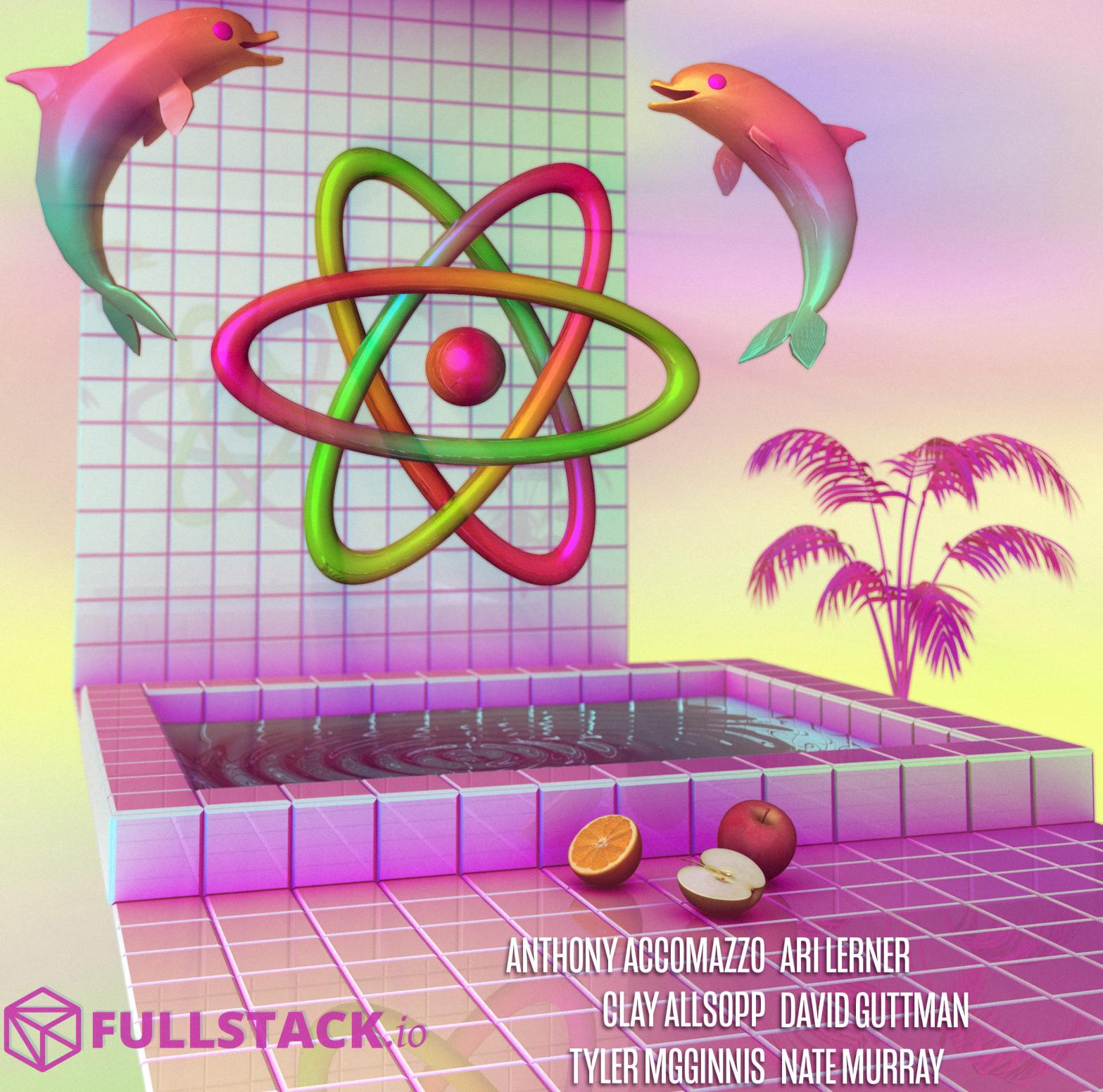




FULLSTACK REACT

The Complete Guide to ReactJS and Friends



FULLSTACK.io

ANTHONY ACCOMAZZO ARI LERNER
CLAY ALLSOPP DAVID GUTTMAN
TYLER MCGINNIS NATE MURRAY

Fullstack React

The Complete Book on ReactJS and Friends

Anthony Accomazzo, Ari Lerner, David Guttman, Nate Murray,
Clay Allsopp and Tyler McGinnis

© 2015 - 2016 Fullstack.io

Contents

Book Revision	1
Prerelease	1
Bug Reports	1
Chat With The Community!	1
Be notified of updates via Twitter	1
We'd love to hear from you!	1
Your first React Web Application	1
Building Product Hunt	1
Getting started	1
Sample Code	1
Code editor	1
Node.js and NPM	1
Browser	2
Previewing the application	2
Prepare the app	5
Our first component	8
React.createClass()	10
JSX	11
The developer console	12
Babel	15
ReactDOM.render()	17
Our second component	18
Making Product data-driven	22
The data model	22
Using props	22
Rendering multiple products	26
React the vote (your app's first interaction)	31
Propagating the event	31
Using state	36
Setting state with this.setState()	37
Updating state	39
Congratulations!	41

CONTENTS

Components	42
A time-logging app	42
Getting started	44
Previewing the app	44
Prepare the app	44
Breaking the app into components	47
The steps for building React apps	58
Step 2: Build a static version of the app	60
TimersDashboard	60
EditableTimer	61
TimerForm	62
ToggleableTimerForm	64
Timer	64
Render the app	66
Try it out	66
Step 3: Determine what should be stateful	68
State criteria	68
Applying the criteria	69
Step 4: Determine in which component each piece of state should live	70
The list of timers and properties of each timer	70
Whether or not the edit form of a timer is open	71
Visibility of the create form	71
Step 5: Hard-code initial states	71
Adding state to TimersDashboard	72
Receiving props in EditableTimerList	73
Props vs. state	74
Adding state to EditableTimer	74
Timer and TimerForm remain stateless	75
Adding state to ToggleableTimerForm	75
Step 6: Add inverse data flow	77
TimerForm	78
ToggleableTimerForm	80
TimersDashboard	81
Updating timers	83
Adding editability to Timer	84
Updating EditableTimer	84
Updating EditableTimerList	85
Defining onEditFormSubmit() in TimersDashboard	86
Deleting timers	90
Adding the event handler to Timer	90
Routing through EditableTimer	91
Routing through EditableTimerList	91

CONTENTS

Implementing the delete function in TimersDashboard	91
Adding timing functionality	93
Adding a forceUpdate() interval to Timer	94
Try it out	95
Add start and stop functionality	95
Add timer action events to Timer	95
Create TimerActionButton	96
Run the events through EditableTimer and EditableTimerList	97
Try it out	100
Methodology review	101
Components & Servers	103
Introduction	103
Preparation	103
server.js	103
The Server API	104
text/html endpoint	105
JSON endpoints	105
Playing with the API	106
Loading state from the server	109
Try it out	112
client	112
Fetch	113
Sending starts and stops to the server	116
Sending creates, updates, and deletes to the server	118
Give it a spin	120
Next up	120
JSX and the Virtual DOM	122
React Uses a Virtual DOM	122
Why Not Modify the Actual DOM?	122
What is a Virtual DOM?	122
Virtual DOM Pieces	123
ReactElement	124
Experimenting with ReactElement	124
Rendering Our ReactElement	126
Adding Text (with children)	128
ReactDOM.render()	129
JSX	130
JSX Creates Elements	130
JSX Attribute Expressions	131
JSX Conditional Child Expressions	132
JSX Boolean Attributes	132

CONTENTS

JSX Comments	133
JSX Spread Syntax	133
JSX Gotchas	134
JSX Summary	137
References	137
Advanced Component Configuration with <code>props</code>, <code>state</code>, and <code>children</code>	138
Intro	138
ReactComponent	139
Creating ReactComponents - <code>createClass</code> or ES6 Classes	139
<code>render()</code> Returns a ReactElement Tree	139
Getting Data into <code>render()</code>	140
<code>props</code> are the parameters	141
PropTypes	142
Default props with <code>get defaultProps()</code>	143
context	143
state	149
Using state: Building a Custom Radio Button	149
<code>getInitialState()</code>	154
Thinking About State	156
Stateless Components	157
Switching to Stateless	158
Stateless Encourages Reuse	160
Talking to Children Components with <code>props.children</code>	160
<code>React.Children.map()</code> & <code>React.Children.forEach()</code>	163
<code>React.Children.toArray()</code>	164
ReactComponent Static Methods	165
Summary	166
References	166
Forms	167
Forms 101	167
Preparation	167
The Basic Button	168
Events and Event Handlers	170
Back to the Button	171
Text Input	173
Accessing User Input With <code>refs</code>	174
Using User Input	176
Uncontrolled vs. Controlled Components	179
Accessing User Input With <code>state</code>	180
Multiple Fields	182
On Validation	186

CONTENTS

Adding Validation to Our App	187
Creating the Field Component	191
Using our new Field Component	195
Remote Data	200
Building the Custom Component	201
Adding CourseSelect	206
Separation of View and State	209
Async Persistence	210
Redux	216
Form Component	220
Connect the Store	224
Form Modules	226
formsy-react	226
react-input-enhancements	226
tcomb-form	227
winterfell	227
react-redux-form	227
Intro to Flux and Redux	228
Why Flux?	228
Flux is a Design Pattern	228
Flux overview	229
Flux implementations	230
Redux	230
Redux's key ideas	230
Building a counter	231
Preparation	231
Overview	232
The counter's actions	233
Incrementing the counter	234
Decrementing the counter	235
Supporting additional parameters on actions	237
Building the store	239
Try it out	243
The core of Redux	244
Next up	245
The beginnings of a chat app	246
Previewing	246
State	248
Actions	248
Building the reducer()	249
Initializing state	249
Handling the ADD_MESSAGE action	250

CONTENTS

Handling the DELETE_MESSAGE action	253
Subscribing to the store	256
createStore() in full	257
Connecting Redux to React	260
Using store.getState()	260
Using store.subscribe()	261
Using store.dispatch()	261
The app's components	262
Preparing app.js	263
The App component	263
The MessageInput component	265
The MessageView component	267
ReactDOM.render()	269
Next up	270
Intermediate Redux	271
Preparation	271
Using createStore() from the redux library	272
Try it out	273
Representing messages as objects in state	273
Updating ADD_MESSAGE	274
Updating DELETE_MESSAGE	276
Updating the React components	277
Introducing threads	280
Supporting threads in initialState	282
Supporting threads in the React components	284
Modifying App	285
Turning MessageView into Thread	286
Try it out	287
Adding the ThreadTabs component	289
Updating App	289
Creating ThreadTabs	290
Try it out	291
Supporting threads in the reducer	293
Updating ADD_MESSAGE in the reducer	293
Updating the MessageInput component	298
Try it out	299
Updating DELETE_MESSAGE in the reducer	301
Try it out	304
Adding the action OPEN_THREAD	305
The action object	305
Modifying the reducer	305
Dispatching from ThreadTabs	306

CONTENTS

Try it out	307
Breaking up the reducer function	309
A new reducer()	309
Updating threadsReducer()	311
Try it out	315
Adding messagesReducer()	315
Modifying the ADD_MESSAGE action handler	316
Creating messagesReducer()	317
Modifying the DELETE_MESSAGE action handler	318
Adding DELETE_MESSAGE to messagesReducer()	321
Defining the initial state in the reducers	322
Initial state in reducer()	323
Adding initial state to activeThreadIdReducer()	324
Adding initial state to threadsReducer()	325
Try it out	326
Using combineReducers() from redux	327
Next up	327
Using Presentational and Container Components with Redux	329
Presentational and container components	329
Splitting up ThreadTabs	331
Splitting up Thread	336
Removing store from App	344
Try it out	345
Generating containers with react-redux	345
The Provider component	346
Wrapping App in Provider	346
Using connect() to generate ThreadTabs	347
Using connect() to generate ThreadDisplay	350
Action creators	356
Conclusion	359
Asynchronicity and server communication	360
Using GraphQL	361
Your First GraphQL Query	361
GraphQL Benefits	363
GraphQL vs. REST	364
GraphQL vs. SQL	365
Relay and GraphQL Frameworks	365
Chapter Preview	367
Consuming GraphQL	367
Exploring With GraphiQL	367
GraphQL Syntax 101	375

CONTENTS

Complex Types	379
Unions	380
Fragments	381
Interfaces	382
Exploring a Graph	383
Graph Nodes	386
Viewer	388
Graph Connections and Edges	389
Mutations	392
Subscriptions	393
GraphQL With JavaScript	394
GraphQL With React	396
Wrapping Up	397
GraphQL Server	398
Writing a GraphQL Server	398
Game Plan	398
Express HTTP Server	398
Adding First GraphQL Types	401
Adding GraphiQL	403
Introspection	406
Mutation	407
Rich Schemas and SQL	410
Setting Up The Database	411
Schema Design	415
Object and Scalar Types	416
Lists	422
Performance: Look-Ahead Optimizations	424
Lists Continued	427
Connections	430
Authentication	437
Authorization	439
Rich Mutations	443
Relay and GraphQL	446
Performance: N+1 Queries	447
Summary	451
Appendix A: PropTypes	453
Validators	453
string	454
number	454
boolean	455
function	456

CONTENTS

object	456
object shape	457
multiple types	457
instanceOf	458
array	458
array of type	459
node	460
element	460
any type	462
Optional & required props	462
custom validator	463
Appendix B: Tools	464
Curl	464
A GET Request	464
A POST Request	464
Chrome “Copy as cURL”	465
More Resources	466
Changelog	467
Revision 14 - 2016-08-26	467
Revision 13 - 2016-08-02	467
Revision 12 - 2016-07-26	467
Revision 11 - 2016-07-08	467
Revision 10 - 2016-06-24	467
Revision 9 - 2016-06-21	467
Revision 8 - 2016-06-02	467
Revision 7 - 2016-05-13	467
Revision 6 - 2016-05-13	468
Revision 5 - 2016-04-25	468
Revision 4 - 2016-04-22	468
Revision 3 - 2016-04-08	468
Revision 2 - 2016-03-16	468
Revision 1 - 2016-02-14	468

Book Revision

Revision 14 - Covers up to React (15.3.1, 2016-08-26)

Prerelease

This book is a prerelease version and a work-in-progress.

Bug Reports

If you'd like to report any bugs, typos, or suggestions just email us at: react@fullstack.io¹.

Chat With The Community!

We're experimenting with a community chat room for this book using Gitter. If you'd like to hang out with other people learning React, come [join us on Gitter](#)²!

Be notified of updates via Twitter

If you'd like to be notified of updates to the book on Twitter, [follow @fullstackio](#)³

We'd love to hear from you!

Did you like the book? Did you find it helpful? We'd love to add your face to our list of testimonials on the website! Email us at: react@fullstack.io⁴.

¹<mailto:react@fullstack.io?Subject=Fullstack%20React%20book%20feedback>

²<https://gitter.im/fullstackreact/fullstackreact>

³<https://twitter.com/fullstackio>

⁴<mailto:react@fullstack.io?Subject=react%202%20testimonial>

Your first React Web Application

Building Product Hunt

In this chapter, you're going to get a crash course on React by building a simple voting application inspired by [Product Hunt](#)⁵. You'll become familiar with how React approaches front-end development and all of the fundamentals necessary to build an interactive React app from start to finish. Thanks to React's core simplicity, by the end of the chapter you'll already be well on your way to writing a variety of fast, dynamic interfaces.

We'll focus on getting our React app up and running fast. We take a deeper look at concepts covered in this section throughout the course.

Getting started

Sample Code

All the code samples are included with this course. Still, we highly recommend implementing the code yourself along with us throughout each section. Playing around with examples and sample code will help solidify and strengthen concepts.

Provided with the sample code is a very simple Node.js web server that we'll be using to run our application.

Code editor

As you'll be writing code throughout this course, you'll need to make sure you have a code editor you're comfortable working with. If you don't already have a preferred editor, we recommend installing [Atom](#)⁶ or [Sublime Text](#)⁷.

Node.js and NPM

For all the projects in this course, we'll need to make sure we have a working [Node.js](#)⁸ development environment along with NPM.

⁵<http://producthunt.com>

⁶<http://atom.io>

⁷<https://www.sublimetext.com/>

⁸<http://nodejs.org>

There are a couple different ways you can install Node.js so please refer to the Node.js website for detailed information: <https://nodejs.org/download/>⁹



If you're on a Mac, your best bet is to install Node.js directly from the Node.js website instead of through another package manager (like Homebrew). Installing Node.js via Homebrew is known to cause some issues.

The Node Package Manager (npm for short) is installed as a part of Node.js. To check if npm is available as a part of our development environment, we can open a Terminal window and type:

```
$ npm -v
```

If a version number is not printed out and you receive an error, make sure to download a Node.js installer that includes npm.

Browser

Lastly, we highly recommend using the [Google Chrome Web Browser](#)¹⁰ to develop React apps. We'll use the Chrome developer toolkit throughout this course. To follow along with our development and debugging we recommend downloading it now.

Previewing the application

We'll be building a basic React app that will allow us to touch on React's most important concepts at a high-level before diving into them in subsequent sections. Let's begin by taking a look at a working implementation of the app.

Open up the sample code that came with the course, changing to the `voting_app/` directory in the terminal:

```
$ cd voting_app/
```



If you're not familiar with `cd`, it stands for "change directory." If you're on a Mac, do the following to open terminal and change to the proper directory:

1. Open up `/Applications/Utilities/Terminal.app`.
2. Type `cd`, without hitting enter.
3. Tap the spacebar.
4. In the Finder, drag the `voting_app/` folder on to your terminal window.
5. Hit Enter.

Your terminal is now in the proper directory.

⁹<https://nodejs.org/download/>

¹⁰<https://www.google.com/chrome/>



Throughout the book, a codeblock starting with a \$ signifies a command to be run in your terminal.



Windows Users: We'll be using Linux/Mac-style commands on the command line throughout this book. We'd highly recommend installing [Cygwin¹¹](#) as it will let you run commands just as we have printed here.

First, we'll need to use `npm` to install all our dependencies:

```
$ npm install
```

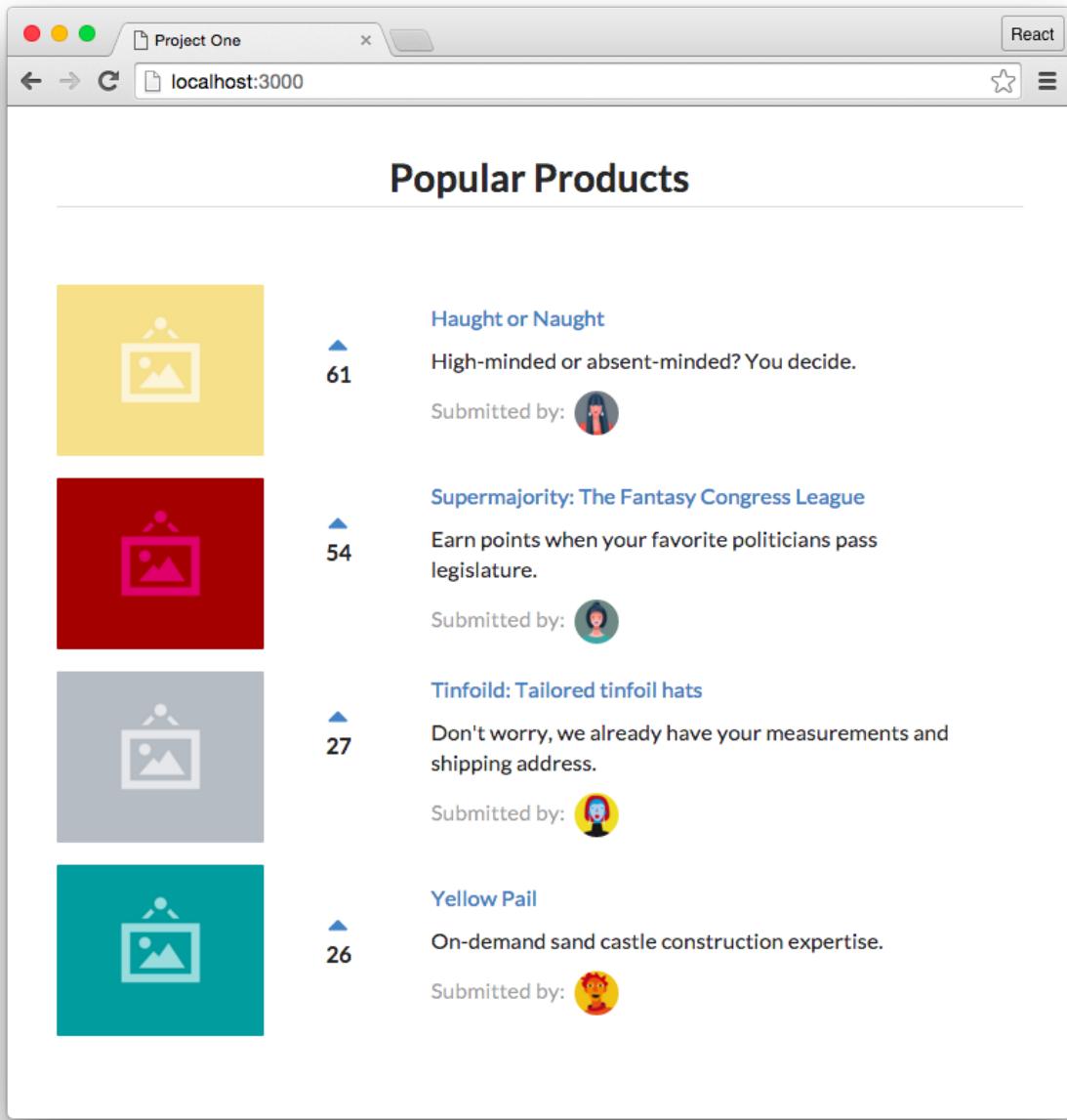
With our dependencies installed, we can boot the server using the `npm run` script `server`:

```
$ npm run server
```

Heading to our browser, we can view the running application at the URL: [http://localhost:3000¹²](http://localhost:3000).

¹¹<https://www.cygwin.com/>

¹²<http://localhost:3000>



The screenshot shows a web browser window titled "Project One" with the URL "localhost:3000". The page displays a list of four products under the heading "Popular Products". Each product entry includes a small image icon, a rank number, the product name, a description, and a "Submitted by:" section with a user icon.

Rank	Product Name	Description	Submitted by:
61	Haught or Naught	High-minded or absent-minded? You decide.	
54	Supermajority: The Fantasy Congress League	Earn points when your favorite politicians pass legislature.	
27	Tinfoild: Tailored tinfoil hats	Don't worry, we already have your measurements and shipping address.	
26	Yellow Pail	On-demand sand castle construction expertise.	

Completed version of the app



Mac users can click on links inside of terminal. Just hold command and double-click on the link:

```
Find the server at http://localhost:3000
Starting up http-server, serving .
Available on:
  http://127.0.0.1:3000
  http://192.168.105.102:3000
Hit CTRL-C to stop the server
```

Clicking a link in the console

This demo app is a site like [Product Hunt¹³](#) or [Reddit¹⁴](#). These sites have lists of links that users can vote on.

In our app we can up-vote products and all products are sorted, instantaneously, by number of votes.



The app we build in this section has a slightly different style than the completed version we just looked at. This is because the completed version has some additional HTML structure that's purely for aesthetics. Feel free to use the HTML in `app-complete.js` to style your component after completing this section.



To quit a running Node server, hit `CTRL+C`.

Prepare the app

In the terminal, run `ls -lp` to see the project's layout:

```
$ ls -lp
```

¹³<http://producthunt.com>

¹⁴<http://reddit.com>

```
README.md  
app-complete.js  
app.js  
data.js  
images/  
index.html  
node_modules/  
package.json  
style.css  
vendor/
```

We'll be working with `index.html` and `app.js` for this project. `app-complete.js` is the completed application that we will be building towards.



All projects include a handy `README.md` that have instructions on how to run them.

To get started, we'll ensure `app-complete.js` is no longer loaded in `index.html`. We'll then have a blank canvas to begin work inside `app.js`.

Open up `index.html` in your favorite text editor. It should look like this:

```
<!DOCTYPE html>  
<html>  
  
<head>  
  <meta charset="utf-8">  
  <!-- Disable browser cache -->  
  <meta http-equiv="cache-control" content="max-age=0" />  
  <meta http-equiv="cache-control" content="no-cache" />  
  <meta http-equiv="expires" content="0" />  
  <meta http-equiv="expires" content="Tue, 01 Jan 1980 1:00:00 GMT" />  
  <meta http-equiv="pragma" content="no-cache" />  
  <title>Project One</title>  
  <link rel="stylesheet" href="vendor/semantic-ui/semantic.min.css" />  
  <link rel="stylesheet" href="style.css" />  
  <script src="vendor/babel-core-5.8.25.js"></script>  
  <script src="vendor/react.js"></script>  
  <script src="vendor/react-dom.js"></script>  
</head>  
  
<body>
```

```

<div class="main ui text container">
  <h1 class="ui dividing centered header">Popular Products</h1>
  <div id="content"></div>
</div>
<script src=".data.js"></script>
<script src=".app.js"></script>
<!-- Delete the line below to get started. -->
<script type="text/babel" src=".app-complete.js"></script>
</body>

</html>

```

We'll go over all of the dependencies being loaded under the `<head>` tag later. The main HTML document is these few lines here:

```

<div class="main ui text container">
  <h1 class="ui dividing centered header">Popular Products</h1>
  <div id="content"></div>
</div>

```



For this project, we're using [Semantic UI¹⁵](#) for styling.

Semantic UI is a CSS framework, much like Twitter's [Bootstrap¹⁶](#). It provides us with a grid system and some simple styling. You don't need to know Semantic UI in order to use this book. We'll provide all of the styling code that you need. At some point, you might want to check out the docs [Semantic UI docs¹⁷](#) to get familiar with the framework and explore how you can use it in your own projects.

The `class` attributes here are just concerned with style and are safe to ignore. Stripping those away, our core markup is quite succinct:

```

<div>
  <h1>Popular Products</h1>
  <div id="content"></div>
</div>

```

We have a title for the page (`h1`) and a `div` with an `id` of `content`. This `div` is where we will ultimately mount our React app. We'll see shortly what that means.

The next few lines tell the browser what JavaScript to load. To start building our own application, let's remove the `./app-complete.js` script tag completely. The comments in the code indicate which line to remove:

¹⁵<http://semantic-ui.com/>

¹⁶<http://getbootstrap.com/>

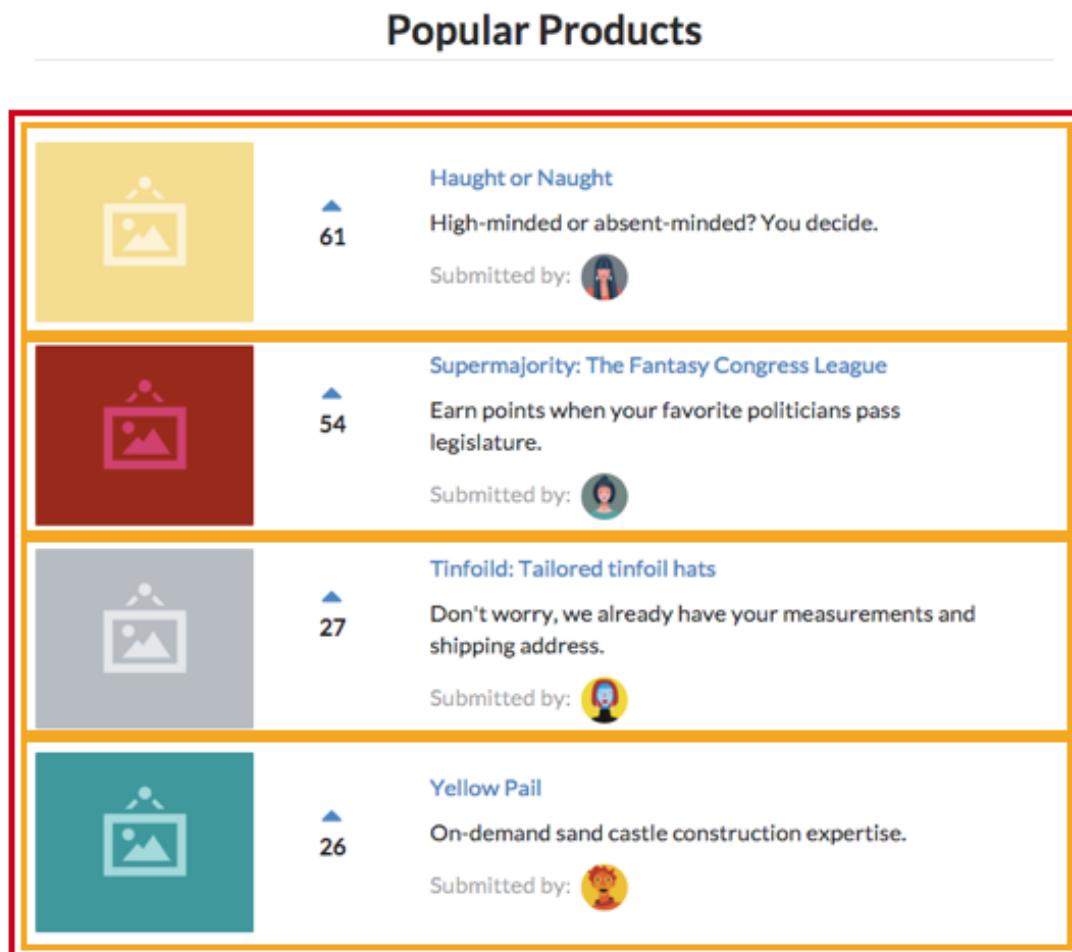
¹⁷<http://semantic-ui.com/introduction/getting-started.html>

```
<script src="./data.js"></script>
<script src="./app.js"></script>
<!-- Delete the line below to get started. --&gt;
&lt;script type="text/babel" src="./app-complete.js"&gt;&lt;/script&gt;</pre>
```

After we save our updated `index.html` and reload the web browser, we'll see that our app has disappeared.

Our first component

Building a React app is all about **components**. An individual React component can be thought of as a *UI* component in an app. Let's take a look at the components in our app:



The app's components

We have a hierarchy of one parent component and many sub-components:

1. ProductList (**red**): Contains a list of product components
2. Product (**orange**): Displays a given product

Not only do React components map cleanly to UI components, but they are self-contained. The markup, view logic, and often component-specific style is all housed in one place. This feature makes React components reusable. Furthermore, as we'll see in this chapter and throughout this course, React's paradigm for component data flow and interactivity is rigidly defined. We'll see how this well-defined system is beneficial.

The classic UI paradigm is to manipulate the DOM with JavaScript directly. As complexity of an app grows, this can lead to all sorts of inconsistencies and headaches around managing state and transitions. In React, when the inputs for a component change, the framework simply re-renders that component. This gives us a robust UI consistency guarantee:

With a given set of inputs, the output (how the component looks on the page) will always be the same.

Let's start off by building the `ProductList` component. We'll write all of our React code for this section inside the file `app.js`. Let's open the `app.js` file and insert the component:

```
const ProductList = React.createClass({  
  render: function () {  
    return (  
      <div className='ui items'>  
        Hello, friend! I am a basic React component.  
      </div>  
    );  
  },  
});
```

ES6: Prefer `const` and `let` over `var`

Both the `const` and `let` statements declare variables. They are introduced in ES6.

`const` is a superior declaration in cases where a variable is never re-assigned. Nowhere else in the code will we re-assign the `ProductList` variable. Using `const` makes this clear to the reader. It refers to the “constant” state of the variable in the context it is defined within.

If the variable will be re-assigned, use `let`.

If you've worked with JavaScript before, you're likely used to seeing variables declared with `var`:

```
var ProductList = ...
```

We encourage the use of `const` and `let` instead of `var`. In addition to the restriction introduced by

`const`, both `const` and `let` are *block scoped* as opposed to *function scoped*. Typically this separation of scope helps avoid unexpected bugs.

React.createClass()

To create a component, we use the function `React.createClass()`. This is how all components are defined in React. We pass in a single argument to this function: a JavaScript object.

This *class definition object* (the argument we pass to the `React.createClass()` method) in our case has just one key, `render`, which defines a rendering function. `render()` is the only required method for a React component. React uses the return value from this method to determine what exactly to render to the page.



The `createClass()` method is one way to create a React component. The other main way of defining one is by using the ES6 classical implementation:

```
class ProductList extends React.Component {}
```

We'll primarily be using the `createClass()` method throughout this course.

If you have some familiarity with JavaScript, the return value may be surprising:

```
return (
  <div className='ui items'>
    Hello, friend! I am a basic React component.
  </div>
);
```

The syntax of the return value doesn't look like traditional JavaScript. We're using **JSX** (JavaScript eXtension syntax), a syntax extension for JavaScript written by Facebook. Using JSX enables us to write the markup for our component views in a familiar, HTML-like syntax. In the end, this JSX code compiles to vanilla JavaScript. Although using JSX is not a necessity, we'll use it in this course as it pairs really well with React.



If you don't have much familiarity with JavaScript, we recommend you follow along and use JSX in your React code too. You'll learn the boundaries between JSX and JavaScript with experience.

JSX

React components ultimately render HTML which is displayed in the browser. As such, the `render()` method of a component needs to describe how the view should be represented as HTML. React builds our apps with a fake representation of the Document Object Model (DOM, for short, refers to the browser's HTML tree that makes up a web page). React calls this the *virtual DOM*. Without getting deep into details for now, React allows us to describe a component's HTML representation in JavaScript.

JSX was created to make this JavaScript representation of HTML more HTML-like. To understand the difference between HTML and JSX, consider this JavaScript syntax:

```
React.createElement('div', {className: 'ui items'},
  'Hello, friend! I am a basic React component.'
)
```

Which can be represented in JSX as:

```
<div className='product'>
  Hello, friend! I am a basic React component.
</div>
```

The code readability is slightly improved in the latter example. This is exacerbated in a nested tree structure:

```
React.createElement('div', {className: 'ui items'},
  React.createElement('p', null, 'Hello, friend! I am a basic React component.')
)
```

In JSX:

```
<div className='ui items'>
  <p>
    Hello, friend! I am a basic React component.
  </p>
</div>
```

JSX presents a light abstraction over the JavaScript version, yet the legibility benefits are huge. Readability boosts our app's longevity and makes it easier to onboard new developers.



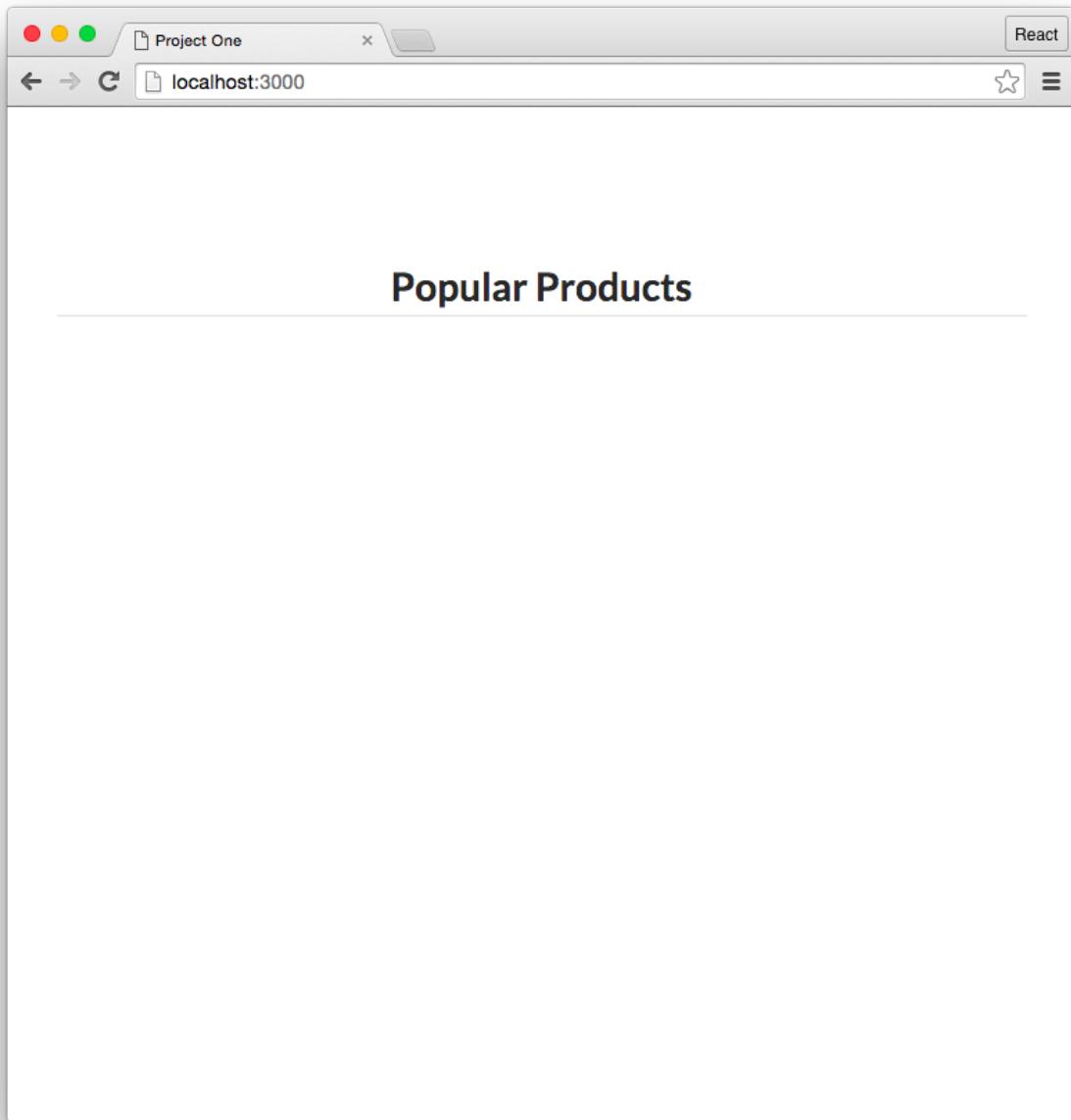
Even though the JSX above looks exactly like HTML, it's important to remember that JSX is actually just compiled into JavaScript (ex: `React.createElement('div')`).

During runtime React takes care of rendering the actual HTML in the browser for each component.

The developer console

Our first component is written and we now know that it uses a special flavor of JavaScript called JSX for improved readability.

After editing and saving our `app.js`, let's refresh the page in our web browser and see what changed:



Nothing?

Every major browser comes with a toolkit that helps developers working on JavaScript code. A

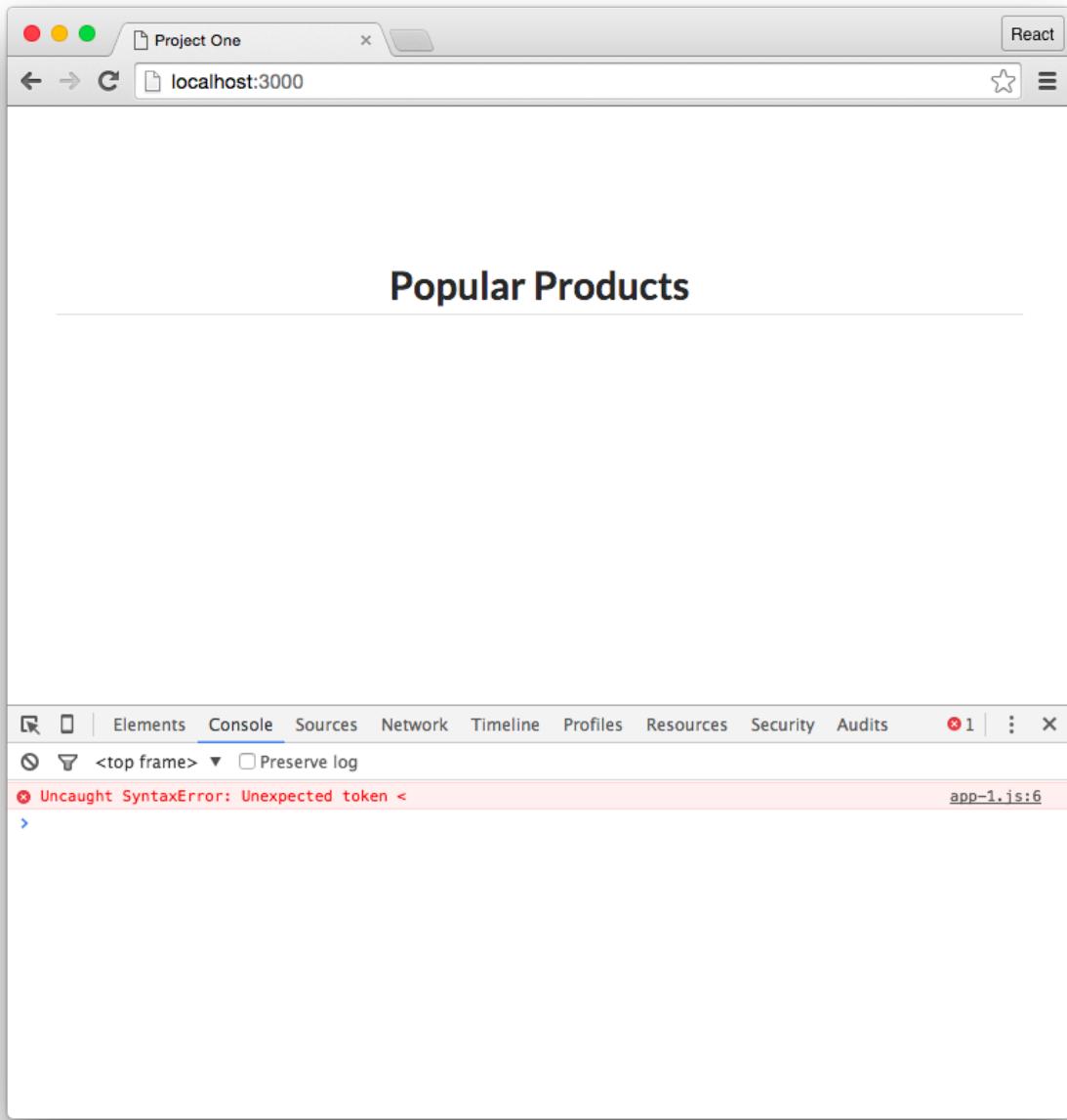
central part of this toolkit is a console. Think of the console as JavaScript's primary communication medium back to the developer. If JavaScript encounters any errors in its execution, it will alert you in this developer console.



- To open the console in Chrome, navigate to View > Developer > JavaScript Console.
- Or, just press Command + Option + J on a Mac or Control + Shift + L on Windows/Linux.

Opening the console, we are given a cryptic clue:

```
Uncaught SyntaxError: Unexpected token <
```



Error in the console

This `SyntaxError` prevented our code from running. A `SyntaxError` is thrown when the JavaScript engine encounters tokens or token order that doesn't conform to the syntax of the language when parsing code. This error type indicates some code is out of place or mistyped.

The issue? Our browser's JavaScript parser exploded when it encountered the JSX. The parser doesn't know anything about JSX. As far as it is concerned, this `<` is completely out of place.

We know that JSX is an extension to standard JavaScript. Let's empower our browser's plain old

JavaScript interpreter to use this extension.

Babel

Babel is a JavaScript transpiler. For those familiar with ES6 JavaScript, Babel turns ES6/ES7 code into ES5 code so that our browser can use lots of these features with browsers that only understand ES5.

For our purposes, a handy feature of Babel is that it understands JSX. Babel compiles our JSX into vanilla JS that our browser can then interpret and execute. Let's tell the browser's JS engine we want to use babel to compile and run our JavaScript code.

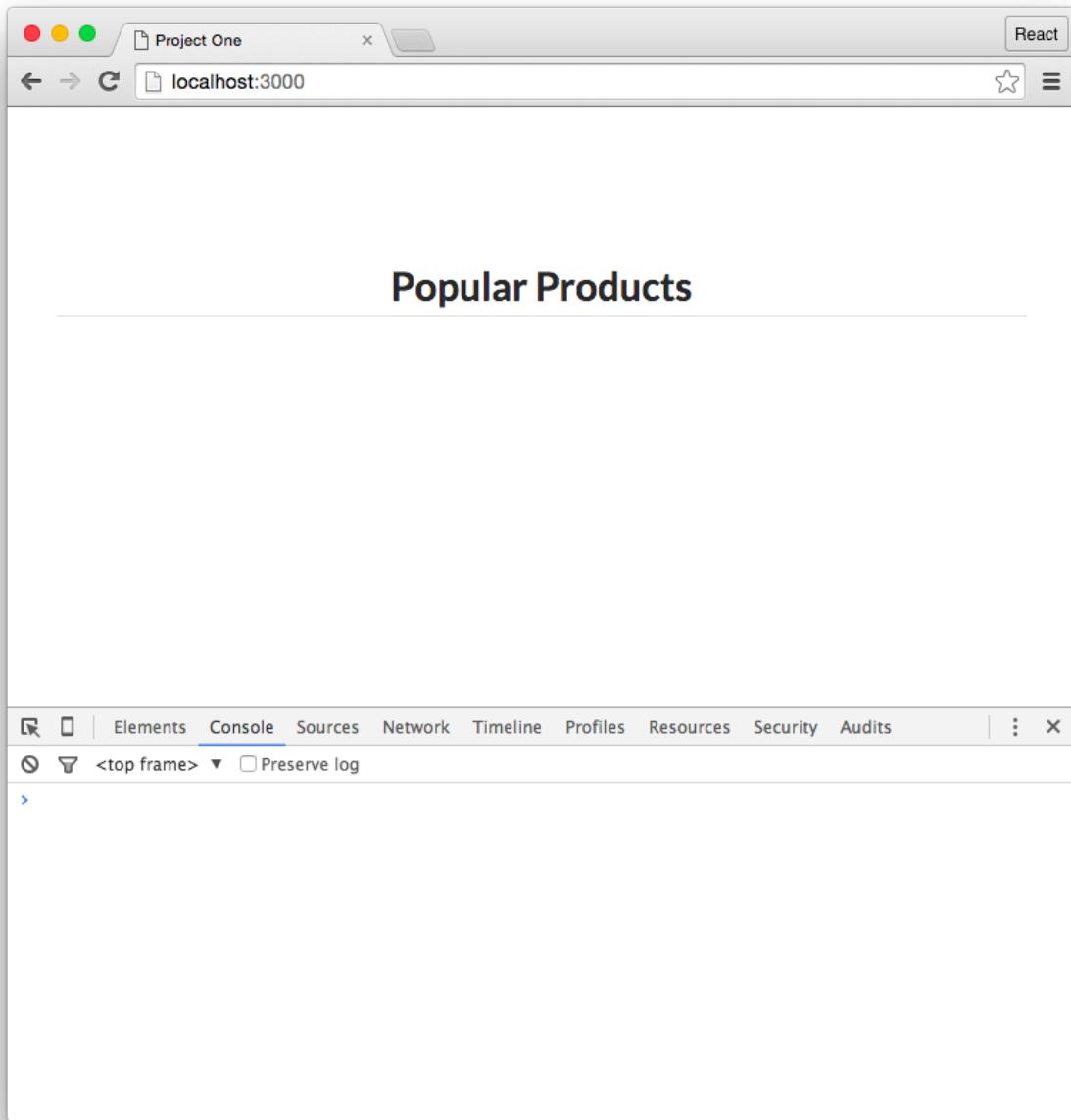
The sample code's `index.html` already imports Babel in the `head` tags of `index.html`:

```
<head>
  <!-- ... -->
  <script src="vendor/babel-core-5.8.25.js"></script>
  <!-- ... -->
</head>
```

All we need to do is tell our JavaScript runtime that our code should be compiled by Babel. We can do this by setting the `type` attribute when we import the script in `index.html` from `text/javascript` to `text/babel`. Open `index.html` and change this line:

```
<script src=".data.js"></script>
<script type="text/babel" src=".app.js"></script>
```

Save `index.html` and reload the page.



Still nothing. However, the console reveals we no longer have any JavaScript compilation errors. Babel successfully compiled our JSX into JS and our browser was able to run that JS without any issues.

- i** If your console encounters an error, check that the file has been saved before reloading the browser. If an error still persists, check the code previously added to make sure there aren't any syntactical mistakes.

So what's happening? We've defined the component, but we haven't yet told React to do anything with it yet. We need to tell the React framework that we want to run our app.

ReactDOM.render()

We're going to instruct React to render this `ProductList` inside a specific DOM node.

Add the following code below the component inside `app.js`:

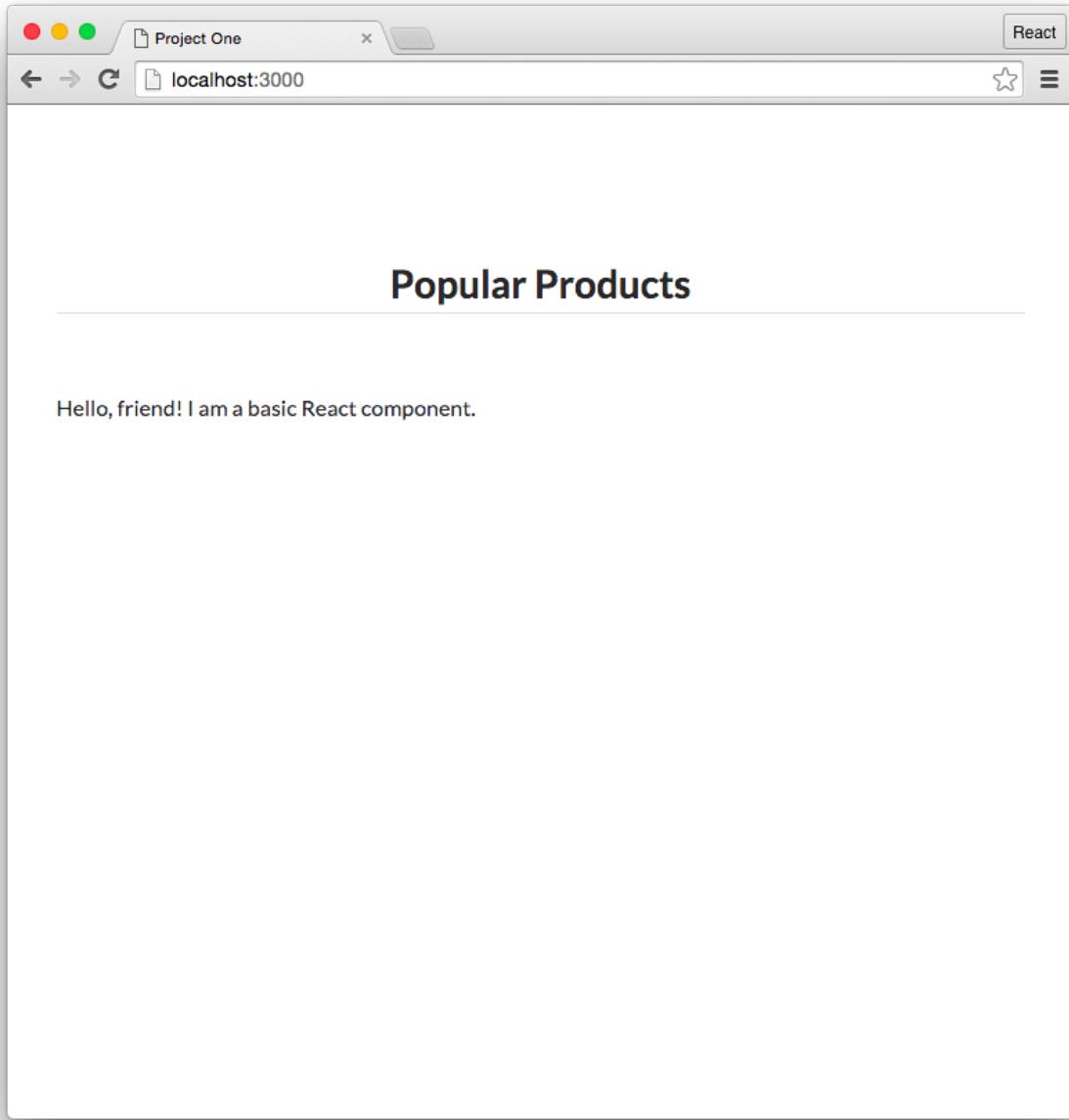
```
ReactDOM.render(  
  <ProductList />,  
  document.getElementById('content')  
);
```

We pass in two arguments to the `ReactDOM.render()` method. The first argument is *what* we'd like to render. Here, we're passing in a reference to our React component `ProductList` in JSX. The second argument is *where* to render it. We'll send a reference to the browser's DOM element.

```
ReactDOM.render([what], [where]);
```

In our code, we have a difference in casing between the different types of React element declarations. We have HTML DOM elements like `<div>` and a React component called `<ProductList />`. In React, native HTML elements *always* start with a lowercase letter whereas React component names *always* start with an uppercase letter.

With `ReactDOM.render()` now at the end of `app.js`, save the file and refresh the page in the browser:



We successfully implemented a React component in JSX, ensured it was being compiled to JS, and rendered it in the DOM in the web browser.

Our second component

Currently, our only component is `ProductList`. We'll want `ProductList` to render a list of products (its “sub-components” or “child components”). In HTML, we could render this entirely in the JSX

that `ProductList` returns. Although this works, we don't get any benefit from encoding our entire app in a single component. Just like we can embed HTML elements in the JSX of our components, we can embed other React components.

Let's build a child component, `Product`, that will contain a product listing. Just like with the `ProductList` component, we'll use the `React.createClass()` function with a single key of `render`:

```
const Product = React.createClass({
  render: function () {
    return (<div></div>)
  }
});
```

For every product, we'll add an image, a title, a description, and an avatar of the post author. The relevant code might look something like:

```
const Product = React.createClass({
  render: function () {
    return (
      <div className='item'>
        <div className='image'>
          <img src='images/products/image-aqua.png' />
        </div>
        <div className='middle aligned content'>
          <div className='description'>
            <a>Fort Knight</a>
            <p>Authentic renaissance actors, delivered in just two weeks.</p>
          </div>
          <div className='extra'>
            <span>Submitted by:</span>
            <img
              className='ui avatar image'
              src='images/avatars/daniel.jpg'
            />
          </div>
        </div>
      </div>
    );
  },
});
```

We've used a bit of SemanticUI styling in our code here. As we discussed previously, this JSX code will be compiled to JavaScript in the browser. As it runs in the browser as JavaScript, we cannot use

any reserved JavaScript words in JSX. Setting the `class` attribute on an HTML element is how we add a Cascading StyleSheet (CSS, for short) to apply styles to the class. In JSX, however, we cannot use `class`, so React changes the key from `class` to `className`.

Structurally, the `Product` component is similar to the `ProductList` component. Both have a single `render()` method which returns information about an eventual HTML structure to display.

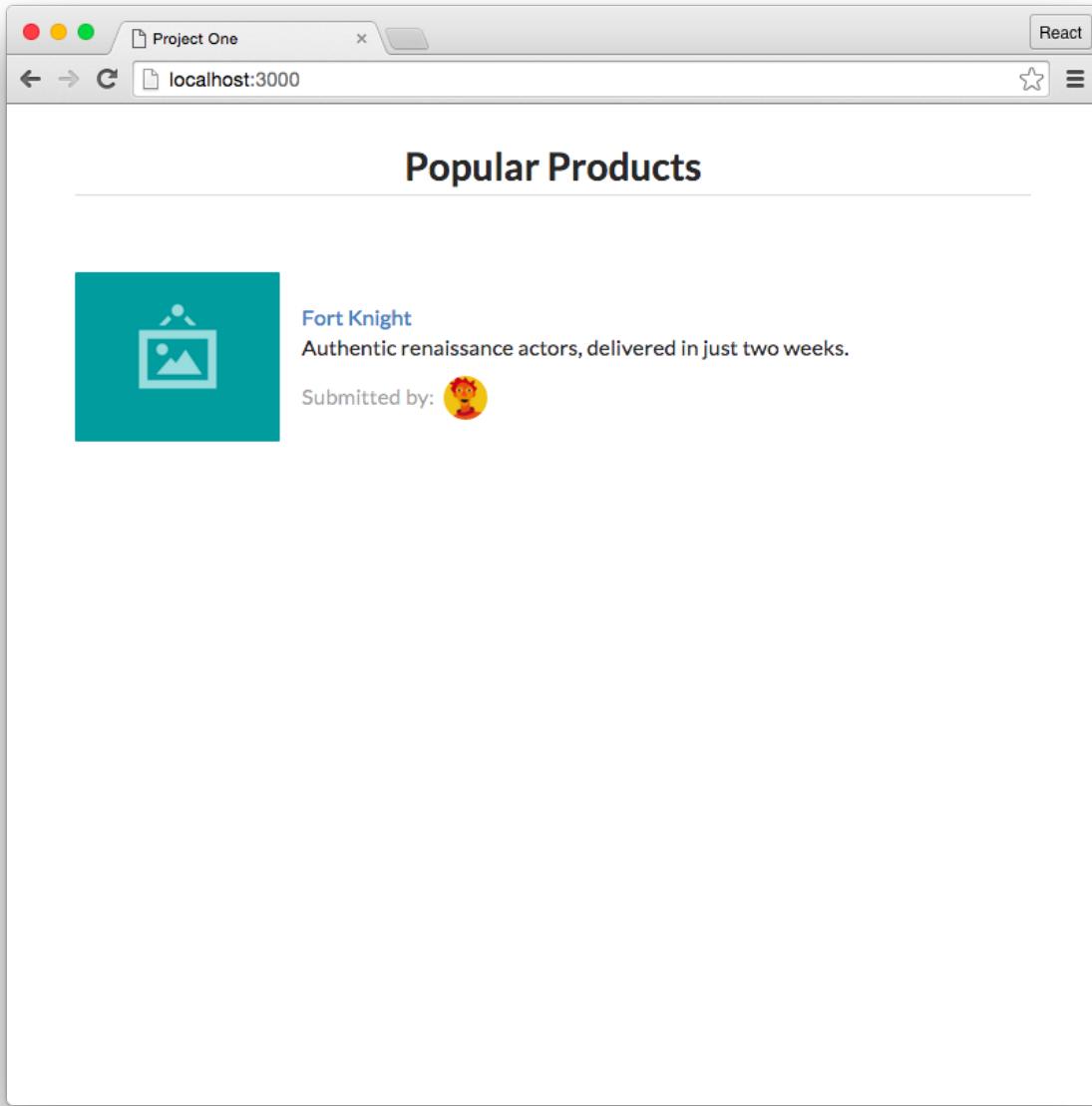


Remember, the JSX components return is *not* actually the HTML that gets rendered, but is the *representation* that we want React to render in the DOM. This course looks at this in-depth in later sections.

To use the `Product` component, we can modify our parent `ProductList` component to list the `Product` component:

```
const ProductList = React.createClass({
  render: function () {
    return (
      <div className='ui items'>
        <Product />
      </div>
    );
  },
});
```

Save `app.js` and refresh the web browser.



With this update, we now have two React components being rendered in our webapp. The `ProductList` parent component is rendering the `Product` component as a child nested underneath its root `div` element.

At the moment, the child `Product` component is static. We hardcoded an image, the name, the description, and author details. To use this component in a meaningful way, we'll change it to be data-driven and therefore dynamic.

Making Product data-driven

Attributes in Product like title and description are currently hard-coded. We will need to tweak our Product component to make it data-driven. Having the component be driven by data will allow us to dynamically render the component based upon the data that we give it. Let's familiarize ourselves with the product data model.

The data model

In the sample code, we've included a file called `data.js` which contains some example data for our products. In the future, we might fetch this data over a network request (we cover this in later sections). The `data.js` file contains a JavaScript object called `Data` that contains an array of JavaScript objects, each representing a product object:

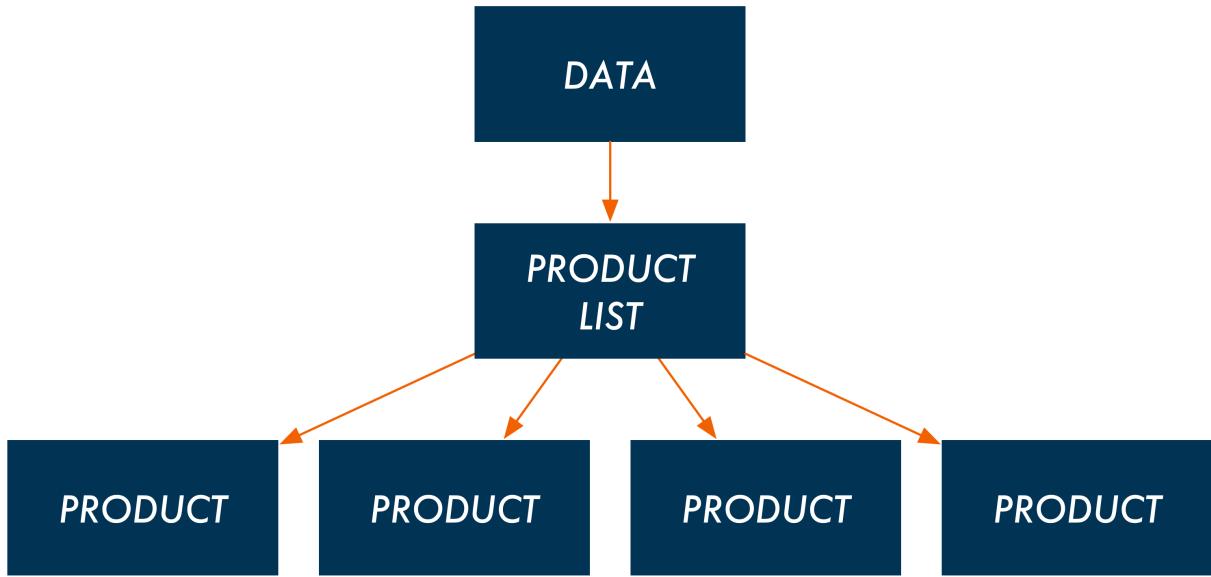
```
id: 1,  
title: 'Yellow Pail',  
description: 'On-demand sand castle construction expertise.',  
url: '#',  
votes: generateVoteCount(),  
submitter_avatar_url: 'images/avatars/daniel.jpg',  
product_image_url: 'images/products/image-aqua.png',  
},  
{
```

Each product has a unique `id` and a handful of properties including a `title` and `description`. `votes` are randomly generated for each one with the included function `generateVoteCount()`.

We can use the same attribute keys in our React code.

Using props

We want to modify our `Product` component so that it no longer uses static, hard-coded attributes, but instead is able to accept data passed down from its parent, `ProductList`. Setting up our component structure in this way enables our `ProductList` component to dynamically render any number of `Product` components that each have their own unique attributes. Data flow will look like this:



The way data flows from parent to child in React is through **props**. When a parent renders a child, it can send along props the child can depend upon.

Let's see this in action. First, let's modify `ProductList` to pass down props to `Product`. Using our `Data` object instead of typing the data in, let's pluck the first object off of the `Data` array and use that for the single product:

```
const ProductList = React.createClass({
  render: function () {
    const product = Data[0];
    return (
      <div className='ui items'>
        <Product
          id={product.id}
          title={product.title}
          description={product.description}
          url={product.url}
          votes={product.votes}
          submitter_avatar_url={product.submitter_avatar_url}
          product_image_url={product.product_image_url}>
        </Product>
      </div>
    );
  },
});
```

Here, the `product` variable is a JavaScript object that describes the first of our products. We pass

the product's attributes along individually to the Product component using the syntax `[prop_name]=[prop_value]`. The syntax of assigning attributes in JSX is exactly the same as HTML and XML.

There are two interesting things here. The first is the braces (`{ }`) around each of the property values:

```
id={product.id}
```

In JSX, braces are a delimiter, signaling to JSX that what resides in-between the braces is an expression. To pass in a string instead of an expression, for instance, we can do so like this:

```
id='1'
```

Using the `'` as a delimiter for the string instead of the `{ }`.



JSX attribute values **must** be delimited by either braces or quotes.

If type is important and we want to pass in something like a Number or a `null`, use braces.

Now the `ProductList` component is passing props down to `Product`. Our `Product` component isn't using them yet, so let's modify the component to use these props:

```
const Product = React.createClass({
  render: function () {
    return (
      <div className='item'>
        <div className='image'>
          <img src={this.props.product_image_url} />
        </div>
        <div className='middle aligned content'>
          <div className='header'>
            <a>
              <i className='large caret up icon'></i>
            </a>
            {this.props.votes}
          </div>
          <div className='description'>
            <a href={this.props.url}>
              {this.props.title}
            </a>
          </div>
          <div className='extra'>
```

```
<span>Submitted by:</span>
<img
  className='ui avatar image'
  src={this.props.submitter_avatar_url}
/>
</div>
</div>
</div>
);
},
));
});
```

In React, we can access all component props through the `this.props` in a component object. We can access all of the various props we passed along with the names assigned by `ProductList`. Again, we're using braces as a delimiter.

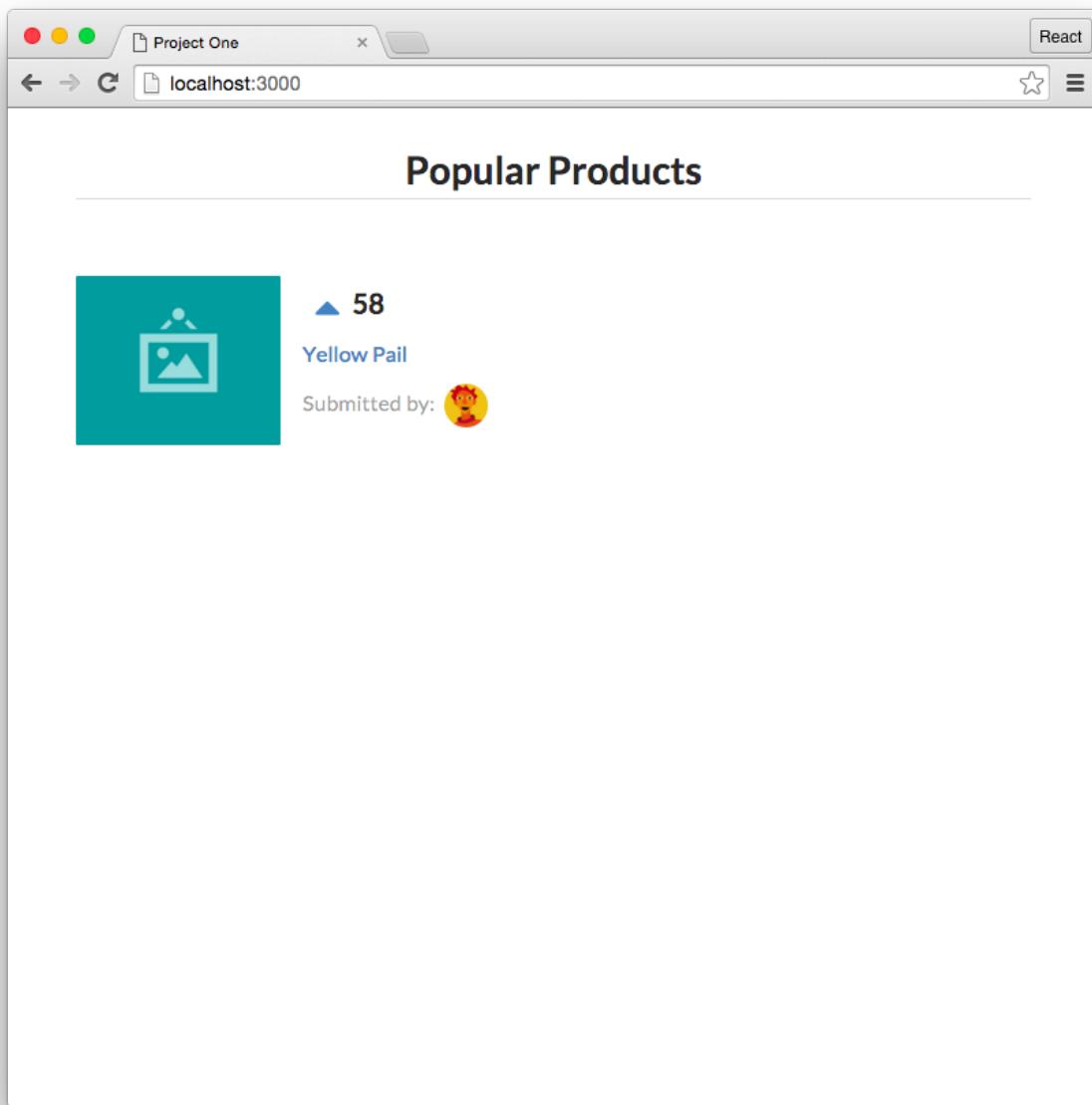


`this` is a special keyword in JavaScript. The details about `this` are a bit nuanced, but for the purposes of the majority of this book, `this` will be bound to the React component class and we'll discuss when it differs in later sections. We use `this` to call methods on the component.

For more details on `this`, check out [this page on MDN](#)¹⁸.

With our updated `app.js` file saved, let's refresh the web browser again:

¹⁸<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this>



The `ProductList` component now shows a single product listed, the first object pulled from `Data`. Now that `Product` is rendering itself based on the props that it receives from `ProductList`, our code is poised to render any number of unique products.

Rendering multiple products

Modify the `ProductList` component again. This time, we're going to make an array of `Product` components, each representing an individual object in the `Data` array. Rather than passing in a single product, let's map over all of our products:

```
const ProductList = React.createClass({
  render: function () {
    const products = Data.map((product) => {
      return (
        <Product
          key={'product-' + product.id}
          id={product.id}
          title={product.title}
          description={product.description}
          url={product.url}
          votes={product.votes}
          submitter_avatar_url={product.submitter_avatar_url}
          product_image_url={product.product_image_url}
        />
      );
    });
    return (
      <div className='ui items'>
        {products}
      </div>
    );
  },
});
```

Before `render`'s `return` method, we create a variable called `products`. We use JavaScript's `map` function to iterate over each one of the objects in the `Data` array.

The function passed to `map` simply returns a `Product` component. Notably, we're able to represent the `Product` component instance in JSX inside of `return` without issue because it compiles to JavaScript. This `Product` is created exactly as before with the same props. As such, the `products` variable ends up just being an array of `Product` components, which in this case is four total.



Array's `map` method takes a function as an argument. It calls this function with each item inside of the array (in this case, each object inside `Data`) and builds a **new** array by using the return value from each function call.

Because the `Data` array has four items, `map` will call this function four times, once for each item. When `map` calls this function, it passes in as the first argument an item. The return value from this function call is inserted into the new array that `map` is constructing. After handling the last item, `map` returns this new array. Here, we're storing this new array in the variable `products`.



Note the use of the `key={'product-' + product.id}` prop. React uses this special property to create unique bindings for each instance of the `Product` component. The `key` prop is not used by our `Product` component, but by the React framework. It's a special property that we discuss deeper in our advanced components section. For the time being, it's enough to note that this property needs to be unique per React instance in a map.

ES6: Arrow functions

Up until this point, we've been using the traditional JavaScript function declaration syntax:

```
render: function () { ... }
```

We've been using this syntax to declare **object methods** for our React component classes.

Inside of the `render()` method of `ProductList`, we pass an **anonymous arrow function** to `map()`. Arrow functions were introduced in ES6. Throughout the book, whenever we're declaring anonymous functions we will use arrow functions. This is for two reasons.

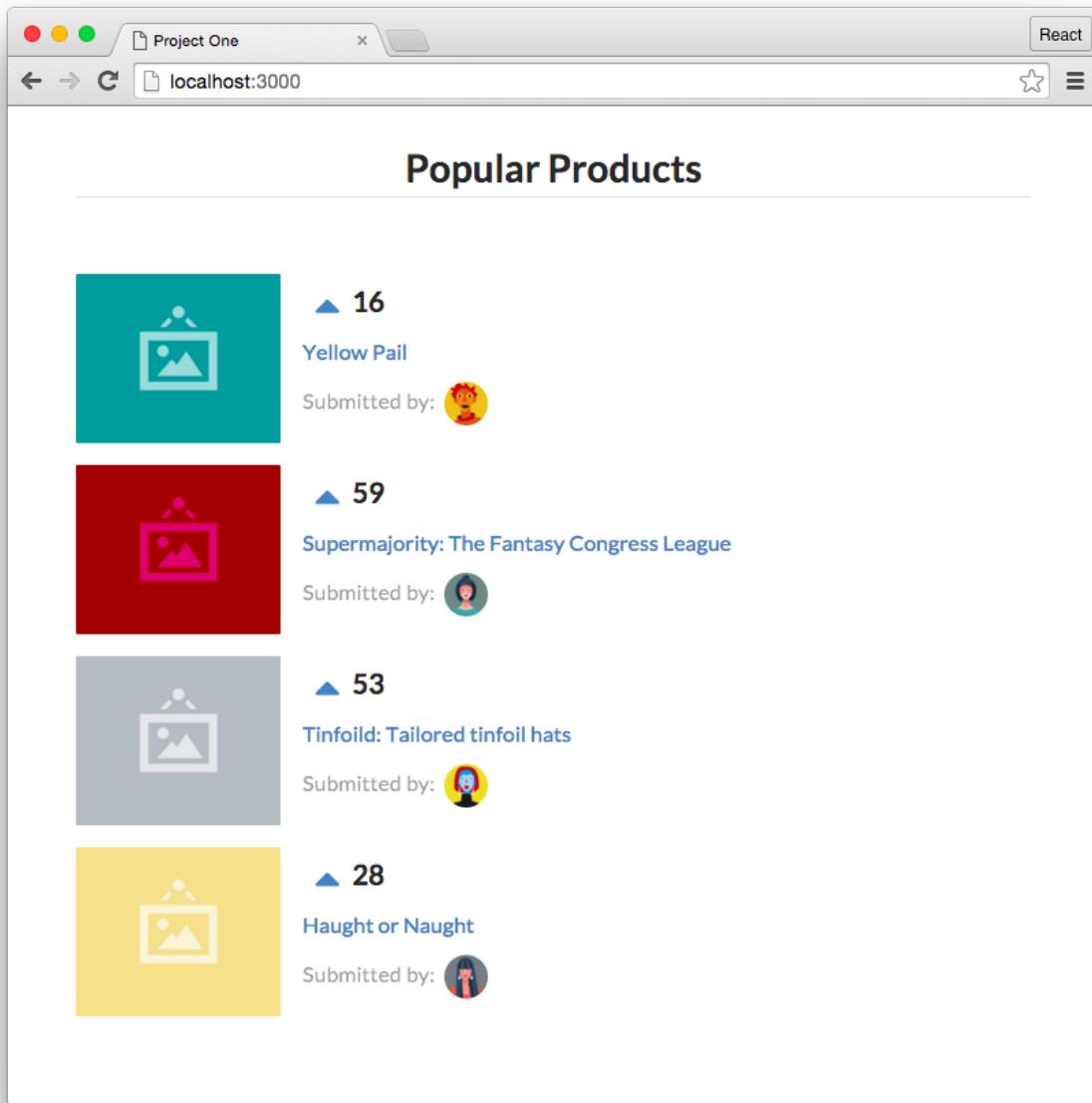
The first is that the syntax is much terser. Compare this declaration:

```
const timestamps = messages.map(function(m) { return m.timestamp });
```

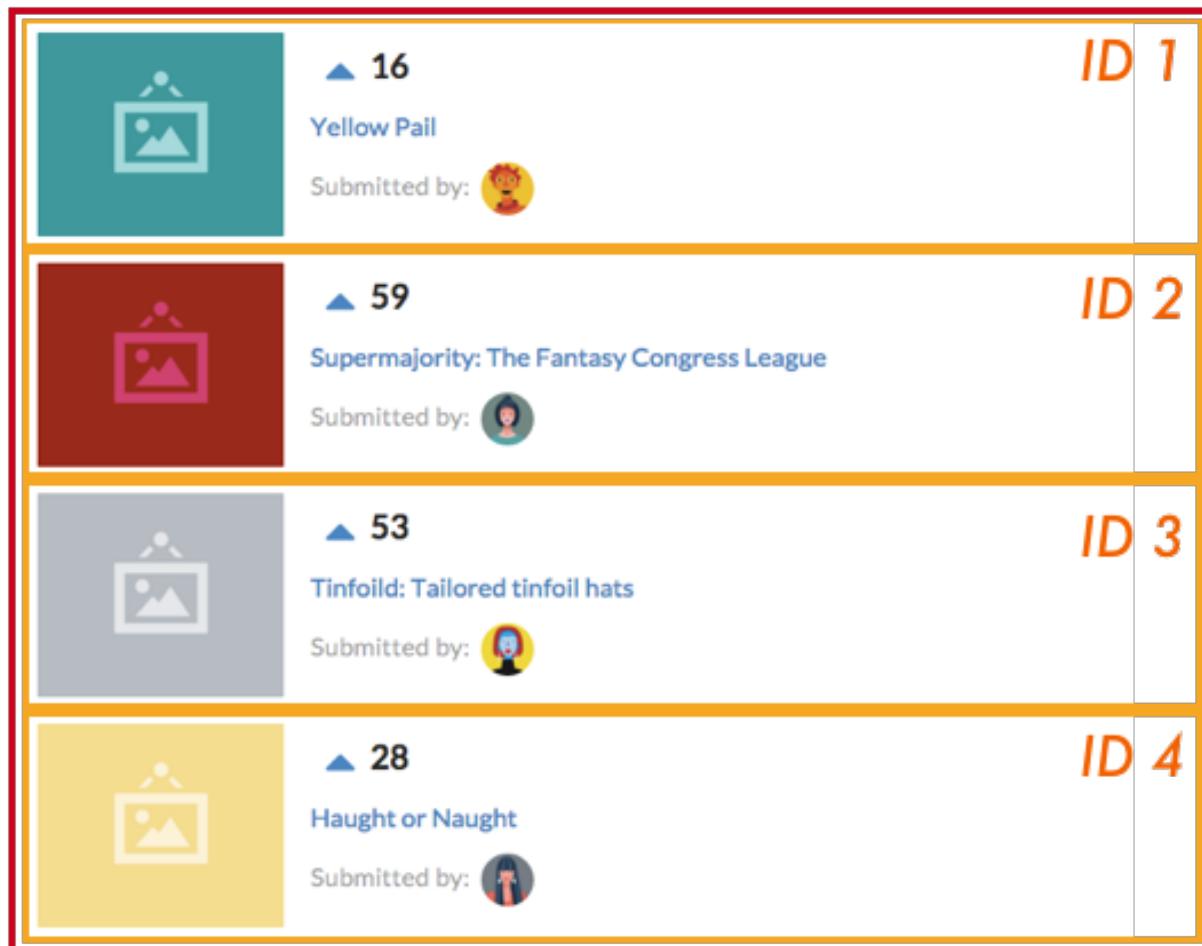
To this one:

```
const timestamps = messages.map(m => m.timestamp);
```

Of greater benefit, though, is how arrow functions bind the `this` object. We cover the semantics of this difference later in the sidebar titled "[Arrow functions and this](#)."



After we save the updates to our `app.js` and refresh the page, we'll see that we have five total React components at work. We have a single parent component, our `ProductList` component which contains four child `Product` components, one for each product object in our `Data` variable (from `data.js`).



Product components (orange) inside of the `ProductList` component (red)

We added an ‘up vote’ caret icon in our `Product` component above. If we click on one of these buttons, we’ll see that nothing happens. We’ve yet to hook up an event to the button.

Although we have a dynamic React app running in our web browser, this page still lacks interactivity. While React has given us an easy and clean way to organize our HTML thus far and enabled us to drive HTML generation based on a flexible, dynamic JavaScript object, we’ve still yet to tap into its true power in enabling super dynamic, interactive interfaces.

The rest of this course digs deep into this power. Let’s start with something simple: the ability to up-vote a given product.

React the vote (your app's first interaction)

When the up-vote button on each one of the Product components is clicked, we expect it to update the votes attribute for that Product, increasing it by one. In addition, this vote should be updated in the Data variable. While Data is just a JavaScript variable right now, in the future it could just as easily be a remote database. We would need to inform the remote database of the change. While we cover this process in-depth in later sections of this course, we'll get used to this practice by updating Data.

The Product component can't modify its votes. `this.props` is immutable.

Remember, while the child has access to *read* its own props, it doesn't own them. In our app, the parent component, ProductList, owns them. React favors the idea of *one-way data flow*. This means that data changes come from the "top" of the app and are propagated "downwards" through its various components.



A child component does not own its props. Parent components own the props of the child component.

We need a way for the Product component to let ProductList know that a click on its up-vote event happened and then let ProductList, the owner of both that product's props as well as the store (Data), handle the rest. It can update Data and then data will flow downward from Data, through the ProductList component, and finally to the Product component.

Propagating the event

Fortunately, propagating an event from a child component to a parent is easy. We can pass down *functions* as props too. We can have the ProductList component give each Product component a function to call when the up-vote button is clicked. Functions passed down through props are the canonical manner in which children communicate events with their parent components.

Let's start by modifying Product to call a function when the up-vote button is clicked.

First, add a new component function to Product, `handleUpVote()`. We'll anticipate the existence of a new prop called `onVote()` that's a function passed down by ProductList:

```
const Product = React.createClass({
  handleUpVote: function () {
    this.props.onVote(this.props.id);
  },
  render: function () {
    // ...
  }
});
```

We're setting up an expectation that the `ProductList` component sends a function as a prop to the `Product` component. We'll call this function `onVote()`. `onVote()` accepts a single argument. The argument we're passing in is the `id` of the product (`this.props.id`). The `id` is sufficient information for the parent to deduce which `Product` component has produced this event. We'll implement the function shortly.

We can have an HTML element inside the `Product` component call a function when it is clicked. We'll set the `onClick` attribute on the `a` HTML tag that is the up-vote button. By passing the name of the function `handleUpVote()` to this attribute, it will call our new `handleUpVote()` function when the up-vote button is clicked:

```
const Product = React.createClass({
  handleUpVote: function () {
    this.props.onVote(this.props.id);
  },
  render: function () {
    return (
      // ...
      <div className='header'>
        <a onClick={this.handleUpVote}>
          <i className='large caret up icon'></i>
        </a>
        {this.props.votes}
      </div>
      // ...
    );
  }
});
```

Let's define the function in `ProductList` that we pass down to `Product` as the prop `onVote()`. This is the function that the `Product` component's `handleUpVote()` calls whenever the up-vote button is clicked:

```
const ProductList = React.createClass({
  handleProductUpVote: function (productId) {
    console.log(productId + " was upvoted.");
  },
  render: function () {
    const products = Data.map((product) => {
      return (
        <Product
          key={'product-' + product.id}
          id={product.id}
          title={product.title}
          description={product.description}
          url={product.url}
        >
      );
    });
    return (
      <div>
        <h1>Product List</h1>
        <ul>
          {products}
        </ul>
      </div>
    );
  }
});
```

```
    votes={product.votes}
    submitter_avatar_url={product.submitter_avatar_url}
    product_image_url={product.product_image_url}
    onVote={this.handleProductUpVote}
  />
);
});
return (
  <div className='ui items'>
    {products}
  </div>
);
},
));

```

First we define a `handleProductUpVote()` function on `ProductList`. We've set it up to accept a `productId`, as anticipated.

