):

Code Listing 33: Twitter *UserHomePage*

```
class Avatar extends React.Component {
  render() {
    return <img ... />;
  }
}

class UserInfo extends React.Component {
  render() {
    // name, bio, ...
  }
}

class Counts extends React.Component {
  render() {
    // tweets, following, ...
  }
}

class Tweets extends React.Component {
  render() {
    // The list of tweets.
  }
}

class UserHomePage extends React.Component {
  render() {
    return (
      <div>
        <Avatar src={} />
        <UserInfo user={} />
        <Counts tweets={} following={} ... />
        <Tweets list={} />
      </div>
    );
  }
}
```

Understanding the hierarchical relationships between components here is important. Since **Avatar**, **UserInfo**, **Counts**, and **Tweets** all appear in the **render** method for **UserHomePage**, this makes **UserHomePage** the owner of them.

An owner component can set the **props** of the components it uses in the **render** method. For example, the **src** property for the **Avatar** component was passed to it from its owner, **UserHomePage**.

## Component children

Just like we can include HTML elements within the opening and closing tags of another HTML element, we can also include React elements within the opening and closing tags of another React element. We can mix HTML elements and React elements in both cases.

*Code Listing 34: Children Example*

```
<Counts>
  <TweetsCount />
  <FollowingCount />
  <FollowersCount />
  <LikesCount />
</Counts>
```

The inner elements (**TweetsCount**, **FollowingCount**, etc.) are called the “children” of the outer element (**Counts**). Within the definition of the **Counts** component, we can access the list of children used within any instance of the component using the special **this.props.children** property.

*Code Listing 35: **this.props.children***

```
class Counts extends React.Component {
  render() {
    <div id="counts-headers">
      {this.props.children}
    </div>
  }
}
```

React’s tree reconciliation process uses the order of the children in the diffing algorithm. This unfortunately means that if a child is removed from the tree, all siblings after it will need to be updated in the process. Be aware of this problem.

This problem becomes a serious one when the children are driven by a dynamic array that can be shifted and unshifted. It’s also a problem when the elements of that array are shuffled.

To force React to respect the identity and state of each child in a list, we need to uniquely identify each child instance by assigning it a **key** prop.

*Code Listing 36: **key** Prop*

```
class ProductList extends React.Component {
  render() {
    return (
      <div>
        {this.products.map(product =>
          <Product key={product.id} product={product} />)}
      </div>
    )
  }
}
```

```
    }  
  }  
);  
}
```

We used **product.id** here as the special **key** prop for the **Product** component.

React will actually warn us if we do not supply a unique key to a mapped array. Take that warning—and all React's warnings, really—seriously.

Note that this **key** prop cannot be used within the **Product** component definition—it will not be available as **this.props.key**.

Do not use the index of the array elements as the **key** value here. If you do that, when you remove an element from the array, you would practically be changing the identities of the children, and you'll get very unpredictable behavior.

### One-way data binding flow

When owners set the props of owned components, they are doing one-way data binding there. The owner will most likely bind the props of its owned components based on some computations on its own props and state. This process happens recursively, and any changes in the computations for the top-level owner component will be automatically reflected everywhere down the chain where they're passed to other components.

# Chapter 6 Reusability

Function components are the best components when it comes to reusability because they are pure function with no state. They are predictable—the same input will always give us the same output.

Stateful components can be reusable too, as long as the state being managed is completely private to every instance of the component. When we start depending on an external state, reusability becomes more challenging. There are a few things we can add to a React component to enhance its reusability score.

## Input validation

If you are the only one using your component, you probably know exactly what that component needs in terms of input. What are the different props needed? What data type does it need for every prop?

When that component is reused in multiple places, the manual validation of input becomes a problem. Sometimes you end up with a component that does not work, but no errors are being reported.

Even if you are the only one using your component, and you use it only once, you may not have complete control over the input values. For example, you could be feeding the component something that is read from an API, which makes the component dependent on what the API returns, and that could change any time.

Let's say you have a component that takes an array of unique numbers and displays each one of them in a `div`. The component also takes a number value, and it highlights that value in the array if it exists.

Here's a possible implementation with a pure stateless component:

*Code Listing 37: **NumbersList***

```
const NumbersList = props => (  
  <div>  
    {props.list.map(number =>  
      <div key={number} className={`highlight-${number === props.x}`}>  
        {number}  
      </div>  
    )}  
  </div>  
);
```

**props.list** is the array of numbers, and **props.x** is the number to highlight if it exists in **props.list**.

To style highlighted elements differently, let's just give them a different color.

```
.highlight-true {  
  color: red;  
}
```

Here's how we can use this **NumbersList** component in the DOM:

*Code Listing 38: Using **NumbersList***

```
ReactDOM.render(  
  <NumbersList list={[5, 42, 12]} x={42} />,  
  document.getElementById("react")  
);
```

Now, try to feed this component the string **"42"** instead of a number.

*Code Listing 39: Wrong Prop Type*

```
ReactDOM.render(  
  <NumbersList list={[5, 42, 12]} x="42" />,  
  document.getElementById("react")  
);
```

Not only will things not work as expected, but we will also not get any errors or warnings about the problem.

It would be great if we could perform input validation on this **x** prop (and all other props), effectively telling the component to expect only an integer value for **x**.

React has a way for us to do exactly that, through **React.PropTypes**.

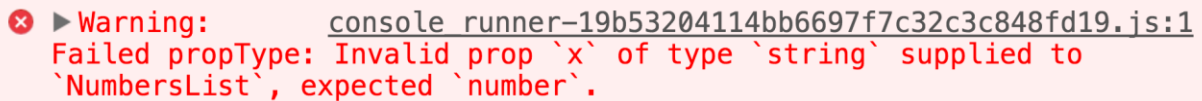
We can set a **propTypes** property on every component. In that property, we can provide an object where the keys are the input props that need to be validated, and the values map to what data type React should use to validate. For example, our **NumbersList** component should have **x** validated as a **number**.

Here's how we can do that:

*Code Listing 40: React **PropTypes***

```
NumbersList.propTypes = {  
  x: React.PropTypes.number  
};
```

If we now try to pass **x** as a string instead of a number, React will give us an explicit warning about it in the console.

A screenshot of a console runner window showing a warning. The text is: "Warning: console\_runner-19b53204114bb6697f7c32c3c848fd19.js:1 Failed propTypes: Invalid prop `x` of type `string` supplied to `NumbersList`, expected `number`." The warning is preceded by a red 'x' icon and the text is in red.

```
✖ Warning: console_runner-19b53204114bb6697f7c32c3c848fd19.js:1
Failed propTypes: Invalid prop `x` of type `string` supplied to
`NumbersList`, expected `number`.
```

*Figure 3: Invalid Prop Type Warning*

If we pass a correct numeric value for `x`, we won't get any warnings.

While these validation errors only show up in development (primarily for performance reasons), they are extremely helpful in making sure all developers use the components correctly. They also make debugging problems in components an easier task.

`React.PropTypes` has a range of validators we can use on any input. Here are some examples:

- JavaScript primitive validators:
  - `React.PropTypes.array`
  - `React.PropTypes.bool`
  - `React.PropTypes.number`
  - `React.PropTypes.object`
  - `React.PropTypes.string`
- `React.PropTypes.node`: Anything that can be rendered.
- `React.PropTypes.element`: A React element.
- `React.PropTypes.instanceOf(SomeClass)`: This uses the `instanceof` operator.
- `React.PropTypes.oneOf(['Approved', 'Rejected'])`: For ENUMs.
- `React.PropTypes.oneOfType([...])`: Either this or that.
- `React.PropTypes.arrayOf(React.PropTypes.number)`: An array of a certain type.
- `React.PropTypes.objectOf(React.PropTypes.number)`: An object with property values of a certain type.
- `React.PropTypes.func`: A function reference.

By default, all props we pass to components are optional. We can change that using `React.PropTypes.isRequired`, which can be chained to other validators.

For example, to make `x` required in our `NumbersList` example, we can do the following:

Code Listing 41: *isRequired*

```
NumbersList.propTypes = {  
  x: React.PropTypes.number.isRequired  
};
```

And if we try to omit **x** from the props at this point, we'll get the following:

```
✖ ▶ Warning: console_runner-19b53204114bb6697f7c32c3c848fd19.js:1  
Failed propTypes: Required prop `x` was not specified in  
`NumbersList`.
```

>

Figure 4: Required Prop Type Warning

In addition to the list of validators we can use with **React.PropTypes**, we can also use custom validators. These are just functions that we can use to do any custom checks on inputs. For example, to validate that the input value **tweetText** is not longer than 140 characters, we can do something like:

Code Listing 43: Custom *PropTypes*

```
Tweet.propTypes = {  
  tweetText: (props, propName) => {  
    if (props[propName] && props[propName].length > 140) {  
      return new Error('Too long');  
    }  
  }  
}
```

For components created with **React.createClass**, **propTypes** is just a property on the input object. For the regular class syntax, we can use a static property (which is a proposed feature in JavaScript).

Code Listing 44: *propTypes* Syntax

```
// With React.createClass syntax:  
let NumbersList = React.createClass({  
  propTypes: {  
    x: React.PropTypes.number  
  },  
});  
  
// For class syntax:  
class NumbersList extends React.Component {  
  static propTypes = {  
    x: React.PropTypes.number  
  };  
};
```



```

}

// For functions syntax (and also works for class syntax):
NumberList.propTypes = {
  x: React.PropTypes.number
}

```

## Input default values

One other useful thing we can do on any input prop for a React component is assign it a default value in case we use the component without passing a value to that prop.

For example, here's an alert box component that displays an error message in an alert **div**. If we don't specify any message when we use it, it will default to "Something went wrong."

*Code Listing 45: React **defaultProps***

```

const AlertBox = props => (
  <div className="alert alert-danger">
    {props.message}
  </div>
);

AlertBox.defaultProps = {
  message: "Something went wrong"
};

ReactDOM.render(
  <AlertBox />,
  document.getElementById("react")
);

```

The syntax to use **defaultProps** with **React.Component** is similar to **propTypes**. For **React.createClass**, we use the method **getDefaultProps** instead:

*Code Listing 46: **getDefaultProps()***

```

const AlertBox = React.createClass({
  getDefaultProps() {
    return { message: "Something went wrong" };
  }
});

```

## Shared component behavior

Sometimes, reusing the whole component is not an option; we'll have cases where some components share most of their functionalities but are different in a few areas.

If we need different components to share common behavior, we have two options:

- If we're using the **React.createClass** syntax, we can use *mixins*.
- If we're using the **React.Component** class syntax, we can create a new class and have all components extend it. We can also manually inject the external methods we'd like our classes to have in their constructor functions.

Mixins are objects that we can “mix” into any component defined with **React.createClass**.

For example, in one application, we have links and buttons with unique **ids**, and we would like to track all clicks that happen on them. Every time a user clicks a button or a link, we want to hit an API endpoint to log that event to our database.

We have two components in this app, **Link** and **Button**, and both have a click handler **handleClick**.

To log every click before handling it, we need something like this:

*Code Listing 47: LogClick Function*

```
logClick() {  
  console.log(`Element ${this.props.id} clicked`);  
  $.post(`/clicks/${this.props.id}`);  
}
```

We can add this method to both components, but instead of repeating it twice, we can put **logClick()** in a mixin, and include the mixin in every component where we need to log the clicks.

Here's how we put the **logClick** feature in a mixin and use it in both **Link** and **Button**:

*Code Listing 48: LogClickMixin*

```
const logClicksMixin = {  
  logClick() {  
    console.log(`Element ${this.props.id} clicked`);  
    $.post(`/clicks/${this.props.id}`);  
  },  
};  
  
const Link = React.createClass({  
  mixins: [logClicksMixin],  
  handleClick(e) {
```

```

    this.logClick();
    e.preventDefault();
    console.log("Handling a link click...");
  },
  render() {
    return (
      <a href="#" onClick={this.handleClick}>Link</a>
    );
  }
});

const Button = React.createClass({
  mixins: [logClicksMixin],
  handleClick(e) {
    this.logClick();
    e.preventDefault();
    console.log("Handling a button click...");
  },
  render() {
    return (
      <button onClick={this.handleClick}>Button</button>
    );
  }
});

ReactDOM.render(
  <div>
    <Link id="link1" />
    <br /><br />
    <Button id="button1" />
  </div>,
  document.getElementById("react")
);

```

This is a better approach, since now the **logClick** implementation is abstracted in one place. If in the future we need to change that implementation, we only need to do it in that one place.

If we want to do the exact same thing with **React.Component** class syntax (which is vanilla JavaScript where we don't have mixins), we have many options. Here's one possible way: Create a new class **Loggable**, implement the **logClick** in there, and then make both **Button** and **Link** components extend **Loggable**.

*Code Listing 49: The **Loggable** Class*

```

class Loggable extends React.Component {
  logClick() {
    console.log(`Element ${this.props.id} clicked`);
    $.post(`/clicks/${this.props.id}`);
  }
}

```

```

    }
}

class Link extends Loggable {
  handleClick(e) {
    this.logClick();
    e.preventDefault();
    console.log("Handling a link click...");
  }
  render() {
    return (
      <a href="#" onClick={this.handleClick.bind(this)}>Link</a>
    );
  }
}

class Button extends Loggable {
  handleClick(e) {
    this.logClick();
    e.preventDefault();
    console.log("Handling a button click...");
  }
  render() {
    return (
      <button onClick={this.handleClick.bind(this)}>Button</button>
    );
  }
}

ReactDOM.render(
  <div>
    <Link id="link1" />
    <br /><br />
    <Button id="button1" />
  </div>,
  document.getElementById("react")
);

```

# Chapter 7 Working with User Input

We modeled our data state and declaratively defined our UI as a function of our state. It's now time to start accounting for UI events and capturing input from the user.

The nature of user input can be described as a reverse data flow when compared to how React's components represent data. In fact, with active user input on a page, the data starts depending on the UI.

Suppose, for example, we have a UI that displays a list of comments. We start with an array of comments in the data layer, make the UI represent that, and then add a form on the page to add a new comment. Our array of data now depends on the user interactions with this form.