This function is then passed down as a prop, `onVote`, just the same as any other prop.

ES6: Arrow functions and `this`

Inside `ProductList`, we use array's `map()` method on `Data` to setup the variable `products`. We pass an anonymous arrow function to `map()`. Inside this arrow function, we call `this.handleProductUpVote`. Here, `this` is bound to the React object.

We introduced arrow functions earlier and mentioned that one of their benefits was how they bind the `this` object.

The traditional JavaScript function declaration syntax (`function () {}`) will bind `this` in anonymous functions to the global object. To illustrate the confusion this causes, consider the following example:

```
function printSong() {
  console.log("Oops - The Global Object");
}

const jukebox = {
  songs: [
    {
      title: "Wanna Be Startin' Somethin'",
      artist: "Michael Jackson",
    },
    {

```

```
        title: "Superstar",
        artist: "Madonna",
    },
],
printSong: function (song) {
    console.log(song.title + " - " + song.artist);
},
printSongs: function () {
    // `this` bound to the object (OK)
    this.songs.forEach(function (song) {
        // `this` bound to global object (bad)
        this.printSong(song);
    });
},
}

jukebox.printSongs();
// > "Oops - The Global Context"
// > "Oops - The Global Context"
```

The method `printSongs()` iterates over `this.songs` with `forEach()`. In this context, `this` is bound to the object (`jukebox`) as expected. However, the anonymous function passed to `forEach()` binds its internal `this` to the global object. As such, `this.printSong(song)` calls the function declared at the top of the example, *not* the method on `jukebox`.

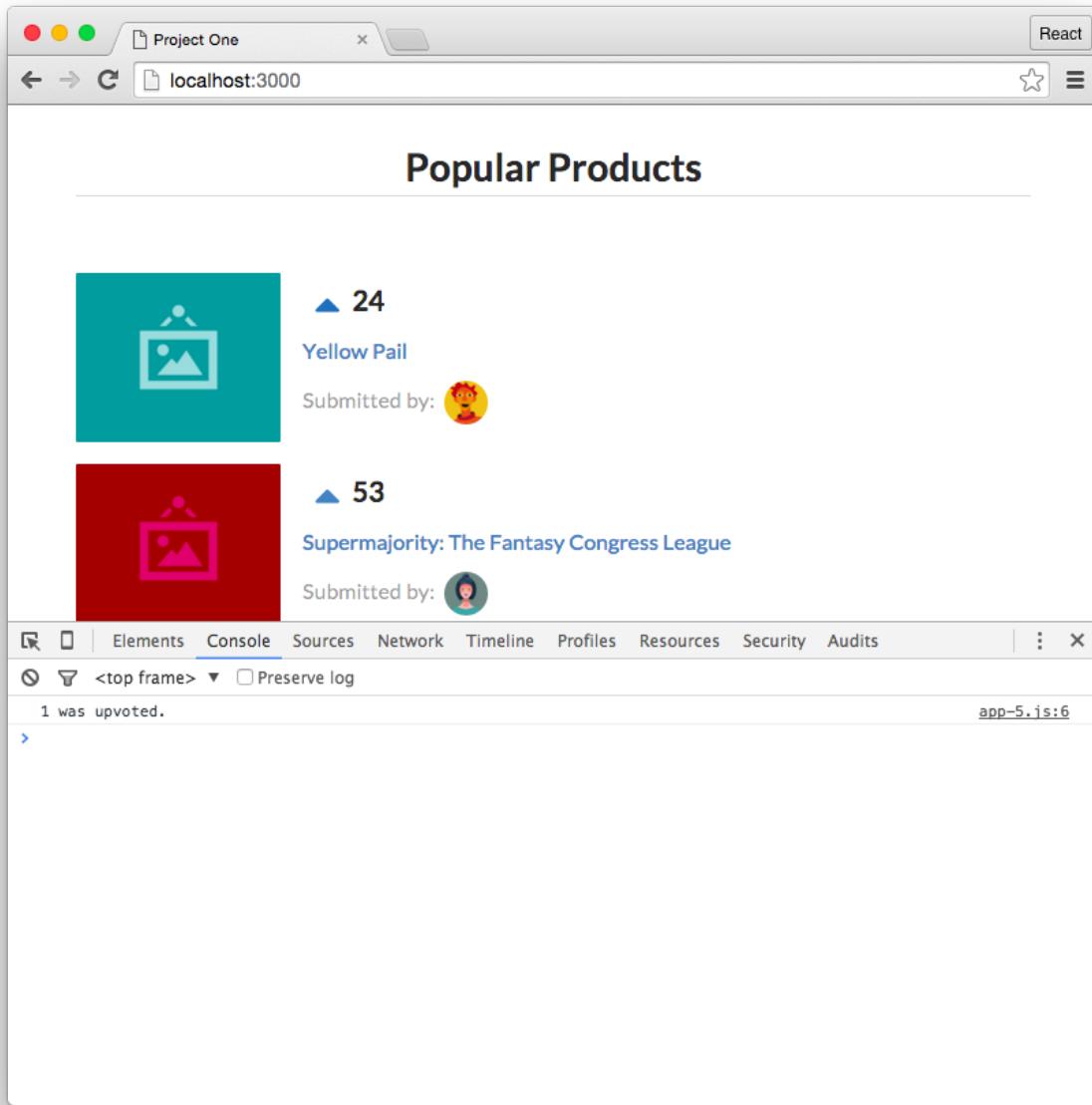
JavaScript developers have traditionally used workarounds for this behavior, but arrow functions solve the problem by **capturing the `this` value of the enclosing context**. Using an arrow function for `printSongs()` has the expected result:

```
// ...
printSongs: function () {
    this.songs.forEach((song) => {
        // `this` bound to same `this` as `printSongs()` (`jukebox`)
        this.printSong(song);
    });
},
}

jukebox.printSongs();
// > "Wanna Be Startin' Somethin' - Michael Jackson"
// > "Superstar - Madonna"
```

For this reason, throughout the book we will use arrow functions for all anonymous functions.

Saving our updated `app.js`, refreshing our web browser, and clicking an up-vote will log some text to our JavaScript console:



The events are being propagated up to the parent. Finally, we need `ProductList` to update the store, `Data`.

It's tempting to modify the `Data` JavaScript object directly. We could hunt through `Data` until we find the corresponding product and then update its vote count. However, React will have no idea this change occurred. So while the vote count will be updated in the store, this update will not be reflected to the user. This would be equivalent to just making a call to a server to update the vote

count but then doing nothing else. The front-end would be none the wiser.

In order to move forward, there's one more critical concept for React components we need to cover: state.

Using state

Whereas props are immutable and owned by a component's parent, state is mutable and owned by the component. `this.state` is private to the component and can be updated with `this.setState()`. As with props, when the state updates the component will re-render itself.

Every React component is rendered as a function of its `this.props` and `this.state`. This rendering is deterministic. This means that given a set of props and a set of state, a React component will always render a single way. As we mentioned earlier, this approach makes for a powerful UI consistency guarantee.

As we are mutating the data for our products (the number of votes), we should consider this data to be stateful. We should treat `Data` as some external store, which is used to update `this.state` for `ProductList`.

Let's modify the `ProductList` component's `render` function so that it uses state as opposed to accessing `Data` directly. In order to tell React that our component is *stateful*, we'll define the function `getInitialState()` and return a non-falsey value:

```
const ProductList = React.createClass({
  getInitialState: function () {
    return {
      products: [],
    };
  },
  handleProductUpVote: function (productId) {
    console.log(productId + " was upvoted.");
  },
  render: function () {
    const products = this.state.products.map((product) => {
      return (
        <Product
        // ...
      )
    });
    return (
      <div>
        <h1>Product List</h1>
        <ul>
          {products}
        </ul>
      </div>
    );
  }
});
```

Like `render()`, `getInitialState()` is a special method on a React component. It is one of several lifecycle methods available. It is executed exactly once during the component lifecycle and defines the initial state of the component.

Now, instead of mapping over `Data` to produce the variable `products`, we are reading from `this.state`. In `getInitialState()`, `this.state` is initialized to the JavaScript object:

```
{  
  products: [],  
}
```

But is never actually updated to anything meaningful. Indeed, if we were to save and refresh now, all of our products would be missing again. We need to update the state using `Data`.



We cover all of the component lifecycle methods in-depth in a later section.

Setting state with `this.setState()`

Let's use another lifecycle method, `componentDidMount()`, to set the state for `ProductList` to `Data`:

```
const ProductList = React.createClass({  
  getInitialState: function () {  
    return {  
      products: [],  
    };  
  },  
  componentDidMount: function () {  
    const products = Data.sort((a, b) => {  
      return b.votes - a.votes;  
    });  
    this.setState({ products: products });  
  },  
  handleProductUpVote: function (productId) {  
    console.log(productId + " was upvoted.");  
  },  
  render: function () {  
    // ...
```

In our `ProductList` component, the `componentDidMount()` function uses the native JavaScript Array's `sort()` method to ensure that we sort products based upon the number of votes in descending order. This sorted array of products is then used in the special component method `this.setState()`. As anticipated, this method updates `this.state` and re-renders the component.

As we'll need to update and sort the state after we click on an up-vote button, we can define this functionality as a single function so we don't need to duplicate this functionality. We'll call this function `updateState()`:

```
const ProductList = React.createClass({
  getInitialState: function () {
    return {
      products: [],
    };
  },
  componentDidMount: function () {
    this.updateState();
  },
  updateState: function () {
    const products = Data.sort((a, b) => {
      return b.votes - a.votes;
    });
    this.setState({ products: products });
  },
  handleProductUpVote: function (productId) {
    console.log(productId + " was upvoted.");
  },
  render: function () {
    // ...
  }
});
```



Never modify state outside of `this.setState()`. This function has important hooks around state modification that we would be bypassing.

We discuss state management in detail throughout the book.



Array's `sort()` method takes an optional function as an argument. If the function is omitted, it will just sort the array by each item's Unicode code point value. This is rarely what a programmer desires. If the function is supplied, elements are sorted according to the functions return value.

On each iteration, the arguments `a` and `b` are two elements in the array. Sorting depends on the return value of the function:

1. If the return value is less than `0`, `a` should come first (have a lower index).
2. If the return value is greater than `0`, `b` should come first.
3. If the return value is equal to `0`, leave order of `a` and `b` unchanged with respect to each other.



`sort()` mutates the original array it was called on. Later on in the course, we discuss why mutating arrays or objects can be a dangerous pattern.

If we save and refresh now, we see that the products are back.

Updating state

With state management in place, we need to modify `handleProductUpVote()` inside `ProductList`. When `handleProductUpVote()` is invoked from inside the `Product` component, it should update `Data` and then trigger a state update for `ProductList`:

```
const ProductList = React.createClass({
  // ...
  handleProductUpVote: function (productId) {
    Data.forEach((el) => {
      if (el.id === productId) {
        el.votes = el.votes + 1;
        return;
      }
    });
    this.updateState();
  },
  // ...
});
```

We use `Array`'s `forEach()` to traverse `Data`. When a matching object is found (based upon the object's `id`), its `votes` attribute is incremented by 1. After `Data` is modified, we'll call `this.updateState()` to update the state to the current `products` value. The component's state is then updated and React intelligently re-renders the UI to reflect these updates.

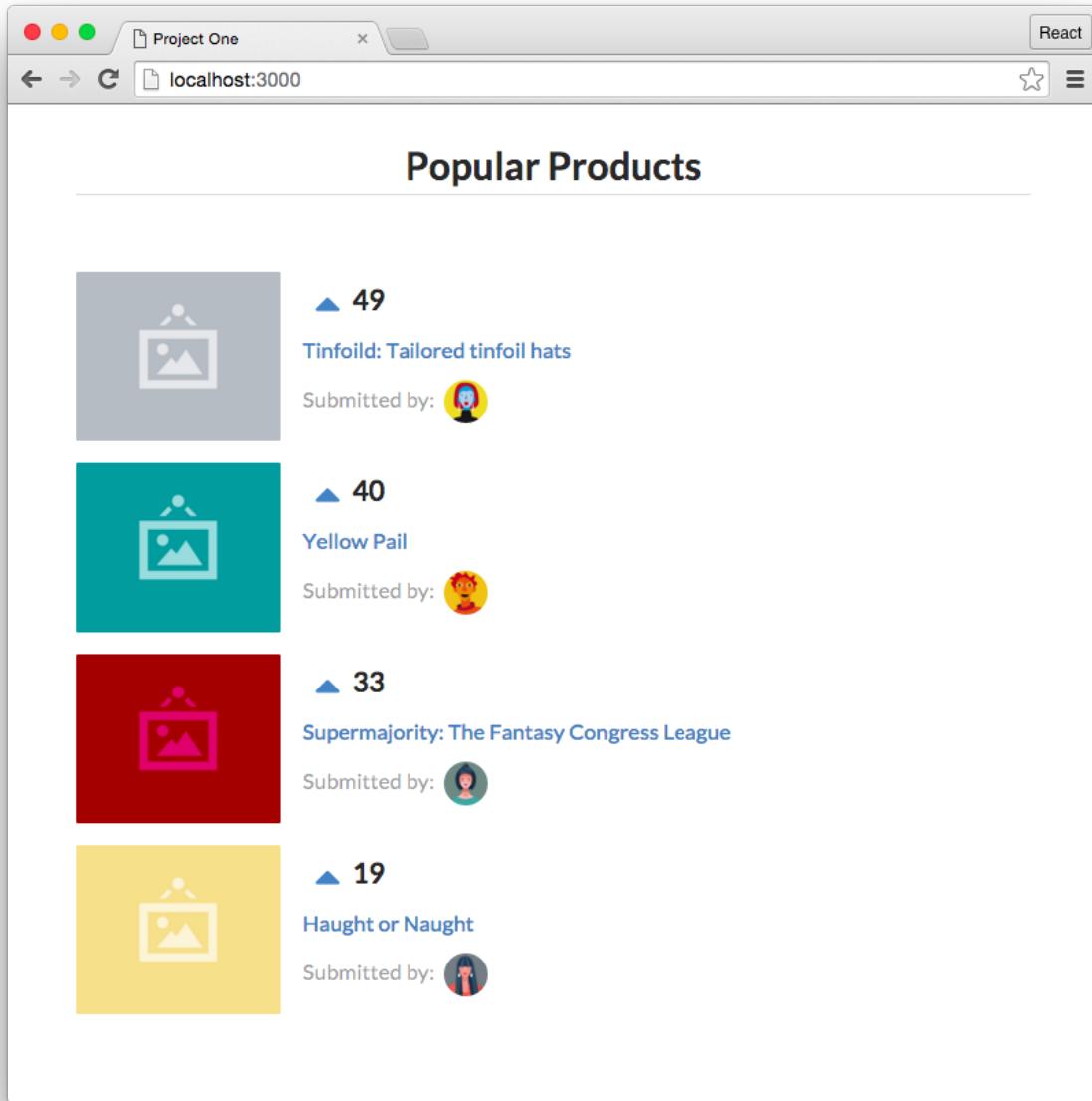


Array's `forEach()` method executes the provided function once for each element present in the array in ascending order.



Handling these updates on a remote store would be a fairly trivial update. Instead of modifying `Data` inside of `handleProductUpVote()`, we would make a call to a remote service. The server's response would trigger a callback that would then make a subsequent call to the service asking for the most recent data, updating the state with the data from the response. We'll be exploring server communication in this course's next project.

Save `app.js`, refresh the browser, and cross your fingers:



At last, the vote counters are working! Try up-voting a product a bunch of times and notice how it quickly jumps above products with lower vote counts.



Technically, we can perform the same operation in `this.setState()` within `getInitialState()` as well, bypassing the brief period where the state is empty. However, this is usually bad practice for a variety of reasons.

We look into why in our advanced components section, but briefly, our state will be set by a call to a remote server. As React is still performing the initial render of the app, we'd have to halt it in its tracks, block on a web request, and then update the state when the result is returned. Instead, by just giving React a valid blank "scaffold" for the state, React will render everything first then make parallel, asynchronous calls to whichever servers to fill in the details. Much more efficient.



Again, the style of this app differs slightly from the demo we saw at the beginning of this section. If you'd like to add some additional style to your components, refer to the HTML structure in `app-complete.js`.

Congratulations!

We have just completed our first React app. Not so bad, eh?

There are a ton of powerful features we've yet to go over, yet all of them build upon the core fundamentals we just covered:

1. Think about and organize your app into components
2. JSX and the `render` method
3. Data flow from parent to children through props
4. Event flow from children to parent through functions
5. State vs props
6. How to manipulate state
7. Utilizing React lifecycle methods

Onward!



Chapter Exercises

1. Add down-voting capability to each Product. You can insert a down arrow with this JSX snippet:

```
<i className='large caret down icon'></i>
```

2. Add a "sort direction" button to the top of `ProductList`, above all the products. It should enable the user to toggle sorting products by ascending or descending.

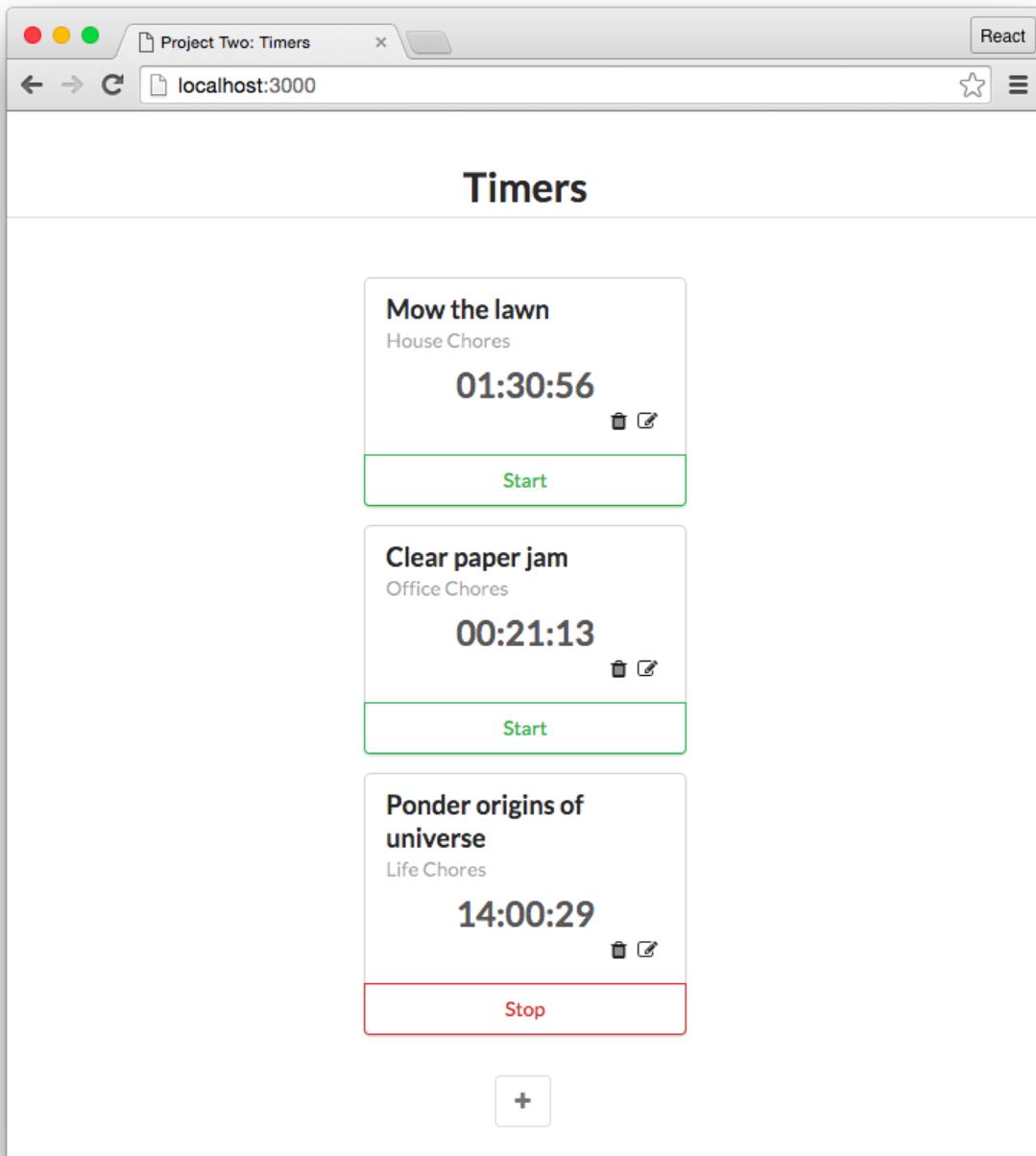
Components

A time-logging app

In the last chapter, we described how React organizes apps into components and how data flows between parent and child components. And we discussed core concepts such as how we manage **state** and pass data between components using **props**.

In this chapter, we construct a more intricate application. We investigate a pattern that we will use to build all React apps and then put those steps to work to build an interface for managing timers.

In this time-tracking app, a user can add, delete, and modify various timers. Each timer corresponds to a different task that the user would like to keep time for:



This app will have significantly more interactive capabilities than the one built in the last chapter. This will present us with some interesting challenges that will deepen our familiarity with React's core concepts.

Getting started



Sample Code

As with all chapters, before beginning make sure you've downloaded the book's sample code and have it at the ready.

Previewing the app

Let's begin by playing around with a completed implementation of the app.

In your terminal, cd into the `time_tracking_app` directory:

```
$ cd time_tracking_app
```

Use NPM to install all the dependencies:

```
$ npm install
```

Then boot the server:

```
$ npm run server
```

Now you can view the app in your browser. Open your browser and enter the URL <http://localhost:3000>¹⁹.

Play around with it for a few minutes to get a feel for all the functionality. Refresh and note that your changes have been persisted.

Prepare the app

In your terminal, run `ls -lp` to see the project's layout:

```
$ ls -lp
```

¹⁹<http://localhost:3000>

```
README.md  
data.json  
node_modules/  
package.json  
public/  
server.js
```

There are a few organizational changes from the last project.

First, notice that there is now a `server.js` in this project. In the last chapter, we used a pre-built Node package (called `http-server`) to serve our assets.

This time we have a custom-built server which serves our assets and also adds a persistence layer. We will cover the server in detail in the next chapter.



When you visit a website, **assets** are the files that your browser downloads and uses to display the page. `index.html` is delivered to the browser and inside its `head` tags it specifies which additional files from the server the browser needs to download.

In the last project, our assets were `index.html` as well as our stylesheets and images.

In this project, everything under `public/` is an asset.

In the voting app, we loaded all of our app's initial data from a JavaScript variable, loaded in the file `data.js`.

This time, we're going to eventually store it in the text file `data.json`. This brings the behavior a bit closer to a database. By using a JSON file, we can make edits to our data that will be persisted even if the app is closed.



JSON stands for JavaScript Object Notation. JSON enables us to serialize a JavaScript object and read/write it from a text file.

If you're not familiar with JSON, take a look at `data.json`. Pretty recognizable, right? JavaScript has a built-in mechanism to parse the contents of this file and initialize a JavaScript object with its data.

Peek inside `public`:

```
$ cd public  
$ ls -lp
```

And you will find some familiar files:

```
vendor/  
app-complete.js  
app.js  
client.js  
helpers.js  
index.html  
style.css
```

Again, we'll be editing only the files `index.html` and `app.js`. `index.html` is the actual webpage which loads our React app which is written inside of `app.js`. `helpers.js` and `client.js` contain some pre-written JavaScript functions that we will use in our app.

As before, our first step is to ensure `app-complete.js` is no longer loaded in `index.html`. We'll instead load the empty file `app.js`.

Open up `index.html`. It looks like this:

```
<!DOCTYPE html>  
<html>  
  
<head>  
  <meta charset="utf-8">  
  <!-- Disable browser cache -->  
  <meta http-equiv="cache-control" content="max-age=0" />  
  <meta http-equiv="cache-control" content="no-cache" />  
  <meta http-equiv="expires" content="0" />  
  <meta http-equiv="expires" content="Tue, 01 Jan 1980 1:00:00 GMT" />  
  <meta http-equiv="pragma" content="no-cache" />  
  <title>Project Two: Timers</title>  
  <link rel="stylesheet" href="vendor/semantic-ui/semantic.min.css" />  
  <link rel="stylesheet" href="style.css" />  
  <script src="vendor/babel-core-5.8.25.js"></script>  
  <script src="vendor/react.js"></script>  
  <script src="vendor/react-dom.js"></script>  
  <script src="vendor/uuid.js"></script>  
  <script src="vendor/fetch.js"></script>  
</head>  
  
<body>  
  <div id="main" class="main ui">  
    <h1 class="ui dividing centered header">Timers</h1>  
    <div id="content"></div>  
  </div>
```

```
<script type="text/babel" src="./client.js"></script>
<script type="text/babel" src="./helpers.js"></script>
<script type="text/babel" src="./app.js"></script>
<!-- Delete the line below to get started. -->
<script type="text/babel" src="./app-complete.js"></script>
</body>

</html>
```

Overall, this file is very similar to the one we used in our voting app. We load in our dependencies within the head tags (the assets). Inside of body we have a few elements. This div is where we will ultimately mount our React app:

```
<div id="content"></div>
```

And this script tag is where we instruct the browser to load app.js into the page:

```
<script type="text/babel" src="./app.js"></script>
```

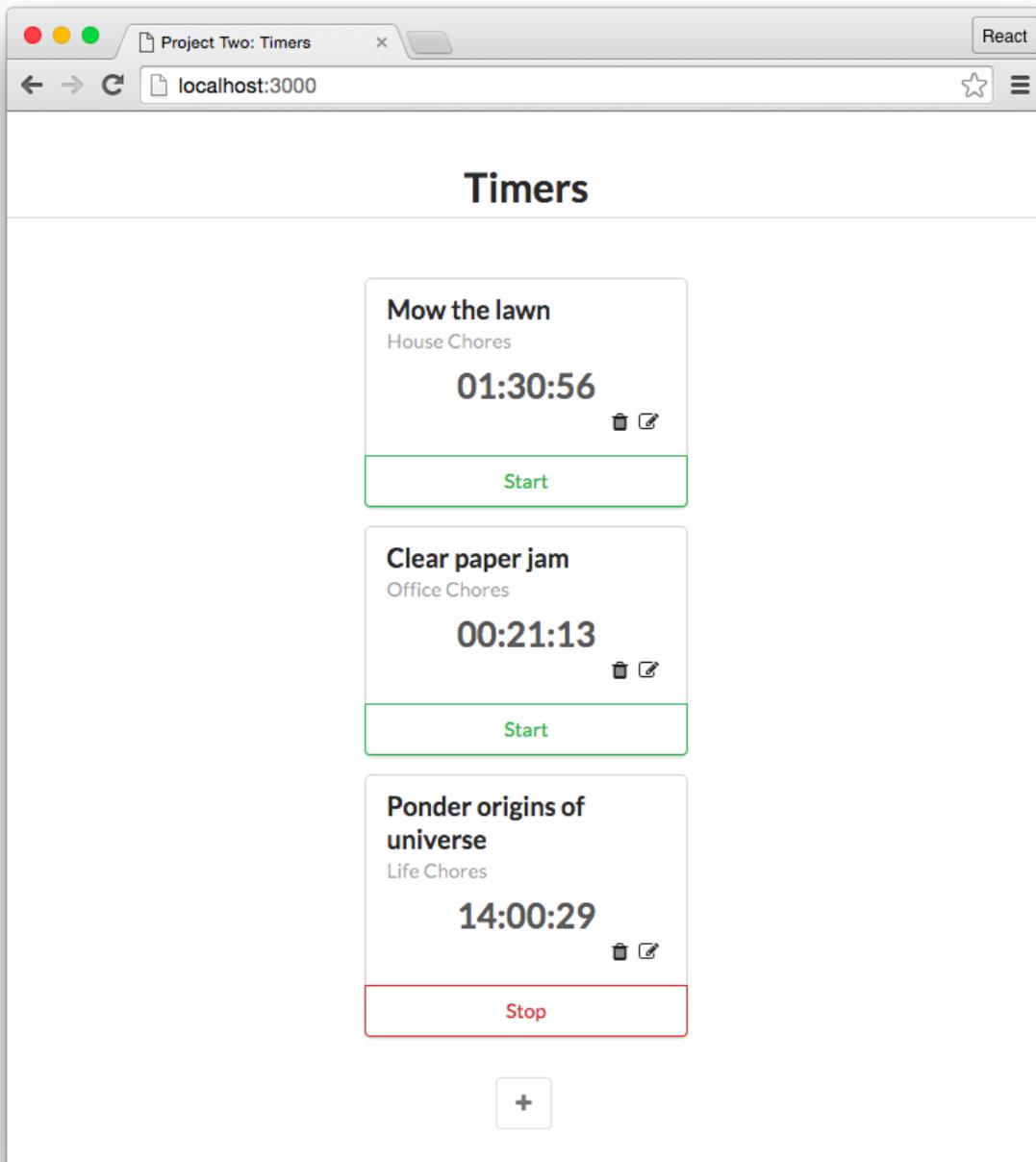
Do as the comment says and delete the line that loads app-complete.js:

```
<script type="text/babel" src="./app.js"></script>
<!-- Delete the line below to get started. -->
<script type="text/babel" src="./app-complete.js"></script>
```

Save index.html. If you reload the page now, you'll see the app has disappeared.

Breaking the app into components

As we did with our last project, we begin by breaking our app down into its components. Again, visual components often map tightly to their respective React components. Let's examine the interface of our app:



In the last project, we had `ProductList` and `Product` components. The first contained instances of the second. Here, we spot the same pattern, this time with `TimerList` and `Timer` components:

TimerList

Mow the lawn *House Chores*

01:30:56

Start

Clear paper jam *Office Chores*

00:21:13

Start

Ponder origins of universe *Life Chores*

14:00:29

Stop

+

However, there's one subtle difference: This list of timers has a little "+" icon at the bottom. As we saw, we're able to add new timers to the list using this button. So, in reality, the `TimerList` component isn't just a list of timers. It also contains a widget to create new timers.

Think about components as you would functions or objects. The [single responsibility principle²⁰](#) applies. A component should, ideally, **only be responsible for one piece of functionality**. So, the proper response here is for us to shrink `TimerList` back into its responsibility of just listing timers and to nest it under a parent component. We'll call the parent `TimersDashboard`. `TimersDashboard` will have `TimerList` and the "+"/create form widget as children:

²⁰https://en.wikipedia.org/wiki/Single_responsibility_principle

TimersDashboard

TimerList

Mow the lawn **Timer**
House Chores
01:30:56

Start

Clear paper jam **Timer**
Office Chores
00:21:13

Start

Ponder origins of universe **Timer**
Life Chores
14:00:29

Stop

Not only does this separation of responsibilities keep components simple, but it often also improves their re-usability. In the future, we can now drop the `TimerList` component anywhere in the app where we just want to display a list of timers. This component no longer carries the responsibility of also creating timers, which might be a behavior we want to have for just this dashboard view.



How you name your components is indeed up to you, but having some consistent rules around language as we do here will greatly improve code clarity.

In this case, developers can quickly reason that any component they come across that ends in the word `List` simply renders a list of children and no more.

The “+”/create form widget is interesting because it has two distinct representations. When the “+” button is clicked, the widget transmutes into a form. When the form is closed, the widget transmutes back into a “+” button.

There are two approaches we could take. The first one is to have the parent component, `TimersDashboard`, decide whether or not to render a “+” component or a form component based on some piece of stateful data. It could swap between the two children. However, this adds more responsibility to `TimersDashboard`. The alternative is to have a new child component own the single responsibility of determining whether or not to display a “+” button or a create timer form. We’ll call it `ToggleableTimerForm`. As a child, it can either render the component `TimerForm` or the HTML markup for the “+” button.

At this point, we’ve carved out four components:

TimersDashboard

TimerList

Mow the lawn **Timer**

House Chores

01:30:56



Start

Clear paper jam **Timer**

Office Chores

00:21:13



Start

Ponder origins of **Timer**
universe

Life Chores

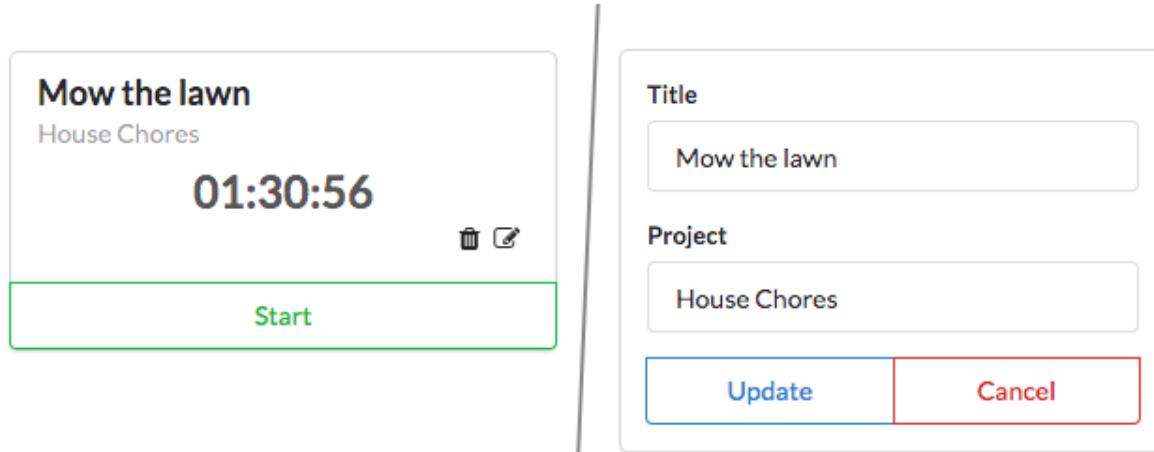
14:00:29



Stop



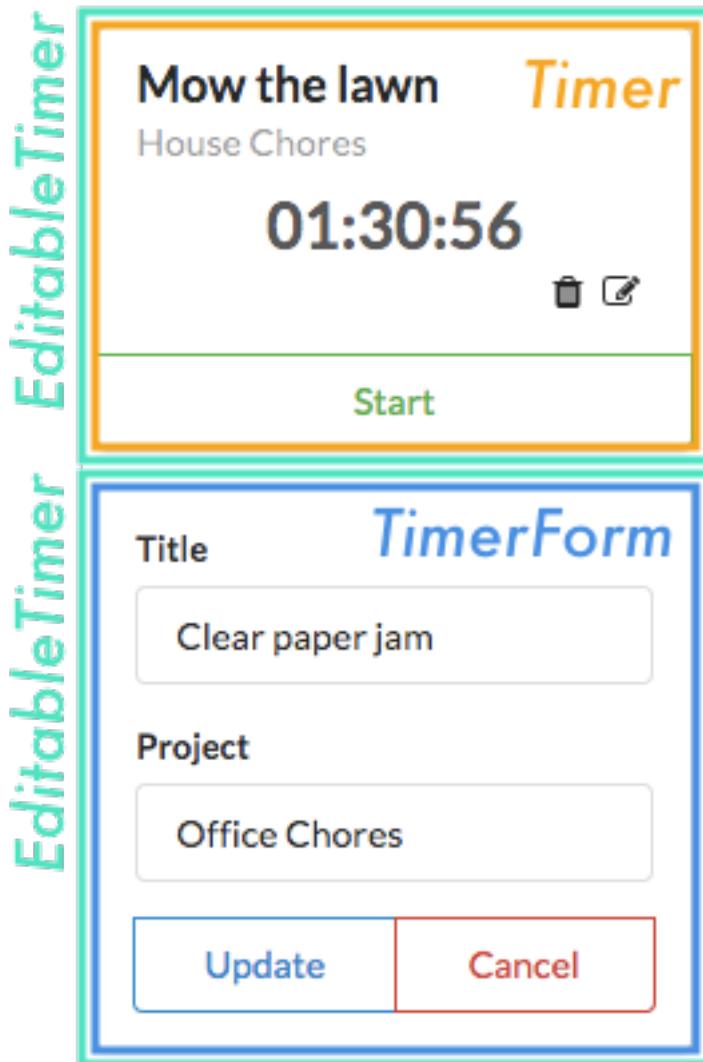
Now that we have a sharp eye for identifying overburdened components, another candidate should catch our eye:



The timer itself has a fair bit of functionality. It can transform into an edit form, delete itself, and start and stop itself. Do we need to break this up? And if so, how?

Displaying a timer and editing a timer are indeed two distinct UI elements. They should be two distinct React components. Like `ToggleableTimerForm`, we need some container component that renders either the timer's face or its edit form depending on if the timer is being edited.

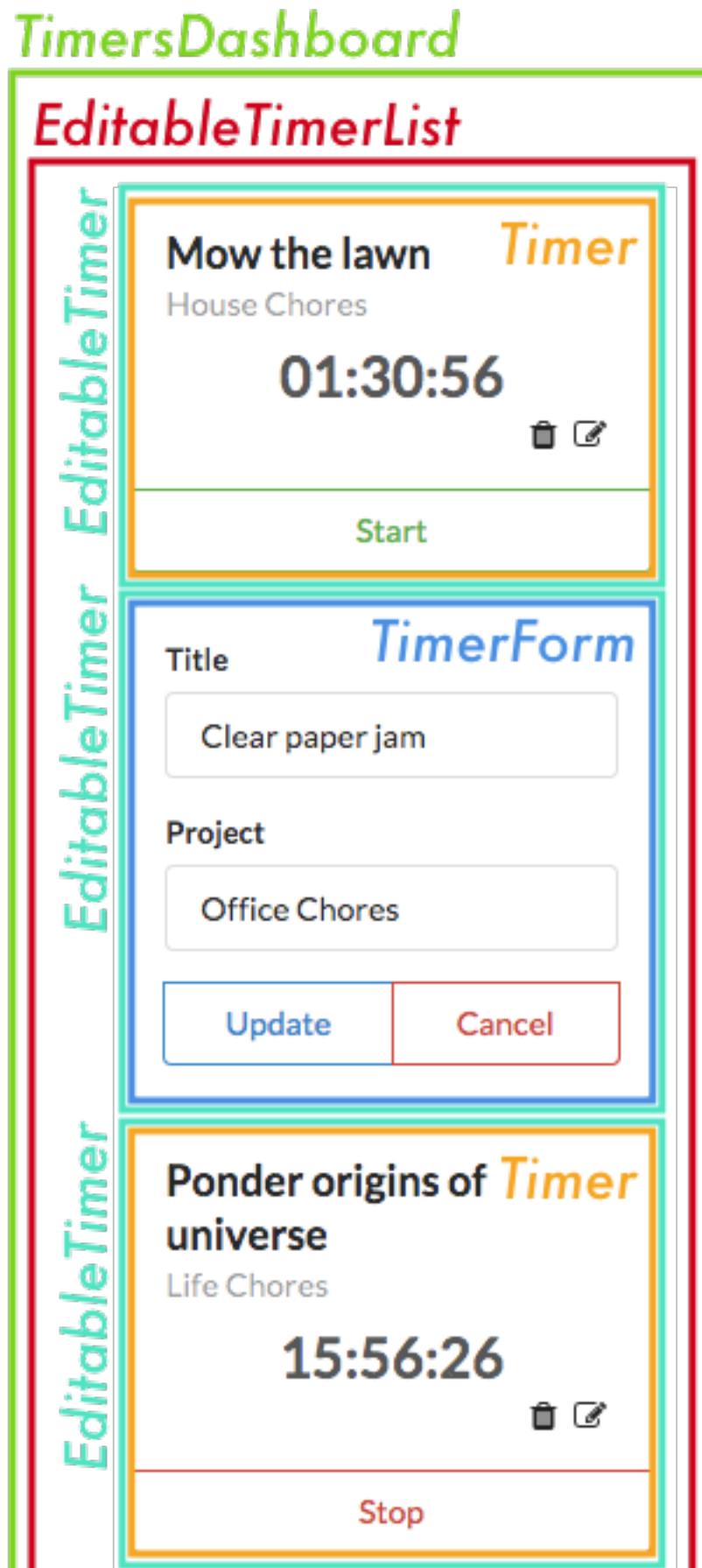
We'll call this `EditableTimer`. The child of `EditableTimer` will then be either a `Timer` component or the edit form component. The form for creating and editing timers is very similar, so let's assume that we can use the component `TimerForm` in both contexts:



As for the other functionality of the timer, like the start and stop buttons, it's a bit tough to determine at this point whether or not they should be their own components. We can trust that the answers will be more apparent after we've written some code.

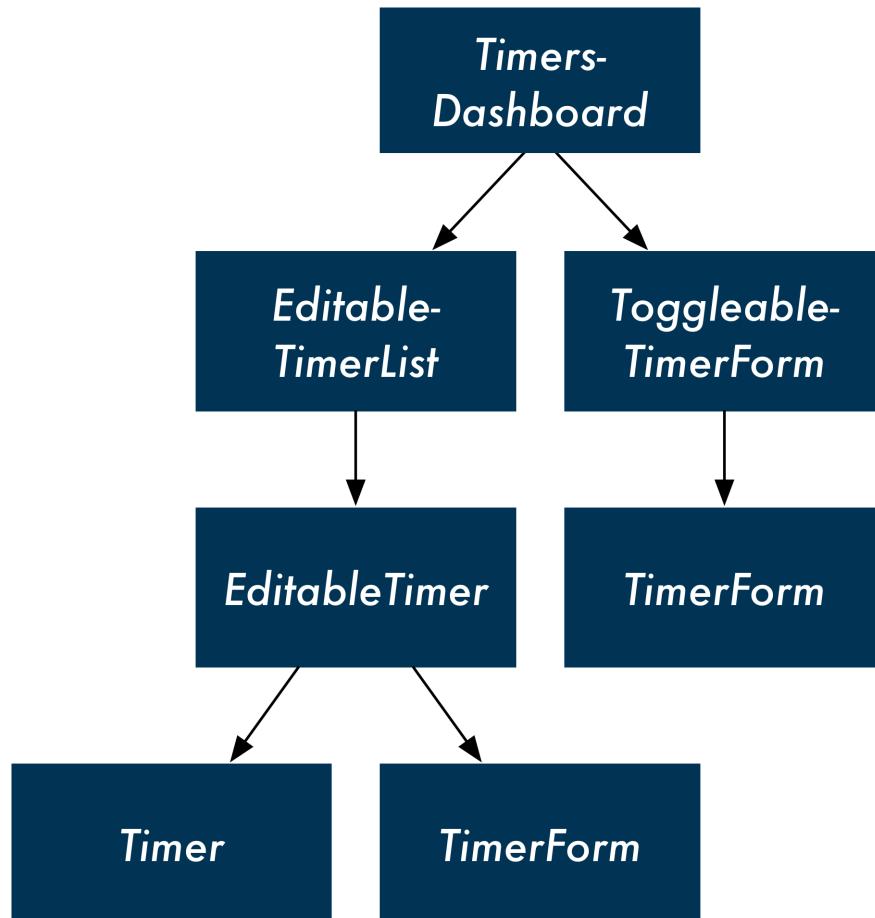
Working back up the component tree, we can see that the name `TimerList` would be a misnomer. It really is a `EditbleTimerList`. Everything else looks good.

So, we have our final component hierarchy, with some ambiguity around the final state of the timer component:



- **TimersDashboard** (green): Parent container
 - **EditableTimerList** (red): Displays a list of timer containers
 - * **EditableTimer** (aqua): Displays either a timer or a timer's edit form
 - **Timer** (yellow): Displays a given timer
 - **TimerForm** (blue): Displays a given timer's edit form
 - **ToggleableTimerForm** (purple): Displays a form to create a new timer
 - * **TimerForm** (not displayed): Displays a new timer's create form

Represented as a hierarchical tree:





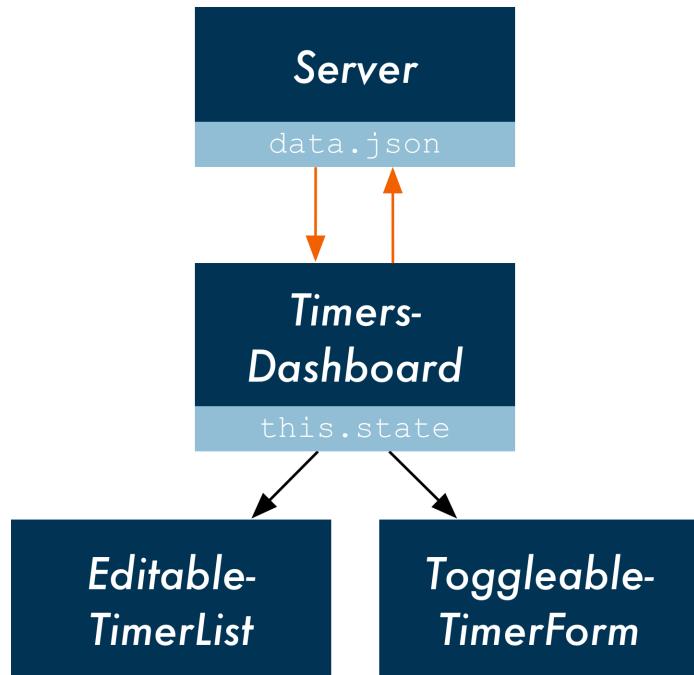
In our previous app, `ProductList` handled not only rendering components, but also the responsibility of handling up-votes and talking to the store. While this worked for that app, you can imagine that as a codebase expands, there may come a day where we'd want to free `ProductList` of this responsibility.

For example, imagine if we added a “sort by votes” feature to `ProductList`. What if we wanted some pages to be sortable (category pages) but other pages to be static (top 10)? We’d want to “hoist” sort responsibility up to a parent component and make `ProductList` the straightforward list renderer that it should be.

This new parent component could then include the sorting-widget component and then pass down the ordered products to the `ProductList` component.

The steps for building React apps

Now that we have a good understanding of the composition of our components, we’re ready to build a static version of our app. Ultimately, our top-level component will communicate with a server. **The server will be the initial source of state**, and React will render itself according to the data the server provides. Our app will also send updates to the server, like when a timer is started:



However, it will simplify things for us if we start off with static components, as we did in the last chapter. Our React components will do little more than render HTML. Clicking on buttons won’t yield any behavior as we will not have wired up any interactivity. This will enable us to lay the framework for the app, getting a clear idea of how the component tree is organized.

Next, we can determine what the **state** should be for the app and in which component it should live. We'll start off by just hard-coding the state into the components instead of loading it from the server.

At that point, we'll have the data flow **from parent to child** in place. Then we can add inverse data flow, propagating events **from child to parent**.

Finally, we'll modify the top-level component to have it communicate with the server.

In fact, the development of each React app will follow this pattern:

1. Break the app into components
2. Build a static version of the app
3. Determine what should be stateful
4. Determine in which component each piece of state should live
5. Hard-code initial states
6. Add inverse data flow
7. Add server communication

We actually followed this pattern in the last project:

1. Break the app into components

We looked at the desired UI and determined we wanted `ProductList` and `Product` components.

2. Build a static version of the app

Our components started off without using `state`. Instead, we had `ProductList` pass down static props to `Product`.

3. Determine what should be stateful

In order for our application to become interactive, we had to be able to modify the `vote` property on each product. Each product had to be mutable and therefore stateful.

4. Determine in which component each piece of state should live

`ProductList` managed the voting state using React component class methods.

5. Hard-code initial state

When we re-wrote `ProductList` to use `this.state`, we had it use the hard-coded variable `Data`.

6. Add inverse data flow

We defined the `handleUpVote` function in `ProductList` and passed it down in props so that each `Product` could inform `ProductList` of up-vote events.

7. Add server communication

We did not add a server component to our last app, but we will be doing so in this one.

If steps in this process aren't completely clear right now, don't worry. The purpose of this chapter is to familiarize yourself with this procedure.

We've already covered step (1) and have a good understanding of all of our components, save for some uncertainty down at the `Timer` component. Step (2) is to build a static version of the app. As in the last project, this amounts to defining React components, their hierarchy, and their HTML representation. We completely avoid state for now.

Step 2: Build a static version of the app

TimersDashboard

Let's start off with the `TimersDashboard` component. Again, all of our React code for this chapter will be inside of the file `public/app.js`.

We'll begin by defining a familiar function, `render()`:

```
const TimersDashboard = React.createClass({
  render: function () {
    return (
      <div className='ui three column centered grid'>
        <div className='column'>
          <EditableTimerList />
          <ToggleableTimerForm
            isOpen={true}>
          />
        </div>
      </div>
    );
  },
});
```

This component renders its two child components nested under `div` tags. `TimersDashboard` passes down one prop to `ToggleableTimerForm`: `isOpen`. This is used by the child component to determine whether to render a “+” or `TimerForm`. When `ToggleableTimerForm` is “open” the form is being displayed.



As in the last chapter, don't worry about the `className` attribute on the `div` tags. This will ultimately define the `class` on HTML `div` elements and is purely for styling purposes.

In this example, classes like `ui three column centered grid` all come from the CSS framework [Semantic UI²¹](#). The framework is included in the head of `index.html`.

²¹<http://semantic-ui.com>

We will define `EditableTimerList` next. We'll have it render two `EditableTimer` components. One will end up rendering a timer's face. The other will render a timer's edit form:

```
const EditableTimerList = React.createClass({
  render: function () {
    return (
      <div id='timers'>
        <EditableTimer
          title='Learn React'
          project='Web Domination'
          elapsed='8986300'
          runningSince={null}
          editFormOpen={false}>
        />
        <EditableTimer
          title='Learn extreme ironing'
          project='World Domination'
          elapsed='3890985'
          runningSince={null}
          editFormOpen={true}>
        />
      </div>
    );
  },
});
```

We're passing five props to each child component. The key difference between the two `EditableTimer` components is the value being set for `editFormOpen`. We'll use this boolean to instruct `EditableTimer` which sub-component to render.



The purpose of the prop `runningSince` will be covered later on in the app's development.

EditableTimer

`EditableTimer` returns either a `TimerForm` or a `Timer` based on the prop `editFormOpen`:

```
const EditableTimer = React.createClass({
  render: function () {
    if (this.props.editFormOpen) {
      return (
        <TimerForm
          title={this.props.title}
          project={this.props.project}
        />
      );
    } else {
      return (
        <Timer
          title={this.props.title}
          project={this.props.project}
          elapsed={this.props.elapsed}
          runningSince={this.props.runningSince}
        />
      );
    }
  },
});
```

Note that `title` and `project` are passed down as props to `TimerForm`. This will enable the component to fill in these fields with the timer's current values.

TimerForm

We'll build an HTML form that will have two input fields. The first input field is for the title and the second is for the project. It also has a pair of buttons at the bottom:

```
const TimerForm = React.createClass({
  render: function () {
    const submitText = this.props.title ? 'Update' : 'Create';
    return (
      <div className='ui centered card'>
        <div className='content'>
          <div className='ui form'>
            <div className='field'>
              <label>Title</label>
              <input type='text' defaultValue={this.props.title} />
            </div>
            <div className='field'>
```

```
<label>Project</label>
<input type='text' defaultValue={this.props.project} />
</div>
<div className='ui two bottom attached buttons'>
  <button className='ui basic blue button'>
    {submitText}
  </button>
  <button className='ui basic red button'>
    Cancel
  </button>
</div>
</div>
</div>
);
},
));
});
```

Look at the `input` tags. We're specifying that they have type of `text` and then we are using the React property `defaultValue`. When the form is used for editing as it is here, this sets the fields to the current values of the timer as desired.



Later, we'll use `TimerForm` again within `ToggleableTimerForm` for *creating* timers. `ToggleableTimerForm` will not pass `TimerForm` any props. `this.props.title` and `this.props.project` will therefore return `undefined` and the fields will be left empty.

At the beginning of `render()`, before the return statement, we define a variable `submitText`. This variable uses the presence of `this.props.title` to determine what text the submit button at the bottom of the form should display. If `title` is present, we know we're editing an existing timer, so it displays "Update." Otherwise, it displays "Create."

With all of this logic in place, `TimerForm` is prepared to render a form for creating a new timer or editing an existing one.



We used an expression with the **ternary operator** to set the value of `submitText`. The syntax is:

```
1 condition ? expression1 : expression2
```

If the condition is true, the operator returns the value of `expression1`. Otherwise, it returns the value of `expression2`. In our example, the variable `submitText` is set to the returned expression.

ToggleableTimerForm

Let's turn our attention next to `ToggleableTimerForm`. Recall that this is a wrapper component around `TimerForm`. It will display either a “+” or a `TimerForm`. Right now, it accepts a single prop, `isOpen`, from its parent that instructs its behavior:

```
const ToggleableTimerForm = React.createClass({
  render: function () {
    if (this.props.isOpen) {
      return (
        <TimerForm />
      );
    } else {
      return (
        <div className='ui basic content center aligned segment'>
          <button className='ui basic button icon'>
            <i className='plus icon'></i>
          </button>
        </div>
      );
    }
  },
});
```

As noted earlier, `TimerForm` does not receive any props from `ToggleableTimerForm`. As such, its title and project fields will be rendered empty.

The return statement under the `else` block is the markup to render a “+” button. You could make a case that this should be its own React component (say `PlusButton`) but at present we'll keep the code inside `ToggleableTimerForm`.

Timer

Time for the `Timer` component. Again, don't worry about all the `div` and `span` elements and `className` attributes. We've provided these for styling purposes:

```
const Timer = React.createClass({
  render: function () {
    const elapsedString = helpers.renderElapsedString(this.props.elapsed);
    return (
      <div className='ui centered card'>
        <div className='content'>
          <div className='header'>
            {this.props.title}
          </div>
          <div className='meta'>
            {this.props.project}
          </div>
          <div className='center aligned description'>
            <h2>
              {elapsedString}
            </h2>
          </div>
          <div className='extra content'>
            <span className='right floated edit icon'>
              <i className='edit icon'></i>
            </span>
            <span className='right floated trash icon'>
              <i className='trash icon'></i>
            </span>
          </div>
        </div>
        <div className='ui bottom attached blue basic button'>
          Start
        </div>
      </div>
    );
  },
});
```

elapsed in this app is in milliseconds. This is the representation of the data that React will keep. This is a good representation for machines, but we want to show our carbon-based users a more readable format.

We use a function defined in `helpers.js`, `renderElapsedString()`. You can pop open that file if you're curious about how it's implemented. The string it renders is in the format 'HH:MM:SS'.



Note that we could store `elapsed` in seconds as opposed to milliseconds, but JavaScript's time functionality is all in milliseconds. We keep `elapsed` consistent with this for simplicity. As a bonus, our timers are also slightly more accurate, even though they round to seconds when displayed to the user.

Render the app

With all of the components defined, the last step before we can view our static app is to ensure we call `ReactDOM#render()`. Put this at the bottom of the file:

```
ReactDOM.render(  
  <TimersDashboard />,  
  document.getElementById('content')  
)
```



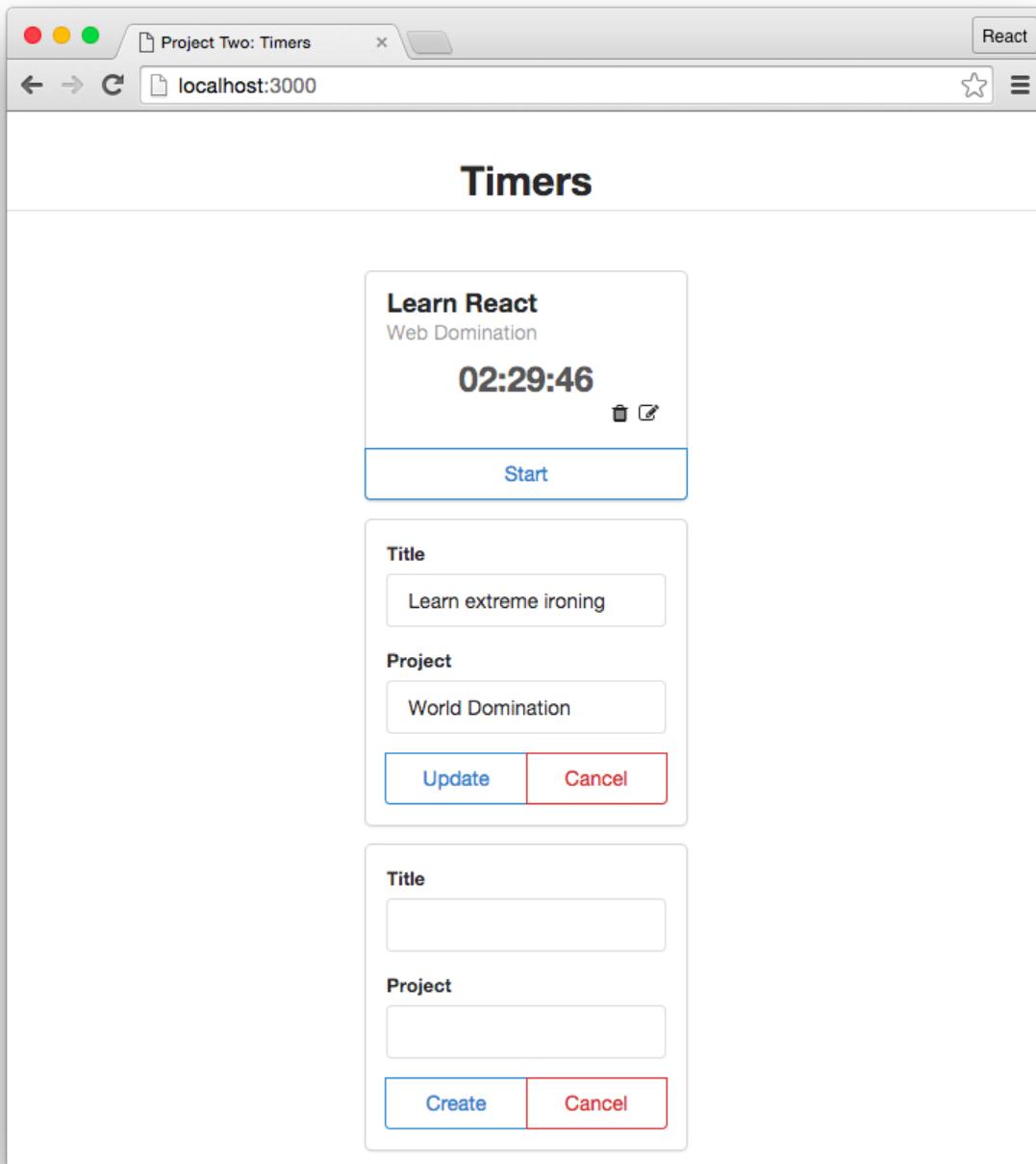
Again, we specify with `ReactDOM#render()` *which* React component we want to render and *where* in our HTML document (`index.html`) to render it.

In this case, we're rendering `TimersDashboard` at the `div` with the `id` of `content`.

Try it out

Save `app.js` and boot the server (`npm run server`). Find it at localhost:3000²²:

²²localhost:3000



Tweak some of the props and refresh to see the results. For example:

- Flip the prop passed down to `ToggleableTimerForm` from `true` to `false` and see the “+” button render.
- Flip parameters on `editFormOpen` and witness `EditableTimer` flip the child it renders accordingly.

Let's review all of the components represented on the page:

Inside `TimersDashboard` are two child components: `EditbleTimerList` and `ToggleableTimerForm`.

`EditbleTimerList` contains two `EditbleTimer` components. The first of these has a `Timer` component as a child and the second a `TimerForm`. These bottom-level components — also known as **leaf components** — hold the majority of the page's HTML. This is generally the case. The components above leaf components are primarily concerned with orchestration.

`ToggleableTimerForm` renders a `TimerForm`. Notice how the two forms on the page have different language for their buttons, as the first is updating and the second is creating.

Step 3: Determine what should be stateful

In order to bestow our app with interactivity, we must evolve it from its static existence to a mutable one. The first step is determining *what*, exactly, should be mutable. Let's start by collecting all of the data that's employed by each component in our static app. In our static app, data will be wherever we are defining or using props. We will then determine which of that data should be stateful.

`TimersDashboard`

In our static app, this declares two child components. It sets one prop, which is the `isOpen` boolean that is passed down to `ToggleableTimerForm`.

`EditbleTimerList`

This declares two child components, each which have props corresponding to a given timer's properties.

`EditbleTimer`

This uses the prop `editFormOpen`.

`Timer`

This uses all the props for a timer.

`TimerForm`

This uses the `title` and `project` props when editing an existing timer.

State criteria

We can apply criteria to determine if data should be stateful:



These questions are from the excellent article by Facebook called "Thinking In React". You can [read the original article here](#)²³.

²³<https://facebook.github.io/react/docs/thinking-in-react.html>

1. Is it passed in from a parent via props? If so, it probably isn't state.

A lot of the data used in our child components are already listed in their parents. This criterion helps us de-duplicate.

For example, “timer properties” is listed multiple times. When we see the properties declared in `EditbleTimerList`, we can consider it state. But when we see it elsewhere, it’s not.

2. Does it change over time? If not, it probably isn't state.

This is a key criterion of stateful data: it changes.

3. Can you compute it based on any other state or props in your component? If so, it's not state.

For simplicity, we want to strive to represent state with as few data points as possible.

Applying the criteria

`TimersDashboard`

- `isOpen` boolean for `ToggleableTimerForm`

Stateful. The data is defined here. It changes over time. And it cannot be computed from other state or props.

`EditbleTimerList`

- Timer properties

Stateful. The data is defined in this component, changes over time, and cannot be computed from other state or props.

`EditbleTimer`

- `editFormOpen` for a given timer

Stateful. The data is defined in this component, changes over time, and cannot be computed from other state or props.

`Timer`

- Timer properties

In this context, **not stateful**. Properties are passed down from the parent.

`TimerForm`

- title and project properties for a given timer

Not stateful. Properties are passed down from parent.

We've identified our stateful data:

- The list of timers and properties of each timer
- Whether or not the edit form of a timer is open
- Whether or not the create form is open

Step 4: Determine in which component each piece of state should live

While the data we've determined to be stateful might live in certain components in our static app, this does not indicate the best position for it in our stateful app. Our next task is to determine the optimal place for each of our three discrete pieces of state to live.

This can be challenging at times but, again, we can apply the following steps from Facebook's guide “Thinking in React²⁴” to help us with the process:

For each piece of state:

- Identify every component that renders something based on that state.
- Find a common owner component (a single component above all the components that need the state in the hierarchy).
- Either the common owner or another component higher up in the hierarchy should own the state.
- If you can't find a component where it makes sense to own the state, create a new component simply for holding the state and add it somewhere in the hierarchy above the common owner component.

Let's apply this method to our application:

The list of timers and properties of each timer

At first glance, we may be tempted to conclude that `TimersDashboard` does not appear to use this state. Instead, the first component that uses it is `EditableTimerList`. This matches the location of the declaration of this data in our static app. Because `ToggleableTimerForm` doesn't appear to use the state either, we might deduce that `EditableTimerList` must then be the common owner.

²⁴<https://facebook.github.io/react/docs/thinking-in-react.html>

While this may be the case for displaying timers, modifying them, and deleting them, what about creates? `ToggleableTimerForm` does not need the state to render, but it *can* affect state. It needs to be able to insert a new timer. It will propagate the data for the new timer up to `TimersDashboard`.

Therefore, `TimersDashboard` is truly the common owner. It will render `EditableTimerList` by passing down the timer state. It can handle modifications from `EditableTimerList` and creates from `ToggleableTimerForm`, mutating the state. The new state will flow downward through `EditableTimerList`.

Whether or not the edit form of a timer is open

In our static app, `EditableTimerList` specifies whether or not a `EditableTimer` should be rendered with its edit form open. Technically, though, this state could just live in each individual `EditableTimer`. No parent component in the hierarchy depends on this data.

Storing the state in `EditableTimer` will be fine for our current needs. But there are a few requirements that might require us to “hoist” this state up higher in the component hierarchy in the future.

For instance, what if we wanted to impose a restriction such that only one edit form could be open at a time? Then it would make sense for `EditableTimerList` to own the state, as it would need to inspect it to determine whether to allow a new “edit form open” event to succeed. If we wanted to allow only one form open at all, including the create form, then we’d hoist the state up to `TimersDashboard`.

Visibility of the create form

`TimersDashboard` doesn’t appear to care about whether `ToggleableTimerForm` is open or closed. It feels safe to reason that the state can just live inside `ToggleableTimerForm` itself.

So, in summary, we’ll have three pieces of state each in three different components:

- **Timer data** will be owned and managed by `TimersDashboard`.
- Each `EditableTimer` will manage the state of its **timer edit form**.
- The `ToggleableTimerForm` will manage the state of its **form visibility**.

Step 5: Hard-code initial states

We’re now well prepared to make our app stateful. At this stage, we won’t yet communicate with the server. Instead, we’ll define our initial states within the components themselves. This means hard-coding a list of timers in the top-level component, `TimersDashboard`. For our two other pieces of state, we’ll have the components’ forms closed by default.

After we’ve added initial state to a parent component, we’ll make sure our props are properly established in its children.

Adding state to TimersDashboard

Start by modifying TimersDashboard to hold the timer data directly inside the component:

```
const TimersDashboard = React.createClass({
  getInitialState: function () {
    return {
      timers: [
        {
          title: 'Practice squat',
          project: 'Gym Chores',
          id: uuid.v4(),
          elapsed: 5456099,
          runningSince: Date.now(),
        },
        {
          title: 'Bake squash',
          project: 'Kitchen Chores',
          id: uuid.v4(),
          elapsed: 1273998,
          runningSince: null,
        },
      ],
    };
  },
  render: function () {
    return (
      <div className='ui three column centered grid'>
        <div className='column'>
          <EditableTimerList
            timers={this.state.timers}
          />
          <ToggleableTimerForm />
        </div>
      </div>
    );
  },
});
```

In `getInitialState`, we set the state to an object with the key `timers`. `timers` points to an array with two timer objects.

For the `id` property, we're using a library called `uuid`. We load this library in `index.html`. We use `uuid.v4()` to randomly generate a [Universally Unique Identifier²⁵](#) for each item.



A UUID is a string that looks like this:

2030efbd-a32f-4fcc-8637-7c410896b3e3

Receiving props in `EditableTimerList`

`EditableTimerList` receives the list of timers as a prop, `timers`. Modify that component to use those props:

```
const EditableTimerList = React.createClass({
  render: function () {
    const timers = this.props.timers.map((timer) => (
      <EditableTimer
        key={timer.id}
        id={timer.id}
        title={timer.title}
        project={timer.project}
        elapsed={timer.elapsed}
        runningSince={timer.runningSince}
      />
    )));
    return (
      <div id='timers'>
        {timers}
      </div>
    );
  },
});
```

Hopefully this looks familiar. We're using `map` on the `timers` array to build a list of `EditableTimer` components. This is exactly how we built our list of `Product` components inside `ProductList` in the last chapter.

We pass the `id` down to `EditableTimer` as well. This is a bit of eager preparation. Remember how `Product` communicated up to `ProductList` by calling a function and passing in its `id`? It's safe to assume we'll be doing this again.

²⁵https://en.wikipedia.org/wiki/Universally_unique_identifier

Props vs. state

With your renewed understanding of React's state paradigm, let's reflect on props again.

Remember, **props are state's immutable accomplice**. What existed as mutable state in Timers-
Dashboard is passed down as immutable props to EditableTimerList.

We talked at length about what qualifies as state and where state should live. Mercifully, we do not need to have an equally lengthy discussion about props. Once you understand state, you can see how props act as its **one-way data pipeline**. State is managed in some select parent components and then that data flows down through children as props.

If state is updated, the component managing that state re-renders by calling `render()`. This, in turn, causes any of its children to re-render as well. And the children of those children. And on and on down the chain.

Let's continue our own march down the chain.

Adding state to EditableTimer

In the static version of our app, `EditableTimer` relied on `editFormOpen` as a prop to be passed down from the parent. We decided that this state could actually live here in the component itself.

Again, we use `getInitialState` to setup state for the component. We set the initial value of `editFormOpen` to `false`, which means that the form starts off as closed. We'll also pass the `id` property down the chain:

```
const EditableTimer = React.createClass({
  getInitialState: function () {
    return {
      editFormOpen: false,
    };
  },
  render: function () {
    if (this.state.editFormOpen) {
      return (
        <TimerForm
          id={this.props.id}
          title={this.props.title}
          project={this.props.project}
        />
      );
    } else {
      return (
        <Timer

```

```
        id={this.props.id}
        title={this.props.title}
        project={this.props.project}
        elapsed={this.props.elapsed}
        runningSince={this.props.runningSince}
      />
    );
}
},
));
});
```

Timer and TimerForm remain stateless

If you look at `Timer` and `TimerForm`, you'll see that they do not need to be modified. They've always been using exclusively props and are so far unaffected by our refactor.

Adding state to ToggleableTimerForm

We know that we'll need to tweak `ToggleableTimerForm` as we've assigned it some stateful responsibility. While we're here, we can also add a little bit of interactivity:

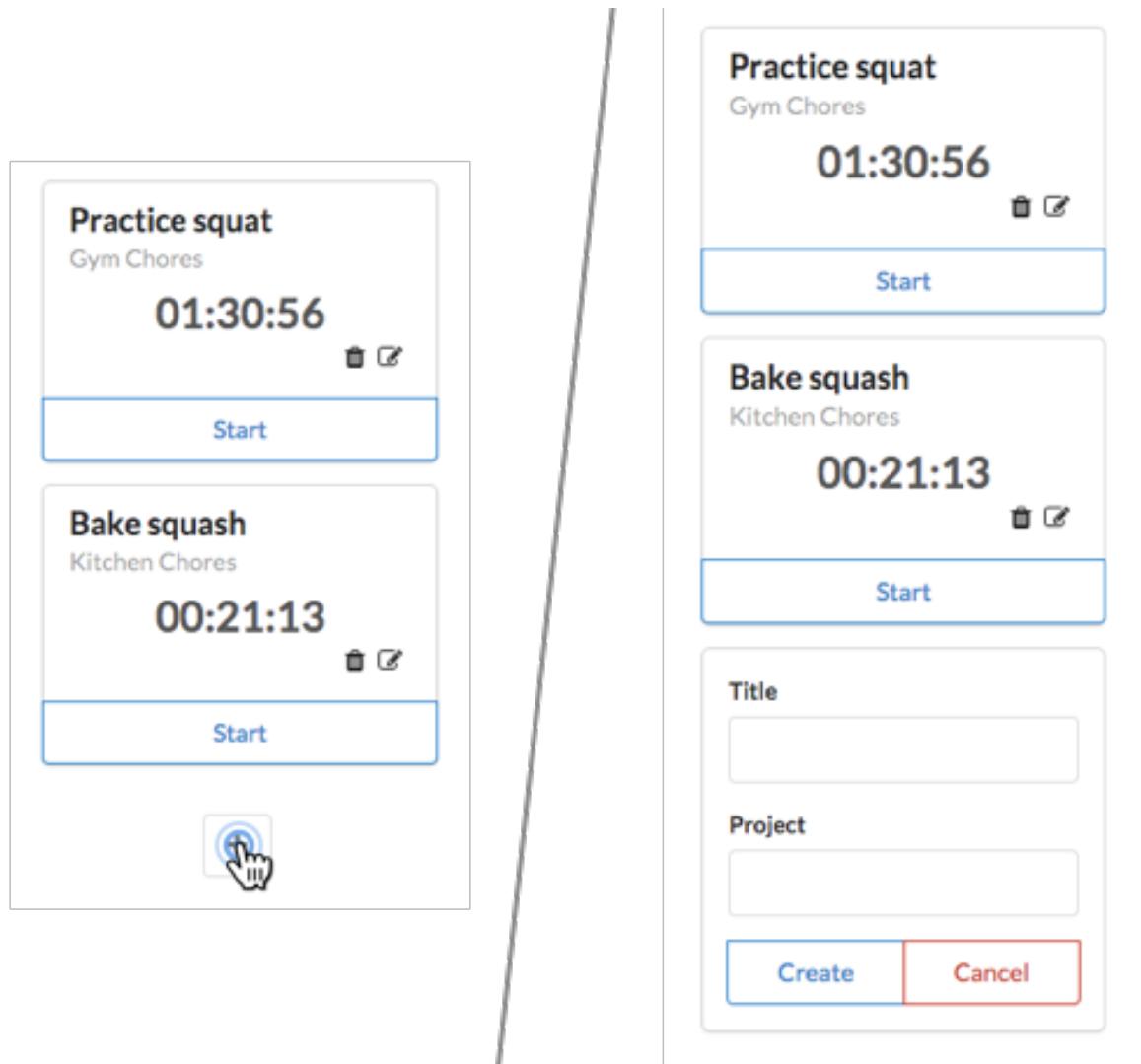
```
const ToggleableTimerForm = React.createClass({
  getInitialState: function () {
    return {
      isOpen: false,
    };
  },
  handleFormOpen: function () {
    this.setState({ isOpen: true });
  },
  render: function () {
    if (this.state.isOpen) {
      return (
        <TimerForm />
      );
    } else {
      return (
        <div className='ui basic content center aligned segment'>
          <button
            className='ui basic button icon'
            onClick={this.handleFormOpen}
          >
```

```
        <i className='plus icon'></i>
      </button>
    </div>
  );
}
},
));
});
```

However small, our app's first bit of interactivity is in place. Like the up-vote button in the last app, we use the `onClick` property on `button` to invoke a function, `handleFormOpen()`. `handleFormOpen()` modifies the state, setting `isOpen` to `true`. This causes the component to re-render. When `render()` is called this second time around, `this.state.isOpen` is `true` and `ToggleableTimerForm` renders `TimerForm`. Neat.

We've established our stateful data inside our elected components. Our downward data pipeline, props, is assembled.

We're ready — and perhaps a bit eager — to build out interactivity using inverse data flow. But before we do, let's save and reload the app to ensure everything is working. We expect to see new example timers based on the hard-coded data in `TimersDashboard`. We also expect clicking the “+” button toggles open a form:



Step 6: Add inverse data flow

As we saw in the last chapter, children communicate with parents by calling functions that are handed to them via props. In the ProductHunt app, when an up-vote was clicked `Product` didn't do any data management. It was not the owner of its state. Instead, it called a function given to it by `ProductList`, passing in its id. `ProductList` was then able to manage state accordingly.

We are going to need inverse data flow in two areas:

- `TimerForm` needs to propagate `create` and `update` events (create while under `ToggleableTimerForm` and update while under `EditableTimer`). Both events will eventually reach `TimersDashboard`.

- Timer has a fair amount of behavior. It needs to handle **delete** and **edit** clicks, as well as the **start** and **stop** timer logic.

Let's start with `TimerForm`.

TimerForm

To get a clear idea of what exactly `TimerForm` will require, we'll start by adding event handlers to it and then work our way backwards up the hierarchy.

`TimerForm` needs two **event handlers**:

- When the form is submitted (creating or updating a timer)
- When the “Cancel” button is clicked (closing the form)

`TimerForm` will receive two functions as props to handle each event. The parent component that uses `TimerForm` is responsible for providing these functions:

- `props.onFormSubmit()`: called when the form is submitted
- `props.onFormClose()`: called when the “Cancel” button is clicked

As we'll see soon, this empowers the parent component to dictate what the behavior should be when these events occur.

Implement these event handlers now:

```
const TimerForm = React.createClass({
  handleSubmit: function () {
    this.props.onFormSubmit({
      id: this.props.id,
      title: this.refs.title.value,
      project: this.refs.project.value,
    });
  },
  render: function () {
    const submitText = this.props.id ? 'Update' : 'Create';
    return (
      <div className='ui centered card'>
        <div className='content'>
          <div className='ui form'>
            <div className='field'>
              <label>Title</label>
```

```
        <input type='text' ref='title'
              defaultValue={this.props.title}
            />
      </div>
      <div className='field'>
        <label>Project</label>
        <input type='text' ref='project'
              defaultValue={this.props.project}
            />
      </div>
      <div className='ui two bottom attached buttons'>
        <button
          className='ui basic blue button'
          onClick={this.handleSubmit}
        >
          {submitText}
        </button>
        <button
          className='ui basic red button'
          onClick={this.props.onFormClose}
        >
          Cancel
        </button>
      </div>
    </div>
  </div>
);
},
});
```

Starting with the form itself, notice how we again capture click events using `onClick`, an attribute on both `button` elements. We also made one modification to both `input` elements. We added the attributes `ref='title'` and `ref='project'`. Looking at `handleSubmit()`, you can see how we use `this.refs` to access the values of these fields.

`handleSubmit()` calls a yet-to-be-defined function, `onFormSubmit()`. It passes in a data object with `id`, `title`, and `project` attributes, extracting values from `this.refs` for `title` and `project`. `id` is pulled from `this.props`. This means `id` will be `undefined` for creates, as no `id` exists yet.

For the cancel button, we just pass in the prop-function `onFormClose()` directly. We do not need to supply this function with any arguments so there's no need to define an additional handler function as we did with submits.



Note that we also tweaked `submitText` so that it uses `this.props.id` to determine if the submission will be a create or update. Using the `id` property to determine whether or not an object has been created is a more common practice.

ToggleableTimerForm

Let's chase this event as it bubbles up the component hierarchy. First, we'll modify `ToggleableTimerForm`. We need it to pass down two prop-functions to `TimerForm`, `onFormClose()` and `onFormSubmit()`:

```
// Inside ToggleableTimerForm
handleFormOpen: function () {
  this.setState({ isOpen: true });
},
handleFormClose: function () {
  this.setState({ isOpen: false });
},
handleFormSubmit: function (timer) {
  this.props.onFormSubmit(timer);
  this.setState({ isOpen: false });
},
render: function () {
  if (this.state.isOpen) {
    return (
      <TimerForm
        onFormSubmit={this.handleFormSubmit}
        onFormClose={this.handleFormClose}
      />
    );
  } else {
    ...
  }
}
```

Looking first at the `render()` function, we can see we pass in the two functions as props. Functions are just like any other prop.

Of most interest here is `handleFormSubmit()`. Remember, `ToggleableTimerForm` is not the manager of timer state. `TimerForm` has an event it's emitting, in this case the submission of a new timer. `ToggleableTimerForm` is just a proxy of this message. So, when the form is submitted, it calls its own prop-function `props.onFormSubmit()`. We'll eventually define this function in `TimersDashboard`.

`handleFormSubmit()` accepts the argument `timer`. Recall that in `TimerForm` this argument is an object containing the desired timer properties. We just pass that argument along here.

After invoking `onFormSubmit()`, `handleFormSubmit()` calls `setState()` to close its form.



Note that the *result* of `onFormSubmit()` will not impact whether or not the form is closed. We invoke `onFormSubmit()`, which may eventually create an **asynchronous** call to a server. Execution will continue before we hear back from the server which means `setState()` will be called.

If `onFormSubmit()` fails — such as if the server is temporarily unreachable — we'd ideally have some way to display an error message and re-open the form. We leave the implementation of this improvement as an exercise at the end of the chapter.

TimersDashboard

We've reached the top of the hierarchy, `TimersDashboard`. As this component will be responsible for the data for the timers, it is here that we will define the logic for handling the events we're capturing down at the leaf components.

The first event we're concerned with is the submission of a form. When this happens, either a new timer is being *created* or an existing one is being *updated*. We'll use two separate functions to handle the two distinct events:

- `handleCreateFormSubmit()` will handle creates and will be the function passed to `TogglableTimerForm`
- `handleEditFormSubmit()` will handle updates and will be the function passed to `EditableTimerList`

Both functions travel down their respective component hierarchies until they reach `TimerForm` as the prop `onFormSubmit()`.

Let's start with `handleCreateFormSubmit`, which inserts a new timer into our timer list state:

```
// Inside TimersDashboard
handleCreateFormSubmit: function (timer) {
  this.createTimer(timer);
},
createTimer: function (timer) {
  const t = helpers.newTimer(timer);
  this.setState({
    timers: this.state.timers.concat(t),
  });
},
render: function () {
  return (
    <div className='ui three column centered grid'>
```

```
<div className='column'>
  <EditableTimerList
    timers={this.state.timers}
  />
  <ToggleableTimerForm
    onFormSubmit={this.handleCreateFormSubmit}
  />
</div>
</div>
);
},
```

We create the timer object with `helpers.newTimer()`. You can peek at the implementation inside of `helpers.js`. We pass in the object that originated down in `TimerForm`. This object has `title` and `project` properties. `helpers.newTimer()` returns an object with those `title` and `project` properties as well as a generated `id`.

The next line calls `setState()`, appending the new timer to our array of timers held under `timers`. We pass the whole state object to `setState()`.



`concat()` creates a new array that contains the elements of the array it was called on followed by the elements passed in as arguments. If an array is passed in as an argument, its elements are appended to the new array. For example:

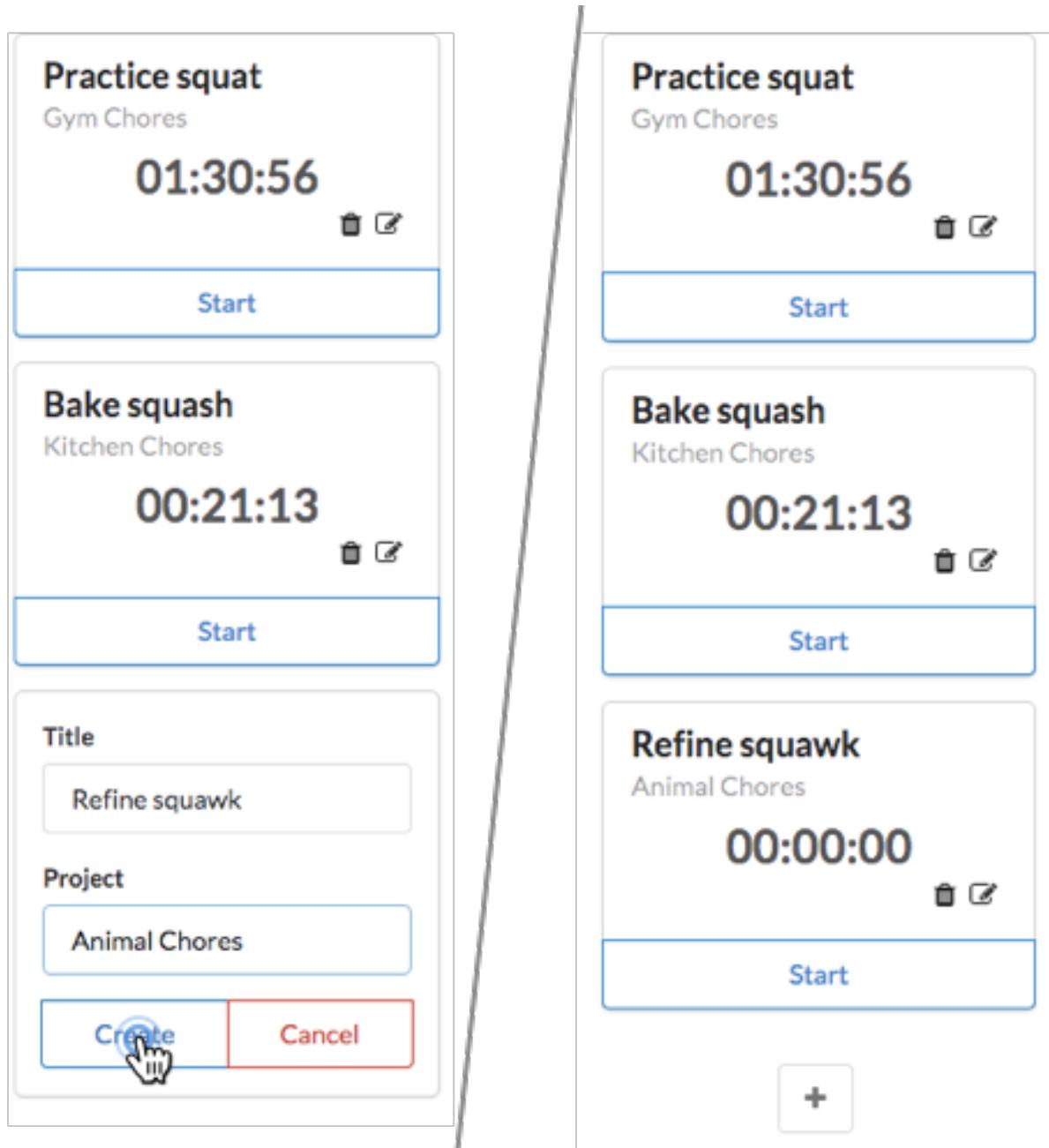
```
> [ 1, 2, 3 ].concat([ 4, 5 ]);
=> [ 1, 2, 3, 4, 5 ]
```



You might wonder: why separate `handleCreateFormSubmit()` and `createTimer()`? While not strictly required, the idea here is that we have one function for handling the event (`handleCreateFormSubmit()`) and another for performing the operation of creating a timer (`createTimer()`).

This separation follows from the Single Responsibility Principle and enables us to call `createTimer()` from elsewhere if needed.

We've finished wiring up the create timer flow from the form down in `TimerForm` up to the state managed in `TimersDashboard`. Save `app.js` and reload your browser. Toggle open the create form and create some new timers:



Updating timers

We need to give the same treatment to the update timer flow. However, as you can see in the current state of the app, we haven't yet added the ability for a timer to be edited. So we don't have a way to display an edit form, which will be a prerequisite to submitting one.

To display an edit form, the user clicks on the edit icon on a Timer. This should propagate an event up to `EditableTimer` and tell it to flip its child component, opening the form.

Adding editability to Timer

To notify our app that the user wants to edit a timer we need to add an `onClick` attribute to the `span` tag of the edit button. We anticipate a prop-function, `onEditClick()`:

```
/* Inside Timer.render() */
<div className='extra content'>
  <span
    className='right floated edit icon'
    onClick={this.props.onEditClick}>
    <i className='edit icon'></i>
  </span>
  <span className='right floated trash icon'>
    <i className='trash icon'></i>
  </span>
</div>
```

Updating EditableTimer

Now we're prepared to update `EditableTimer`. Again, it will display either the `TimerForm` (if we're editing) or an individual `Timer` (if we're not editing).

Let's add event handlers for both possible child components. For `TimerForm`, we want to handle the form being closed or submitted. For `Timer`, we want to handle the edit icon being pressed:

```
// Inside EditableTimer
handleEditClick: function () {
  this.openForm();
},
handleFormClose: function () {
  this.closeForm();
},
handleSubmit: function (timer) {
  this.props.onSubmit(timer);
  this.closeForm();
},
closeForm: function () {
  this.setState({ editFormOpen: false });
},
openForm: function () {
  this.setState({ editFormOpen: true });
},
render: function () {
```

We pass these event handlers down as props:

```

render: function () {
  if (this.state.editFormOpen) {
    return (
      <TimerForm
        id={this.props.id}
        title={this.props.title}
        project={this.props.project}
        onFormSubmit={this.handleSubmit}
        onFormClose={this.handleFormClose}
      />
    );
  } else {
    return (
      <Timer
        id={this.props.id}
        title={this.props.title}
        project={this.props.project}
        elapsed={this.props.elapsed}
        runningSince={this.props.runningSince}
        onEditClick={this.handleEditClick}
      />
    );
  }
},

```

Look a bit familiar? `EditableTimer` handles the same events emitted from `TimerForm` in a very similar manner as `ToggleableTimerForm`. This makes sense. Both `EditableTimer` and `ToggleableTimerForm` are just intermediaries between `TimerForm` and `TimersDashboard`. `TimersDashboard` is the one that defines the submit function handlers and assigns them to a given component tree.

Like `ToggleableTimerForm`, `EditableTimer` doesn't do anything with the incoming timer. In `handleSubmit()`, it just blindly passes this object along to its prop-function `onFormSubmit()`. It then closes the form with `closeForm()`.

We pass along a new prop to `Timer`, `onEditClick`. The behavior for this function is defined in `handleEditClick`, which modifies the state for `EditableTimer`, opening the form.

Updating `EditableTimerList`

Moving up a level, we make a one-line addition to `EditableTimerList` to send the submit function from `TimersDashboard` to each `EditableTimer`:

```
// Inside EditableTimerList
const timers = this.props.timers.map((timer) => (
  <EditableTimer
    key={timer.id}
    id={timer.id}
    title={timer.title}
    project={timer.project}
    elapsed={timer.elapsed}
    runningSince={timer.runningSince}
    onFormSubmit={this.props.onFormSubmit}
  />
));
// ...
```

EditableTimerList doesn't need to do anything with this event so again we just pass the function on directly.

Defining onEditFormSubmit() in TimersDashboard

Last step with this pipeline is to define and pass down the submit function for edit forms in TimersDashboard.

For creates, we have a function that creates a new timer object with the specified attributes and we append this new object to the end of the timers array in the state.

For updates, we need to hunt through the timers array until we find the timer object that is being updated. As mentioned in the last chapter, the state object **cannot** be updated directly. We have to use `setState()`.

Therefore, we'll use `map()` to traverse the array of timer objects. If the timer's id matches that of the form submitted, we'll return a new object that contains the timer with the updated attributes. Otherwise we'll just return the original timer. This new array of timer objects will be passed to `setState()`:

```
// Inside TimersDashboard
handleEditFormSubmit: function (attrs) {
  this.updateTimer(attrs);
},
createTimer: function (timer) {
  const t = helpers.newTimer(timer);
  this.setState({
    timers: this.state.timers.concat(t),
  });
},
```

```

updateTimer: function (attrs) {
  this.setState({
    timers: this.state.timers.map((timer) => {
      if (timer.id === attrs.id) {
        return Object.assign({}, timer, {
          title: attrs.title,
          project: attrs.project,
        });
      } else {
        return timer;
      }
    }),
  });
},
render: function () {

```

We pass this down as a prop inside render():

```

{ /* Inside TimersDashboard.render() */
  <EditableTimerList
    timers={this.state.timers}
    onFormSubmit={this.handleEditFormSubmit}
  />

```

Note that we can call `map()` on `this.state.timers` from *within* the JavaScript object we're passing to `setState()`. This is an often used pattern. The call is evaluated and then the property `timers` is set to the result.

Inside of the `map()` function we check if the `timer` matches the one being updated. If not, we just return the `timer`. Otherwise, we use `Object#assign()` to return a new object with the timer's updated attributes.

As we did with `ToggleableTimerForm` and `handleCreateFormSubmit`, we pass down `handleEditFormSubmit` as the prop `onFormSubmit`. `TimerForm` calls this prop, oblivious to the fact that this function is entirely different when it is rendered underneath `EditableTimer` as opposed to `ToggleableTimerForm`.

ES6: `Object#assign`

We will use `Object#assign()` frequently throughout this book to create new objects as opposed to modifying existing ones.

`Object#assign()` accepts any number of objects as arguments. When the function receives two arguments, it *copies* the properties of the second object onto the first, like so:

```
const coffee = { };
const noCream = { cream: false };
const noMilk = { milk: false };
Object.assign(coffee, noCream);
// coffee is now: `{ cream: false }`
Object.assign(coffee, noMilk);
// coffee is now: `{ cream: false, milk: false }`
```

Throughout this book, we'll often pass three arguments to `Object.assign()`, as seen in `TimersDashboard`. The first argument is a new JavaScript object, the one that `Object#assign()` will ultimately return. The second is the object whose properties we'd like to build off of. The last is the changes we'd like to apply:

```
const coffeeWithMilk = Object.assign({}, coffee, { milk: true });
// coffeeWithMilk is: `{ cream: false, milk: true }`
// coffee was not modified: `{ cream: false, milk: false }`
```

Modifying JavaScript objects

Note that we could have gotten away with updating the `timer` object directly, like so:

```
// ...
if (timer.id === attrs.id) {
  timer.title = attrs.title;
  timer.project = attrs.project;
  return timer;
}
```

In JavaScript objects are **passed by reference**. As such, the code above would have modified the `timer` object sitting inside `this.state.timers`, *not* a copy of that object. We still call `setState()`, though, so React *would* be notified about this change. It is unlikely negative consequences would have emerged.

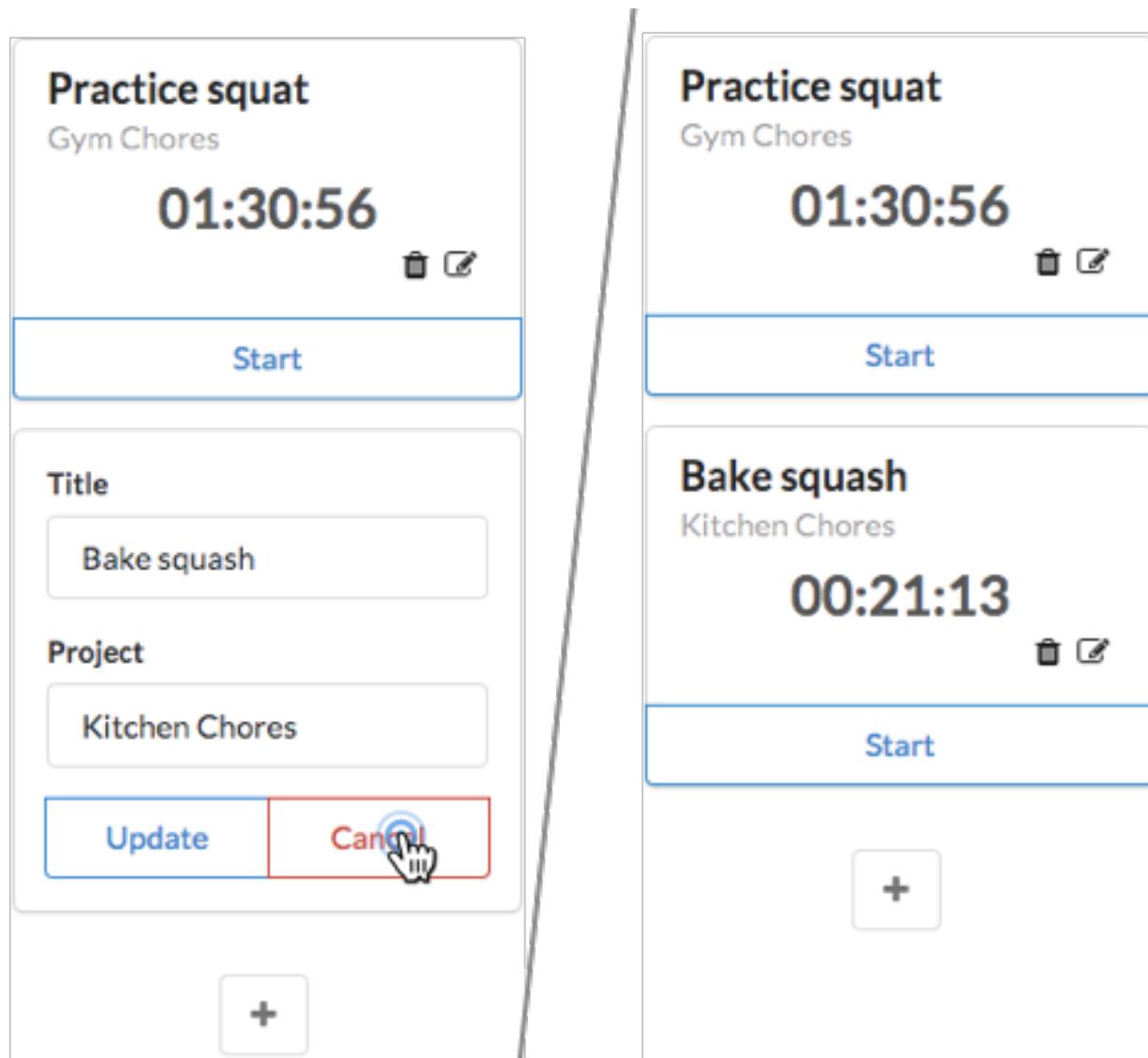
However, manipulating objects in this manner is generally bad practice and something we will avoid throughout this book. The general rule we will adhere to: **unless a function owns an object (it declared the object), it should refrain from making any edits to it.**

To understand why this practice is dangerous, let's reflect on what could happen in this specific instance. Consider how many different components in the app reference the state object. Every single prop corresponding to a timer is referencing properties on this object. If we make direct edits to this object, those changes will ripple out into all sorts of different areas in the code.

Imagine if instead of immediately calling `setState()`, we had some sort of delay after modifying the state object, like a call to a web service. If any children happened to re-render during that period (like if their own state changed), the props they would use would be the *new values*, even though we weren't yet prepared for them to receive this change. This could lead to all sorts of odd, unpredictable behavior.

We will explore the related concept of **pure functions** in a future chapter.

Both of the forms are wired up! Save `app.js`, reload the page, and try both creating and updating timers. You can also click “Cancel” on an open form to close it:



The rest of our work resides within the timer. We need to:

- Wire up the trash button (deleting a timer)
- Implement the start/stop buttons and the timing logic itself

At that point, we'll have a complete server-less solution.

Try it yourself: Before moving on to the next section, see how far you can get wiring up the trash button by yourself. Move ahead afterwards and verify your solution is sound.

Deleting timers

Adding the event handler to Timer

In `Timer`, we define a function to handle trash button click events:

```
const Timer = React.createClass({
  handleTrashClick: function () {
    this.props.onTrashClick(this.props.id);
  },
  render: function () {
    // ...
  }
});
```

And then use `onClick` to connect that function to the trash icon:

```
{/* Inside Timer.render() */}
<div className='extra content'>
  <span
    className='right floated edit icon'
    onClick={this.props.onEditClick}>
    <i className='edit icon'></i>
  </span>
  <span
    className='right floated trash icon'
    onClick={this.handleTrashClick}>
    <i className='trash icon'></i>
  </span>
</div>
```

We've yet to define the function that will be set as the prop `onTrashClick()`. But you can imagine that when this event reaches the top (`TimersDashboard`), we're going to need the `id` to sort out which timer is being deleted. `handleTrashClick()` provides the `id` to this function.

Routing through EditableTimer

`EditableTimer` just proxies the function:

```
// Inside EditableTimer
} else {
  return (
    <Timer
      id={this.props.id}
      title={this.props.title}
      project={this.props.project}
      elapsed={this.props.elapsed}
      runningSince={this.props.runningSince}
      onEditClick={this.handleEditClick}
      onTrashClick={this.props.onTrashClick}
    />
  );
}
```

Routing through EditableTimerList

As does `EditableTimerList`:

```
// Inside EditableTimerList.render()
const timers = this.props.timers.map((timer) => (
  <EditableTimer
    key={timer.id}
    id={timer.id}
    title={timer.title}
    project={timer.project}
    elapsed={timer.elapsed}
    runningSince={timer.runningSince}
    onFormSubmit={this.props.onFormSubmit}
    onTrashClick={this.props.onTrashClick}
  />
));

```

Implementing the delete function in TimersDashboard

The last step is to define the function in `TimersDashboard` that deletes the desired timer from the state array. There are many ways to accomplish this in JavaScript. Don't sweat it if your solution was not the same or if you didn't quite work one out.

We add our handler function that we ultimately pass down as a prop:

```
// Inside TimersDashboard
handleEditFormSubmit: function (attrs) {
  this.updateTimer(attrs);
},
handleTrashClick: function (timerId) {
  this.deleteTimer(timerId);
},
```

deleteTimer() uses Array's filter() method to return a new array with the timer object that has an id matching timerId removed:

```
// Inside TimersDashboard
deleteTimer: function (timerId) {
  this.setState({
    timers: this.state.timers.filter(t => t.id !== timerId),
  });
},
render: function () {
```

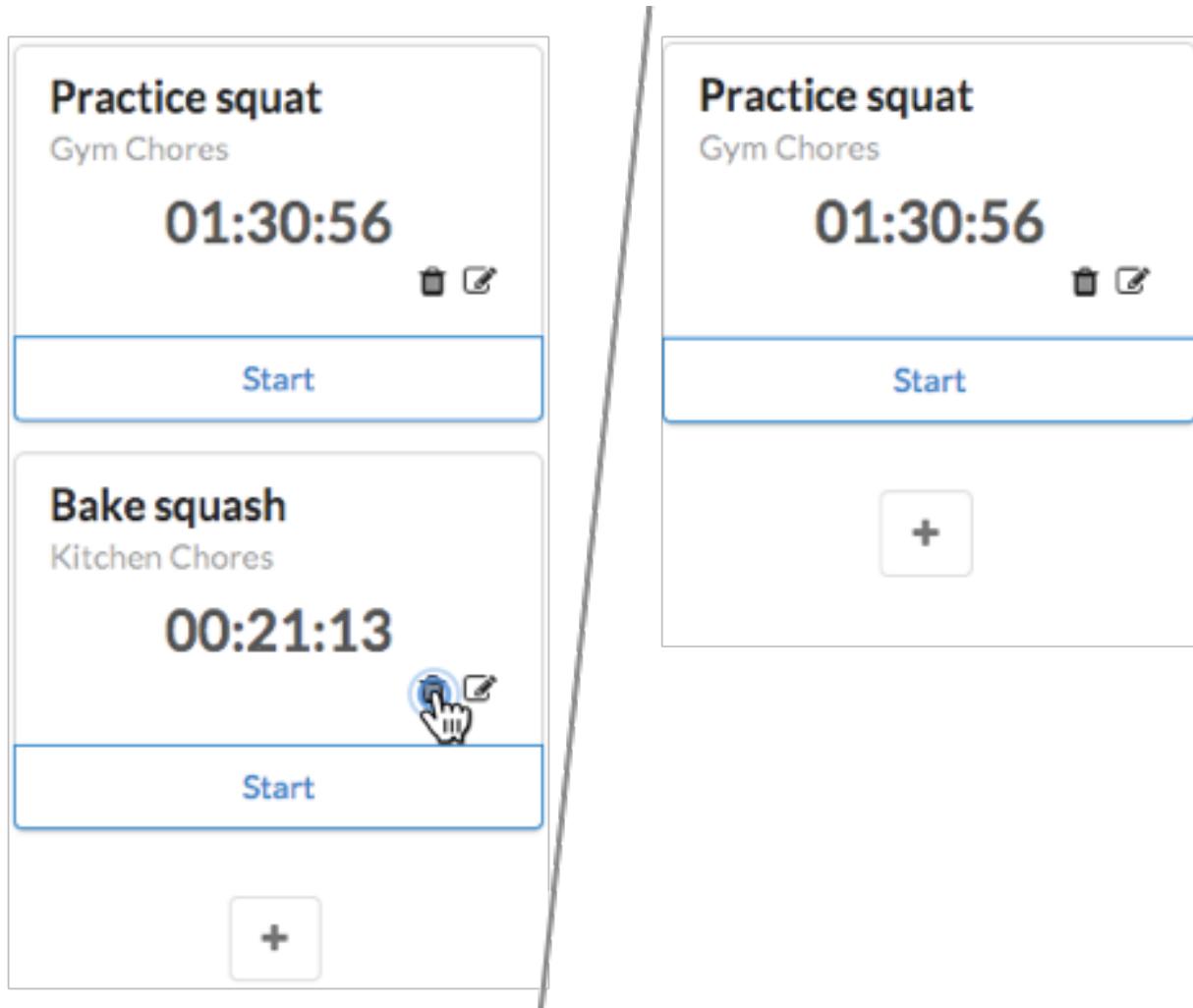
Finally, we pass down handleTrashClick() as a prop:

```
{/* Inside TimersDashboard.render() */}
<EditableTimerList
  timers={this.state.timers}
  onFormSubmit={this.handleEditFormSubmit}
  onTrashClick={this.handleTrashClick}
/>
```



Array's filter() method accepts a function that is used to "test" each element in the array. It returns a new array containing all the elements that "passed" the test. If the function returns true, the element is kept.

Save app.js and reload the app. Low and behold, you can delete timers:



Adding timing functionality

Create, update, and delete (CRUD) capability is now in place for our timers. The next challenge: making these timers functional.

There are several different ways we can implement a timer system. The simplest approach would be to have a function update the `elapsed` property on each timer every second. But this is severely limited. What happens when the app is closed? The timer should continue “running.”

This is why we’ve included the timer property `runningSince`. A timer is initialized with `elapsed` equal to `0`. When a user clicks “Start”, we do not increment `elapsed`. Instead, we just set `runningSince` to the start time.

We can then use the difference between the start time and the current time to render the time for the user. When the user clicks “Stop”, the difference between the start time and the current time is added to `elapsed`. `runningSince` is set to `null`.

Therefore, at any given time, we can derive how long the timer has been running by taking `Date.now() - runningSince` and adding it to the total accumulated time (`elapsed`). We'll calculate this inside the `Timer` component.

For the app to truly feel like a running timer, we want React to constantly perform this operation and re-render the timers. But `elapsed` and `runningSince` *will not be changing while the timer is running*. So the one mechanism we've seen so far to trigger a `render()` call will not be sufficient.

Instead, we can use React's `forceUpdate()` method. This forces a React component to re-render. We can call it on an interval to yield the smooth appearance of a live timer.

Adding a `forceUpdate()` interval to `Timer`

`helpers.renderElapsedString()` accepts an optional second argument, `runningSince`. It will add the delta of `Date.now() - runningSince` to `elapsed` and use the function `millisecondsToHuman()` to return a string formatted as HH:MM:SS.

We will establish an interval to run `forceUpdate()` after the component mounts:

```
const Timer = React.createClass({
  componentDidMount: function () {
    this.forceUpdateInterval = setInterval(() => this.forceUpdate(), 50);
  },
  componentWillUnmount: function () {
    clearInterval(this.forceUpdateInterval);
  },
  handleTrashClick: function () {
    this.props.onTrashClick(this.props.id);
  },
  render: function () {
    const elapsedString = helpers.renderElapsedString(
      this.props.elapsed, this.props.runningSince
    );
    return (

```

In `componentDidMount()`, we use the JavaScript function `setInterval()`. This will invoke the function `forceUpdate()` once every 50 ms, causing the component to re-render. We set the return of `setInterval()` to `this.forceUpdateInterval`.

In `componentWillUnmount()`, we use `clearInterval()` to stop the interval `this.forceUpdateInterval`. `componentWillUnmount()` is called before a component is removed from the app. This will happen if a timer is deleted. We want to ensure we do not continue calling `forceUpdate()` after the timer has been removed from the page. React will throw errors.



`setInterval()` accepts two arguments. The first is the function you would like to call repeatedly. The second is the interval on which to call that function (in milliseconds).

`setInterval()` returns a unique interval ID. You can pass this interval ID to `clearInterval()` at any time to halt the interval.



You might ask: Wouldn't it be more efficient if we did not continuously call `forceUpdate()` on timers that are not running?

Indeed, we would save a few cycles. But it would not be worth the added code complexity. React will call `render()` which performs some inexpensive operations in JavaScript. It will then compare this result to the previous call to `render()` and see that nothing has changed. It stops there — it won't attempt any DOM manipulation.



The 50 ms interval was not derived scientifically. Selecting an interval that's too high will make the timer look unnatural. It would jump unevenly between values. Selecting an interval that's too low would just be an unnecessary amount of work. A 50 ms interval looks good to humans and is ages in computerland.

Try it out

Save `app.js` and reload. The first timer should be running.

We've begun to carve out the app's real utility! We need only wire up the start/stop button and our server-less app will be feature complete.

Add start and stop functionality

The action button at the bottom of each timer should display "Start" if the timer is paused and "Stop" if the timer is running. It should also propagate events when clicked, depending on if the timer is being stopped or started.

We could build all of this functionality into `Timer`. We could have `Timer` decide to render one HTML snippet or another depending on if it is running. But that would be adding more responsibility and complexity to `Timer`. Instead, let's make the button its own React component.

Add timer action events to `Timer`

Let's modify `Timer`, anticipating a new component called `TimerActionButton`. This button just needs to know if the timer is running. It also needs to be able to propagate two events, `onStartClick()` and

`onStopClick()`. These events will eventually need to make it all the way up to `TimersDashboard`, which can modify `runningSince` on the timer.

First, the event handlers:

```
// Inside Timer
componentWillUnmount: function () {
  clearInterval(this.forceUpdateInterval);
},
handleStartClick: function () {
  this.props.onStartClick(this.props.id);
},
handleStopClick: function () {
  this.props.onStopClick(this.props.id);
},
// ...
```

Then, inside `render()`, we'll declare `TimerActionButton` at the bottom of the outermost `div`:

```
/* At the bottom of `Timer.render()`` */
<TimerActionButton
  timerIsRunning={!!this.props.runningSince}
  onStartClick={this.handleStartClick}
  onStopClick={this.handleStopClick}
/>
</div>
);
```

We use the same technique used in other click-handlers: `onClick` on the HTML element specifies a handler function in the component that invokes a prop-function, passing in the timer's id.



We use `!!` here to derive the boolean prop `timerIsRunning` for `TimerActionButton`. `!!` returns `false` when `runningSince` is `null`.

Create `TimerActionButton`

Create the `TimerActionButton` component now:

```
const TimerActionButton = React.createClass({
  render: function () {
    if (this.props.timerIsRunning) {
      return (
        <div
          className='ui bottom attached red basic button'
          onClick={this.props.onStopClick}>
        >
        Stop
        </div>
      );
    } else {
      return (
        <div
          className='ui bottom attached green basic button'
          onClick={this.props.onStartClick}>
        >
        Start
        </div>
      );
    }
  },
});
```

We render one HTML snippet or another based on `this.props.timerIsRunning`.

You know the drill. Now we run these events up the component hierarchy, all the way up to `TimersDashboard` where we're managing state:

Run the events through `EditableTimer` and `EditableTimerList`

First `EditableTimer`:

```
// Inside EditableTimer
} else {
  return (
    <Timer
      id={this.props.id}
      title={this.props.title}
      project={this.props.project}
      elapsed={this.props.elapsed}
      runningSince={this.props.runningSince}
```

```

        onEditClick={this.handleEditClick}
        onTrashClick={this.props.onTrashClick}
        onStartClick={this.props.onStartClick}
        onStopClick={this.props.onStopClick}
      />
    );
}

```

And then EditableTimerList:

```

// Inside EditableTimerList
const timers = this.props.timers.map((timer) => (
  <EditableTimer
    key={timer.id}
    id={timer.id}
    title={timer.title}
    project={timer.project}
    elapsed={timer.elapsed}
    runningSince={timer.runningSince}
    onFormSubmit={this.props.onFormSubmit}
    onTrashClick={this.props.onTrashClick}
    onStartClick={this.props.onStartClick}
    onStopClick={this.props.onStopClick}
  />
));

```

Finally, we define these functions in TimersDashboard. They should hunt through the state timers array using `map`, setting `runningSince` appropriately when they find the matching timer.

First we define the handling functions:

```

// Inside TimersDashboard
handleTrashClick: function (timerId) {
  this.deleteTimer(timerId);
},
handleStartClick: function (timerId) {
  this.startTimer(timerId);
},
handleStopClick: function (timerId) {
  this.stopTimer(timerId);
},

```

And then `startTimer()` and `stopTimer()`:

```
deleteTimer: function (timerId) {
  this.setState({
    timers: this.state.timers.filter(t => t.id !== timerId),
  });
},
startTimer: function (timerId) {
  const now = Date.now();

  this.setState({
    timers: this.state.timers.map((timer) => {
      if (timer.id === timerId) {
        return Object.assign({}, timer, {
          runningSince: now,
        });
      } else {
        return timer;
      }
    }),
  });
},
stopTimer: function (timerId) {
  const now = Date.now();

  this.setState({
    timers: this.state.timers.map((timer) => {
      if (timer.id === timerId) {
        const lastElapsed = now - timer.runningSince;
        return Object.assign({}, timer, {
          elapsed: timer.elapsed + lastElapsed,
          runningSince: null,
        });
      } else {
        return timer;
      }
    }),
  });
},
render: function () {
```

Finally, we pass them down as props:

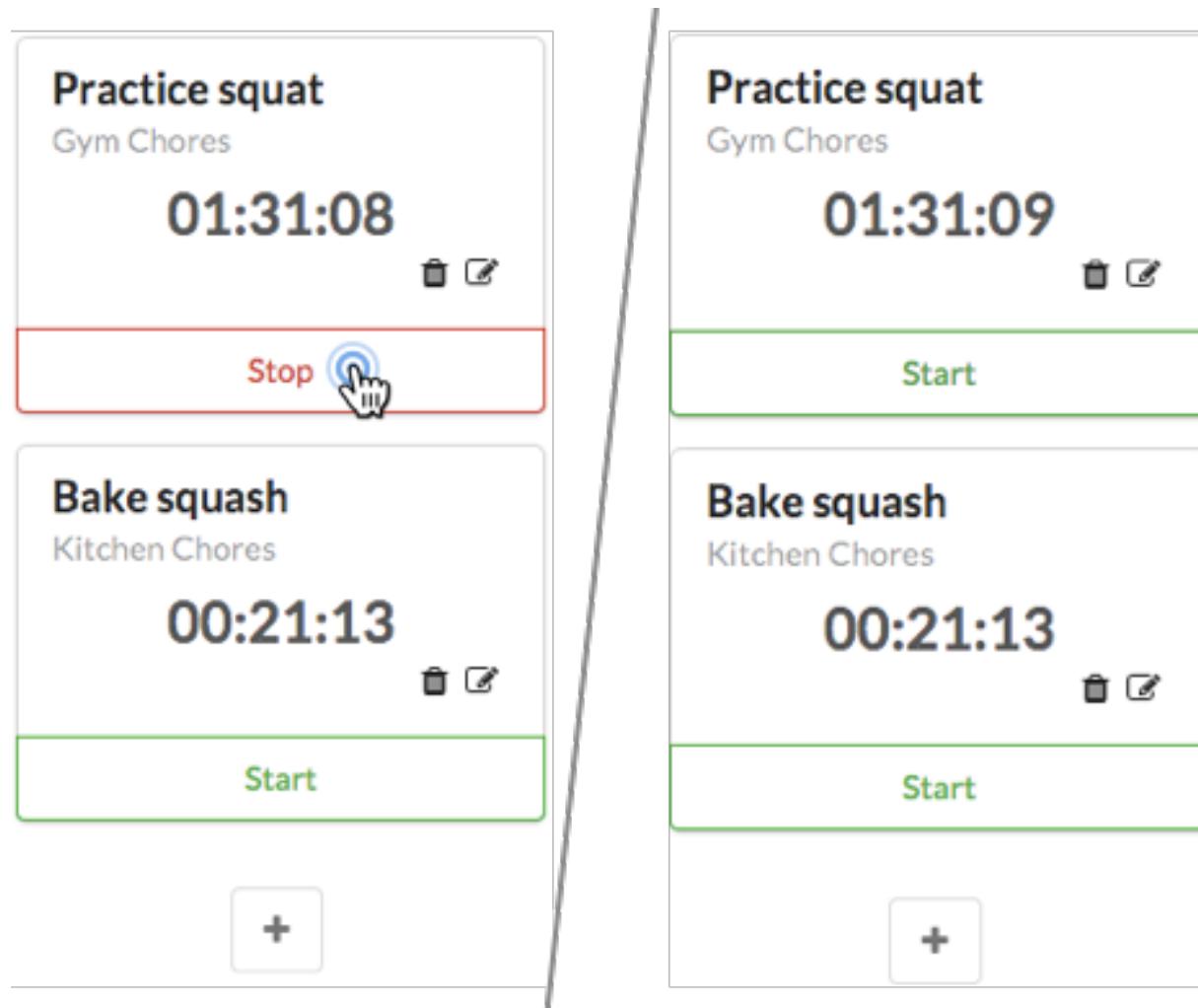
```
/* Inside TimerDashboard.render() */
<EditableTimerList
  timers={this.state.timers}
  onFormSubmit={this.handleEditFormSubmit}
  onTrashClick={this.handleTrashClick}
  onStartClick={this.handleStartClick}
  onStopClick={this.handleStopClick}
/>
```

When `startTimer` comes across the relevant timer within its `map` call, it sets the property `runningSince` to the current time.

`stopTimer` calculates `lastElapsed`, the amount of time that the timer has been running for since it was started. It adds this amount to `elapsed` and sets `runningSince` to `null`, “stopping” the timer.

Try it out

Save `app.js`, reload, and behold! You can now create, update, and delete timers as well as actually use them to time things:



This is excellent progress. But, without a connection to a server, our app is ephemeral. If we refresh the page, we lose all of our timer data. Our app does not have any persistence.

A server can give us persistence. We'll have our server write all changes to timer data to a file. Instead of hard-coding state inside of the `TimersDashboard` component, when our app loads it will query the server and construct its timer state based on the data the server provides. We'll then have our React app notify the server about any state changes, like when a timer is started.

Communicating with a server is the last big major building block you'll need to develop and distribute real-world web applications with React.

Methodology review

While building our time-logging app, we learned and applied a methodology for building React apps. Again, those steps were:

1. Break the app into components

We mapped out the component structure of our app by examining the app's working UI. We then applied the single-responsibility principle to break components down so that each had minimal viable functionality.

2. Build a static version of the app

Our bottom-level (user-visible) components rendered HTML based on static props, passed down from parents.

3. Determine what should be stateful

We used a series of questions to deduce what data should be stateful. This data was represented in our static app as props.

4. Determine in which component each piece of state should live

We used another series of questions to determine which component should own each piece of state. `TimersDashboard` owned timer state data and `ToggleableTimerForm` and `EditbleTimer` both held state pertaining to whether or not to render a `TimerForm`.

5. Hard-code initial states

We then wrote state-owners' `getInitialState` functions with hard-coded values.

6. Add inverse data flow

We added interactivity by decorating buttons with `onClick` handlers. These called functions that were passed in as props down the hierarchy from whichever component owned the relevant state being manipulated.

The final step is 7. Add server communication. We'll tackle this in the next chapter.



Chapter Exercises

1. Modify the `Timer` component so that the "Edit" and "Trash" buttons only display when the mouse is hovering over a given timer. `onMouseEnter` and `onMouseLeave` will serve as your event attributes in much the same way as `onClick` has.
2. Add validation to the `title` and `project` fields on timers. They should not be blank. An error message should be displayed above the field if the validation fails. Check out the forms page on Semantic UI for the HTML for form errors: [http://semantic-ui.com/collections/form.html#error²⁶](http://semantic-ui.com/collections/form.html#error).

²⁶<http://semantic-ui.com/collections/form.html#error>

Components & Servers

Introduction

In the last chapter, we used a methodology to construct a React app. State management of timers takes place in the top-level component `TimersDashboard`. As in all React apps, data flows from the top down through the component tree to leaf components. Leaf components communicate events to state managers by calling prop-functions.

At the moment, `TimersDashboard` has a hard-coded initial state. Any mutations to the state will only live as long as the browser window is open. That's because all state changes are happening in-memory inside of React. We need our React app to communicate with a server. The server will be in charge of persisting the data. In this app, data persistence happens inside of a file, `data.json`.

`EditbleTimer` and `ToggleableTimerForm` also have hard-coded initial state. But because this state is just whether or not their forms are open, we don't need to communicate these state changes to the server. We're OK with the forms starting off closed every time the app boots.

Preparation

To help you get familiar with the API for this project and working with APIs in general, we have a short section where we make requests to the API outside of React.

curl

We'll use a tool called curl to make more involved requests from the command line.

OS X users should already have curl installed.

Windows users can download and install curl here: <https://curl.haxx.se/download.html>²⁷.

server.js

Included in the root of your project folder is a file called `server.js`. This is a Node.js server specifically designed for our time-tracking app.



You don't have to know anything about Node.js or about servers in general to work with the server we've supplied. We'll provide the guidance that you need.

²⁷<https://curl.haxx.se/download.html>

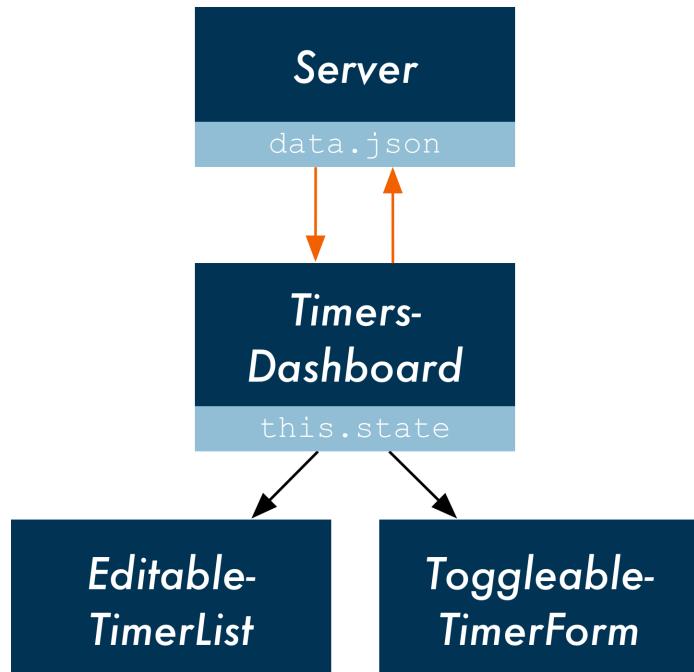
`server.js` uses the file `data.json` as its “store.” The server will read and write to this file to persist data. You can take a look at that file to see the initial state of the store that we’ve provided.

`server.js` will return the contents of `data.json` when asked for all items. When notified, the server will reflect any updates, deletes, or timer stops and starts in `data.json`. This is how data will be persisted even if the browser is reloaded or closed.

Before we start working with the server, let’s briefly cover its API. Again, don’t be concerned if this outline is a bit perplexing. It will hopefully become clearer as we start writing some code.

The Server API

Our ultimate goal in this chapter is to **replicate state changes on the server**. We’re not going to move all state management exclusively to the server. Instead, the server will maintain its state (in `data.json`) and React will maintain its state (in this case, within `this.state` in `TimersDashboard`). We’ll demonstrate later why keeping state in both places is desirable.



`TimersDashboard` communicates with the server

If we perform an operation on the React (“client”) state that we want to be persisted, then we also need to notify the server of that state change. This will keep the two states in sync. We’ll consider these our “write” operations. The write operations we want to send to the server are:

- A timer is created

- A timer is updated
- A timer is deleted
- A timer is started
- A timer is stopped

We'll have just one read operation: requesting all of the timers from the server.



HTTP APIs

This section assumes a little familiarity with HTTP APIs. If you're not familiar with HTTP APIs, you may want to [read up on them²⁸](#) at some point.

However, don't be deterred from continuing with this chapter for the time being. Essentially what we're doing is making a "call" from our browser out to a local server and conforming to a specified format.

text/html endpoint

GET /

This entire time, `server.js` has actually been responsible for serving the app. When your browser requests `localhost:3000/`, the server returns the file `index.html`. `index.html` loads in all of our JavaScript/React code.



Note that React never makes a request to the server at this path. This is just used by the browser to load the app. React only communicates with the JSON endpoints.

JSON endpoints

`data.json` is a JSON document. As touched on in the last chapter, JSON is a format for storing human-readable data objects. We can serialize JavaScript objects into JSON. This enables JavaScript objects to be stored in text files and transported over the network.

`data.json` contains an array of objects. While not strictly JavaScript, the data in this array can be readily loaded into JavaScript.

In `server.js`, we see lines like this:

²⁸<http://www.andrewhavens.com/posts/20/beginners-guide-to-creating-a-rest-api/>

```
fs.readFile(DATA_FILE, function(err, data) {  
  const timers = JSON.parse(data);  
  // ...  
});
```

data is a string, the `JSON.parse()` converts this string into an actual JavaScript array of objects.

GET /api/timers

Returns a list of all timers.

POST /api/timers

Accepts a JSON body with `title`, `project`, and `id` attributes. Will insert a new timer object into its store.

POST /api/timers/start

Accepts a JSON body with the attribute `id` and `start` (a timestamp). Hunts through its store and finds the timer with the matching `id`. Sets its `runningSince` to `start`.

POST /api/timers/stop

Accepts a JSON body with the attribute `id` and `stop` (a timestamp). Hunts through its store and finds the timer with the matching `id`. Updates `elapsed` according to how long the timer has been running (`stop - runningSince`). Sets `runningSince` to `null`.

PUT /api/timers

Accepts a JSON body with the attributes `id` and `title` and/or `project`. Hunts through its store and finds the timer with the matching `id`. Updates `title` and/or `project` to new attributes.

DELETE /api/timers

Accepts a JSON body with the attribute `id`. Hunts through its store and deletes the timer with the matching `id`.

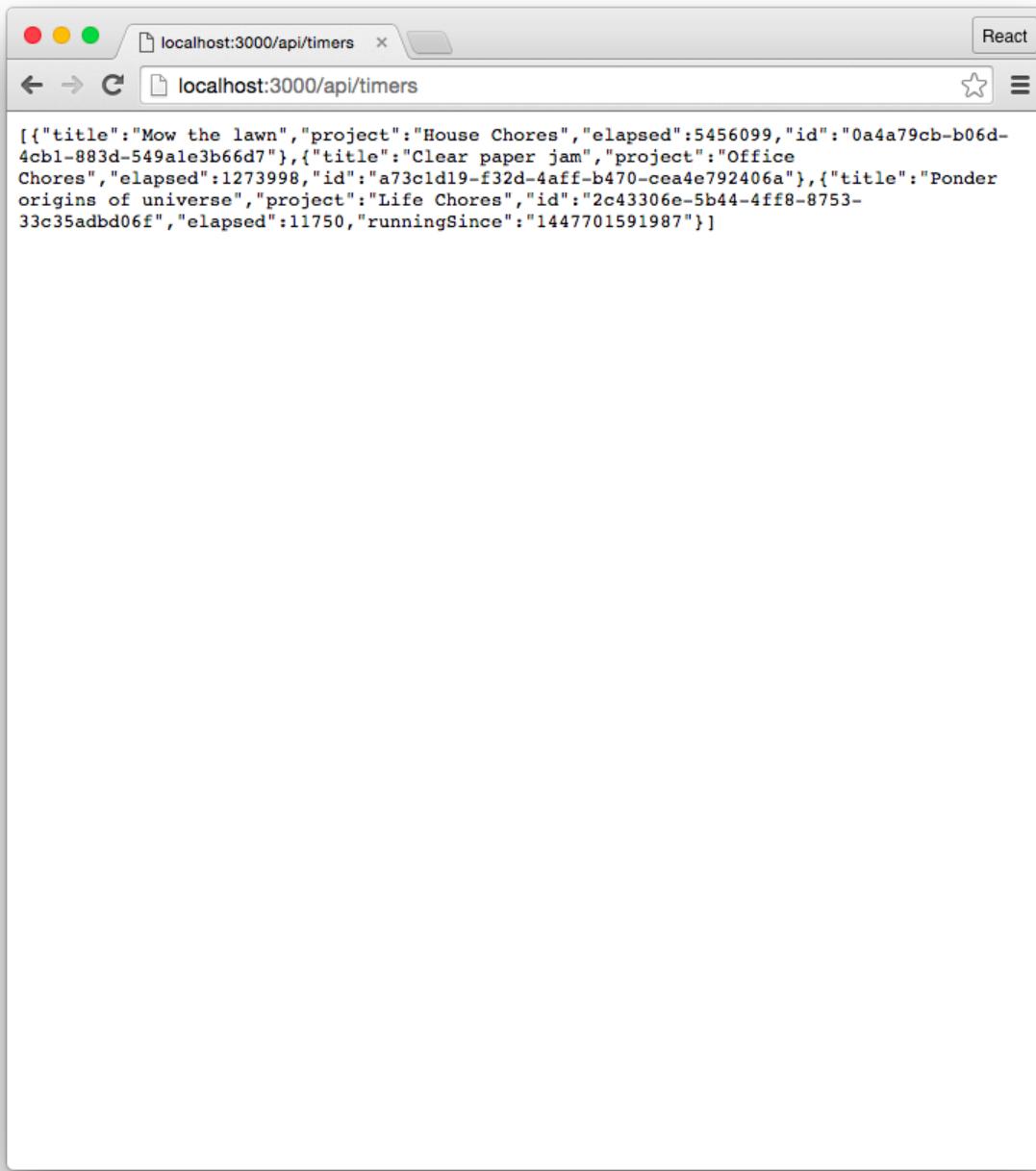
Playing with the API

If your server is not booted, make sure to boot it:

```
npm run server
```

You can visit the endpoint `/api/timers` endpoint in your browser and see the JSON response (localhost:3000/api/timers²⁹). When you visit a new URL in your browser, your browser makes a GET request. So our browser calls `GET /api/timers` and the server returns all of the timers:

²⁹localhost:3000/api/timers



Note that the server stripped all of the extraneous whitespace in `data.json`, including newlines, to keep the payload as small as possible. Those only exist in `data.json` to make it human-readable.

We can only easily use the browser to make GET requests. For *writing* data — like starting and stopping timers — we'll have to make POST, PUT, or DELETE requests. We'll use curl to play around with writing data.

Run the following command from the command line:

```
curl -X GET localhost:3000/api/timers
```

The -X flag specifies which HTTP method to use. It should return a response that looks a bit like this:

```
[{"title": "Mow the lawn", "project": "House Chores", "elapsed": 5456099, "id": "0a4a79\\cb-b06d-4cb1-883d-549a1e3b66d7"}, {"title": "Clear paper jam", "project": "Office Chores", "elapsed": 1273998, "id": "a73c1d19-f32d-4aff-b470-cea4e792406a"}, {"title": "Ponder origins of universe", "project": "Life Chores", "id": "2c43306e-5b44-4ff8-8753\\-33c35adbd06f", "elapsed": 11750, "runningSince": "1456225941911"}]
```

You can start one of the timers by issuing a PUT request to the /api/timers/start endpoint. We need to send along the id of one of the timers and a start timestamp:

```
curl -X POST \
-H 'Content-Type: application/json' \
-d '{"start":1456468632194, "id": "a73c1d19-f32d-4aff-b470-cea4e792406a"}' \
localhost:3000/api/timers/start
```

The -H flag sets a header for our HTTP request, Content-Type. We're informing the server that the body of the request is JSON.

The -d flag sets the body of our request. Inside of single-quotes '' is the JSON data.

When you press enter, curl will quickly return without any output. The server doesn't return anything on success for this endpoint. If you open up data.json, you will see that the timer you specified now has a runningSince property, set to the value we specified as start in our request.

If you'd like, you can play around with the other endpoints to get a feel for how they work. Just be sure to set the appropriate method with -X and to pass along the JSON Content-Type for the write endpoints.

We've written a small library, client, to aid you in interfacing with the API in JavaScript.



Note that the backslash \ above is only used to break the command out over multiple lines for readability.

Tool tip: jq

If you want to parse and process JSON on the command line, we highly recommend the tool “jq.”

You can pipe curl responses directly into jq to have the response pretty-formatted:

```
curl -X GET localhost:3000/api/timers | jq '..'
```

You can also do some powerful manipulation of JSON, like iterating over all objects in the response and returning a particular field. In this example, we extract just the `id` property of every object in an array:

```
curl -X GET localhost:3000/api/timers | jq '.[] | { .id }'
```

You can download jq here: <https://stedolan.github.io/jq/>^a.

^a<https://stedolan.github.io/jq/>

Loading state from the server

Right now, we set initial state in `TimersDashboard` by hardcoding a JavaScript object, an array of timers. Let’s modify this function to load data from the server instead.

We’ve written the client library that your React app will use to interact with the server, `client`. The library is defined in `public/client.js`. We’ll use it first and then take a look at how it works in the next section.

The `GET /api/timers` endpoint provides a list of all timers, as represented in `data.json`. We can use `client.getTimers()` to call this endpoint from our React app. We’ll do this to “hydrate” the state kept by `TimersDashboard`.

When we call `client.getTimers()`, the network request is made *asynchronously*. The function call itself is not going to return anything useful:

```
// Wrong
// `getTimers()` does not return the list of timers
const timers = client.getTimers();
```

Instead, we can pass `getTimers()` a success function. `getTimers()` will invoke that function after it hears back from the server if the server successfully returned a result. `getTimers()` will invoke the function with a single argument, the list of timers returned by the server:

```
// Passing `getTimers()` a success function
client.getTimers((serverTimers) => (
  // do something with the array of timers, `serverTimers`
));
```



client.getTimers() uses the Fetch API, which we cover in the next section. For our purposes, the important thing to know is that when getTimers() is invoked, it fires off the request to the server and then returns control flow *immediately*. The execution of our program does not wait for the server's response. This is why getTimers() is called an **asynchronous function**.

The success function we pass to getTimers() is called a **callback**. We're saying: "When you finally hear back from the server, if it's a successful response, invoke this function." This asynchronous paradigm ensures that execution of our JavaScript is not **blocked by I/O**.

While our initial instinct might be to try to load the timers from the server in getInitialState(), we don't want a request to the server to hold up the mounting of our app. Instead, we should render a blank canvas for the app by setting timers in state to []. This will allow all components to mount and perform their initial render. Then, we can populate the app by making a request to the server and setting the state:

```
const TimersDashboard = React.createClass({
  getInitialState: function () {
    return {
      timers: [],
    };
  },
  componentDidMount: function () {
    this.loadTimersFromServer();
    setInterval(this.loadTimersFromServer, 5000);
  },
  loadTimersFromServer: function () {
    client.getTimers((serverTimers) => (
      this.setState({ timers: serverTimers })
    ));
  },
});
```

A timeline is the best medium for illustrating what happens:

1. Before initial render

React calls `getInitialState()` on `TimersDashboard`. An object with the property `timers`, a blank array, is returned.

2. The initial render

React then calls `render()` on `TimersDashboard`. In order for the render to complete, `EditableTimerList` and `ToggleableTimerForm` — its two children — must be rendered.

3. Children are rendered

`EditableTimerList` has its `render` method called. Because it was passed a blank data array, it simply produces the following HTML output:

```
<div id='timers'>
</div>
```

`ToggleableTimerForm` renders its HTML, which is the “+” button.

4. Initial render is finished

With its children rendered, the initial render of `TimersDashboard` is finished and the HTML is written to the DOM.

5. `componentDidMount` is invoked

Now that the component is mounted, `componentDidMount()` is called on `TimersDashboard`.

This method calls `loadTimersFromServer()`. In turn, that function calls `client.getTimers()`. That will make the HTTP request to our server, requesting the list of timers. When `client` hears back, it invokes our success function.

On invocation, the success function is passed one argument, `serverTimers`. This is the array of timers returned by the server. We then call `setState()`, which will trigger a new render. The new render populates our app with `EditableTimer` children and all of their children. The app is fully loaded and at an imperceptibly fast speed for the end user.

We also do one other interesting thing in `componentDidMount`. We use `setInterval()` to ensure `loadTimersFromServer()` is called every 5 seconds. While we will be doing our best to mirror state changes between client and server, this hard-refresh of state from the server will ensure our client will always be correct should it shift from the server.

The server is considered the master holder of state. Our client is a mere replica. This becomes incredibly powerful in a multi-instance scenario. If you have two instances of your app running — in two different tabs or two different computers — changes in one will be pushed to the other within five seconds.

Try it out

Let's have fun with this now. Save `app.js` and reload the app. You should see a whole new list of timers, driven by `data.json`. Any action you take will be wiped out within five seconds. Every five seconds, state is restored from the server. For instance, try deleting a timer and witness it resiliently spring back unscathed. Because we're not telling the server about these actions, its state remains unchanged.

On the flip-side, you can try modifying `data.json`. Notice how any modifications to `data.json` will be propagated to your app in under five seconds. Neat.

We're loading the initial state from the server. We have an interval function in place to ensure the client app's state does not drift from the server's in a multi-instance scenario.

We'll need to inform our server of the rest of our state changes: creates, updates (including starts and stops), and deletes. But first, let's pop open the logic behind `client` to see how it works.



While it is indeed neat that changes to our server data is seamlessly propagated to our view, in certain applications — like messaging — five seconds is almost an eternity. We'll cover the concept of **long-polling** in a future app. Long-polling enables changes to be pushed to clients near instantly.

client

If you open up `client.js`, the first method defined in the library is `getTimers()`:

```
function getTimers(success) {
  return fetch('/api/timers', {
    headers: {
      Accept: 'application/json',
    },
  }).then(checkStatus)
    .then(parseJSON)
    .then(success);
}
```

We are using the new **Fetch API** to perform all of our HTTP requests. Fetch's interface should look relatively familiar if you've ever used `XMLHttpRequest` or `jQuery's ajax()`.

Fetch

Until Fetch, JavaScript developers had two options for making web requests: Use XMLHttpRequest which is supported natively in all browsers or import a library that provides a wrapper around it (like jQuery's ajax()). Fetch provides a better interface than XMLHttpRequest. And while Fetch is still undergoing standardization, it is already supported by a few major browsers. At the time of writing, Fetch is turned on by default in Firefox 39 and above and Chrome 42 and above.

Until Fetch is more widely adopted by browsers, it's a good idea to include the library just in case. We've already done so inside index.html:

```
<!-- inside `head` tags index.html -->
<script src="vendor/fetch.js"></script>
```

As we can see in `client.getTimers()`, `fetch()` accepts two arguments:

- The path to the resource we want to fetch
- An object of request parameters

By default, Fetch makes a GET request, so we're telling Fetch to make a GET request to /api/timers. We also pass along one parameter: `headers`, the HTTP headers in our request. We're telling the server we'll *accept* only a JSON response.

Attached to the end of our call to `fetch()`, we have a chain of `.then()` statements:

```
} .then(checkStatus)
  .then(parseJSON)
  .then(success);
```

To understand how this works, let's first review the functions that we pass to each `.then()` statement:

- `checkStatus()`: This function is defined inside of `client.js`. It checks if the server returned an error. If the server returned an error, `checkStatus()` logs the error to the console.
- `parseJSON()`: This function is also defined inside of `client.js`. It takes the response object emitted by `fetch()` and returns a JavaScript object.
- `success()`: This is the function we pass as an argument to `getTimers()`. `getTimers()` will invoke this function if the server successfully returned a response.

Fetch returns a **promise**. While we won't go into detail about promises, as you can see here a promise allows you to chain `.then()` statements. We pass each `.then()` statement a function. What we're essentially saying here is: "Fetching the timers from `/api/timers` *then* check the status code returned by the server. *Then*, extract the JavaScript object from the response. *Then*, pass that object to the success function."

At each stage of the pipeline, the result of the previous statement is passed as the argument to the next one:

1. When `checkStatus()` is invoked, it's passed a Fetch response object that `fetch()` returns.
2. `checkStatus()`, after verifying the response, returns the same response object.
3. `parseJSON()` is invoked and passed the response object returned by `checkStatus()`.
4. `parseJSON()` returns the JavaScript array of timers returned by the server.
5. `success()` is invoked with the array of timers returned by `parseJSON()`.

We could attach an infinite number of `.then()` statements to our pipeline. This pattern enables us to chain multiple function calls together in an easy-to-read format that supports asynchronous functions like `fetch()`.



It's OK if you're still uncomfortable with the concept of promises. We've written all the client code for this chapter for you, so you won't have trouble completing this chapter. You can come back afterwards to play around with `client.js` and get a feel for how it works.

You can read more about JavaScript's Fetch [here³⁰](#) and promises [here³¹](#).

Looking at the rest of the functions in `client.js`, you'll note the methods contain much of the same boilerplate with small differences based on the endpoint of the API we are calling.

We just looked at `getTimers()` which demonstrates *reading* from the server. We'll look at one more function, one that *writes* to the server.

`startTimer()` makes a POST request to the `/api/timers/start` endpoint. The server needs the `id` of the timer and the start time. That request method looks like:

³⁰https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

³¹https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

```
function startTimer(data) {
  return fetch('/api/timers/start', {
    method: 'post',
    body: JSON.stringify(data),
    headers: {
      'Accept': 'application/json',
      'Content-Type': 'application/json',
    },
  }).then(checkStatus);
}
```

In addition to `headers`, the request parameters object that we pass to `fetch()` has two more properties:

```
method: 'post',
body: JSON.stringify(data),
```

Those are:

- `method`: The HTTP request method. `fetch()` defaults to a GET request, so we specify we'd like a POST here.
- `body`: The body of our HTTP request, the data we're sending to the server.

`startTimer()` expects an argument, `data`. This is the object that will be sent along in the body of our request. It contains the properties `id` and `start`. An invocation of `startTimer()` might look like this:

```
// Example invocation of `startTimer()`
startTimer(
{
  id: "bc5ea63b-9a21-4233-8a76-f4bca9d0a042",
  start: 1455584369113,
});

```

In this example, the body of our request to the server will look like this:

```
{  
  "id": "bc5ea63b-9a21-4233-8a76-f4bca9d0a042",  
  "start": 1455584369113  
}
```

The server will extract the id and start timestamp from the body and “start” the timer.

We don’t pass `startTimers()` a success function. Our app does not need data from the server for this request and indeed our server will not return anything besides an “OK” anyway.

`getTimers()` is our only read operation and therefore the only one we’ll pass a success function. The rest of our calls to the server are writes. Let’s implement those now.

Sending starts and stops to the server

We can use the methods `startTimer()` and `stopTimer()` on `client` to make calls to the appropriate endpoints on the server. We just need to pass in an object that includes the `id` of the timer as well as the time it was started/stopped:

```
// Inside TimersDashboard  
// ...  
startTimer: function (timerId) {  
  const now = Date.now();  
  
  this.setState({  
    timers: this.state.timers.map((timer) => {  
      if (timer.id === timerId) {  
        return Object.assign({}, timer, {  
          runningSince: now,  
        });  
      } else {  
        return timer;  
      }  
    }),  
  });  
  
  client.startTimer(  
    { id: timerId, start: now }  
  );  
},  
stopTimer: function (timerId) {  
  const now = Date.now();  
}
```

```

this.setState({
  timers: this.state.timers.map((timer) => {
    if (timer.id === timerId) {
      const lastElapsed = now - timer.runningSince;
      return Object.assign({}, timer, {
        elapsed: timer.elapsed + lastElapsed,
        runningSince: null,
      });
    } else {
      return timer;
    }
  }),
});

client.stopTimer(
  { id: timerId, stop: now }
);
},
render: function () {
  // ...
}

```

You might ask: Why do we still manually make the state change within React? Can't we just inform the server of the action taken and then update state based on the server, the source of truth? Indeed, the following implementation is valid:

```

startTimer: function (timerId) {
  const now = Date.now();

  client.startTimer(
    { id: timerId, start: now }
  ).then(loadTimersFromServer);
},

```

We can chain a `.then()` to `startTimer()` as that function returns our original promise object. The last stage of the `startTimer()` pipeline would then be invoking the function `loadTimersFromServer()`. So immediately after the server processes our start timer request, we would make a subsequent request asking for the latest list of timers. This response would contain the now-running timer. React's state updates and the running timer would then be reflected in the UI.

Again, this is valid. However, the user experience will leave something to be desired. Right now, clicking the start/stop button gives *instantaneous* feedback because the state changes locally and React immediately re-renders. If we waited to hear back from the server, there might be a noticeable

delay between the action (mouse click) and the response (timer starts running). You can try it yourself locally, but the delay would be most noticeable if the request had to go out over the internet.

What we're doing here is called **optimistic updating**. We're updating the client locally before waiting to hear from the server. This duplicates our state update efforts, as we perform updates on both the client and the server. But it makes our app as responsive as possible.



The “optimism” we have here is that the request will succeed and not fail with an error.

Using the same pattern as we did with starts and stops, see if you can implement creates, updates, and deletes on your own. Come back and compare your work with the next section.

Optimistic updating: Validations

Whenever we optimistic update, we always try to replicate whatever restrictions the server would have. This way, our client state changes under the same conditions as our server state.

For example, imagine if our server enforced that a timer's title cannot contain symbols. But the client did not enforce such a restriction. What would happen?

A user has a timer named `Gardening`. He feels a bit cheeky and renames it `Gardening :P`. The UI immediately reflects his changes, displaying `Gardening :P` as the new name of the timer. Satisfied, the user is about to get up and grab his shears. But wait! His timer's name suddenly snaps back to `Gardening`.

To successfully pull off eager updating, we must diligently replicate the code that manages state changes on both the client and the server. Furthermore, in a production app we should surface any errors the request to the server returns in the event that there is an inconsistency in the code or a fluke (the server is down). Implementing this feature for this app is left as an exercise at the end of the chapter.

Sending creates, updates, and deletes to the server

```
// Inside TimersDashboard
// ...
createTimer: function (timer) {
  const t = helpers.newTimer(timer);
  this.setState({
    timers: this.state.timers.concat(t),
  });

  client.createTimer(t);
},
updateTimer: function (attrs) {
  this.setState({
    timers: this.state.timers.map((timer) => {
      if (timer.id === attrs.id) {
        return Object.assign({}, timer, {
          title: attrs.title,
          project: attrs.project,
        });
      } else {
        return timer;
      }
    }),
  });

  client.updateTimer(attrs);
},
deleteTimer: function (timerId) {
  this.setState({
    timers: this.state.timers.filter(t => t.id !== timerId),
  });

  client.deleteTimer(
    { id: timerId }
  );
},
startTimer: function (timerId) {
  // ...
}
```

Recall that, in `createTimer()` and `updateTimer()` respectively, the `timer` and `attrs` objects contain an `id` property, as required by the server.

For creates, we need to send a full timer object. It should have an `id`, a `title`, and a `project`. For updates, we can send an `id` along with just whatever attributes are being updated. Right now, we

always send along `title` and `project` regardless of what has changed. But it's worth noting this difference as it's reflected in the variable names that we are using (`timer` vs `attrs`).

Give it a spin

We are now sending all of our state changes to the server. Save `app.js` and reload the app. Add some timers, start some timers, and refresh and note that everything is persisted. You can even make changes to your app in one browser tab and see the changes propagate to another tab.

Next up

We've worked through a reusable methodology for building React apps and now have an understanding of how we connect a React app to a web server. Armed with these concepts, you're already equipped to build a variety of dynamic web applications.

In imminent chapters, we'll cover a variety of different component types that you encounter across the web (like forms and date pickers). We'll also explore state management paradigms for more complex applications.

Stay tuned!



Chapter Exercises

Right now, if a write to the server fails the app does not surface the issue. There are few ways we could surface that there was an error.

For both exercises below, note that `checkStatus()` will throw an error if the server returns a bad response (a non-200). In a promise chain, you can add a `.catch()` that will be executed if a call in the chain has emitted an error. You can have each `client` method accept an additional argument, `onError`. To have this function invoked in the case a server returns an error, you can insert a `.catch()` call after `.then(checkStatus)`, like this:

```
function createTimer(data, onError) {
  return fetch('/api/timers', {
    // request parameters here
  }).then(checkStatus)
    .catch(onError);
}
```

1. Modify how `TimersDashboard` uses `client` such that an error is displayed to the user if communication with the server fails.

You can use a Semantic UI “Segment” to contain the error message: <http://semantic-ui.com/elements/segment.html#emphasis>³²

2. Building off of the first exercise, modify `TimerForm` so that it accepts new props, `fieldErrors`. `TimersDashboard` can maintain this state. If `TimersDashboard` fails to communicate with the server, it should pop the form submission that failed back open and pass it the errors so that they can be displayed to the user.

Check out Semantic UI’s form field errors for the HTML: <http://semantic-ui.com/collections/form.html#field-error>³³

³²<http://semantic-ui.com/elements/segment.html#emphasis>

³³<http://semantic-ui.com/collections/form.html#field-error>

JSX and the Virtual DOM

React Uses a Virtual DOM

React works differently than many earlier front-end JavaScript frameworks in that instead of working with the **browser's DOM**, it builds a **virtual representation** of the DOM. By **virtual**, we mean a tree of JavaScript objects that *represent* the “actual DOM”. More on this in a minute.

In React, we *do not directly manipulate the actual DOM*. Instead, we must manipulate the virtual representation and let React take care of changing the browser’s DOM.

As we’ll see in this chapter, this is a very powerful feature but it requires us to think differently about how we build web apps.

Why Not Modify the Actual DOM?

It’s worth asking: why do we need a Virtual DOM? Can’t we just use the “actual-DOM”?

When we do “classic-“ (e.g. jQuery-) style web development, we would typically:

1. locate an element (using `document.querySelector` or `document.getElementById`) and then
2. modify that element directly (say, by calling `.innerHTML()` on the element).

This style of development is problematic in that:

- **It’s hard to keep track of changes** - it can become difficult to keep track of current (and prior) state of the DOM to manipulate it into the form we need
- **It can be slow** - modifying the actual-DOM is a costly operation, and modifying the DOM on every change can cause poor performance

What is a Virtual DOM?

The Virtual DOM was created to deal with these issues. But *what is the Virtual DOM anyway?*

The Virtual DOM is a tree of JavaScript objects that represents the actual DOM.

One of the interesting reasons to use the Virtual DOM is the API it gives us. When using the Virtual DOM we code as if we’re **recreating the entire DOM on every update**.

This idea of re-creating the entire DOM results in an easy-to-comprehend development model: instead of the developer keeping track of all DOM state changes, the developer simply returns the DOM *they wish to see*. React takes care of the transformation behind the scenes.

This idea of re-creating the Virtual DOM every update might sound like a bad idea: isn't it going to be slow? In fact, React's Virtual DOM implementation comes with important performance optimizations that make it very fast.

The Virtual DOM will:

- use efficient diffing algorithms, in order to know what changed
- update subtrees of the DOM simultaneously
- batch updates to the DOM

All of this results in an easy-to-use and optimized way to build web apps.

Virtual DOM Pieces

Again, when building a web app in React, we're not working directly with the browser's "actual DOM" directly, but instead a *virtual representation* of it. Our job is to provide React with enough information to build a **JavaScript object** that *represents* what the browser will render.

But what does this Virtual DOM JavaScript object actually consist of?

React's Virtual DOM is a tree of `ReactElements`.

Understanding the Virtual DOM, `ReactElements`, and how they interact with the "actual DOM" is a lot easier to understand by working through some examples, which we'll do below.



Q: Virtual DOM vs. Shadow DOM, are they the same thing? (A: No)

Maybe you've heard of the "Shadow DOM" and you're wondering, is the Shadow DOM the same thing as the Virtual DOM? The answer is **no**.

The *Virtual DOM* is a tree of JavaScript objects that represent the real DOM elements.

The *Shadow DOM* is a form of encapsulation on our elements. Think about using the `<video>` tag in your browser. In a `video` tag, your browser will create a set of video controls such as a play button, a timecode number, a scrubber progress bar etc. These elements aren't part of your "regular DOM", but instead, part of the "Shadow DOM".

Talking about the Shadow DOM is outside the scope of this chapter. But if you want to learn more about the Shadow DOM checkout this article: [Introduction to Shadow DOM³⁴](http://webcomponents.org/articles/introduction-to-shadow-dom/)

³⁴<http://webcomponents.org/articles/introduction-to-shadow-dom/>

ReactElement

A ReactElement is a representation of a DOM element in the Virtual DOM.

React will take these ReactElements and place them into the “actual DOM” for us.

One of the best ways to get an intuition about ReactElement is to play around with it in our browser, so let’s do that now.

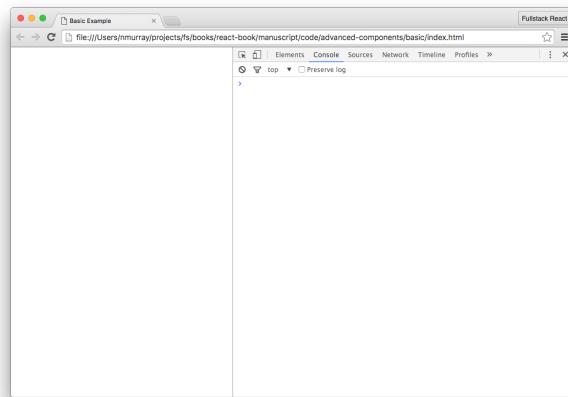
Experimenting with ReactElement



Try this in your browser

For this section, open up the file `code/jsx/basic/index.html` (from the code download) in your browser.

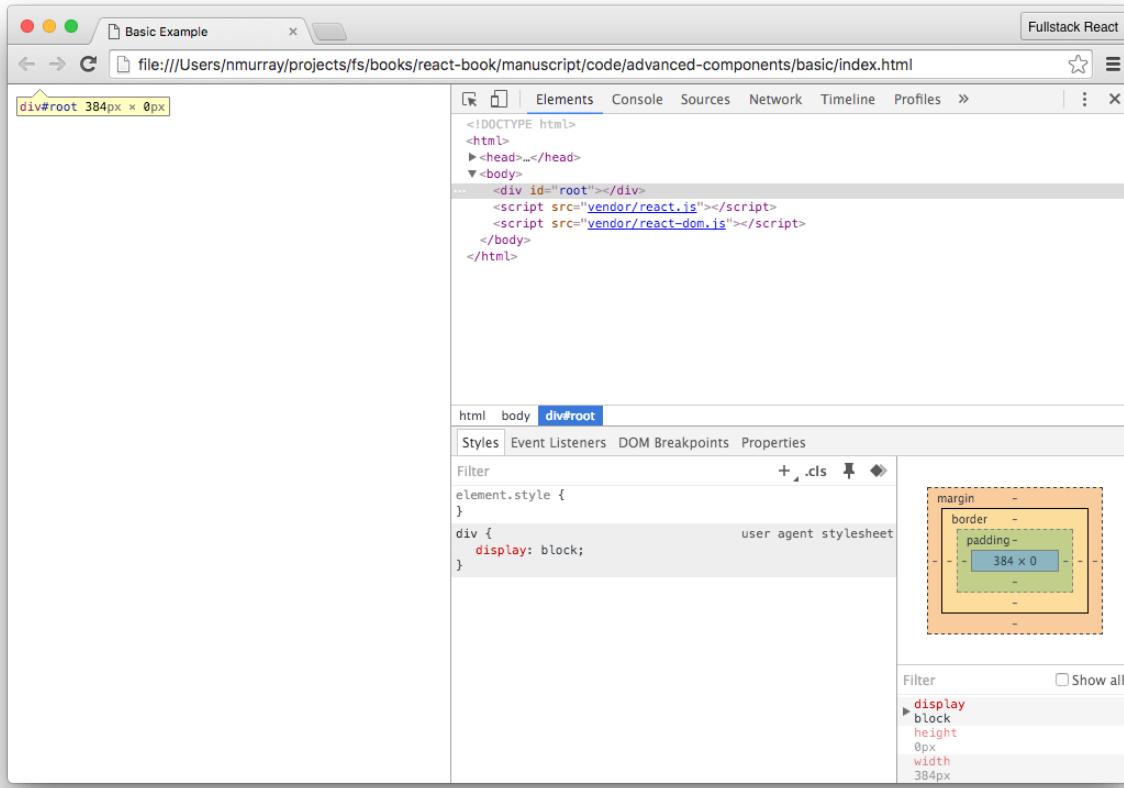
Then open up your developer console and type commands into the console. You can access the console in Chrome by right-clicking and picking “Inspect” and then clicking on “Console” in the inspector.



Basic Console

We’ll start by using a simple HTML template that includes one `<div>` element with an `id` tag:

```
1 <div id='root' />
```



Root Element

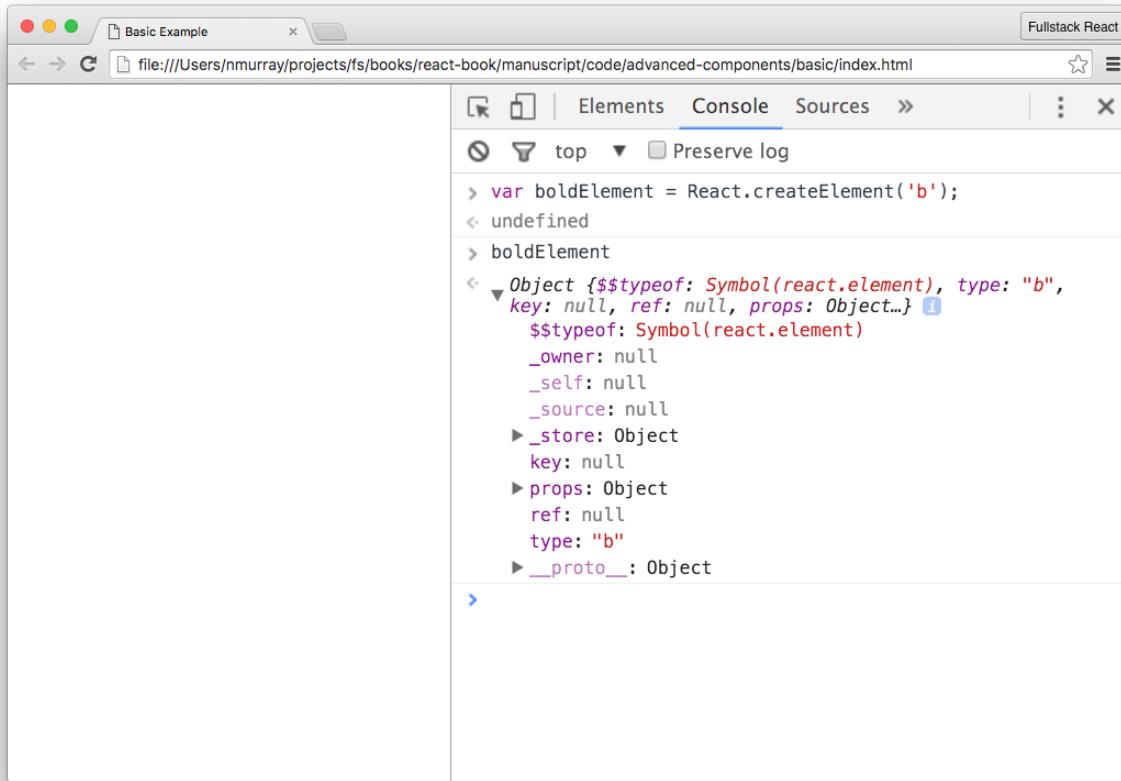
Let's walk through how we render a `` tag in our (actual) DOM using React. Of course, we are **not** going to create a `` tag directly in the DOM (like we might if we were using jQuery).

Instead, React expects us to provide a **Virtual DOM tree**. That is, we're going to give React a set of JavaScript objects which **React will turn into a real DOM tree**.

The objects that make up the tree will be `ReactElements`. To create a `ReactElement`, we use the `createElement` method provided by React.

For instance, to create a `ReactElement` that represents a `` (bold) element in React, type the following in the browser console:

```
1 var boldElement = React.createElement('b');
```



boldElement is a `ReactElement`

Our `boldElement` above is an instance of a `ReactElement`. Now, we have this `boldElement`, but it's not visible without giving it to React to render in the actual DOM tree.

Rendering Our `ReactElement`

In order to render this element to the actual DOM tree we need to use `ReactDOM.render()` (which we cover in more detail [a bit later in this chapter](#)). `ReactDOM.render()` requires two things:

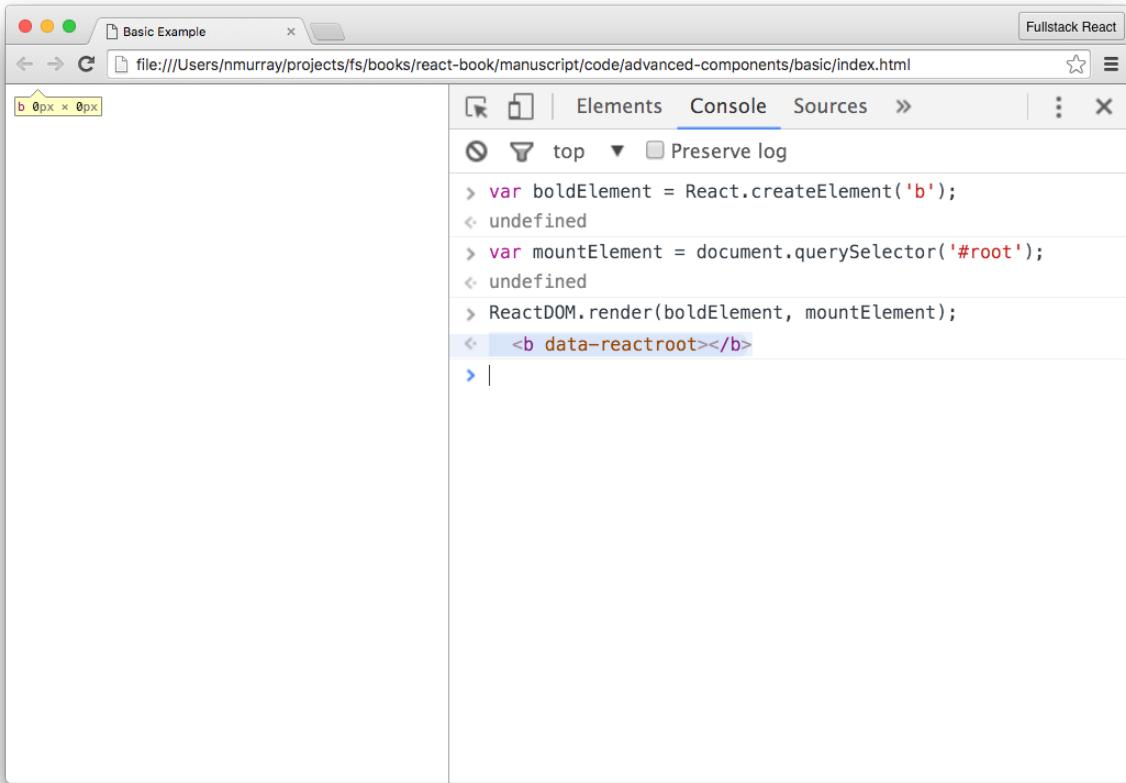
1. The *root* of our virtual tree
2. the *mount location* where we want React write to the *actual* browser DOM

In our simple template we want to get access to the `div` tag with an `id` of `root`. To get a reference to our actual DOM root element, we use one of the following:

```
1 // Either of these will work
2 var mountElement = document.getElementById('root');
3 var mountElement = document.querySelector('#root');
4
5 // if we were using jQuery this would work too
6 var mountElement = $('#root')
```

With our `mountElement` retrieved from the DOM, we can give React a point to insert its own rendered DOM.

```
1 var boldElement = React.createElement('b');
2 var mountElement = document.querySelector('#root');
3 // Render the boldElement in the DOM tree
4 ReactDOM.render(boldElement, mountElement);
```



Despite the fact that nothing appears in the DOM, a new empty element has been inserted into the document as a child of the `mountElement`.



If we click the “Elements” tab in the Chrome inspector, we can see that a **b** tag was created in the actual DOM.

Adding Text (with children)

Although we now have a **b** tag in our DOM, it would be nice if we could add some text in the tag. Because text is in-between the opening and closing **b** tags, adding text is a matter of creating a *child* of the element.

Above, we used `React.createElement` with only a single argument ('**b**' for the **b** tag), however the `React.createElement()` function accepts three arguments:

1. The DOM element type
2. The element props
3. The children of the element

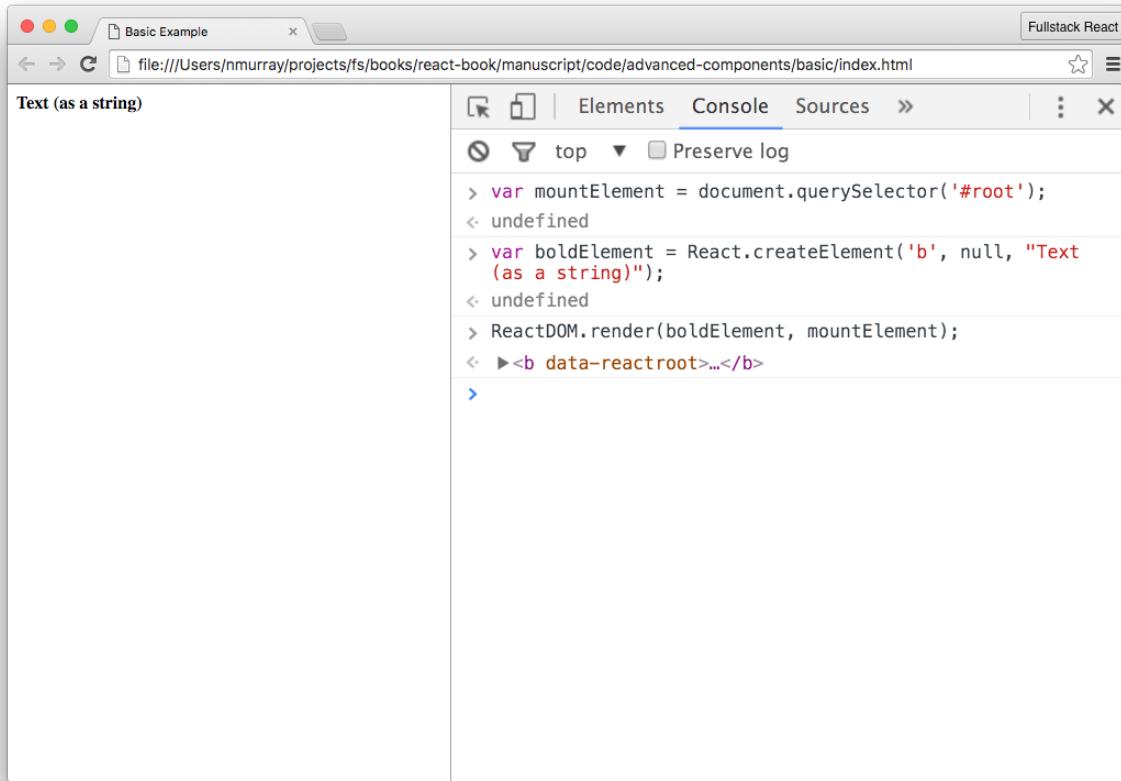
We'll walk through `props` in detail later in this section, so for now we'll set this parameter to `null`.