If we handle the input events, capture the data that users enter, and then manipulate our original data source to account for the new input, React will refresh the views that are using that data.

I'll first explain the simple way to work with user input: *refs attributes*. Then I'll explain how to work with user input using the recommended way: *controlled components*.

Before we go into that, however, we need to understand the synthetic events system in React.

## React's synthetic events

We can define native browser events on DOM elements. For example, to `console.log` the content of a `div` when it's clicked, we can do the following:

*Code Listing 50: HTML Event*

```
<div onclick="console.log(this.textContent);">
  You might not need React...
</div>
```

To do that exact same thing in React, we use an `onClick` synthetic event.

*Code Listing 51: React Event*

```
<div onClick={se => console.log(se.target.textContent)} >
  You might not need React...
</div>
```

The differences between these events are:

- Casing is important. React uses **camelCase** for the event attributes (`onClick`, `onKeyPress`).

- We don't use strings with React events; we use JavaScript directly.
- We use a function for the value. The function gets defined when we define the component, and it will be invoked when we click on the element.

An **onClick** used within a React component works with a synthetic event. That's why I named the variable passed to the function "**se**". React's synthetic events are wrappers around the browser's native events. We can still access the native event object itself using the **nativeEvent** attribute on any synthetic event. We can see the native event if we **console.log(se.nativeEvent)** in the previous example.

Browsers historically had significant differences in the way they handled and invoked their native events. This is getting a lot better with the W3C standards now, but browser quirks are not to be trusted. With React synthetic events, we don't have to worry about that. React will make sure events work identically across all browsers.

One notable example of an event that React made standard between multiple inputs and in all browsers is the **onChange** event. The native **onChange** event works differently among different input types, and it usually does not mean "change," but rather that "the user is done changing." With React events, the **onChange** event means that on any change, anywhere, anytime, and for any input:

- If we're typing in an **input** or **textarea** element, React will fire the **onChange** event on every key stroke (much like keyboard events, but also with support for paste and automated entries).
- If we select or clear a check box, React will fire the **onChange** event.
- When we select an option from a drop-down select element, React will fire the **onChange** event.

Here are examples of some of the popular events that we can use with React:

- Keyboard events: **onKeyDown**, **onKeyPress**, **onKeyUp**
- Focus events: **onFocus**, **onBlur**
- Form events: **onChange**, **onInput**, **onSubmit**
- Touch events: **onTouchStart**, **onTouchEnd**, **onTouchMove**, **onTouchCancel**
- Mouse events: **onClick**, **onDrag**, **onMouseOver**, **onMouseOut**, etc.

There are also clipboard events, UI events, wheel events, composition events, media events, and image events.

We can even work with events that are not supported in React by using **addEventListener** (usually in **componentDidMount**) and **removeEventListener** (in **componentWillUnmount**).

## Working with DOM nodes in the browser

React's powerful performance feature, the virtual DOM, frees us from ever needing to touch the DOM. When adding React's synthetic events to the mix, React (in a way) has created a "fake browser" that respects all the standards, is way faster, and is a lot easier to work with.

This, however, does not mean that we can't use the original browser's DOM and events with React if we need to. Sometimes we need to do special things with the DOM, such as integrate our components with third-party libraries.

If we need to access a native DOM node in any React component, we can use React's special attribute, **ref**.

We can pass that attribute either a string or a function. You'll find a lot of examples that use a string value (which acts like a unique identifier), but don't do that; instead, use functions.

To work through an example, let's say we have an email input field where the user is supposed to type an email, and we have a Save button. Before we save, we want to make sure that the input is a valid email. If we use the `<input type="email" />` element, we can use the native API method `element.checkValidity()` to figure out whether or not the user entered a valid email. To invoke that API method, we need access to the native DOM element.

Here's the full example implemented with React's **ref** attribute:

*Code Listing 52: Ref Attributes*

```
const EmailForm = React.createClass({
  handleClick() {
    if (this.inputRef.checkValidity()) {
      console.log(`Email Ok. Saving Email as ${this.inputRef.value}`);
    }
  },
  render() {
    return (
      <div>
        <input type="email" ref={inputRef => this.inputRef = inputRef} />
        <button onClick={this.handleClick}>Save</button>
      </div>
    );
  }
});

ReactDOM.render(<EmailForm />, document.getElementById("react"));
```

When we define a **ref** attribute on the input element and use a function for its value, React will execute that function when the input element gets mounted with the **EmailForm** component. React will also pass a reference to the DOM input element (**inputRef** in the example) as an argument to that **ref** function. Inside the **ref** function, we can access the **EmailForm** component instance via the **this** keyword, so we can store the input reference as an instance variable.

Once we have the reference to the native DOM element (**this.inputRef**), we can access all of its API normally. In **handleClick**, we are calling the native **checkValidity()** function on the DOM reference.

We can use the same trick to read the value of any input at any point. In **handleClick**, the **console.log** line actually reports the email input text value (using the native API property **"value"**) when the input is valid.

### Changing a React component's state with native API calls

We can use React's **ref** attributes to read input from the user by assigning component variables to input fields like we did in the previous example. However, every time the user types something into an input field, they're practically changing the "state" of that field. Since we are representing elements with stateful components in React, not reflecting the input state change back to the enclosing component's state means that our component is no longer an exact representation of its current state that is defined in React.

Here's an example to make that point easy to understand:

*Code Listing 53: Input State*

```
const EmailForm = React.createClass({
  getInitialState() {
    return {
      currentEmail: this.props.currentEmail
    };
  },
  render() {
    return (
      <div>
        <input type="email" value={this.state.currentEmail} />
        <button>Save</button>
      </div>
    );
  }
});

ReactDOM.render(
  <EmailForm currentEmail="mark@fb.com" />,
  document.getElementById("react")
);
```



We rendered the same **EmailForm** to save an email field, but this time we're displaying an initial value of the email field, and users can change it if they want (an "edit" feature).

Since the email is something that can be changed in this component, we put it on the state of the component. React rendered the input email and used the component state to display the default value. If we change the **currentEmail** state in memory, React will update the value displayed in the input box.

However, if the user is allowed to change the value of that input field directly using the browser API (by typing in the text box), then the displayed DOM in the browser will be different than the current copy of the virtual DOM that we have in memory because that one is reflecting the **currentEmail** state, which has not changed. If something else changes in the state of the **EmailForm** component, React will re-render the **EmailForm** component, and the value the user typed in the email will be lost. This is why React would not allow the user to type in this example's email input.

React components should always represent their state in the DOM, and the input field is part of that state. If we can make the input field always represent the React component's state, even when the user types in it, then we can just read React's state whenever we need to read the new input from the user.

A component where we control the input to always reflect the state is called a *controlled component*.

## Controlled components

By using controlled components, we don't have to reach to an input field's native DOM element to read its value since whatever the user types in there will be reflected on the component state itself.

To achieve this level of control, we use an **onChange** event handler. Every time a change event is fired, we update the component state associated with the input.

Here's the previous example updated to be a controlled component:

*Code Listing 54: Controlled Component*

```
const EmailForm = React.createClass({
  getInitialState() {
    return { currentEmail: this.props.currentEmail };
  },
  setCurrentEmailState(se) {
    this.setState({ currentEmail: se.target.value });
  },
  handleClick() {
    console.log(`Saving New Email value: ${this.state.currentEmail}`);
  },
  render() {
```

```
    return (
      <div>
        <input type="email" value={this.state.currentEmail}
          onChange={this.setCurrentEmailState} />
        <button onClick={this.handleClick}>Save</button>
      </div>
    );
  }
})

ReactDOM.render(
  <EmailForm currentEmail="mark@fb.com" />,
  document.getElementById("react")
);
```

By using the **setCurrentEmailState** **onChange** handler, we're updating the state of the **EmailForm** component every time the user types in the email field. This way we make sure that the new DOM in the browser and React's in-memory virtual DOM are both in sync, reflecting the current state of the **EmailForm** component.

Note how in the **console.log** line, we can read any new input value the user entered using **this.state.currentEmail**.

## Chapter 8 Component Lifecycle

Every React component has a story.

The story starts when we define the component class. This is the *template* that we use every time we want to create a component instance to be mounted in the browser.

Let me tell you the story of a **Quote** component that we are going to use to display funny, short quotes on a webpage. The **Quote** story begins when we define its class, which might start with a mockup like the following:

*Code Listing 55: Quote Component Mockup*

```
class Quote extends React.Component {
  render() {
    return (
      <div className="quote-container">
        <div className="quote-body">Quote body here...</div>
        <div className="quote-author-name">Quote author here...</div>
      </div>
    );
  }
}
```

The component class is our guide for the markup that should be used to represent a single quote element. By looking at this guide, we know that we're going to display the quote body and its author's name. The previous definition, however, is just a mockup of what a quote would look like. To make the component class useful and able to generate different quotes, the definition should be made generic.

For example:

*Code Listing 56: Generic Quote Component*

```
class Quote extends React.Component {
  render() {
    return (
      <div className="quote-container">
        <div className="quote-body">{this.props.body}</div>
        <div className="quote-author-name">{this.props.authorName}</div>
      </div>
    );
  }
}
```

This template is now ready to be used to represent any quote object, as long as it has a **body** attribute and an **authorName** attribute.

Our **Quote** component story continues; the next major event in its history is when we *instantiate* it. This is when we tell the component class to *generate* a copy from the template to represent an actual quote data object.

For example:

Code Listing 57: Quote React Element

```
<Quote body="..." authorName="..." />
```

The instantiated **<Quote />** element is now full-term and ready to be *born*. We can render it somewhere (for example, in the browser).

Let's define some actual data to help us through the next events in our component's lifecycle.

Code Listing 58: Quotes Data

```
var quotesData = [
  {
    body: "Insanity is hereditary. You get it from your children",
    authorName: "Sam Levenson"
  },
  {
    body: "Be yourself; everyone else is already taken",
    authorName: "Oscar Wilde" },
  {
    body: "Underpromise and overdeliver",
    authorName: "Unknown"
  },
  ...
];
```

To render the first quote in the browser, we first instantiate it with an object representing the first quote in our data: **quotesData[0]**.

Code Listing 59: Instantiating a React Element

```
var quote1 = quotesData[0];

var quote1Element = <Quote body={quote1.body}
                        authorName={quote1.authorName} />;

// Or using the spread operator
var quote1Element = <Quote {...quote1} />;
```

We now have an official **<Quote />** element (**quote1Element**), which represents the first object from our quote data.

Let's take a look at its content.

Code Listing 60: `renderToStaticMarkup`

```
console.log(ReactDOMServer.renderToStaticMarkup(quote1Element));

// Output
<div className="quote-container">
  <div className="quote-body">
    Insanity is hereditary. You get it from your children.
  </div>
  <div className="quote-author-name">
    Sam Levenson
  </div>
</div>
```

The output will be one big string that represents the first quote in our data, according to the HTML template defined in the component class.

We used `renderToStaticMarkup` to inspect the content, which works just like `render`, but does not require a DOM node. `renderToStaticMarkup` is a useful method if we want to generate static HTML from data. There is also a `renderToString` method that's similar, but compatible with React's virtual DOM on the client. We can use it to generate HTML on the server and send it to the client on the initial request of the application. This makes for a faster page load and allows search engines to crawl your application and see the actual data, not just JavaScript with one empty HTML node.



**Note:** Applications that leverage the trick of rendering HTML for the initial request are known as *universal applications* (or *isomorphic applications*). They use the same components to render a ready HTML string for any client (including search engine bots). Normal clients will also get the static version, and they can start their process with it. For example, if we give React on the client-side a static version generated on the server-side using the same components, React will start by doing nothing at first, and it will update the DOM only when the state changes.

Both `renderToString` and `renderToStaticMarkup` are part of the `ReactDOMServer` library, which we can import from `"react-dom/server"`.

Code Listing 61: `ReactDOMServer` (ES2015 Import Syntax)

```
import ReactDOMServer from "react-dom/server";

// For static content
ReactDOMServer.renderToStaticMarkup(<Quote {...quote1} />);

// To work with React virtual DOM
ReactDOMServer.renderToString(<Quote {...quote1} />);
```

However, on the client-side, we would like our application to be interactive, so we need to render it using the regular `ReactDOM.render` method.

Code Listing 62: Component Instance

```
ReactDOM.render(  
  <Quote {...quote1} />,  
  document.getElementById("react")  
);
```

A **Quote** component instance is now in the browser, fully mounted, and is part of the browser's native DOM.

React has two lifecycle methods that we can use to inject custom behavior before or after a component instance is mounted in the DOM. These are **componentWillMount** and **componentDidMount**.

The best way to understand these lifecycle methods is to define them in our component class and put a debugger line in both of them.

Code Listing 63: Understanding Lifecycle Methods

```
class Quote extends React.Component {  
  componentWillMount() {  
    console.log("componentWillMount...");  
    debugger;  
  }  
  componentDidMount() {  
    console.log("componentDidMount...");  
    debugger;  
  }  
  
  render() { ... }  
}
```

If we run this in the browser now, dev tools will stop the execution for debugging twice.

The first stop will be in **componentWillMount**. React exposes this method for us to write custom behavior *before* the DOM of the component instance is written to the browser. In the following figure, notice how the browser's document would still be empty at this point.