The `children` of the DOM element must be a `ReactNode` object, which is any of the following:

1. `ReactElement`
2. A string or a number (a `ReactText` object)
3. An array of `ReactNodes`

For example, to place text in our `boldElement`, we can pass a string as the third argument in the `createElement()` function from above:

```
1 var mountElement = document.querySelector('#root');
2 // Third argument is the inner text
3 var boldElement = React.createElement('b', null, "Text (as a string)");
4 ReactDOM.render(boldElement, mountElement);
```



ReactDOM.render()

As we've seen, we use a React renderer places the virtual tree into a "hard" browser view (the "actual" DOM).

But there's a neat side effect of React using it's own virtual representation of the view-tree: it can render this tree in **multiple types of canvases**.

That is, not only can React render into the browser's DOM, but it can also be used to **render views in other frameworks such as mobile apps**. In React Native (which we talk about later in this book), this tree is rendered into *native mobile views*.

That said, in this section we'll spend most of our time in the DOM, so we'll use the ReactDOM renderer to manage elements in the browser DOM.

As we've seen ReactDOM.render() is the way we get our React app into the DOM:

```
1 // ...
2 const component = ReactDOM.render(boldElement, mountElement);
```

We can call the `ReactDOM.render()` function multiple times and it will only perform updates (mutations) to the DOM as necessary.

The `ReactDOM.render` function accepts a 3rd argument: a callback argument that is executed after the component is rendered/updated. We can use this callback as a way to run functions after our app has started:

```
1 ReactDOM.render(boldElement, mountElement, function() {  
2   // The React app has been rendered/updated  
3 });
```

JSX

JSX Creates Elements

When we created our `ReactElement` earlier, we used `React.createElement` like this:

```
1 var boldElement = React.createElement('b', null, "Text (as a string)");
```

This works fine as we had a small component, but if we had many nested components the syntax could get messy very quickly. Our DOM is hierarchical and our React component tree is hierarchical as well.

We can think of it this way: to describe pages to our browser we write HTML; the HTML is parsed by the browser to create HTML Elements which become the DOM.

HTML works very well for specifying tag hierarchies. It would be nice to represent our React component tree using markup, much like we do for HTML.

This is the idea behind JSX.

When using JSX, creating the `ReactElement` objects are handled for us. Instead of calling `React.createElement` for each element, the equivalent structure in JSX is:

```
1 var boldElement = <b>Text (as a string)</b>;  
2 // => boldElement is now a ReactElement
```

The JSX parser will read that string and call `React.createElement` for us.

JSX stands for **JavaScript Syntax Extension**, and it is a syntax React provides that looks a lot like HTML/XML. Rather than building our component trees using normal JavaScript directly, we write our components almost as if we were writing HTML.

JSX provides a syntax that is similar to HTML. However, in JSX we can create our own tags (which encapsulate functionality of other components).

Although it has a scary-sounding name, writing JSX is not much more difficult than writing HTML. For instance, here is a JSX component:

```
1 const element = <div>Hello world</div>;
```

One difference between React components and HTML tags is in the naming. HTML tags start with a lowercase letter, while React components start with an uppercase. For example:

```
1 // html tag
2 const htmlElement = (<div>Hello world</div>);
3
4 // React component
5 const Message = React.createClass({
6   render() {
7     return (<div>{this.props.text}</div>)
8   }
9 });
10
11 // Use our React component with a `Message` tag
12 const reactComponent = (<Message text="Hello world" />);
```

We often surround JSX with parenthesis (). Although this is not always technically required, it helps us set apart JSX from JavaScript.

Our browser doesn't know how to read JSX, so how is JSX possible?

JSX is transformed into JavaScript by using a pre-processor build-tool before we load it with the browser.

When we write JSX, we pass it through a “compiler” (sometimes we say the code is *transpiled*) that converts the JSX to JavaScript. The most common tool for this is a plugin to `babel`, which we’ll cover later.

Besides being able to write HTML-like component trees, JSX provides another advantage: we can mix JavaScript with our JSX markup. This lets us add logic inline with our views.

We’ve seen basic examples of JSX several times in this book already. What is different in this section is that we’re going to take a more structured look at the different ways we can use JSX. We’ll cover tips for using JSX and then talk about how to handle some tricky cases.

Let’s look at:

- attribute expressions
- child expressions
- boolean attributes
- and comments

JSX Attribute Expressions

In order to use a JavaScript expression in a component’s attribute, we wrap it in curly braces {} instead of quotes "".

```

1 // ...
2 const warningLevel = 'debug';
3 const component = (<Alert
4             color={warningLevel === 'debug' ? 'gray' : 'red'}
5             log={true} />)

```

This example uses the [ternary operator³⁵](#) on the `color` prop.

If the `warningLevel` variable is set to `debug`, then the `color` prop will be '`gray`', otherwise it will be '`red`'.

JSX Conditional Child Expressions

Another common pattern is to use a boolean checking expression and then render another element conditionally.

For instance, if we're building a menu that shows options for an `admin` user, we might write:

```

1 // ...
2 const renderAdminMenu = function() {
3   return (<MenuLink to="/users">User accounts</MenuLink>)
4 }
5 // ...
6 const userLevel = this.props.userLevel;
7 return (
8   <ul>
9     <li>Menu</li>
10    {userLevel === 'admin' && renderAdminMenu()}
11  </ul>
12 )

```

We can also use the ternary operator to render one component or another.

For instance, if we want to show a `<UserMenu>` component for a logged in user and a `<LoginLink>` for an anonymous user, we can use this expression:

```
1 const Menu = (<ul>{loggedInUser ? <UserMenu /> : <LoginLink />}</ul>)
```

JSX Boolean Attributes

In HTML, the presence of some attributes sets the attribute to true. For instance, a disabled `<input>` HTML element can be defined:

³⁵https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional_Operator

```
1 <input name='Name' disabled />
```

In React we need to set these as booleans. That is, we need to pass a `true` or `false` explicitly as an attribute:

```
1 // Set the boolean in brackets directly
2 <input name='Name' disabled={true} />
3
4 // ... or use JavaScript variables
5 let formDisabled = true;
6 <input name='Name' disabled={formDisabled} />
```

If we ever need to *enable* the input above, then we set `formDisabled` to `false`.

JSX Comments

We can define comments inside of JSX by using the curly braces (`{}`) with comment delimiters (`/* */`):

```
1 let userLevel = 'admin';
2 /*
3   Show the admin menu if the userLevel is 'admin'
4 */
5 {userLevel === 'admin' && <AdminMenu />}
```

JSX Spread Syntax

Sometimes when we have many props to pass to a component, it can be cumbersome to list each one individually. Thankfully, JSX has a shortcut syntax that makes this easier.

For instance, if we have a `props` object that has two keys: `msg: "Hello"`, `recipient: "World"`

We *could* pass each prop individually like this:

```
1 <Component msg={"Hello"} recipient={"World"} />
```

But by using the *JSX spread syntax* we can do it like this instead:

```

1 <Component {...props} />
2 <!-- essentially the same as this: -->
3 <Component msg={"Hello"} recipient={"World"} />
```

JSX Gotchas

Although JSX mimics HTML, there are a few important differences to pay attention to.

Here's a few things to keep in mind:

JSX Gotcha: `class` and `className`

When we want to set the CSS class of an HTML element, we normally use the `class` attribute in the tag:

```
1 <div class='box'></div>
```

Since JSX is so closely tied to JavaScript, we cannot use identifiers that JavaScript uses in our tag attributes. Attributes such as `for` and `class` conflict with the JavaScript keywords `for` and `class`.

Instead of using `class` to identify a class, JSX uses `className`:

```

1 <!-- Same as <div class='box'></div> -->
2 <div className='box'></div>
```

The `className` attribute works similarly to the `class` attribute in HTML. It expects to receive a string that identifies the class (or classes) associated with a CSS class.

To pass multiple classes in JSX, we can join an array to convert it to a string:

```

1 var cssNames = ['box', 'alert']
2 // and use the array of cssNames in JSX
3 (<div className={cssNames.join(' ')}></div>)
```

Tip: Managing `className` with `classnames`

The `classnames` npm package³⁶ is a great extension that we use to help manage CSS classes. It can take a list of strings or objects and allows us to conditionally apply classes to an element.

The `classnames` package takes the arguments, converts them to an object and conditionally applies a CSS class if the value is truthy.

³⁶<https://www.npmjs.com/package/classnames>

```

1 const App = React.createClass({
2   // ...
3   render: function() {
4     const klass =classnames({
5       box: true, // always apply the box class
6       alert: this.props.isAlert, // if a prop is set
7       severity: this.state.onHighAlert, // with a state
8       timed: false // never apply this class
9     });
10    return (<div className={klass}>
11    )
12  })

```

The package `readme`³⁷ provides alternate examples for more complex environments.

JSX Gotcha: `for` and `htmlFor`

For the same reason we cannot use the `class` attribute, we cannot apply the `for` attribute to a `<label>` element. Instead, we must use the attribute `htmlFor`. The property is a pass-through property in that it applies the attribute as `for`:

```

1 <!-- ... -->
2 <label htmlFor='email'>Email</label>
3 <input name='email' type='email' />
4 <!-- ... -->

```

JSX Gotcha: HTML Entities and Emoji

Entities are reserved characters in HTML which include characters such as less-than `<`, greater-than `>`, the copyright symbol, etc. In order to display entities, we can just place the entity code in literal text.

```

1 <ul>
2   <li>phone: &phone;</li>
3   <li>star: &star;</li>
4 </ul>

```

In order to display entities in dynamic data, we need to surround them in a string inside of curly braces `({})`. Using unicode directly in JS works as expected. Just as we can send our JS to the browser as UTF-8 text directly. Our browser knows how to display UTF-8 code natively.

Alternatively, instead of using the entity character code, we can use unicode version instead.

³⁷<https://github.com/JedWatson/classnames/blob/master/README.md>

```

1  return (
2    <ul>
3      <li>phone: {'\u0260e'}</li>
4      <li>star: {'\u2606'}</li>
5    </ul>
6  )

```

Emoji are just unicode character sequences, so we can add emoji the same way:

```

1  return(
2    <ul>
3      <li>dolphin: {'\uD83D\uDC2C'}</li>
4      <li>dolphin: {'\uD83D\uDC2C'}</li>
5      <li>dolphin: {'\uD83D\uDC2C'}</li>
6    </ul>
7  )

```



Everyone needs more dolphins

JSX Gotcha: data-anything

If we want to apply our own attributes that the HTML spec does not cover, we have to prefix the attribute key with the string `data-`.

```

1  <div className='box' data-dismissible={true} />
2  <span data-highlight={true} />

```

This requirement *only* applies to DOM components that are native to HTML and does not mean custom components cannot accept arbitrary keys as props. That is, we can accept *any* attribute on a custom component:

```
1 <Message dismissable={true} />
2 <Note highlight={true} />
```

There are a standard set of [web accessibility³⁸](#) [attributes³⁹](#) and its a good idea to use them because there are many people who will have a hard time using our site without them. We can use any of these attribute on an element with the key prepended with the string `aria-`. For instance, to set the `hidden` attribute:

```
1 <div aria-hidden={true} />
```

JSX Summary

JSX isn't magic. The key thing to keep in mind is that JSX is syntactic sugar to call `React.createElement`.

JSX is going to parse the tags we write and then create JavaScript objects. JSX is a convenience syntax to help build the component tree.

As we saw earlier, when we use JSX tags in our code, it gets converted to a `ReactElement`:

```
1 var boldElement = <b>Text (as a string)</b>;
2 // => boldElement is now a ReactElement
```

We can pass that `ReactElement` to `ReactDOM.render` and see our code rendered on the page.

There's one problem though: `ReactElement` is stateless and immutable. If we want to add interactivity (with state) into our app, we need another piece of the puzzle: `ReactComponent`.

In the next chapter, we'll talk about `ReactComponents` in depth.

References

Here are some places to read more about JSX and the Virtual DOM:

- [JSX in Depth⁴⁰](#) - (Facebook)
- [If-Else in JSX⁴¹](#) - (Facebook)
- [React \(Virtual\) DOM Terminology⁴²](#) - (Facebook)
- [What is Virtual DOM⁴³](#) - (Jack Bishop)

³⁸<https://www.w3.org/WAI/intro/aria>

³⁹<https://www.w3.org/TR/wai-aria/>

⁴⁰<https://facebook.github.io/react/docs/jsx-in-depth.html>

⁴¹<https://facebook.github.io/react/tips/if-else-in-JSX.html>

⁴²<https://facebook.github.io/react/docs/glossary.html>

⁴³<http://jbi.sh/what-is-virtual-dom/>

Advanced Component Configuration with props, state, and children

Intro

In this chapter we're going to dig deep into the configuration of components.

A `ReactComponent` is a JavaScript object that, at a minimum, has a `render()` function. `render()` is expected to **return a `ReactElement`**.

Recall that `ReactElement` is a representation of a DOM element in the Virtual DOM.



In the [chapter on JSX and the Virtual DOM](#) we talked about `ReactElement` extensively.
Checkout that chapter if you want to understand `ReactElement` better.

The goal of a `ReactComponent` is to

- `render()` a `ReactElement` (which will eventually become the real DOM) and
- attach functionality to this section of the page

“Attaching functionality” is a bit ambiguous; it includes attaching event handlers, managing state, interacting with children, etc. In this chapter we’re going to cover:

- `render()` - the one required function on every `ReactComponent`
- `props` - the “input parameters” to our components
- `context` - a “global variable” for our components
- `state` - a way to hold data that is local to a component (that affects rendering)
- [Stateless components](#) - a simplified way to write reusable components
- `children` - how to interact and manipulate child components
- `statics` - how to create “class methods” on our components

Let's get started!

ReactComponent

Creating ReactComponents - `createClass` or ES6 Classes

There are two ways to define a ReactComponent, either by

1. `React.createClass()`
2. ES6 classes or

These two methods of creating components are roughly equivalent:

```
1 // React.createClass
2 const App = React.createClass({
3   render: function() {} // required method
4 });
5
6 // ES6 class-style
7 class App extends React.Component {
8   render() {} // required
9 }
```



React.createClass vs. ES6 Classes: Which should I use?

There is a lot of debate around whether one should use `createClass` or ES6 classes. The main functional difference is that ES6 classes can't use mixins. Other than that, the preference is largely one of style.

For more details, checkout [Facebook's documentation on this topic⁴⁴](#)

Regardless of the method we used to define the ReactComponent, React expects us to define the `render()` function.

render() Returns a ReactElement Tree

The `render()` method is the only required method to be defined on a ReactComponent.

After the component is **mounted** and **initialized**, `render()` will be called. The `render()` function's job is to provide React a *virtual representation* of a native DOM component.

An example of using `React.createClass` with the `render` function might look like this:

⁴⁴<https://facebook.github.io/react/docs/reusable-components.html>

```
1 const Heading = React.createClass({
2   render: function() {
3     return (
4       <h1>Hello</h1>
5     )
6   }
7});
```

The above code should look familiar. It describes a `Heading` component class with a single `render()` method that returns a simple, single Virtual DOM representation of a `<h1>` tag.

Remember that this `render()` method returns a `ReactElement` which isn't part of the "actual DOM", but instead a description of the Virtual DOM.

React expects the method to return a *single* child element. It can be a virtual representation of a DOM component or can return the falsy value of `null` or `false`. React handles the falsy value by rendering an empty element (a `<noscript />` tag). This is used to remove the tag from the page.

Keeping the `render()` method side-effect free provides an important optimization and makes our code easier to understand.

Getting Data into `render()`

Of course, while `render` is the only required method, it isn't very interesting if the only data we can render is known at compile time. That is, we need a way to:

- input "arguments" into our components and
- maintain state within a component.

React provides ways to do both of these things, with `props` and `state`, respectively.

Understanding these are crucial to making our components dynamic and *useable* within a larger app.

In React, `props` are immutable pieces of **data that are passed into** child components from parents (if we think of our component as the "function" we can think of `props` as our component's "arguments").

Component `state` is where we hold data, local to a component. Typically, when our component's state changes, the component needs to be re-rendered. Unlike `props`, `state` is private to a component and is mutable.

We'll look at both `props` and `state` in detail below. Along the way we'll also talk about `context`, a sort of "implicit `props`" that gets passed through the whole component tree.

Let's look at each of these in more detail.

props are the parameters

props are the inputs to your components. If we think of our component as a “function”, we can think of the props as the “parameters”.

Let's look at an example:

```
1 <div>
2   <Header headerText="Hello world" />
3 </div>
```

In the example code, we're creating both a `<div>` and a `<Header>` element, where the `<div>` is a usual DOM element, while `<Header>` is an instance of our `Header` component.

In this example, we're passing data from the component (the string "Hello world") through the attribute `headerText` to the component.



Passing data through attributes to the component is often called *passing props*.

When we pass data to a component through an attribute it becomes available to the component through the `this.props` property. So in this case, we can access our `headerText` through the property `this.props.headerText`:

```
1 const Header = React.createClass({
2   render: function() {
3     return (
4       <h1>{this.props.headerText}</h1>
5     )
6   }
7});
```

While we can access the `headerText` property, we *cannot* change it.

By using `props` we've taken our static component and allowed it to dynamically render whatever `headerText` is passed into it. The `<Header>` component cannot change the `headerText`, but it can use the `headerText` itself or pass it on to its children.

We can pass any JavaScript object through `props`. We can pass primitives, simple JavaScript objects, atoms, functions etc. We can even pass other React elements and Virtual DOM nodes.

We can document the functionality of our components using `props` and we can specify the *type* of each prop by using `PropTypes`.

PropTypes

PropTypes are a way to validate the values that are passed in through our props. Well-defined interfaces provide us with a layer of safety at the run time of our apps. They also provide a layer of documentation to the consumer of our components.

We define PropTypes by passing them as an option to `createClass()`:

```
1 const Component = React.createClass({
2   propTypes: {
3     name: React.PropTypes.string,
4     totalCount: React.PropTypes.number
5   },
6   // ...
7 })
```

In the example above, our component will validate that `name` is a `string` and that `totalCount` is a `number`.

There are a number of built-in PropTypes, and we can define our own.

We've written a code example for many of the PropTypes validators [here in the appendix on PropTypes](#). For more details on PropTypes, check out that appendix.

For now, we need to know that there are validators for scalar types:

- `string`
- `number`
- `boolean`

We can also validate complex types such as:

- `function`
- `object`
- `array`
- `arrayOf` - expects an array of a particular type
- `node`
- `element`

We can also validate a particular `shape` of an input object, or validate that it is an `instanceOf` a particular class.



Checkout the [appendix on PropTypes](#) for more details and code examples on PropTypes

Default props with getDefaultProps()

Sometimes we want our props to have defaults. We can use the `getDefaultProps()` method to do this.

For instance, create a `Counter` component definition and tell the component that if no `initialValue` is set in the props to set it to 1 using `getDefaultProps()`:

```
1 const Counter = React.createClass({
2   getDefaultProps: function() {
3     return {
4       initialValue: 1
5     }
6   },
7   // ...
8 });
```

Now the component can be used without setting the `initialValue` prop. The two usages of the component are functionally equivalent:

```
1 <Counter />
2 <Counter initialValue={1} />
```

The `getDefaultProps()` method is called once when the class is defined (and cached). The values in the mapped object returned by this method will be set on `this.props` if the prop is not specified by the parent component.

As the `getDefaultProps()` method is invoked before any instances are created, we cannot use any instance variables, such as `this.props` in this method. In addition, any complex objects returned by `getDefaultProps()` are **shared** across all instances, not copied.

context

Sometimes we might find that we have a prop which we want to expose “globally”. In this case, we might find it cumbersome to pass this particular prop down from the root, to every leaf, through every intermediate component.

Instead, specifying context allows us to automatically pass down variables from component to component, rather than needing to pass down our props at every level,



The context feature is experimental and it's similar to using a global variable to handle state in an application - i.e. minimize the use of context as relying on it too frequently is a code smell.

That is, context works best for things that truly are global, such as the central store in Redux.

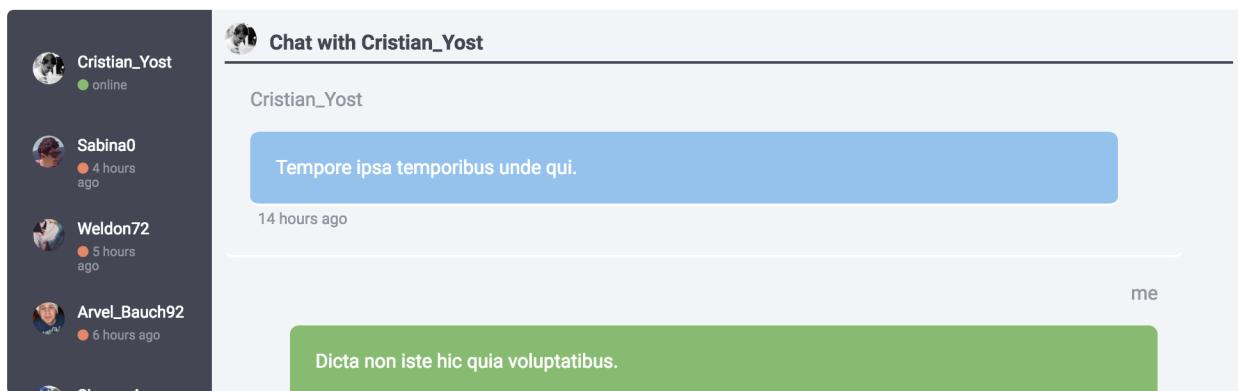
When we specify a context, React will take care of passing down context from component to component so that at any point in the tree hierarchy, any component can reach up to the “global” context where it's defined and get access to the parent's variables.

In order to tell React we want to pass context from a parent component to the rest of its children we need to define two attributes in the parent class:

- `childContextTypes` and
- `getChildContext`

To retrieve the context inside a child component, we need to define the `contextTypes` in the child.

To illustrate, let's look at a possible message reader implementation:



```
1 const Messages = React.createClass({
2   propTypes: {
3     users: PropTypes.array.isRequired,
4     messages: PropTypes.array.isRequired
5   },
6   render: function() {
7     return (
8       <div>
9         <ThreadList />
10        <ChatWindow />
11      </div>
```

```
12      )
13  }
14 });
15 const ThreadList = React.createClass({
16   render: function() {
17     // ...
18   }
19 });
20 const ChatWindow = React.createClass({
21   render: function() {
22     // ...
23   }
24 });
25 const ChatMessage = React.createClass({
26   render: function() {
27     // ...
28   }
29 });
```

Without context, our `MessagesApp` will have to pass the users along with the messages to the two child components (which in turn pass them to their children). Let's set up our hierarchy to accept context instead of needing to pass down `this.props.users` and `this.props.messages` along with every component.

In the `MessagesApp` component, we'll define the two required properties. First, we need to tell React what the *types* of our context.

We define this with the `childContextTypes` key. Similar to `propTypes`, the `childContextTypes` is a key-value object that lists the keys as the name of a context item and the value is a `React.PropTypes`.

Implementing `childContextTypes` in our `MessagesApp` component looks like the following:

```
1 const MessagesApp = React.createClass({
2   childContextTypes: {
3     users: PropTypes.array
4   },
5   // ...
6 });
```

Just like `propTypes`, the `childContextTypes` doesn't populate the context, it just defines it. In order to fill data the `this.context` object, we need to define the second required property function: `getChildContext()`.

The `getChildContext()` function is akin to the `getInitialState()` function in that we can set the values of our context in the function. Back in our `MessagesApp` component, we will set our `users` context object to the value of the `this.props.users` given to the component.

```
1 const MessagesApp = React.createClass({
2   childContextTypes: {
3     users: PropTypes.array
4   },
5   getChildContext: function() {
6     return {
7       users: this.getUsers()
8     }
9   },
10  // ...
11});
```

Since the state and props of a component can change, the context can change as well. The `getChildContext()` method in the parent component gets called every time the state or props change on the parent component. If the context is updated, then the children will receive the updated context and will subsequently be re-rendered.

With the two required properties set on the parent component, React *automatically* passes the object down it's subtree where any component can reach into it. In order to grab the context in a child component, we need to tell React we want access to it. We communicate this to React using the `contextTypes` definition in the child.

Without the `contextTypes` property on the child React component, React won't know what to send our component. Let's give our child components access to the context of our `MessagesApp`.

```
1 const ThreadList = React.createClass({
2   contextTypes: {
3     users: PropTypes.array,
4   },
5   render: function() {
6     // ...
7   }
8 });
9 const ChatWindow = React.createClass({
10  contextTypes: {
11    users: PropTypes.array,
12  },
13  render: function() {
14    // ...
15  }
16 });
17 const ChatMessage = React.createClass({
18  contextTypes: {
```

```

19     users: PropTypes.array,
20   },
21   render: function() {
22     // ...
23   }
24 });

```

Now anywhere in any one of our child components (that have `contextTypes` defined), we can reach into the parent and grab the `users` without needing to pass them along manually via `props`. The context data is set on the `this.context` object of the component with `contextTypes` defined.

For instance, our complete `ThreadList` might look something like:

```

1 const ThreadList = React.createClass({
2   contextTypes: {
3     users: PropTypes.array,
4   },
5
6   render: function() {
7     return (
8       <div>
9         <ul>
10           {this.context.users.map((u, idx) => (
11             <UserListing onClick={this.props.onClick}
12               key={idx}
13               index={idx}
14               user={u} />))
15           </ul>
16         </div>
17       )
18     }
19   })

```

If `contextTypes` is defined on a component, then several of its lifecycle methods will get passed an additional argument of `nextContext`:



contextTypes and Lifecycle methods

We talk about component lifecycle, such as `componentDidUpdate` in the [Component Lifecycle Chapter](#).

```
1 const ThreadList = React.createClass({
2   contextTypes: {
3     users: PropTypes.array,
4   },
5
6   componentWillReceiveProps(nextProps, nextContext) {
7     // ...
8   },
9   shouldComponentUpdate(nextProps, nextState, nextContext) {
10    // ...
11  },
12   componentWillUpdate(nextProps, nextState, nextContext) {
13    // ...
14  },
15   componentDidUpdate(prevProps, prevState, prevContext) {
16    // ...
17  }
18  // ...
19 })
```

In a functional stateless component, context will get passed as the second argument:



We talk about [stateless components](#) below

```
1 const ChatHeader = (props, context) => {
2   const user = props.participants[0];
3   return (
4     <div>
5       <img src={user.avatar} />
6       <div className={styles.chatWith}>Chat with {user.username}</div>
7     </div>
8   )
9 }
10 ChatHeader.propTypes = {participants: PropTypes.array}
11 ChatHeader.contextTypes = {users: PropTypes.array}
```

Using global variables in JavaScript is usually a never good idea and context is usually best reserved for limited situations where a global variable needs to be retrieved, such as a logged-in user. We err on the side of caution in terms of using context in our production apps and tend to prefer props.

state

The second type of data we'll deal with in our components is `state`. To know when to apply state, we need to understand the concept of *stateful* components. Any time a component needs to *hold on to a dynamic piece of data*, that component can be considered stateful.

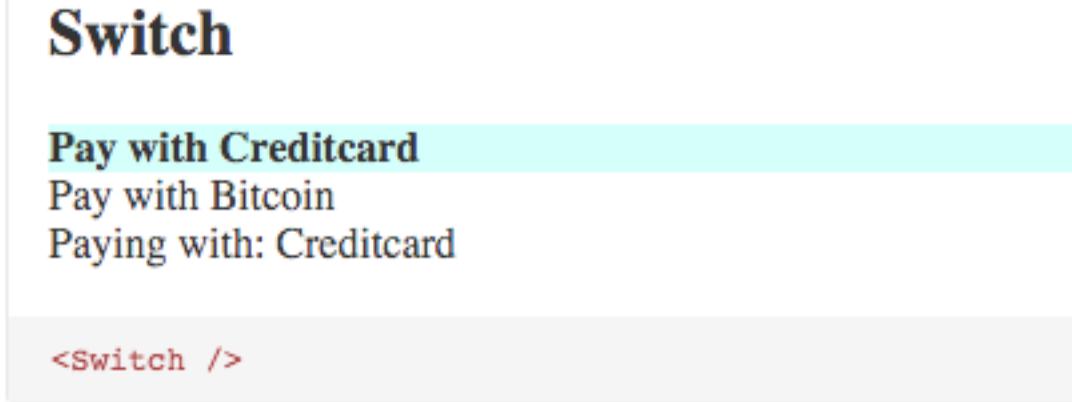
For instance, when a light switch is turned on, that light switch is holding the state of "on." Turning a light off can be described as flipping the state of the light to "off."

In building our apps, we might have a switch that describe a particular setting, such as an input that requires validation, or a presence value for a particular user in a chat application. These are all cases for keeping state about a component within it.

We'll refer to components that hold local-mutable data as *stateful* components. We'll talk a bit more below about when we should use component state. For now, know that it's a good idea to have as **few stateful components as possible**. This is because state introduces complexity and makes composing components more difficult. That said, sometimes we need component-local state, so let's look at how to implement it, and we'll discuss *when* to use it later..

Using `state`: Building a Custom Radio Button

In this example, we're going to use internal state to build a radio button to switch between payment methods. Here's what the form will look like when we're done:



Switch between choices.

Simple Switch

Let's look at how to make a component stateful:

code/advanced-components/components-cookbook/src/components/Switch/steps/Switch1.js

```
3 const Switch = React.createClass({
4   getInitialState() {
5     return {}; // The initial state
6   ,
7
8   render() {
9     return <div><em>Template will be here</em></div>;
10  }
11});
```

That's it! Of course, just setting state on the component isn't all that interesting. To *use* the state on our component, we'll reference it using `this.state`.

code/advanced-components/components-cookbook/src/components/Switch/steps/Switch2.js

```
3 const CREDITCARD = 'Creditcard';
4 const BTC = 'Bitcoin';
5
6 const Switch = React.createClass({
7   getInitialState() {
8     return {
9       payMethod: BTC
10    };
11  },
12
13  render() {
14    return (
15      <div className="switch">
16        <div className="choice">Creditcard</div>
17        <div className="choice">Bitcoin</div>
18        Pay with: {this.state.payMethod}
19      </div>
20    )
21  }
22});
```

In our render function, we can see the choices our users can pick from (although we can't change a method of payment yet) and their current choice stored in the component's state. This `Switch` component is now stateful as it's keeping track of the user's preferred method of payment.

Our payment switch isn't yet interactive; we cannot change the state of the component. Let's hook up our first bit of interactivity by adding an event handler to run when our user selects a different payment method.

In order to add interaction, we'll want to respond to a click event. To add a callback handler to *any* component, we can use the `onClick` attribute on a component. The `onClick` handler will be fired anytime the component it's defined on is clicked.

`code/advanced-components/components-cookbook/src/components/Switch/steps/Switch3.js`

```
21 render() {
22   return (
23     <div className="switch">
24       <div className="choice"
25         onClick={this.select(CREDITCARD)} // add this
26           >Creditcard</div>
27       <div className="choice"
28         onClick={this.select(BTC)}      // ... and this
29           >Bitcoin</div>
30       Pay with: {this.state.payMethod}
31     </div>
32   )
33 }
```

Using the `onClick` attribute, we've attached a callback handler that will be called every time either one of the `<div>` elements are clicked.

The `onClick` handler expects to receive a *function* that it will call when the click event occurs. Let's look at the `select` function:

`code/advanced-components/components-cookbook/src/components/Switch/steps/Switch3.js`

```
6 const Switch = React.createClass({
7   getInitialState() {
8     return {
9       payMethod: BTC
10      };
11    },
12
13   select(choice) {
14     return (evt) => { // <-- handler starts here
15       this.setState({
16         payMethod: choice
17       })
18     }
19   }
20 }
21
22 module.exports = Switch
```

```
18      }
19 },
```

Notice two things about `select`:

1. It returns a function
2. It uses `setState`

Returning a New Function

Notice something interesting about `select` and `onClick`: `onClick` expects a *function* to be passed in, but we're *calling* a function first. That's because the `select` function will *return* a function itself.

This is a common pattern for passing arguments to handlers. We *close over* the choice argument when we call `select`. `select` returns a new function that will call `setState` with the appropriate choice.

When one of the child `<div>` elements are clicked, the handler function will be called. Note that `select` is actually called during `render`, and it's the *return value* of `select` that gets called `onClick`.

Updating the State

When the handler function is called, the component will call `setState` on itself. Calling `setState` triggers a refresh, which means the `render` function will be called again, and we'll be able to see the current state `.payMethod` in our view.



`setState` has performance implications

Since the `setState` method triggers a refresh, we want to be careful about how often we call it.

Modifying the actual-DOM is slow so we don't want to cause a cascade of `setStates` to be called, as that could result in poor performance for our user.

Viewing the Choice

In our component we don't (yet) have a way to indicate which choice has been selected other than the accompanying text.

It would be nice if the choice itself had a visual indication of being the selected one. We usually do this with CSS by applying an `active` class. In our example, we use the `className` attribute.

In order to do this, we'll need to add some logic around which CSS classes to add depending upon the current state of the component.

But before we add too much logic around the CSS, let's refactor component to use a function to render each choice:

code/advanced-components/components-cookbook/src/components/Switch/steps/Switch4.js

```
21 renderChoice(choice) {
22   return (
23     <div className="choice"
24       onClick={this.select(choice)}
25       {choice}
26     </div>
27   )
28 },
29
30 render() {
31   return (
32     <div className="switch">
33       {this.renderChoice(CREDITCARD)}
34       {this.renderChoice(BTC)}
35       Pay with: {this.state.payMethod}
36     </div>
37   )
38 }
```

Now, instead of putting all render code into `render()` function, we isolate the choice rendering into it's own function.

Lastly, let's add the `.active` class to the `<div>` choice component.

code/advanced-components/components-cookbook/src/components/Switch/steps/Switch5.js

```
22 renderChoice(choice) {
23   // create a set of cssClasses to apply
24   let cssClasses = [];
25
26   if (this.state.payMethod === choice) {
27     cssClasses.push(styles.active); // add .active class
28   }
29
30   return (
31     <div className="choice"
32       onClick={this.select(choice)}
33       className={cssClasses}>
34       {choice}
35     </div>
36   )
}
```

```
37     },
38
39     render() {
40       return (
41         <div className="switch">
42           {this.renderChoice(CREDITCARD)}
43           {this.renderChoice(BTC)}
44           Pay with: {this.state.payMethod}
45         </div>
46     )
47 }
```



Notice that we push the style `styles.active` onto the `cssClasses` array. Where did `styles` come from?

For this code example, we're using a webpack loader to import the CSS. Diving in to how webpack works is beyond the scope of this chapter. But just so you know how we're using it, there are two things to know:

1. We're importing the styles like this: `import styles from './Switch.css';`
2. This means all of the styles in that file are accessible like an object - e.g. `styles.active` gives us a reference to the `.active` class from `Switch.css`

We do it this way because it's a form of *CSS encapsulation*. That is, the *actual* CSS class won't actually be `.active`, which means we won't conflict with other components that might use the same class name.

getInitialState()

The `getInitialState()` method serves two purposes:

1. It allows us to define the *initial* state of our component.
2. It tells React that our component will be stateful. Without this method defined, our component will be considered to be stateless.

`getInitialState()` is expected to return a value which can be referenced throughout the component by `this.state`.

```
1 const Component = React.createClass({
2   getInitialState: function() {
3     return {
4       currentValue: 1,
5       currentUser: {
6         name: 'Ari'
7       }
8     }
9   }
10});
```

In this example, the state object is just a JavaScript object, but we can return anything in this function. For instance, we may want to return a single value:

```
1 const Component = React.createClass({
2   getInitialState: function() {
3     return 1;
4   },
5   // ...
6 });
```

This method is the *only* time when we should use props inside of our state. That is, if we ever want to set the value of a prop to the state, we should do it here.

For instance, if we have a component where the prop indicates a value of the component, we should apply that value to the state in the `getInitialState()` method. A better name for the value as a prop is `initialValue`, indicating that the initial state of the value will be set.

For example, consider a `Counter` component that is responsible for counting down the minutes until midnight (like the ball dropping on New Years Eve, for instance):

```
1 const Wrapper = React.createClass({
2   render: function() {
3     return (
4       <div>
5         <Counter initialValue={Date.now()} />
6       </div>
7     )
8   }
9 });
```

From the usage of the `<Counter>` component, we know that the value of the `Counter` will change simply by the name `initialValue`. The `Counter` component can use this prop in the `getInitialState()` method, like so:

```
1 const Counter = React.createClass({
2   getInitialState: function() {
3     return {
4       currentValue: this.props.initialValue
5     }
6   },
7   // ...
8 });


```

The `getInitialState()` method is run only once and before the component itself is actually mounted in the view.

Thinking About State

Spreading state throughout your program can make it difficult to reason about. When building stateful components, we should be mindful about **what** we put in state and **why** we're using state. Generally, we want to minimize the number of components in our apps that keep component-local state.

If we have a component that has UI states which:

1. cannot be “fetched” from outside or
2. cannot be passed into the component,

that's usually a case for building state into the component.

However, any data that can be passed in through `props` or by other components is usually best to leave untouched. The *only* information we should ever put in state are values that are not computed and do not need to be *sync'd* across the app.



The decision to put state in our components or not is deeply related to the tension between “object-oriented programming” and “functional programming”.

In functional programming, if you have a pure function, then calling the same function, with the same arguments, will always return the same value for a given set of inputs. This makes the behavior of a pure function easy to reason about, because the output is consistent at all times, with respect to the inputs.

In object-oriented programming you have objects which hold on to state within that object. The object state then becomes implicit parameters to the methods on the object. The state can change and so calling the same function, with the same arguments, at different times in your program can return different answers.

This is related to `props` and `state` in React components because you can think of `props` as “arguments” to our components and `state` as “instance variables” to an object.

If our component uses only `props` for configuring a component (and it does not use `state` or any other outside variables) then we can easily predict how a particular component will render.

However, if we use mutable, component-local state then it becomes more difficult to understand what a component will render at a particular time.

So while carrying “implicit arguments” through `state` can be convenient, it can also make the system difficult to reason about.

That said, `state` can’t be avoided entirely. The real world has `state`: when you flip a light switch the world has now changed - our programs have to be able to deal with `state` in order to operate in the real world.

The good news is that there are a variety of tools and patterns that have emerged for dealing with `state` in React (notably Flux and its variants), which we talk about elsewhere in the book. The rule of thumb you should work with is to **minimize the number of components with `state`.**

Keeping `state` is usually good to enforce and maintain consistent UI that wouldn’t otherwise be updated. Additionally, one more thing to keep in mind is that we should try to minimize the amount of information we put into our `state`. The smaller and more serializable we can keep it (i.e. can we easily turn it into JSON), the better. Not only will our app be the faster, but it will be easier to reason about. It’s usually a red-flag when our `state` gets large and/or unmanageable.

One way that we can mitigate and minimize the complex `states` is by building our apps with a single stateful component composed of stateless components: components that do not keep `state`.

Stateless Components

An alternative approach to building *stateful* components would be to use *stateless* components. Stateless components are intended to be lightweight components that do not need any special handling around the component.

Stateless components are React's lightweight way of building components that only need the `render()` method.

Let's look at an example of a stateless component:

```
1 const Header = function(props) {  
2   return (<h1>{props.headerText}</h1>)  
3 }
```

Notice that we don't reference `this` when accessing our props as they are simply passed into the function. In fact, the stateless component here isn't actually a class in the sense that it isn't a `ReactElement`.

Functional, stateless components do *not* have a `this` property to reference. In fact, when we use a stateless component, the React rendering processes does *not* introduce a new `ReactComponent` instance, but instead it is null. They are just functions and do not have a backing instance. These components *cannot* contain state and do not get called with the normal component lifecycle methods. We cannot use `refs` (described below), cannot reference the DOM, etc.

React **does** allow us to use `propTypes` and `defaultProps` on stateless components

With so many constraints, why would we want to use stateless components? There are two reasons:

- Minimizing stateful components and
- Performance

As we discussed above, stateful components often spread complexity throughout a system. A component being stateless, when used properly, can help contain the state in fewer locations, which can make our programs easier to reason about.

Also, since React doesn't have to keep track of component instance in memory, do any dirty checking etc., we can get a significant performance increase.

A good rule of thumb is to use stateless components as much as we can. If we don't need any lifecycle methods and can get away with only a rendering function, using a stateless component is a great choice.

Switching to Stateless

Can we convert our `Switch` component above to a stateless component? Well, the currently selected payment choice *is* state and so it has to be kept somewhere.

While we can't remove state completely, we could at least isolate it. This is a common pattern in React apps: try to pull the state into a few parent components.

In our `Switch` component we pulled each choice out into the `renderChoice` function. This indicates that this is a good candidate for pulling into its own stateless component. There's one problem: `renderChoice` is the function that calls `select`, which means that it indirectly is the function that calls `setState`. Let's take a look at how to handle this issue:

code/advanced-components/components-cookbook/src/components/Switch/steps/Switch6.js

```

7  const Choice = function(props) {
8    let cssClasses = [];
9
10   if (props.active) { // <-- check props, not state
11     cssClasses.push(styles.active);
12   }
13
14   return (
15     <div className="choice"
16       onClick={props.onClick /* <-- interesting! */}
17       className={cssClasses}>
18       {props.label} /* <-- allow any label */
19     </div>
20   )
21 }
```

Here we've created a `Choice` function which is *a stateless component*. But we have a problem: if our component is stateless then we can't read from state. What do we do instead? **Pass the arguments down through props**.

In `Choice` we make three changes (which is marked by comments in the code above):

1. We determine if this choice is the active one by reading `props.active`
2. When a `Choice` is clicked, we call whatever function that is on `props.onClick`
3. The label is determined by `props.label`

All of these changes mean that `Choice` is *decoupled* from the `Switch` statement. We could now conceivably use `Choice` anywhere, as long as we pass `active`, `onClick`, and `label` through the `props`.

Let's look at how this changes `Switch`:

code/advanced-components/components-cookbook/src/components/Switch/steps/Switch6.js

```

38  render() {
39    return (
40      <div className="switch">
41        <Choice
42          onClick={this.select(CREDITCARD)}
43          active={this.state.payMethod === CREDITCARD}
44          label="Pay with Creditcard" />
45    
```

```
46      <Choice
47        onClick={this.select(BTC)}
48        active={this.state.payMethod === BTC}
49        label="Pay with Bitcoin" />
50
51      Paying with: {this.state.payMethod}
52    </div>
53  )
54 }
```

Here we're using our `Choice` component and passing the three props (parameters) `active`, `onClick`, and `label`. What's neat about this is that we could easily:

1. Change what happens when we click this choice by changing the input to `onClick`
2. Change the condition by which a particular choice is considered active, but changing the `active` prop
3. Change what the label is to any arbitrary string

By creating a stateless component `Choice` we're able to make `Choice` reusable and not be tied to any particular state.

Stateless Encourages Reuse

Stateless components are a great way to create reusable components. Because stateless components need to have all of their configuration passed from the outside, we can often reuse stateless components in nearly any project, provided that we supply the right hooks.

Now that we've covered both `props`, `context`, and `state` we're going to cover a couple more advanced features we can use with components.

Our components exist in a hierarchy and sometimes we need to communicate (or manipulate) the children components. The the next section, we're going to discuss how to do this.

Talking to Children Components with `props.children`

While we generally specify `props` ourselves, React provides provides some special `props` for us. In our components, we can refer to child components in the tree using `this.props.children`.

For instance, say we have a `Container` that holds an `Article`:

```
1 <Container>
2   <Article headline="An interesting Article">
3     Content Here
4   </Article>
5 </Container>
```

The container component above contains a single child, the `Article` component. How many children does the `Article` component contain? It contains a single child, the text `Content Here`.

In the `Container` component, say that we want to add markup *around* whatever the `Article` component renders. To do this, we write our JSX in the `Container` component, and then place `this.props.children`:

```
1 const Container = React.createClass({
2   render: function() {
3     return (
4       <div className="container">
5         {this.props.children}
6       </div>
7     );
8   }
9});
```

The `Container` component above will create a `div` with `class='container'` and the children of this React tree will render within that `div`.

Generally, React will pass the `this.props.children` prop as an list of components if there are multiple children, whereas it will pass a single element if there is only one component.

Now that we know how `this.props.children` works, we should rewrite the previous `Container` component to use `propTypes` to document the API of our component. We can expect that our `Container` is likely to contain multiple `Article` components, but it might also contain only a single `Article`. So let's specify that the `children` prop can be either an element or an array.



If `PropTypes.oneOfType` seems unfamiliar, checkout [the appendix on PropTypes](#) which explains how it works

```
1 const Container = React.createClass({
2   propTypes: {
3     children: React.PropTypes.oneOfType([
4       React.PropTypes.element,
5       React.PropTypes.array
6     ])
7   },
8   render: function() {
9     return (
10      <div className="container">
11        {this.props.children}
12      </div>
13    );
14  }
15});
```

It can become cumbersome to check what type our `children` prop is every time we want to use `children` in a component. We can handle this a few different ways:

1. Require `children` to be a single child every time (e.g., wrap our children in their own element).
2. Use the `Children` helper provided by React.

The first method of requiring a child to be a single element is straightforward. Rather than defining the `children` above as `oneOfType()`, we can set the `children` to be a single element.

```
1 const Container = React.createClass({
2   propTypes: {
3     children: React.PropTypes.element.isRequired
4   },
5   render: function() {
6     return (
7       <div className="container">
8         {this.props.children}
9       </div>
10      );
11    }
12});
```

Inside the `Container` component we can deal with the `children` *always* being able to be rendered as a single leaf of the hierarchy.

The second method of is to use the `React.Children` utility helper for dealing with the child components. There are a number of helper methods for handling children, let's look at them now.

React.Children.map() & React.Children.forEach()

The most common operation we'll use on children is mapping over the list of them. We'll often use a map to call `React.cloneElement()` or `React.createElement()` along the children.



`map()` and `forEach()`

Both the `map()` and `forEach()` function execute a provided function once per each element in an iterable (either an object or array).

```

1 [1, 2, 3].forEach(function(n) {
2   console.log("The number is: " + n);
3   return n; // we won't see this
4 })
5 [1, 2, 3].map(function(n) {
6   console.log("The number is: " + n);
7   return n; // we will get these
8 })

```

The difference between `map()` and `forEach()` is that the `return` value of `map()` is a an array of the result of the callback function, whereas `forEach()` does not collect results.

So in this case, while both `map()` and `forEach()` will print the `console.log` statements, `map()` will return the array `[1, 2, 3]` whereas `forEach()` will not.

Let's rewrite the previous Container to allow a **configurable wrapper component** for each child. The idea here is that this component takes:

1. A prop `component` which is going wrap each child
2. A prop `children` which is the list of children we're going to wrap

To do this, we call `React.createElement()` to generate a new `ReactElement` for each child:

```

1 const Container = React.createClass({
2   propTypes: {
3     component: React.PropTypes.element.isRequired,
4     children: React.PropTypes.element.isRequired
5   },
6   renderChild: function(childData, index) {
7     return React.createElement(
8       this.props.component,
9       {}, // ~ child props
10      childData // the child's children

```

```
11     );
12   },
13   render: function() {
14     return (
15       <div className="container">
16         {React.children.map(
17           this.props.children,
18           this.renderChild)}
19         </div>
20       );
21     }
22   });
});
```

Again, the difference between `React.Children.map()` and `React.Children.forEach()` is that the former creates an array and returns the result of each function and the latter does not. We'll mostly use `.map()` when we render a child collection.

React.Children.toArray()

`props.children` returns a data structure that can be tricky to work with. Often when dealing with children, we'll want to convert our `props.children` object into a regular array, for instance when we want to re-sort the ordering of the children elements. `React.Children.toArray()` converts the `props.children` data structure into an array of the children.

```
1 const Container = React.createClass({
2   propTypes: {
3     component: React.PropTypes.element.isRequired,
4     children: React.PropTypes.element.isRequired
5   },
6   render: function() {
7     const arr =
8       React.Children.toArray(this.props.children);
9
10    return (
11      <div className="container">
12        {arr.sort((a, b) => a.id < b.id; )}
13        </div>
14      )
15    }
16  });
});
```

ReactComponent Static Methods

Most of what we've talked about so far deals with props or state on the particular *instance* of a component. However, sometimes we want to attach functionality to the "class" of the component that isn't bound to a particular instance.

React provides a way for us to write class methods on our components by using `statics`. Unlike component methods, `statics` can be run without instantiating a component instance.

For example, in a multi-page app, we may have a component for each page. We could use a static `getPageTitle` on each component, which could return the title for that page.

Let's take a two-page app that has a `HomePage` and an `AboutPage` page (without worrying about the routing details yet). We can define the title for each page using `statics`, i.e.:

```
1 const HomePage = React.createClass({
2   statics: {
3     getPageTitle: function() {
4       return 'Home';
5     }
6   },
7   render: function() {
8     return (<div>Welcome home</div>)
9   }
10 })
11 const AboutPage = React.createClass({
12   statics: {
13     getPageTitle: function() {
14       return 'About';
15     }
16   },
17   render: function() {
18     return (<div>About us</div>)
19   }
20 })
```

Using the `statics` definition, we can retrieve these page definitions without instantiating either page by calling the `getPageTitle()` method on the component class.

```
1 // Pseudo-code for routing logic
2 const currentUrl = window.location.pathname;
3 if (currentUrl === '/') {
4   page = HomePage;
5 } else {
6   page = AboutPage;
7 }
8 const title = page.getPageTitle();
9 // ...
```

Summary

By using `props` and `context` we get data in to our components and by using `PropTypes` we can specify clear expectations about what we require that data to be.

By using `state` we hold on to component-local data, and we tell our components to re-render whenever that state changes. However state can be tricky! One technique to minimize stateful components is to use stateless, functional components.

Using these tools we can create powerful interactive components. However there is one important set of configurations that we did not cover here: lifecycle methods.

Lifecycle methods like `componentDidMount` and `componentWillUpdate` provide us with powerful hooks into the application process. In the next chapter, we're going to dig deep into component lifecycle and show how we can use those hooks to validate forms, hook in to external APIs, and build sophisticated components.

References

- [React Top-Level API Docs⁴⁵](https://facebook.github.io/react/docs/top-level-api.html)
- [React Component API Docs⁴⁶](https://facebook.github.io/react/docs/component-api.html)

⁴⁵<https://facebook.github.io/react/docs/top-level-api.html>

⁴⁶<https://facebook.github.io/react/docs/component-api.html>

Forms

Forms 101

Forms are one of the most crucial parts of our application. While we get some interaction through clicks and mouse moves, it's really through forms where we'll get the majority of our rich input from our users.

In a sense, it's where the rubber meets the road. It's through a form that a user can add their payment info, search for results, edit their profile, upload a photo, or send a message. Forms transform your web site into a web app.

Forms can be deceptively simple. All you really need are some `input` tags and a `submit` tag wrapped up in a `form` tag. However, creating a rich, interactive, easy to use form can often involve a significant amount of programming:

- Form inputs modify data, both on the page and the server.
- Changes often have to be kept in sync with data elsewhere on the page.
- Users can enter unpredictable values, some that we'll want to modify or reject outright.
- The UI needs to clearly state expectations and errors in the case of validation failures.
- Fields can depend on each other and have complex logic.
- Data collected in forms is often sent asynchronously to a back-end server, and we need to keep the user informed of what's happening.
- We want to be able to test our forms.

If this sounds daunting, don't worry! This is exactly why React was created: to handle the complicated forms that needed to be built at Facebook.

We're going to explore how to handle these challenges with React by building a sign up app. We'll start simple and add more functionality in each step.

Preparation

Inside the code download that came with this book, navigate to `forms`:

```
$ cd forms
```

This folder contains all the code examples for this chapter. To view them in your browser install the dependencies by running `npm install` (or `npm i` for short):

```
$ npm i
```

Once that's finished, you can start the app with `npm start`:

```
$ npm start
```

You should expect to see the following in your terminal:

```
$ npm start
```

```
> forms@1.0.0 start /code/forms
> budo index.jsx -l -- -t [ babelify --presets [ react es2015 ] ]

[0000] info Server running at http://192.168.1.100:9966/ (connect)
[0000] info LiveReload running on 35729
[0006] 5562ms    1.8MB (browserify)
```



To run the app we're using [Budo⁴⁷](#), a dev server for rapid prototyping.

The `npm start` command (visible above or in `package.json`) starts `budo` by passing it the entry point (`index.jsx`), the “live reload” flag (`-l`), and the `babelify` transform with `react` and `es2015` presets enabled (`-t [babelify --presets [react es2015]]`).

For more information see [Budo’s Command Line Usage⁴⁸](#)

You should now be able to see the app in your browser if you go to [http://localhost:9966⁴⁹](http://localhost:9966).

The Basic Button

At their core, forms are a conversation with the user. Fields are the app’s questions, and the values that the user inputs are the answers.

Let’s ask the user what they think of React.

We could present the user with a text box, but we’ll start even simpler. In this example, we’ll constrain the response to just one of two possible answers. We want to know whether the user thinks React is either “great” or “amazing”, and the simplest way to do that is to give them two buttons to choose from.

Here’s what the first example looks like:

⁴⁷<https://github.com/mattdesl/budo>

⁴⁸<https://github.com/mattdesl/budo/blob/master/docs/command-line-usage.md>

⁴⁹<http://localhost:9966>

What do you think of React?

Great Amazing

Basic Buttons

To get our app to this stage we create a component with a `render()` method that returns a `div` with three child elements: an `h1` with the question, and two `button` elements for the answers. This will look like the following:

code/forms/01-basic-button.jsx

```
17  render() {
18    return (
19      <div>
20        <h1>What do you think of React?</h1>
21
22        <button
23          name='button-1'
24          value='great'
25          onClick={this.onGreatClick}
26        >
27          Great
28        </button>
29
30        <button
31          name='button-2'
32          value='amazing'
33          onClick={this.onAmazingClick}
34        >
35          Amazing
36        </button>
37      </div>
38    );
39  },
```

So far this looks a lot like how you'd handle a form with vanilla HTML. The important part to pay attention to is the `onClick` prop of the `button` elements. When a button is clicked, if it has a function

set as its `onClick` prop, that function will be called. We'll use this behavior to know what our user's answer is.

To know what our user's answer is, we pass a different function to each button. Specifically, we'll create function `onGreatClick()` and provide it to the "Great" button and create function `onAmazingClick()` and provide it to the "Amazing" button.

Here's what those functions look like:

`code/forms/01-basic-button.jsx`

```
9  onGreatClick(evt) {  
10    console.log('The user clicked button-1: great', evt);  
11  },  
12  
13  onAmazingClick(evt) {  
14    console.log('The user clicked button-2: amazing', evt);  
15  },
```

When the user clicks on the "Amazing" button, the associated `onClick` function will run (`onAmazingClick()` in this case). If, instead, the user clicked the "Great" button, `onGreatClick()` would be run instead.



Notice that in the `onClick` handler we pass `this.onGreatClick` and *not* `this.onGreatClick()`.

What's the difference?

In the first case (without parens), we're passing the *function* `onGreatClick`, whereas in the second case we're passing the *result of calling the function* `onGreatClick` (which isn't what we want right now).

This becomes the foundation of our app's ability to respond to a user's input. Our app can do different things depending on the user's response. In this case, we log different messages to the console.

Events and Event Handlers

Note that our `onClick` functions (`onAmazingClick()` and `onGreatClick()`) accept an argument, `evt`. This is because these functions are *event handlers*.

Handling events is central to working with forms in React. When we provide a function to an element's `onClick` prop, that function becomes an event handler. The function will be called when that event occurs, and it will receive an **event object as its argument**.

In the above example, when the button element is clicked, the corresponding event handler function is called (`onAmazingClick()` or `onGreatClick()`) and it is provided with a mouse click event object

(`evt` in this case). This object is a `SyntheticMouseEvent`. This `SyntheticMouseEvent` is just a cross-browser wrapper around the browser's native `MouseEvent`, and you'll be able to use it the same way you would a native DOM event. In addition, if you need the original native event you can access it via the `nativeEvent` attribute (e.g. `evt.nativeEvent`).

Event objects contain lots of useful information about the action that occurred. A `MouseEvent` for example, will let you see the `x` and `y` coordinates of the mouse at the time of the click, whether or not the shift key was pressed, and (most useful for this example) a reference to the element that was clicked. We'll use this information to simplify things in the next section.



Instead, if we were interested in mouse movement, we could have created an event handler and provided it to the `onMouseMove` prop. In fact, there are many such element props that you can provide mouse event handlers to: `onClick`, `onContextMenu`, `onDoubleClick`, `onDrag`, `onDragEnd`, `onDragEnter`, `onDragExit`, `onDragLeave`, `onDragOver`, `onDragStart`, `onDrop`, `onMouseDown`, `onMouseEnter`, `onMouseLeave`, `onMouseMove`, `onMouseOut`, `onMouseOver`, and `onMouseUp`.

And those are only the mouse events. There are also clipboard, composition, keyboard, focus, form, selection, touch, ui, wheel, media, image, animation, and transition event groups. Each group has its own types of events, and not all events are appropriate for all elements. For example, here we will mainly work with the form events, `onChange` and `onSubmit`, which are related to `form` and `input` elements.

For more information on events in React, see React's documentation on the [Event System](#)⁵⁰.

Back to the Button

In the previous section, we were able to perform different actions (log different messages) depending on the action of the user. However, the way that we set it up, we'd need to create a separate function for each action. Instead, it would be much cleaner if we provided the same event handler to both buttons, and used information from the event itself to determine our response.

To do this, we replace the two event handlers `onGreatClick()` and `onAmazingClick()` with a new single event handler, `onButtonClick()`.

`code/forms/02-basic-button.jsx`

```
9  onButtonClick(evt) {
10    const btn = evt.target;
11    console.log(`The user clicked ${btn.name}: ${btn.value}`);
12  },
```

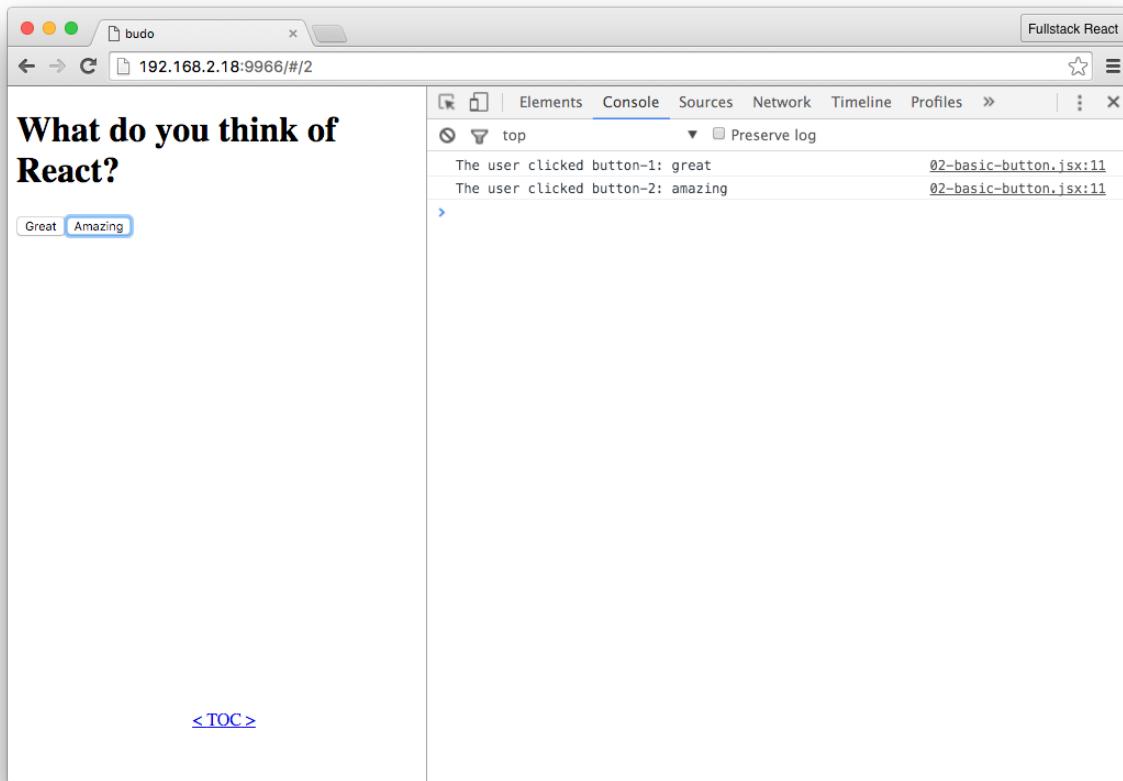
⁵⁰<https://facebook.github.io/react/docs/events.html>

Our click handler function receives an event object, evt. evt has an attribute target that is a reference to the button that the user clicked. This way we can access the button that the user clicked without creating a function for each button. We can then log out different messages for different user behavior.

Next we update our render() function so that our button elements both use the same event handler, our new onButtonClick() function.

code/forms/02-basic-button.jsx

```
14  render() {
15    return (
16      <div>
17        <h1>What do you think of React?</h1>
18
19        <button
20          name='button-1'
21          value='great'
22          onClick={this.onButtonClick}
23        >
24          Great
25        </button>
26
27        <button
28          name='button-2'
29          value='amazing'
30          onClick={this.onButtonClick}
31        >
32          Amazing
33        </button>
34      </div>
35    );
36  },
```



One Event Handler for Both Buttons

By taking advantage of the event object and using a shared event handler, we could add 100 new buttons, and we wouldn't have to make any other changes to our app.

Text Input

In the previous example, we constrained our user's response to only one of two possibilities. Now that we know how to take advantage of event objects and handlers in React, we're going to accept a much wider range of responses and move on to a more typical use of forms: text input.

To showcase text input we'll create a “sign-up sheet” app. The purpose of this app is to allow a user to record a list of names of people who want to sign up for an event.

The app presents the user a text field where they can input a name and hit “Submit”. When they enter a name, it is added to a list, that list is displayed, and the text box is cleared so they can enter a new name.

Here's what the initial version looks like:

Sign Up Sheet

Names

- Nate Murray
- Ari Lerner

Sign-Up v1

Accessing User Input With `refs`

We want to be able to read the contents of the text field when the user submits the form. A simple way to do this is to wait until the user submits the form, find the text field in the DOM, and finally grab its value.

To begin we'll start by creating a form element with two child elements: a text input field and a submit button:

`code/forms/03-basic-input.jsx`

```
14  render() {
15    return (
16      <div>
17        <h1>Sign Up Sheet</h1>
18
19        <form onSubmit={this.onFormSubmit}>
20          <input
21            placeholder='Name'
22            ref='name'
23          />
24
25          <input type='submit' />
26        </form>
27      </div>
```

```
28     );
29 },
```

This is very similar to the previous example, but instead of two button elements, we now have a `form` element with two child elements: a text field and a submit button.

There are two things to notice. First, we've added an `onSubmit` event handler to the `form` element. Second, we've given the text field a `ref` prop of '`'name'`'.

By using an `onSubmit` event handler on the `form` element this example will behave a little differently than before. One change is that the handler will be called either by clicking the "Submit" button, or by pressing "enter"/"return" while the `form` has focus. This is more user-friendly than forcing the user to click the "Submit" button.

However, because our event handler is tied to the `form`, the event object argument to the handler is less useful than it was in the previous example. Before, we were able to use the `target` prop of the event to reference the `button` and get its value. This time, we're interested in the text field's value. One option would be to use the event's `target` to reference the `form` and from there we could find the child `input` we're interested in, but there's a simpler way.

In React, if we want to easily access a DOM element in a component we can use `refs` (references). Above, we gave our text field a `ref` property of '`'name'`'. Later when the `onSubmit` handler is called, we have the ability to access `this.refs.name` to get a reference to that text field. Here's what that looks like in our `onFormSubmit()` event handler:

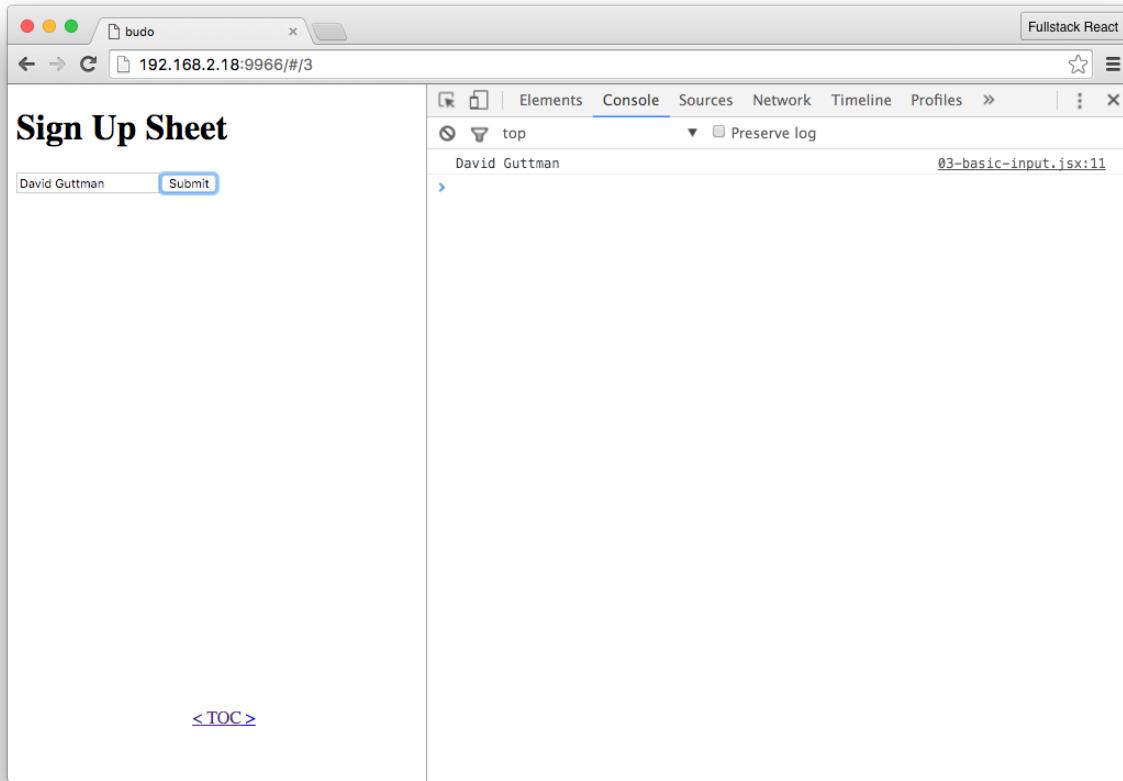
`code/forms/03-basic-input.jsx`

```
9  onFormSubmit(evt) {
10    evt.preventDefault();
11    console.log(this.refs.name.value);
12 },
```



Use `preventDefault()` with the `onSubmit` handler to prevent the browser's default action of submitting the form.

As you can see, by using `this.refs.name` we gain a reference to our text field element and we can access its `value` property. That `value` property contains the text that was entered into the field.



Logging The Name

With just the two functions `render()` and `onFormSubmit()`, we should now be able to see the value of the text field in our console when we click “Submit”. In the next step we’ll take that value and display it on the page.

Using User Input

Now that we’ve shown that we can get user submitted names, we can begin to use this information to change the app’s state and UI.

The goal of this example is to show a list with all of the names that the user has entered. React makes this easy. We will have an array in our state to hold the names, and in `render()` we will use that array to populate a list.

When our app loads, the array will be empty, and each time the user submits a new name, we will add it to the array. To do this, we’ll make a few additions to our component.

First, we’ll create a `names` array in our state. In React, we can set the initial value of our state object by defining a `getInitialState()` function. Our `getInitialState()` function should return an object, and that object will be come the initial value of state. To add a `names` array to our state

our `getInitialState()` function should return an object like `{ names: [] }`. Here's what that looks like:

code/forms/04-basic-input.jsx

```
9  getInitialState() {
10    return { names: [] };
11  },
```

Next, we'll modify `render()` to show this list. Below our `form` element, we'll create a new `div`. This new container `div` will hold a heading, `h3`, and our names list, a `ul` parent with a `li` child for each name. Here's our updated `render()` method:

code/forms/04-basic-input.jsx

```
21 render() {
22   return (
23     <div>
24       <h1>Sign Up Sheet</h1>
25
26       <form onSubmit={this.onFormSubmit}>
27         <input
28           placeholder='Name'
29           ref='name'
30           />
31
32         <input type='submit' />
33       </form>
34
35       <div>
36         <h3>Names</h3>
37         <ul>
38           { this.state.names.map((name, i) => <li key={i}>{name}</li>) }
39         </ul>
40       </div>
41     </div>
42   );
43 },
```

ES2015 gives us a compact way to insert `li` children. Since `this.state.names` is an array, we can take advantage of its `map()` method to return a `li` child element for each name in the array. Also, by using “arrow” syntax, for our iterator function in `map()`, the `li` element is returned without us explicitly using `return`.



One other thing to note here is that we provide a key prop to the `li` element. React will complain when we have children in an array or iterator (like we do here) and they don't have a key prop. React wants this information to keep track of the child and make sure that it can be reused between render passes.

We won't be removing or reordering the list here, so it is sufficient to identify each child by its index. If we wanted to optimize rendering for a more complex use-case, we could assign an immutable id to each name that was not tied to its value or order in the array. This would allow React to reuse the element even if its position or value was changed.

See React's documentation on [Multiple Components and Dynamic Children⁵¹](#) for more information.

Now that `render()` is updated, the `onFormSubmit()` method needs to update the state with the new name. To add a name to the `names` array in our state we might be tempted to try to do something like `this.state.names.push(name)`. However, React relies on `this.setState()` to mutate our state object, which will then trigger a new call to `render()`.

The way to do this properly is to:

1. create a new variable that copies our current `names`
2. add our new name to that array, and finally
3. use that variable in a call to `this.setState()`.

We also want to clear the text field so that it's ready to accept additional user input. It would not be very user friendly to require the user to delete their input before adding a new name. Since we already have access to the text field via `refs`, we can set its `value` to an empty string to clear it.

This is what `onFormSubmit()` should look like now:

[code/forms/04-basic-input.jsx](#)

```
13  onFormSubmit(evt) {
14    const name = this.refs.name.value;
15    const names = [ ...this.state.names, name ];
16    this.setState({ names: names });
17    this.refs.name.value = '';
18    evt.preventDefault();
19  },
```

At this point, our sign-up app is functional. Here's a rundown of the application flow:

1. User enters a name and clicks "Submit".

⁵¹<https://facebook.github.io/react/docs/multiple-components.html#dynamic-children>

2. `onFormSubmit` is called.
3. `this.refs.name` is used to access the value of the text field (a name).
4. The name is added to our `names` list in the state.
5. The text field is cleared so that it is ready for more input.
6. `render` is called and displays the updated list of names.

So far so good! In the next sections we'll improve it further.

Uncontrolled vs. Controlled Components

In the previous sections we took advantage of `refs` to access the user's input. When we created our `render()` method we added an `input` field with a `ref` attribute. We later used that attribute to get a reference to the rendered `input` so that we could access and modify its `value`.

We covered using `refs` with forms because it is conceptually similar to how one might deal with forms without React. However, by using `refs` this way, we opt out of a primary advantage of using React.

In the previous example we access the DOM directly to retrieve the name from the text field, as well as manipulate the DOM directly by resetting the field after a name has been submitted.

With React we shouldn't have to worry about modifying the DOM to match application state. We should concentrate only on altering state and rely on React's ability to efficiently manipulate the DOM to match. This provides us with the certainty that for any given value of state, we can predict what `render()` will return and therefore know what our app will look like.

In the previous example, our text field is what we would call an "uncontrolled component". This is another way of saying that React does not "control" how it is rendered – specifically its `value`. In other words, React is hands-off, and allows it to be freely influenced by user interaction. This means that knowing the application state is not enough to predict what the page (and specifically the `input` field) looks like. Because the user could have typed (or not typed) input into the field, the only way to know what the `input` field looks like is to access it via `refs` and check its `value`.

There is another way. By converting this field to a "controlled component", we give React control over it. Its `value` will always be specified by `render()` and our application state. When we do this, we can predict how our application will look by examining our `state` object.

By directly tying our view to our application state we get certain features for very little work. For example, imagine a long `form` where the user must answer many questions by filling out lots of `input` fields. If the user is halfway through and accidentally reloads the page that would ordinarily clear out all the fields. If these were controlled components and our application state was persisted to `localStorage`, we would be able to come back exactly where they left off. Later, we'll get to another important feature that controlled components pave the way for: validation.

Accessing User Input With state

Converting an uncontrolled input component to a controlled one requires three things. First, we need a place in state to store its value. Second, we provide that location in state as its value prop. Finally, we add an onChange handler that will update its value in state. The flow for a controlled component looks like this:

1. The user enters/changes the input.
2. The onChange handler is called with the “change” event.
3. Using event.target.value we update the input element’s value in state.
4. render() is called and the input is updated with the new value in state.

Here’s what our render() looks like after converting the input to a controlled component:

code/forms/05-state-input.jsx

```
26  render() {
27    return (
28      <div>
29        <h1>Sign Up Sheet</h1>
30
31        <form onSubmit={this.onFormSubmit}>
32          <input
33            placeholder='Name'
34            value={this.state.name}
35            onChange={this.onNameChange}
36          />
37
38          <input type='submit' />
39        </form>
40
41        <div>
42          <h3>Names</h3>
43          <ul>
44            { this.state.names.map((name, i) => <li key={i}>{name}</li>) }
45          </ul>
46        </div>
47      </div>
48    );
49  },
```

The only difference in our input is that we’ve removed the ref prop and replaced it with both a value and an onChange prop.

Now that the `input` is “controlled”, its `value` will always be set equal to a property of our `state`. In this case, that property is `name`, so the `value` of the `input` is `this.state.name`.

While not strictly necessary, it’s a good habit to provide sane defaults for any properties of `state` that will be used in our component. Because we now use `state.name` for the `value` of our `input`, we’ll want to choose what value it will have before the user has had a chance to provide one. In our case, we want the field to be empty, so the default value will be an empty string, `''`:

`code/forms/05-state-input.jsx`

```
9  getInitialState() {
10    return {
11      name: '',
12      names: [],
13    };
14  },
```

If we had just stopped there, the `input` would effectively be disabled. No matter what the user types into it, its `value` wouldn’t change. In fact, if we left it like this, React would give us a warning in our console.

To make our `input` operational, we’ll need to listen to its `onChange` events and use those to update the `state`. To achieve this, we’ve created an event handler for `onChange`. This handler is responsible for updating our `state` so that `state.name` will always be updated with what the user has typed into the field. We’ve created the method `onNameChange()` for that purpose.

Here’s what `onNameChange()` looks like now:

`code/forms/05-state-input.jsx`

```
22  onNameChange(evt) {
23    this.setState({ name: evt.target.value });
24  },
```

`onNameChange()` is a very simple function. Like we did in a previous section, we use the event passed to the handler to reference the field and get its `value`. We then update `state.name` with that `value`.

Now the controlled component cycle is complete. The user interacts with the field. This triggers an `onChange` event which calls our `onNameChange()` handler. Our `onNameChange()` handler updates the `state`, and this in turn triggers `render()` to update the field with the new `value`.

Our app still needs one more change, however. When the user submits the form, `onFormSubmit()` is called, and we need that method to add the entered name (`state.name`) to the `names` list (`state.names`). When we last saw `onFormSubmit()` it did this using `this.refs`. Since we’re no longer using `refs`, we’ve updated it to the following:

code/forms/05-state-input.jsx

```
16  onFormSubmit(evt) {
17    const names = [ ...this.state.names, this.state.name ];
18    this.setState({ names: names, name: '' });
19    evt.preventDefault();
20  },
```

Notice that to get the current entered name, we simply access `this.state.name` because it will be continually updated by our `onNameChange()` handler. We then append that to our `names` list, `this.state.names` and update the state. We also clear `this.state.name` so that the field is empty and ready for a new name.

While our app didn't gain any new features in this section, we've both paved the way for better functionality (like validation and persistence) while also taking greater advantage of the React paradigm.

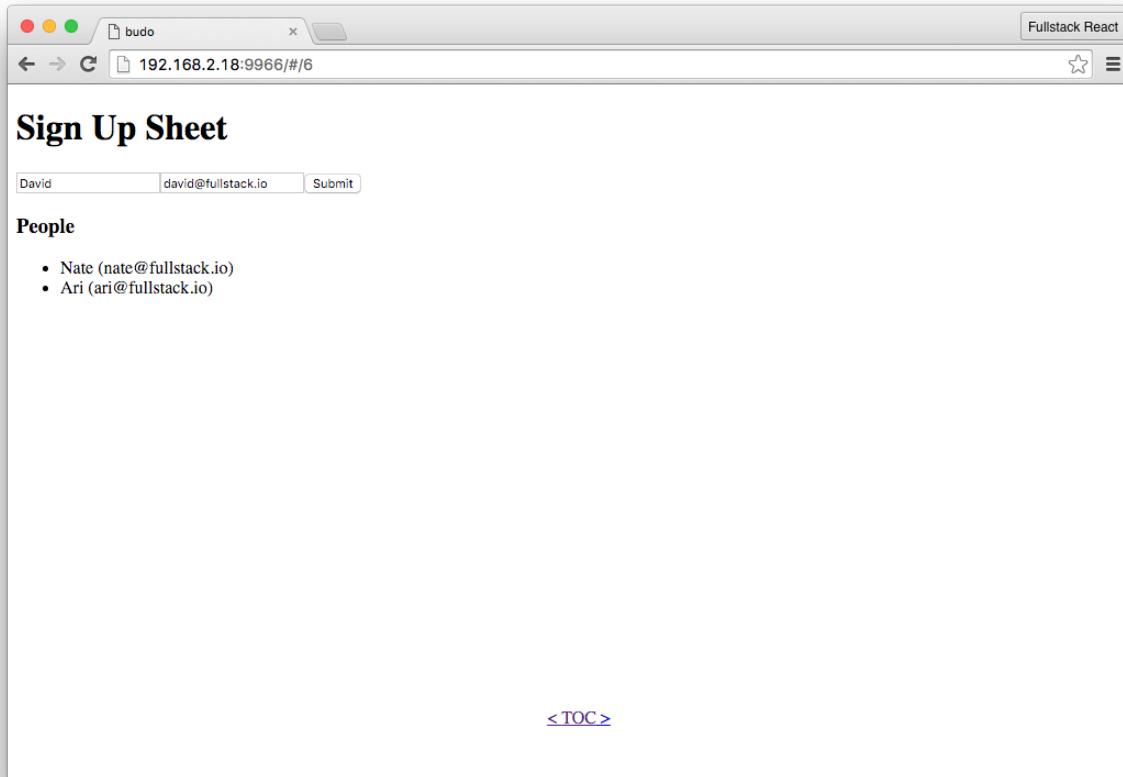
Multiple Fields

Our sign-up sheet is looking good, but what would happen if we wanted to add more fields? If our sign-up sheet is like most projects, it's only a matter of time before we want to add to it. With forms, we'll often want to add inputs.

If we continue our current approach and create more controlled components, each with a corresponding state property and an `onChange` handler, our component will become quite verbose. Having a one-to-one-to-one relationship between our inputs, state, and handlers is not ideal.

Let's explore how we can modify our app to allow for additional inputs in a clean, maintainable way. To illustrate this, let's add email address to our sign-up sheet.

In the previous section our `input` field has a dedicated property on the root of our `state` object. If we were to do that here, we would add another property, `email`. To avoid adding a property for each input on `state`, let's instead add a `fields` object to store the values for all of our fields in one place. This object can store state for as many inputs as we'd like. Now, we will find those values at `state.fields.name` and `state.fields.email` instead of `state.name` and `state.email`.



Name and Email Fields

Of course, those values will need to be updated by an event handler. We *could* create an event handler for each field we have in the form, but that would involve a lot of copy/paste and needlessly bloat our component. Also it would make maintainability more difficult, as any change to a form would need to be made in multiple places.

Instead of creating an `onChange` handler for each input, we can create a single method that accepts change events from **all** of our inputs. The trick is to write this method in such a way that it updates the correct property in state depending on the `input` field that triggered the event. To pull this off, the method uses the `event` argument to determine which input was changed and update our `state.fields` object accordingly. For example, if we have an `input` field and we were to give it a `name` prop of "email", when that field triggers an event, we would be able to know that it was email field, because `event.target.name` would be "email".

To see what this looks like in practice, here's the updated `render()`:

code/forms/06-state-input-multi.jsx

```
31  render() {
32    return (
33      <div>
34        <h1>Sign Up Sheet</h1>
35
36        <form onSubmit={this.onFormSubmit}>
37          <input
38            placeholder='Name'
39            name='name'
40            value={this.state.fields.name}
41            onChange={this.onInputChange}
42          />
43
44          <input
45            placeholder='Email'
46            name='email'
47            value={this.state.fields.email}
48            onChange={this.onInputChange}
49          />
50
51          <input type='submit' />
52        </form>
53
54        <div>
55          <h3>People</h3>
56          <ul>
57            { this.state.people.map(({ name, email }, i) =>
58              <li key={i}>{name} ({email})</li>
59            ) }
60          </ul>
61        </div>
62      </div>
63    );
64  },
```

There are a several things to note: first, we've added a second `input` to handle email addresses.

Second, we've changed the `value` prop of the `input` fields so that they don't access attributes on the root of the `state` object. Instead they access the attributes of `state.fields`. Looking at the code above, the `input` for name now has its `value` set to `this.state.fields.name`.

Third, both `input` fields have their `onChange` prop set to the same event handler, `onInputChange()`. We'll see below how we modified `onNameChange()` to be a more general event handler that can accept events from any field, not just "name".

Fourth, our `input` fields now have a `name` prop. This is related to the last point. To allow our general event handler, `onInputChange()`, to be able to tell where the change event came from and where we should store it in our state (e.g. if the change comes from the "email" input the new value should be stored at `state.fields.email`), we provide that `name` prop so that it can be pulled off of the event via its `target` attribute.

Finally, we modify how our people list is rendered. Because it's no longer just a list of names, we modify our `li` element to display both the previous `name` attribute as well as the new `email` data we plan to have.

To make sure that all the data winds up in the right place, we'll need to make sure that our event handlers are properly modified. Here's what the `onInputChange()` event handler (that gets called when any field's input changes) should look like:

`code/forms/06-state-input-multi.jsx`

```
25  onInputChange(evt) {
26    const fields = this.state.fields;
27    fields[evt.target.name] = evt.target.value;
28    this.setState({ fields });
29  },
```

At its core this is similar to what we did before in `onNameChange()` in the last section. The two key differences are that:

1. we are updating a value nested in the `state` (e.g. updating `state.fields.email` instead of `state.email`), and
2. we're using `evt.target.name` to inform which attribute of `state.fields` needs to be updated.

To properly update our state, we first grab a local reference to `state.fields`. Then, we use information from the event (`evt.target.name` and `evt.target.value`) to update the local reference. Lastly, we `setState()` with the modified local reference.

To get concrete, let's go through what would happen if the user enters "someone@somewhere.com" into the "email" field.

First, `onInputChange()` would be called with the `evt` object as an argument. `evt.target.name` would be "email" (because "email" is set as its `name` prop in `render()`) and `evt.target.value` would be "someone@somewhere.com" (because that's what they entered into the field).

Next, `onInputChange()` would grab a local reference to `state.fields`. If this is the first time there was input, `state.fields` and our local reference will be an empty object, `{}`. Then, that local reference would be modified so that it becomes `{email: "someone@somewhere.com"}`.

And finally, `setState()` is called with that change.

At this point, `this.state.fields` will always be in sync with any text in the `input` fields. However, `onFormSubmit()` will need to be changed to get that information into the list of people who have signed up. Here's what the updated `onFormSubmit()` looks like:

`code/forms/06-state-input-multi.jsx`

```
16  onFormSubmit(evt) {
17    const people = [
18      ...this.state.people,
19      this.state.fields,
20    ];
21    this.setState({ people, fields: {} });
22    evt.preventDefault();
23  },
```

In `onFormSubmit()` we first obtain a local reference to the list of people who have signed up, `this.state.people`. Then, we add our `this.state.fields` object (an object representing the name and email currently entered into the fields) onto the `people` list. Finally, we use `this.setState()` to simultaneously update our list with the new information and clear all the fields by returning `state.fields` to an empty object, `{}`.

The great thing about this is that we can easily add as many input fields as we want with very minimal changes. In fact, only the `render()` method would need to change. For each new field, all we would have to do is add another `input` field and change how the list is rendered to display the new field.

For example, if we wanted to add a field for phone number, we would add a new `input` with appropriate `name` and `value` props: `name` would be `phone` and `value` would be `this.state.fields.phone`. `onChange`, like the others, would be our existing `onInputChange()` handler.

After doing that our state will automatically keep track of the phone field and will add it to the `state.people` array and we could change how the view (e.g. the `li`) displays the information.

At this point we have a functional app that's well situated to be extended and modified as requirements evolve. However, it is missing one crucial aspect that almost all forms need: validation.

On Validation

Validation is so central to building forms that it's rare to have a form without it. Validation can be both on the level of the **individual field** and on the **form as a whole**.

When you validate on an individual field, you're making sure that the user has entered data that conforms to your application's expectations and constraints as it relates to that piece of data.

For example, if we want a user to enter an email address, we expect their input to look like a valid email address. If the input does not look like an email address, they might have made a mistake and we're likely to run into trouble down the line (e.g. they won't be able to activate their account). Other common examples are making sure that a zip code has exactly five (or nine) numerical characters and enforcing a password length of at least some minimum length.

Validation on the form as a whole is slightly different. Here is where you'll make sure that all required fields have been entered. This is also a good place to check for internal consistency. For example you might have an order form where specific options are required for specific products.

Additionally, there are tradeoffs for “how” and “when” we validate. On some fields we might want to give validation feedback in real-time. For example, we might want to show password strength (by looking at length and characters used) while the user is typing. However, if we want to validate the availability of a username, we might want to wait until the user has finished typing before we make a request to the server/database to find out.

We also have options for how we display validation errors. We might style the field differently (e.g. a red outline), show text near the field (e.g. “Please enter a valid email.”), and/or disable the form’s submit button to prevent the user from progressing with invalid information.

As for our app, we can begin with validation of the form as a whole and

1. make sure that we have both a name and email and
2. make sure that the email is a valid address.

Adding Validation to Our App

To add validation to our sign-up app we've made some changes. At a high level these changes are

1. add a place in state to store validation errors if they exist,
2. change our `render()` method will show validation error messages (if they exist) with red text next to each field,
3. add a new `validate()` method that takes our `fields` object as an argument and returns a `fieldErrors` object, and
4. `onFormSubmit()` will call the new `validate()` method to get the `fieldErrors` object, and if there are errors it will add them to the state (so that they can be shown in `render()`) and return early without adding the “person” to the list, `state.people`.

First, we've changed our `getInitialState()` method:

code/forms/07-basic-validation.jsx

```
10  getInitialState() {
11    return {
12      fields: {},
13      fieldErrors: {},
14      people: [],
15    };
16  },
```

The only change here is that we've created a default value for the `fieldErrors` property. This is where we'll store errors for each of the field if they exist.

Next, here's what the updated `render()` method looks like:

code/forms/07-basic-validation.jsx

```
45  render() {
46    return (
47      <div>
48        <h1>Sign Up Sheet</h1>
49
50        <form onSubmit={this.onFormSubmit}>
51
52          <input
53            placeholder='Name'
54            name='name'
55            value={this.state.fields.name}
56            onChange={this.onInputChange}
57          />
58
59          <span style={{ color: 'red' }}>{ this.state.fieldErrors.name }</span>
60
61          <br />
62
63          <input
64            placeholder='Email'
65            name='email'
66            value={this.state.fields.email}
67            onChange={this.onInputChange}
68          />
69
70          <span style={{ color: 'red' }}>{ this.state.fieldErrors.email }</span>
```

```
71
72      <br />
73
74      <input type='submit' />
75  </form>
76
77  <div>
78      <h3>People</h3>
79      <ul>
80          { this.state.people.map(({ name, email }, i) =>
81              <li key={i}>{name} ({email})</li>
82          )
83      </ul>
84  </div>
85 </div>
86 );
87 },
```

The only differences here are two new `span` elements, one for each field. Each `span` will look in the appropriate place in `state.fieldErrors` for an error message. If one is found it will be displayed in red next to the field. Next up, we'll see how those error messages can get into the state.

It is after the user submits the form that we will check the validity of their input. So the appropriate place to begin validation is in the `onFormSubmit()` method. However, we'll want to create a standalone function for that method to call. We've created the pure function, `validate()` method for this:

`code/forms/07-basic-validation.jsx`

```
21  const fieldErrors = this.validate(person);
```

Our `validate()` method is pretty simple and has two goals. First, we want to make sure that both `name` and `email` are present. By checking their truthiness we can know that they are defined and not empty strings. Second, we want to know that the provided email address looks valid. This is actually a bit of a thorny issue, so we rely on `validator`⁵² to let us know. If any of these conditions are not met, we add a corresponding key to our `errors` object and set an error message as the value.

Afterwards, we've updated our `onFormSubmit()` to use this new `validate()` method and act on the returned error object:

⁵²<http://npm.im/validator>

code/forms/07-basic-validation.jsx

```
18  onFormSubmit(evt) {
19    const people = [ ...this.state.people ];
20    const person = this.state.fields;
21    const fieldErrors = this.validate(person);
22    this.setState({ fieldErrors });
23    evt.preventDefault();
24
25    if (Object.keys(fieldErrors).length) return;
26
27    people.push(person);
28    this.setState({ people, fields: {} });
29 },
```

To use the `validate()` method, we get the current values of our fields from `this.state.fields` and provide it as the argument. `validate()` will either return an empty object if there are no issues, or if there are issues, it will return an object with keys corresponding to each field name and values corresponding to each error message. In either case, we want to update our `state.fieldErrors` object so that `render()` can display or hide the messages as necessary.

If the validation errors object has any keys (`Object.keys(fieldErrors).length > 0`) we know there are issues. If there are no validation issues, the logic is the same as in previous sections – we add the new information and clear the fields. However, if there are any errors, we return early. This prevents the new information from being added to the list.

Sign Up Sheet

Name	<input type="text" value="Nate"/>
Email	<input type="text"/> Email Required
<input type="button" value="Submit"/>	

People

Email Required

Sign Up Sheet

Nate
@#d3

Invalid Email

Submit

People

Email Invalid

At this point we've covered the fundamentals of creating a form with validation in React. In the next section we'll take things a bit further and show how we can validate in real-time at the field level and we'll create a `Field` component to improve the maintainability when an app has multiple fields with different validation requirements.

Creating the Field Component

In the last section we added validation to our form. However, our form component is responsible for running the validations on the form as a whole as well as **the individual validation rules for each field**.

It would be ideal if *each field* was responsible for identifying validation errors on *its own input*, and the parent form was only responsible for identifying issues at the form-level. This comes with several advantages:

1. an email field created in this way could check the format of its input while the user types in real-time.
2. the field could incorporate its validation error message, freeing the parent form from having to keep track of it.

To do this we're first going to create a new separate `Field` component, and we will use it instead of `input` elements in the form. This will let us combine a normal `input` with both validation logic and error messaging.

Before we get into the creation of this new component, it will be useful to think of it at a high level in terms of inputs and outputs. In other words, "what information do we need to provide this component?", and "what kinds of things would we expect in return?"

These inputs are going to become this component's props and the output will be used by any event handlers we pass into it.

Because our `Field` component will contain a child `input`, we'll need to provide the same baseline information so that it can be passed on. For example, if we want a `Field` component rendered with a specific `placeholder` prop on its child `input`, we'll have to provide it as a prop when we create the `Field` component in our form's `render()` method.

Two other props we'll want to provide are `name`, and `value`. `name` will allow us to share an event handler between components like we've done before, and `value` allows the parent form to pre-populate `Field` as well as keep it updated.

Additionally, this new `Field` component is responsible for its own validation. Therefore we'll need to provide it rules specific to data it contains. For example, if this is the "email" `Field`, we'll provide it a function as its `validate` prop. Internally it will run this function to determine if its input is a valid email address.

Lastly, we'll provide an event handler for `onChange` events. The function we provide as the `onChange` prop will be called every time the input in the `Field` changes, and it will be called with an event argument that we get to define. This event argument should have three properties that we're interested in: (1) the name of the `Field`, (2) the current `value` of the `input`, and (3) the current validation error (if present).

To quickly review, for the new `Field` component to do its job it will need the following:

- `placeholder`: This will be passed straight through to the `input` child element. Similar to a `label`, this tells the user what data to the `Field` expects.
- `name`: We want this for the same reason we provide `name` to `input` elements: we'll use this in the event handler decide where to store input data and validation errors.
- `value`: This is how our parent form can initialize the `Field` with a value, or it can use this to update the `Field` with a new value. This is similar to how the `value` prop is used on an `input`.
- `validate`: A function that returns validation errors (if any) when run.
- `onChange`: An event handler to be run when the `Field` changes. This function will accept an event object as an argument.

Following this, we're able to set up `propTypes` on our new `Field` component:

`code/forms/08-field-component-field.jsx`

```
4  propTypes: {
5    placeholder: PropTypes.string,
6    name: PropTypes.string.isRequired,
7    value: PropTypes.string,
8    validate: PropTypes.func,
9    onChange: PropTypes.func.isRequired,
10   },
```

Next, we can think about the state that `Field` will need to keep track of. There are only two pieces of data that `Field` will need, the current value and error. Like in previous sections where our form component needed that data for its `render()` method, so too does our `Field` component. Here's how we'll set up our `getInitialState()`:

`code/forms/08-field-component-field.jsx`

```
12  getInitialState() {
13    return {
14      value: this.props.value,
15      error: false,
16    };
17  },
```

One key difference is that our `Field` has a parent, and sometimes this parent will want to update the `value` prop of our `Field`. To allow this, we'll need to create a new lifecycle method, `componentWillReceiveProps()` to accept the new `value` and update the state. Here's what that looks like:

`code/forms/08-field-component-field.jsx`

```
19  componentWillReceiveProps(update) {
20    this.setState({ value: update.value });
21  },
```

The `render()` method of `Field` should be pretty simple. It's just the `input` and the corresponding `span` that will hold the error message:

code/forms/08-field-component-field.jsx

```
33 render() {
34   return (
35     <div>
36       <input
37         placeholder={this.props.placeholder}
38         value={this.state.value}
39         onChange={this.onChange}
40       />
41       <span style={{ color: 'red' }}>{ this.state.error }</span>
42     </div>
43   );
44 },
```

For the `input` element, the `placeholder` will be passed in from the parent and is available from `this.props.placeholder`. As mentioned above, the `value` of the `input` and the error message in the `span` will both be stored in the `state`. `value` comes from `this.state.value` and the error message is at `this.state.error`. And lastly, we'll set an `onChange` event handler that will be responsible for accepting user input, validating, updating state, *and calling the parent's event handler as well*. The method that will take care of that is `this.onChange`:

```
1  onChange (evt) {
2    const name = this.props.name;
3    const value = evt.target.value;
4    const error = this.props.validate ? this.props.validate(value) : false;
5
6    this.setState({value, error});
7
8    this.props.onChange({name, value, error});
9 }
```

`this.onChange` is a pretty efficient function. It handles four different responsibilities in as many lines. As in previous sections, the event object gives us the current text in the `input` via its `target.value` property. Once we have that, we see if it passes validation. If `Field` was given a function for its `validate` prop, we use it here. If one was not given, we don't have to validate the `input` and `error` sets to `false`. Once we have both the `value` and `error` we can update our `state` so that they both appear in `render()`. However, it's not just the `Field` component that needs to be updated with this information.

When `Field` is used by a parent component, it passes in its own event handler in as the `onChange` prop. We call this function so that we can pass information up the parent. Here in `this.onChange()`,

it is available as `this.props.onChange()`, and we call it with three pieces of information: the name, value, and error of the Field.

This might be a little confusing since “onChange” appears in multiple places. You can think of it as carrying information in a chain of event handlers. The form contains the Field which contains an input. Events occur on the input and the information passes first to the Field and finally to the form.

At this point our Field component is ready to go, and can be used in place of the input and error message combos in our app.

Using our new Field Component

Now that we’re ready to use our brand new Field component, we can make some changes to our app. The most obvious change is that Field will take the place of both the input and error message span elements in our `render()` method. This is great because Field can take care of validation at the field level. But what about at the form level?

If you remember, we can employ two different levels of validation, one at the field level, and one at the form level. Our new Field component will let us validate the format of each field in real-time. What they won’t do, however, is validate the entire form to make sure we have all the data we need. For that, we also want form-level validation.

Another nice feature we’ll add here is disabling/enabling the form submit button in real-time as the form passes/fails validation. This is a nice bit of feedback that can improve a form’s UX and make it feel more responsive.

Here’s how our update `render()` looks:

code/forms/08-field-component-form.jsx

```
54  render() {
55    return (
56      <div>
57        <h1>Sign Up Sheet</h1>
58
59        <form onSubmit={this.onFormSubmit}>
60
61          <Field
62            placeholder='Name'
63            name='name'
64            value={this.state.fields.name}
65            onChange={this.onInputChange}
66            validate={(val) => (val ? false : 'Name Required')}
67          />
68
```

```
69      <br />
70
71      <Field
72          placeholder='Email'
73          name='email'
74          value={this.state.fields.email}
75          onChange={this.onInputChange}
76          validate={(val) => (isEmail(val) ? false : 'Invalid Email')}
77      />
78
79      <br />
80
81      <input type='submit' disabled={this.validate()} />
82  </form>
83
84      <div>
85          <h3>People</h3>
86          <ul>
87              { this.state.people.map(({ name, email }, i) =>
88                  <li key={i}>{name} ({email})</li>
89              )
90          </ul>
91      </div>
92      </div>
93  );
94 },
```

You can see that `Field` is a drop-in replacement for `input`. All the props are the same as they were on `input`, except we have one additional prop this time: `validate`.

Above in the `Field` component's `onChange()` method, we make a call to the `this.props.validate()` function. What we provide as the `validate` prop to `Field`, will be that function. Its goal is to take user provided input as its argument and give a return value that corresponds to the validity of that input. If the input is not valid, `validate` should return an error message. Otherwise, it should return `false`.

For the “name” `Field` the `validate` prop is pretty simple. We’re just checking for a truthy value. As long as there are characters in the box, validation will pass, otherwise we return the ‘Name Required’ error message.

For the “email” `Field`, we’re going to use the `isEmail()` function that we imported from the `validator` module. If that function returns `true`, we know it’s a valid-looking email and validation passes. If not, we return the ‘`Invalid Email`’ message.

Notice that we left their `onChange` prop alone, it is still set to `this.onInputChange`. However, since `Field` uses the function differently than `input`, we must update `onInputChange()`.

Before we move on, notice the only other change that we've made to `render()`: we conditionally disable the submit button. To do this, we set the value of the `disabled` prop to the return value of `this.validate()`. Because `this.validate()` will have a truthy return value if there are validation errors, the button will be disabled if the form is not valid. We'll show what the `this.validate()` function looks like in a bit.

Sign Up Sheet

Nate

hello Invalid Email

Submit

People

Disabled Submit Button

As mentioned, both `Field` components have their `onChange` props set to `this.onInputChange`. We've had to make some changes to match the difference between `input` and our `Field`. Here's the updated version:

`code/forms/08-field-component-form.jsx`

```
32  onInputChange({ name, value, error }) {
33    const fields = this.state.fields;
34    const fieldErrors = this.state.fieldErrors;
35
36    fields[name] = value;
37    fieldErrors[name] = error;
38
39    this.setState({ fields, fieldErrors });
40  },
```

Previously, the job of `onInputChange()` was to update `this.state.fields` with the current user input values. In other words, when an a text field was edited, `onInputChange()` would be called with an event object. That event object had a `target` property that referenced the `input` element. Using that reference, we could get the `name` and `value` of the `input`, and with those, we would update `state.fields`.

This time around `onInputChange()` has the same responsibility, but it is our `Field` component that calls this function, not `input`. In the previous section, we show the `onChange()` method of `Field`, and that's where `this.props.onChange()` is called. When it is called, it's called like this: '`this.props.onChange({name, value, error})`'.

This means that instead of using `evt.target.name` or `evt.target.value` as we did before, we get `name` and `value` directly from the argument object. In addition, we also get the validation error for each field. This is necessary – for our form component to prevent submission, it will need to know about field-level validation errors.

Once we have the `name`, `value`, and `error`, we can update two objects in our `state`, the `state.fields` object we used before, and a new object, `state.fieldErrors`. Soon, we will show how `state.fieldErrors` will be used to either prevent or allow the form submit depending on the presence or absence of field-level validation errors.

With both `render()` and `onInputChange()` updated, we again have a nice feedback loop set up for our `Field` components:

- First, the user types into the `Field`.
- Then, the event handler of the `Field` is called, `onInputChange()`.
- Next, `onInputChange()` updates the `state`.
- After, the form is rendered again, and the `Field` passed an updated `value` prop.
- Then, `componentWillReceiveProps()` is called in `Field` with the new `value`, and its state is updated.
- Finally, `Field.render()` is called again, and the text field shows the appropriate input and (if applicable) validation error.

At this point, our form's state and appearance are in sync. Next, we need to change how we handle the submit event. Here's the updated event handler for the form, `onFormSubmit()`:

code/forms/08-field-component-form.jsx

```
20  onFormSubmit(evt) {
21    const people = this.state.people;
22    const person = this.state.fields;
23
24    evt.preventDefault();
25
26    if (this.validate()) return;
27
28    people.push(person);
29    this.setState({ people, fields: {} });
30  },
```

The objective of `onFormSubmit()` hasn't changed. It is still responsible for either adding a person to the list, or preventing that behavior if there are validation issues. To check for validation errors, we call `this.validate()`, and if there are any, we return early before adding the new person to the list.

Here's what the current version of `validate()` looks like:

```
1  validate () {
2    const person = this.state.fields;
3    const fieldErrors = this.state.fieldErrors;
4    const errMessages = Object.keys(fieldErrors).filter((k) => fieldErrors[k])
5
6    if (!person.name) return true;
7    if (!person.email) return true;
8    if (errMessages.length) return true;
9
10   return false
11 }
```

Put simply, `validate()` is checking to make sure the data is valid at the form level. For the form to pass validation at this level it must satisfy two requirements: (1) neither field may be empty and (2) there must not be any field-level validation errors.

To satisfy the first requirement, we access `this.state.fields` and ensure that both `state.fields.name` and `state.fields.email` are truthy. These are kept up to date by `onInputChange()`, so it will always match what is in the text fields. If either `name` or `email` are missing, we return `true`, signaling that there is a validation error.

For the second requirement, we look at `this.state.fieldErrors`. `onInputChange()` will set any field-level validation error messages on this object. We use `Object.keys` and `Array.filter` to get

an array of all present error messages. If there are any field-level validation issues, there will be corresponding error messages in the array, and its length will be non-zero and truthy. If that's the case, we also return `true` to signal the existence of a validation error.

`validate()` is a simple method that can be called at any time to check if the data is valid at the form-level. We use it both in `onFormSubmit()` to prevent adding invalid data to the list and in `render()` to disable the submit button, providing nice feedback in the UI.

And that's it. We're now using our custom `Field` component to do field-level validation on the fly, and we also use form-level validation to toggle the submit button in real-time.

Remote Data

Our form app is coming along. A user can sign up with their name and email, and we validate their information before accepting the input. But now we're going to kick it up a notch. We're going to explore how to allow a user to select from hierarchical, asynchronous options.

The most common example is to allow the user to select a car by year, make, and model. First the user selects a year, then the manufacturer, then the model. After choosing an option in one `select`, the next one becomes available. There are two interesting facets to building a component like this.

First, not all combinations make sense. There's no reason to allow your user to choose a 1965 Tesla Model T. Each option list (beyond the first) depends on a previously selected value.

Second, we don't want to send the entire database of valid choices to the browser. Instead, the browser only knows the top level of choices (e.g. years in a specific range). When the user makes a selection, we provide the selected value to the server and ask for next level (e.g. manufacturers available for a given year). Because the next level of options come from the server, this is an asynchronous.

Our app won't be interested in the user's car, but we will want to know what they're signing up for. Let's make this an app for users to learn more JavaScript by choosing a [NodeSchool](#)⁵³ workshop to attend.

A NodeSchool workshop can be either "core" or "elective". We can think of these as "departments" of NodeSchool. Therefore, depending on which department the user is interested in, we can allow them to choose a corresponding workshop. This is similar to the above example where a user chooses a car's year before its manufacturer.

If a user chooses the core department, we would enable them to choose from a list of core workshops like `learnyounode` and `stream-adventure`. If instead, they choose the elective department, we would allow them to pick workshops like `Functional JavaScript` or `Shader School`. Similar to the car example, the course list is provided asynchronously and depends on which department was selected.

⁵³<http://nodeschool.io>

The simplest way to achieve this is with two `select` elements, one for choosing the department and the other for choosing the course. However, we will hide the second `select` until: (1) the user has selected a department, and (2) we've received the appropriate course list from the server.

Instead of building this functionality directly into our form. We'll create a custom component to handle both the hierarchical and asynchronous nature of these fields. By using a custom component, our form will barely have to change. Any logic specific to the workshop selection will be hidden in the component.

Building the Custom Component

The purpose of this component is to allow the user to select a NodeSchool course. From now on we'll refer to it as our `CourseSelect` component.

However, before starting on our new `CourseSelect` component, we should think about how we want it to communicate with its form parent. This will determine the component's props.

The most obvious prop is `onChange()`. The purpose of this component is to help the user make a department/course selection and to make that data available to the form. Additionally, we'll want to be sure that `onChange()` to be called with the same arguments we're expecting from the other field components. That way we con't have to create any special handling for this component.

We also want the form to be able to set this component's state if need be. This is particularly useful when we want to clear the selections after the user has submitted their info. For this we'll accept two props. One for department and one for course.

And that's all we need. This component will accept three props. Here's how they'll look in our new `CourseSelect` component:

`code/forms/09-course-select.jsx`

```
12  propTypes: {
13    department: PropTypes.string,
14    course: PropTypes.string,
15    onChange: PropTypes.func.isRequired,
16  },
```

Next, we can think about the state that `CourseSelect` will need to keep track of. The two most obvious pieces of state are department and course. Those will change when the user makes selections, and when the form parent clears them on a submit.

`CourseSelect` will also need to keep track of available courses for a given department. When a user selects a department, we'll asynchronously fetch the corresponding course list. Once we have that list we'll want to store it in our state as `courses`.

Lastly, when dealing with asynchronous fetching, it's nice to inform the user that data is loading behind the scenes. We will also keep track of whether or not data is "loading" in our state as `_loading`.



The underscore prefix of `_loading` is just a convention to highlight that it is purely presentational. Presentational state is only used for UI effects. In this case it will be used to hide/show the loading indicator image.

Here's what `getInitialState()` looks like:

`code/forms/09-course-select.jsx`

```
18  getInitialState() {
19      return {
20          department: null,
21          course: null,
22          courses: [],
23          _loading: false,
24      };
25 },
```

As mentioned above, this component's form parent will want to update the `department` and `course` props. Our `componentWillReceiveProps()` method will use the update to appropriately modify the state:

`code/forms/09-course-select.jsx`

```
27  componentWillReceiveProps(update) {
28      this.setState({
29          department: update.department,
30          course: update.course,
31      });
32 },
```

Now that we have a good idea of what our data looks like, we can get into how the component is rendered. This component is a little more complicated than our previous examples, so we take advantage of composition to keep things tidy. You will notice that our `render()` method is mainly composed of two functions, `renderDepartmentSelect()` and `renderCourseSelect()`:

code/forms/09-course-select.jsx

```
104     render() {
105       return (
106         <div>
107           { this.renderDepartmentSelect() }
108           <br />
109           { this.renderCourseSelect() }
110         </div>
111       );
112     },

```

Aside from those two functions, `render()` doesn't have much. But this nicely illustrates the two "halves" of our component: the "department" half and the "course" half. Let's first take a look at the "department" half. Starting with `renderDepartmentSelect()`:

code/forms/09-course-select.jsx

```
107   { this.renderDepartmentSelect() }
```

This method returns a `select` element that displays one of three options. The currently displayed option depends on the `value` prop of the `select`. The option whose `value` matches the `select` will be shown. The options are:

- "Which department?" (value: *empty string*)
- "NodeSchool: Core" (value: "core")
- "NodeSchool: Electives" (value: "electives")

The value of `select` is `this.state.department || ''`. In other words, if `this.state.department` is falsy (it is by default), the `value` will be an *empty string* and will match "Which department?". Otherwise, if `this.state.department` is either "core" or "electives", it will display one of the other options.

Because `this.onSelectDepartment` is set as the `onChange` prop of the `select`, when the user changes the option, `onSelectDepartment()` is called with the `change` event. Here's what that looks like:

[code/forms/09-course-select.jsx](#)

```
34  onSelectDepartment(evt) {
35      const department = evt.target.value;
36      const course = null;
37      this.setState({ department, course });
38      this.props.onChange({ name: 'department', value: department });
39      this.props.onChange({ name: 'course', value: course });
40
41      if (department) this.fetch(department);
42 },
```

When the department is changed, we want three things to happen. First, we want to update state to match the selected department option. Second, we want to propagate the change via the onChange handler provided in the props of CourseSelect. Third, we want to fetch the available courses for the department.

When we update the state, we update it to the value of the event's target, the select. The value of the select is the value of the chosen option, either '', "core", or "electives". After the state is set with a new value, render() and renderDepartmentSelect() are run and a new option is displayed.

Notice that we also reset the course. Each course is only valid for its department. If the department changes, it will no longer be a valid choice. Therefore, we set it back to its initial value, null.

After updating state, we propagate the change to the component's change handler, this.props.onChange. Because we use the arguments as we have previously, this component can be used just like Field and can be given the same handler function. The only trick is that we need to call it twice, once for each input.

Finally, if a department was selected, we fetch the course list for it. Here's the method it calls, fetch():

[code/forms/09-course-select.jsx](#)

```
50  fetch(department) {
51      this.setState({ _loading: true, courses: [] });
52      apiClient(department).then((courses) => {
53          this.setState({ _loading: false, courses: courses });
54      });
55 },
```

The responsibility of this method is to take a department string, use it to asynchronously get the corresponding course list, courses, and update the state with it. However, we also want to be sure to affect the state for a better user experience.

We do this by updating the state *before* the `apiClient` call. We know that we'll be waiting for the response with the new course list, and in that time we should show the user a loading indicator. To do that, we need our state to reflect our fetch status. Right before the `apiClient` call, we set the state of `_loading` to `true`. Once the operation completes, we set `_loading` back to `false` and update our course list.

Previously, we mentioned that this component had two “halves” illustrated by our `render()` method:

`code/forms/09-course-select.jsx`

```
104     render() {
105       return (
106         <div>
107           { this.renderDepartmentSelect() }
108           <br />
109           { this.renderCourseSelect() }
110         </div>
111       );
112     },

```

We've already covered the “department” half. Let's now take a look at the “course” half starting with `renderCourseSelect()`:

`code/forms/09-course-select.jsx`

```
109   { this.renderCourseSelect() }
```

The first thing that you'll notice is that `renderCourseSelect()` returns a different root element depending on particular conditions.

If `state._loading` is `true`, `renderCourseSelect()` only returns a single `img`: a loading indicator. Alternatively, if we're not loading, but a department has not been selected (and therefore `state.department` is falsy), an empty `span` is returned – effectively hiding this half of the component.

However, if we're not loading, and the user *has* selected a department, `renderCourseSelect()` returns a `select` similar to `renderDepartmentSelect()`.

The biggest difference between `renderCourseSelect()` and `renderDepartmentSelect()` is that `renderCourseSelect()` dynamically populates the option children of the `select`.

The first option in this `select` is “Which course?” which has an empty string as its value. If the user has not yet selected a course, this is what they should see (just like “Which department?” in the other `select`). The options that follow the first come from the course list stored in `state.courses`.

To provide all the child `option` elements to the `select` at once, the `select` is given a single array as its child. The first item in the array is an element for our “Which course?” option. Then, we use

the spread operator combined with `map()` so that from the second item on, the array contains the course options from `state`.

Each item in the array is an `option` element. Like before, each element has text that it displays (like “Which course?”) as well as a `value` prop. If the `value` of the `select` matches the `value` of the `option`, that `option` will be displayed. By default, the `value` of the `select` will be an empty string, so it will match the “Which course?” option. Once the user chooses a course and we are able to update `state.course`, the corresponding course will be shown.



This is a dynamic collection, we must also provide a `key` prop to each `option` to avoid warnings from React.

Lastly, we provide a change handler function, `onSelectCourse()` to the `select` prop `onChange`. When the user chooses a course, that function will be called with a related event object. We will then use information from that event to update the state and notify the parent.

Here’s `onSelectCourse()`:

`code/forms/09-course-select.jsx`

```
44  onSelectCourse(evt) {
45    const course = evt.target.value;
46    this.setState({ course });
47    this.props.onChange({ name: 'course', value: course });
48  },
```

Like we’ve done before, we get the `value` of the target element from the event. This `value` is the `value` of whichever option the user picked in the courses `select`. Once we update `state.course` with this `value`, the `select` will display the appropriate option.

After the `state` is updated, we call the change handler provided by the component’s parent. Same as with the department selection, we provide `this.props.onChange()` an object argument with the `name/value` structure the handler expects.

And that’s it for our `CourseSelect` component! As we’ll see in next part, integration with the form requires very minimal changes.

Adding CourseSelect

Now that our new `CourseSelect` component is ready, we can add it to our form. Only three small changes are necessary:

1. We add the `CourseSelect` component to `render()`.

2. We update our “People” list in `render()` to show the new fields (department and course).
3. Since department and course are required fields, we modify our `validate()` method to ensure their presence.

Because we were careful to call the change handler from within `CourseSelect(this.props.onChange)` with a `{name, value}` object the way that `onInputChange()` expects, we’re able to reuse that handler. When `onInputChange()` is called by `CourseSelect`, it can appropriately update state with the new department and course information – just like it does with calls from the `Field` components.

Here’s the updated `render()`:

code/forms/09-async-fetch.jsx

```
57  render() {
58    return (
59      <div>
60        <h1>Sign Up Sheet</h1>
61
62        <form onSubmit={this.onFormSubmit}>
63
64          <Field
65            placeholder='Name'
66            name='name'
67            value={this.state.fields.name}
68            onChange={this.onInputChange}
69            validate={(val) => (val ? false : 'Name Required')}
70          />
71
72          <br />
73
74          <Field
75            placeholder='Email'
76            name='email'
77            value={this.state.fields.email}
78            onChange={this.onInputChange}
79            validate={(val) => (isEmail(val) ? false : 'Invalid Email')}
80          />
81
82          <br />
83
84          <CourseSelect
85            department={this.state.fields.department}
86            course={this.state.fields.course}
87            onChange={this.onInputChange}>
```

```

88      />
89
90      <br />
91
92      <input type='submit' disabled={this.validate()} />
93      </form>
94
95      <div>
96          <h3>People</h3>
97          <ul>
98              { this.state.people.map(({ name, email, department, course }, i) =>
99                  <li key={i}>{[ name, email, department, course ].join(' - ')</li>
100                 )
101             </ul>
102         </div>
103     </div>
104 );
105 },

```

When adding CourseSelect we provide three props:

1. The current department from state (if one is present)
2. The current course from state (if one is present)
3. The onChange() handler (same function used by Field)

Here it is by itself:

```

1  <CourseSelect
2      department={this.state.fields.department}
3      course={this.state.fields.course}
4      onChange={this.onChange} />

```

The other change we make in render() is we add the new department and course fields to the “People” list. Once a user submits sign-up information, they appear on this list. To show the department and course information, we need to get that data from state and display it:

```
1  <h3>People</h3>
2  <ul>
3    { this.state.people.map( ({name, email, department, course}, i) =>
4      <li key={i}>{[name, email, department, course].join(' - ')</li>
5    )
6  </ul>
```

This is as simple as pulling more properties from each item in the `state.people` array.

The only thing left to do is add these fields to our form-level validation. Our `CourseSelect` controls the UI to ensure that we won't get invalid data, so we don't need to worry about field-level errors. However, department and course are required fields, we should make sure that they are present before allowing the user to submit. We do this by updating our `validate()` method to include them:

`code/forms/09-async-fetch.jsx`

```
43  validate() {
44    const person = this.state.fields;
45    const fieldErrors = this.state.fieldErrors;
46    const errMessages = Object.keys(fieldErrors).filter((k) => fieldErrors[k]);
47
48    if (!person.name) return true;
49    if (!person.email) return true;
50    if (!person.course) return true;
51    if (!person.department) return true;
52    if (errMessages.length) return true;
53
54    return false;
55  },
```

Once `validate()` is updated, our app will keep the submit button disabled until we have both department and course selected (in addition to our other validation requirements).

Thanks to the power of React and composition our form was able to take on complicated functionality while keeping high maintainability.

Separation of View and State

Once we've received information from the user and we've decided that it's valid, we then need to convert the information to JavaScript objects. Depending on the form, this could involve casting input values from strings to numbers, dates, or booleans, or it could be more involved if you need to impose a hierarchy by corralling the values into arrays or nested objects.

After we have the information as JavaScript objects, we then have to decide how to use them. The objects might be sent to a server as JSON to be stored in a database, encoded in a url to be used as a search query, or maybe only used to configure how the UI looks.

The information in those objects will almost always affect the UI and in many cases will also affect your application's behavior. It's up to us to determine how to store that info in our app.

Async Persistence

At this point our app is pretty useful. You could imagine having the app open on a kiosk where people can come up to it and sign up for things. However, there's one big shortcoming: if the browser is closed or reloaded, all data is lost.

In most web apps, when a user inputs data, that data should be sent to a server for safe keeping in a database. When the user returns to the app, the data can be fetched, and the app can pick back up right where it left off.

In this example, we'll cover three aspects of persistence: saving, loading, and handling errors. While we won't be sending the data to a remote server or storing it in a database (we'll be using `localStorage` instead), we'll treat it as an asynchronous operation to illustrate how almost any persistence strategy could be used.

To persist the sign up list (`state.people`), we'll only need to make a few changes to our parent form component. At a high level they are:

1. Modify `getInitialState()` to keep track of persistence status. Basically, we'll want to know if the app is currently loading, is currently saving, or encountered an error during either operation.
2. Make a request using our API client to get any previously persisted data and load it into our `state`.
3. Update our `onFormSubmit()` event handler to trigger a save.
4. Change our `render()` method so that the "submit" button both reflects the current save status and prevents the user from performing an unwanted action like a double-save.

First, we'll want to modify `getInitialState()` to keep track of our "loading" and "saving" status. This is useful to both accurately communicate the status of persistence and to prevent unwanted user actions. For example, if we know that the app is in the process of "saving", we can disable the submit button. Here's the updated `getInitialState()` method with the two new properties:

code/forms/10-remote-persist.jsx

```
16  getInitialState: function () {
17    return {
18      fields: {},
19      fieldErrors: {},
20      people: [],
21      _loading: false,
22      _saveStatus: 'READY',
23    };
24  },
```

The two new properties are `_loading` and `_saveStatus`. As before, we use the underscore prefix convention to signal that they are private to this component. There's no reason for a parent or child component to ever know their values.

`_saveStatus` is initialized with the value "READY", but we will have four possible values: "READY", "SAVING", "SUCCESS", and "ERROR". If the `_saveStatus` is either "SAVING" or "SUCCESS", we'll want to prevent the user from making an additional save.

Next, when the component has been successfully loaded and is about to be added to the DOM, we'll want to request any previously saved data. To do this we'll add the lifecycle method `componentWillMount()` which is automatically called by React at the appropriate time. Here's what that looks like:

code/forms/10-remote-persist.jsx

```
26  componentWillMount() {
27    this.setState({ _loading: true });
28    apiClient.loadPeople().then((people) => {
29      this.setState({ _loading: false, people: people });
30    });
31  },
```

Before we start the fetch with `apiClient`, we set `state._loading` to `true`. We'll use this in `render()` to show a loading indicator. Once the fetch returns, we update our `state.people` list with the previously persisted list and set `_saveStatus` back to `false`.



`apiClient` is a simple object we created to simulate asynchronous loading and saving. If you look at the code for this chapter, you'll see that the "save" and "load" methods are thin async wrappers around `localStorage`. In your own apps you could create our own `apiClient` with similar methods to perform network requests.

Unfortunately, our app doesn't yet have a way to persist data. At this point there won't be any data to load. However, we can fix that by updating `onFormSubmit()`.

As in the previous sections, we'll want our user to be able to fill out each field and hit "submit" to add a person to the list. When they do that, `onFormSubmit()` is called. We'll make a change so that we not only perform the previous behavior (validation, updating `state.people`), but we *also* persist that list using `apiClient.savePeople()`:

`code/forms/10-remote-persist.jsx`

```
33  onFormSubmit(evt) {
34      const person = this.state.fields;
35
36      evt.preventDefault();
37
38      if (this.validate()) return;
39
40      const people = [ ...this.state.people, person ];
41
42      this.setState({ _saveStatus: 'SAVING' });
43      apiClient.savePeople(people)
44          .then(() => {
45              this.setState({
46                  people: people,
47                  fields: {},
48                  _saveStatus: 'SUCCESS',
49              });
50          })
51          .catch((err) => {
52              console.error(err);
53              this.setState({ _saveStatus: 'ERROR' });
54          });
55      },

```

In the previous sections, if the data passed validation, we would just update our `state.people` list to include it. This time we'll also add the `person` to the `people` list, but we only want to update our `state` if `apiClient` can successfully persist. The order of operations looks like this:

1. Create a new array, `people` with both the contents of `state.people` and the new `person` object.
2. Update `state._saveStatus` to "SAVING"
3. Use `apiClient` to begin persisting the new `people` array from #1.

4. If `apiClient` is successful, update `state` with our new people array, an empty `fields` object, and `_saveStatus: "SUCCESS"`. If `apiClient` is *not* successful, leave everything as is, but set `state._saveStatus` to `"ERROR"`.

Put simply, we set the `_saveStatus` to `"SAVING"` while the `apiClient` request is “in-flight”. If the request is successful, we set the `_saveStatus` to `"SUCCESS"` and perform the same actions as before. If not, the only update is to set `_saveStatus` to `"ERROR"`. This way, our local state does not get out of sync with our persisted copy. Also, since we don’t clear the fields, we give the user an opportunity to try again without having to re-input their information.



For this example we are being conservative with our UI updates. We only add the new person to the list *if apiClient is successful*. This is in contrast to an optimistic update, where we would add the person to the list locally *first*, and later make adjustments if there was a failure. To do an optimistic update we could keep track of which `person` objects were added before which `apiClient` calls. Then if an `apiClient` call fails, we could selectively remove the particular `person` object associated with that call. We would also want to display a message to the user explaining the issue.

Our last change is to modify our `render()` method so that the UI accurately reflects our status with respect to loading and saving. As mentioned, we’ll want the user to know if we’re in the middle of a load or a save, or if there was a problem saving. We can also control the UI to prevent them from performing unwanted actions such as a double save.

Here’s the updated `render()` method:

code/forms/10-remote-persist.jsx

```
81  render() {
82    if (this.state._loading) {
83      return <img alt='loading' src='/img/loading.gif' />;
84    }
85
86    return (
87      <div>
88        <h1>Sign Up Sheet</h1>
89
90        <form onSubmit={this.onFormSubmit}>
91
92          <Field
93            placeholder='Name'
94            name='name'
95            value={this.state.fields.name}
96            onChange={this.onInputChange}>
```

```
97      validate={(val) => (val ? false : 'Name Required')}>
98    />
99
100   <br />
101
102   <Field
103     placeholder='Email'
104     name='email'
105     value={this.state.fields.email}
106     onChange={this.onInputChange}
107     validate={(val) => (isEmail(val) ? false : 'Invalid Email')}>
108   />
109
110   <br />
111
112   <CourseSelect
113     department={this.state.fields.department}
114     course={this.state.fields.course}
115     onChange={this.onInputChange}>
116   />
117
118   <br />
119
120   {{
121     SAVING: <input value='Saving...' type='submit' disabled />,
122     SUCCESS: <input value='Saved!' type='submit' disabled />,
123     ERROR: <input
124       value='Save Failed - Retry?'
125       type='submit'
126       disabled={this.validate()}>
127     />,
128     READY: <input
129       value='Submit'
130       type='submit'
131       disabled={this.validate()}>
132     />,
133   }[this.state._saveStatus]}
```

```

139     <ul>
140       { this.state.people.map(({ name, email, department, course }, i) =>
141         <li key={i}>{[ name, email, department, course ].join(' - ')</li>
142       ) }
143     </ul>
144   </div>
145 </div>
146 );
147 ,

```

First, we want is to show the user a loading indicator while we are loading previously persisted data. Like the previous section, this is done on the first line of `render()` with a conditional and an early return. While we are loading (`state._loading` is truthy), we won't render the form, only the loading indicator:

```
1 if (this.state._loading) return <img src='/img/loading.gif' />
```

Next, we want the submit button to communicate the current save status. If no save request is in-flight, we want the button to enabled if the field data is valid. If we are in the process of saving, we want the button to read "Saving..." to be disabled. The user will know that the app is busy, and since the button is disabled, they won't be able to submit duplicate save requests. If the save request resulted in an error, we use the button text to communicate that and indicate that they can try again. The button will be enabled if the input data is still valid. Finally, if the save request completed successfully, we use the button text to communicate that. Here's how we render the button:

```

1 {{{
2   SAVING: <input value='Saving...' type='submit' disabled />,
3   SUCCESS: <input value='Saved!' type='submit' disabled/>,
4   ERROR: <input value='Save Failed - Retry?' type='submit' disabled={this.valida\
5   te()}/>,
6   READY: <input value='Submit' type='submit' disabled={this.validate()}/>
7 }}[this.state._saveStatus]}

```

What we have here are four different buttons – one for each possible `state._saveStatus`. Each button is the value of an object keyed by its corresponding status. By accessing the key of the current save status, this expression will evaluate to the appropriate button.

The last thing that we have to do is related to the "SUCCESS" case. We want to show the user that the addition was a success, and we do that by changing the text of the button. However, "Saved!" is not a call to action. If the user enters another person's information and wants to add it to the list, our button would still say "Saved!". It should say "Submit" to more accurately reflect its purpose.

The easy fix for this is to change our `state._saveStatus` back to "READY" as soon as they start entering information again. To do this, we update our `onInputChange()` handler:

code/forms/10-remote-persist.jsx

```
57  onInputChange({ name, value, error }) {
58    const fields = this.state.fields;
59    const fieldErrors = this.state.fieldErrors;
60
61    fields[name] = value;
62    fieldErrors[name] = error;
63
64    this.setState({ fields, fieldErrors, _saveStatus: 'READY' });
65  },
```

Now instead of just updating `state.fields` and `state.fieldErrors`, we also set `state._saveStatus` to 'READY'. This way after the user acknowledges their previous submit was a success and starts to interact with the app again, the button reverts to its "ready" state and invites the user to submit again.

At this point our sign-up app is a nice illustration of the features and issues that you'll want to cover in your own forms using React.

Redux

In this section we'll show how you can modify the form app we've built up so that it can work within a larger app using Redux.



Chronologically we haven't talked about Redux in this book. [The next two chapters](#) are all about Redux in depth. If you're unfamiliar with Redux, hop over to those chapters and come back here when you need to deal with forms in Redux.

Our form, which used to be our entire app, will now become a component. In addition, we'll adapt it to fit within the Redux paradigm. At a high level, this involves moving state and functionality from our form component to Redux reducers and actions. For example, we will no longer call API functions from within the form component – we use Redux async actions for that instead. Similarly, data that used to be held as state in our form will become read-only props – it will now be held in the Redux store.

When building with Redux, it is very helpful to start by thinking about the "shape" your state will take. In our case, we have a pretty good idea already since our functionality has been built. When using Redux, you'll want to centralize state as much state as possible – this will be the `store`, accessible by all components in the app. Here's what our `initialState` should look like:

code/forms/11-redux-reducer.js

```
6 const initialState = {
7   people: [],
8   isLoading: false,
9   saveStatus: 'READY',
10  person: {},
11};
```

No surprises here. Our app cares about the list of people who have signed up, the current person being typed in the form, whether or not we're loading, and the status of our save attempt.

Now that we know the shape of our state, we can think of different actions that would mutate it. For example, since we're keeping track of the list of people, we can imagine one action to retrieve the list from the server when the app starts. This action would affect multiple properties of our state. When the request to the server returns with the list, we'll want to update our state with it, and we'll also want to update `isLoading`. In fact, we'll want to set `isLoading` to true when we *start* the request, and we'll want to set it to `false` when the request finishes. With Redux, it's important to realize that we can often split one objective into multiple actions.

For our Redux app, we'll have five action types. The first two are related to the objective just mentioned, they are `FETCH_PEOPLE_REQUEST` and `FETCH_PEOPLE_SUCCESS`. Here are those action types with their corresponding action creator functions:

code/forms/11-redux-actions.js

```
1 export const FETCH_PEOPLE_REQUEST = 'FETCH_PEOPLE_REQUEST';
2 function fetchPeopleRequest () {
3   return {type: FETCH_PEOPLE_REQUEST};
4 }
5
6 export const FETCH_PEOPLE_SUCCESS = 'FETCH_PEOPLE_SUCCESS';
7 function fetchPeopleSuccess (people) {
8   return {type: FETCH_PEOPLE_SUCCESS, people};
9 }
```

When we start the request we don't need to provide any information beyond the action type to the reducer. The reducer will know that the request started just from the type and can update `isLoading` to `true`. When the request is successful, the reducer will know to set it to `false`, but we'll need to provide the people list for that update. This is why `people` is on the second action, `FETCH_PEOPLE_SUCCESS`.



We skip `FETCH_PEOPLE_FAILURE` only for expediency, but you'll want to handle fetch failures in your own app. See below for how to do that for saving the list.

We can now imagine dispatching these actions and having our state updated appropriately. To get the people list from the server we would dispatch the `FETCH_PEOPLE_REQUEST` action, use our API client to get the list, and finally dispatch the `FETCH_PEOPLE_SUCCESS` action (with the people list on it). With Redux, we'll use an asynchronous action creator, `fetchPeople()` to perform those actions:

code/forms/11-redux-actions.js

```
26 export function fetchPeople () {
27   return function (dispatch) {
28     dispatch(fetchPeopleRequest())
29     apiClient.loadPeople().then((people) => {
30       dispatch(fetchPeopleSuccess(people))
31     })
32   }
33 }
```

Instead of returning an action object, asynchronous action creators return functions that dispatch actions.



Asynchronous action creators are not supported by default with Redux. To be able to dispatch functions instead of action objects, we'll need to use the `redux-thunk` middleware when we create our store.

We'll also want to create actions for *saving* our list to the server. Here's what they look like:

code/forms/11-redux-actions.js

```
11 export const SAVE_PEOPLE_REQUEST = 'SAVE_PEOPLE_REQUEST';
12 function savePeopleRequest () {
13   return {type: SAVE_PEOPLE_REQUEST};
14 }
15
16 export const SAVE_PEOPLE_FAILURE = 'SAVE_PEOPLE_FAILURE';
17 function savePeopleFailure (error) {
18   return {type: SAVE_PEOPLE_FAILURE, error};
19 }
20
21 export const SAVE_PEOPLE_SUCCESS = 'SAVE_PEOPLE_SUCCESS';
22 function savePeopleSuccess (people) {
23   return {type: SAVE_PEOPLE_SUCCESS, people};
24 }
```

Just like the fetch we have `SAVE_PEOPLE_REQUEST` and `SAVE_PEOPLE_SUCCESS`, but we also have `SAVE_PEOPLE_FAILURE`. The `SAVE_PEOPLE_REQUEST` action happens when we start the request, and like before we don't need to provide any data besides the action type. The reducer will see this type and know to update `saveStatus` to '`SAVING`'. Once the request resolves, we can trigger either `SAVE_PEOPLE_SUCCESS` or `SAVE_PEOPLE_FAILURE` depending on the outcome. We will want to pass additional data with these though – people on a successful save and error on a failure.

Here's how we use those together within an asynchronous action creator, `savePeople()`:

code/forms/11-redux-actions.js

```
35 export function savePeople (people) {
36   return function (dispatch) {
37     dispatch(savePeopleRequest())
38     apiClient.savePeople(people)
39       .then((resp) => { dispatch(savePeopleSuccess(people)) })
40       .catch((err) => { dispatch(savePeopleFailure(err)) })
41   }
42 }
```

Now that we've defined all of our action creators, we have everything we need for our reducer. By using the two asynchronous action creators above, the reducer can make all the updates to our state that our app will need. Here's what our reducer looks like:

code/forms/11-redux-reducer.js

```
6 const initialState = {
7   people: [],
8   isLoading: false,
9   saveStatus: 'READY',
10  person: {},
11 };
12
13 export function reducer (state = initialState, action) {
14   switch (action.type) {
15     case FETCH_PEOPLE_REQUEST:
16       return Object.assign({}, state, {
17         isLoading: true
18       });
19     case FETCH_PEOPLE_SUCCESS:
20       return Object.assign({}, state, {
21         people: action.people,
22         isLoading: false
23       });
24 }
```

```
24  case SAVE_PEOPLE_REQUEST:
25    return Object.assign({}, state, {
26      saveStatus: 'SAVING'
27    });
28  case SAVE_PEOPLE_FAILURE:
29    return Object.assign({}, state, {
30      saveStatus: 'ERROR'
31    });
32  case SAVE_PEOPLE_SUCCESS:
33    return Object.assign({}, state, {
34      people: action.people,
35      person: {},
36      saveStatus: 'SUCCESS'
37    });
38  default:
39    return state;
40  }
41  return state;
42 }
```

By just looking at the actions and the reducer you should be able to see all the ways our state can be updated. This is one of the great things about Redux. Because everything is so explicit, state becomes very easy to reason about and test.

Now that we've established the shape of our state and how it can change, we'll create a store. Then we'll want to make some changes so that our form can connect to it properly.

Form Component

Now that we've created the foundation of our app's data architecture with Redux, we can adapt our form component to fit in. In broad strokes, we need to remove any interaction with the API client (our asynchronous action creators handle this now) and shift dependence from component-level state to props (Redux state will be passed in as props).

The first thing we need to do is set up propTypes that will align with the data we expect to get from Redux:

`code/forms/11-redux-form.jsx`

```
10  propTypes: {
11    people: PropTypes.array.isRequired,
12    isLoading: PropTypes.bool.isRequired,
13    saveStatus: PropTypes.string.isRequired,
14    fields: PropTypes.object,
15    onSubmit: PropTypes.func.isRequired,
16  },
```

We will require one additional prop that is not related to data in our Redux store, `onSubmit()`. When the user submits a new person, instead of using the API client, our form component will call this function instead. Later we'll show how we hook this up to our asynchronous action creator `savePeople()`.

Next, we limit the amount of data that we'll keep in state. We keep `fields` and `fieldErrors`, but we remove `people`, `_loading`, and `_saveStatus` – those will come in on props. Here's the updated `getInitialState()`

`code/forms/11-redux-form.jsx`

```
18  getInitialState: function () {
19    return {
20      fields: this.props.fields || {},
21      fieldErrors: {},
22    };
23  },
```

`state.fields` will be initialized to `props.fields` (or `{}` if not provided). Additionally, if a new `fields` object comes in on `props`, we will update our state:

`code/forms/11-redux-form.jsx`

```
25  componentWillReceiveProps(update) {
26    this.setState({ fields: update.fields });
27  },
```

Now that our `props` and `state` are in order, we can remove any usage of `apiClient` since that will be handled by our asynchronous action creators. The two places that we used the API client were in `componentWillMount()` and `onFormSubmit()`.

Since the only purpose of `componentWillMount()` was to use the API client, we have removed it entirely. In `onFormSubmit()`, we remove the block related to the API and replace it with a call to `props.onSubmit()`:

code/forms/11-redux-form.jsx

```
29  onFormSubmit(evt) {
30    const person = this.state.fields;
31
32    evt.preventDefault();
33
34    if (this.validate()) return;
35
36    this.props.onSubmit([ ...this.props.people, person ]);
37 },
```

With all of that out of the way, we can make a few minor updates to `render()`. In fact, the only modifications we have to make to `render()` are to replace references to `state._loading`, `state._saveStatus`, and `state.people` with their counterparts on `props`.

code/forms/11-redux-form.jsx

```
63 render() {
64   if (this.props.isLoading) {
65     return <img alt='loading' src='/img/loading.gif' />;
66   }
67
68   const dirty = Object.keys(this.state.fields).length;
69   let status = this.props.saveStatus;
70   if (status === 'SUCCESS' && dirty) status = 'READY';
71
72   return (
73     <div>
74       <h1>Sign Up Sheet</h1>
75
76       <form onSubmit={this.onFormSubmit}>
77
78         <Field
79           placeholder='Name'
80           name='name'
81           value={this.state.fields.name}
82           onChange={this.onInputChange}
83           validate={(val) => (val ? false : 'Name Required')}
84         />
85
86         <br />
87       </form>
88     </div>
89   );
90 }
```

```
88     <Field
89         placeholder='Email'
90         name='email'
91         value={this.state.fields.email}
92         onChange={this.onInputChange}
93         validate={(val) => (isEmail(val) ? false : 'Invalid Email')}
94     />
95
96     <br />
97
98     <CourseSelect
99         department={this.state.fields.department}
100        course={this.state.fields.course}
101        onChange={this.onInputChange}
102    />
103
104    <br />
105
106    {{
107        SAVING: <input value='Saving...' type='submit' disabled />,
108        SUCCESS: <input value='Saved!' type='submit' disabled />,
109        ERROR: <input
110            value='Save Failed - Retry?'
111            type='submit'
112            disabled={this.validate()}>
113        />,
114        READY: <input
115            value='Submit'
116            type='submit'
117            disabled={this.validate()}>
118        />,
119    }[status]}
120
121    </form>
122
123    <div>
124        <h3>People</h3>
125        <ul>
126            {this.props.people.map(({ name, email, department, course }, i) =>
127                <li key={i}>{[ name, email, department, course ].join(' - ')</li>
128            ) }
129        </ul>
```

```
130      </div>
131      </div>
132  );
133 },
```



You may notice that we handle `saveStatus` a bit differently. In the previous iteration, our form component was able to control `state._saveStatus` and could set it to 'READY' on a field change. In this version, we get that information from `props.saveStatus` and it is read-only. The solution is to check if `state.fields` has any keys – if it does, we know the user has entered data and we can set the button back to the "ready" state.

Connect the Store

At this point we have our actions, our reducer, and our streamlined form component. All that's left is to connect them together.

First, we will use Redux's `createStore()` method to create a store from our reducer. Because we want to be able to dispatch asynchronous actions, we will also use `thunkMiddleware` from the `redux-thunk` module. To use middleware in our store, we'll use Redux's `applyMiddleware()` method. Here's what that looks like:

code/forms/11-redux-app.jsx

```
10 const store = createStore(reducer, applyMiddleware(thunkMiddleware));
```

Next, we will use the `connect()` method from `react-redux` to optimize our form component for use with Redux. We do this by providing it two methods: `mapStateToProps` and `mapDispatchToProps`.

When using Redux, we want our components to subscribe to the store. However, with `react-redux` it will do that for us. All we need to do is provide a `mapStateToProps` function that defines the mapping between data in the store and props for the component. In our app, they line up neatly:

code/forms/11-redux-app.jsx

```
30 function mapStateToProps(state) {
31   return {
32     isLoading: state.isLoading,
33     fields: state.person,
34     people: state.people,
35     saveStatus: state.saveStatus,
36   };
37 }
```

From within our form component, we call `props.onSubmit()` when the user submits and validation passes. We want this behavior to dispatch our `savePeople()` asynchronous action creator. To do this, we provide `mapDispatchToProps()` to define the connection between the `props.onSubmit()` function and the dispatch of our action creator:

code/forms/11-redux-app.jsx

```
39 function mapDispatchToProps(dispatch) {
40   return {
41     onSubmit: (people) => {
42       dispatch(savePeople(people));
43     },
44   };
45 }
```

With both of those functions created, we use the `connect()` method from `react-redux` to give us an optimized `ReduxForm` component:

code/forms/11-redux-app.jsx

```
12 const ReduxForm = connect(mapStateToProps, mapDispatchToProps)(Form);
```

The final step is to incorporate the `store` and the `ReduxForm` into our app. At this point our app is a very simple component with only two methods, `componentWillMount()` and `render()`.

In `componentWillMount()` we dispatch our `fetchPeople()` asynchronous action to load the people list from the server:

code/forms/11-redux-app.jsx

```
17 componentWillMount() {
18   store.dispatch(fetchPeople());
19 }
```

In `render()` we use a helpful component `Provider` that we get from `react-redux`. `Provider` will make the `store` available to all of its child components. We simply place `ReduxForm` as a child of `Provider` and our app is good to go:

code/forms/11-redux-app.jsx

```
21 render() {
22   return (
23     <Provider store={store}>
24       <ReduxForm />
25     </Provider>
26   );
27 }
```

And that's it! Our form now fits neatly inside a Redux-based data architecture.

After reading this chapter, you should have a good handle on the fundamentals of forms in React. That said, if you'd like to outsource some portion of your form handling to an external module, there are several available. Read on for a list of some of the more popular options.

Form Modules

formsy-react

[https://github.com/christianalfoni/formsy-react⁵⁴](https://github.com/christianalfoni/formsy-react)

formsy-react tries to strike a balance between flexibility and reusability. This is a worthwhile goal as the author of this module acknowledges that forms, inputs, and validation are handled quite differently across projects.

The general pattern is that you use the `Formsy.Form` component as your form element, and provide your own input components as children (using the `Formsy.Mixin`). The `Formsy.Form` component has handlers like `onValidSubmit()` and `onInvalid()` that you can use to alter state on the form's parent, and the mixin provides some validation and other general purpose helpers.

react-input-enhancements

[http://alexkuz.github.io/react-input-enhancements⁵⁵](http://alexkuz.github.io/react-input-enhancements)

react-input-enhancements is a collection of five rich components that you can use to augment forms. This module has a nice demo to showcase how you can use the `Autosize`, `Autocomplete`, `Dropdown`, `Mask`, and `DatePicker` components. The author does make a note that they aren't quite ready for production and are more conceptual. That said, they might be useful if you're looking for a drop-in datepicker or autocomplete element.

⁵⁴<https://github.com/christianalfoni/formsy-react>

⁵⁵<http://alexkuz.github.io/react-input-enhancements>

tcomb-form

<http://gcanti.github.io/tcomb-form>⁵⁶

tcomb-form is meant to be used with tcomb models (<https://github.com/gcanti/tcomb>⁵⁷) which center around Domain Driven Design. The idea is that once you create a model, the corresponding form can be automatically generated. In theory, the benefits are that you don't have to write as much markup, you get usability and accessibility for free (e.g. automatic labels and inline validation), and your forms will automatically stay in sync with changes to your model. If tcomb models seem to be a good fit for your app, this tcomb-form is worth considering.

winterfell

<https://github.com/andrewthaway/winterfell>⁵⁸

If the idea of defining your forms and fields entirely with JSON, winterfell might be for you. With winterfell, you sketch out your entire form in a JSON schema. This schema is a large object where you can define things like CSS class names, section headers, labels, validation requirements, field types, and conditional branching. winterfell is organized into "form panels", "question panels", and "question sets". Each panel has an ID and that ID is used to assign sets to it. One benefit of this approach is that if you find yourself creating/modifying lots of forms, you could create a UI to create/modify these schema objects and persist them to a database.

react-redux-form

<https://github.com/davidkpiano/react-redux-form>⁵⁹

If Redux is more your style react-redux-form is a "collection of action creators and reducer creators" to simplify "building complex and custom forms with React and Redux". In practice, this module provides a `modelReducer` and a `formReducer` helper to use when creating your Redux store. Then within your form you can use the provided `Form`, `Field`, and `Error` components to help connect your `label` and `input` elements to the appropriate reducers, set validation requirements, and display appropriate errors. In short, this is a nice thin wrapper to help you build forms using Redux.

⁵⁶<http://gcanti.github.io/tcomb-form>

⁵⁷<https://github.com/gcanti/tcomb>

⁵⁸<https://github.com/andrewthaway/winterfell>

⁵⁹<https://github.com/davidkpiano/react-redux-form>

Intro to Flux and Redux

Why Flux?

In our projects so far, we've managed state inside of React components. The top-level React component managed our primary state. In this type of data architecture, data flows downward to child components. To make changes to state, child components propagate events up to their parent components by calling prop-functions. Any state mutations took place at the top and then flowed downward again.

Managing application state with React components works fine for a wide variety of applications. However, as apps grow in size and complexity, managing state inside of React components (or the *component-state paradigm*) can become cumbersome.

A common pain point is the **tight coupling between user interactions and state changes**. For complex web applications, oftentimes a single user interaction can affect many different, discrete parts of the state.

For example, consider an app for managing email. Clicking an email must:

1. Replace the “inbox view” (the list of emails) with the “email view” (the email the user clicked)
2. Mark the email as read locally
3. Reduce the total unread counter locally
4. Change the URL of the browser
5. Send a web request to mark the email as read on the server

The function in the top-level component that handles a user clicking on an email must describe all of the state changes that occur. This loads a single function with lots of complexity and responsibility. Wading through all of this logic for managing many disparate parts of an app's state tree can make updates difficult to manage and error-prone.

Facebook was running into this and other architectural problems with their apps. This motivated them to invent Flux.

Flux is a Design Pattern

Flux is a design pattern. The predecessor to Flux at Facebook was another design pattern, [Model-View-Controller⁶⁰](#) (MVC). MVC is a popular design pattern for both desktop and web applications.

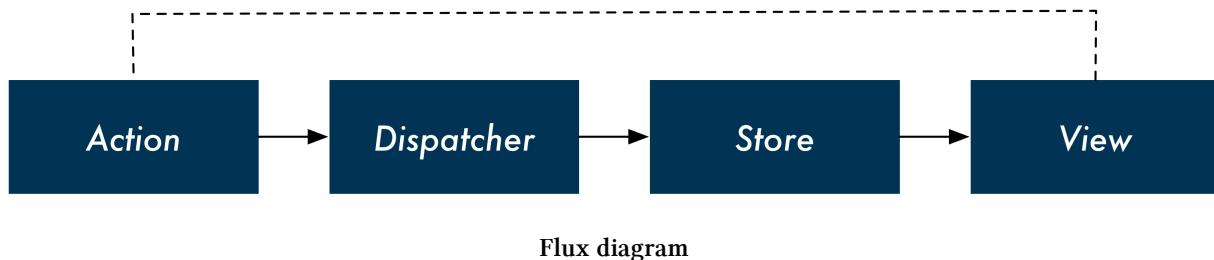
⁶⁰<https://en.wikipedia.org/wiki/Model–view–controller>

In MVC, user interactions with the View trigger logic in the Controller. The Controller instructs the Model how to update itself. After the Model updates, the View re-renders.

While React does not have three discrete “actors” like a traditional MVC implementation, it suffers from the same coupling between user interactions and state changes.

Flux overview

The Flux design pattern is made up of four parts, organized as a one-way data pipeline:



Flux diagram

The **view** dispatches **actions** that describe what happened. The **store** receives these actions and determines what state changes should occur. After the state updates, the new state is pushed to the View.

Returning to our email example, in Flux, we no longer have a single function handling email clicks that describes all of the state changes. Instead, React notifies the store (through an action) that the user clicked on an email. As we'll see over the next few chapters, we can organize the store such that each discrete part of the state has its own logic for handling updates.

In addition to decoupling interaction handling and state changes, Flux also provides the following benefits:

Breaking up state management logic

As parts of the state tree become interdependent, most of an app's state usually gets rolled up to a top-level component. Flux relieves the top-level component of state management responsibility and allows you to break up state management into isolated, smaller, and testable parts.

React components are simpler

Certain component-managed state is fine, like activating certain buttons on mouse hover. But by managing all other state externally, React components become simple HTML rendering functions. This makes them smaller, easier to understand, and more composable.

Mis-match between the state tree and the DOM tree

Oftentimes, we want to store our state with a different representation than how we want to display it. For example, we might want to have our app store a timestamp for a message (`createdAt`) but in the view we want to display a human-friendly representation, like “23 minutes ago.” Instead of

having components hold all this computational logic for *derived data*, we'll see how Flux enables us to perform these computations *before* providing state to React components.

We'll reflect on these benefits as we dig deep into the design of a complex application in the next chapter. Before that, we'll implement the Flux design pattern in a basic application so that we can review Flux's fundamentals.

Flux implementations

Flux is a design pattern, not a specific library or implementation. Facebook has [open-sourced a library they use⁶¹](#). This library provides the interface for a dispatcher and a store that you can use in your application.

But Facebook's implementation is not the exclusive option. Since Facebook started sharing Flux with the community, the community has responded by writing [tons of different Flux implementations⁶²](#). A developer has many compelling choices.

While the available choices can be overwhelming, one community favorite has emerged: [Redux⁶³](#).

Redux

Redux has gained widespread popularity and respect in the React community. The library has even won [the endorsement of the creator of Flux⁶⁴](#).

Redux's best feature is its simplicity. Stripped of its comments and sanity checks, Redux is only about 100 lines of code.

Because of this simplicity, throughout this chapter we'll be implementing the Redux core library ourselves. We'll use small example applications to see how everything fits together.

In the following chapters, we'll build on this foundation by constructing a feature-rich messaging application that mirrors Facebook's. We'll see how using Redux as the backbone of our application equips our app to handle increasing feature complexity.

Redux's key ideas

Throughout this chapter, we'll become familiar with each of Redux's key ideas. Those ideas are:

- All of your application's data is in a single data structure called the **state** which is held in the **store**

⁶¹<https://github.com/facebook/flux>

⁶²<https://github.com/voroniantski/flux-comparison>

⁶³<https://github.com/reactjs/redux>

⁶⁴<https://twitter.com/jingc/status/616608251463909376>

- Your app reads the **state** from this **store**
- The **state** is never mutated directly outside the **store**
- The **views** emit **actions** that describe what happened
- A **new state** is created by combining the **old state** and the **action** by a function called the **reducer**

These key ideas are probably a bit cryptic at the moment, but you'll come to understand each of them over the course of this chapter.



Throughout the rest of the chapter, we'll be referring to Redux. Because Redux is an implementation of Flux, many of the concepts that apply to Redux apply to Flux as well.

While the Flux creators approve of Redux, Redux is arguably not a "strict" Flux implementation. You can read about the nuances on the [Redux website⁶⁵](#).

Building a counter

We'll explore Redux's core ideas by building a simple counter. For now, we'll focus only on Redux and state management. We'll see later how Redux connects to React views.

Preparation

Inside of the code download that came with this book, navigate to `redux/counter`:

```
$ cd redux/counter
```

All the code for the counter will go inside `app.js`.

Because we're focusing on Redux and state management to start, we'll be running our code in the terminal as opposed to the browser.

The `package.json` for both of the projects contains the package `babel-cli`. As we'll indicate in the **Try it out** sections below, we'll be using the `babel-node` command that comes with `babel-cli` to run our code examples:

```
# example of using `babel-node` to run code in the terminal
$ ./node_modules/.bin/babel-node app.js
```

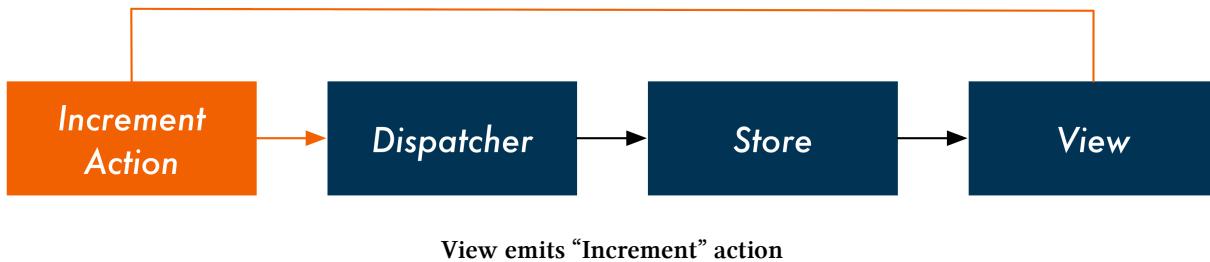
Run `npm install` now inside of `redux/counter` to install `babel-cli`:

⁶⁵<http://redux.js.org/docs/introduction/PriorArt.html>

```
$ npm install
```

Overview

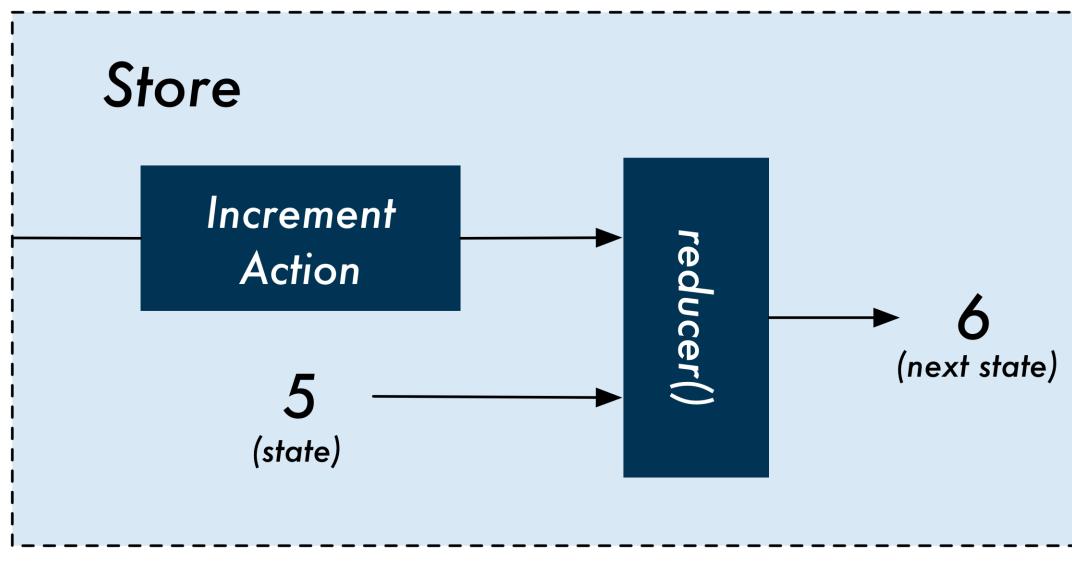
Our state will be a number. The number will start off as 0. Our actions will either be to **increment** or **decrement** the state. We know from our Redux diagram that the views would dispatch these actions to the store:



When the store receives an action from the views, the store uses a **reducer function** to process the action. The store provides the reducer function with the current state and the action. The reducer function returns the new state:

```
// Inside the store, receives `action` from the view
state = reducer(state, action);
```

For example, consider a store with a current state of 5. The store receives an increment action. The store uses its reducer to derive the next state:



We'll start building our Redux counter by constructing its reducer. We'll then work our way up to see what a Redux store looks like. Our store will be the maintainer of state, accepting actions and using the reducer to determine the next version of the state.



While we're starting with a simple representation of state (a number), we'll be working with much more complicated state in the next chapter.

The counter's actions

We know the reducer function for our counter will accept two arguments, `state` and `action`. We know `state` for our counter will be an integer. But how are actions represented in Redux?

Actions in Redux are objects. Actions always have a `type` property.

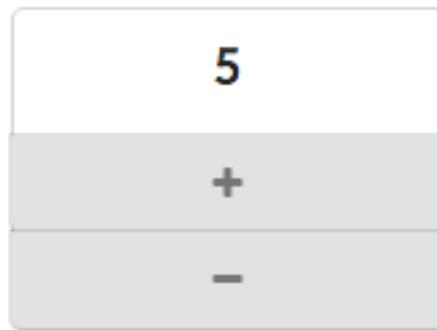
Our increment actions will look like this:

```
{  
  type: 'INCREMENT',  
}
```

And decrement actions like this:

```
{  
  type: 'DECREMENT',  
}
```

We can envision what a simple interface for this counter app might look like:



An example counter interface

When the user clicks the “+” icon, the view would dispatch the increment action to the store. When the user clicks the “-“ icon, the view would dispatch the decrement action to the store.



The image of the interface for the counter app is provided as just an example of what the view *might* look like. We will not be implementing a view layer for this app.

Incrementing the counter

Let's begin writing our reducer function. We'll start by handling the increment action.

The reducer function for our counter accepts two arguments, `state` and `action`, and returns the next version of the state. When the reducer receives an `INCREMENT` action, it should return `state + 1`.

Inside of `app.js`, add the code for our counter's reducer:

`redux/counter/app.js`

```
1 function reducer(state, action) {
2   if (action.type === 'INCREMENT') {
3     return state + 1;
4   } else {
5     return state;
6   }
7 }
```

If the `action.type` is `INCREMENT`, we return the incremented state. Otherwise, our reducer returns the state unmodified.



You might be wondering if it would be a better idea to raise an error if our reducer receives an `action.type` that it does not recognize.

In the next chapter, we'll see how **reducer composition** "breaks up" state management into smaller, more focused functions. These smaller reducers might only handle a subset of the app's state and actions. As such, if they receive an action they do not recognize, they should just ignore it and return the state unmodified.

Try it out

At the bottom of `app.js`, let's add some code to test our reducer.

We'll call our reducer, passing in integers for `state` and seeing how the reducer increments the number. If we pass in an unknown action type, our reducer returns the state unchanged:

redux/counter/app.js

```
9 const incrementAction = { type: 'INCREMENT' };
10
11 console.log(reducer(0, incrementAction)); // -> 1
12 console.log(reducer(1, incrementAction)); // -> 2
13 console.log(reducer(5, incrementAction)); // -> 6
14
15 const unknownAction = { type: 'UNKNOWN' };
16
17 console.log(reducer(5, unknownAction)); // -> 5
18 console.log(reducer(8, unknownAction)); // -> 8
```

Save app.js and run it with ./node_modules/.bin/babel-node:

```
$ ./node_modules/.bin/babel-node app.js
```

And your output should look like this:

```
1
2
6
5
8
```

Decrementing the counter

Again, decrement actions have a type of DECREMENT:

```
{
  type: 'DECREMENT',
}
```

To support decrement actions, we add another clause to our reducer:

redux/counter/app.js

```
1 function reducer(state, action) {  
2   if (action.type === 'INCREMENT') {  
3     return state + 1;  
4   } else if (action.type === 'DECREMENT') {  
5     return state - 1;  
6   } else {  
7     return state;  
8   }  
9 }
```

Try it out

At the bottom of `app.js`, below the code where we dispatched increment actions, add some code to dispatch decrement actions:

redux/counter/app.js

```
22 const decrementAction = { type: 'DECREMENT' };  
23  
24 console.log(reducer(10, decrementAction)); // -> 9  
25 console.log(reducer(9, decrementAction)); // -> 8  
26 console.log(reducer(5, decrementAction)); // -> 4
```

Run `app.js` with `./node_modules/.bin/babel-node`:

```
$ ./node_modules/.bin/babel-node app.js
```

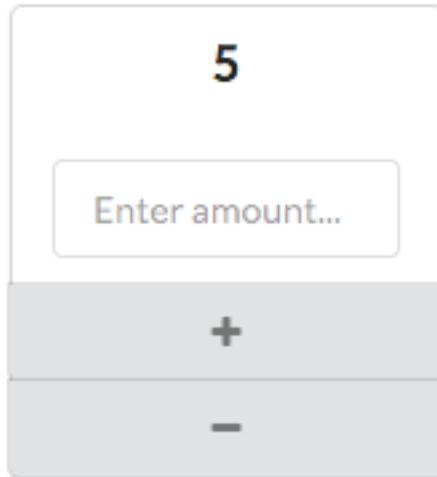
And your output should look like this:

```
1  
2  
6  
5  
8  
9  
8  
4
```

Supporting additional parameters on actions

In the last example, our actions contained only a type which told our reducer either to increment or decrement the state. But often behavior in our app can't be described by a single value. In these cases, we need additional parameters to describe the change.

For example, what if we wanted our app to allow the user to specify an *amount* to increment or decrement by?



An example counter interface with an amount field

We'll have our actions carry the additional property `amount`. An `INCREMENT` action would then look like this:

```
{  
  type: 'INCREMENT',  
  amount: 7,  
}
```

We modify our reducer to increment and decrement by `action.amount`, expecting all actions to now carry this property:

redux/counter/app.js

```
1 function reducer(state, action) {
2   if (action.type === 'INCREMENT') {
3     return state + action.amount;
4   } else if (action.type === 'DECREMENT') {
5     return state - action.amount;
6   } else {
7     return state;
8   }
9 }
```

Try it out

Clear out the code we used to test out `reducer()` in `app.js` previously.

This time, we'll test calling the reducer with our modified actions that now carry the `amount` property:

redux/counter/app.js

```
11 const incrementAction = {
12   type: 'INCREMENT',
13   amount: 5,
14 };
15
16 console.log(reducer(0, incrementAction)); // -> 5
17 console.log(reducer(1, incrementAction)); // -> 6
18
19 const decrementAction = {
20   type: 'DECREMENT',
21   amount: 11,
22 };
23
24 console.log(reducer(100, decrementAction)); // -> 89
```

Run `app.js` with `./node_modules/.bin/babel-node`:

```
$ ./node_modules/.bin/babel-node app.js
```

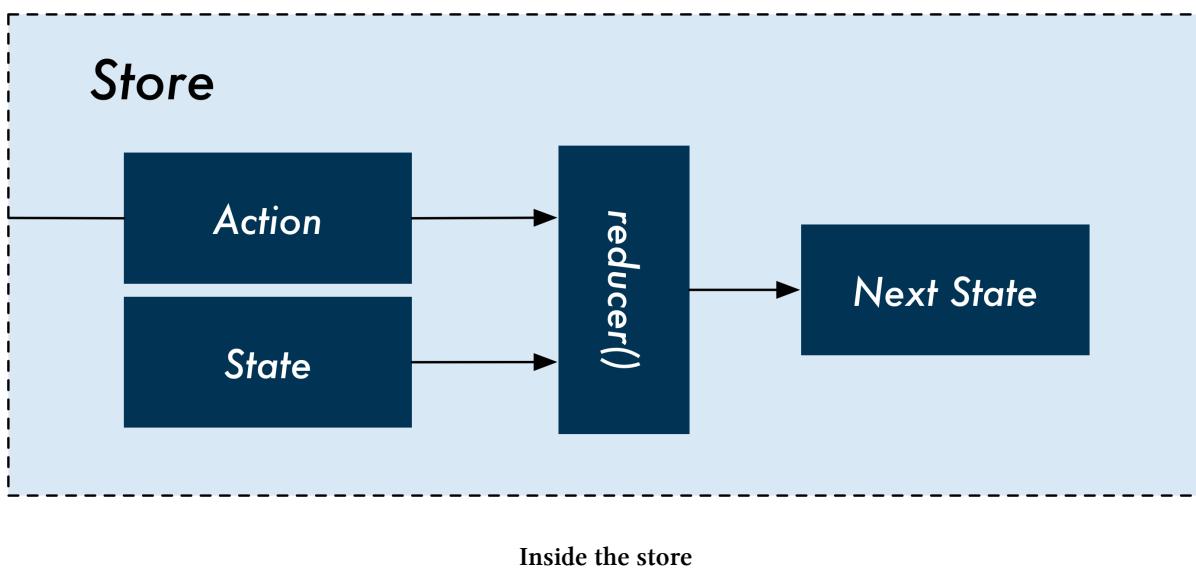
And note the output:

5
6
89

Building the store

So far, we've been calling our reducer and manually supplying the last version of the state along with an action.

In Redux, the store is responsible for both maintaining the state and accepting actions from the view. Only the store has access to the reducer:



The Redux library provides a function for creating stores, `createStore()`. This function returns a store object that keeps an internal variable, `state`. In addition, it provides a few methods for interacting with the store.

We will write our own version of `createStore()` so that we fully understand how Redux stores work. By the end of this chapter, our code for `createStore()` will behave almost exactly like the one provided by the Redux library.

At the moment, our store will provide two methods:

- `dispatch`: The method we'll use to send the store actions
- `getState`: The method we'll use to read the current value of `state`

Inside of `app.js`, clear out the code we used to test out `reducer()` previously. Below the definition of `reducer()`, let's define `createStore()`. `createStore()` will accept a single argument, the desired reducer for the store.

Let's take a look at the full `createStore()` function. We'll break it down piece-by-piece below the code block:

`redux/counter/app.js`

```
11 function createStore(reducer) {
12   let state = 0;
13
14   const getState = () => (state);
15
16   const dispatch = (action) => {
17     state = reducer(state, action);
18   };
19
20   return {
21     getState,
22     dispatch,
23   };
24 }
```

The `reducer` argument

`createStore()` accepts a single argument, `reducer`. This is how we will indicate what reducer function our store should use.

`state`

We initialize the `state` to `0` at the top of `createStore()`. Note that we close over the `state` variable. This makes `state` private and inaccessible outside of `createStore()`.