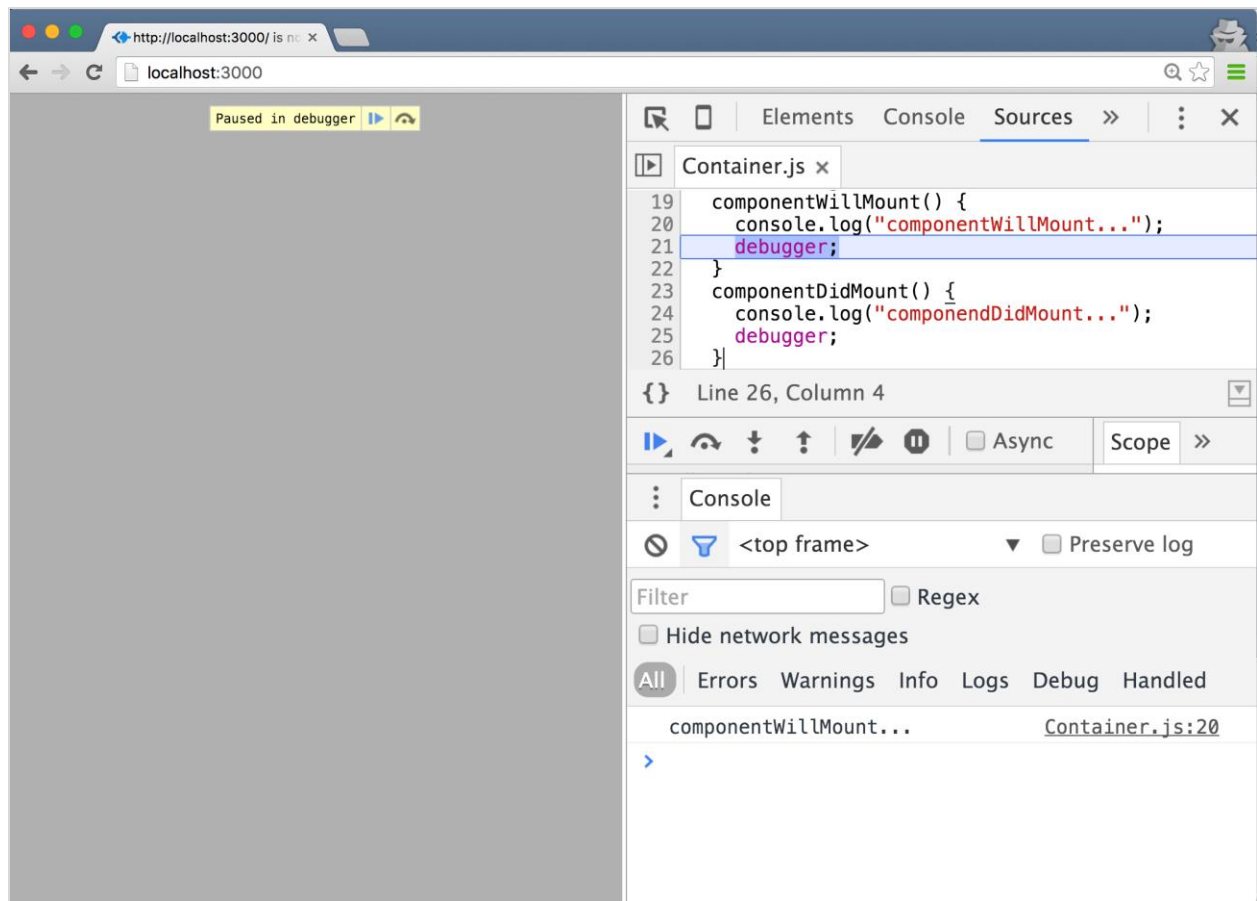


Figure 5: *componentWillMount()* Debugger Line

The second step will be in **componentDidMount()**. React exposes this method for us to write custom behavior *after* the DOM of the component instance is written to the browser. In the following figure, notice how the browser's document would show the HTML for our first quote at this point.

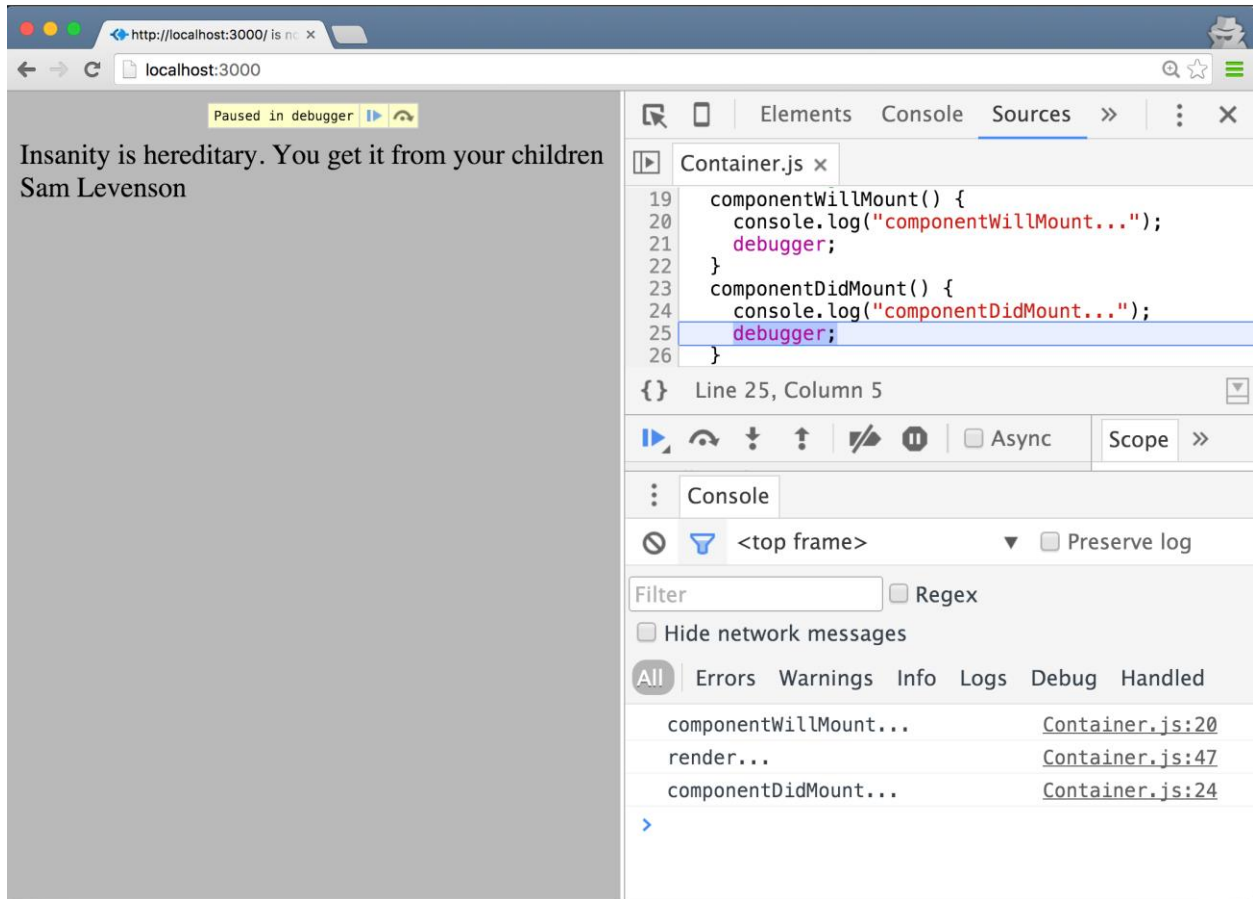


Figure 6: `componentDidMount()` Debugger Line

React exposes other lifecycle methods for updating and unmounting components. Every lifecycle method has specific advantages and use cases. I'll give one practical example for each method so that you can understand them in context.

## componentWillMount()

React invokes this method right before it attempts to render the component instance to its target. This works on both the client (when we use `ReactDOM.render`) and the server (with `ReactDOMServer` render methods).

### Practical example

We want to create a log entry in our database every time a quote is rendered using our `Quote` component. This should include quotes rendered server-side for search engine optimization (SEO) purposes. Since `componentWillMount` is triggered on both the client and the server, it's an ideal place to implement this feature.



Assuming the API team wrote an endpoint for us to use, and that we just need to post to `/componentLog` and send it the name of the component and its used props, and also assuming we have an AJAX library (like `jQuery.ajax`, for example), we can do something like this:

Code Listing 64: `componentWillMount()`

```
componentWillMount() {
  Ajax.post("/componentLog", {
    name: this.constructor.name,
    props: this.props
  });
}
```

## `componentDidMount()`

React invokes this method right after it successfully mounts the component instance inside the browser. This only happens when we use `ReactDOM.render`. React does not invoke `componentDidMount` when we use `ReactDOMServer` render methods.

`componentDidMount` is the ideal place to make our component integrate with plugins and APIs to customize the rendered DOM.

### Practical example

The boss wants you to integrate an API to the **Quote** component to display how popular a quote is. To get the current popularity rate of a quote, you need to hit an API endpoint:

**`https://ratings.example.com/quotes/<quote-text-here>`**

The API will give you a number between 1 and 5, which you can use to show the popularity on a five-star scale.

The boss also requested that this feature is not to be implemented server-side because it would slow down the initial render, and the feature should not block the rendering of a quote on the client. The quote should be rendered right away, and once we have a value for its stars-rating, display it.

We can't use `componentWillMount` here because it is invoked on both server and client render calls. `componentDidMount`, on the other hand, is invoked only on client calls.

Since we need to display the stars-rating number in our component somewhere, and since it's not part of the component props but instead read from an external source, we'll need to make it part of the component's state to make sure React is going to trigger a re-render of the component when the stars-rating variable gets a value.

We can do something like this:

Code Listing 65: `componentDidMount()`

```
componentDidMount() {  
  Ajax.get(`https://rating.example.com/quotes/${this.props.body}`)  
    .then(starRating => this.setState({ starRating }));  
}
```

Once the quote is displayed in the browser, we initiate a request to the API, and when we have the data back from the API, we'll tell React to re-render the component's DOM (which would now have the stars-rating) by using a `setState` call.



**Note:** Be careful about using `setState` *inside* `componentDidMount`, as it usually leads to twice the amount of browser render operations.

Integrating jQuery plugins is another popular task where `componentDidMount` is a good option, but be careful with that—when we add event listeners to the mounted component's DOM, we need to remove them if the component is unmounted. React exposes the lifecycle method `componentWillUnmount` for that purpose.

To see the rest of the component lifecycle methods in action, let's add control buttons to our Quotes application to enable users to browse through all the quotes we have. So far, we're only showing the first quote.

Let's create a **Container** component to host the currently active quote instance plus all the control buttons we need. The only state needed by this **Container** component will be the index of the current quote. To show the next quote, we just increment the index.

The **Container** component would be something like:

Code Listing 66: **Container** Component

```
class Container extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = { currentQuoteIdx: 0 };  
  }  
  render() {  
    var currentQuote = this.props.quotesData[this.state.currentQuoteIdx];  
    return (  
      <div className="container">  
        <Quote {...currentQuote} />  
        <hr />  
        <div className="control-buttons">  
          <button>Previous Quote</button>  
          <button>Next Quote</button>  
        </div>  
      </div>  
    );  
  }  
}
```

```
}  
}
```

And we use it with:

*Code Listing 67: Using the **Container** Component*

```
ReactDOM.render(  
  <Container quotesData={quotesData} />,  
  document.getElementById("react")  
);
```

This is what we should see in the browser at this point:

Insanity is hereditary. You get it from your children  
Sam Levenson

---

Previous Quote   Next Quote

*Figure 7: One Quote and Buttons*

Now let's make the buttons work. All we need to do is increment or decrement the **currentQuoteIdx** in the buttons' click handlers.

Here's one way to do that:

*Code Listing 68: **nextQuote** Function*

```
nextQuote(increment) {  
  var newQuoteIdx = this.state.currentQuoteIdx + increment;  
  if (!this.props.quotesData[newQuoteIdx]) {  
    return;  
  }  
  this.setState({ currentQuoteIdx: newQuoteIdx });  
}
```

Define the **nextQuote** function in the **Container** component class.

The **if** statement protects against going beyond the limits of our data—clicking “Previous Quote” on the first quote or “Next Quote” on the last one would do nothing.



**Note:** The *if* statement check is the minimum validation that we should do. Buttons should also be disabled if they can't be clicked. Try to implement that on your own.

Here's how to use the **nextQuote** handler when we click the buttons:

Code Listing 69: Buttons Click Handlers

```
<button onClick={this.nextQuote.bind(this, -1)}>
  Previous Quote
</button>
<button onClick={this.nextQuote.bind(this, 1)}>
  Next Quote
</button>
```

The bind call here is basically a fancy way to wrap our **nextQuote** function with another function, but this new outer function would remember the increment variable value for each button.

Go ahead and try the buttons now. They should work.

Every time we click on the buttons (given that the **if** statement in the handler is false), we're updating the DOM for the **<Quote />** element. We're doing this through React by controlling the props passed to the mounted **<Quote />**.

Here's what happens in more detail:

- The user clicks the "Next Quote" button.
- The **<Container />** instance gets a new value for the **currentQuoteIdx** state.
- React responds to the state change in **<Container />** by triggering its **render()** function.
- React computes the new DOM for the **<Container />** instance, and that involves re-rendering the **<Quote />** instance. Since the **currentQuoteIdx** was incremented, the **currentQuote** object would now be different than the one we used previously.
- In a way, React *updates* the mounted **<Quote />** instance with new props.

During that process, React invokes four lifecycle methods for us to customize the behavior if we need to. Let's see them in action.

Code Listing 70: **Quote** Update Lifecycle Methods

```
class Quote extends React.Component {
  componentWillMount() {
    console.log("componentWillReceiveProps...");
  }
}
```

```

    debugger;
  }
  shouldComponentUpdate() {
    console.log("shouldComponentUpdate...");
    debugger;
    return true;
  }
  componentWillUpdate() {
    console.log("componentWillUpdate...");
    debugger;
  }
  componentDidUpdate() {
    console.log("componentDidUpdate...");
    debugger;
  }
}

render() { ... }
}

```

Refresh your browser now, and note how none of these **console.log** lines will fire up on the initial render of the first quote. However, when we click **Next Quote**, we'll see all of them fire, one by one.

I've added debugger lines here for you to see the UI state between these four stages. For the first three, the browser's DOM will still have the old quote displayed, and once you get to the **componentDidUpdate** debugger line, you'll see the new quote in the browser.

Let me explain these methods with practical examples.

## componentWillReceiveProps(nextProps)

Whenever a mounted component gets called with a new set of props, React will invoke this method passing the new props as the argument.

### Practical example

You wrote a random even number generator function **generateEvenRandomNumber**, and you used it in a component **TestRun** to render a random even number in the browser every second using a **setInterval** call.

*Code Listing 71: TestRun*

```

// Render every second:
<TestRun randomNumber={generateEvenRandomNumber()} />

```

To test the accuracy of your generator code, you rendered 100 of these **<TestRun />** instances in your browser and let the timers run for a while.

You want to make sure that no component is rendered with an odd number. Instead of watching the components, you can use **componentWillReceiveProps** to make the component “remember” if it was rendered with an odd number, and how many times this happened.

Code Listing 72: *TestRun* Component

```
class TestRun extends React.Component {
  constructor(props) {
    super(props);
    this.state = { badRuns: 0 };
  }
  componentWillReceiveProps(nextProps) {
    if (nextProps.randomNumber % 2 === 1) {
      // Bad Run. Log it.
      this.setState({ badRuns: this.state.badRuns + 1 });
    }
  }
  render() { ... }
}
```



**Note:** React will trigger the *componentWillReceiveProps* method even if *nextProps* is identical to *currentProps*. The virtual DOM operation is what determines if a render to the DOM should actually happen.

## shouldComponentUpdate(nextProps, nextState)

This is a special method. If you noticed, when we tested the update lifecycle methods, this was the only one where we returned **true**.

This method is similar to **componentWillReceiveProps**, but has some differences:

- In addition to **nextProps**, React also passes a **nextState** object to this method.
- If we write code that returns **false** in this method, the update process will be stopped, and the component will not be updated. That’s why we returned **true** when we tested **shouldComponentUpdate** previously. Go ahead and test returning **false** there instead, and see how the “Next” and “Previous” buttons stop working.

This method can be used to enhance the performance of some React components. If a component only uses its props and state in the **render** function and nothing global, for example, we can compare the current props and state with **nextProps** and **nextState** in **shouldComponentUpdate**, and return **false** if the component is receiving similar values.

Components that only read from props and state in their **render** functions are known as *pure components*. They’re very similar to pure functions in the sense that their return value (the output of **render**) is only determined by their input values (**props** and **state**).

For pure components we can safely do the following:

*Code Listing 73: Pure Components*

```
class PureComponentExample extends React.Component {
  shouldComponentUpdate(nextProps, nextState) {
    return notEqual(this.props, nextProps) ||
      notEqual(this.state, nextState);
  }
  render() {
    // Read only from this.props and this.state
    // Don't use any global state
  }
}
```

**notEqual()** would be a function that can compare two objects for their keys' values.

### Practical example

You have a component that takes a timestamp prop and renders the date part of it, ignoring the time.

*Code Listing 74: Date Component Element*

```
<Date timestamp={new Date()} />
```

If we're rendering this component frequently, the only time we would actually want it to update would be tomorrow, so we can short-circuit the update process with **shouldComponentUpdate()**.

*Code Listing 75: Date Component **shouldComponentUpdate()***

```
class Date extends React.Component {
  shouldComponentUpdate(nextProps, nextState) {
    return this.props.timestamp.toString() !==
      nextProps.timestamp.toString();
  }
  render() { ... }
}
```

## **componentWillUpdate(nextProps, nextState)**

When a mounted component receives new props, or when its state changes, React invokes the **componentWillUpdate** method. This happens right before the **render** function is called.

Note that if we customized `shouldComponentUpdate` and returned `false`, React will not invoke `componentWillUpdate`.

We cannot use `setState` in `componentWillUpdate`. It's simply too late for that.

## Practical example

In our quotes application, we're now updating a single `<Quote />` instance to render multiple quotes. The database log entry that we do in `componentWillMount` is not going to be invoked when we click the Next Quote button. `componentWillMount` is only invoked on the initial render.

For this example, we could use `componentWillUpdate` to invoke the exact same code we used in `componentWillMount`.

*Code Listing 76: `componentWillUpdate()` Reusing Code*

```
logEntry(component) {
  Ajax.post("/componentLog", {
    name: component.constructor.name,
    props: component.props
  });
}
componentWillMount() {
  logEntry(this);
}
componentWillUpdate() {
  logEntry(this);
}
```

Note, however, that this method gets invoked every time React re-renders, even if it's rendering with the exact same props. If we want the log entry to only happen when the props change, we'll need to introduce an `if` statement about that in `componentWillUpdate`.

## `componentDidUpdate(prevProps, prevState)`

React invokes this final method after a component is updated and after the changes are synced to the browser. If we need access to the previous props and state, we can read them from the parameters of this method.

Just like `componentDidMount`, `componentDidUpdate` is helpful when we want to integrate external libraries with our components, set up listeners, or work with an API.



## Practical example

Our **componentDidMount** example also applies to **componentDidUpdate**, given the change we made to render a new quote by updating the props. However, since we're potentially hitting an external API for this example, we should be careful about doing so directly in **componentDidUpdate**, because we might be hitting the API endpoint for a stars-rating value that we already have.

One thing we can do for this current example is to simply cache the stars-rating values we read from the API.

*Code Listing 77: **componentDidUpdate()** Reusing Code*

```
setStarRating(ci) {
  if (ci.starRatings[ci.props.id]) {
    ci.setState({ starRating: ci.starRatings[ci.props.id] });
    return;
  }
  Ajax.get("https://rating.example.com/quotes/" + ci.props.body)
    .then(starRating => {
      ci.starRatings[ci.props.id] = starRating;
      ci.setState({ starRating });
    });
}
componentDidMount() {
  setStarRating(this);
}
componentDidUpdate() {
  setStarRating(this);
}
```

Note how we used a component instance variable (**ci.starRating**) to hold the cache of all API calls. We can use instance variables when we don't need React to trigger a re-render call when their values change.

There are a lot of similarities between pairs of mounting and updating lifecycle methods, and often you'll find yourself extracting code into another function and invoking that function from multiple methods (which is what we did in the previous example). However, the separation is helpful sometimes, especially when you want to integrate third-party code, like jQuery plugins, with your initially rendered DOM.

The last lifecycle method you should be aware of is **componentWillUnmount**.

## **componentWillUnmount()**

React invokes this method right before it attempts to remove a component instance from the DOM.

To see an example of that, put a **componentWillUnmount** method on the **Quote** class.

*Code Listing 78: **componentWillUnmount()***

```
componentWillUnmount() {  
  console.log("componentWillUnmount...");  
}
```

Then try to remove all mounted content from the DOM. **ReactDOM** has a method for that.

*Code Listing 79: **unmountComponentAtNode()***

```
ReactDOM.unmountComponentAtNode(document.getElementById("react"));
```

This will unmount any React components rendered inside the element passed to it as an argument. We should see the **console.log** line from **componentWillMount** in the console.

### Practical example

When we set up listeners or start timers in **componentDidMount**, we should clear them in **componentWillUnmount**.

*Code Listing 80: Start and Stop Listeners*

```
componentDidMount() {  
  // start listening for event X when triggered by Y  
}  
componentWillUnmount() {  
  // stop listening for event X when triggered by Y  
}
```

# Chapter 9 Let's Build a Game with React

## The memory grid game

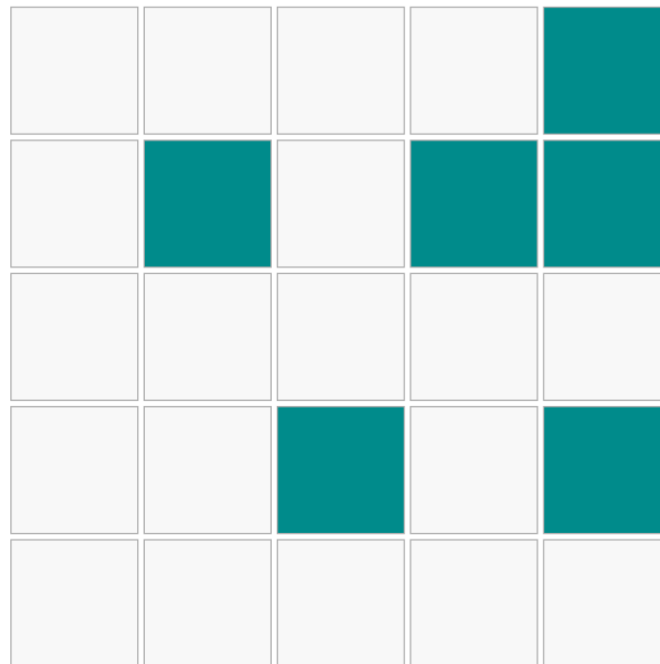
Let's use what we learned so far to build something from scratch with React. Instead of building a boring example app, let's build a somewhat entertaining game.

I love memory games; I think they are a fantastic way to "maintain" your useful short-term memory. I picked a popular and simple memory game for us to build together. This game has many names with many variations, but I'll just call it "the memory grid."

This game will show the player a grid with many cells and highlight a few random ones. The player is expected to memorize the positions of the highlighted cells. The game will then clear the highlighted cells and challenge the user to recall their positions from memory.

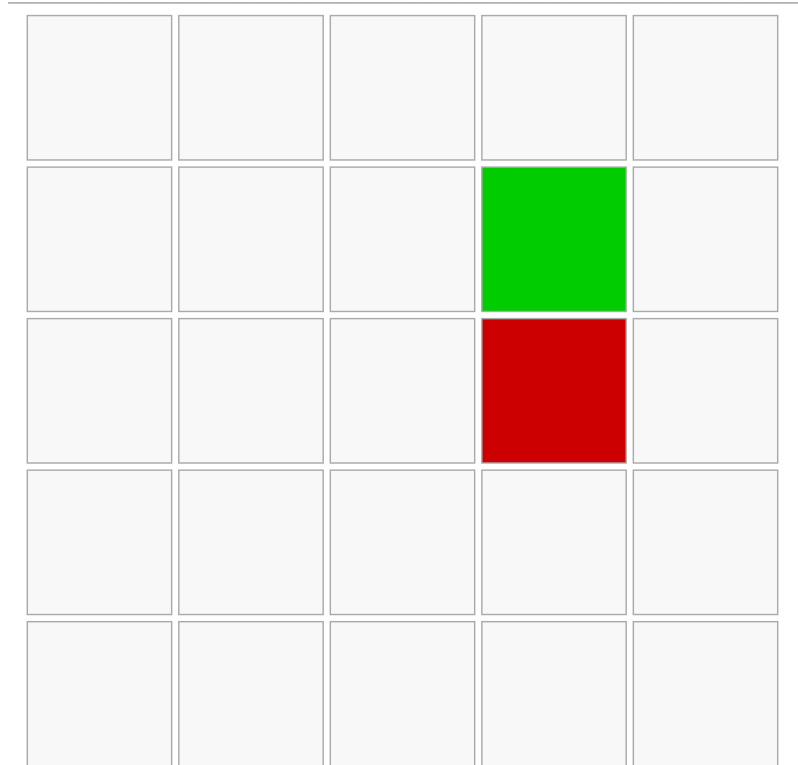
I picked this game because it's not a trivial example, but is still simple enough to fit in one chapter. The game also has a few moving parts: there will be nested components with different states on different levels, there will be timers, and there will be user input through clicks.

To see a demo of what exactly we will be building in this chapter, you can play the final version of the memory grid game [here](#).



Memorize...

Figure 8: The Memorize State



## Recall...

*Figure 9: The Recall State*

Here's what the client told us when they requested the game:

*You start with an empty grid that is X by Y (make it configurable). Tell the player to get ready, then highlight a number of cells (also configurable) for a few seconds while telling the user to memorize those cells. The game then goes into "Recall" mode, where the user is challenged to remember those cells after we clear the cells from the grid. If they guess a correct cell, mark it green, and if they guess a wrong cell, mark it red. They win if they guess all the right cells with a maximum of two wrong attempts.*

This request is not too bad; you'll be lucky to get a detailed request like that from a client. Usually, client requests are vague and you have to innovate around them—that's why we need agility in the process by always engaging the clients when we have something ready to be tested.

As with any complex problem, the key to getting to a solution is to split the problem into small, manageable chunks. Focus on the smallest testable increment that you can do next that would push the project forward a tiny bit, and implement just that.

# Implementation increments

## Increment 1: Make some decisions

- We will use a locally hosted React app, because that's realistically what you'll be doing from now on. This would mean that we need to do a few configuration steps to get things working. Alternatively, you can do the whole game on sites like jsbin.com or codepen.io, but I'd recommend that you follow along and get a local React app working. It's not too hard, I promise.
- We will use JSX for this project. This means we need a tool to process all files that have JSX in them for React. We will use [Babel.js](#) to do that.
- We will use JSPM, the [JavaScript Package Manager](#), to invoke Babel.js and bundle our app for the browser. JSPM is possibly the easiest way to start a local React project.
- We will use the modern ES2015 JavaScript syntax. By using Babel.js, we don't have to worry about browser support (which might be irrelevant by the time you read this anyway). We'll also use the React class syntax to define our components.
- We will be as modular as possible; we'll put every component in its own file, and we'll try to keep components small and pure.
- We will not be spending a lot of time on the styling of the game. We're doing this to learn React. We'll do minimal styling where needed.
- We will not write tests for this game to keep it simple. This is a bad decision, and unless you're writing an example for a *Succinctly* series book, do not make a similar decision.
- We will focus on readability over performance.
- We will minimize the use of component state where possible.



***Note: I'll be intentionally "planting" some bugs and problems in this game and potentially introduce other not-so-intentional ones. Try to identify problems as we make progress on the game. I'll solve the intentional ones eventually (after I challenge you to find them), but if you find other bugs, please make sure to report them with an issue on the GitHub project of the game.***

When you're done following along in this book, the full source code of the game is available on [GitHub](#).