For more info on closures, see the aside “[The Factory Pattern](#).”

`getState`

To get read access to the `state` from outside `createStore()`, we have the method `getState` which returns `state`.

`dispatch`

The `dispatch` method is how we send actions to the store. We'll call it like this:

```
store.dispatch({ type: 'INCREMENT', amount: 7 });
```

`dispatch` calls the reducer function passed in as an argument with the current state and the action. `dispatch` sets `state` to the reducer's return value.

Note that `dispatch` **does not return the state**. Dispatching actions in Redux are “fire-and-forget.” When we call `dispatch`, we’re sending a notification to the store with no expectation on when or how that action will be processed.

Dispatching actions to the store is decoupled from reading the latest version of the state. We’ll see how this works in practice when we connect the store to React views at the end of this chapter.

The return object

At the bottom of `createStore()`, we return a new object. This object has `getState` and `dispatch` as methods.

ES6: Enhanced object literals

The return object of `createStore()` above uses ES6’s [enhanced literal syntax^a](#):

```
{  
  getState,  
  dispatch,  
}
```

This is the same as writing:

```
{  
  getState: getState,  
  dispatch: dispatch,  
}
```

You can use this terser syntax whenever the property name and variable name are the same.

Lots of open source libraries use this syntax, so it’s important to be familiar with it. Whether you use it in your own code is a matter of stylistic preference.

^a<https://github.com/lukehoban/es6features#enhanced-object-literals>

The Factory Pattern

In `createStore()` above, we’re using a pattern called the “Factory Pattern.” This is a ubiquitous pattern in JavaScript for creating complex objects like our store object.

The Factory Pattern provides a **closure** for variables declared inside the factory function.

At the top of `createStore()`, we declare the variable `state`:

```
function createStore(reducer) {  
  let state = 0;  
  // ...
```

`state` is a **private variable**. Only the functions declared inside of `createStore()` have access to it. Furthermore, because `state` is inside of this closure, the variable “lives on” between function calls.

As an example, consider the following factory function:

```
function createAdder() {
  let value = 0;

  const add = (amount) => (value = value + amount);
  const getValue = () => (value);

  return {
    add,
    getValue,
  }
}
```

We first call the factory to instantiate our `adder` object. The private variable `value` is initialized to `0`:

```
const adder = createAdder();
```

While `createAdder()` has returned our new object and exited, the variable `value` lives on in memory as `0`. Whenever we call `add()`, we are modifying this same `value`:

```
adder.add(1);
adder.getValue();
// => 1
adder.add(1);
adder.getValue();
// => 2
adder.add(5);
adder.getValue();
// => 7
```

Importantly, `value` is *only accessible to the functions inside the factory*. This prevents unintended reads or writes.

In the case with our store, this prevents any modifications from being made to `state` outside of the `dispatch()` function.

Try it out

We'll write the code to test out our store in `app.js` below `createStore()`.

We'll create our store object with `createStore()`. Then, instead of calling `reducer()` with a state

and an action, we'll dispatch actions to the store. Because our store is keeping the internal variable state, our state persists between dispatches.

We can use `getState()` to read state between dispatches:

`redux/counter/app.js`

```
26 const store = createStore(reducer);
27
28 const incrementAction = {
29   type: 'INCREMENT',
30   amount: 3,
31 };
32
33 store.dispatch(incrementAction);
34 console.log(store.getState()); // -> 3
35 store.dispatch(incrementAction);
36 console.log(store.getState()); // -> 6
37
38 const decrementAction = {
39   type: 'DECREMENT',
40   amount: 4,
41 };
42
43 store.dispatch(decrementAction);
44 console.log(store.getState()); // -> 2
```

Run `app.js` with `./node_modules/.bin/babel-node`:

```
$ ./node_modules/.bin/babel-node app.js
```

And note the output:

```
3
6
2
```

The core of Redux

As it stands, our `createStore()` function closely resembles the `createStore()` function that ships with the Redux library. By the end of this chapter, we'll have made just a couple of tweaks and additions to `createStore()` to bring it closer to that of the Redux library.

Now that we've seen a Redux store in action, let's recap Redux's key ideas:

All of your application's data is in a single data structure called the state which is held in the store.

We saw that the store has a single private variable for the state, `state`.

Your app reads the state from this store.

We use `getState()` to access the store's state.

The state is never mutated directly outside the store.

Because `state` is a private variable, it cannot be mutated outside of the store.

The views emit actions that describe what happened.

We use `dispatch()` to send these actions to the store.

A new state is created by combining the old state and the action by a function called the reducer.

Inside of `dispatch()`, our store uses `reducer()` to get the new state, passing in the current state and the action.

There is one more key idea of Redux we have yet to cover:

Reducers functions must be pure functions.

We will explore this idea in the next app.

Next up

In the next app and over the next two chapters, we'll work with examples of increasing complexity. All the ideas we cover are patterns that flow from this core: a single store controls state, making updates by using a reducer. This reducer takes the current state and an action and returns a new state.

If you understand the ideas presented above, it's likely you'd be able to invent many of the patterns and libraries we'll be discussing.

To see how Redux operates inside of a feature-rich web application, we'll cover:

- How to carefully handle more complex data structures in our state
- How to be notified when our state changes without having to poll the store with `getState()` (*with subscriptions*)
- How to split up large reducers into more manageable, smaller ones (and recombine them)
- How to organize our React components in a Redux-powered app

Let's first deal with handling more complex data structures in our state. For the remainder of this chapter, we'll switch gears from our counter app to the beginnings of a chat app. In subsequent chapters, the interface for our chat app will begin to mirror the richness and complexity of Facebook's.

The beginnings of a chat app

Previewing

We'll build our chat app inside of the folder `redux/chat_simple`. From inside of the `redux/counter` directory, you can type:

```
$ cd ../chat_simple
```

First, run `npm install`:

```
$ npm install
```

Run `ls -1` to see the contents of this folder:

```
$ ls -1
README.md
node_modules
package.json
public
```

As with previous apps, there are two different JavaScript files under `public/`:

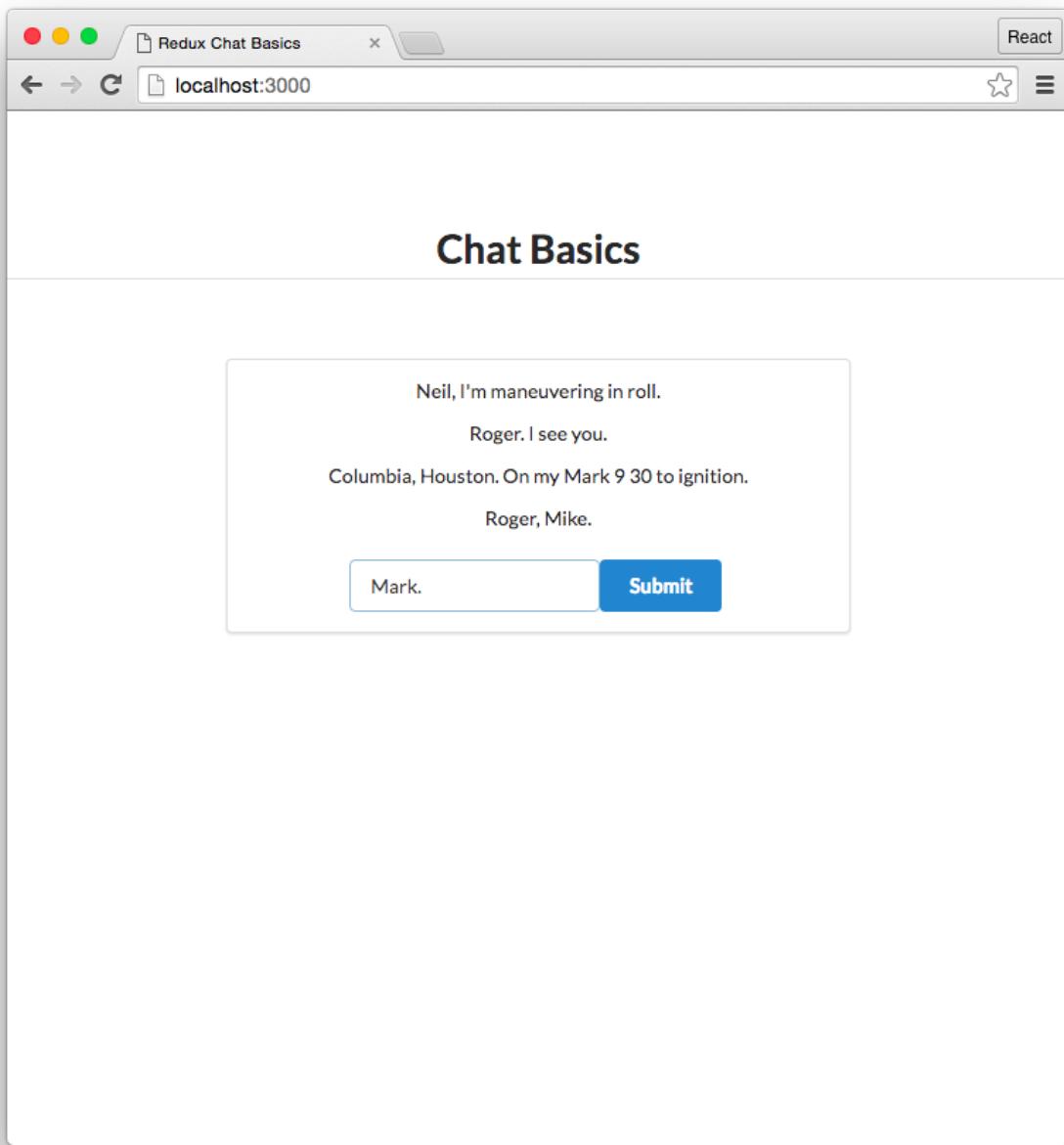
- `app-complete.js`
- `app.js`

```
$ ls -1 public/
app-complete.js
app.js
index.html
style.css
vendor
```

Right now, `app-complete.js` is being loaded in `index.html`. We can boot our app:

```
$ npm run server
```

And then navigate to `localhost:3000` in our browser to see the completed version of the chat app iteration we build in this chapter:



The iteration of the chat app that we build in this chapter

Over the next few chapters, we'll be significantly enhancing the feature-set (and complexity) of this app. For now, we can add messages using the input box and delete messages by clicking on them.

Quit the server with **CTRL+C** when you're ready to continue.

Open up `public/index.html` and remove the line that includes `app-complete.js`:

```
1 <script type="text/babel" src=".//app.js"></script>
2 <!-- Delete the line below to get started. -->
3 <script type="text/babel" src=".//app-complete.js"></script>
```

The code for the remainder of this chapter will go inside of `app.js`. When you open that file, you'll see that the same `createStore()` that we built for the counter app is already there.

As with the counter app, we'll start by building the chat app's reducer. Once our reducer is built, we'll see how we connect our Redux store to React views.

Before we build our reducer, though, we should examine how the chat app will represent both its state and its actions.

State

The state in our counter app was a single number. In our chat app, the state is going to be an object.

This state object will have a single property, `messages`. `messages` will be an array of strings, with each string representing an individual message in the application. For example:

```
// an example `state` value
{
  messages: [
    'here is message one',
    'here is message two',
  ],
}
```

Actions

Our app will process two actions: `ADD_MESSAGE` and `DELETE_MESSAGE`.

The `ADD_MESSAGE` action object will always have the property `message`, the message to be added to the state. The `ADD_MESSAGE` action object has this shape:

```
{
  type: 'ADD_MESSAGE',
  message: 'Whatever message is being added here',
}
```

The `DELETE_MESSAGE` action object will delete a specified message from the state.

If each of our messages were objects, we could assign each message an `id` property when it is created. `DELETE_MESSAGE` could then specify the message to delete with an `id` property.

However, for simplicity, our messages at the moment are strings. To specify the message to be removed from state, we can use the index of the message in the array.

With that in mind, the `DELETE_MESSAGE` action object has this shape:

```
{  
  type: 'DELETE_MESSAGE',  
  index: 2, // <- index of whichever message is being removed here  
}
```

Building the `reducer()`

Initializing state

Right now, we initialize state at the top of `createStore()` to `0`:

```
1 function createStore(reducer) {  
2   let state = 0;  
3   // ...  
4 }
```

While this works fine for our counter app, we want the initial state for our messaging app to look like this:

```
{          // an object  
  messages: [], // no messages  
}
```

We'll need to modify `createStore()` so that it will work for this and any representation of state.

We'll have `createStore()` accept a second argument, `initialState`. The function will initialize state to this value.

Inside of `app.js`, edit `createStore()` now:

redux/chat_simple/public/app.js

```

1 function createStore(reducer, initialState) {
2   let state = initialState;
3   // ...

```

We'll pass in `initialState` when we initialize the store a bit later.

Handling the ADD_MESSAGE action

Begin writing the `reducer()` inside of `app.js`, below `createStore()`:

redux/chat_simple/public/app.js

```

16 function reducer(state, action) {
17   if (action.type === 'ADD_MESSAGE') {
18     return {
19       messages: state.messages.concat(action.message),
20     };
21   } else {
22     return state;
23   }
24 }

```

When our reducer receives the `ADD_MESSAGE` action we want to append the new message to the end of the `messages` array in `state`. Otherwise, we return `state` unmodified.

We might be tempted to use `Array's push` to append the new message to `messages`:

```
// tempting, but flawed
if (action.type === 'ADD_MESSAGE') {
  state.messages.push(action.message);
  return state;
}
```

This would yield the desired result: `state.messages` would contain the new message.

However, this violates a principle of Redux reducers, our last key idea of Redux from our list above: **reducers must be pure functions**.

A **pure function**⁶⁶ is one that:

⁶⁶https://en.wikipedia.org/wiki/Pure_function

- Will always return the same value given the same set of arguments.
- Does not alter the “world” around it in any way. This includes mutating variables external to the function or altering an entry in a database.

Because state is a variable external to `reducer()` and passed in as an argument, `reducer()` does not “own” this variable. Modifying state, as we do with `push` above, would make `reducer()` impure.

When writing Redux reducers, our pure reducer functions will always return a new array or object in the event that state has to be modified. Reducers should treat the state object as **read-only**.



Reducers should treat the state object as **read-only**.

Because we do not want to modify the state argument, `ADD_MESSAGE` should instead create a *new* state object with a *new* `messages` array. The new array should have the desired message appended to it.

Look again at how we produce the next state in `ADD_MESSAGE`:

`redux/chat_simple/public/app.js`

```
18  return {  
19    messages: state.messages.concat(action.message),  
20  };
```

Crucially, `Array`'s `concat` does *not* modify the original array. Instead, it creates a *new* copy of the array that includes `action.message` appended to it.



In general, writing functions purely can help reduce surprises or enigmatic bugs in your code. We explore the specific motivations and benefits of Redux's insistence on pure reducer functions in a subsequent chapter.

Try it out

We'll write our testing code at the bottom of `app.js`, below the definition for `reducer()`.

Our `createStore()` function now accepts `initialState` as an argument. Let's first define this variable:

redux/chat_simple/public/app.js

```
26 const initialState = { messages: [] };
```

And then initialize the store:

redux/chat_simple/public/app.js

```
28 const store = createStore(reducer, initialState);
```

Let's add code to dispatch add message actions to the store. This time, we'll save each state "version" in two variables, stateV1 and stateV2. We'll print out the two versions of our state at the end:

redux/chat_simple/public/app.js

```
30 const addMessageAction1 = {
31   type: 'ADD_MESSAGE',
32   message: 'How does it look, Neil?',
33 };
34
35 store.dispatch(addMessageAction1);
36 const stateV1 = store.getState();
37
38 const addMessageAction2 = {
39   type: 'ADD_MESSAGE',
40   message: 'Looking good.',
41 };
42
43 store.dispatch(addMessageAction2);
44 const stateV2 = store.getState();
45
46 console.log('State v1:');
47 console.log(stateV1);
48 console.log('State v2:');
49 console.log(stateV2);
```

Running app.js with babel-node:

```
./node_modules/.bin/babel-node public/app.js
```

Yields the following result:

```

State v1:
{ messages: [ 'How does it look, Neil?' ] }
State v2:
{ messages: [ 'How does it look, Neil?', 'Looking good.' ] }

```

Importantly, the `state` object was not modified between dispatches. We saved the first version of the state as the variable `stateV1`. Although this object was passed into `reducer()`, `reducer()` did not modify it. Instead, it created a new object with our second message appended. This new object was returned and set to the variable `stateV2`.

Handling the DELETE_MESSAGE action

As discussed, the `DELETE_MESSAGE` action object has the following shape:

```
{
  type: 'DELETE_MESSAGE',
  index: 2, // <- index of whichever message is being removed here
}
```

To support this action, we need to add a new `else if` statement to handle an action with a type of '`DELETE_MESSAGE`'. When the reducer receives this action, it should return an object with a `messages` array that contains every message **except** the one specified by the action's `index` property.

The most succinct solution might seem to be Array's `splice` method⁶⁷. The first argument for `splice` is the starting index of the element(s) you want to remove. The second is the number of elements to remove:

```
// tempting, but flawed
case 'DELETE_MESSAGE':
  state.messages.splice(action.index, 1);
  return state;
```

However, like `push`, `splice` modifies the original array. This would make `reducer()` impure. Again, we cannot modify `state` — we must treat it as read-only.

Instead, we can create a new object as we did in `ADD_MESSAGE`. That new object will contain a new `messages` array that includes all of the elements in `state.messages` *except* the one being removed.

To do this in JavaScript, we can create a new array that contains:

- All of the elements from `0` to `action.index`
- All of the elements from `action.index + 1` to the end of the array

We use Array's `slice` to grab the desired “chunks” of the array:

⁶⁷https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/splice

redux/chat_simple/public/app.js

```
16 function reducer(state, action) {
17   if (action.type === 'ADD_MESSAGE') {
18     return {
19       messages: state.messages.concat(action.message),
20     };
21   } else if (action.type === 'DELETE_MESSAGE') {
22     return {
23       messages: [
24         ...state.messages.slice(0, action.index),
25         ...state.messages.slice(
26           action.index + 1, state.messages.length
27         ),
28       ],
29     };
30   } else {
31     return state;
32   }
}
```

Importantly, `slice` does not modify the original array. Instead, it returns a new array with the elements in the range you specify. We create a new array that combines two ranges: up to and excluding `action.index` and every element after `action.index`.

ES6: The spread operator (...)

The syntax for creating an array like this is new in ES6. The ellipsis `...` operator will *expand* the array that follows into the parent array.

You can read more about the [spread operator here^a](#), but here is a basic example:

```
1 const a = [ 1, 2, 3 ];
2 const b = [ 4, 5, 6 ];
3 const c = [ ...a, ...b, 7, 8, 9 ];
4
5 console.log(c); // -> [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
```

Notice how this is different than if we wrote:

```
1 const d = [ a, b, 7, 8, 9 ];
2
3 console.log(d); // -> [ [ 1, 2, 3 ], [ 4, 5, 6 ], 7, 8, 9 ]
```

The spread operator enables us to succinctly construct new arrays as a composite of existing arrays. Because of this feature, we'll be using this operator often to keep our reducer functions pure.

^ahttps://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_operator

Try it out

At the very bottom of `app.js`, we'll add on to our code that tested the `ADD_MESSAGE` action. Write the following below the last `console.log()` statement in the file:

`redux/chat_simple/public/app.js`

```
60 const deleteMessageAction = {
61   type: 'DELETE_MESSAGE',
62   index: 0,
63 };
64
65 store.dispatch(deleteMessageAction);
66 const stateV3 = store.getState();
67
68 console.log('State v3:');
69 console.log(stateV3);
```

By the second version of the state, we've added two messages to the state. We then dispatch a `DELETE_MESSAGE` action, specifying the message at index 0.

Run the file with `babel-node`:

```
./node_modules/.bin/babel-node public/app.js
```

As expected, in the third version of the state the first message has been removed:

```
State v1:
{ messages: [ 'How does it look, Neil?' ] }
State v2:
{ messages: [ 'How does it look, Neil?', 'Looking good.' ] }
State v3:
{ messages: [ 'Looking good.' ] }
```

Subscribing to the store

Our store so far provides methods for the view to dispatch actions and to read the current version of the state.

One important feature is missing before we can connect the store to React, however. While the view can read the state at any time with `getState()`, **the view needs to know when the state has changed**. Constantly polling the store with `getState()` is inefficient.

In our previous apps, when we wanted to modify the state we called `setState()`. Importantly, `setState()` triggers a `render()` call on the component.

Now, state is being modified outside of React and inside of the store. Our views are unaware of when it changes. If we're going to keep our views up to date with the most current state in the store, then our views should receive a notification whenever the state changes.

Our store will use the **observer pattern** to allow the views to immediately update when the state changes. The views will register a callback function that they would like to be invoked when the state changes. The store will keep a list of all of these callback functions. When the state changes, the store will invoke each function, "notifying" the listeners of the change.

The best way to illustrate this pattern is to implement it.

Inside `createStore()`, we will:

1. Define an array called `listeners`
2. Add a `subscribe()` method which adds a new listener to `listeners`
3. Call each listener function when the state is changed

1. Define an array called `listeners`

We declare `listeners` at the top of `createStore()`:

`redux/chat_simple/public/app.js`

```
1 function createStore(reducer, initialState) {
2   let state = initialState;
3   const listeners = [];
4   // ...
```

2. Add a `subscribe()` method which adds a new listener to `listeners`

Next, below the declaration of `listeners`, let's add `subscribe()`:

redux/chat_simple/public/app.js

```
4 // ...
5 const subscribe = (listener) => (
6   listeners.push(listener)
7 );
8 // ...
```

The `listener` argument of `subscribe()` is a function, the function that the view would like invoked whenever the state changes. We add this function to the `listeners` array.

To make `subscribe()` accessible, we need to expose `subscribe` by adding it to the store object returned by `createStore()`:

redux/chat_simple/public/app.js

```
14 // ...
15 return {
16   subscribe,
17   getState,
18   dispatch,
19 };
```

3. Call each listener function when the state is changed

Whenever the state changes, we need to invoke all the functions kept in `listeners`. The state may change whenever we dispatch actions. As such, we'll add the invocation logic to `dispatch()`:

redux/chat_simple/public/app.js

```
10 // ...
11 const dispatch = (action) => {
12   state = reducer(state, action);
13   listeners.forEach(l => l());
14 };
15 // ...
```

Note that there are no arguments passed to the listeners. This callback is solely a notification that the state changed.

createStore() in full

Here's our `createStore()` function, in full:

redux/chat_simple/public/app.js

```
1 function createStore(reducer, initialState) {
2   let state = initialState;
3   const listeners = [];
4
5   const subscribe = (listener) => (
6     listeners.push(listener)
7   );
8
9   const getState = () => (state);
10
11  const dispatch = (action) => {
12    state = reducer(state, action);
13    listeners.forEach(l => l());
14  };
15
16  return {
17    subscribe,
18    getState,
19    dispatch,
20  };
21 }
```

Stripped of comments, warnings, and sanity checks, the Redux library's `createStore()` looks and behaves quite like our function.

Try it out

With `subscribe()` in place, our store is complete. Let's test everything out.

In `app.js`, clear out all the previous testing code below the line where we initialize the store:

redux/chat_simple/public/app.js

```
44 const store = createStore(reducer, initialState);
```

We'll dispatch add and delete message actions like before. Except, now we'll use `subscribe()` to register a function that will perform a `console.log()` every time the state changes.

Our listener prints the current state to the console:

redux/chat_simple/public/app.js

```
46 const listener = () => (
47   console.log(store.getState())
48 );
```

Next, let's subscribe this listener:

redux/chat_simple/public/app.js

```
50 store.subscribe(listener);
```

Now, we can dispatch our actions. After every `dispatch()` call, the listening function we passed to `subscribe()` will be called, writing to the console:

redux/chat_simple/public/app.js

```
52 const addMessageAction1 = {
53   type: 'ADD_MESSAGE',
54   message: 'How do you read?',
55 };
56 store.dispatch(addMessageAction1);
57   // -> `listener()` is called
58
59 const addMessageAction2 = {
60   type: 'ADD_MESSAGE',
61   message: 'I read you loud and clear, Houston.',
62 };
63 store.dispatch(addMessageAction2);
64   // -> `listener()` is called
65
66 const deleteMessageAction = {
67   type: 'DELETE_MESSAGE',
68   index: 0,
69 };
70 store.dispatch(deleteMessageAction);
71   // -> `listener()` is called
```

Save `app.js` and run it with `babel-node`:

```
$ ./node_modules/.bin/babel-node public/app.js
```

And note the output:

Current state:

```
{ messages: [ 'How do you read?' ] }
```

Current state:

```
{ messages: [ 'How do you read?', 'I read you loud and clear, Houston.' ] }
```

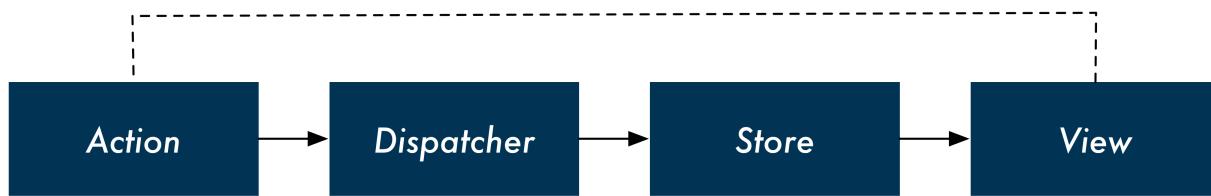
Current state:

```
{ messages: [ 'I read you loud and clear, Houston.' ] }
```

With our store feature complete, we're prepared to connect our Redux store to some React views and see a complete, working Redux pipeline.

Connecting Redux to React

Revisiting the Flux diagram from earlier, we can now explore the specifics behind how Redux and React work together to fulfill this design pattern:



Using `store.getState()`

React is no longer managing state. Redux is. Therefore, top-level React components will use `store.getState()` as opposed to `this.state` to drive their `render()` functions. The state provided by Redux will trickle down from there.

For instance, if we want to render our state's `messages`, we fetch them from our Redux store:

```
// An example top-level component
const App = React.createClass({
  // ...
  render: function () {
    const messages = store.getState().messages;
    // ...
  }
});
```

Using `store.subscribe()`

When React manages state, we call `setState()` to modify `this.state`. `setState()` will trigger a re-render after the state is modified.

When Redux is managing state, we use `subscribe()` inside the top-level React component to setup a listening function that initiates the re-render.

We can subscribe out component inside of `componentDidMount`. The listening function that we pass to `subscribe()` will call `this.forceUpdate()`, triggering the component (`this`) to re-render.

As an example, subscribing a React component looks like this:

```
// An example top-level component
const App = React.createClass({
  // ...
  componentDidMount: function () {
    store.subscribe(() => this.forceUpdate());
  },
  // ...
});
```

Using `store.dispatch()`

Lower-level components will dispatch actions in response to events that should modify state. For instance, a React component might dispatch an action to the store whenever a delete button is clicked:

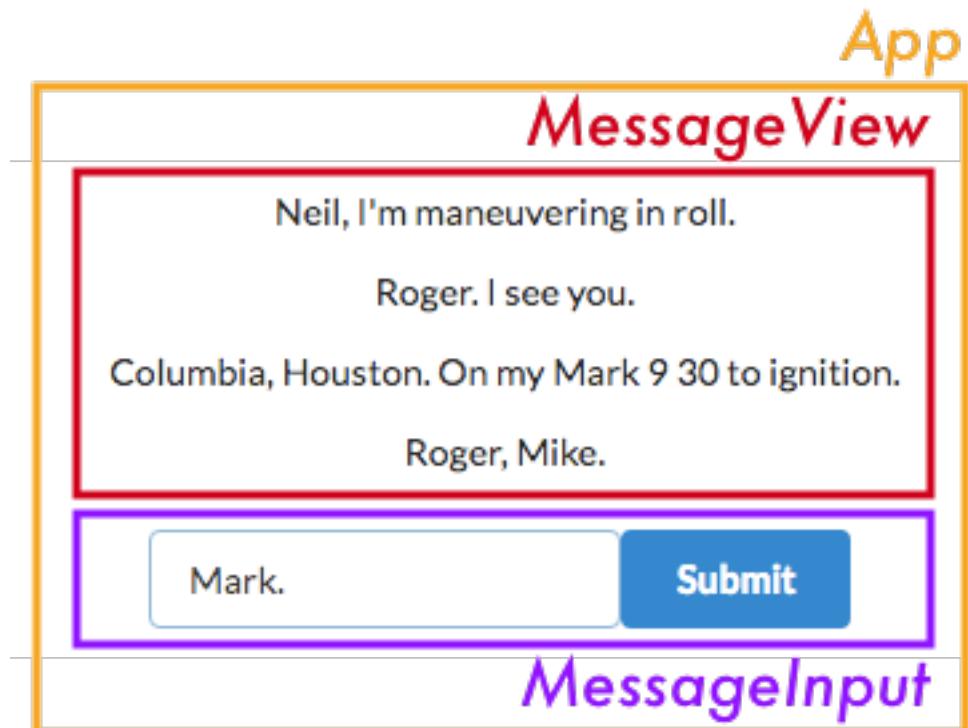
```
// An example leaf component
const Message = React.createClass({
  handleDeleteClick: function () {
    store.dispatch({
      type: 'DELETE_MESSAGE',
      index: this.props.index,
    });
  },
  // ...
});
```

This `dispatch()` call will modify the state. `dispatch()` will then invoke the listener, which we registered with `subscribe()`. This forces the App component to re-render. When `render()` is invoked, the App component reads from the store again with `getState()`. App then passes the latest version of the state down to its child components.

This cycle repeats every time React dispatches an action.

The app's components

The chat app has three React components:



- App (**orange**): The top-level container
- MessageView (**red**): The list of messages
- MessageInput (**purple**): The input to add new messages

As we saw, the input box enables adding messages. Clicking on a message deletes it.

MessageView will render the state's messages. It will also dispatch `DELETE_MESSAGE` actions every time the user clicks an individual message.

MessageInput will not use state to render. However, it will dispatch `ADD_MESSAGE` actions every time the user submits a new message.



We could have broken MessageView into MessageList and Message. This would follow the pattern from previous apps. However, each message is simple enough at the moment that this is not necessary.

Preparing `app.js`

For this project, the store logic and React components will all be inside `public/app.js`. We will be able to reference `store` directly in each of our components.

In more complex applications, the store will likely be located in a different file than React components. In a later chapter, we'll explore other ways to have React components communicate with a Redux store object.

Clear out all the testing code below the declaration of `store` at the end of `public/app.js`:

`redux/chat_simple/public/app.js`

44 `const store = createStore(reducer, initialState);`

The App component

App is the top-level React component in our app. App will be the component that reads from the store. We'll need it to subscribe to our Redux store.

Subscribing to changes

Like we talked about above, we will `subscribe()` inside of `componentDidMount`. The callback function we give to `subscribe()` calls `this.forceUpdate()`. This will cause the App component to re-render every time the state changes:

redux/chat_simple/public/app.js

```
46 const App = React.createClass({
47   componentDidMount: function () {
48     store.subscribe(() => this.forceUpdate());
49   },

```

Rendering the view

In render, we'll first use `getState()` to read messages from the store. We'll then render the two children, `MessageView` and `MessageInput`. Only `MessageView` needs the list of messages:

redux/chat_simple/public/app.js

```
50   render: function () {
51     const messages = store.getState().messages;
52
53     return (
54       <div className='ui segment'>
55         <MessageView messages={messages} />
56         <MessageInput />
57       </div>
58     );
59   },
60 });

```

We have our downward data pipeline, from the store through `App` down to `MessageView`. But what about the inverse direction? We want `MessageView` to have the ability to delete messages and `MessageInput` to be able to add them.

When React is managing state, we pass down functions as props from state-managing components to children. This enables children to propagate events up to the parent who modifies the state.

Now, we have a `store` object that we can dispatch actions to. While we could still concentrate all communication with the store inside of `App`, let's explore allowing our child components to dispatch actions to the store directly.



As in previous projects, the two `div` tags (and their `className` attributes) are provided for styling.

The `App` component in full:

redux/chat_simple/public/app.js

```

46 const App = React.createClass({
47   componentDidMount: function () {
48     store.subscribe(() => this.forceUpdate());
49   },
50   render: function () {
51     const messages = store.getState().messages;
52
53     return (
54       <div className='ui segment'>
55         <MessageView messages={messages} />
56         <MessageInput />
57       </div>
58     );
59   },
60 });

```

The MessageInput component

MessageInput will have a single input field and a submit button. When the user clicks the submit button, the component should dispatch an ADD_MESSAGE action.

We'll begin by defining handleSubmit(), the function that will call dispatch(). To grab the input value, we'll use this.refs, anticipating we'll use the ref messageInput on an input tag in the view:

redux/chat_simple/public/app.js

```

62 const MessageInput = React.createClass({
63   handleSubmit: function () {
64     store.dispatch({
65       type: 'ADD_MESSAGE',
66       message: this.refs.messageInput.value,
67     });
68     this.refs.messageInput.value = '';
69   },

```

render will contain an input and a button wrapped in a div. button will have its onClick attribute set to this.handleSubmit:

redux/chat_simple/public/app.js

```
70  render: function () {
71    return (
72      <div className='ui input'>
73        <input
74          ref='messageInput'
75          type='text'
76        >
77        </input>
78        <button
79          onClick={this.handleSubmit}
80          className='ui primary button'
81          type='submit'
82        >
83          Submit
84        </button>
85      </div>
86    );
87  },
88});
```



We could have also avoided defining `handleClick()` and instead just wrapped a `dispatch()` call inside of an anonymous function, defined in-line:

```
1  // ...
2  key={index}
3  onClick={() => (
4    store.dispatch({
5      type: 'DELETE_MESSAGE',
6      index: index,
7    })
8  )}
9  >
10 {message}
11 // ...
```

The choice is a matter of your stylistic preference.

The `MessageInput` component in full:

redux/chat_simple/public/app.js

```
62 const MessageInput = React.createClass({
63   handleSubmit: function () {
64     store.dispatch({
65       type: 'ADD_MESSAGE',
66       message: this.refs.messageInput.value,
67     });
68     this.refs.messageInput.value = '';
69   },
70   render: function () {
71     return (
72       <div className='ui input'>
73         <input
74           ref='messageInput'
75           type='text'
76           >
77         </input>
78         <button
79           onClick={this.handleSubmit}
80           className='ui primary button'
81           type='submit'
82           >
83           Submit
84         </button>
85       </div>
86     );
87   },
88 });
```

The MessageView component

The MessageView component's messages prop is an array of strings. MessageView will render these messages as a list. Furthermore, whenever the user clicks a message we want to dispatch a DELETE_MESSAGE action.

Let's begin by defining the component and its function handleClick(). handleClick() will be the function that calls dispatch(). handleClick() accepts one argument, index, which it uses in the action object it dispatches:

redux/chat_simple/public/app.js

```
90 const MessageView = React.createClass({
91   handleClick: function (index) {
92     store.dispatch({
93       type: 'DELETE_MESSAGE',
94       index: index,
95     });
96   },
}
```

The render function will use map to create the list of messages to render. We want each individual message to be wrapped inside of a div:

redux/chat_simple/public/app.js

```
97   render: function () {
98     const messages = this.props.messages.map((message, index) => (
99       <div
100         className='comment'
101         key={index}
102         onClick={() => this.handleClick(index)}
103       >
104         {message}
105       </div>
106     )));
}
```

On this div we set the onClick attribute. We want this to call a function that calls handleClick(), passing in the index of the target message.

Finally, we return messages wrapped inside of a div:

redux/chat_simple/public/app.js

```
107   return (
108     <div className='ui comments'>
109       {messages}
110     </div>
111   );
112 },
113});
```

The MessagesView component in full:

redux/chat_simple/public/app.js

```
90 const MessageView = React.createClass({
91   handleClick: function (index) {
92     store.dispatch({
93       type: 'DELETE_MESSAGE',
94       index: index,
95     });
96   },
97   render: function () {
98     const messages = this.props.messages.map((message, index) => (
99       <div
100         className='comment'
101         key={index}
102         onClick={() => this.handleClick(index)}
103       >
104         {message}
105       </div>
106     ));
107     return (
108       <div className='ui comments'>
109         {messages}
110       </div>
111     );
112   },
113 });
```

ReactDOM.render()

The last piece we need to write is the call to `ReactDOM.render()`. We pass it *what* to render and *where*:

redux/chat_simple/public/app.js

```
115 ReactDOM.render(
116   <App />,
117   document.getElementById('content')
118 );
```

Try it out

Save `app.js`. In your terminal, from the root of the project folder, boot the server:

```
$ npm run server
```

Add a few messages, then click on them to see them instantly disappear.

Next up

Redux is a powerful way to manage state in your app. By using a few simple ideas you can get an understandable data architecture that scales well to large apps.

Admittedly, in the current state of our app it's difficult to see how Redux provides any advantage over managing state in React. Indeed, as stated in the introduction of this chapter, using React for state management is a preferable choice for a wide variety of applications.

However, as we'll see as we scale up the complexity of our messaging app, Redux is advantageous as an app's interactivity and state management grows more complicated. This is because:

1. All of the data is in a central data structure
2. Data changes are also centralized
3. The actions that views emit are decoupled from the state mutations that occur
4. One-way data flow makes it easy to trace how changes flow through the system

With the core ideas of Redux behind us, we're prepared to significantly increase the functionality of the messaging app. In doing so, we'll explore solutions for a variety of real-world challenges.

As we evolve the messaging app, we'll cover:

- How to use the Redux library
- How to use the React-Redux library
- How to deal with more complicated state
- How to split up our reducers (and recombine them)
- How to re-organize our React components

React and Redux pair wonderfully, and we'll see first-hand how well they scale to handle our escalating requirements.

Intermediate Redux

In the last chapter, we learned about a specific Flux implementation, Redux. By building our own Redux store from scratch and integrating the store with React components, we got a feel for how data flows through a Redux-powered React app.

In this chapter, we build on these concepts by adding additional features to our chat application. Our chat app will begin to look like a real-world messaging interface.

In the process, we'll explore strategies for handling more complex state management. We'll also use a couple of functions directly from the `redux` library.

Preparation

Inside of the code download that came with the book, navigate to `redux/chat_intermediate`:

```
$ cd redux/chat_intermediate
```

If you run `ls -1`, you'll see a familiar setup:

```
$ ls -1
README.md
package.json
public
```

Running `ls -1 public/` also yields a familiar set of files:

```
$ ls -1 public/
app-complete.js
app.js
index.html
style.css
vendor
```

`app-complete.js` contains the finished version of the chat app that we write in this chapter. `app.js` is where we'll be working. It contains the app as we left it in the previous chapter.

As usual, run `npm install` to install all of the dependencies for the project:

```
$ npm install
```

And then execute `npm run server` to boot the server:

```
$ npm run server
```

View the app by visiting <http://localhost:3000>⁶⁸ in your browser.

In this iteration of the chat app, our app has threads. Each message belongs to a particular thread with another user. We can switch between threads using the tabs at the top.

As in the last iteration, note that we can add messages with the text field at the bottom as well as delete messages by clicking on them.

Remove `app-complete.js` from `public/index.html` before beginning:

```
33 <script type='text/babel' src='./app.js'></script>
34 <!-- Delete the line below to get started. -->
35 <script type='text/babel' src='./app-complete.js'></script>
```

Using `createStore()` from the redux library

In the last chapter, we implemented our own version of `createStore()`. At the top of `public/app.js`, you'll find the `createStore()` function just as we left it. The store object that this function creates has three methods: `getState()`, `dispatch()`, and `subscribe()`.

As we noted in the last chapter, our `createStore()` function is very similar to that which ships with the `redux` library. Let's remove our implementation and use the one from `redux`.

In `public/index.html`, inside of the `head` tags, we already include the `redux.js` library:

`redux/chat_intermediate/public/index.html`

```
20   <!-- Include the Redux library -->
21   <script src='vendor/redux.js'></script>
```

This means that the variable `Redux` will be available for use inside of `public/app.js`.

Go ahead and remove `createStore()` from `app.js`.

Then, down where we instantiate the store, use the `createStore()` method on `Redux`:

⁶⁸<http://localhost:3000>

redux/chat_intermediate/public/app.js

```
22 const store = Redux.createStore(reducer, initialState);
```

Try it out

To ensure everything is working properly, ensure the server is running:

```
$ npm run server
```

And then check out the app on <http://localhost:3000>⁶⁹.

Behavior for the app will be the same as we left it in the previous chapter. We can add new messages and click on them to delete.

There are a couple subtle behavioral differences between our `createStore()` and the one that ships with the Redux library. Our app has yet to touch on them. We'll address these differences when they come up.

Representing messages as objects in state

Our state up to this point has been simple. State has been an object, with a `messages` property. Each message has been a string:

```
// Example of our state object so far
{
  messages: [
    'Roger. Eagle is undocked',
    'Looking good.',
  ],
}
```

To bring our app closer to a real-world chat app, each message will need to carry more data. For example, we might want each message to specify when it was sent or who sent it. To support this, we can use an object to represent each message as opposed to a string.

For now, we'll add two properties to each message, `timestamp` and `id`:

⁶⁹<http://localhost:3000>

```
// Example of our new state object
{
  messages: [
    // An example message
    // messages are now objects
    {
      text: 'Roger. Eagle is undocked',
      timestamp: '1461974250213',
      id: '9da98285-4178',
    },
    // ...
  ]
}
```



The `Date.now()` function in JavaScript returns a number representing the number of milliseconds since January 1, 1970 00:00 UTC. This is called “Epoch” or “Unix” time. We’re using this representation for the `timestamp` property above.

You can use a JavaScript library like [Moment.js⁷⁰](#) to render more human-friendly timestamps.

In order to support messages that are objects, we’ll need to tweak our reducer as well as our React components.

Updating ADD_MESSAGE

The reducer function we wrote in the last chapter handles two actions, `ADD_MESSAGE` and `DELETE_MESSAGE`. Let’s start by updating the `ADD_MESSAGE` action handler.

As you recall, the `ADD_MESSAGE` action currently contains a `message` property:

```
{
  type: 'ADD_MESSAGE',
  message: 'Looking good.',
}
```

`reducer()` receives this action and returns a new object with a `messages` property. `messages` is set to a new array that contains the previous `state.messages` with the new message appended to it:

⁷⁰<http://momentjs.com/>

redux/chat_intermediate/public/app.js

```
2  if (action.type === 'ADD_MESSAGE') {
3      return {
4          messages: state.messages.concat(action.message),
5      };
}
```

Let's tweak our ADD_MESSAGE action so that it uses the property name `text` instead of `message`:

```
// Example of what our new ADD_MESSAGE will look like
{
    type: 'ADD_MESSAGE',
    text: 'Looking good.',
}
```

`text` matches the property name that we'll be using for the message object.

Next, let's modify our reducer's ADD_MESSAGE handler so that it uses message objects as opposed to string literals.

In `public/app.js`, let's modify ADD_MESSAGE so that it creates a new object to represent the message. It will use `action.text` for the `text` property and then generate a `timestamp` and an `id`:

redux/chat_intermediate/public/app.js

```
2  if (action.type === 'ADD_MESSAGE') {
3      const newMessage = {
4          text: action.text,
5          timestamp: Date.now(),
6          id: uuid.v4(),
7      };
}
```



`uuid.v4()` is the same UUID generator function from the same `uuid` library that we used previously in the timers project. We include this library in `index.html`.



`Date.now()` is part of the JavaScript standard library. It returns the current time in the Unix time format, in milliseconds.

We'll use `concat` again. This time, we'll use `concat` to return a new array that contains `state.messages` and our `newMessage` object:

redux/chat_intermediate/public/app.js

```
8  return {
9    messages: state.messages.concat(newMessage),
10   };
```

Our modified ADD_MESSAGE handler in full:

redux/chat_intermediate/public/app.js

```
2  if (action.type === 'ADD_MESSAGE') {
3    const newMessage = {
4      text: action.text,
5      timestamp: Date.now(),
6      id: uuid.v4(),
7    };
8    return {
9      messages: state.messages.concat(newMessage),
10     };
```

Updating DELETE_MESSAGE

The DELETE_MESSAGE action up until now contained an index, the index of the message in the state.messages array to be deleted:

```
{
  type: 'DELETE_MESSAGE',
  index: 5,
}
```

Now that all of our messages have a unique id, we can use that:

```
// Example of what our new DELETE_MESSAGE will look like
{
  type: 'DELETE_MESSAGE',
  id: '9da98285-4178',
}
```

We'll see the advantage of using an id as opposed to index for this action later in this chapter.

To remove the message from state.messages we'll:

1. Find the index of the message in the array that has the matching id
2. Use `slice()` to build a new array that includes every entry in `state.messages` *except* the one we want to remove

Modify the `DELETE_MESSAGE` handler now:

`redux/chat_intermediate/public/app.js`

```
11 } else if (action.type === 'DELETE_MESSAGE') {
12   const index = state.messages.findIndex(
13     (m) => m.id === action.id
14   );
15   return {
16     messages: [
17       ...state.messages.slice(0, index),
18       ...state.messages.slice(
19         index + 1, state.messages.length
20       ),
21     ],
22   };

```

We use the spread operator along with `slice()` again to grab two chunks:

- All of the entries in `state.messages` up to `index`
- All of the entries in `state.messages` after `index`

With these changes in place, our reducers are ready to handle our new message objects. We'll update our React components next. We need to update both the actions they emit as well as how they render messages.

Updating the React components

`MessageInput` dispatches an `ADD_MESSAGE` action whenever the user clicks its submit button. We'll need to modify this component so that it uses the property name `text` as opposed to `message` for the action:

redux/chat_intermediate/public/app.js

```
48 const MessageInput = React.createClass({
49   handleSubmit: function () {
50     store.dispatch({
51       type: 'ADD_MESSAGE',
52       text: this.refs.messageInput.value,
53     });
54     this.refs.messageInput.value = '';
55   },
}
```

MessageView dispatches a `DELETE_MESSAGE` action whenever the user clicks on a message. We need to tweak the action it dispatches so that it uses the property `id` as opposed to `index`:

redux/chat_intermediate/public/app.js

```
76 const MessageView = React.createClass({
77   handleClick: function (id) {
78     store.dispatch({
79       type: 'DELETE_MESSAGE',
80       id: id,
81     });
82   },
}
```

Then, we need to change the `render()` function for `MessageView`. We'll modify the HTML for each message so that it also includes the `timestamp` property. To render the text of the message, we call `message.text`:

redux/chat_intermediate/public/app.js

```
83   render: function () {
84     const messages = this.props.messages.map((message, index) => (
85       <div
86         className='comment'
87         key={index} // We can still just use the `index` for key
88         onClick={() => this.handleClick(message.id)} // Use `id`
89       >
90         <div className='text'> // Wrap message data in `div`
91           {message.text}
92           <span className='metadata'>@{message.timestamp}</span>
93         </div>
94       </div>
95     )));
}
```

Note that we now pass in `message.id` as opposed to `index` to `this.handleClick()`. We wrap the display logic for each message inside a `div` with class `text`.

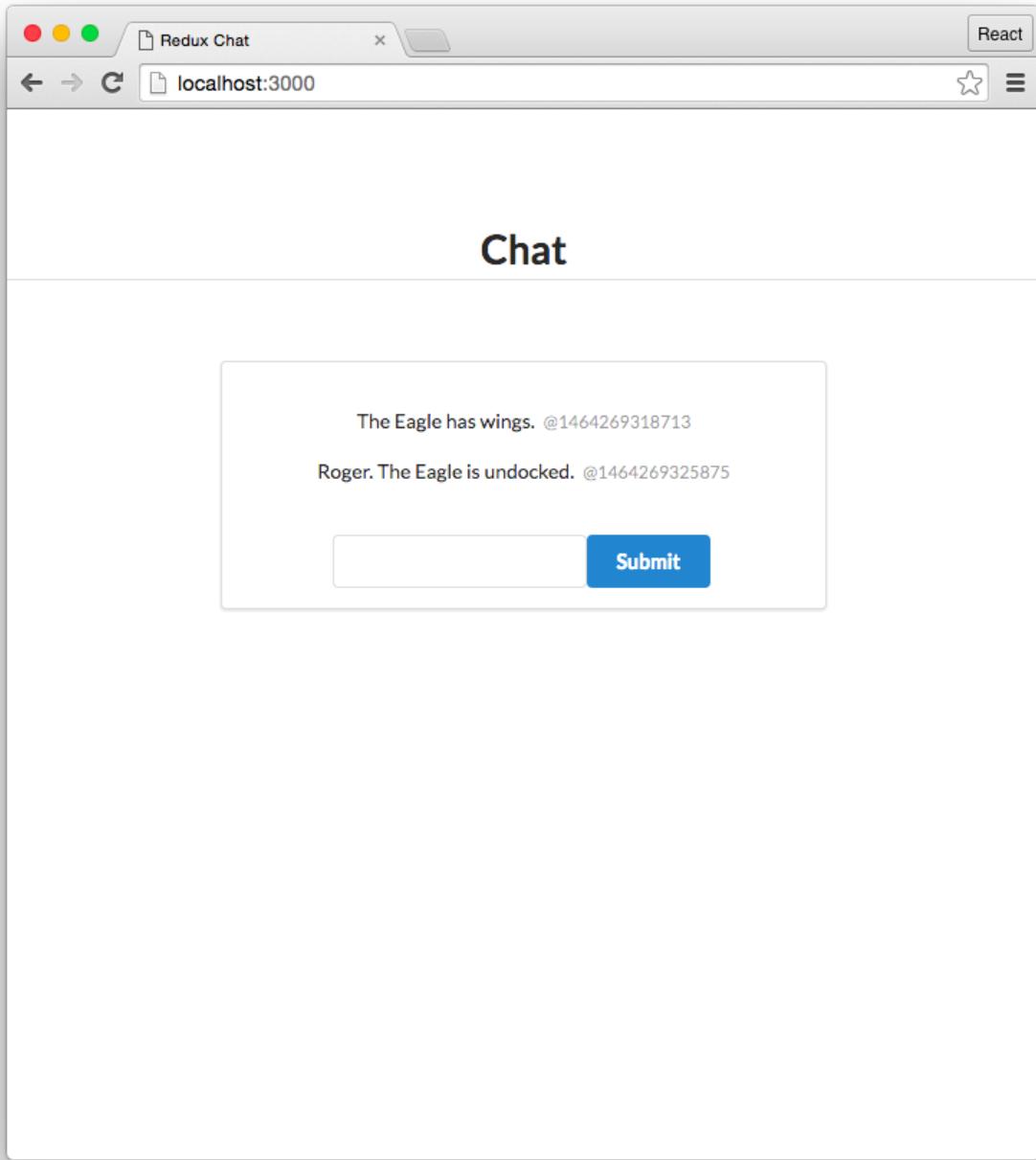
Our reducers and React components are now on the same page. We're using our new representation of both the state and actions.

Save `app.js`. If your server isn't already running, boot it:

```
$ npm run server
```

Navigate to <http://localhost:3000/>⁷¹. When you add messages, a timestamp should appear to the right of each message. You can also delete them as before by clicking on them:

⁷¹<http://localhost:3000/>

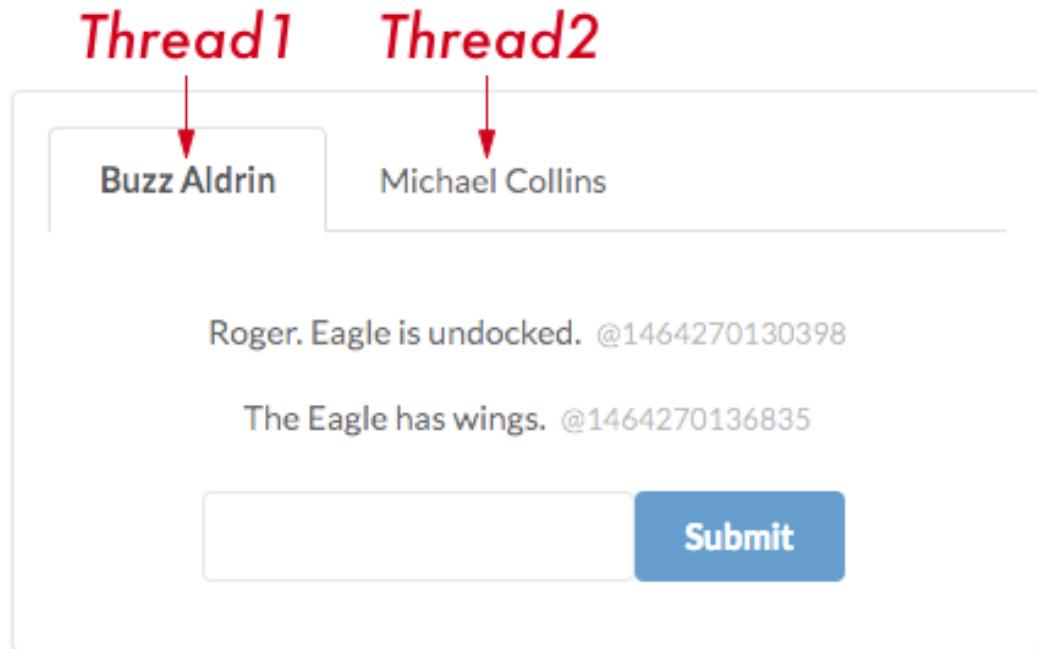


Introducing threads

Our state now uses message objects, which will allow us to carry information about each message in our app (like `timestamp`).

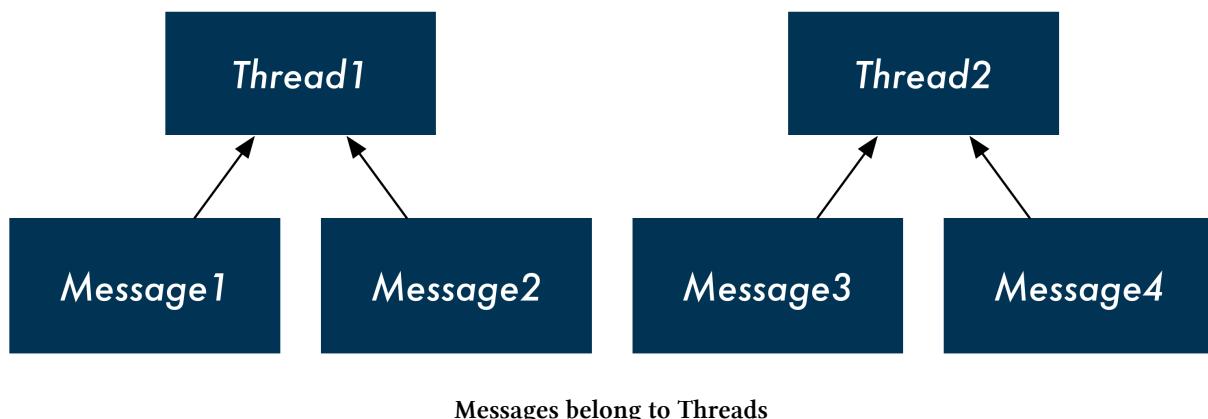
But in order for our app to begin reflecting a real-world chat app, we'll need to introduce another concept: **threads**.

In a chat app, a “thread” is a distinct set of messages. A thread is a conversation between you and one or more other users:



Two threads in the interface

As the completed version of the app demonstrated, our app will use tabs to enable a user to switch between threads. Each message belongs to a single thread:



To support threads, we'll update the shape of our state object. The top-level property will now be `threads`, an array of thread objects. Each thread object will have a `messages` property which will contain the message object we introduced to the system in the previous section:

```
{
  threads: [
    {
      id: 'd7902357-4703', // UUID of the thread
      title: 'Buzz Aldrin', // Who the conversation is with
      messages: [
        {
          id: 'e8596e6b-97cc',
          text: 'Twelve minutes to ignition.',
          timestamp: 1462122634882,
        },
        // ... other messages with Buzz Aldrin
      ]
    },
    // ... other threads (with other users)
  ],
}
```

Supporting threads in `initialState`

To support threads, let's first update our initial state.

At the moment, we're initializing state to an object with a `messages` property:

`redux/chat_intermediate/public/app.js`

28 `const initialState = { messages: [] };`

Now we want our top-level property to be `threads`. We could initialize state to this:

```
{ threads: [] }
```

But that would quickly add a significant amount of complexity to our app. Not only do we need to update our reducers to support our new thread-driven state, but we'd need to also add some way to create new threads.

For our app to reflect a real-world chat app, we'd need the ability to create new threads in the future. But for now, we can take a smaller step by just initializing our state with a hard-coded set of threads.

Modify `initialState` now, initializing it to an object with a `threads` property. We'll have two thread objects in state:

redux/chat_intermediate/public/app.js

```
28 const initialState = {
29   threads: [    // Two threads in state
30     {
31       id: '1-fca2',    // hardcoded pseudo-UUID
32       title: 'Buzz Aldrin',
33       messages: [
34         {    // This thread starts with a single message already
35           text: 'Twelve minutes to ignition.',
36           timestamp: Date.now(),
37           id: uuid.v4(),
38         },
39       ],
40     },
41     {
42       id: '2-be91',
43       title: 'Michael Collins',
44       messages: [],
45     },
46   ],
47 };
```



Because we're hardcoding the `id` for our threads for now, we're using a clipped version of `UUID` for each of them.

In addition to threads and messages, our app should also have one more piece of state. Our front-end displays only one thread at a time. Our view needs to know *which* thread to display. We'll call this the "active" thread. We can add another top-level property to our state object, `activeThreadId`.

Modify `initialState` again, adding this new property. We'll initialize it to our first thread which has an `id` of '`1-fca2`':

redux/chat_intermediate/public/app.js

```
28 const initialState = {
29   activeThreadId: '1-fca2',
30   threads: [
```

We now have an initial state object that our React components can use to render a threaded version of our app. We'll update the components first to render from this new state shape.

Our app will be locked at this initial state though; we won't be able to add or delete any messages or switch between tabs. Once we confirm the views look good, we'll update our actions and our reducer to support our new thread-based chat app.

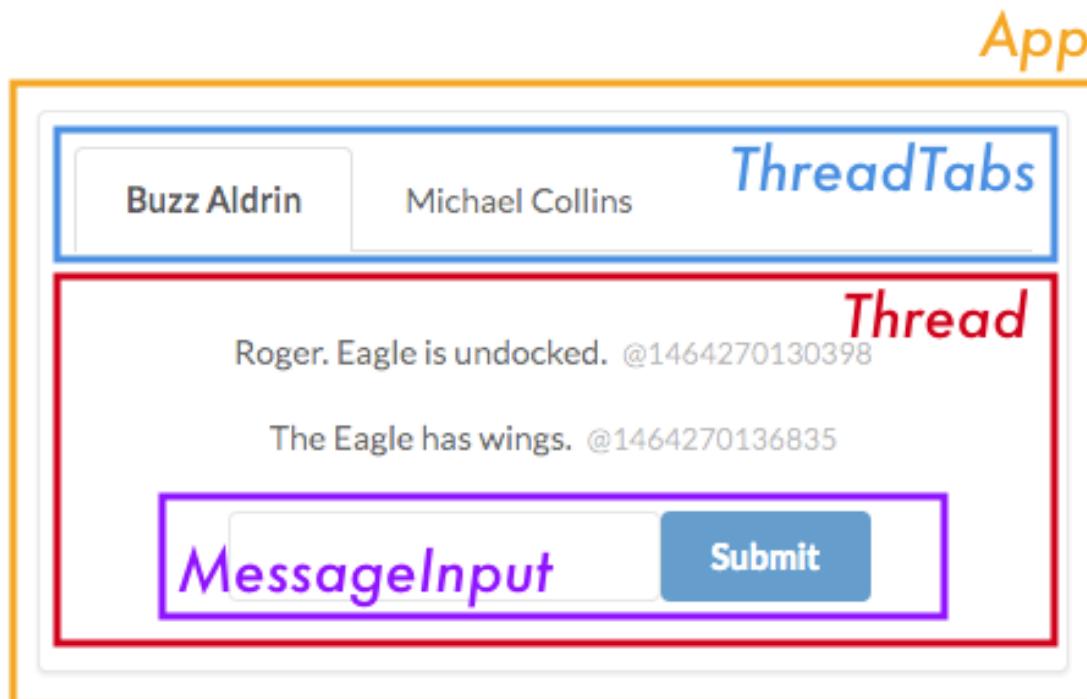


For now, we're initializing the first thread object with a single message already under messages. This will enable us to verify our React components are properly rendering a thread with a message ahead of our reducers supporting our updated ADD_MESSAGE action.

Supporting threads in the React components

To enable switching between threads in the app, the interface will have tabs above the messages view. We'll need to both add new React components and modify existing ones to support this.

Looking at the completed version of this chapter's chat app, we can identify the following components:



- **App (yellow)**: The top-level component.
- **ThreadTabs (blue)**: The tabs widget for switching between threads.
- **Thread (red)**: Displays all the messages in a thread. This component has been called MessageView, but we'll update its name to reflect our new thread-based state paradigm.

- **MessageInput (purple)**: The input to add new messages *to the open thread*. We can have this nested under Thread.

Let's first update our existing components to support our thread-based state. We'll then add our new component, `ThreadTabs`.

Modifying App

App currently subscribes to the store. App uses `getState()` to read the `messages` property and then renders its two children.

We'll have the component use our state's `activeThreadId` property to deduce which thread is active. The component will then pass the active thread to `Thread` (formerly `MessageView`) to render its messages.

First, we read both `activeThreadId` and `threads` from the state. We then use `Array`'s `find` to find the thread object that has an `id` that matches `activeThreadId`:

`redux/chat_intermediate/public/app.js`

```
52 const App = React.createClass({
53   componentDidMount: function () {
54     store.subscribe(() => this.forceUpdate());
55   },
56   render: function () {
57     const state = store.getState();
58     const activeThreadId = state.activeThreadId;
59     const threads = state.threads;
60     const activeThread = threads.find((t) => t.id === activeThreadId);
```

We pass our `Thread` component the `activeThread` for it to render. We're removing `MessageInput` from `App` as it will now be a child of `Thread`:

`redux/chat_intermediate/public/app.js`

```
62   return (
63     <div className='ui segment'>
64       <Thread thread={activeThread} />
65     </div>
66   );
```

The updated `App` component, in full:

redux/chat_intermediate/public/app.js

```
52 const App = React.createClass({
53   componentDidMount: function () {
54     store.subscribe(() => this.forceUpdate());
55   },
56   render: function () {
57     const state = store.getState();
58     const activeThreadId = state.activeThreadId;
59     const threads = state.threads;
60     const activeThread = threads.find((t) => t.id === activeThreadId);
61
62     return (
63       <div className='ui segment'>
64         <Thread thread={activeThread} />
65       </div>
66     );
67   },
68 });
```

Turning MessageView into Thread

Now that messages are collected under a single thread, our front-end is displaying one thread's messages at a given time. We'll rename `MessageView` to `Thread` to reflect this.

`Thread` will render both the list of messages pertaining to the thread it's rendering as well as the `MessageInput` component for adding new messages to that thread.

First, rename the component:

redux/chat_intermediate/public/app.js

```
98 const Thread = React.createClass({
```

Now, to create messages in `render()`, we'll use `this.props.thread.messages`:

redux/chat_intermediate/public/app.js

```
105   render: function () {
106     const messages = this.props.thread.messages.map((message, index) => (
```

Last, we'll add `MessageInput` as a child of `Thread`. While we'll eventually need to update `MessageInput` to work properly with our new threaded state paradigm, we'll hold off on that update for now:

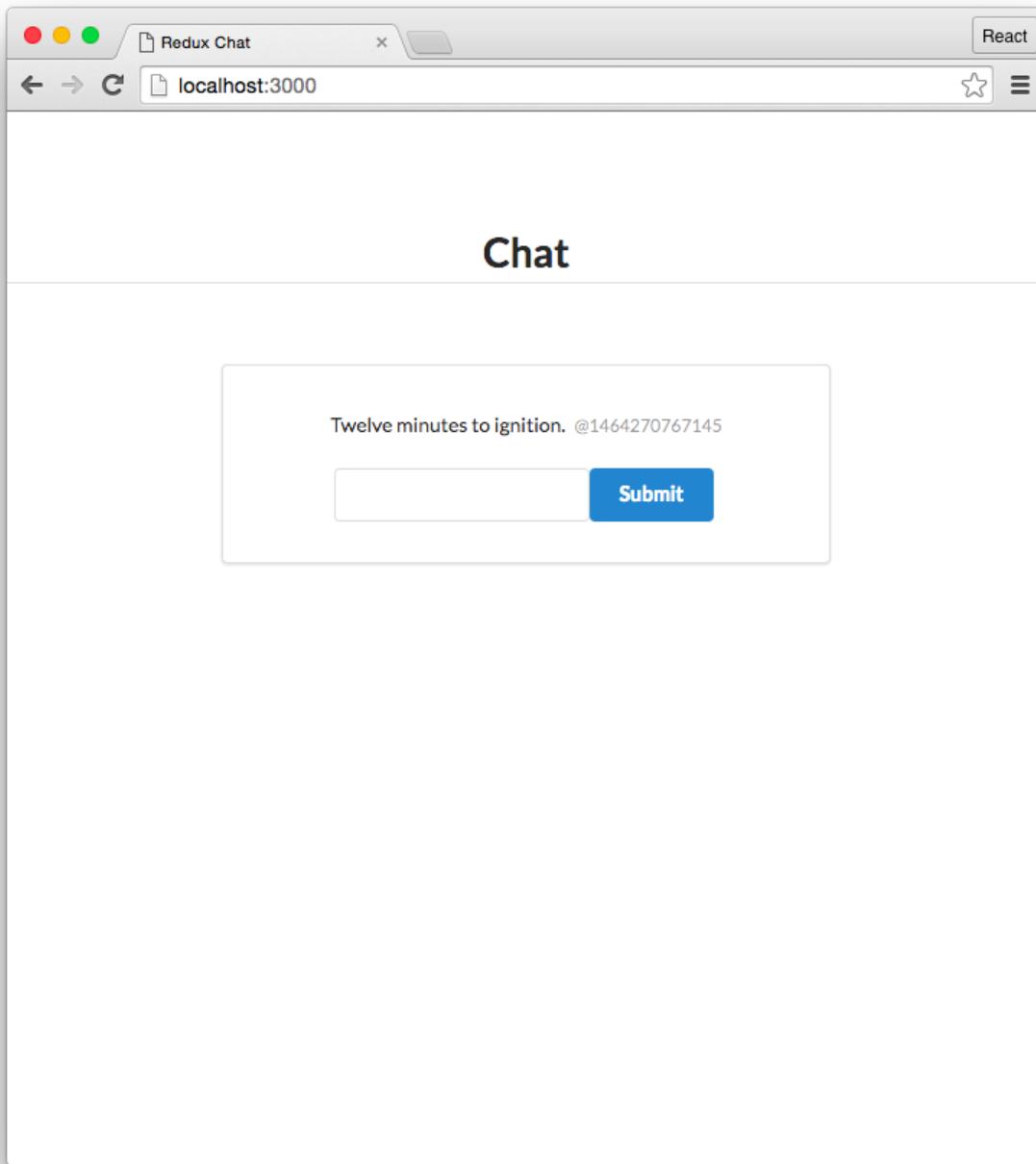
redux/chat_intermediate/public/app.js

```
118  return (
119    <div className='ui center aligned basic segment'>
120      <div className='ui comments'>
121        {messages}
122      </div>
123      <MessageInput />
124    </div>
125  );
```

Try it out

Save app.js. Navigating to <http://localhost:3000/>⁷², we see our app with a single message, the message we set in initialState. However, we cannot add or delete any messages:

⁷²<http://localhost:3000/>



Our actions and our reducer do not yet support our new state shape. Before we update them, though, let's add the `ThreadTabs` component to our app.

Adding the ThreadTabs component

Our App component will render `ThreadTabs` above its other child components. `ThreadTabs` will need a list of thread titles to render as tabs. We'll eventually have the component dispatch an action to update the `activeThreadId` piece of state whenever a tab is clicked, but for now we'll just have it render the thread titles.

Updating App

First, we'll prepare a `tabs` array. This array will contain objects that correspond to the information `ThreadTabs` needs to render each tab.

`ThreadTabs` will need two pieces of information:

- The title of each tab
- Whether or not the tab is the “active” tab

Indicating whether or not the tab is active is for style purposes. Here's an example of two tabs. In the markup, the left tab is indicated as active:



Inside `render` for `App`, we'll create an array of objects, `tabs`. Each object will contain a `title` and an `active` property. `active` will be a boolean:

redux/chat_intermediate/public/app.js

```
62 const tabs = threads.map(t => (
63   { // a "tab" object
64     title: t.title,
65     active: t.id === activeThreadId,
66   }
67 ));
```

We add ThreadTabs to the App component's markup, passing down tabs as a prop:

redux/chat_intermediate/public/app.js

```
69 return (
70   <div className='ui segment'>
71     <ThreadTabs tabs={tabs} />
72     <Thread thread={activeThread} />
73   </div>
74 );
```

Creating ThreadTabs

Next, let's add the ThreadTabs component right beneath the declaration of App. While we'll soon dispatch actions from ThreadTabs whenever a tab is clicked, for now it will just render the HTML for our tabs.

We first map over `this.props.tabs`, preparing the markup for each tab. In Semantic UI, we represent each tab as a `div` with a `class` of `item`. The active tab has a `class` of `active item`. We'll use `index` for the React-required `key` prop of each tab:

redux/chat_intermediate/public/app.js

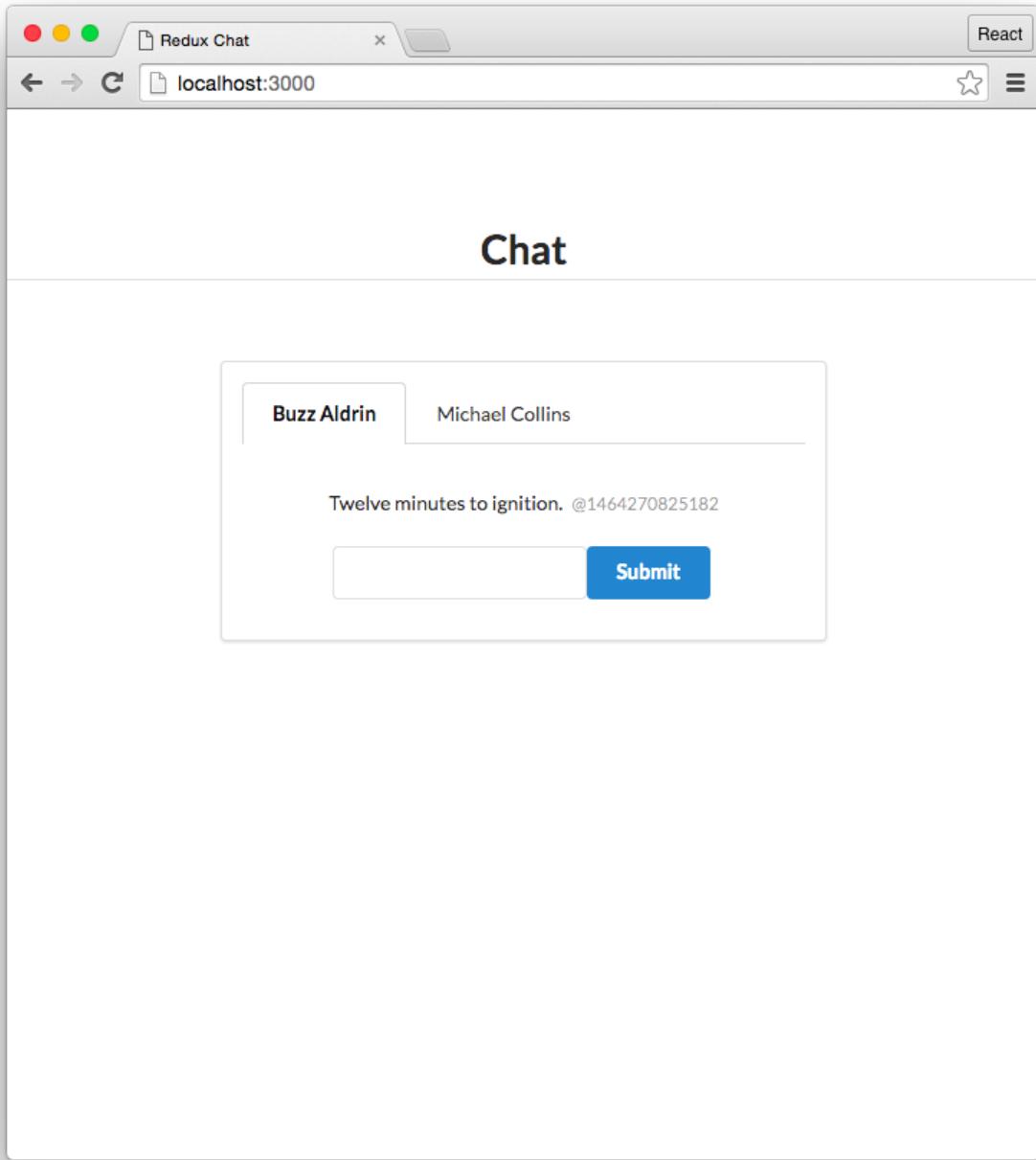
```
78 const ThreadTabs = React.createClass({
79   render: function () {
80     const tabs = this.props.tabs.map((tab, index) => (
81       <div
82         key={index}
83         className={tab.active ? 'active item' : 'item'}
84       >
85         {tab.title}
86       </div>
87     )));
88   return (
```

```
89      <div className='ui top attached tabular menu'>
90        {tabs}
91      </div>
92    );
93  },
94});
```

Try it out

Save `app.js`. Ensure the server is still running, and browse to <http://localhost:3000/>⁷³. Everything is as before — we can't add or delete messages — but we now have tabs in the interface. We can't switch between the tabs yet, though:

⁷³<http://localhost:3000/>



The first thread (“Buzz Aldrin”) is our active thread. Its corresponding `tab` object has its `active` property set to `true`. This sets the `class` of the tab to `active item`, which gives us a nice visual indication of the active tab on the interface.

We’ve updated our state to support threads and our React components are properly rendering based on this new representation. Next, we’ll restore interactivity by updating the actions and the reducer to work with this new state model.

Supporting threads in the reducer

Because our representation of state for this app has changed, we'll need to update the action handlers in our reducer.

Updating ADD_MESSAGE in the reducer

Because messages now belong to a thread, our ADD_MESSAGE action handler will need to add new messages to a specific thread. We'll add a property to this action object, `threadId`:

```
{  
  type: 'ADD_MESSAGE',  
  text: 'Looking good.',  
  threadId: '1-fca2', // <- Or whichever thread is appropriate  
}
```

Before, our ADD_MESSAGE handler in `reducer()` created a new message object and then used `concat` to append it to `state.messages`.

Now, we need to do the following:

1. Create the new message object `newMessage`
2. Find the corresponding thread (`action.threadId`) in `state.threads`
3. Append `newMessage` to the end of `thread.messages`

Let's modify `reducer()` in `app.js` now.

We keep the instantiation of the `newMessage` object. Next, we define `threadIndex` by hunting through `state.threads` and identifying the thread that corresponds to `action.threadId`:

`redux/chat_intermediate/public/app.js`

```
3  const newMessage = {  
4    text: action.text,  
5    timestamp: Date.now(),  
6    id: uuid.v4(),  
7  };  
8  const threadIndex = state.threads.findIndex(  
9    (t) => t.id === action.threadId  
10   );
```

Now, we might be tempted to just update the `messages` property on the thread like this:

```
// tempting, but flawed
const thread = state.threads[threadIndex];
thread.messages = thread.messages.concat(newMessage);
return state;
```

This would technically work; `thread` is a reference to the `thread` object sitting in `state.threads`. So by setting `thread.messages` to a new array with the new message included, we're modifying the `thread` object inside `state.threads`.

But this **mutates state**. As we discussed in the last chapter, `reducer()` *must* be a pure function. This means treating the state object as read-only.

We can't, therefore, modify the `thread` object. Instead, we can create a *new* `thread` object that contains the updated `thread.messages` property.

So, adding some detail to our plan from before, we actually want to do the following:

1. Create the new message object `newMessage`
2. Find the corresponding `thread` (`state.activeThreadId`) in `state.threads`
3. Create a **new thread object that contains all of the properties of the original thread object, plus an updated `messages` property**
4. Return `state` with a `threads` property that contains our *new* `thread` object in place of the *original* one

We already have step 1 taken care of. We have the `threadIndex` defined. Let's see how we create our new `thread` object:

`redux/chat_intermediate/public/app.js`

```
11 const oldThread = state.threads[threadIndex];
12 const newThread = {
13   ...oldThread,
14   messages: oldThread.messages.concat(newMessage),
15 }
```

To create `newThread`, we use ES7's spread syntax *for objects*. We used the spread syntax in the last chapter for arrays, creating a new array based off of chunks of an existing one. Here, we're using it to create a new *object* based on properties of an existing object.

This line *copies* all of the properties from `oldThread` to `newThread`:

redux/chat_intermediate/public/app.js

13 ... oldThread,

And then this line sets the `messages` property of `newThread` to the new messages array that contains `newMessage`:

redux/chat_intermediate/public/app.js

14 messages: oldThread.messages.concat(newMessage),

Note that by having the property `messages` appear *after* `oldThread`, we're effectively "overwriting" the `messages` property from `oldThread`.

We could have used `Object.assign()` to perform the same operation:

```
Object.assign({}, oldThread, {  
  messages: oldThread.messages.concat(newMessage),  
});
```

As you may recall, the first argument for `Object.assign()` is the target object. You can pass in as many additional arguments which are all of the objects you would like to copy properties from.

Because we perform lots of operations like this when working with pure reducer functions in Redux, we prefer the terser syntax of ES7's spread operator.



So far in the book, we've introduced new syntax for ES6. The ES6 standard was finalized in 2015 and is being rapidly adopted by both browsers and the community.

While the spread operator for arrays was a part of ES6, the spread operator for *objects* is still in the proposal stage for ES7, the next standard for JavaScript. Because ES7 proposals are subject to change, we avoid ES7 syntax in this book.

We're making an exception for the spread syntax for objects, however. The new syntax makes writing pure reducer functions in Redux much easier and cleaner. The syntax is supported with the Babel preset `stage-0` which is included in `package.json`.

If you want more info on how the spread syntax works with objects, see the aside "[ES7: Spread syntax with objects](#)".

Now, we have a `newThread` object that contains all of the properties of the original thread except with an updated `messages` property that contains the message being added.

The last step is to return the updated state. We want to return an object which has a `state.threads` that's set to the original list of threads except with our new thread object "subbed-in" for our old one.

We can re-use our strategy for creating a new array that contains chunks of a previous one. We have `threadIndex`, the index of the thread in the array we want to swap out. We want to create a new array that looks like this:

- All of the threads in `state.threads` up to but not including `threadIndex`
- `newThread`
- All of the threads in `state.threads` after `threadIndex`

That would look like this:

```
// Building a new array of threads
[
  ...state.threads.slice(0, threadIndex), // up to the thread
  newThread,                         // insert the new thread object
  ...state.threads.slice(
    threadIndex + 1, state.threads.length // after the thread
  ),
]
```

We can't set `state.threads` to this new array. That would be modifying `state`. Instead, we can create a new object and again use the spread operator to copy over all of the properties of `state` into the new object. Then, we can overwrite the `threads` property with our new array:

`redux/chat_intermediate/public/app.js`

```
17  return {
18    ...state,
19    threads: [
20      ...state.threads.slice(0, threadIndex),
21      newThread,
22      ...state.threads.slice(
23        threadIndex + 1, state.threads.length
24      ),
25    ],
26  };
```

Updating `ADD_MESSAGE` required some new concepts, but now we have an important strategy we'll be re-using in the future: how to update state objects while avoiding mutations.

Our new logic for handling the `ADD_MESSAGE` action, in full:

redux/chat_intermediate/public/app.js

```
2  if (action.type === 'ADD_MESSAGE') {
3      const newMessage = {
4          text: action.text,
5          timestamp: Date.now(),
6          id: uuid.v4(),
7      };
8      const threadIndex = state.threads.findIndex(
9          (t) => t.id === action.threadId
10     );
11      const oldThread = state.threads[threadIndex];
12      const newThread = {
13          ...oldThread,
14          messages: oldThread.messages.concat(newMessage),
15      };
16
17      return {
18          ...state,
19          threads: [
20              ...state.threads.slice(0, threadIndex),
21              newThread,
22              ...state.threads.slice(
23                  threadIndex + 1, state.threads.length
24              ),
25          ],
26      };
}
```

Introducing threads to our state significantly increased the complexity of this action handler. Soon, we'll explore ways to break this action handler apart into smaller pieces. For now, let's update the areas where we're dispatching the ADD_MESSAGE action. There's just one: the MessageInput component.

ES7: The spread operator with objects (...)

While the spread operator was introduced for arrays in ES6, it is a proposal for objects in ES7. The ellipsis ... operator copies one object into another:

```
const commonDolphin = {
    family: 'Delphinidae',
    genus: 'Delphinus',
};
```

```
const longBeakedDolphin = {
  ...commonDolphin,
  species: 'D. capensis',
};

// =>
// {
//   family: 'Delphinidae',
//   genus: 'Delphinus',
//   species: 'D. capensis',
// }

const spottedDolphin = {
  ...commonDolphin,
  genus: 'Stenella',
  species: 'S. attenuata',
};

// =>
// {
//   family: 'Delphinidae',
//   genus: 'Stenella',
//   species: 'S. attenuata',
// }

const atlanticSpottedDolphin = {
  ...spottedDolphin,
  species: 'S. frontalis',
};

// =>
// {
//   family: 'Delphinidae',
//   genus: 'Stenella',
//   species: 'S. frontalis',
// }
```

The spread operator enables us to succinctly construct new objects by copying over properties from existing ones. Because of this feature, we'll be using this operator often to keep our reducer functions pure.

Updating the MessageInput component

Our action object for ADD_MESSAGE should now contain the property `threadId`.