## Increment 2: Write the full specifications of the game

The client's paragraph is a good start, but there are gaps in the specifications that we should attempt to fill:

- The game starts out with a grid rendered in the center of the screen, with X rows and Y columns. By default, it will start as 5 x 5.
- Under the initial grid, a hint message will read “Get Ready...”
- After two seconds, the hint message under the grid will change to “Memorize...” and the grid will highlight Z cells in blue. The Z cells will be picked randomly. By default, Z will be 6.
- The user can’t do anything up to this point except watch what’s happening.
- After two more seconds, the hint message under the grid will change to “Recall...”, and the grid will clear the six highlighted cells.
- The game will remember the positions of the highlighted cells before it clears them.
- At this point, the user can click on any cell. Clicking on a cell that was highlighted before will turn it green. Clicking on the other cells that were not highlighted will turn them red.
- In the footer, we’ll show the number of remaining correct guesses needed to win the game. Every new correct guess will decrement that number.
- The user loses the game if they click three wrong cells.
- The user wins the game if they guess all six cells correctly with no more than two wrong attempts.
- If the user wins the game, the message under the grid will read “Well Played.” If they lose, the message will read “Game Over.”
- When the game is over, it will display the active cells that the user could not guess.

Obviously, this is a lot more than what the client detailed for us, but it’s the minimum that makes sense to me. I assumed a few things while coming up with the specs, and the client may not like them. A logical thing to do after we have this list on paper is to make sure that the client likes everything about it.

If approved by the client, the list will be the initial contract between you and the client. We can go back to it for reference.

If you can’t come up with the whole list at once, that’s okay. Some people might even argue that a list like this up front is closer to a waterfall model than to a lean one. Look at this list as the MVP (minimal viable product) of the game, which is what the client wanted, but we made it detailed so that it can map to a list of tasks. This list is not constant; we might need to modify things, and we will certainly add more to it.

### Increment 3: Get a React app running locally with JSPM

Time for the exciting stuff: open your terminal and create a new directory for the game. Name it **memory-grid-game**.

Code Listing 81: *mkdir*

```
~ $ mkdir memory-grid-game
```



**Note:** I'll be assuming a Linux-based environment for all the commands in this example. On Microsoft Windows, the commands and outputs might be slightly different.

The JSPM library can be installed using Node Package Manager (NPM). We'll need an active Node.js installation to be able to install NPM libraries. The easiest way to install Node.js depends on what operating system you're using. On Mac and Linux, I've never had any issues with [Node Version Manager](#) (NVM).

For Microsoft Windows, there are some alternatives to NVM. There is [nvm-windows](#), and there is [nodist](#). You can also always just use the direct installer available [here](#).

Once you have Node installed, you'll also get NPM with it. You should be able to check the version with the `-v` argument. Here are the versions I am using right now:

Code Listing 82: *Node Version*

```
~ $ node -v
v6.3.1
~ $ npm -v
3.10.3
```

Go ahead and install JSPM now.

Code Listing 83: *JSPM Install*

```
~ $ npm install jspm -g
└─ jspm@0.16.41

~ $ cd memory-grid-game

~/memory-grid-game $ npm init
# Answer the questions

~/memory-grid-game $ npm install jspm --save-dev

~/memory-grid-game $ jspm init
Would you like jspm to prefix the jspm package.json properties under jspm?
[yes]:
Enter server baseURL (public folder path) [./]:
Enter jspm packages folder [./jspm_packages]:
Enter config file path [./config.js]:
Configuration file config.js doesn't exist, create it? [yes]:
Enter client baseURL (public folder URL) [/]:
Do you wish to use a transpiler? [yes]:
```

Which ES6 transpiler would you like to use, Babel, TypeScript or Traceur?  
[babel]:

I kept all the default answers as they were for the `jspm init` command.

At this point, JSPM will install all the system dependencies it needs and put them all under a `jspm_packages` directory. This includes Babel.js.



**Note:** You should track your increments with Git. If you use Git, don't forget to add both `node_modules` and `jspm_packages` to `.gitignore` at this point.

JSPM is ready. We can now start a local React app. We need an `index.html` file first. I'll use a global include of the latest **React** and **ReactDOM** here for simplicity, but you can also use JSPM itself to load a local copy of React using an `import` statement.

On the root level of the `memory-grid-game` directory, create an `index.html` file.

Code Listing 84: `index.html`—JSPM Template

```
<!doctype html>
<head>
  <script src="jspm_packages/system.js"></script>
  <script src="config.js"></script>
  <script src="https://fb.me/react-15.3.0.js"></script>
  <script src="https://fb.me/react-dom-15.3.0.js"></script>
</head>
<body>
  <div id="react">
    Loading...
  </div>
  <script>
    System.import('lib/main.js');
  </script>
</body>
</html>
```

The `div` with `id="react"` is where we're going to mount our React app.

Now in `lib/main.js`, import a **Container** component, which will be the main top-level component for our app, and then mount that component in the `div#react` element.

Code Listing 85: `lib/main.js`—Rendering the Container Component

```
import Container from "../Container";

ReactDOM.render(
  <Container />,
```



```
document.getElementById("react")
);
```



**Note:** In any *import* line, when we import from a local JavaScript file like “Container.js” in the previous example, we can omit the “.js” as JSPM assumes it’s the default.

Let’s create a “Hello React” line in the **Container** component to test everything.

*Code Listing 86: Lib/Container.js—Hello React*

```
class Container extends React.Component {
  render() {
    return <div>Hello React</div>;
  }
}

export default Container;
```

Go ahead and use your favorite command-line web server to serve the memory-grid-game directory over a web server. My favorite is the NPM package **serve**.

*Code Listing 87: Running a Local Server with serve*

```
~/memory-grid-game $ npm install serve -g
└─ serve@1.4.0
~/memory-grid-game $ serve
serving /Users/samer/memory-grid-game on port 3000
```



**Note:** At the time of this writing, JSPM used Babel 5.x, which did not require configuring any presets. Babel 6 is different. When JSPM supports Babel 6, you’ll potentially need to do a few different things. I’ll write a follow-up blog post about what you need to do differently for Babel 6 on JSPM. Look for it [here](#).

In a browser window, go to <http://localhost:3000/> now, and you should see “Hello React.”



*Figure 10: Hello React*

## Increment 4: Create an empty X by Y grid

I always start a React app with a **Container** component. You'll find examples where this component is named "App" or "Main." It's a good idea to have a top-level wrapper component because sometimes you need to control your actual starting component, and you can't do that from within.

It's now time to "think in React." What components do we need? Where should the state live? What actions do we need to plan a flow for in our components? There are no immediately right answers to these questions. Start with what makes sense to you at this moment, and iterate as you go.

Let's start with a **Game** component. The **Container** component will mount one instance of the **Game** component and pass it the rows and columns variables as props:

*Code Listing 88: Lib/Container.js—Container Class*

```
import Game from "../Game";

class Container extends React.Component {
  render() {
    return (
      <div>
        <Game rows={5} columns={5} />
      </div>
    );
  }
}

export default Container;
```

The **Game** component will have **this.props.rows** rows and **this.props.columns** columns, both of which have a value of 5.

Having these two variables passed down from **Container** gives us the power to later render another game with different grid dimensions.

Within the **Game** component, we'll need to draw a grid and show a message and some stats under that grid. We'll worry about the message and stats in a later increment.

We definitely need a **Cell** component to represent one cell in the grid. The **Cell** component is important, as it will have its own logic, style, and actions.

Should we create a **Row** component? We don't really have anything specific to a row in the grid, but a **Row** component will probably make the grid code read better. So we'll create one.

Should we create a **Grid** component to represent that section of the game? You'll probably be tempted to do that, but I think it will just introduce an unneeded level of complexity, so let's start by drawing rows and columns directly in the **Game** component itself.

Do we have to manage any state yet? To answer this question, let's answer this other question: Do we need React to re-render the DOM when something changes? If yes, that thing should be in a state somewhere. So far, however, nothing we defined should trigger a reload in the UI. Everything is an initial property to drive the initial UI.

We have a **Row** component and a **Cell** component. We need to render **this.props.rows** **Row** instances, and within each row, we need to render **this.props.columns** **Cell** instances.

This sounds like a nested loop. We can't do regular loops in JSX. We have two options:

- Prepare the grid in a variable with loops, then put that variable inside the returned JSX.
- Prepare a data matrix (rows × columns), then map that matrix into **Rows** and **Cells** components inside the returned JSX.

You might be tempted to do the first option, which I think is a bit imperative. The second option, though it might sound weird, is the more declarative way of rendering this grid, and it would make things easier going forward.

If we think ahead a bit, we need to eventually pick random cells in the grid, and we need to remember them. Having a data structure representing the grid cells and giving a unique **id** to every cell will certainly help us with these tasks.

Instead of preparing DOM nodes in the nested loops we identified, we're going to create a data matrix (which is just an array of arrays), give each cell in that matrix a unique **id**, then map the matrix arrays into **Rows** and **Cells** components. Here's what I came up with:

*Code Listing 89: **Lib/Row.js**—The Row Component*

```
class Row extends React.Component {
  render() {
    return (
      <div className="row">
        {this.props.children}
      </div>
    );
  }
}

export default Row;
```

*Code Listing 90: **Lib/Cell.js**—The Cell Component*

```
class Cell extends React.Component {
  render() {
    return (
      <div className="cell">
        {this.props.id}
      </div>
    );
  }
}
```

```

    });
  }
}

export default Cell;

```

Code Listing 91: *lib/Game.js*—The Game Component

```

import Row from "../Row";
import Cell from "../Cell";

class Game extends React.Component {
  render() {
    let matrix = [], row;
    for (let r = 0; r < this.props.rows; r++) {
      row = [];
      for (let c = 0; c < this.props.columns; c++) {
        row.push(`${r}${c}`);
      }
      matrix.push(row);
    }
    return (
      <div className="grid">
        {matrix.map((row, ri) => (
          <Row key={ri}>
            {row.map(cellId => <Cell key={cellId} id={cellId} />)}
          </Row>
        ))}
      </div>
    );
  }
}

export default Game;

```

I added these minimal styles to get the cells **divs** to look like a grid:

Code Listing 92: In *index.html head* Element—CSS Styles

```

...
<style>
  body {
    text-align: center;
  }
  .cell {
    width: 100px;
    height: 100px;
  }
</style>

```

```

    display: inline-block;
    border: 1px solid #aaa;
    background: #f8f8f8;
    margin-right: 4px;
  }
</style>
...

```

This is what I see when I refresh the browser now:

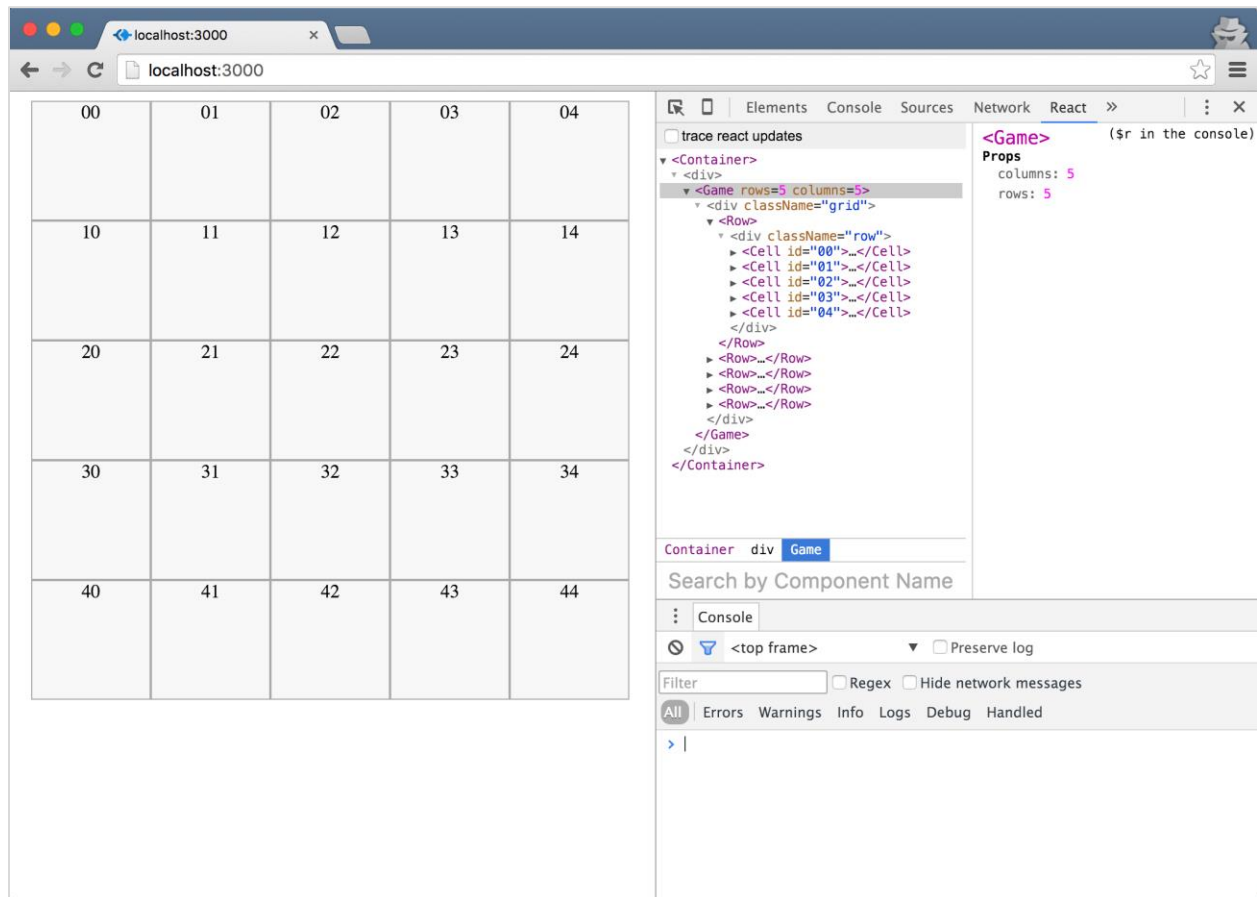


Figure 5: Empty Grid

Questions about the code so far:

- In the **Row** component, since this is just a wrapper around a number of cells, all we need to do is render all the children of the instance. We do this in React using **this.props.children**, which is a special structure that holds one or more elements. In this case, it has five **Cell** components. We also could have made a **Row** component responsible for rendering the five **Cell** components directly inside its render function. Which approach do you think is better? Why?

- We came up with a unique `id` for every cell by concatenating the row index with the column index. `ids` of the cells for our default grid will be strings like “00”, “01”, ... “44”. We’re showing the `ids` on the grid for testing, but they will not be part of the final product. We could have also just used a serial number for the `ids` here. Which approach do you think is better? Why?