MessageInput is the only component that dispatches this action. The component should set `threadId` to the `id` of the active thread. We'll have `Thread` pass `MessageInput` the active thread's `id` as a prop:

redux/chat_intermediate/public/app.js

```
160 return (
161   <div className='ui center aligned basic segment'>
162     <div className='ui comments'>
163       {messages}
164     </div>
165     <MessageInput threadId={this.props.thread.id} />
166   </div>
167 );
```

Then, we can read from `this.props` in `MessageInput` to set `threadId` on the action object:

redux/chat_intermediate/public/app.js

```
112 const MessageInput = React.createClass({
113   handleSubmit: function () {
114     store.dispatch({
115       type: 'ADD_MESSAGE',
116       text: this.refs.messageInput.value,
117       threadId: this.props.threadId,
118     });
119     this.refs.messageInput.value = '';
120   },

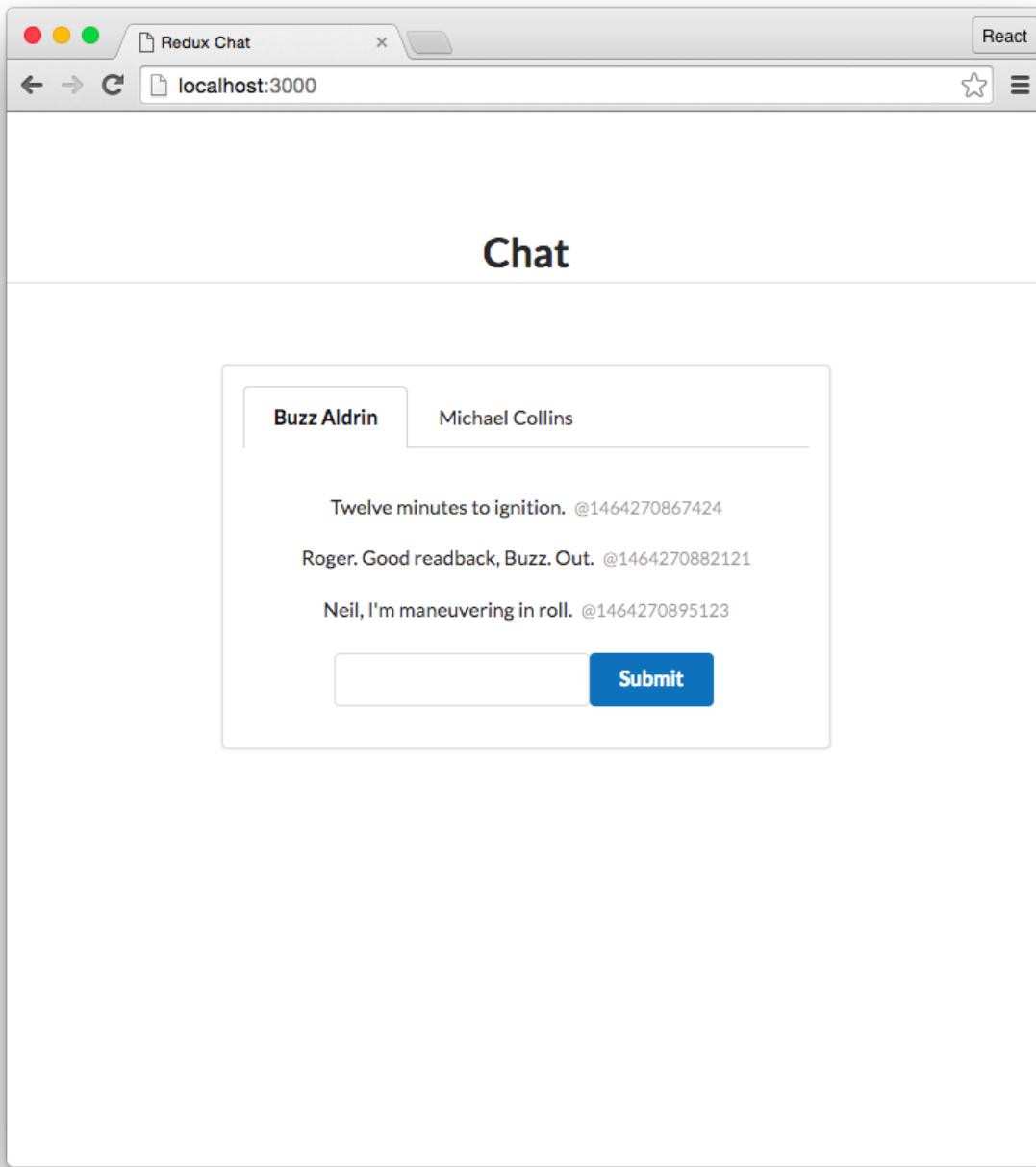
```

With `MessageInput` dispatching our updated `ADD_MESSAGE` action, let's verify that the functionality for adding messages has returned.

Try it out

Save `app.js`. Refreshing <http://localhost:3000/>⁷⁴, we see that we can now submit messages again:

⁷⁴<http://localhost:3000/>



We still can't delete messages or switch between our threads. Let's focus on updating the `DELETE_MESSAGE` action first.

Updating DELETE_MESSAGE in the reducer

The DELETE_MESSAGE action currently carries the property `id` which indicates which message should be deleted. While our app at the moment only permits a message to be deleted from the active thread, we'll write our reducer so that it searches all of the threads for the matching message.

When removing a message from state, we face a similar challenge as to the one that we faced with ADD_MESSAGE. The message is sitting in the `messages` array of a thread. But we can't modify the `thread` object in state.

We can use a similar strategy:

1. Grab the thread that holds the message we want to remove
2. Create a new thread object that contains all of the properties of the original thread object, plus an updated `messages` property that does *not* include the message we're removing
3. Return state with a `threads` property that contains our *new* thread object in place of the *original* one

Let's modify the reducer's DELETE_MESSAGE handler now.

First, we identify the thread that holds the message we're deleting:

redux/chat_intermediate/public/app.js

```
27 } else if (action.type === 'DELETE_MESSAGE') {
28     const threadIndex = state.threads.findIndex(
29         (t) => t.messages.find((m) => (
30             m.id === action.id
31         ))
32     );
33     const oldThread = state.threads[threadIndex];
```

Inside of the `findIndex` callback, we perform a `find` against the `messages` property of that thread. If `find` finds the message matching the action's `id`, it returns the message. This satisfies the `findIndex` testing function, meaning that the thread's index is returned.

Next, let's prepare the new version of the `messages` array for this thread. We're preparing this array exactly the same as before we introduced threads. After finding the `messageIndex`, we can build a new array that contains the chunks of the `messages` array before and after `messageIndex`:

redux/chat_intermediate/public/app.js

```

34  const messageIndex = oldThread.messages.findIndex(
35    (m) => m.id === action.id
36  );
37  const messages = [
38    ...oldThread.messages.slice(0, messageIndex),
39    ...oldThread.messages.slice(
40      messageIndex + 1, oldThread.messages.length
41    ),
42  ];

```

We then create a new thread object. We use the spread syntax to copy over all of the attributes from `oldThread` to this new object. We overwrite the `messages` property with the array we've prepared:

redux/chat_intermediate/public/app.js

```

43  const newThread = {
44    ...oldThread,
45    messages: messages,
46  };

```

The return statement for `DELETE_MESSAGE` is identical to the one for `ADD_MESSAGE`. We're performing the same operation: We ultimately want to “swap out” the original thread object in `state.threads` with our new one. So we want to:

1. Create a new object and copy over all the attributes from `state`
2. Overwrite the `threads` property with a new array of threads that has our new thread object swapped in for the original one

Again, the code is identical to that of `ADD_MESSAGE`:

redux/chat_intermediate/public/app.js

```

48  return {
49    ...state,
50    threads: [
51      ...state.threads.slice(0, threadIndex),
52      newThread,
53      ...state.threads.slice(
54        threadIndex + 1, state.threads.length
55      ),
56    ],
57  };

```

In full, our `DELETE_MESSAGE` action handler:

redux/chat_intermediate/public/app.js

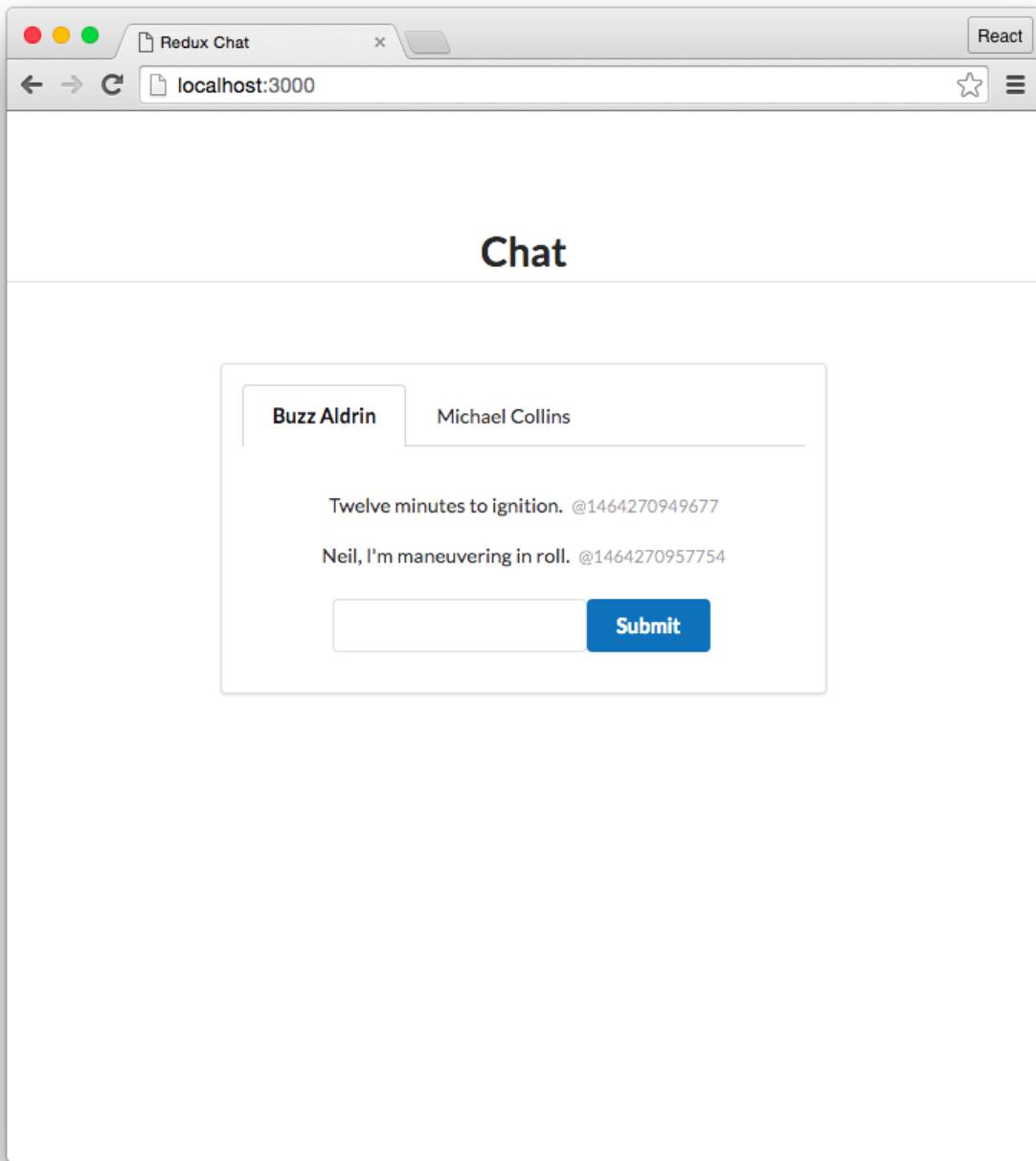
```
27 } else if (action.type === 'DELETE_MESSAGE') {
28   const threadIndex = state.threads.findIndex(
29     (t) => t.messages.find((m) => (
30       m.id === action.id
31     ))
32   );
33   const oldThread = state.threads[threadIndex];
34   const messageIndex = oldThread.messages.findIndex(
35     (m) => m.id === action.id
36   );
37   const messages = [
38     ...oldThread.messages.slice(0, messageIndex),
39     ...oldThread.messages.slice(
40       messageIndex + 1, oldThread.messages.length
41     ),
42   ];
43   const newThread = {
44     ...oldThread,
45     messages: messages,
46   };
47
48   return {
49     ...state,
50     threads: [
51       ...state.threads.slice(0, threadIndex),
52       newThread,
53       ...state.threads.slice(
54         threadIndex + 1, state.threads.length
55       ),
56     ],
57   };
}
```

The complexity of this action handler, like that of ADD_MESSAGE, was significantly complicated by the introduction of threads. What's more, we've witnessed similar patterns emerge between the two action handlers: They both instantiate a new thread object that's a derivative of an existing thread object. And they both swap this thread object in for the thread object being "modified" in state.

Soon, we'll explore a new strategy to share this code and break these procedures down. But for now, let's test out deleting messages again.

Try it out

Save `app.js`. Make sure your server is running and navigate to <http://localhost:3000>⁷⁵. Adding and deleting messages are both working now:



⁷⁵<http://localhost:3000>

Clicking on the tab to switch threads still does not work, however. We initialize our state with `activeThreadId` set to '`1-fca2`' but have no actions in the system to modify this part of the state.

Adding the action OPEN_THREAD

We'll introduce another action, `OPEN_THREAD`, which React will dispatch whenever the user clicks on a thread tab to open it. This action will ultimately modify the `activeThreadId` property on state.

The component `ThreadTabs` will be the one to dispatch this action.

The action object

The action object just needs to specify the `id` of the thread the user wants to open:

```
{  
  type: 'OPEN_THREAD',  
  id: '2-be91', // <- or whichever `id` is appropriate  
}
```

Modifying the reducer

Let's add another clause to our reducer to handle this new action. We'll add this clause below the `DELETE_MESSAGE` clause but before the final `else` statement.

We can't modify the state object directly:

```
// Incorrect  
} else if (action.type === 'OPEN_THREAD') {  
  state.activeThreadId = action.id;  
  return state;  
}
```

Instead, we'll copy over all of the attributes from `state` into a new object. We'll overwrite the `activeThreadId` property of this new object:

redux/chat_intermediate/public/app.js

```

58 } else if (action.type === 'OPEN_THREAD') {
59   return {
60     ...state,
61     activeThreadId: action.id,
62   };

```

Using this strategy, we do not modify state.

Now, we just need to have ThreadTabs dispatch this action whenever a tab is clicked.

Dispatching from ThreadTabs

In order for ThreadTabs to dispatch the OPEN_THREAD action, it needs the id of the thread that was clicked on.

Right now, the tab object that we're giving to ThreadTabs contains the properties title and active. We'll need to modify the instantiation of tabs in App to also include the property id.

Let's do that first. Inside the App component, add this property to the tab object that we create:

redux/chat_intermediate/public/app.js

```

102 const tabs = threads.map(t => (
103   {
104     title: t.title,
105     active: t.id === activeThreadId,
106     id: t.id,
107   }
108 ));

```

Next, we'll add the component function handleClick to ThreadTabs. The function accepts id:

redux/chat_intermediate/public/app.js

```

119 const ThreadTabs = React.createClass({
120   handleClick: function (id) {
121     store.dispatch({
122       type: 'OPEN_THREAD',
123       id: id,
124     });
125   },

```

Finally, we add an onClick attribute to the div tag for each tab. We set it to a function that calls handleClick with the thread's id:

redux/chat_intermediate/public/app.js

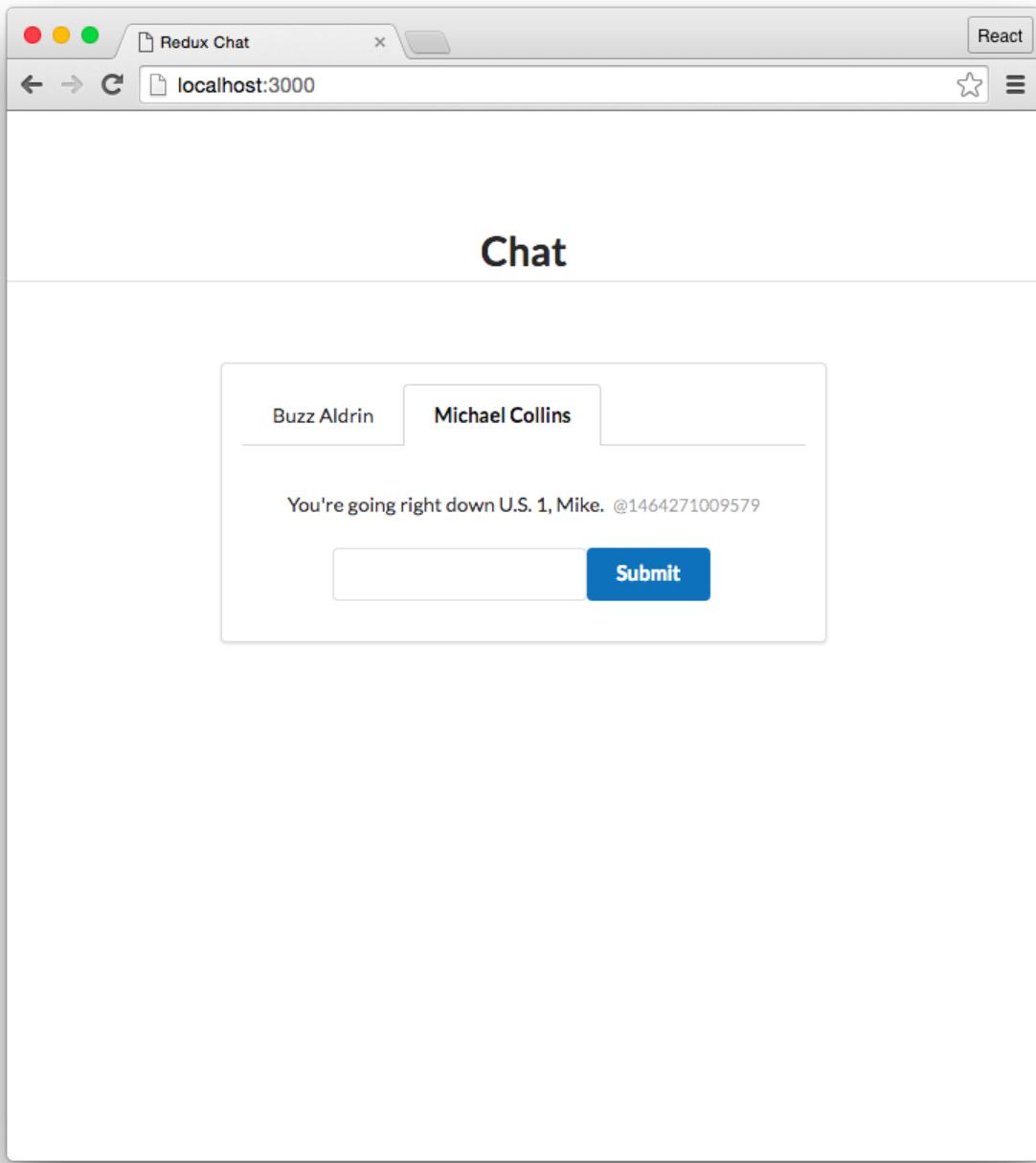
```
127  const tabs = this.props.tabs.map((tab, index) => (
128    <div
129      key={index}
130      className={tab.active ? 'active item' : 'item'}
131      onClick={() => this.handleClick(tab.id)}
132    >
```

ThreadTabs is now emitting our new action. Let's test everything out.

Try it out

Save app.js. Pull up <http://localhost:3000/>⁷⁶ in your browser. At this point, adding and deleting messages works and we can switch between tabs. If we add a message to one thread, it's added to just that thread:

⁷⁶<http://localhost:3000/>



This is starting to look like an actual chat application! We've introduced the concept of threads and now support an interface that can switch between conversations with multiple users.

However, in the process, we significantly complicated our reducer function. While it's nice that all of our state is being managed in a single location, each of our action handlers contain a lot of logic. What's more, our ADD_MESSAGE and DELETE_MESSAGE handlers duplicate a lot of the same code.

It's also odd that the management of two distinct pieces of state — adding/deleting messages and switching between threads — are both managed in the same place. The idea of having a single reducer function manage the entirety of our state as the complexity of our app grows might feel daunting and even a bit dubious.

Indeed, Redux apps have a strategy for breaking down state management logic: **reducer composition**.

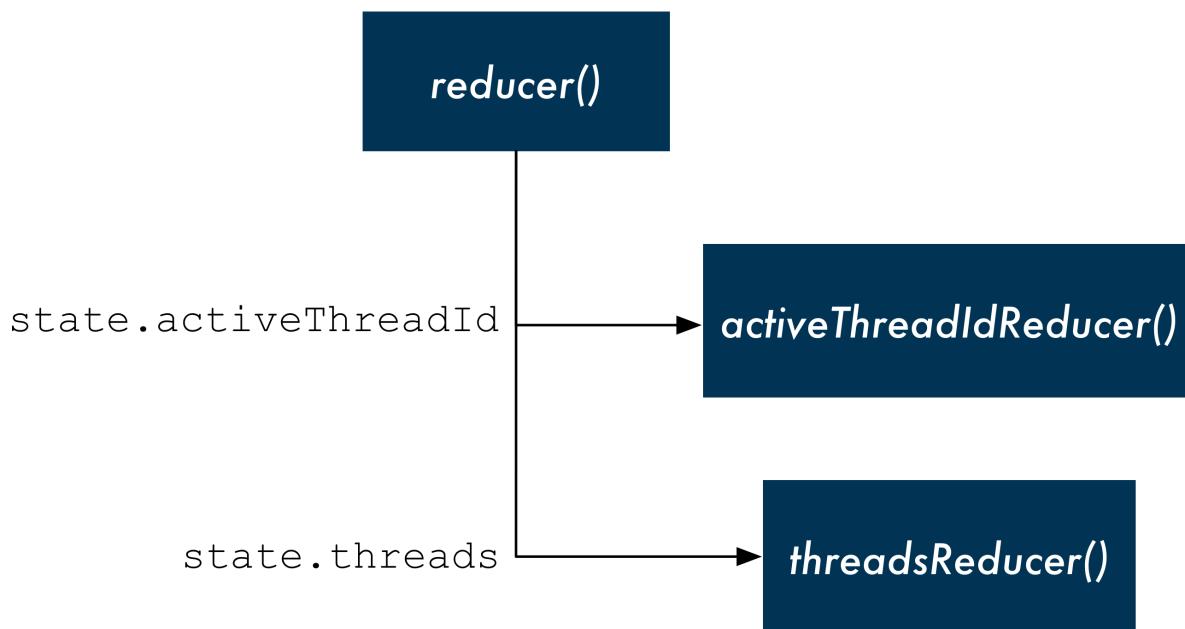
Breaking up the reducer function

With reducer composition, we can break up the state management logic of our app into smaller functions. We'll still pass in a single reducer function to `createStore()`. But this top-level function will then call one or more other functions. Each reducer function will manage a different part of the state tree.

Adding and deleting messages is a distinct piece of functionality from switching between threads. Let's break these two apart first.

A new `reducer()`

We'll still call our top-level function `reducer()`. Except now, we'll have two other reducer functions, each managing their part of the state's top-level properties. We'll name each of these sub-reducers after the property name that they manage:



To implement this, let's first see what our new top-level `reducer()` function should look like. We can then create our sub-reducer functions.

To avoid confusion, let's rename our current reducer function from `reducer()` to `threadsReducer()`:

`redux/chat_intermediate/public/app.js`

```
1 function threadsReducer(state, action) {
```

We still have to tweak this function, but we'll get back to it in a bit.

Now, above `threadsReducer()`, we'll insert our *new* function, `reducer()`:

`redux/chat_intermediate/public/app.js`

```
1 function reducer(state, action) {
2   return {
3     activeThreadId: activeThreadIdReducer(state.activeThreadId, action),
4     threads: threadsReducer(state.threads, action),
5   };
6 }
```

The function is short but there's a lot going on.

We are returning a brand new object with the keys `activeThreadId` and `threads`. Importantly, we are *delegating the responsibility of how to update activeThreadId and threads to their respective reducers*.

Check out this line again:

`redux/chat_intermediate/public/app.js`

```
3   activeThreadId: activeThreadIdReducer(state.activeThreadId, action),
```

For the property `activeThreadId` on next state, we're delegating responsibility to the yet-to-be-defined function `activeThreadIdReducer()`. Notice how the first argument is not the whole state — it's only the part of the state that this reducer is responsible for. We pass in the action as the second argument.

Same strategy with `threads`:

`redux/chat_intermediate/public/app.js`

```
4   threads: threadsReducer(state.threads, action),
```

We delegate responsibility of the state's `threads` property to `threadsReducer()`. The state that we pass this reducer — the first argument — is the part of the state that `threadsReducer()` is responsible for updating.

This is how sub-reducers work in tandem with the top-level reducer. The top-level reducer breaks up each part of the state tree and delegates the management of those pieces of state to the appropriate reducer.

To get a better idea, let's write `activeThreadIdReducer()` below `reducer()`:

`redux/chat_intermediate/public/app.js`

```
8 function activeThreadIdReducer(state, action) {
9   if (action.type === 'OPEN_THREAD') {
10     return action.id;
11   } else {
12     return state;
13   }
14 }
```

This reducer only handles one type of action, `OPEN_THREAD`. Remember, the `state` argument being passed to the reducer is actually `state.activeThreadId`. This is the only part of the state that this reducer needs to be concerned with. As such, `activeThreadIdReducer()` receives a string as `state` (the id of a thread) and will return a string.

Notice how this simplifies the logic for handling `OPEN_THREAD`. Before, our logic had to consider the *entire* state tree. So we had to create a new object, copy over all of the state's properties, and then overwrite the `activeThreadId` property. Now, our reducer just has to worry about this single property.

If the action is `OPEN_THREAD`, `activeThreadIdReducer()` simply returns `action.id`, the id of the thread to be opened. Otherwise, it returns the same id it was passed.

Up in `reducer()`, the return value of `activeThreadIdReducer()` is set to the property `activeThreadId` in the new state object.

Let's update `threadsReducer()` next. Before, this function received the entire state object. Now, it's only receiving part of it: `state.threads`. We'll be able to simplify the code a bit as a result.

Updating `threadsReducer()`

`threadsReducer()` is now receiving only part of the state. Its `state` argument is the array of threads.

As a result, we can simplify our returns like we did with `activeThreadIdReducer()`. We no longer need to create a new state object, copy all the old values over, then overwrite `threads`. Instead, we can just return the updated array of threads.

What's more, instead of referencing `state.threads` everywhere we'll reference `state` as this is now the array of threads.

Let's make these modifications first for `ADD_MESSAGE`.

We reference `state` as opposed to `state.threads`:

`redux/chat_intermediate/public/app.js`

```
23  const threadIndex = state.findIndex(
24    (t) => t.id === action.threadId
25  );
26  const oldThread = state[threadIndex];
27  const newThread = {
28    ...oldThread,
29    messages: oldThread.messages.concat(newMessage),
30  };
```

For the return statement, we no longer need to return a full state object, just the array of threads:

`redux/chat_intermediate/public/app.js`

```
32  return [
33    ...state.slice(0, threadIndex),
34    newThread,
35    ...state.slice(
36      threadIndex + 1, state.length
37    ),
38  ];
```

`DELETE_MESSAGE` will receive the same treatment.

We first change our references from `state.threads` to `state`:

`redux/chat_intermediate/public/app.js`

```
40  const threadIndex = state.findIndex(
41    (t) => t.messages.find((m) => (
42      m.id === action.id
43    ))
44  );
45  const oldThread = state[threadIndex];
```

Then, we return just the array as opposed to the whole state object:

redux/chat_intermediate/public/app.js

```
60  return [
61    ...state.slice(0, threadIndex),
62    newThread,
63    ...state.slice(
64      threadIndex + 1, state.length
65    ),
66  ];
```

Finally, we can remove the OPEN_THREAD handler from `threadsReducer()`. This reducer is not concerned about the OPEN_THREAD action. That action pertains to a part of the state tree that this reducer does not work with. So, whenever that action is received, the function will reach the last `else` clause and will just return `state` unmodified.

Our updated `threadsReducer()` function, in full:

redux/chat_intermediate/public/app.js

```
18  if (action.type === 'ADD_MESSAGE') {
19    const newMessage = {
20      text: action.text,
21      timestamp: Date.now(),
22      id: uuid.v4(),
23    };
24    const threadIndex = state.findIndex(
25      (t) => t.id === action.threadId
26    );
27    const oldThread = state[threadIndex];
28    const newThread = {
29      ...oldThread,
30      messages: oldThread.messages.concat(newMessage),
31    };
32
33    return [
34      ...state.slice(0, threadIndex),
35      newThread,
36      ...state.slice(
37        threadIndex + 1, state.length
38      ),
39    ];
40  } else if (action.type === 'DELETE_MESSAGE') {
41    const threadIndex = state.findIndex(
```

```
42     (t) => t.messages.findIndex((m) => (
43         m.id === action.id
44     ))
45 );
46 const oldThread = state[threadIndex];
47 const messageIndex = oldThread.messages.findIndex(
48     (m) => m.id === action.id
49 );
50 const messages = [
51     ...oldThread.messages.slice(0, messageIndex),
52     ...oldThread.messages.slice(
53         messageIndex + 1, oldThread.messages.length
54     ),
55 ];
56 const newThread = {
57     ...oldThread,
58     messages: messages,
59 };
60
61 return [
62     ...state.slice(0, threadIndex),
63     newThread,
64     ...state.slice(
65         threadIndex + 1, state.length
66     ),
67 ];
68 } else {
69     return state;
70 }
71 }
```

By focusing this reducer and having it deal with only the `threads` property in state, we were able to simplify our code a little. Our return functions no longer have to worry about the entire state tree and we removed an action handler from this function.

We still duplicate logic between the two action handlers. Notably, their return functions are the same.

In fact, while we simplified the reducer by having it work with only `state.threads`, this reducer is actually dealing with two levels of the state tree: both threads *and* messages. We can further break our reducer logic into smaller pieces and share code by having `threadsReducer()` delegate the responsibility to another reducer: `messagesReducer()`.

We'll explore this in the next section. For now, let's boot up the app and verify our solution so far

works.



You might be asking: Why not just rename the first argument in `threadsReducer()` to `threads` as opposed to calling it `state`?

Indeed, doing this could improve readability of the reducer function. You'd no longer need to hold the idea in working memory that `state` is actually `state.threads`.

However, consistently calling the first argument to a Redux reducer `state` can be helpful for avoiding errors. It's a clear reminder that the argument is a part of the state tree and that you should avoid accidentally mutating it.

Ultimately, it's a matter of personal preference.

Try it out

Save `app.js`. Boot the server if it isn't running:

```
$ npm run server
```

And then point your browser to <http://localhost:3000/>⁷⁷. On the surface, it should appear as though nothing has changed; adding and deleting messages works and we can switch between threads.

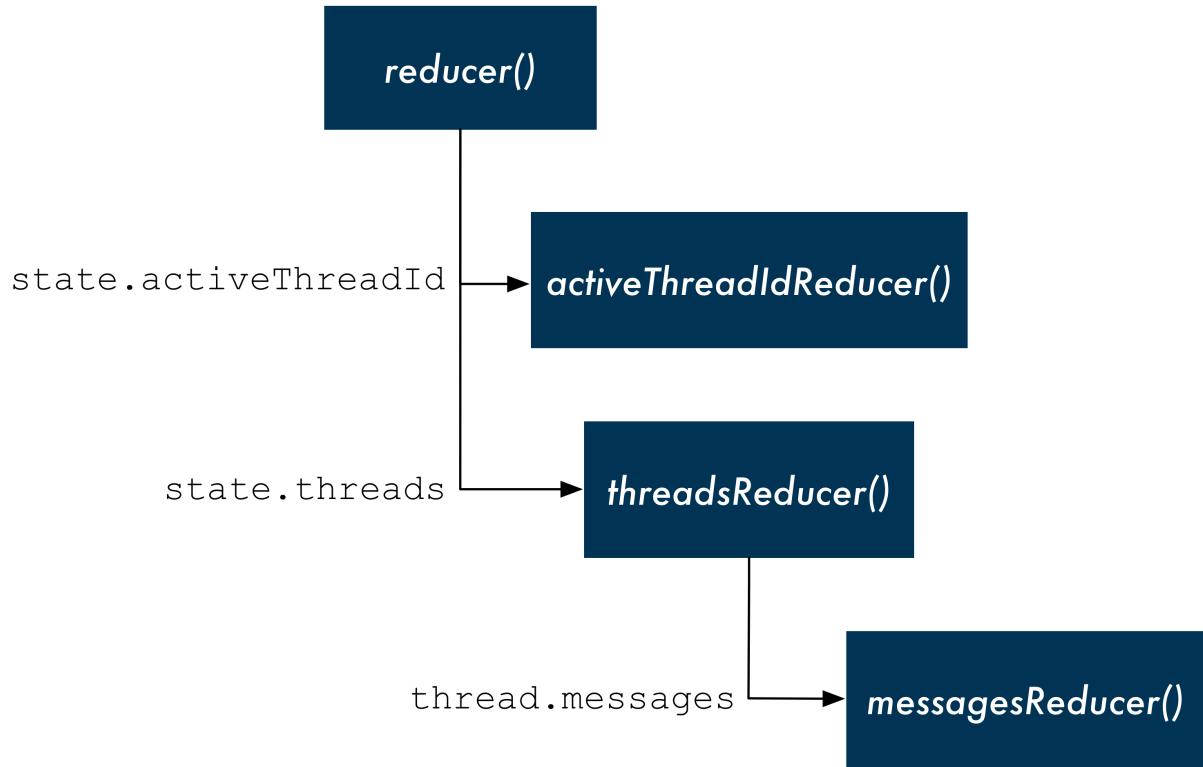
Adding `messagesReducer()`

We used the concept of reducer composition to break up the management of our state. Our top-level `reducer()` function delegates the management of the two top-level state properties to their respective reducers.

We can take this concept even further. We've noticed that `threadsReducer()` has the responsibility of managing the state of both threads and messages. What's more, we see an opportunity to share code between the two action handlers.

We can have `threadsReducer()` delegate the management of each thread's `messages` property to another reducer, `messagesReducer()`. In full, the reducer tree will look like this:

⁷⁷<http://localhost:3000/>



Let's first modify `threadsReducer()`, anticipating this new messages reducer function. We want the threads reducer function to know as little as possible about messages. `threadsReducer()` will rely on `messagesReducer()` to determine how a given thread's messages should be updated based on the action received.

Modifying the ADD_MESSAGE action handler

Let's see what happens when we delegate all message handling functionality to our anticipated `messagesReducer()` function.

We'll no longer declare `newMessage`. We expect that `messagesReducer()` will take care of this.

Then, when specifying the `messages` attribute for `newThread`, we'll delegate the creation of this array to `messagesReducer()`:

redux/chat_intermediate/public/app.js

```
16  function threadsReducer(state, action) {
17    if (action.type === 'ADD_MESSAGE') {
18      const threadIndex = state.findIndex(
19        (t) => t.id === action.threadId
20      );
21      const oldThread = state[threadIndex];
22      const newThread = {
23        ...oldThread,
24        messages: messagesReducer(oldThread.messages, action),
25      };

```

We pass `messagesReducer()` the array `oldThread.messages` as the first argument and `action` as the second. This follows our pattern so far: `reducer()` passed `threadsReducer()` `state.threads` as its first argument. `threadsReducer()` passes `thread.messages` as the first argument to `messagesReducer()`. We now know that when we build `messagesReducer()`, the first argument (`state`) will be an array of a given thread's messages.

The full `action` object is passed along in its entirety down the chain.

We do not need to make any modifications to the return value.

At the moment, this might only seem like a minor — perhaps negligible — victory. Before, we were calling `concat` in-line to create a new `messages` array with the new message inside. Now we have the added complexity of calling another function to do this.

We are breaking apart the responsibility of our functions, which is a worthwhile endeavor in its own right. Furthermore, another benefit will become apparent when we give `DELETE_MESSAGE` the same treatment.

Prior to modifying our `DELETE_MESSAGE` action handler, let's start on our `messagesReducer()` function to get an idea of how these reducers will work together.

Creating `messagesReducer()`

We'll write `messagesReducer()` below `threadsReducer()` in `app.js`. We'll start it off with one action handler (`ADD_MESSAGE`).

As we saw above, `threadsReducer()` passes `messagesReducer()` a thread's `messages` array as the first argument. This will be `state` inside of `messagesReducer()`.

`messagesReducer()` needs to:

1. Create the new message
2. Return a new array of messages that includes this new message appended to the end of it

We'll use the same logic that we had previously in `threadsReducer()` to accomplish this:

redux/chat_intermediate/public/app.js

```
67 function messagesReducer(state, action) {
68   if (action.type === 'ADD_MESSAGE') {
69     const newMessage = {
70       text: action.text,
71       timestamp: Date.now(),
72       id: uuid.v4(),
73     };
74     return state.concat(newMessage);
75   } else {
76     return state;
77   }
78 }
```

`messagesReducer()` receives an array of messages as its first argument, `state`. The `ADD_MESSAGE` handler creates a new message using `action.text`. We use `concat` just as before to return a new messages array with our new message appended to it.

Now that we've seen how `threadsReducer()` delegates `ADD_MESSAGE` to `messagesReducer()`, let's take a look at how `DELETE_MESSAGE` can do the same thing.

Modifying the `DELETE_MESSAGE` action handler

As with the `ADD_MESSAGE` action handler, we want our `DELETE_MESSAGE` action handler in `threadsReducer()` to delegate the responsibility of handling the `messages` property of each thread to `messagesReducer()`.

To do this, we can first remove messages-specific logic, anticipating that `messagesReducer()` will handle it. This means we can remove the creation of a new messages array that doesn't include the message being deleted. Instead, we can call `messagesReducer()` inside of the declaration of `newThread` and have it set the `messages` property, like this:

redux/chat_intermediate/public/app.js

```
34 } else if (action.type === 'DELETE_MESSAGE') {
35   const threadIndex = state.findIndex(
36     (t) => t.messages.findIndex((m) => (
37       m.id === action.id
38     ))
39   );
40   const oldThread = state[threadIndex];
41   const newThread = {
```

```
42     ...oldThread,
43     messages: messagesReducer(oldThread.messages, action),
44 };

---


```

Look familiar? With these modifications, the threads reducer's `DELETE_MESSAGE` looks almost exactly like its `ADD_MESSAGE`. The only difference is how the two handlers identify `threadIndex`.

On reflection, this makes sense. The action that we're dealing with is adding and removing messages. This will only ever affect the `messages` property on an individual thread. Because the `messages` property is now being managed by `messagesReducer()`, the differences between `ADD_MESSAGE` and `DELETE_MESSAGE` are expressed in that function. Aside from determining `threadIndex`, the rest of the ceremony — creating a new thread object with an updated `messages` property — is identical.

We can define a new function, `findThreadIndex()`, where we can store the logic to determine `threadIndex`. Then, we can combine our two action handlers as their code will be identical.

First, we'll write `findThreadIndex()` above `threadsReducer()`. This function will take both `threads` (or state in `threadsReducer()`) and the `action` as arguments. We'll copy the logic for finding the index of the affected thread into this function:

redux/chat_intermediate/public/app.js

```
16 function findThreadIndex(threads, action) {
17   switch (action.type) {
18     case 'ADD_MESSAGE': {
19       return threads.findIndex(
20         (t) => t.id === action.threadId
21       );
22     }
23     case 'DELETE_MESSAGE': {
24       return threads.findIndex(
25         (t) => t.messages.findIndex(
26           (m) => m.id === action.id
27         )
28       );
29     }
30   }
31 }

---


```

We decided to use a `switch` statement here as opposed to an `if/else` clause. Using a `switch` statement in reducers and their helper functions can be slightly easier to read and manage. You'll see it used in many Redux apps. As the actions in a system grows, a single reducer will have to have multiple action handlers. `switch` is built for this use case.

Another reason for using `switch` in `findThreadIndex()` is because we'll use one in our refactored `threadsReducer()`:

redux/chat_intermediate/public/app.js

```
33 function threadsReducer(state, action) {
34   switch (action.type) {
35     case 'ADD_MESSAGE':
36     case 'DELETE_MESSAGE': {
37       const threadIndex = findThreadIndex(state, action);
38
39       const oldThread = state[threadIndex];
40       const newThread = {
41         ...oldThread,
42         messages: messagesReducer(oldThread.messages, action),
43       };
44
45       return [
46         ...state.slice(0, threadIndex),
47         newThread,
48         ...state.slice(
49           threadIndex + 1, state.length
50         ),
51       ];
52     }
53     default: {
54       return state;
55     }
56   }
57 }
```

switch is nice here because this:

redux/chat_intermediate/public/app.js

```
34   switch (action.type) {
35     case 'ADD_MESSAGE':
36     case 'DELETE_MESSAGE': {
```

Reads a bit clearer than the alternative:

```
if (action.type === 'ADD_MESSAGE' || action.type === 'DELETE_MESSAGE') {  
  // ...  
}
```

This readability concern will be exacerbated if we continue to introduce more actions to the system that just affect the `messages` property of a given thread (like `UPDATE_MESSAGE`). With a `switch` statement, we can cleanly specify that all of these actions share the same code block.

Other than the call to `findThreadIndex()`, the body of the combined action handler matches what we had individually for each above.

Adding `DELETE_MESSAGE` to `messagesReducer()`

The `threadsReducer()` function now treats `ADD_MESSAGE` and `DELETE_MESSAGE` actions in the exact same way. When the reducer receives a `DELETE_MESSAGE` action, it will call `messagesReducer()` with a list of messages and the action. The list of messages corresponds to an individual thread's `messages` property. The action carries the directive for which message to remove.

Let's add the `DELETE_MESSAGE` logic to `messagesReducer()`. You can keep the `if/else` clause for now or change to `switch`:

`redux/chat_intermediate/public/app.js`

```
59  function messagesReducer(state, action) {  
60    switch (action.type) {  
61      case 'ADD_MESSAGE': {  
62        const newMessage = {  
63          text: action.text,  
64          timestamp: Date.now(),  
65          id: uuid.v4(),  
66        };  
67        return state.concat(newMessage);  
68      }  
69      case 'DELETE_MESSAGE': {  
70        const messageIndex = state.findIndex(m => m.id === action.id);  
71        return [  
72          ...state.slice(0, messageIndex),  
73          ...state.slice(  
74            messageIndex + 1, state.length  
75          ),  
76        ];  
77      }  
78      default: {  
79        return state;
```

```
80      }
81    }
82 }
```

Remember, state here is the array of messages. The logic for deleting the message is the same that we had in `threadsReducer()`. We find the message's index and then return a new array that has two chunks: all the elements before that index and all the elements after.

At this point, we've broken out the logic to manage state across three reducer functions. Each function manages a different part of the state tree. Our tree of reducer functions is combined up at `reducer()`, which is the function that we pass to `createStore()`.

The complexity of both our app and our state increased significantly in this chapter. We now see how using reducer composition allows us to manage this complexity. We have a pattern for scaling our system to handle the addition of many more actions and pieces of state.

There's one more improvement we can make. While the logic for managing state updates is contained within our reducers, the initial state of the app is defined elsewhere, in `initialState`. Let's bring the parts of this initialization closer to their respective reducers. This will scale better as our state tree grows. In addition, it means *all* of the logic around a particular part of the state tree is contained inside of its reducer.

Defining the initial state in the reducers

Right now, we declare our `initialState` object and then pass it into `Redux.createStore()`:

```
106 const store = Redux.createStore(reducer, initialState);
```

This argument is not required. Let's change this line so that we no longer pass in `initialState`:

```
106 const store = Redux.createStore(reducer);
```

What will happen?

The `createStore()` function in the `redux` library contains one key difference from the one we wrote in the last chapter. After the store is initialized but *right before* it's returned, `createStore()` actually dispatches an initialization action. That dispatch call looks like this:

```
// ...
// Inside of `createStore()` in `redux`
dispatch({ type: '@@redux/INIT' });

return { // returns store object
  dispatch,
  subscribe,
  getState,
}
}
```

While the *initialization action* has a type, chances are you'll never need to use it.

The important part is that because we didn't specify an `initialState`, `state` will be `undefined` when `createStore()` dispatches the initialization action. The only time `state` will be `undefined` is for this first dispatch.

Therefore, we can have our reducers specify the initial state for their part of the state tree whenever `state` is `undefined`.

Initial state in `reducer()`

When `createStore()` dispatches the initialization object, `reducer()` will receive a state that is `undefined`.

In this situation, we want to set `state` to a blank object (`{}`). This will enable `reducer()` to delegate the initialization of each property in the state object to its reducer. We'll see how this works in practice shortly.

We can use ES6's **default arguments** to achieve this:

`redux/chat_intermediate/public/app.js`

```
1 function reducer(state = {}, action) {
```

When `reducer()` receives a state of `undefined`, `state` is set to `{}`.

Importantly, when `reducer()` then calls each of its sub-reducers, each one of them will receive a `state` argument that is `undefined`.

A `state` that is `undefined` is the canonical way to initialize a reducer in Redux. While we could use this special initialization action object's type, with ES6's default arguments just relying on an `undefined` state is much easier.

ES6: Default arguments

With ES6, you can specify a default value for an argument in the case that it is `undefined` when the function is called.

This:

```
function divide(a, b) {
  // Default divisor to `1`
  const divisor = typeof b === 'undefined' ? 1 : b;

  return a / divisor;
}
```

Can be written as this:

```
function divide(a, b = 1) {
  return a / b;
}
```

In both cases, using the function looks like this:

```
divide(14, 2);
// => 7
divide(14, undefined);
// => 14
divide(14);
// => 14
```

Whenever the argument `b` in the example above is `undefined`, the default argument is used. Note that `null` will not use the default argument:

```
divide(14, null); // `null` is used as divisor
// => Infinity // 14 / null
```

Adding initial state to `activeThreadIdReducer()`

We can use default arguments to initialize the state in `activeThreadIdReducer()`. We want the initial state to be '`1-fca2`', the id we'll specify for our first thread:

redux/chat_intermediate/public/app.js

```
8 function activeThreadIdReducer(state = '1-fca2', action) {
```

Let's recap the initialization flow for `activeThreadIdReducer()`.

At the end of `createStore()`, right before the function returns the new store object, it dispatches the initialization action. Then:

1. `reducer()` is called with a `state` of `undefined` and the initialization action object
2. `state` defaults to `{}`
3. `reducer()` calls `activeThreadIdReducer()` with `state.activeThreadId`, which is `undefined`
4. `activeThreadIdReducer()` sets `state` to '`1-fca2`' with its default argument
5. The `else` clause in `activeThreadIdReducer()` returns `state ('1-fca2')`

Adding initial state to `threadsReducer()`

We'll use the same strategy for `threadsReducer()`. Its initial state is more complex, so we'll split it up over multiple lines:

redux/chat_intermediate/public/app.js

```
33 function threadsReducer(state = [
34   {
35     id: '1-fca2',
36     title: 'Buzz Aldrin',
37     messages: messagesReducer(undefined, {}),
38   },
39   {
40     id: '2-be91',
41     title: 'Michael Collins',
42     messages: messagesReducer(undefined, {}),
43   },
44 ], action) {
```



If we weren't initializing our app with two threads already in state, the default argument for `threadsReducer()` would just be `[]`:

```
function threadsReducer(state = [], action) {
  // ...
}
```

Now, let's set the default argument in `messagesReducer()`. While we were initializing one of the threads with a message before, we don't need that anymore:

redux/chat_intermediate/public/app.js

70 `function messagesReducer(state = [], action) {`

We could have just set the `messages` property in the initial state for threads to `[]`. But by calling `messagesReducer()` with `undefined`, we're still delegating initialization responsibility of `messages` to `messagesReducer()`.

We could have done this too:

```
// ...
{
  id: '2-be91',
  title: 'Michael Collins',
  messages: messagesReducer(
    undefined, { type: '@@redux/INIT' }
  ),
},
// ...
```

But passing along a blank action object is sufficient because we won't ever need to switch off of the special type of the initialization action object.

Last, go ahead and remove `initialState` from `app.js`. We don't need it anymore.

Let's kick the tires and make sure things are still working.

Try it out

Save `app.js`. Make sure the server is running, and then load <http://localhost:3000/>⁷⁸ in your browser.

As expected, everything works as before: we can add and delete messages and switch between tabs. The only difference is that we no longer have a default message initialized with the state.

While on the surface the behavior has not changed, we know that under the hood our code is a lot cleaner.

⁷⁸<http://localhost:3000/>

Using `combineReducers()` from redux

The pattern we implemented of combining different reducers to manage distinct parts of the state tree is a common one in Redux. In fact, the `redux` library includes a function `combineReducers()` that can generate a top-level `reducer()` function like the one we wrote by hand.

You can pass `combineReducers()` an object that specifies to which function it should delegate each property of the state object. To see how it works, let's use it now. Let's replace our definition of `reducer()`:

`redux/chat_intermediate/public/app.js`

```
1 const reducer = Redux.combineReducers({
2   activeThreadId: activeThreadIdReducer,
3   threads: threadsReducer,
4 });
```

We're telling `combineReducers()` that our state object has two properties, `activeThreadId` and `threads`. We set those properties to the functions that handle them.

`combineReducers()` returns a reducer function that behaves exactly like the combinatorial `reducer()` function we wrote by hand.



A common pattern is to have the name of the reducer function match that of the property it manages. If we renamed `activeThreadIdReducer()` to `activeThreadId()` and `threadsReducer()` to `threads()`, we could then use ES6's shorthand notation for maximum terseness:

```
const reducer = Redux.combineReducers({
  activeThreadId,
  threads,
});
```

We append `Reducer` to the names of all our reducer functions here because our app is entirely contained in one file. When a Redux app reaches a certain size, it usually makes sense to break out the reducer functions into their own file, like `reducers.js`. At that point, appending `Reducer` to each function name is no longer necessary and you could apply this shorthand.

Next up

After adding complexity to the app and its state with the introduction of threads, we did some significant refactoring with our reducer logic. At first, our single reducer function was managing

many different parts of the state tree and was duplicating logic. With the introduction of reducer composition, we managed to break all our state management apart into smaller pieces.

Our app now neatly isolates responsibility. Not only does this make the code easier to read, but it sets us up for scale.

If we wanted to add a new action to the system that deals with messages, we wouldn't have to worry about logic around managing threads. We'd re-use the same code path as `ADD_MESSAGE` and `DELETE_MESSAGE` in `threadsReducer()` and write all action-specific logic in `messagesReducer()`.

If we wanted to add an entirely new piece of state to the system — like a notifications panel — the management of that state would be entirely isolated to its own function. We wouldn't have to tread lightly over existing code.

There's an opportunity to refactor our React components as well. We'll be focusing on our organization of React components in the next chapter.

Using Presentational and Container Components with Redux

In the last chapter, we added complexity to both the state as well as the view-layer of our application. To support threads in our app, we nested message objects inside of thread objects in our state tree. By using reducer composition, we were able to break up the management of our more complex state tree into smaller parts.

We added a new React component to support our threaded model, `ThreadTabs`, which lets the user switch between threads on the view. We also added some complexity to existing components.

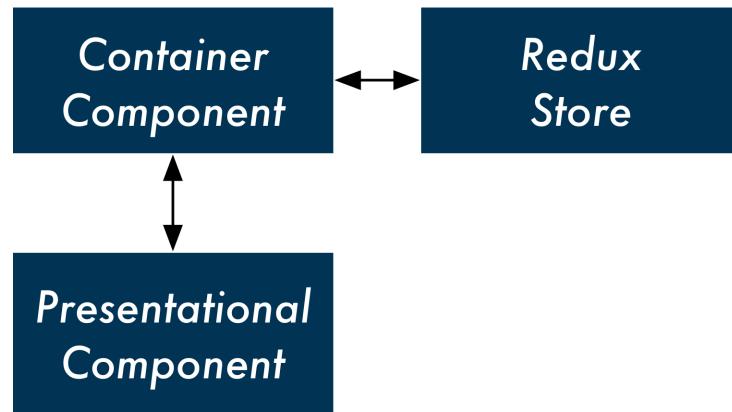
At the moment, we have four React components in our app. Every React component interacts directly with the Redux store. App subscribes to the store and uses `getState()` to read the state and passes down this state as props to its children. Child components dispatch actions directly to the store.

In this chapter, we'll explore a new paradigm for organizing our React components. We can divide up our React components into two types: **presentational components** and **container components**. We'll see how doing so limits knowledge of our Redux store to container components and provides us with flexible and re-usable presentational components.

Presentational and container components

In React, a **presentational component** is a component that *just renders HTML*. The component's only function is presentational markup. In a Redux-powered app, a presentational component does not interact with the Redux store.

The presentational component accepts props from a **container component**. The container component specifies the data a presentational component should render. The container component also specifies behavior. If the presentational component has any interactivity — like a button — it calls a prop-function given to it by the container component. The container component is the one to dispatch an action to the Redux store:



Take a look at the ThreadTabs component:

redux/chat_intermediate/public/app.js

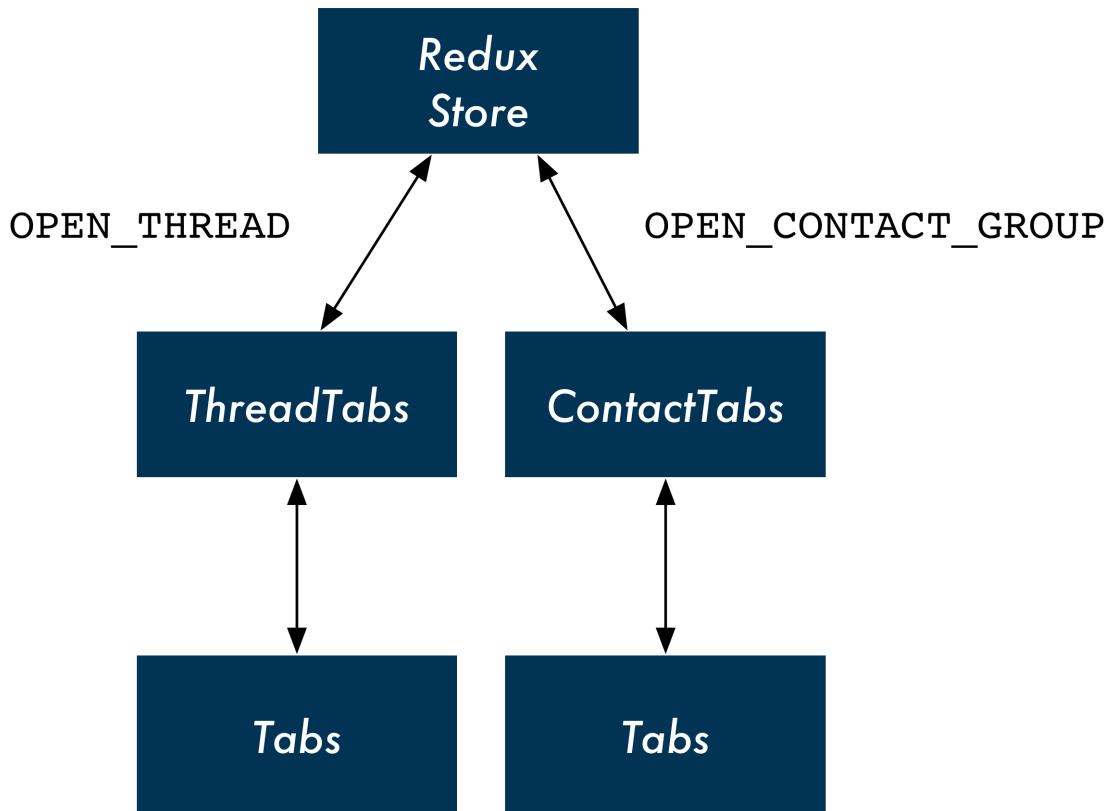
```
122 const ThreadTabs = React.createClass({
123   handleClick: function (id) {
124     store.dispatch({
125       type: 'OPEN_THREAD',
126       id: id,
127     });
128   },
129   render: function () {
130     const tabs = this.props.tabs.map((tab, index) => (
131       <div
132         key={index}
133         className={tab.active ? 'active item' : 'item'}
134         onClick={() => this.handleClick(tab.id)}
135       >
136         {tab.title}
137       </div>
138     ));
139     return (
140       <div className='ui top attached tabular menu'>
141         {tabs}
142       </div>
143     );
144   },
145});
```

At the moment, this component both renders HTML (the text field input) and communicates with the store. It dispatches the OPEN_THREAD action whenever a tab is clicked.

But what if we wanted to have *another* set of tabs in our app? This other set of tabs would probably have to dispatch another type of action. So we'd have to write an entirely different component even though the HTML it renders would be the same.

What if we instead made a generic tabs component, say `Tabs`? This presentational component would not specify what happens when the user clicks a tab. Instead, we could wrap it in a container component wherever we want this particular markup in our app. That container component could then specify what action to take by dispatching to the store.

We'll call our container component `ThreadTabs`. It will do all of the communicating with the store and let `Tabs` handle the markup. In the future, if we wanted to use tabs elsewhere — say, in a “contacts” view that has a tab for each group of contacts — we could re-use our presentational component:



Splitting up `ThreadTabs`

We'll split up `ThreadTabs` by first writing the presentational component `Tabs`. This component will only be concerned with rendering the HTML — the array of horizontal tabs. It will also expect a prop, `onClick`. The presentational component will allow its container component to specify whatever behavior it wants when a tab is clicked.

Let's add Tabs to `app.js` now. Write it above the current `ThreadTab` component. The JSX for the HTML markup is the same as before:

`redux/chat_intermediate/public/app.js`

```

122 const Tabs = (props) => (
123   <div className='ui top attached tabular menu'>
124     {
125       props.tabs.map((tab, index) => (
126         <div
127           key={index}
128           className={tab.active ? 'active item' : 'item'}
129           onClick={() => props.onClick(tab.id)}
130         >
131           {tab.title}
132         </div>
133       ))
134     }
135   </div>
136 );

```

A unique aspect of our new presentational component is how it's declared. So far, we've been using the `React.createClass()` syntax like this:

```

const App = React.createClass({
  // ...
});

```

React components declared in this manner are wrapped in React's component API. This declaration gives the component all of the React-specific features that we've been using, like lifecycle hooks and state management.

However, as we cover in the "Advanced Components" chapter, React also allows you to declare **stateless components**. Stateless components, like `Tabs`, are just JavaScript functions that return markup. They are not special React objects.

Because `Tabs` does not need any of React's component methods, it can be a stateless component.

In fact, all our presentational components can be stateless components. This reinforces their single responsibility of rendering markup. The syntax is terser. What's more, the React core team recommends using stateless components whenever possible. Because these components are not "dressed up" with any of the capabilities of React component objects, the React team anticipates there will be many performance advantages introduced for stateless components in the near future.

As we can see, the first argument passed in to a stateless component is `props`:

redux/chat_intermediate/public/app.js

```
122 const Tabs = (props) => (
```

Because `Tabs` is not a React component object, it does not have the special property `this.props`. Instead, parents pass props to stateless components as an argument. So we'll access this component's props everywhere using `props` as opposed to `this.props`.



Our map call for `Tabs` is in-line, nested inside of the `div` tag in the function's return value.

You could also put this logic above the function's return statement, like we had before in the `render` function of `ThreadTabs`. It's a matter of stylistic preference.

Our presentational component is ready. Let's see what the container component that uses it looks like. Modify the current `ThreadTabs` component:

redux/chat_intermediate/public/app.js

```
138 const ThreadTabs = React.createClass({
139   render: function () {
140     return (
141       <Tabs
142         tabs={this.props.tabs}
143         onClick={(id) => (
144           store.dispatch({
145             type: 'OPEN_THREAD',
146             id: id,
147           })
148         )}
149       />
150     );
151   },
152 });
```

Although we don't use any of React's component methods, we're still using the `createClass` syntax as opposed to declaring a stateless component. We'll see why in a moment.

Our container component specifies the props and behavior for our presentational component. We set the prop `tabs` to `this.props.tabs`, specified by `App`. Next, we set the prop `onClick` to a function that calls `store.dispatch()`. We expect `Tabs` to pass the `id` of the clicked tab to this function.

If we were to test the app out now, we'd be happy to note that our new container/presentational component combination is working.

However, there's one odd thing about `ThreadTabs`: It sends actions to the store *directly* with `dispatch`, yet at the moment it's reading from the store *indirectly* through props (through `this.props.tabs`). `App` is the one reading from the store and this data trickles down to `ThreadTabs`. But if `ThreadTabs` is dispatching directly to the store, is this indirection for reading from the store necessary?

Instead, we can have all of our container components be responsible for both sending actions to the store and reading from it.

In order to achieve this with `ThreadTabs`, we can subscribe directly to the store in `componentDidMount`, the same way that `App` does:

`redux/chat_intermediate/public/app.js`

```
138 const ThreadTabs = React.createClass({
139   componentDidMount: function () {
140     store.subscribe(() => this.forceUpdate());
141   },

```

Then, inside of `render`, we can read `state.threads` directly from the store with `getState()`. We'll generate `tabs` here using the same logic that we used in `App`:

`redux/chat_intermediate/public/app.js`

```
142   render: function () {
143     const state = store.getState();
144
145     const tabs = state.threads.map(t => (
146       {
147         title: t.title,
148         active: t.id === state.activeThreadId,
149         id: t.id,
150       }
151     ));

```

Now we don't need to read from `this.props` at all. We pass `Tabs` the `tabs` variable that we created:

redux/chat_intermediate/public/app.js

```

153     return (
154       <Tabs
155         tabs={tabs}
156         onClick={(id) => (
157           store.dispatch({
158             type: 'OPEN_THREAD',
159             id: id,
160           })
161         )}
162       />
163     );

```

Our Tabs component is purely presentational. It specifies no behavior of its own and could be dropped-in anywhere in the app.

The ThreadTabs component is a container component. It renders no markup. Instead, it interfaces with the store and specifies which presentational component to render. The container component is the connector of the store to the presentational component.

Our presentational and container component combination, in full:

redux/chat_intermediate/public/app.js

```

122 const Tabs = (props) => (
123   <div className='ui top attached tabular menu'>
124     {
125       props.tabs.map((tab, index) => (
126         <div
127           key={index}
128           className={tab.active ? 'active item' : 'item'}
129           onClick={() => props.onClick(tab.id)}
130         >
131           {tab.title}
132         </div>
133       ))
134     }
135   </div>
136 );
137
138 const ThreadTabs = React.createClass({
139   componentDidMount: function () {
140     store.subscribe(() => this.forceUpdate());

```

```

141  },
142  render: function () {
143    const state = store.getState();
144
145    const tabs = state.threads.map(t => (
146      {
147        title: t.title,
148        active: t.id === state.activeThreadId,
149        id: t.id,
150      }
151    ));
152
153    return (
154      <Tabs
155        tabs={tabs}
156        onClick={(id) => (
157          store.dispatch({
158            type: 'OPEN_THREAD',
159            id: id,
160          })
161        )}
162      />
163    );
164  },
165 });

```

In addition to the ability to re-use our presentational component elsewhere in the app, this paradigm gives us another significant benefit: We've de-coupled our presentational view code entirely from our state and its actions. As we'll see, this approach isolates all knowledge of Redux and our store to our app's container components. This minimizes the switching costs in the future. If we wanted to move our app to another state management paradigm, we wouldn't need to touch any of our app's presentational components.

Splitting up Thread

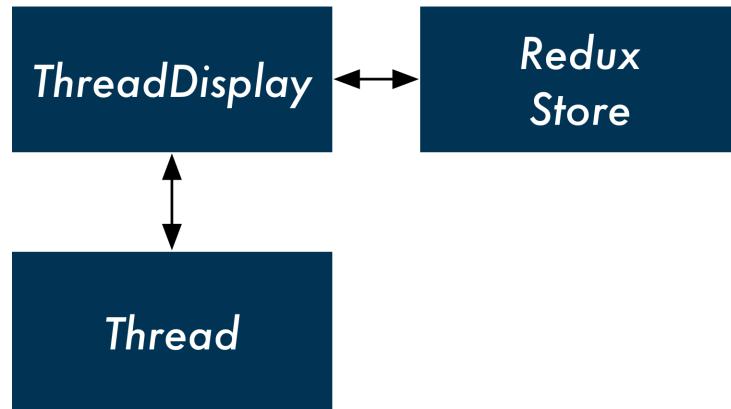
Let's continue refactoring with our new design pattern.

`Thread` receives the `thread` as a prop and contains all the markup for rendering the messages inside of that thread as well as `MessageInput`. The component will dispatch to the store a `DELETE_MESSAGE` action if a message is clicked.

Part of rendering the view for a thread involves rendering the view for its messages. We could have separate container and presentational components for threads and messages. In this setup, the presentational component for a thread would render the container component for a message.

But because we don't anticipate ever rendering a list of messages *outside* of a thread, it's reasonable to just have the container component for the thread also manage the presentational component for a message.

We can have one container component, `ThreadDisplay`. This container component will render the presentational component `Thread`:

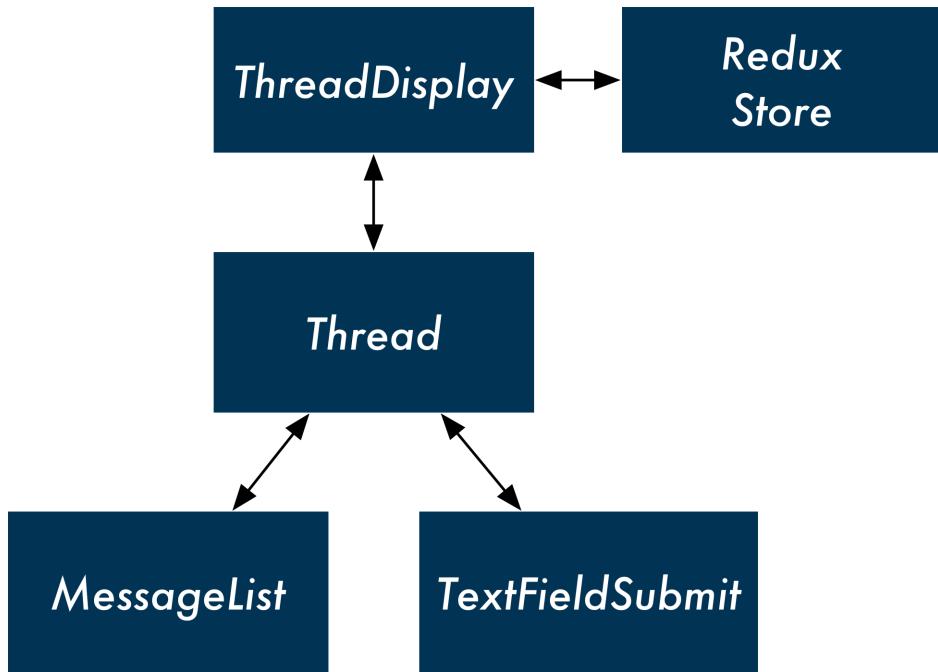


For the list of messages, we can have `Thread` render another presentational component, `MessageList`.

But what about the component `MessageInput`? Like our previous version of `ThreadTabs`, the component contains two responsibilities. The component renders markup, a single text field with a submit button. In addition, it specifies the behavior for what should happen when the form is submitted.

We could, instead, just have a generic presentational component, `TextFieldSubmit` only renders markup and allows its parent to specify what happens when the text field is submitted. `ThreadDisplay`, through `Thread`, could control the behavior of this text field.

With this design, we'd have one container component for a thread up top. The presentational component `Thread` would be a composite of two child presentational components, `MessageList` and `TextFieldSubmit`:



Let's first rename our current `Thread` component to `ThreadDisplay` to avoid confusion:

```
// Rename from `Thread`
const ThreadDisplay = React.createClass({
  // ...
});
```

We'll begin at the bottom, writing the presentational components `TextFieldSubmit` and `MessageList`. We'll work our way up to `Thread` and then `ThreadDisplay`.

`TextFieldSubmit`

Like with `ThreadTabs`, `MessageInput` has two distinct roles: the component both renders the HTML for an input field but also specifies what the behavior around submitting that input field should be (dispatching `ADD_MESSAGE`).

If we remove the dispatch call from `MessageInput`, we'd be left with a generic component that just rendered markup: a text field with an adjacent submit button. The presentational component will allow its container component to specify whatever behavior it wants when the input field is submitted.

Write `TextFieldSubmit` above the current `MessageInput` component. The JSX for the HTML markup is the same as before, but we make a couple of tweaks to the JavaScript:

redux/chat_intermediate/public/app.js

```
167 const TextFieldSubmit = (props) => {
168   let input;
169
170   return (
171     <div className='ui input'>
172       <input
173         ref={node => input = node}
174         type='text'
175       >
176       </input>
177       <button
178         onClick={() => {
179           props.onSubmit(input.value);
180           input.value = '';
181         }}
182         className='ui primary button'
183         type='submit'
184       >
185         Submit
186       </button>
187     </div>
188   );
189 };
```

TextFieldSubmit is a stateless component. It expects one prop, `onSubmit`.

A peculiarity is the `ref` attribute of `input`:

redux/chat_intermediate/public/app.js

```
172   <input
173     ref={node => input = node}
174     type='text'
175   >
```

Just like with `this.props`, in a stateless component we don't have the special property `this.refs`. But we need some way to read whatever the user has typed into the input field.

So far, we've always set the `ref` attribute on an HTML element to a string. React would then make the `ref` available to us in `this.refs`.

In React, if `ref` is set to a function, React will invoke that function after the component is mounted. The argument it supplies to that function is a reference to the DOM node of that element.

We're implementing a common pattern for presentational components here. We set `ref` to a function. This function accepts the argument `node` which is a reference to the `input` element. We're then "saving it for later." We declared a variable at the top of the function, `input`. By setting `input = node`, we can access the DOM node for our input field anywhere else in our render function.

This is exactly what we do in the `onClick` handler:

redux/chat_intermediate/public/app.js

```
178     onClick={() => {
179       props.onSubmit(input.value);
180       input.value = '';
181     }}
```

`onClick` is set to a function that calls `props.onSubmit()`. This is the prop-function that will be specified by the container component that parents `TextFieldSubmit`.

Because we set the variable `input` to the DOM node of our `input` tag, we can read that field here with `input.value`.

We don't need `MessageInput` anymore. We're going to have `ThreadDisplay` be the container for `TextFieldSubmit`.

As such, go ahead and remove the `MessageInput` component.

MessageList

The `MessageList` component will accept two props: `messages` and `onClick`. As before, this presentational component will not specify any behavior. As a stateless component, it will only render HTML.

Write it below `TextFieldSubmit` and above the `ThreadDisplay` component in `app.js`:

redux/chat_intermediate/public/app.js

```
191 const MessageList = (props) => (
192   <div className='ui comments'>
193     {
194       props.messages.map((m, index) => (
195         <div
196           className='comment'
197           key={index}
198           onClick={() => props.onClick(m.id)}
199         >
```

```
200      <div className='text'>
201          {m.text}
202          <span className='metadata'>@{m.timestamp}</span>
203      </div>
204  </div>
205  ))
206 }
207 </div>
208 );
```

The `map` that we perform over `props.messages` is the same logic we had previously in `Thread`. We perform it in-line, nested inside of the `div` tag which is responsible for styling. The three changes:

- We perform the map over `props.messages` as opposed to `this.props.threads`
- The `onClick` attribute is now set to `props.onClick`
- For brevity, we're using the variable `m` in place of `message`



You could optionally break this presentational component down further by adding another component, `Message`. The markup for each message is still simple enough that we opted not to do this just yet here.

Thread

We have two presentational components related to displaying a thread. One is `MessageList`, which renders all of the messages in that thread. The other is `TextFieldSubmit`, a generic text field entry that we're going to have submit new messages to the thread.

We're collecting these two presentational components under `Thread`, another presentational component. The container component `ThreadDisplay` will render `Thread` which in turn will render `MessageList` and `TextFieldSubmit`.

We anticipate that `ThreadDisplay` will pass `Thread` three props:

- `thread`: The thread itself
- `onMessageClick`: The message click handler
- `onMessageSubmit`: The text field submit handler

We'll have `Thread` pass along the appropriate props to each of its child presentational components:

redux/chat_intermediate/public/app.js

```

210 const Thread = (props) => (
211   <div className='ui center aligned basic segment'>
212     <MessageList
213       messages={props.thread.messages}
214       onClick={props.onMessageClick}
215     />
216     <TextFieldSubmit
217       onSubmit={props.onMessageSubmit}
218     />
219   </div>
220 );

```

ThreadDisplay

`ThreadDisplay` (previously named `Thread`) is our container component. Like with our two previous container components, it will subscribe to the store. The component will be responsible for both reading from the store and dispatching actions to it.

First, subscribe to the store in `componentDidMount`:

redux/chat_intermediate/public/app.js

```

222 const ThreadDisplay = React.createClass({
223   componentDidMount: function () {
224     store.subscribe(() => this.forceUpdate());
225   },

```

`ThreadDisplay` will read from the store directly to get the active thread. The container component will then render `Thread`, passing in the props `thread` and `onMessageClick`.

Inside of `render`, we'll use the same logic we had used in `App` to grab the active thread:

redux/chat_intermediate/public/app.js

```

226   render: function () {
227     const state = store.getState();
228     const activeThreadId = state.activeThreadId;
229     const activeThread = state.threads.find(
230       t => t.id === activeThreadId
231     );

```

We return `Thread`, passing it `thread` as well as specifying its behavior for `onMessageClick` and `onMessageSubmit`:

redux/chat_intermediate/public/app.js

```

233     return (
234       <Thread
235         thread={activeThread}
236         onMessageClick={(id) => (
237           store.dispatch({
238             type: 'DELETE_MESSAGE',
239             id: id,
240           })
241         )}
242         onMessageSubmit={(text) => (
243           store.dispatch({
244             type: 'ADD_MESSAGE',
245             text: text,
246             threadId: activeThreadId,
247           })
248         )}
249       />
250     );

```

Our container component `ThreadDisplay`, in full:

redux/chat_intermediate/public/app.js

```

222 const ThreadDisplay = React.createClass({
223   componentDidMount: function () {
224     store.subscribe(() => this.forceUpdate());
225   },
226   render: function () {
227     const state = store.getState();
228     const activeThreadId = state.activeThreadId;
229     const activeThread = state.threads.find(
230       t => t.id === activeThreadId
231     );
232
233     return (
234       <Thread
235         thread={activeThread}
236         onMessageClick={(id) => (
237           store.dispatch({
238             type: 'DELETE_MESSAGE',
239             id: id,

```

```

240      })
241    )}
242    onMessageSubmit={(text) => (
243      store.dispatch({
244        type: 'ADD_MESSAGE',
245        text: text,
246        threadId: activeThreadId,
247      })
248    )}
249  />
250 );
251 },
252 });

```

We've split up all of our view components into container and presentational components. Our two container components are communicating directly with the store, performing both reads (`getState()`) and sending actions (`dispatch()`).

Because of this, we actually don't need `App` to talk to the store at all. The `render` function for `App` right now reads from the store and then sends props down to its children. Now that its children are communicating with the store, it's no longer necessary that `App` supply them with any props.

Removing store from App

Because our container components are now interfacing with the store themselves, we can remove all communication with the store from `App`. In fact, we can just turn `App` into a stateless component:

`redux/chat_intermediate/public/app.js`

```

95 const App = () => (
96   <div className='ui segment'>
97     // `Thread` changed to `ThreadDisplay` below
98     <ThreadTabs />
99     <ThreadDisplay />
100   </div>
101 );

```

Quite a contrast to the top-level component of some of our previous apps, no?

Due to a combination of our new container and presentational component paradigm and a Redux state manager, the top-level component for this app is just specifying what container components to include on the page. All of the responsibility for reading and writing to state has been pushed down to each one of our container components.

Because we're not dispatching actions directly from our leaf components, we've isolated all knowledge of the Redux store to our container components. We're free to re-use our presentational components in other contexts within our app. What's more, if we wanted to switch state management paradigms from Redux to something else, we would only need to modify our container components.

Our container components all look pretty similar. They subscribe to the store and then map state and actions to props on a presentational component. In the next section, we'll explore an option for reducing some of the ceremony around writing container components.

But for now, let's pause briefly to verify that everything still works as before.

Try it out

Save `app.js`. While we've made some big architectural changes to our React components, viewing the app at <http://localhost:3000/>⁷⁹ everything is working as before.

Generating containers with react-redux

Looking at our two container components (`ThreadTabs` and `ThreadDisplay`), they have similar behavior:

- They subscribe to the store in `componentDidMount`.
- They might have some logic to massage data from state into a format fit as a prop for the presentational component (like `tabs` in `ThreadTabs`).
- They map actions on the presentational component (like click events) to functions that dispatch to the store.

Because container components rely on presentational components to render markup, they just contain "glue" code between the store and presentational components.

A popular library, `react-redux`, gives us a couple conveniences when writing React apps that use a Redux store. The primary convenience is its `connect()` function.

The `connect()` function in `react-redux` **generates container components**. For each presentational component, we can write functions that specify how state should map to props and how events should map to dispatches.

Let's see what this looks like in action.

⁷⁹<http://localhost:3000/>

The Provider component

Before we can use `connect()` to generate container components, we need to make a small addition to our app.

Right now, our containers are referencing the `store` variable directly. This works because we declare this variable inside the same file as our components.

In order for `connect()` to be able to generate container components, it needs some canonical mechanism for containers to access the Redux store. The function can't rely on the `store` variable being declared and available in the same file.

To solve this, the `react-redux` library supplies a special `Provider` component. You can wrap your top-level component in `Provider`. `Provider` will then make the store available to all components via React's context feature.

When we use `connect()` to generate container components, those container components will assume that the `store` is available to them via context.



Context is a React feature that you can use to make certain data available to all React components. We talk about context in the “Advanced Component Configuration” chapter.

With props, data is explicitly passed down the component hierarchy. A parent must specify what data is available to its children.

Context allows you to make data available *implicitly* to all components in a tree. Any component can “opt-in” to receiving context and that component’s parent doesn’t need to do anything.

While we cover context elsewhere in the book, it is a rarely used feature in React. The React core team actively discourages its use, except in special circumstances.

Wrapping App in Provider

Inside of `index.html`, we’re already including the `react-redux` library:

`redux/chat_intermediate/public/app.js`

18

```
<script src='vendor/react-redux.js'></script>
```

Therefore, we already have the `ReactRedux` module available to us in `app.js`.

To allow our generated container components to access the store through context, we need to wrap `App` in the special `Provider` component supplied by the `react-redux` library:

redux/chat_intermediate/public/app.js

```
234 ReactDOM.render(  
235   <ReactRedux.Provider store={store}>  
236     <App />  
237     </ReactRedux.Provider>,  
238     document.getElementById('content')  
239 );
```

Provider expects to receive the prop `store`. The store is now available anywhere in our component hierarchy under the context variable `store`.

Using `connect()` to generate `ThreadTabs`

The `ThreadTabs` component connects the presentational component `Tabs` with our Redux store. It does so by:

- Subscribing to the store in `componentDidMount`
- Creating a `tabs` variable based on the store's `threads` property and using that for the prop `tabs` on `Tabs`
- Setting the `onClick` prop on `Tabs` to a function that dispatches an `OPEN_THREAD` action

We can use `connect()` to generate this component.

We need to pass two arguments to `connect()`. The first will be a function that maps the state to the props of `Tabs`. The second will be a function that maps dispatch calls to the component's props.

We'll see how this works by implementing it.

Mapping state to props

At the moment, we perform our “mapping” between the state and the props for `Tabs` inside of the render function for `ThreadTabs` by creating the `tabs` variable.

We'll write a function that `connect()` will use to perform this same operation. We'll call this function `mapStateToPropsToProps()`.

Whenever the state is changed, this function will be invoked to determine how to map the new state to the props for `Tabs`.

Declare the function above `ThreadTabs` in `app.js`. The function expects to receive the state as an argument:

redux/chat_intermediate/public/app.js

```
118 const mapStateToProps = (state) => {
```

We can copy and paste the logic that we had in ThreadTabs to produce the variable tabs, based on state.threads:

redux/chat_intermediate/public/app.js

```
119 const tabs = state.threads.map(t => (
120   {
121     title: t.title,
122     active: t.id === state.activeThreadId,
123     id: t.id,
124   }
125 ));
```

Our state-to-props mapping function needs to return an object. The properties on this object are the prop names for Tabs. Because the prop is called tabs and the variable we're setting it to is also tabs, we can use the ES6 object shorthand:

redux/chat_intermediate/public/app.js

```
127 return {
128   tabs,
129 };
130 };
```

Our mapStateToProps() function, in full:

redux/chat_intermediate/public/app.js

```
118 const mapStateToProps = (state) => {
119   const tabs = state.threads.map(t => (
120     {
121       title: t.title,
122       active: t.id === state.activeThreadId,
123       id: t.id,
124     }
125   ));
126
127   return {
128     tabs,
129   };
130 };
```

This function encapsulates the logic that was previously in `ThreadTabs`, describing how the state maps to the prop `tabs` for `Tabs`. Now we have to do the same for the prop `onClick`, which maps to a dispatch call.

Mapping dispatches to props

We'll declare this function below `mapStateToProps()`. We'll call it `mapDispatchToProps()`:

`redux/chat_intermediate/public/app.js`

```
132 const mapDispatchToProps = (dispatch) => (
```

We'll pass this function second to `connect()`. It will be invoked on setup with `dispatch` passed in as an argument.

Like with `mapStateToProps()`, we'll return an object that maps the prop `onClick` to the function that will perform the dispatching. This function is identical to the one that `ThreadTabs` previously specified:

`redux/chat_intermediate/public/app.js`

```
132 const mapDispatchToProps = (dispatch) => (
133   {
134     onClick: (id) => (
135       dispatch({
136         type: 'OPEN_THREAD',
137         id: id,
138       })
139     ),
140   }
141 );
```

We now have a function that maps the store's state to the prop `tabs` on `Tabs` and another that maps the prop `onClick` to a function that dispatches `OPEN_THREAD`.

We can now replace our `ThreadTabs` component by using `connect()`. Delete the entire `ThreadTabs` component currently in `app.js`.

The first argument to `connect()` is the function that maps the state to props. The second argument is the function that maps props to dispatch functions. `connect()` returns a function that we will immediately invoke with the presentational component we'd like to "connect" to the store with our container component:

```
// Signature of `connect()`  
// (Note this is partial, we see the full signature later)  
connect(  
  mapStateToProps(state),  
  mapDispatchToProps(dispatch),  
)  
(PresentationalComponent)
```

Let's use connect() to create ThreadTabs:

`redux/chat_intermediate/public/app.js`

```
143 const ThreadTabs = ReactRedux.connect(  
144   mapStateToProps,  
145   mapDispatchToProps  
146 )(Tabs);
```

On the surface it may not look like it, but ThreadTabs is a React container component, not too unlike the one we had before.

Using connect() to generate ThreadDisplay

The next component we'll generate with connect() is ThreadDisplay.

The container component specifies three props on Thread:

- `thread`
- `onMessageClick`
- `onMessageSubmit`

State to props

We'll call this mapping function `mapStateToProps()`. It maps one prop:

- `thread`: maps to the active thread in state

Dispatch to props

We'll call this mapping function `mapDispatchToProps()`. It maps two props:

- `onMessageClick`: maps to a function that dispatches `DELETE_MESSAGE`
- `onMessageSubmit`: maps to a function that dispatches `ADD_MESSAGE`

`mapStateToProps()`

We'll write our state-to-props glue function above ThreadDisplay in `app.js`.