**Note:** Did you identify any problems with the code so far? There is one problem in `Game.js`, which we will solve in Increment 6. Try to identify it.

## Increment 5: Get Ready... Memorize... Recall...

Before we get into the details of the grid, let’s prepare the three game states: “ready,” “memorize,” and “recall.”

The game starts with a “ready” state. After two seconds, we need to switch that to a “memorize” state, and after two more seconds, we need to switch that to a “recall” state. We can use the browser timers to accomplish this with `setTimeout`.

When the game state changes, we want to display a hint line in the UI of the game, right below the grid. Let’s put that logic in a **Footer** component. To keep this component pure, we’ll assume that it will receive the `gameState` as a prop, and we’ll use `defaultProps` to prepare an object of user-friendly hints to display to the user.

Code Listing 93: `Lib/Footer.js`—The Footer Component

```
class Footer extends React.Component {
  render() {
    return (
      <div className="footer">
        <div className="hint">
          {this.props.hints[this.props.gameState]}...
        </div>
      </div>
    );
  }
}

Footer.defaultProps = {
  hints: {
    ready: "Get Ready",
    memorize: "Memorize",
    recall: "Recall"
  }
}

export default Footer;
```

We use **defaultProps** to plan for the future; instead of hard-coding the **hints** object in the component instance, we'll have a way to use a **Footer** component instance with different hints if we need to. For example, when it's time to render this game in a different language, we have a higher level of control over these hints.

The **Game** component instance will render a **<Footer />**, passing in a **gameState** prop.

The **gameState** variable is not the best example of whether something should go into the component state or not, because I picked a name that's already hinting at it. But the same question we asked before applies here: Do we need React to re-render the UI when this **gameState** variable changes? Absolutely.

Every time the **gameState** changes, we want to re-render the mounted **Footer** component instance to update the hint line. Putting the **gameState** variable on the state of the **Game** component instance will make React do that for us.

To go from “ready” to “memorize” to “recall,” we can use two **setTimeout** calls, which should start when we mount the **Game** component in the browser.

*Code Listing 94: In **Lib/Game.js**—Adding Footer with Timers*

```
...
import Footer from "../Footer";

class Game extends React.Component {
  constructor(props) {
    super(props);
    this.state = { gameState: 'ready' };
  }
  componentDidMount() {
    setTimeout(() => this.setState({ gameState: 'memorize' }), 2000);
    setTimeout(() => this.setState({ gameState: 'recall' }), 4000);
  }
  render() { ... }
}
...
```

To complete the feature, the **Footer** component instance needs access to the **gameState** as a prop. When I am creating components for readability and organization, I usually pass the whole state object down to them so that I don't have to worry about items I add later to the state.

In **Game's render()**, right under the grid:

*Code Listing 95: Within **Lib/Game.js** **render()** Function—Spreading the **state** Object for **Footer***

```
...
// The Matrix Map
<Footer {...this.state} />
...
```

The three-dots spread operator will take **this.state** and spread all of its keys as props for the **Footer** component.

When we refresh the browser now, we should see a “Get Ready...” hint. After two seconds, it should switch to “Memorize...”, and after two more seconds, it should switch to “Recall...”.

The timers code in **componentDidMount** is okay, but it could be better. The “recall” state does not depend on the “memorize” state at all, which means if the first line fails, the second one might still run. That’s not ideal and could lead to problems down the line.

We should only go to a “recall” state once we are in a “memorize” state. **setState** does not guarantee its operation to be synchronous, but it does provide a second argument optional callback. The function we pass as a callback will be executed once the **setState** operation is complete.

So we can do the following:

*Code Listing 96: In **Lib/Game.js**—**setState** Callback*

```
componentDidMount() {
  setTimeout(() => {
    this.setState({ gameState: 'memorize' }, () => {
      setTimeout(() => this.setState({ gameState: 'recall' }), 2000);
    });
  }, 2000);
}
```

The timer to set **gameState** as “recall” will now only be invoked if the first **setState** is complete and the game is in the “memorize” state.

## Increment 6: Highlight Z random cells on the grid during the memorize state

Have you identified the problem I mentioned in Increment 4?

We computed the data matrix in the **render** method, which means that every time React computes the component’s virtual DOM, it will create a new matrix object. If you put a **console.log** line in the render, you’ll see that it is called three times with **gameState** changes, and each time we’re creating a new matrix object.



**Note:** Putting *console.log* lines in the *render()* function can often be used to uncover unexpected problems. Don’t be alarmed with how many times React invokes the *render()* function—React will only take to the real DOM what needs to change.

This is avoidable simply by moving the matrix computation into the component’s constructor function instead. The constructor function is executed only once when React creates the component instance out of the component class.



However, since we need to access the data matrix in the **render** function, we'll need to either put it on the state (**this.state.matrix**) or use an instance variable (**this.matrix**). Both approaches are valid, but we should put on the state only what requires a React DOM refresh.

The data matrix isn't changing once we mount the component: We're only using it to give the cells unique ids and declaratively mapping the structure into **Rows** and **Cells**. Using an instance variable here should suffice.

Here's the exact matrix we have for our 5 × 5 grid:

*Code Listing 97: The Data Matrix for the 5 × 5 Grid*

```
[
  ["00", "01", "02", "03", "04"],
  ["10", "11", "12", "13", "14"],
  ["20", "21", "22", "23", "24"],
  ["30", "31", "32", "33", "34"],
  ["40", "41", "42", "43", "44"]
]
```

We want to pick Z random cells from that matrix. We'll pass this Z variable into the **Game** component instance as a prop. Let's call it **activeCellsCount** and give it a default value of 6. We need to update the **Container** component source so that the **activeCellsCount** is passed from there.

*Code Listing 98: In Lib/Container.js render() Function—Adding activeCellsCount*

```
...
<div>
  <Game rows={5} columns={5} activeCellsCount={6} />
</div>
...
```

An easy way to pick random elements from an array is to use a library like Underscore or Lodash.

Lodash has a **sampleSize** function which can do that, but what we have here is an array of arrays, not just one array, so we need to *flatten* it first. Luckily, we can also use Lodash to flatten our array of arrays.

Here's the update code for the **Game constructor** function:

*Code Listing 99: In Lib/Game.js—Picking Random Values Using Lodash*

```
import _ from "lodash";

...
constructor(props) {
  super(props);
```

```

    this.matrix = [];
    for (let r = 0; r < this.props.rows; r++) {
      let row = [];
      for (let c = 0; c < this.props.columns; c++) {
        row.push(`${r}${c}`);
      }
      this.matrix.push(row);
    }

    let flatMatrix = _.flatten(this.matrix);
    this.activeCells = _.sampleSize(flatMatrix,
                                    this.props.activeCellsCount);

    this.state = {
      gameState: "ready"
    };
  }
  ...

```

Since our app depends on **lodash** now, we need to install it.

*Code Listing 100: Installing Lodash*

```
memory-grid-game $ jspm install lodash
```

The **flatMatrix** variable will now be:

*Code Listing 101: flatMatrix*

```

["00", "01", "02", "03", "04",
 "10", "11", "12", "13", "14",
 "20", "21", "22", "23", "24",
 "30", "31", "32", "33", "34",
 "40", "41", "42", "43", "44"]

```

The **activeCells** variable would be an array of six random ids chosen from **flatMatrix**.

Once again, I opted not to put **activeCells** on the component state and just use an instance variable instead. You might be tempted to put it on the state given that these cells need to show up on the grid during the “memorize” state (and that sounds like a UI re-render). But if you think carefully about this, all we need to do is have the **activeCells** in memory once, then we’ll use the **gameState** to determine if they should show up or not. We compute the **activeCells** at the initialization of the component instance instead of updating the state with **activeCells** when we move from “ready” to “memorize” **gameState**.

Just like we passed the state object to the **Footer** component, we’ll pass it to the **Cell** components. We’ll also pass the **activeCells** array to **Cell**.

Code Listing 102: In *Lib/Game.js*—Mapping the Matrix to Rows and Cells

```
render() {
  return (
    <div className="grid">
      {this.matrix.map((row, ri) => (
        <Row key={ri}>
          {row.map(cellId => <Cell key={cellId} id={cellId}
                                activeCells={this.activeCells}
                                {...this.state} />)}
        </Row>
      ))}
      <Footer {...this.state} />
    </div>
  );
}
```

The goal here is to have the **Cell** component completely in control of its different UI states, but through props coming from the parent. This might not be the most efficient approach, but I think it makes the code more readable.

To further simplify things in the **Cell** component, I'll just use different CSS classes to represent different states of a **Cell**.

Here's what I came up with for the **Cell** component:

Code Listing 103: *Lib/Cell.js*—The Cell Component

```
class Cell extends React.Component {
  active() {
    return this.props.activeCells.indexOf(this.props.id) >= 0;
  }
  render() {
    let className = "cell";
    if (this.props.gameState === "memorize" && this.active()) {
      className += " active";
    }

    return (
      <div className={className}>
        </div>
    );
  }
}

export default Cell;
```

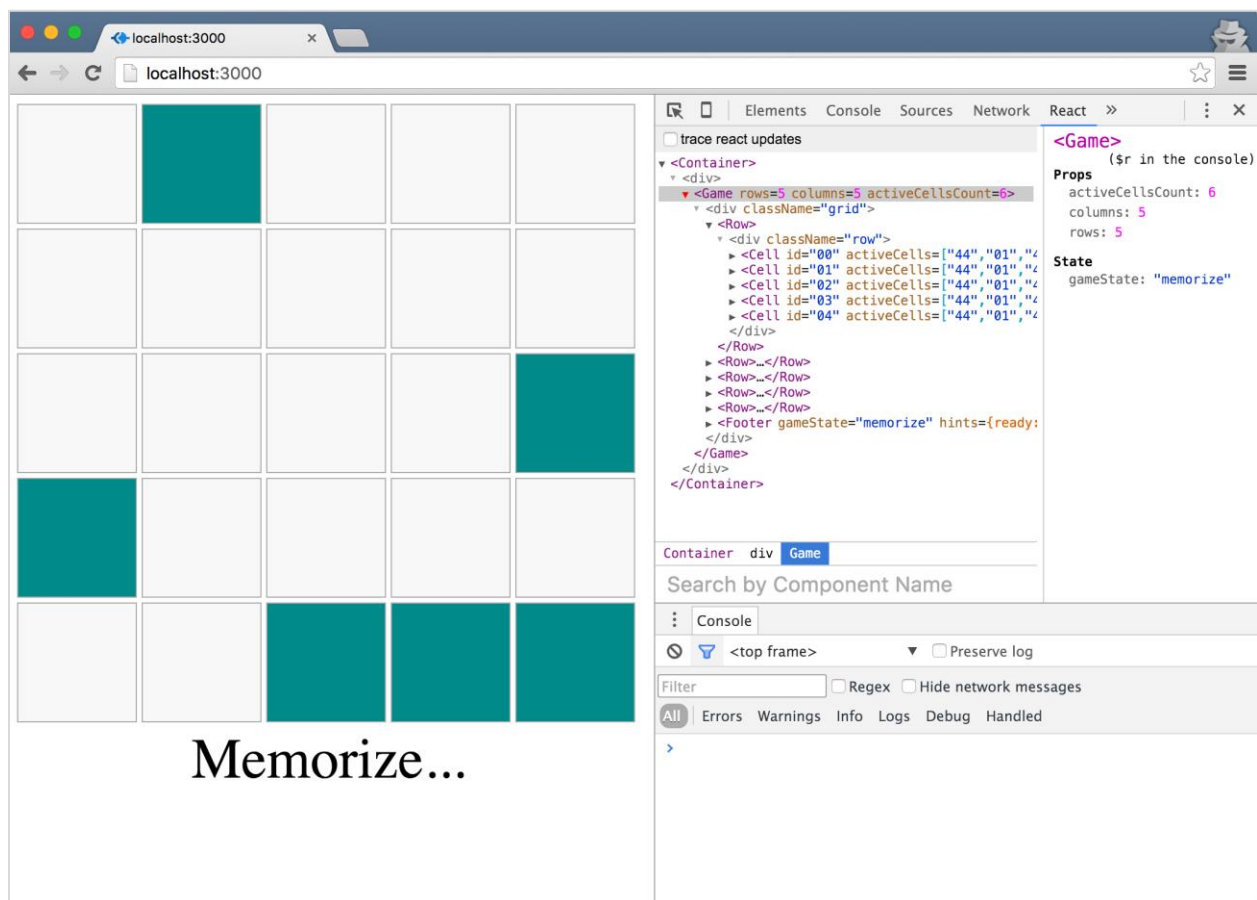
A cell is active if its **id** is part of the **activeCells** array, and we only want the active cells to show up on the grid during the “memorize” **gameState**.

Let's give the active cells a different color.

*Code Listing 104: In **index.html** **style** Element—The Active Cell Color*

```
.active {  
  background-color: #058BDA;  
}
```

If we refresh the browser now, during the “memorize” state, we should see six randomly highlighted cells on the grid.



*Figure 6: Active Cells on the Grid*

## Increment 7: Click a cell to guess during the recall state

Here's the pre-analysis:

- A **Cell** component should accept a click event only during a “recall” **gameState**.

- The click handler in a **Cell** component can compute whether this click is a correct guess or not since it has access to all the data it needs to do so.
- We need the UI to change while guessing. Correct guesses should be marked green, and wrong ones red. This means something has to be stored in a state somewhere when we click.
- We want to keep the state on the **Game** component level, so the click handler in a **Cell** component would need to invoke a function on the **Game** component to mutate the state there.

What should go in the state? This is a challenging question, and the answer will greatly shape the rest of the code for this game. Keep in mind that we only put on the state the variables for which we want React to trigger a reload of the UI when they change.

We want the UI to be updated when there is a new correct guess, and when there is a new wrong guess. Let's maintain both pieces of data with arrays on the state.

*Code Listing 105: In **Lib/Game.js**—Initial Game State*

```
...
this.state = {
  gameState: "ready",
  wrongGuesses: [],
  correctGuesses: []
};
...
```

This way, we'll know that if we push new values to either of the arrays, React will refresh the DOM in the browser.

Let's now create a click handler in the **Cell** component. To invoke the handler, we add an **onClick={this.handleClick.bind(this)}** on the cell's **div** and define **handleClick** to call a top-level function. Let's name the top-level function **recordGuess**.

*Code Listing 106: In **Lib/Cell.js**—**handleClick***

```
...
handleClick() {
  if (this.props.gameState === "recall") {
    this.props.recordGuess({
      cellId: this.props.id,
      userGuessIsCorrect: this.active()
    });
  }
}
...
render() {
  ...
}
```

```

    return (
      <div className={className} onClick={this.handleClick.bind(this)}>
        </div>
    );
  }
  ...

```

We want to record a guess only when the game is in the “recall” state. That’s why we have an **if** statement in **handleClick**.

The **recordGuess** function in the **Game** component needs access to the cells being clicked and whether the user’s guess is correct. From the point of view of a **Cell** component, the value returned from the **active()** function is what makes the guess correct or wrong.

**recordGuess** needs to be passed to **<Cell />** as a prop. Add **recordGuess={this.recordGuess.bind(this)}** in the **<Cell />** line in **Game.js**.

**recordGuess** will need to update the state and push the **id** of the guessed cell to either **correctGuesses** or **wrongGuesses**.

*Code Listing 107: In **Lib/Game.js**—The **recordGuess()** Function*

```

...
recordGuess({ cellId, userGuessIsCorrect }) {
  let { wrongGuesses, correctGuesses } = this.state;
  if (userGuessIsCorrect) {
    correctGuesses.push(cellId);
  } else {
    wrongGuesses.push(cellId);
  }
  this.setState({ correctGuesses, wrongGuesses });
}
...

```

This completes the recording of a guess, but we need to reflect those guesses in the UI.

In the **Cell** component, if the cell’s **id** is in the state’s **correctGuesses** or **wrongGuesses**, we want to give it different CSS classes so that we can mark it green or red.

Let’s create a function to return either true or false about whether the cell’s **id** is in **correctGuesses** or **wrongGuesses**.

*Code Listing 108: In **Lib/Cell.js**—The **Guess State***

```

...
guessState() {
  if (this.props.correctGuesses.indexOf(this.props.id) >= 0) {

```

```

    return true;
  } else if (this.props.wrongGuesses.indexOf(this.props.id) >= 0) {
    return false;
  }
}
...

```

Both **correctGuesses** and **wrongGuesses** are available to a `<Cell />` instance as props since we spread out the full **Game** state keys as props on the cells and the footer.

So now all we need to do is add CSS classes to every cell based on its **guessState()**.

The `Cell.js` **render** function would now look like:

*Code Listing 109: In `Lib/Cell.js`—The `render()` Function*

```

...
render() {
  let className = "cell";
  if (this.props.gameState === "memorize" && this.active()) {
    className += " active";
  }
  className += " guess-" + this.guessState();

  return (
    <div className={className} onClick={this.handleClick.bind(this)}>
      </div>
  );
}
...

```

And we need the following CSS style to complete this feature:

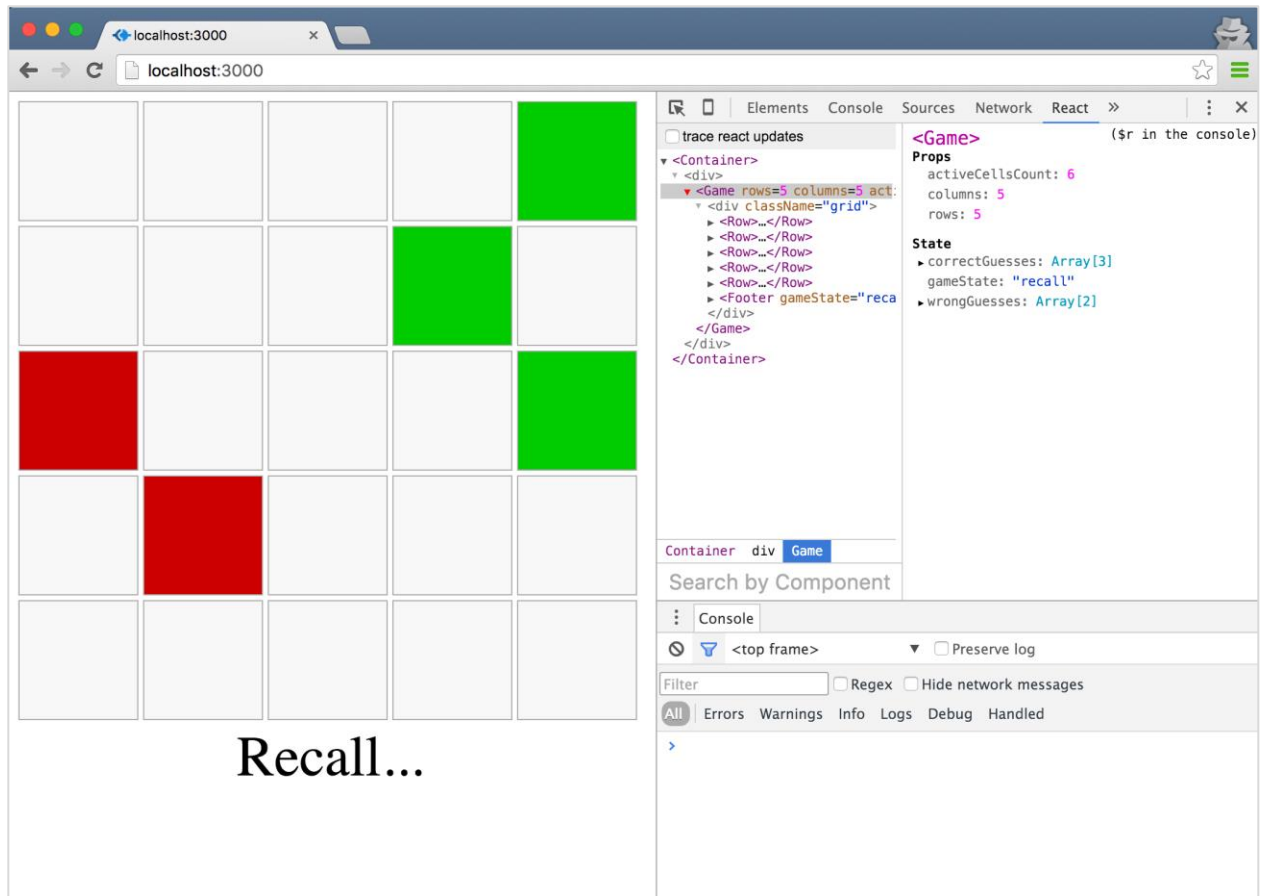
*Code Listing 110: In `index.html style` Element—Red and Green CSS Styles*

```

.guess-true {
  background-color: #00CC00;
}
.guess-false {
  background-color: #CC0000
}

```

If you test now, during the “recall” state, you can click to guess. Correct guesses should be marked green, and wrong guesses should be marked red:



Recall...

Figure 7: Guessing Correctly and Incorrectly

The code we have so far has a small bug. Try to identify it.

What happens if you click on an already-guessed cell?

Our code will gladly push the same cell **id** one more time to the suitable array. This might sound harmless at first, but it will potentially introduce problems down the line.

The solution is simple: in the **if** statement of the **handleClick** function, don't call **recordGuess** for a cell that returns any value (true or false) from its **guessState()** function.

Code Listing 111: In **lib/Cell.js**—Don't Guess an Already Guessed Cell

```
...
handleClick() {
  if (this.guessState() === undefined &&
      this.props.gameState === "recall") { ... }
```



## Increment 8: Show the number of remaining guesses needed to win the game

This one is easy; all we need to do is display the computed count. The **Footer** component already has access to the total number of current **correctGuesses** from the state, but it needs access to the **activeCellsCount** prop as well.

*Code Listing 112: In **Lib/Game.js**—Footer Props*

```
...
<Footer {...this.state}
        activeCellsCount={this.props.activeCellsCount} />
...
```

In the component's **render** function, we'll display the result of a **remainingCount()** function. This function will return null (to display nothing) when the state of the game is not "recall," and it will otherwise compute the number of correct guesses needed to finish the game.

*Code Listing 113: In **Lib/Footer.js**—**remainingCount()***

```
class Footer extends React.Component {
  remainingCount() {
    if (this.props.gameState !== "recall") { return null; }
    return (
      <div className="remaining-count">
        {this.props.activeCellsCount - this.props.correctGuesses.length}
      </div>
    );
  }
  render() {
    return (
      <div className="footer">
        <div className="hint">
          {this.props.hints[this.props.gameState]}...
        </div>
        {this.remainingCount()}
      </div>
    );
  }
}
```

If you test now, guessing correctly will decrement the counter below the hint line.

## Increment 9: Game over after three wrong guesses, or win after six correct guesses

We can compute the lost/won game state from the `wrongGuesses` and `correctGuesses` arrays:

- The game is “won” if we have the original `activeCellsCount` in `correctGuesses`.
- The game is “lost” if we have three items in `wrongGuesses`.

Should the won/lost variable be part of the `Game`’s component state—just like we did with “read,” “memorize,” and “recall”—or should it be computed?

The won/lost variable can be computed. It does not need to go into the component state. However, since we have a `gameState` structure already in place, I’d argue that making the won/lost variable part of that structure will make the code simpler and more readable.

All we need to do in the footer is add the new state keys and their hint message.

*Code Listing 114: In `Lib/Footer.js`—Footer Hints*

```
...
Footer.defaultProps = {
  hints: {
    ready: "Get Ready",
    memorize: "Memorize",
    recall: "Recall",
    won: "Well Played",
    lost: "Game Over"
  }
};
...
```

To set `gameState` to either “won” or “lost,” we add this computing `if` statement in `recordGuess`.

*Code Listing 115: In `Lib/Game.js`—Updating the State for a Guess*

```
...
recordGuess({ cellId, userGuessIsCorrect }) {
  let { wrongGuesses, correctGuesses, gameState } = this.state;
  if (userGuessIsCorrect) {
    correctGuesses.push(cellId);
    if (correctGuesses.length === this.props.activeCellsCount) {
      gameState = "won";
    }
  } else {
    wrongGuesses.push(cellId);
    if (wrongGuesses.length > this.props.allowedWrongAttempts) {
      gameState = "lost";
    }
  }
}
```

```

    }
  }
  this.setState({ correctGuesses, wrongGuesses, gameState });
}
...

```

Note how I used an **allowedWrongAttempts** prop on `<Game />` instead of hardcoding a value of “2” in there. With this variable, we can later spin another game with a different **allowedWrongAttempts** value. Since we don’t plan on doing that yet, we can use React components’ **defaultProps** to use a default value for this prop.

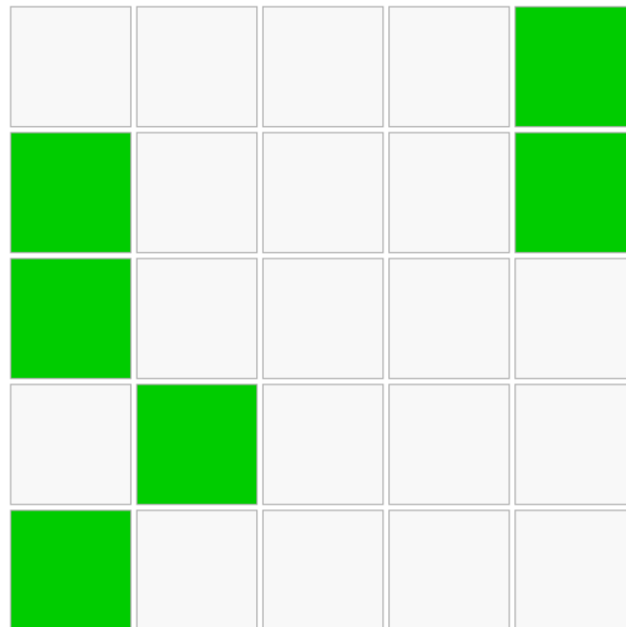
*Code Listing 116: In `Lib/Game.js`—`allowedWrongAttempts`*

```

...
Game.defaultProps = {
  allowedWrongAttempts: 2
};
...

```

Go ahead and test this feature now. You should be able to win or lose the game.



Well Played...

*Figure 8: Game “Won” State*

## Increment 10: Show the original active cells when game is over

Looking at the **if** statement controlling whether an **activeCell** should be displayed or not, it checks if the **gameState** is “memorize” at this point. All other states will hide **activeCells**, which has made sense so far.

For this feature, we need to also show the **activeCells** if the **gameState** becomes “lost.” Instead of making a longer **if** statement, and for better readability, let’s extract the state-checking logic into its own method.

*Code Listing 117: In **Lib/Cell.js**—**showActiveCells()***

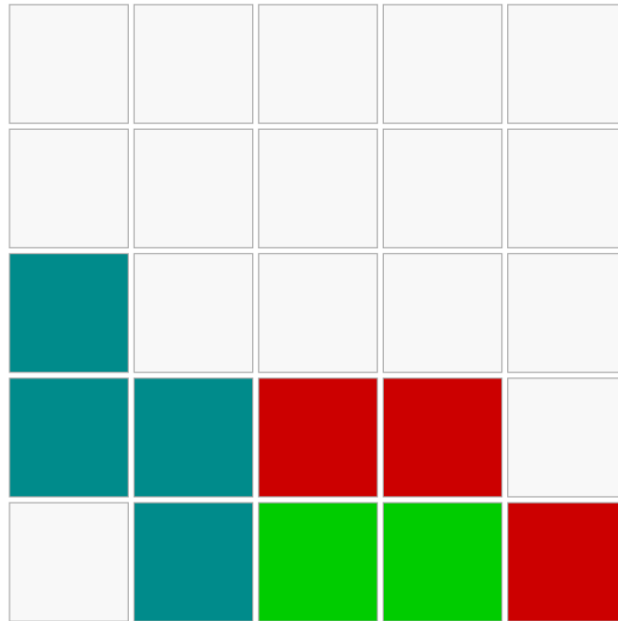
```
...
showActiveCells() {
  return ["memorize", "lost"].indexOf(this.props.gameState) >= 0;
}
...
```

The **if** statement inside the **render** function now becomes:

*Code Listing 118: In **Lib/Cell.js**—Inside the **render()** Function*

```
...
if (this.showActiveCells() && this.active()) {
  className += " active";
}
...
```

This is much more readable. The feature will now work as requested.



## Game Over...

Figure 9: Game “Lost” State

Just because it works, though, does not mean we’re done.

Imagine yourself in a technical interview now, and I ask you out of nowhere: “What’s wrong with this **showActiveCells** code? Can we do better?”

If you can’t identify a problem, I’ll give you a hint. Put a **console.log** line inside **showActiveCells** and observe how many times it gets fired.

For our 5 × 5 grid example, on every **gameState** change, we call this function 25 times. More importantly, on every cell click, we call this function 25 times. Why?

Every time the **Game** state changes, React will recompute its **render** function, which means recomputing the **render** function of 25 children cells, which means invoking **showActiveCells** 25 times. This is not a huge deal for the small codebase we’re working with here, but we can easily eliminate this problem.

When the state for our **<Game />** instance changes, we need to compute the **showActiveCells** only once. This function has no dependency on the cell-specific data, so we don’t need this function to be on the **Cell** level. We can move it up to the **Game** level and pass it as a prop to all cells.

The **Game render** function becomes:

Code Listing 119: In *Lib/Game.js*—**showActiveCells** in **Game’s render()**

```
render() {  
  let showActiveCells =
```

```

    ["memorize", "lost"].indexOf(this.state.gameState) >= 0;
  return (
    <div className="grid">
      {this.matrix.map((row, ri) => (
        <Row key={ri}>
          {row.map(cellId =>
            <Cell key={cellId} id={cellId}
              showActiveCells={showActiveCells}
              activeCells={this.activeCells}
              recordGuess={this.recordGuess.bind(this)}
              {...this.state} />)}
          </Row>
        )}}
      <Footer {...this.state}
        activeCellsCount={this.props.activeCellsCount} />
    </div>
  );
}

```

In the **Cell** render function, we now use **this.props.showActiveCells** instead of **this.showActiveCells()**.

The **indexOf()** line is now called just once instead of 25 times per state change.

This was an easy problem to spot because we refactored our original **if** statement into a function and realized that the function does not depend on the **Cell** instances at all. Whenever you render a component multiple times in a loop, like our **<Cell />** example here, be careful about any computation you make in it. Always ask the question, “does this computation need to go on that level or can we just pass it from the parent?”

With this increment, we have officially reached MVP status. We can proudly take the product to the client now and have them test it for feedback.

The moment of truth about the design of our code happens when the client comes back to us with enhancements or bugs. If we made good calls early on about the structure of our components and state, extending and maintaining the application will be easier.

Of course, coming up with good design decisions early on is purely a matter of experience. The more applications you build with React, the better you’ll get at making these decisions.

Let’s assume our client came back to us with two extra features that they want done before shipping this game.

Here’s what they requested:

- Give the players 10 seconds to play the game, and “game over” if they don’t finish in time.

- When a game is won or lost, show a “play again” button.

Go ahead and try to implement these two features on your own first, and then see my solution in the next section once you’re done.

## Increment 11: Give the players a 10-second window to play the game

First, let’s not use the number 10 directly in code because this might need to change, and possibly different levels would require different timeout seconds.

Let’s put it as a default prop on the **Game** component for now.

*Code Listing 120: In `Lib/Game.js`—`timeoutSeconds`*

```
...
Game.defaultProps = {
  allowedWrongAttempts: 2,
  timeoutSeconds: 10
};
...
```

Two things are clear about this 10-second timeout feature:

- During the “recall” **gameState**, we need to decrement the seconds-remaining variable every second. We can do that with a **setInterval** function.
- When the seconds-remaining variable hits a 0 value, we need to update **gameState** to “lost.”

Did you implement this feature by adding a new variable to the game state?

You might be tempted to put a **secondsRemaining** variable on the state, but since the feature didn’t instruct us to show the remaining seconds in the game UI, we don’t really need to make this variable part of the game state. If we do, we’ll be introducing unnecessary renders.

The only UI update needed here is when **secondsRemaining** hits a 0 value. This update will happen anyway because the **gameState** variable will change at that point.

If you need to hold a variable that is specific to a component instance outside of its official React-supported state, you can use an instance variable: **this.secondsRemaining**.

If you suspect the client will come back and tell you to display the seconds remaining in the UI, you can go ahead and use a state variable. It’s important to understand the difference.

The current code starts the “recall” **gameState** using a timer, inside another timer, in **componentDidMount**.

Since we’re going to be adding more logic when switching the game to the “recall” state, let’s introduce a **startRecallMode** function and use that in the timer.

Code Listing 121: In *Lib/Game.js*—Calling a *startRecallMode()* Function

```
...
componentDidMount() {
  setTimeout(() => {
    this.setState({ gameState: 'memorize' }, () => {
      setTimeout(this.startRecallMode.bind(this), 2000);
    });
  }, 2000);
}
...
```

**startRecallMode** will update **gameState** to “recall.” When that update is complete, it should initialize the **secondsRemaining** variable and kick off a timer to decrement it using **setInterval**. When **this.secondsRemaining** hits 0, it will update the **gameState** to “lost.”

Code Listing 122: In *Lib/Game.js*—The *startRecallMode()* Function

```
...
startRecallMode() {
  this.setState({ gameState: 'recall' }, () => {
    this.secondsRemaining = this.props.timeoutSeconds;
    setInterval(() => {
      if (--this.secondsRemaining === 0) {
        this.setState({ gameState: "lost" });
      }
    }, 1000);
  });
}
...
```

This should work now. Start the game and wait 10 seconds during the “recall” state, and you should see the “Game Over” hint line.

Easy. Right?

**There are at least three big problems with the code so far. Try to identify them.**

**Problem 1:** The game will be lost after 10 seconds, even if you win it during those 10 seconds.

**Solution:** Stop the interval timer when the user wins the game. You should also stop it when the user loses the game via the three wrong attempts, as it’s also not needed after that.

**Problem 2:** If you let the game expire, that interval timer will continue running, and **this.secondsRemaining** will go negative. React will continue to re-render the **Game** instance every second because of the state update.

**Solution:** Stop the interval timer inside its **if** statement.



These two problems are easy. We just need to put the interval timer `id` in an instance variable and use the `clearInterval` function on that when needed. Since we're doing the same task in multiple places, it's probably a good idea to do the `clearInterval` call in a function. Let's call it `finishGame`.

*Code Listing 123: In `Lib/Game.js`—`finishGame()`*

```
...
finishGame(gameState) {
  clearInterval(this.playTimerId);
  return gameState;
}
...
```

`playTimerId` is what we'll name the interval timer id. I also passed `gameState` to this function because I'll be invoking the function for both "won" and "lost" `if` statements in `recordGuess`, and it makes the code cleaner.

*Code Listing 124: In `Lib/Game.js`—Using `this.finishGame()`*

```
...
recordGuess({ cellId, userGuessIsCorrect }) {
  let { wrongGuesses, correctGuesses, gameState } = this.state;
  if (userGuessIsCorrect) {
    correctGuesses.push(cellId);
    if (correctGuesses.length === this.props.activeCellsCount) {
      gameState = this.finishGame("won");
    }
  } else {
    wrongGuesses.push(cellId);
    if (wrongGuesses.length > this.props.allowedWrongAttempts) {
      gameState = this.finishGame("lost");
    }
  }
  this.setState({ correctGuesses, wrongGuesses, gameState });
}
...
```

The timer code inside `startRecallMode()` becomes:

*Code Listing 125: In `Lib/Game.js`—Using `this.finishGame()` in the Interval Code*

```
...
this.playTimerId = setInterval(() => {
  if (--this.secondsRemaining === 0) {
    this.setState({ gameState: this.finishGame("lost") });
  }
}, 1000);
```

...

**Problem 3:** What happens if we unmount the **Game** component during the recall mode?

In fact, it's time to uncover another hidden bug: What happens now if we unmount the component during the "ready" state, and during the "memorize" state? Go ahead and try that.

To unmount our game component, in the dev tools JavaScript console, invoke the `ReactDOM.unmountComponentAtNode` function.

*Code Listing 126: Unmounting the Game*

```
ReactDOM.unmountComponentAtNode(document.getElementById("react"));
```

You will see something like this:

```
> ReactDOM.unmountComponentAtNode(document.getElementById("react"));
< true
✖ ▶ Warning: 12624079 897774290317920 1379776191 n.js:18794
  setState(...): Can only update a mounted or mounting component.
  This usually means you called setState() on an unmounted
  component. This is a no-op. Please check the code for the
  undefined component.
>
```

*Figure 10: Updating State on Unmounted Components*

Any idea why is this happening?

We used two timers in `componentDidMount` that each set the state, and now we're adding a third timer that will set the state after 10 seconds. When you unmount the `<Game />` instance before these timers get invoked, they'll try to set the state on an unmounted component.

**Solution:** Every time you create a timer in a component, give it an `id` and clear that timer using the `componentWillUnmount` lifecycle hook.

*Code Listing 127: In `Lib/Game.js`—Clearing the Timers*

```
componentDidMount() {
  this.memorizeTimerId = setTimeout(() => {
    this.setState({ gameState: 'memorize' }, () => {
      this.recallTimerId = setTimeout(
        this.startRecallMode.bind(this),
        2000
      );
    });
  });
}
```

```

    }, 2000);
  }
  componentWillUnmount() {
    clearTimeout(this.memorizeTimerId);
    clearTimeout(this.recallTimerId);
    this.finishGame();
  }
}

```

Try the previous unmount test now and make sure you're no longer seeing the warning.

## Increment 12: Add a “Play Again” button

The “Play Again” button will go in the **Footer** component, and we only want to display it when the **gameState** is either “won” or “lost.”

When there is a case like that, I usually create a function that returns the needed DOM only when the condition is met, and I use that function inside the render function.

*Code Listing 128: In **Lib/Footer.js**—The **playAgainButton()***

```

...
playAgainButton() {
  if (["won", "lost"].indexOf(this.props.gameState) >= 0) {
    return (
      <button className="play-again-button"
        onClick={this.props.playAgain}>
        Play Again
      </button>
    );
  }
}
render() {
  return (
    <div className="footer">
      <div className="hint">
        {this.props.hints[this.props.gameState]}...
      </div>
      {this.remainingCount()}
      {this.playAgainButton()}
    </div>
  );
}
...

```

The action to “reset” the game will not be part of the **Footer** component, so we'll pass that action to **<Footer />** as a prop. This is why the “Play Again” button's **onClick** value is using a prop.

Let's think about the **playAgain** action. You can implement this feature in multiple ways. You might be tempted to reset the state of the **Game** component, something like:

*Code Listing 129: `resetGame()`*

```
resetGame() {
  this.setState({
    gameState: "ready",
    wrongGuesses: [],
    correctGuesses: []
  }, () => {
    // invoke the timers to change gameState
  });
}
```

This is certainly a valid approach, but wouldn't it be easier if we trashed the current `<Game />` instance that we have in the DOM and mounted a new one in its place? This way, we don't have to have the **resetGame** logic.

This latter approach, although it might sound like cheating, is actually very powerful. Imagine if the client came back to you with another feature where they wanted the "level" of the game to increase every time the user hits play again. Since the configurations for number of cells, allowed wrong attempts, and timeout seconds are all passed to `<Game />` as props, if we renew the `<Game />` instance rendered, we can just pass it a new set of props.

However, using a direct unmount command for the browser feels a bit imperative and does not fit well with the React way.

If you give a mounted component instance a **key** attribute (like the ones needed for looped-over components), React exclusively uses the value of **key** to "identify" the instance. This means that if the value of **key** changes, React sees a complete new instance there. We can leverage this concept to change our mounted `<Game />` instance. All we need to do is give it a key and then change the value for that key to "play again."

Of course, we can't change the key of a mounted `<Game />` instance from within that instance, so this action has to happen inside the container component.

*Code Listing 130: `Lib/Container.js`—Generating a New Game*

```
import Game from "../Game";

class Container extends React.Component {
  constructor(props) {
    super(props);
    this.state = { gameId: 1 };
  }
  createNewGame() {
    this.setState({ gameId: this.state.gameId + 1 });
  }
}
```

```

render() {
  return (
    <div>
      <Game key={this.state.gameId}
        createNewGame={this.createNewGame.bind(this)}
        rows={5} columns={5}
        activeCellsCount={6} />
    </div>
  );
}
}

export default Container;

```

By putting a **gameId** on the state and using it as the **key** of the game, we can now declaratively update the **gameId** to create a new game.

Since we need this action to happen from the **Footer** component, we'll pass it as a prop for **<Game />** and make **<Game />** pass it to **<Footer />**.

*Code Listing 131: In **Lib/Game.js**—Footer's Props*

```

<Footer {...this.state}
  playAgain={this.props.createNewGame}
  activeCellsCount={this.props.activeCellsCount} />

```

When the user clicks the “Play Again” button now, here’s what happens:

- **<Footer />** will tell **<Game />** to invoke the function it has under its **createNewGame** prop.
- **<Game />** will tell **<Container />** to invoke the function it has under its **createNewGame** prop.
- **<Container />** will invoke its **createNewGame** function, which will update the state of the component instance.
- React will re-render the **<Container />** instance, and it'll see a new key for **<Game />**.
- React will unmount the **<Game />** it has in the DOM, and it will mount a new instance with the new key.

## Challenges

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 personal challenge on this game:

### 1. Track scores:

- A perfect score of 3 happens when the user 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 user finishes the game in the first five seconds, double their score.
- Display the total score in the UI somewhere.
- When the user plays the game again, add the new score to the total score.

### 2. Make it harder:

- When the user clicks “Play Again,” make the grid bigger: 6 × 6, then 7 × 7, and so on.
- Increment the **activeCells** with each new game.

Have fun!