`mapStateToProps()` accepts the argument `state`. We have it return an object that maps the `thread` property to the active thread in state:

redux/chat_intermediate/public/app.js

```
203 const mapStateToProps = (state) => (
204   {
205     thread: state.threads.find(
206       t => t.id === state.activeThreadId
207     ),
208   }
209 );
```

This follows from the same logic that `ThreadDisplay` used to set the `thread` property of `Thread`.

`mapDispatchToProps()`

Below `mapStateToProps()`, we'll write our dispatch-to-props glue function.

The first dispatch prop we'll write is that for `onMessageClick`. This function accepts an `id` and dispatches a `DELETE_MESSAGE` action. Again, this logic matches that of `ThreadDisplay`:

redux/chat_intermediate/public/app.js

```
211 const mapDispatchToProps = (dispatch) => (
212   {
213     onMessageClick: (id) => (
214       dispatch({
215         type: 'DELETE_MESSAGE',
216         id: id,
217       })
218     ),
219   },
220 );
```

Next, we need to define the dispatch function for `onMessageSubmit`.

As you recall, inside of `ThreadDisplay`, this function dispatched an `ADD_MESSAGE` action like this:

```
store.dispatch({
  type: 'ADD_MESSAGE',
  text: text,
  threadId: activeThreadId,
})
```

`connect()` will not pass our dispatch-to-props function the state. How do we get the active thread's `id`, then?

We might be tempted to try something like this:

```
store.dispatch({  
  type: 'ADD_MESSAGE',  
  text: text,  
  // just read `activeThreadId` from the store directly  
  threadId: store.getState().activeThreadId,  
})
```

For the purposes of our app, this will work just fine. `store` is defined in this file, so we could just read from it directly.

But, it's preferable that the mapping functions that you pass to `connect()` **do not access the store directly**.

Why so?

We're about to replace our declaration of `ThreadDisplay` with a container component generated by `connect()`. Powerfully, after we've done so, the only reference to the store from our React components will be right here:

```
// Inside `ReactDOM.render()`  
<ReactRedux.Provider store={store}>
```

Isolating the reference to `store` to just one location has two huge benefits.

The first benefit is one we covered earlier when we discussed container components. The less references we have to `store`, the less work we'd have to do if we wanted to move from Redux to some other state management paradigm. Our mapping functions are just JavaScript functions and could conceivably perform mapping for some other type of store, so long as the API for the store was somewhat similar to that of our Redux store.

The more immediate benefit is for testing. When writing tests for a React app, you might want to inject a fake store into your app. With a fake store, you could specify what it should return for each spec or assert that certain methods on that store are called.

By passing the `store` as a prop to `ReactRedux.Provider` and not referencing it directly anywhere else in the app, we could easily swap in a mock store during tests.

So, we need the `id` of the thread we're displaying in order to dispatch our `ADD_MESSAGE` action. But we don't have access to this property inside our dispatch-to-props function.

`connect()` allows you to pass in a *third* function, what it calls `mergeProps`. So, in full, the three functions you can pass to `connect()`:

```
// Full function signature of `connect()`
connect(
  mapStateToProps(state, [ownProps]),
  mapDispatchToProps(dispatch, [ownProps]),
  mergeProps(stateProps, dispatchProps, [ownProps])
)
```



In `connect()`, `ownProps` refers to the props set on the *container component* that we're generating. In this instance, these would be any props that `App` sets on either of our container components, `ThreadTabs` or `ThreadDisplay`.

Accepting and using `ownProps` is optional. Because `App` does not specify any props on our container components, none of our mapping functions use this second argument.

`mergeProps` is called with two arguments: `stateProps` and `dispatchProps`. These are just the objects that are returned by `mapStateToProps` and `mapDispatchToProps`.

So we can pass `connect()` a third function, `mergeThreadProps()`. This function will be invoked with two arguments:

- The object we return in `mapStateToProps()`
- The object we return in `mapDispatchToProps()`

`connect()` will use the object returned by `mergeThreadProps()` as the final object to determine the props for `Thread`.

In sequence, `connect()` will do the following:

1. Call `mapStateToProps()` with `state`
2. Call `mapDispatchToProps()` with `dispatch`
3. Call `mergeThreadProps()` with the results of the two previous map functions (`stateProps` and `dispatchProps`)
4. Use the object returned by `mergeThreadProps()` to set the props on `Thread`

Inside of `mergeThreadProps()`, we'll need access to two items to create our `ADD_MESSAGE` dispatch function:

- The `id` of the thread
- The `dispatch` function itself

We'll get the `id` of the thread via `stateProps`, as that object has the full thread object under `thread`.

To get access to `dispatch`, we can pass it along in `mapDispatchToThreadProps()`.

As such, our `mapDispatchToThreadProps()` function will define two properties:

- `onMessageClick`
- `dispatch`

The dispatch-to-props function, in full:

`redux/chat_intermediate/public/app.js`

```
211 const mapDispatchToThreadProps = (dispatch) => (
212   {
213     onMessageClick: (id) => (
214       dispatch({
215         type: 'DELETE_MESSAGE',
216         id: id,
217       })
218     ),
219     dispatch: dispatch,
220   }
221 );
```

Now we'll define our final mapping function. Again, this "merging" function will be passed the results of our state-to-props mapping function and our dispatch-to-props mapping function. The object it returns is the one that `connect()` will use to bind the props of `Thread`.

We'll declare this function below `mapDispatchToThreadProps()`. Our merging function receives two arguments:

`redux/chat_intermediate/public/app.js`

```
223 const mergeThreadProps = (stateProps, dispatchProps) => (
```

We want to create a new object that contains:

- All the properties from `stateProps`
- All the properties from `dispatchProps`
- An additional property, `onMessageSubmit`

Let's see what this looks like:

redux/chat_intermediate/public/app.js

```
223 const mergeThreadProps = (stateProps, dispatchProps) => (
224   {
225     ...stateProps,
226     ...dispatchProps,
227     onMessageSubmit: (text) => (
228       dispatchProps.dispatch({
229         type: 'ADD_MESSAGE',
230         text: text,
231         threadId: stateProps.thread.id,
232       })
233     ),
234   }
235 );
```

We copy both `stateProps` and `dispatchProps` over to our new object using the spread operator (`...`).

`onMessageSubmit` dispatches the same `ADD_MESSAGE` action that `ThreadDisplay` previously dispatched. Note that we're grabbing the `dispatch` function from `dispatchProps`:

redux/chat_intermediate/public/app.js

```
228   dispatchProps.dispatch({
```

And then, we're grabbing the thread's id from `stateProps`:

redux/chat_intermediate/public/app.js

```
231     threadId: stateProps.thread.id,
```

With our two mapping functions and one merge function prepared, we can now generate `ThreadDisplay` with `ReactRedux.connect()`.

We'll declare `ThreadDisplay` below `mergeThreadProps()`:

redux/chat_intermediate/public/app.js

```
237 const ThreadDisplay = ReactRedux.connect(  
238   mapStateToProps,  
239   mapDispatchToProps,  
240   mergeThreadProps  
241 )(Thread);
```

Be sure to remove the old declaration of the `ThreadDisplay` component from `app.js`.

Using the `mergeProps` argument of `connect()` feels like a bit of a workaround. This is because the parameters for using `connect()` are quite strict. This is by design. The library enforces this usage for both performance reasons and to prevent a few possible developer mistakes.

However, with our `merge` function, we got `connect()` to generate a `ThreadDisplay` component as desired. We removed some of the boilerplate around our container components. And, we've isolated the connection between Redux and React to a single area — as a prop for `ReactRedux.Provider`.

Let's verify everything is working properly.

Try it out

Make sure the server is running. Navigate to <http://localhost:3000>⁸⁰ and observe that all of the functionality of the app is working.

Action creators

Right now, in every instance where we want to dispatch an action, we declare an action object of a certain type as well as its required properties. For example, for the `DELETE_MESSAGE` action:

```
dispatch({  
  type: 'DELETE_MESSAGE',  
  id: id,  
})
```

In the current iteration of the app, we only dispatch each type of action from a single location. As Redux apps grow, it's common for the same action to be dispatched from multiple locations.

A popular pattern is to use **action creators** to create the action objects. An action creator is a function that returns an action object. An action creator for our `DELETE_MESSAGE` action would look like this:

⁸⁰<http://localhost:3000>

```
// Example action creator for `DELETE_MESSAGE`  
function deleteMessage(id) {  
  return {  
    type: 'DELETE_MESSAGE',  
    id: id,  
  };  
}
```

Then, anywhere in our app, if we wanted to dispatch a `DELETE_MESSAGE` action, we could just use our action creator:

```
dispatch(deleteMessage(id));
```

It's a light abstraction that hides the action's type as well as its property names from React components. More importantly, using action creators enables certain advanced patterns, like coupling an API request with an action dispatch.

Let's swap out our action objects and use action creators instead.

First, we'll write our action creators. Let's declare them below the line where we initialize the store with `createStore()`.

We already saw what the `deleteMessage()` action creator looks like. The action creator is a function that accepts the `id` of the message to be deleted. It then returns an object of type `DELETE_MESSAGE`:

`redux/chat_intermediate/public/app.js`

```
95 function deleteMessage(id) {  
96   return {  
97     type: 'DELETE_MESSAGE',  
98     id: id,  
99   };  
100 }
```

The action creator `addMessage()` expects both a `text` and `threadId` argument:

redux/chat_intermediate/public/app.js

```
102 function addMessage(text, threadId) {
103   return {
104     type: 'ADD_MESSAGE',
105     text: text,
106     threadId: threadId,
107   };
108 }
```

And `openThread` expects the id of the thread to open:

redux/chat_intermediate/public/app.js

```
110 function openThread(id) {
111   return {
112     type: 'OPEN_THREAD',
113     id: id,
114   };
115 }
```

We can now work down `app.js` and replace action objects with our new action creators.

Inside `mapDispatchToProps()`:

redux/chat_intermediate/public/app.js

```
154 const mapDispatchToProps = (dispatch) => (
155   {
156     onClick: (id) => (
157       dispatch(openThread(id))
158     ),
159   }
160 );
```

Then `mapStateToProps()`:

redux/chat_intermediate/public/app.js

```
230 const mapDispatchToThreadProps = (dispatch) => (
231   {
232     onMessageClick: (id) => (
233       dispatch(deleteMessage(id))
234     ),
235     dispatch: dispatch,
236   }
237 );
```

And finally `mergeThreadProps()`:

redux/chat_intermediate/public/app.js

```
239 const mergeThreadProps = (stateProps, dispatchProps) => (
240   {
241     ...stateProps,
242     ...dispatchProps,
243     onMessageSubmit: (text) => (
244       dispatchProps.dispatch(
245         addMessage(text, stateProps.thread.id)
246       )
247     ),
248   }
249 );
```

Another benefit of using action creators is that they list out all of the possible actions in our system in one place. We no longer have to infer the shapes of possible actions by hunting through the reducers or React components. This is exacerbated as the number of actions in a system grows.

Conclusion

Over the last three chapters, we've explored the fundamentals of the Redux design pattern. The architecture of our Redux-powered chat app is a stark difference from what it would be if we'd used React's component state.

We don't have to worry about pipelining props through tons of intermediary components, as we define functionality close to leaf components in our container components. Consider the scenario where we wanted to add more data to each thread, like a profile picture of the user. We'd just have to tweak the state-to-props mapping function in our container component to make the profile picture's URL available to our presentational component.

Furthermore, instead of a heavy top-level component that performs all of our state management, we have a series of reducer functions that each handle their own part of our state tree. The advantage here is accentuated if one action should affect multiple parts of the state tree.

Imagine if we introduced a notifications counter to the chat app. The counter indicates the number of unread threads. Every time the user opened a thread, we'd want to decrement this counter.

In the component-state paradigm, this means that the function — say, `handleThreadOpen()` — would not only affect the `activeThreadId` part of the state tree. It would also be responsible for modifying the notifications counter. In this model, single actions that affect multiple parts of the state tree can become cumbersome, quickly.

With Redux, the effect of a given action on each part of the state tree is isolated to each reducer. In our case, we wouldn't have to touch `activeThreadIdReducer()` to incorporate a new notifications counter. The effect of `OPEN_THREAD` on the notifications counter would be isolated to the reducer for that piece of state.

Asynchronicity and server communication

The last concept you'll need to begin composing real-world applications with Redux is how to handle server communication.

Redux does not have an established mechanism for handling asynchronicity built-in. Instead, there are a variety of tools and patterns within the Redux ecosystem for handling asynchronicity.

While asynchronicity in Redux is outside the scope of this book, with the Redux fundamentals covered in the last few chapters you're equipped to integrate these patterns and libraries into your own applications.

The most popular strategy is to use a light piece of middleware called `redux-thunk`⁸¹. Using `redux-thunk`, you can have a `dispatch` call execute a function as opposed to dispatching an action directly to the store. Inside of that function, you can make a network request and dispatch an action when the request finishes.

For a detailed example of using `redux-thunk` to handle asynchronicity, check out this tutorial on the Redux site: <http://redux.js.org/docs/advanced/AsyncActions.html>⁸².

⁸¹<https://github.com/gaearon/redux-thunk>

⁸²<http://redux.js.org/docs/advanced/AsyncActions.html>

Using GraphQL

Over the last few chapters, we've explored how to build React applications that interact with servers using JSON and HTTP APIs. In this chapter, we're going to explore GraphQL, which is a specific API protocol developed by Facebook that is a natural fit in the React ecosystem.

What is GraphQL? Most literally it means "Graph Query Language", which may sound familiar if you've worked with other query languages like SQL. If your server "speaks" GraphQL, then you can send it a GraphQL query string and expect a GraphQL response. We'll dive into the particulars soon, but the features of GraphQL make it particularly well-suited for cross-platform applications and large product teams.

To discover why, let's start with a little show-and-tell.

Your First GraphQL Query

Typically GraphQL queries are sent using HTTP requests, similar to how we sent API requests in earlier chapters; however, there is usually just **one URL endpoint** per-server that handles all GraphQL requests.

[GraphQLHub⁸³](#) is a GraphQL service that we'll use throughout this chapter to learn about GraphQL. Its GraphQL endpoint is `https://www.graphqlhub.com/graphql`, and we issue GraphQL queries using an HTTP POST method. Fire up a terminal and issue this cURL command:



If you'd like to learn more about cURL, checkout [our section on cURL in the Appendix](#)

```
$ curl -H 'Content-Type:application/graphql' -XPOST https://www.graphqlhub.com/graphql?pretty=true -d '{ hn { topStories(limit: 2) { title url } } }'  
{  
  "data": {  
    "hn": {  
      "topStories": [  
        {  
          "title": "Bank of Japan Is an Estimated Top 10 Holder in 90% of the Nikkei 225",  
          "url": "http://www.bloomberg.com/news/articles/2016-04-24/the-tokyo-wh"
```

⁸³<https://www.graphqlhub.com/>

```
ale-is-quietly-buying-up-huge-stakes-in-japan-inc"
},
{
  "title": "Dropbox as a Git Server",
  "url": "http://www.anishathalye.com/2016/04/25/dropbox-as-a-true-git-s\
erver/"
}
]
}
}
}
```

It may take a second to return, but you should see a JSON object describing the title and url of the top stories on [Hacker News⁸⁴](#). Congratulations, you've just executed your first GraphQL query!

Let's break down what happened in that cURL. We first set the Content-Type header to application/graphql - this is how the GraphQLHub server knows that we're sending a GraphQL request, which is a common pattern for many GraphQL servers (we'll see later on in [Writing Your GraphQL Server](#)).

Next we specified a POST to the /graphql?pretty=true endpoint. The /graphql portion is the path, and the pretty query parameters instructs the server to return the data in a human-friendly, indented format (instead of returning the JSON in one line of text).

Finally, the -d argument to our cURL command is how we specify the **body of the POST request**. For GraphQL requests, the body is often a **GraphQL query**. We had to write our request in one line for cURL, but here's what our query looks like when we expand and indent it properly:

```
1 // one line
2 { hn { topStories(limit: 2) { title url } } }
3
4 // expanded
5 {
6   hn {
7     topStories(limit: 2) {
8       title
9       url
10    }
11  }
12 }
```

This is a GraphQL query. On the surface it may look similar to JSON, and they do have a tree structure and nested brackets in common, but there are crucial differences in syntax and function.

⁸⁴<https://news.ycombinator.com>

Notice that the structure of our query is the same structure returned in the JSON response. We specified some properties named `hn`, `topStories`, `title`, and `url`, and the response object has that exact tree structure - there are no extra or missing entries. This is one of the key features of GraphQL: **you request the specific data you want from the server**, and no other data is returned implicitly.

It isn't obvious from this example, but GraphQL not only tracks the properties **available** to query, but the **type** of each property as well ("type" as in number, string, boolean, etc). This GraphQL server knows that `topStories` will be a list of objects consisting of `title` and `url` entries, and that `title` and `url` are strings. The type system is much more powerful than just strings and objects, and really saves time in the long-run as a product grows more complex.

GraphQL Benefits

Now that we've seen a bit of GraphQL in action, you may be wondering, "why anyone would prefer GraphQL over URL-centric APIs such as REST?" At first glance, it seems like strictly more work and setup than traditional protocols - what do we get for the extra effort?

First, our API calls become easier to understand by **declaring the exact data we want from the server**. For a newcomer to a web app codebase, seeing GraphQL queries makes it immediately obvious what data is coming from the server versus what data is derived on the clients.

GraphQL also opens doors for better unit and integration testing: it's easy to mock data on the client, and it's possible to assert that your server GraphQL changes don't break GraphQL queries in client code.

GraphQL is also designed with performance in mind, especially with respect to mobile clients. Specifying only the data needed in each query prevents over-fetching (i.e., where the server retrieves and transmits data that ultimately goes unused by the client). This reduces network traffic and helps to improve speed on size-sensitive mobile environments.

The development experience for traditional JSON APIs is often acceptable at best (and infuriating more often). Most APIs are lacking in documentation, or worse have documentation that is inconsistent with the behavior of the API. APIs can change and it's not immediately obvious (i.e. with compile-time checks) what parts of your application will break. Very few APIs allow for discovery of properties or new endpoints, and usually it occurs in a bespoke mechanism.

GraphQL dramatically improves the developer experience - its type system provides a **living form of self-documentation**, and tooling like GraphiQL (which we play with in this chapter) allow for natural exploration of a GraphQL server.

The screenshot shows a GraphQL query being run against the HackerNewsAPI. The query is:

```

1 # Welcome to GraphQLHub! Type your Gra
2 # explore the "Docs" to the right
3
4 {
5   hn {
6     topStories {
7       url
8       title
9       by {
10      id
11    }
12  }
13}
14
15

```

The results are:

```

{
  "data": {
    "hn": {
      "topStories": [
        {
          "url": "http://www.su-tesla.space/2016/04",
          "title": "Gentoo Tesla - T2 Edition",
          "by": {
            "id": "lelf"
          }
        },
        {
          "url": "https://github.com/Homebrew/brew",
          "title": "Homebrew now sends usage inform",
          "by": {
            "id": "aorth"
          }
        },
        {
          "url": "http://motherboard.vice.com/read/",
          "title": "DARPA Is Looking For The Perfect",
          "by": {
            "id": "mr_golyadkin"
          }
        }
      ]
    }
  }
}

```

On the right, there is a sidebar for the `HackerNewsItem` type with the following fields:

- FIELDS**
- `id: String!`
- `deleted: Boolean`
- `type: ItemType!`
- `by: HackerNewsUser!`
- `time: Int!`
- `timelSO: String!`
- `text: String`
- `dead: Boolean`
- `url: String`
- `score: Int`

Navigating a Schema with GraphiQL

Finally, the declarative nature of GraphQL pairs nicely with the declarative nature of React. A few projects, including the Relay framework from Facebook, are trying to bring the idea of “colocated queries” in React to life. Imagine writing a React component that automatically fetches the data it needs from the server, with no glue code around issuing API calls or tracking the lifecycle of a request.

All of these benefits compound as your team and product grow, which is why it evolved to be the primary way data is exchanged between client and server at Facebook.

GraphQL vs. REST

We’ve mentioned alternative protocols a bit, but let’s compare GraphQL to REST specifically.

One drawback to REST is “endpoint creep.” Imagine you’re building a social network app and start with a `/user/:id/profile` endpoint. At first the data this endpoint returns is the same for every platform and every part of the app, but slowly your product evolves and accumulates more features that fit under the “profile” umbrella. This is problematic for new parts of the app that only need *some* profile data, such as tooltips on a news feed. So you might end up creating something like `/user/:id/profile_short`, which is a restricted version of the full profile.

As you run into more cases where new endpoints are required (imagine a `profile_medium!`), you now duplicate some data with calls to `/profile` and `/profile_short`, possibly wasting server and network resources. It also makes the development experience harder to understand - where does a developer find out what data is returned in each variation? Are your docs up-to-date?

An alternative to creating N endpoints is to add some GraphQL-esque functionality to query parameters, such as `/user/:id/profile?include=user.name,user.id`. This allows clients to specify the data they want, but still lacks many of the features of GraphQL that make such a system

work in the long-run. For example, that sort of API there is still no strong typing information that enables resilient unit testing and long-lived code. This is especially critical for mobile apps, where old binaries might exist in the wild for long periods of time.

Lastly the tooling for developing against and debugging REST APIs is often lack-luster because there are so few commonalities among popular APIs. At the lowest level you have very general purpose tools like cURL and wget, some APIs might support [Swagger⁸⁵](#) or other documentation formats, and at the highest level for specific APIs like Elasticsearch or Facebook's Graph API you may find bespoke utilities. GraphQL's type system supports introspection (in other words, you can use GraphQL itself to discover information about a GraphQL server), which enables pluggable and portable developer tools.

GraphQL vs. SQL

It's also worthwhile to compare GraphQL to SQL (in the abstract, not tied to a specific database or implementation). There's some precedent for using SQL for web apps with the [Facebook Query Language \(FQL\)⁸⁶](#) - what makes GraphQL a better choice?

SQL is very helpful for accessing relational databases and works well with the way such databases are structured internally, but it is not how front-end applications are oriented to consume data. Dealing with concepts like joins and aggregations at the browser level feels like an abstraction leak - instead, we usually do want to think of information as a graph. We often think, "Get me this particular user, and then get me the friends of that user," where "friends" could be any type of connection, like photos or financial data.

There's also a security concern - it's awfully tempting to shove SQL from the web apps straight into the underlying databases, which will inevitably lead to security issues. And as we'll see later on, GraphQL also enables precise access control logic around who can see what kinds of data, and is generally more flexible and less likely to be as insecure as using raw SQL.

Remember that using GraphQL does not mean you have to abandon your backend SQL databases - GraphQL servers can sit on top of any data source, whether it's SQL, MongoDB, Redis, or even a third-party API. In fact, one of the benefits of GraphQL is that it is possible to write a single GraphQL sever that serves as an abstraction over several data stores (or other APIs) simultaneously.

Relay and GraphQL Frameworks

We've talked a lot about *why* you should consider GraphQL, but haven't gone much into *how* you use GraphQL. We'll start to uncover more about that very soon, but we should mention Relay.

⁸⁵<http://swagger.io/>

⁸⁶https://en.wikipedia.org/wiki/Facebook_Query_Language

Relay⁸⁷ is Facebook's framework for connecting React components to a GraphQL server. It allows you to write code like this, which shows how an Item component can automatically retrieve data from a Hacker News GraphQL server:

```
1 class Item extends React.Component {
2   render() {
3     let item = this.props.item;
4
5     return (
6       <div>
7         <h1><a href={item.url}>{item.title}</a></h1>
8         <h2>{item.score} - {item.by.id}</h2>
9         <hr />
10        </div>
11    );
12  }
13};
14
15 Item = Relay.createContainer(Item, {
16   fragments: {
17     item: () => Relay.QL`  

18       fragment on HackerNewsItem {
19         id
20         title,
21         score,
22         url
23         by {
24           id
25         }
26       }
27     `
28   },
29 });
```

Behind the scenes, Relay handles intelligently batching and caching data for improved performance and a consistent user experience. We'll go in-depth on Relay later on, but this should give you an idea of how nicely GraphQL and React can work together.

There are other emerging approaches on how to integrate GraphQL and React, such as Apollo⁸⁸ by the Meteor team.

⁸⁷<https://facebook.github.io/relay/>

⁸⁸<http://www.apollostack.com/>

You can also use GraphQL without using React - it's easy to use GraphQL anywhere you'd traditionally use API calls, including with other technologies like Angular or Backbone.

Chapter Preview

There are two sides to using GraphQL: as an author of a client or front-end web application, and as an author of a GraphQL server. We're going to cover both of these aspects in this chapter and the next.

As a GraphQL client, consuming GraphQL is as easy as an HTTP request. We'll cover the syntax and features of the GraphQL language, as well as design patterns for integrating GraphQL into your JavaScript applications. This is what we'll cover in this chapter.

As a GraphQL server, using GraphQL is a powerful way to provide a query layer over any data source in your infrastructure (or even third-party APIs). GraphQL is just a standard for a query language, which means you can implement a GraphQL server in any language you like (such as Java, Ruby, or C). We're going to use Node for our GraphQL server implementation. We'll cover writing GraphQL servers in the next chapter.

Consuming GraphQL

If you're retrieving data from a server using GraphQL - whether it's with React, another JavaScript library, or a native iOS app - we think of that as a GraphQL "client." This means you'll be writing GraphQL queries and sending them up to the server.

Since GraphQL is its own language, we'll spend this chapter getting you familiar with it and learning to write idiomatic GraphQL. We'll also cover some mechanics of querying GraphQL servers, including various libraries and starting off with an in-browser IDE: GraphiQL.

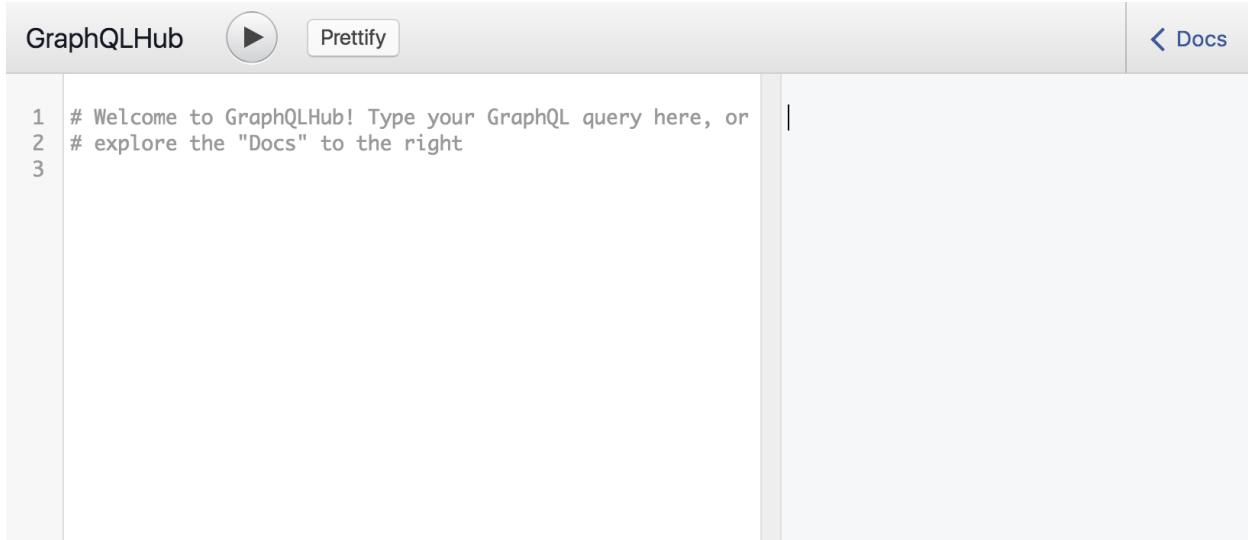
Exploring With GraphiQL

At the start of the chapter we used GraphQLHub with cURL to execute a GraphQL query. This isn't the only way GraphQLHub provides access to its GraphQL endpoint: it also hosts a visual IDE called [GraphiQL](#)⁸⁹. GraphiQL is developed by Facebook and can be used hosted on any GraphQL server with minimal configuration.

You can always issue GraphQL requests using tools like cURL or any language that supports HTTP requests, but GraphiQL is particularly helpful while you become familiar with a particular GraphQL server or GraphQL in general. It provides type-ahead support for errors or suggestions, searchable documentation (generated dynamically using the GraphQL introspection queries), and a JSON viewer that supports code folding and syntax highlighting.

⁸⁹<https://github.com/graphql/graphiql>

Head to [https://graphqlhub.com/playground⁹⁰](https://graphqlhub.com/playground) and get your first look at GraphiQL:



The screenshot shows the GraphiQL interface. At the top, there's a header bar with the text "GraphQLHub" and a play button icon. To the right of the play button are two buttons: "Prettify" and "Docs". The main area is a code editor with three numbered lines of text:

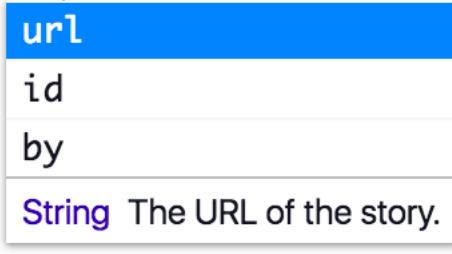
```
1 # Welcome to GraphQLHub! Type your GraphQL query here, or
2 # explore the "Docs" to the right
3
```

Below the code editor, the text "Empty GraphiQL" is centered.

Not much going on yet - go ahead and enter the GraphQL query we cURL'd from earlier:

⁹⁰<https://graphqlhub.com/playground>

```
1 # Welcome to GraphQLHub! Type your GraphQL
2 # explore the "Docs" to the right
3
4 {  
5   hn {  
6     topStories(limit: 1) {  
7       title  
8       url  
9     }  
10    }  
11  }
```



The screenshot shows a code editor with a GraphQL query. At line 9, there is a cursor after 'url'. A dropdown menu is open, listing four options: 'url', 'id', 'by', and 'String'. The 'url' option is highlighted with a blue background. Below the dropdown, a tooltip provides the documentation: 'String The URL of the story.'

GraphQL query

As you type the query, you'll notice the helpful typeahead support. If you make mistakes, such as entering a field that doesn't exist, GraphQL will warn you immediately:

```
{  
  hn {  
    topStories(limit: 2) {  
      title  
      urls  
    }  
  }  
}
```



The screenshot shows a code editor with a GraphQL query. At line 5, there is a cursor after 'urls'. A red error message box appears, stating 'Cannot query field "urls" on type "HackerNewsItem".' with a small error icon.

GraphQL error

This is a great example of GraphQL's type system at work. GraphQL knows what fields and types exist, and in this case the field `urls` does not exist on the `HackerNewsItem` type. We'll explore how types get their fields and names later on.

Hit the “Play” button in the top navigation bar to execute your query. You’ll see the new data appear

in the right pane:

The screenshot shows the GraphQLHub interface. On the left, there is a code editor window containing a GraphQL query. On the right, there is a results pane displaying the JSON response from the query.

GraphQL Query (Left):

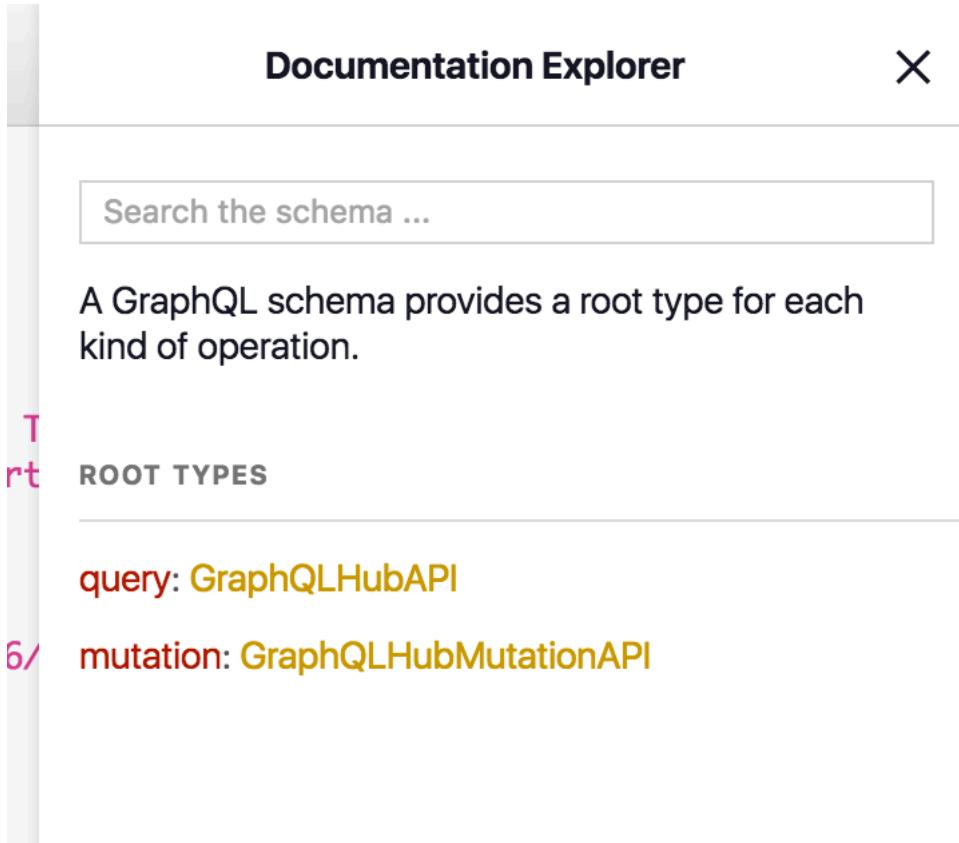
```
1 # Welcome to GraphQLHub! Type your GraphQL q
2 # explore the "Docs" to the right
3
4 { hn {
5   topStories(limit: 2) {
6     title
7     url
8   }
9 }
10 }
11 }
```

GraphQL Data (Right):

```
{ "data": { "hn": { "topStories": [ { "title": "Bank of Japan Is an Estimated Top 10", "url": "http://www.bloomberg.com/news/articles/" }, { "title": "Dropbox as a Git Server", "url": "http://www.anishathalye.com/2016/04/25/" } ] } } }
```

GraphiQL data

See that “Docs” button in the top right corner of GraphiQL? Give that a click to expand the full documentation browser:



[GraphiQL docs](#)

Feel free to click around and choose-your-own-adventure, but eventually come back to the page in the screenshot (the top-level page) and search that `HackerNewsItem` type we ran into trouble with earlier:

The screenshot shows the GraphQL playground interface. At the top, there are three buttons: a left arrow labeled "Schema", a central button labeled "Search Results", and a right arrow labeled "X". Below these buttons is a search bar containing the text "HackerNewsItem". A horizontal line separates the search bar from the results area. The results area has a header "SEARCH RESULTS" and contains a single result entry: "HackerNewsItem". This entry is highlighted with a yellow background, indicating it is the selected item. Another horizontal line separates the results from a "GraphiQL search" section at the bottom. The "GraphiQL search" section contains a single line of text: "GraphiQL search".

Click the matching entry. This takes you to the documentation describing the `HackerNewsItem` type, which includes a description (written by a human, not generated) and a list of all the fields on the type. You can click the fields and their types to find out more information:

The screenshot shows a user interface for a GraphQL schema. At the top, there's a header with a back arrow, the text "Search Results", the title "HackerNewsItem", and a close button (X). Below the header, there's a descriptive text block: "Stories, comments, jobs, Ask HNs and even polls are just items. They're identified by their ids, which are unique integers". Underneath this, there's a section titled "FIELDS" with four listed fields: "id: String!", "deleted: Boolean", "type: ItemType!", and "by: HackerNewsUser!". At the bottom of the visible area, it says "GraphiQL HackerNewsItem".

Stories, comments, jobs, Ask HNs and even polls are just items. They're identified by their ids, which are unique integers

FIELDS

`id: String!`

`deleted: Boolean`

`type: ItemType!`

`by: HackerNewsUser!`

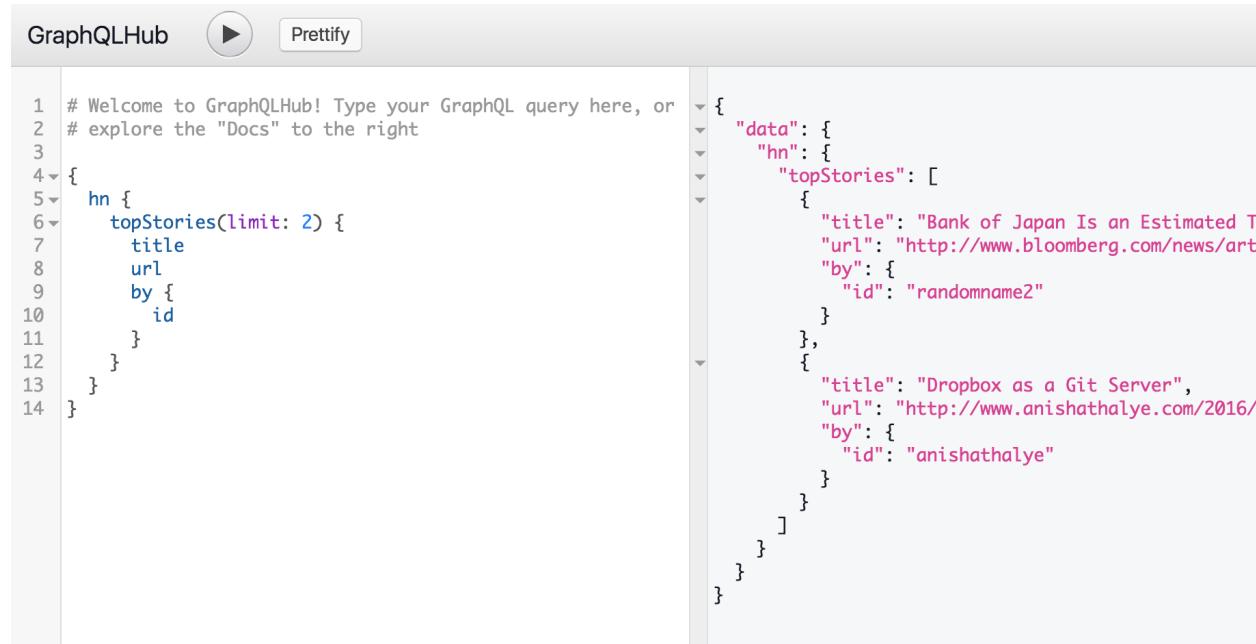
GraphiQL HackerNewsItem

< HackerNewsItem **HackerNewsUser** X

Users are identified by case-sensitive ids. Only users that have public activity (comments or story submissions) on the site are available through the API.

FIELDS**id: String!****delay: Int!****GraphQL HackerNewsUser**

As you can see, the by field of HackerNewsItem has the type HackerNewsUser, which has its own set of fields and links. Let's change our query to grab the information about the author:



The screenshot shows the GraphQLHub interface. At the top, there are buttons for "GraphQLHub", a play button, and "Prettify". Below the header, the code editor contains the following GraphQL query:

```

1 # Welcome to GraphQLHub! Type your GraphQL query here, or
2 # explore the "Docs" to the right
3
4 {
5   hn {
6     topStories(limit: 2) {
7       title
8       url
9       by {
10         id
11       }
12     }
13   }
14 }
```

To the right of the code editor, the results pane displays the JSON response. It shows two stories from the HackerNews API. Each story includes its title, URL, and the user ID of the author (by.id). The author information is expanded, showing their name (by.name) and a link to their profile (by.url). The JSON structure is nested, reflecting the GraphQL schema definition.

GraphQL query with HackerNewsUser

Notice how the by field now shows up both in our query and in the final data - ta da!

Keep GraphQLHub's GraphiQL open in a tab as we start to dig into the mechanics of GraphQL.

GraphQL Syntax 101

Let's dig into the semantics of GraphQL. We've used some terms like "query", "field", and "type", but we haven't properly defined them yet, and there are still a few more to cover before we delve further into our work. For a complete and formal examination of these topics, you can always refer to the [GraphQL specification⁹¹](#).

The entire string you send to a GraphQL server is called a *document*. A document can have one or more *operations* - so far our example documents have just only a *query operation*, but you can also send *mutation operations*.

A *query* operation is read-only - when you send a query, you're asking the server to give you some data. A *mutation* is intended to be a write followed by a fetch; in other words, "Change this data, and then give me some other data." We'll explore mutations more in a bit, but they use the same type system and have the same syntax as queries.

Here's an example of a document with just one query:

```
1 query getTopTwoStories {  
2   hn {  
3     topStories(limit: 2) {  
4       title  
5       url  
6     }  
7   }  
8 }
```

Note that we have prefixed our original query with `query getTopTwoStories`, which is the full and formal way to specify an operation within a document. First we declare the type of operation (query or mutation) and then the name of the operation (`getTopTwoStories`). If your GraphQL document contains just one operation, you can omit the formal declaration and the GraphQL will assume you mean a query:

⁹¹<https://facebook.github.io/graphql/>

```

1  {
2    hn {
3      topStories(limit: 2) {
4        title
5        url
6      }
7    }
8  }

```

In the case that your document has multiple operations, you need to give each of them a unique name. Here's an example of a document with several operations:

```

1 query getTopTwoStories {
2   hn {
3     topStories(limit: 2) {
4       title
5       url
6     }
7   }
8 }
9
10 mutation upvoteStory {
11   hn {
12     upvoteStory(id: "11565911") {
13       id
14       score
15     }
16   }
17 }

```

Generally you won't be sending multiple operations to the server, as the GraphQL specification states a server can only run one operation per document.



Multiple operations are allowed in a document for [advanced performance optimizations⁹²](#) detailed by Facebook.

So that's documents and operations, now let's dig into a typical query. An operation is composed of *selections*, which are generally *fields*. Each field in GraphQL represents a piece of data, which can either be an irreducible *scalar* type (defined below) or a more complex type composed of yet more scalars and complex types.

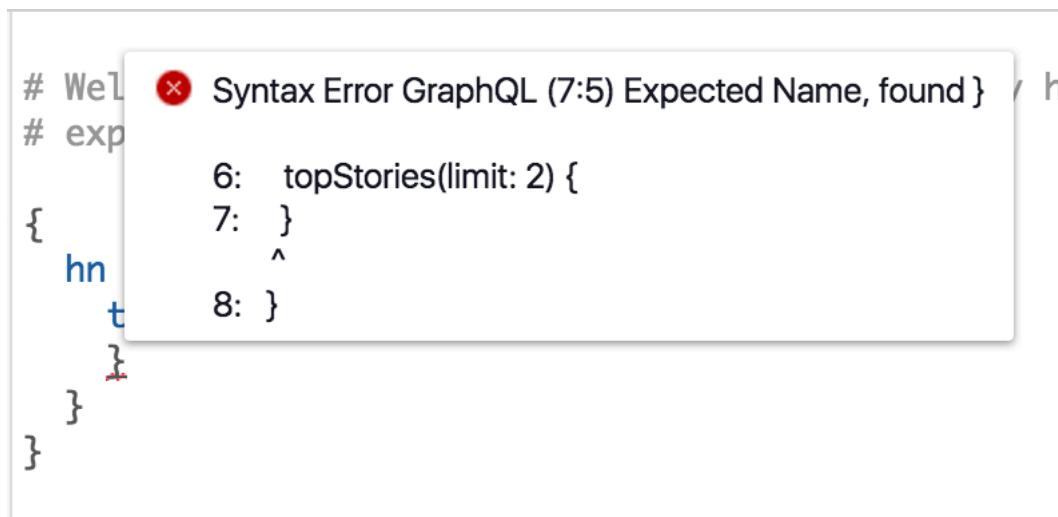
⁹²<https://github.com/facebook/graphql/issues/29#issuecomment-118994619>

In our previous examples, `title` and `url` are scalar fields (as `string` is a scalar type), and `hn` and `topStories` are complex types.

A unique trait of GraphQL is that you **must** specify your selection until it is entirely composed of scalar types. In other words, this query is invalid because `hn` and `topStories` are complex types and the query does not end in any scalar fields:

```
1  {
2    hn {
3      topStories(limit: 2) {
4      }
5    }
6 }
```

If you try this in GraphiQL, it will tell you eagerly that it is invalid:



GraphiQL error without scalar

More philosophically, this means that GraphQL queries must be unambiguous and reinforces the concept that GraphQL is a protocol where you fetch only the data you demand.

Scalar types include `Int`, `Float`, `String`, `Boolean` and `ID` (coerced to a string). GraphQL provides ways of composing these scalars into more complex types using `Object`, `Interface`, `Union`, `Enum`, and `List` types. We'll go into each of those later, but it should be intuitive that they allow you to compose different scalar (and complex) types to create powerful type hierarchies.

Additionally, fields can have *arguments*. It's useful to think of all fields as being functions, and some of them happen to take arguments like functions in other programming languages. Arguments are declared between parenthesis after the name of a field, are unordered, and can even be optional. In our previous example, `limit` is an argument to `topStories`:

```
1 {
2   hn {
3     topStories(limit: 10) {
4       url
5     }
6   }
7 }
```

Arguments are also typed in the same way as fields. If we try to use a string, GraphiQL shows us the error in our ways:

```
# Welcome to GraphQLHub! Type your GraphQL query here, or
# explore the "Docs" to the right
```

```
{  
  hn {  
    topStories(limit: "10") {  
      url  
    }  
  }  
}
```

A screenshot of the GraphiQL interface. A tooltip is displayed over the argument 'limit' in the 'topStories' field of the query. The tooltip contains a red 'X' icon and the text: 'Argument "limit" has invalid value "10". Expected type "Int", found "10"'.

GraphiQL error argument type

It turns out that `limit` is actually an optional argument for this GraphQL server, and omitting it is still a perfectly valid query:

```
1 {
2   hn {
3     topStories {
4       url
5     }
6   }
7 }
```

The arguments to a GraphQL field can also be complex objects, referred to as *input objects*. These are not just the string or numeric scalars we've shown, but are arbitrarily deeply nested maps of keys and values. Here's an example where the argument `storyData` takes an input object which has a `url` property:

```
1  {
2    hn {
3      createStory(storyData: { url: "http://fullstackreact.com" }) {
4        url
5      }
6    }
7 }
```

The collection of fields of a GraphQL server is called its *schema*. Tools like GraphiQL can download the entire schema (we'll show how to do that later) and use that for auto-complete and other functionality.

Complex Types

We've discussed scalars but only alluded to complex types, though we've been using them in our examples. The `hn` and `topStories` fields are examples of `Object` and `List` type fields, respectively. In GraphiQL we can explore their exact types - search HackerNewsAPI to see details about `hn`:

The screenshot shows a GraphQL schema for the Hacker News V0 API. At the top, there are navigation icons: a left arrow, the text "GraphQLHubAPI", the title "HackerNewsAPI" in bold, and a right arrow. Below the title, the text "The Hacker News V0 API" is displayed. A horizontal line separates this from the "FIELDS" section. Under "FIELDS", a list of queries is shown, each starting with a blue "item", "user", or "topStories" keyword followed by parameters in purple and return types in yellow. The list includes:

- item(id: Int!): HackerNewsItem
- user(id: String!): HackerNewsUser
- topStories(limit: Int, offset: Int): [HackerNewsItem]
- newStories(limit: Int, offset: Int): [HackerNewsItem]
- showStories(limit: Int, offset: Int): [HackerNewsItem]
- askStories(limit: Int, offset: Int): [HackerNewsItem]
- jobStories(limit: Int, offset: Int): [HackerNewsItem]
- stories(limit: Int, offset: Int, storyType: String!): [HackerNewsItem]

Below the fields, the text "HackerNewsAPI type" is centered.

Unions

What do you do if your field should actually be more than one type? For example, if your schema has some kind of universal search functionality, it's likely that it will return many different types. So far we've only seen examples where each field is one type, either a scalar or a complex object - how would we handle something like search?

GraphQL provides a few mechanisms for this use-case. First is *unions*, which allow you to define a new type that is one of a list of other types. This example of unions comes straight from the [GraphQL spec⁹³](#):

⁹³<https://facebook.github.io/graphql/#sec-Unions>

```
1 union SearchResult = Photo | Person
2
3 type Person {
4   name: String
5   age: Int
6 }
7
8 type Photo {
9   height: Int
10  width: Int
11 }
12
13 type SearchQuery {
14   firstSearchResult: SearchResult
15 }
```

This syntax look unfamiliar? It's an informal variant of pseudo-code used by the GraphQL spec to describe GraphQL schemas - we won't be writing any code using this format, it's purely to make it easier to describe GraphQL types independent of server code.

In that example, the `SearchQuery` type has a `firstSearchResult` field which may either be a `Photo` or `Person` type. The types in a union don't have to be objects - they can be scalars, other unions, or even a mix.

Fragments

If you look closer, you'll notice there are no fields in common between `Person` and `Photo`. How do we write a GraphQL query which handles both cases? In other words, if our search returns a `Photo`, how do we know to return the `height` field?

This is where *fragments* come into play. Fragments allow you to group sets of fields, independent of type, and re-use them throughout your query. Here's what a query with fragments could look like for the above schema:

```
1 {
2   firstSearchResult {
3     ... on Person {
4       name
5     }
6     ... on Photo {
7       height
8     }
9   }
10 }
```

The `... on Person` bit is referred to as an *inline fragment*. In plain-English we could read this as, “If the `firstSearchResult` is a `Person`, then return the `name`; if it’s a `Photo`, then return the `height`.”

Fragments don’t have to be inline; they can be named and re-used throughout the document. We could rewrite the above example using *named fragments*:

```
1  {
2    firstSearchResult {
3      ... searchPerson
4      ... searchPhoto
5    }
6  }
7
8 fragment searchPerson on Person {
9   name
10 }
11
12 fragment searchPhoto on Photo {
13   height
14 }
```

This would allow us to use `searchPerson` in other parts of the query without duplicating the same inline fragment everywhere.

Interfaces

In addition to unions, GraphQL also supports *interfaces*, which you might be familiar with from programming languages like Java. In GraphQL, if an object type implements an interface, then the GraphQL server enforces that the type will have all the fields the interface requires. A GraphQL type that implements an interface may also have its own fields that are not specified by the interface, and a single GraphQL type can implement multiple interfaces.

To continue the search example, you can imagine our search engine having this type of schema:

```
1 interface Searchable {  
2   searchResultPreview: String  
3 }  
4  
5 type Person implements Searchable {  
6   searchResultPreview: String  
7   name: String  
8   age: Int  
9 }  
10  
11 type Photo implements Searchable {  
12   searchResultPreview: String  
13   height: Int  
14   width: Int  
15 }  
16  
17 type SearchQuery {  
18   firstSearchResult: Searchable  
19 }
```

Let's break this down. Our `firstSearchResult` is now guaranteed to return a type that implements `Searchable`. Because this otherwise unknown type implements `Searchable`,

These are the primitives of GraphQL - operations, types (scalar and complex), and fields - and we can use them to compose higher-order patterns.

Exploring a Graph

We've explored the "QL" of GraphQL, but haven't touched too much on the "Graph" part.

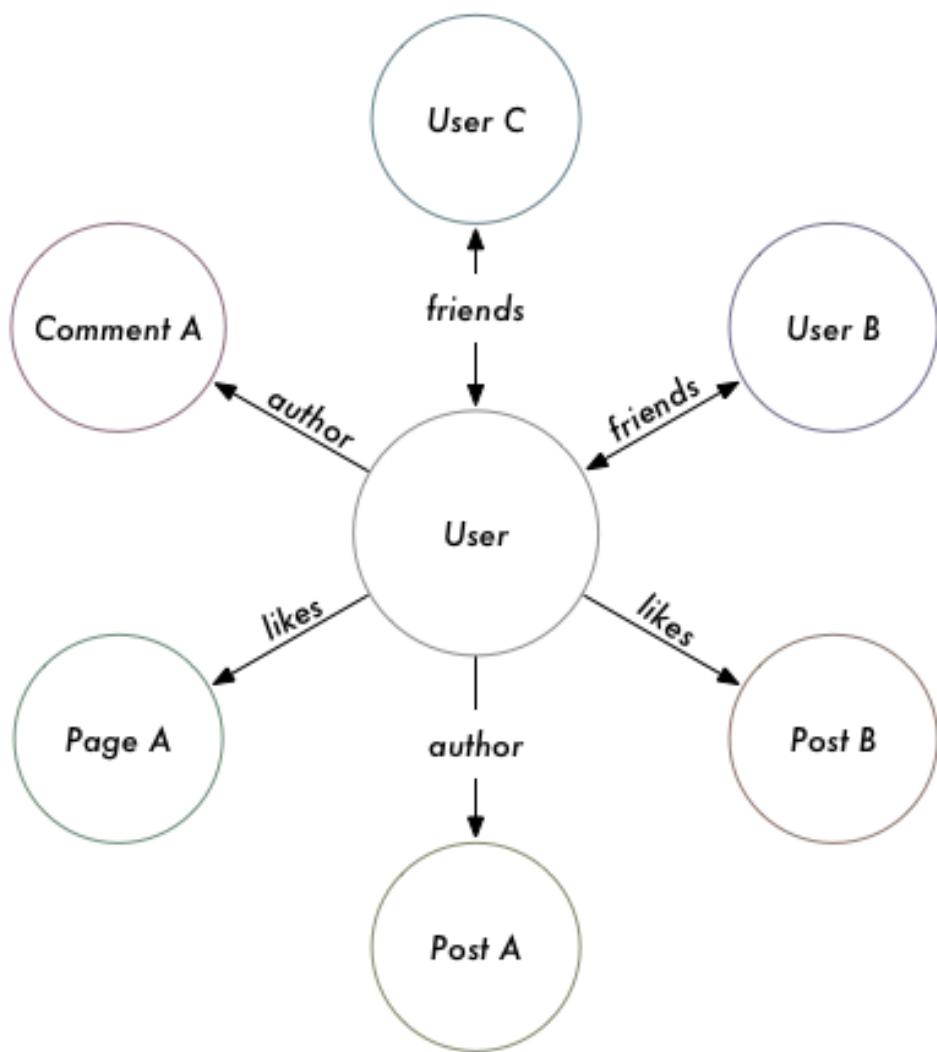


When we say "Graph", we don't mean in the sense of a bar chart or other data visualizations - we mean a graph in the more [mathematical sense](#)⁹⁴.

A graph is composed of a set of objects that are linked together. Each object is called a *node*, and the link between a pair of objects is called an *edge*. You might not be used to thinking about your product's data with this vocabulary, but it is surprisingly representative of most applications.

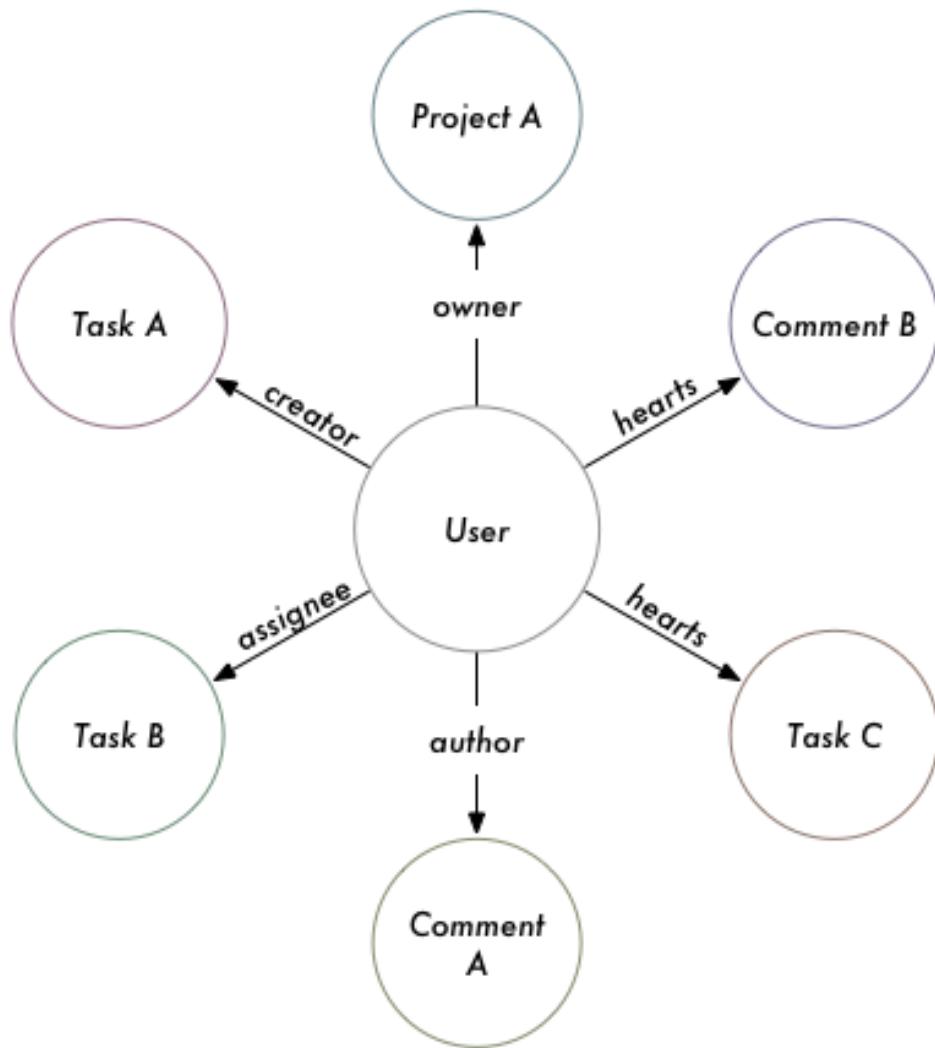
Let's consider the graph of a Facebook user:

⁹⁴[https://en.wikipedia.org/wiki/Graph_\(discrete_mathematics\)](https://en.wikipedia.org/wiki/Graph_(discrete_mathematics))



Facebook Graph

But also consider the graph of a productivity application like Asana:



Asana Graph

Even though Asana is not really a social network, the product's data still forms a graph.

Note that just because your product's data resembles a graph structure doesn't mean you need to use a graph database: Facebook and Asana's underlying datastore is often MySQL, which doesn't natively support graph operations. Any data store, whether a relational database, key-value store, or document store, can be exposed as a graph in GraphQL.

The ideal GraphQL schema closely models the shape and terms of a mathematical graph. Here's a preview of the type of graph-like schema we'll be exploring:

```
1  {
2    viewer {
3      id
4      name
5      likes {
6        edges {
7          cursor
8          node {
9            id
10           name
11         }
12       }
13     }
14   }
15   node(id: "123123") {
16     id
17     name
18   }
19 }
```

Adding these extra layers of fields (edges, node, etc) may seem like over-engineering when coming from something like REST. However, these patterns emerged from building Facebook and have provides a foundation as your product grows and adds new features.

The patterns we're going to describe in the next few sections are derived from the [GraphQL patterns required by Relay⁹⁵](#). Even if you don't use Relay, this way of thinking of your GraphQL schema will prevent you from "fighting the framework" and it wouldn't be surprising if future GraphQL front-end libraries beyond Relay continue to embrace them.

Graph Nodes

When querying a graph, you generally need to start your query with a node. Your app is either fetching fields directly on a node or finding out about what connections it has.

For example, the querying "the current user's feed" in Facebook would start with the current user's node, and explore all "feed item" nodes connected to it.

In idiomatic GraphQL, you generally define a simple Node interface like this:

⁹⁵<https://facebook.github.io/relay/docs/graphql-relay-specification.html>

```
1 interface Node {  
2   id: ID  
3 }
```

So anything that implements this interface is expected to have an `id` field - that's it. Remember from earlier that the `ID` scalar type is casted to a `string`. If you're coming from the REST world, this is sort of like when you can query a resource using a URL like `/$nouns/:id`.

One key difference that idiomatic GraphQL and other protocols is all of your node IDs should be globally unique. In other words, it would be invalid to have a User with an ID of "1" and a Photo with an ID of "1", as the IDs would collide.

The reason behind this is that you should also expose a top-level field that lets you query arbitrary nodes by ID, something like this:

```
1 {  
2   node(id: "the_id") {  
3     id  
4   }  
5 }
```

If you have IDs that collide across types, it's impossible to determine what to return from that field. If you're using a relational database, this might be puzzling because by default primary keys are only guaranteed to be unique per-table. A common technique is to prefix your database IDs with a string that corresponds to the type and makes the IDs unique again. In pseudo-code:

```
1 const findNode = (id) => {  
2   const [ prefix, dbId ] = id.split(":");  
3   if (prefix === 'users') {  
4     return database.usersTable.find(dbId);  
5   }  
6   else if ( ... ) {  
7   }  
8 };  
9  
10 const getUser = (id) => {  
11   let user = database.usersTable.find(id);  
12   user.id = `users:${user.id}`;  
13   return user;  
14 };
```

Whenever our GraphQL server returns a user (via `getUser`), it changes the database ID to make it unique. Then when we lookup a node (via `findNode`), we read the "scope" of the ID and act

appropriately. This also highlights a benefit of why IDs are strings in GraphQL, because they allow for readable changes like these.

Why do we want the ability to query any node using the `node(id:)` field? It makes it easier for our web applications to “refresh” stale data without having to keep track of where a piece of data comes from in the schema. If we know a todo-list item changed its state, our app should be able to look-up the latest state from the server without knowing what project or group it belongs to.

Generally our apps don’t query global node IDs from the start - how do we describe the “current user’s node” in GraphQL? It turns out the *viewer pattern* comes in handy.

Viewer

In addition to the `node(id:)` field, it’s very helpful if your GraphQL schema has a top-level `viewer` field. The “viewer” represents the current user and the connections to that user. At the schema-level, the `viewer`’s type should implement the `Node` interface like any other node type.

If you’re coming from REST, this either happens implicitly or all of your endpoints are prefixed being with a path like `/users/me/$resources`. GraphQL tends to prefer explicit behaviors, which is why the `viewer` field tends to exist.

Imagine we’re writing a Slack app with GraphQL. Everything is viewer-centric in a messaging app (“what messages do *I* have”, “what channels am *I* subscribed to”, etc), so our queries would look something like this:

```
1 {
2   viewer {
3     id
4     messages {
5       participants {
6         id
7         name
8       }
9       unreadCount
10    }
11    channels {
12      name
13      unreadCount
14    }
15  }
16}
```

If we didn’t have the top-level `viewer` field on its own, we’d either have to make two server calls (“get me the ID of the current user, and then get me the messages for that user”) or make top-level `messages` and `channels` fields that implicitly return the data for the current user.

When we implement our own GraphQL server later on, we'll see that using a `viewer` field also makes it easier to implement authorization logic (i.e. prevent one user from seeing another user's messages).

Graph Connections and Edges

In that last example of a `viewer`-centric query, we had two fields that you can think of as *connections* to sets of other nodes (`messages` and `channels`). For simple and small sets of data, we could just return an array of all these nodes - but what happens if the data set is very large? If we were loading something like posts of Reddit, we definitely couldn't fit them into one array.

The usual way to load large sets of data is *pagination*. Many APIs will let you pass some kind of query parameters like `?page=3` or `?limit=25&offset=50` to iterate over a bigger list. This works well for apps where the data is relatively stable, but can accidentally lead to a poor experience in apps where data updates in near-real-time: something like Twitter's feed is may add new tweets while the user is browsing, which will throw off the offset calculations and lead to duplicated data when loading new pages.

Idiomatic GraphQL takes a strong opinion on how to solve that problem using *cursor-based pagination*. Instead of using pages or limits and offsets, GraphQL requests pass *cursors* (usually strings) to specify the location in the list to load next.

In plain-English, a GraphQL request retrieves an initial set of nodes and each node is returned with a unique cursor; when the app needs to load more data, it sends a new request equivalent to, "Give me the 10 nodes after cursor `XYZ`".

Let's try using a GraphQL endpoint that supports cursors. Take a look at this query that you can send to GraphQLHub:

```
1  {
2    hn2 {
3      nodeFromHnId(id: "dhouston", isUserId:true) {
4        ... on HackerNewsV2User {
5          submitted(first: 2) {
6            pageInfo {
7              hasNextPage
8              hasPreviousPage
9              startCursor
10             endCursor
11           }
12           edges {
13             cursor
14             node {
15               id
```

```

16          ... on HackerNewsV2Comment {
17              text
18          }
19      }
20  }
21 }
22 }
23 }
24 }
25 }
```

Pretty big query there, let's break it down - first we get our initial node using `nodeFromHnId` (which retrieves the node for [Drew Houston⁹⁶](#)'s Hacker News account). We then grab the first two nodes in the submitted connection.

A GraphQL connection has two fields, `pageInfo` and `edges`. `pageInfo` is metadata about that particular “page” (remember that this is more of a moving “window” than a page). Your front-end code can use this metadata to determine when and how to load more information - for example, if `hasNextPage` is true then can show the appropriate button to load more items. The `edges` field is a list of the actual nodes. Each entry in `edges` contains the cursor for that node, as well as the node itself.

Notice that the node `id` and `cursor` are separate fields - in some systems it may be appropriate to use `id` as part of the cursor (such as if your identifiers are atomically incrementing integers), but others may prefer to make `cursor` a function of timestamp, offset, or both. In general, cursors are intended to be opaque strings, and may become invalid after a certain period of time (in the case that the backend is caching search results temporarily).

Let's say this is the data returned by that query:

```

1  {
2      "nodeFromHnId": {
3          "submitted": {
4              "pageInfo": {
5                  "hasNextPage": true,
6                  "hasPreviousPage": false,
7                  "startCursor": "YXJyYX1jb25uZWN0aW9u0jE=",
8                  "endCursor": "YXJyYX1jb25uZWN0aW9u0jI="
9              },
10             "edges": [
11                 {
12                     "cursor": "YXJyYX1jb25uZWN0aW9u0jE=",
```

⁹⁶https://en.wikipedia.org/wiki/Drew_Houston

```

13     "node": {
14       "id": "aXRlbTo1MzgxNjk0",
15       "text": "it's not going anywhere :)<p>(actually, come work on it\
16 : <a href=\"https://www.dropbox.com/jobs\" rel=\"nofollow\">https://www.dropbox.\\
17 com/jobs</a> :)"
18     }
19   },
20   {
21     "cursor": "YXJyYX1jb25uZWN0aW9uOjI=",
22     "node": {
23       "id": "aXRlbTo0NjgxMzY2",
24       "text": "yes we are :)"
25     }
26   }
27 ]
28 }
29 }
30 }
```

Take a look at how some of the fields match up - the first cursor in edges matches the startCursor, as well as the endCursor matching the last node's cursor. Our front-end code could use endCursor to construct the query to fetch the next set of data using the after argument:

```

1 {
2   hn2 {
3     nodeFromHnId(id:"dhouston", isUserId:true) {
4       ... on HackerNewsV2User {
5         submitted(first: 2, after: "YXJyYX1jb25uZWN0aW9uOjI=") {
6           pageInfo {
7             hasNextPage
8             hasPreviousPage
9             startCursor
10            endCursor
11          }
12          edges {
13            cursor
14            node {
15              id
16              ... on HackerNewsV2Comment {
17                text
18              }
19            }
20          }
21        }
22      }
23    }
24  }
25 }
```

```
20      }
21    }
22  }
23}
24}
25}
```

The other arguments that exist on connections are `before` and `after` (which accept cursors), and `first` and `last` (which accepts integers).

The cursor pattern may come off as verbose if you're used to pagination in REST, but give it five minutes and explore the Hacker News pagination API we demonstrated. Cursors are robust to real-time updates and allow for more reusable app-level code when loading nodes. When we implement our own GraphQL server in a bit, we'll show how to implement this kind of schema and you'll see it isn't as daunting as it may initially appear.

Mutations

When we first introduced operations, we mentioned that `mutation` exists alongside the read-only `query` operation. Most apps will need a way to write data to the server, which is the intended use for mutations.

With REST-like protocols, mutations are generally occur with POST, PUT, and DELETE HTTP requests. In that sense, both GraphQL and REST try to separate read-only requests from writes. So what do we gain with GraphQL? Because GraphQL mutations leverage GraphQL's type system, you can declare the data you want returned following your mutation.

For example, you can try this mutation on GraphQLHub to edit an in-memory key value store:

```
1 mutation {
2   keyValue_setValue(input: {
3     clientMutationId: "browser", id: "this-is-a-key", value: "this is a value"
4   }) {
5     item {
6       value
7       id
8     }
9   }
10 }
```

The mutation field here is `keyValue_setValue`, and it takes an `input` argument that gives information what key and value to set. But we also get to pick and choose the fields returned by the mutation, namely the `item` and whatever set of fields we want from that. If you run that request, you'll get back this sort of payload:

```

1  {
2    "data": {
3      "keyValue_setValue": {
4        "item": {
5          "value": "some value",
6          "id": "someKey"
7        }
8      }
9    }
10 }
```

In a REST world, we would be stuck with whatever data the request returns, and as our product evolves we may have to make many changes to that payload on the client and server. Using GraphQL means that our server and client will be more resilient and flexible in the future.

Other than requiring specifying the `mutation` operation type, everything about mutations is normal GraphQL: you have types, fields, and arguments. As we'll see later on when we implement our own GraphQL schema, implementing them on the server is similar as well.

Subscriptions

We've discussed the two main types of GraphQL operations, `query` and `mutation`, but there is a third type of operation currently in development: `subscription`. The use-case of subscriptions is to handle the kinds of real-time updates seen in apps like Twitter and Facebook, where the number of likes or comments on an item will update without manual refreshing by the user.

This provides a great user experience, but is often complicated to implement on a technical level. GraphQL takes the opinion that the server should publish the set of events that it's possible to subscribe to (such as new likes to a post) and clients can opt-in to subscribing to them. Check out the example subscription that Facebook gives in [their documentation⁹⁷](#):

```

1 input StoryLikeSubscribeInput {
2   storyId: string
3   clientSubscriptionId: string
4 }
5
6 subscription StoryLikeSubscription($input: StoryLikeSubscribeInput) {
7   storyLikeSubscribe(input: $input) {
8     story {
9       likers { count }
10      likeSentence { text }
```

⁹⁷<http://graphql.org/blog/subscriptions-in-graphql-and-relay/>

```
11      }
12  }
13 }
```

Issuing a GraphQL request with this subscription essentially tells the server, “Hey, here’s the data I want whenever a `StoryLikeSubscription` occurs, and here’s my `clientSubscriptionId` so you know where to find me.” Note that the use of `clientSubscriptionId` or any details about what subscription operations should look like is not specific by GraphQL; it merely reserves the subscription operation type as an acceptable and defers to each application on how to handle real-time updates.

The mechanics of how the clients subscribe to updates are outside of the scope of GraphQL - Facebook mentions using [MQTT⁹⁸](#) with the `clientSubscriptionId`, but other possibilities include [WebSockets⁹⁹](#), [Server-Sent Events¹⁰⁰](#), or any of a number of other mechanisms. In pseudo-code, the process looks like:

```
1 var clientSubscriptionId = generateSubscriptionId();
2 // this "channel" could be WebSockets, MQTT, etc
3 connectToRealtimeChannel(clientSubscriptionId, (newData) => {});
4 // send the GraphQL request to tell the server to start sending updates
5 sendGraphQLSubscription(clientSubscriptionId);
```

The way GraphQL has decided to implement subscriptions, where a server allows a finite list of possible events, is different than the way other frameworks like Meteor handle updates, where all data is subscribable by default. As Facebook details in [their writing¹⁰¹](#), that type of system is generally very difficult to engineer, especially at scale.

GraphQL With JavaScript

So far we’ve been using cURL and GraphiQL for all of our GraphQL queries, but at the end of the day we’re going to be writing JavaScript web apps. How do we send GraphQL requests in the browser?

Well, you can use any HTTP library you like - jQuery’s AJAX methods will work, or even raw XMLHttpRequests, which enables you to use GraphQL in older non-ES2015 apps. But because we’re examining how modern JavaScript apps work in the React ecosystem, we’re going to examine ES2015 `fetch`.

Fire up Chrome, open up [the GraphQLHub website¹⁰²](#) and open a JavaScript debugger.

⁹⁸<https://en.wikipedia.org/wiki/MQTT>

⁹⁹<https://en.wikipedia.org/wiki/WebSocket>

¹⁰⁰https://en.wikipedia.org/wiki/Server-sent_events

¹⁰¹<http://graphql.org/blog/subscriptions-in-graphql-and-relay/#why-not-live-queries>

¹⁰²<https://www.graphqlhub.com/>



You can open the Chrome DevTools JavaScript Debugger by clicking the Chrome “hamburger” icon and picking More Tools > Developer Tools or by right-clicking on the page, pick Inspect, and then clicking on the Console tab.

Modern versions of Chrome support `fetch` out of the box, which makes it handy for prototyping, but you can also use any other tooling that supports poly-filling `fetch`. Give this code a shot:

```
1 var query = ' { graphQLHub } ';
2 var options = {
3   method: 'POST',
4   body: query,
5   headers: {
6     'content-type': 'application/graphql'
7   }
8 };
9
10 fetch('https://graphqlhub.com/graphql', options).then((res) => {
11   return res.json();
12 }).then((data) => {
13   console.log(JSON.stringify(data, null, 2));
14 });
```

The configuration here should look similar to our settings for cURL. We use a POST method, set the appropriate content-type header, and then use our GraphQL query string as the request body. Give your code a moment and you should see this output:

```
1 {
2   "data": {
3     "graphQLHub": "Use GraphQLHub to explore popular APIs with GraphQL! Created \
4 by Clay Allsopp @clayallsopp"
5   }
6 }
```

Congratulations, you just ran a GraphQL query with JavaScript! Because GraphQL requests are Just HTTP at the end of the day, you can incrementally move your API calls over to GraphQL, it doesn’t have to be done in one big-bang.

Making API calls is usually a fairly low-level operation, though - how can it integrate into a larger app?

GraphQL With React

This book is all about React, so it's about time we integrate GraphQL with React, right? Almost!

The most promising way of using GraphQL and React is Relay, to which we're dedicating an entire chapter. Relay automates many of the best practices for React/GraphQL applications, such as caching, cache-busting, and batching. It would be a tall-order to cover the mechanics of how Relay does these things, and ultimately using Relay is the better solution than writing your own.

But adopting Relay may have more friction in an existing app, so it's worthwhile to discuss a few techniques for adding GraphQL to an existing React app.

If you're using Redux, you can probably swap out your REST or other API calls with GraphQL calls using the `fetch` technique we showed earlier. You won't get the colocated queries API that Relay or other GraphQL-specific libraries provide, but GraphQL's benefits (such as development experience and testability) will still shine.

There are burgeoning alternatives to Relay as well. [Apollo¹⁰³](#) is a collection of projects including [react-apollo¹⁰⁴](#). `react-apollo` allows you to colocate views and their GraphQL queries in a manner similar to Relay, but uses Redux under-the-hood to store your GraphQL cache and data. Here's an example of a simple Apollo component:

```
1 class AboutGraphQLHub extends React.Component {
2   render() {
3     return <div>{ this.props.about.graphQLHub }</div>;
4   }
5 }
6
7 const mapQueriesToProps = () => {
8   return {
9     about : {
10       query: '{ graphQLHub }'
11     }
12   };
13 };
14
15 const ConnectedAboutGraphQLHub = connect({
16   mapQueriesToProps
17 })(AboutGraphQLHub);
```

Building upon Redux means you can more easily integrate it into an existing Redux store like any other middleware or reducer:

¹⁰³<http://apollostack.com>

¹⁰⁴<http://docs.apollostack.com/apollo-client/react.html>

```
1 import ApolloClient from 'apollo-client';
2 import { createStore, combineReducers, applyMiddleware } from 'redux';
3
4
5 const client = new ApolloClient();
6
7 const store = createStore(
8   combineReducers({
9     apollo: client.reducer(),
10    // other reducers here
11  }),
12  applyMiddleware(client.middleware())
13 );
```

Check out the [Apollo docs¹⁰⁵](#) if this sounds like it could be a good fit for your project.

Wrapping Up

Now you've written a few GraphQL queries, learned about different its features, and even written a bit of code to get GraphQL in your browser. If you have an existing GraphQL server for your product, it might be okay to move on and jump to the chapter on Relay - but in most cases you'll also need to draft your own GraphQL server. Excelsior!

¹⁰⁵<http://docs.apollostack.com/index.html>

GraphQL Server

Writing a GraphQL Server

In order to use Relay or any other GraphQL library, you need a server that speaks GraphQL. In this chapter, we’re going to write a backend GraphQL server with NodeJS and other technologies we’ve used in earlier chapters.

We’re using Node because we can leverage the tooling used elsewhere in the React ecosystem, and Facebook targets Node for many of their backend libraries. However, there are GraphQL server libraries in every popular language, such as [Ruby¹⁰⁶](#), [Python¹⁰⁷](#), and [Scala¹⁰⁸](#). If your existing backend uses a framework in a language other than JavaScript, such as Rails, it might make sense to look into a GraphQL implementation in your current language.

The lessons we’ll go over in this section, such as how to design a schema and work with an existing SQL database, are applicable to all GraphQL libraries and languages. We encourage you to follow along with the section and apply what you learn to your own projects, regardless of language.

Let’s get to it!

Game Plan

At a high-level, here’s what we’re going to do:

- Create an [Express¹⁰⁹](#) HTTP server
- Add an endpoint which accepts GraphQL requests
- Construct our GraphQL schema
- Write the glue-code that resolves data for each GraphQL field in our schema
- Support GraphiQL so we can debug and iterate quickly

The schema we’re going to draft is going to be for a social network, a sort of “Facebook-lite,” backed by a SQLite database. This will show common GraphQL patterns and techniques to efficiently engineer GraphQL servers talking to existing data stores.

Express HTTP Server

Let’s start setting up our web server. Create a new directory called `graphql-server` and run some initial `npm` commands:

¹⁰⁶<https://github.com/rmosolgo/graphql-ruby>

¹⁰⁷<https://github.com/graphql-python/graphene>

¹⁰⁸<https://github.com/sangria-graphql/sangria>

¹⁰⁹<http://expressjs.com>

```
$ mkdir graphql-server
$ cd ./graphql-server
$ npm init
$ touch index.js server.js
# hit enter a bunch, accept the defaults
$ npm install babel-register@6.3.13 babel-preset-es2015@6.3.13 express@4.13.3 --\n
save --save-exact
$ echo '{ "presets": ["es2015"] }' > .babelrc
```

Let's run through what happened: we created a new folder called `graphql-server` and then jumped inside of it. We ran `npm init`, which creates a `package.json` for us, and then we manually created two files called `server.js` and `index.js`. Then we installed some dependencies, Babel and Express. The name Babel should be familiar from earlier chapters - in this case, we installed `babel-register` to transpile NodeJS files and `babel-preset-es2015` to instruct Babel on how to transpile said files. The final command created a file called `.babelrc`, which configured Babel to use the `babel-preset-es2015` package.

Open up `index.js` and add these lines:

```
1 require('babel-register');
2
3 require('./server');
```

Not a lot going on here, but it's important. By requiring `babel-register`, every subsequent call to `require` (or `import` when using ES2015) will go through Babel's transpiler. Babel will transpile the files according to the settings in `.babelrc`, which we configured to use the `es2015` settings.

For our next trick, open `server.js` and add a quick line to debug that our code is working:

```
1 console.log({ starting: true });
```

If you run `node index.js`, you should see this happen:

```
$ node index.js
{ starting: true }
```

Wonderful start! Now let's add some HTTP.

Express is a very powerful and extensible HTTP framework, so we're not going to go too in-depth; if you're ever curious to learn more about it, check out their [documentation](#)¹¹⁰.

Open up `server.js` again and add code to configure Express:

¹¹⁰<http://expressjs.com>

```
1 console.log({ starting: true });
2
3 import express from 'express';
4
5 const app = express();
6
7 app.use('/graphql', (req, res) => {
8   res.send({ data: true });
9 });
10
11 app.listen(3000, () => {
12   console.log({ running: true });
13 });
```

The first few lines are straight-forward - we import the `express` package and create a new instance (you can think of this as creating a new server). At the end of the file, we tell that server to start listening for traffic on port 3000 and show some output after that's happening.

But before we start the server, we need to tell it how to handle different kinds of requests. `app.use` is how we're going to do that today. It's first argument is the path to handle, and the second argument is a handler function. `req` and `res` are shorthand for "request" and "response", respectively. By default, paths registered with `app.use` will respond on all HTTP methods, so as of now `GET /graphql` and `POST /graphql` do the same thing.

Let's give a short and test it out. Run your server again with `node index.js`, and in a separate terminal fire off a cURL:

```
$ node index.js
{ starting: true }
{ running: true }

$ curl -XPOST http://localhost:3000/graphql
{"data":true}
$ curl -XGET http://localhost:3000/graphql
{"data":true}
```

We have a working HTTP server! Now time to "do some GraphQL," so to speak.



Tired of restarting your server after every change? You can setup a tool like `Nodemon`¹¹¹ to automatically restart your server when you make edits. `npm install -g nodemon && nodemon index.js` should do the trick.

¹¹¹<https://github.com/remy/nodemon#nodemon>

Adding First GraphQL Types

We need to install some GraphQL libraries; stop your server if it's running, and run these commands:

```
$ npm install graphql@0.6.0 express-graphql@0.5.3 --save --save-exact
```

Both have "GraphQL" in their name, so that should sound promising. These are two libraries maintained by Facebook and also serve as reference implementations for GraphQL libraries in other languages.

The [graphql library¹¹²](#) exposes APIs that let us construct our schema, and then exposes an API for resolving raw GraphQL document strings against that schema. It can be used in any JavaScript application, whether an Express web server like in this example, or another servers like Koa, or even in the browser itself.

In contrast, the [express-graphql package¹¹³](#) is meant to be used only with Express. It handles ensuring that HTTP requests and responses are correctly formatted for GraphQL (such dealing with the content-type header), and will eventually allow us to support GraphiQL with very little extra work.

Let's get to it - open up `server.js` and add these lines after you create the `app` instance:

```
5 const app = express();
6
7 import graphqlHTTP from 'express-graphql';
8 import { GraphQLSchema, GraphQLObjectType, GraphQLString } from 'graphql';
9
10 const RootQuery = new GraphQLObjectType({
11   name: 'RootQuery',
12   description: 'The root query',
13   fields: {
14     viewer: {
15       type: GraphQLString,
16       resolve() {
17         return 'viewer!';
18       }
19     }
20   }
21 });
22
23 const Schema = new GraphQLSchema({
24   query: RootQuery
```

¹¹²<https://github.com/graphql/graphql-js>

¹¹³<https://github.com/graphql/express-graphql>

```

25 });
26
27 app.use('/graphql', graphqlHTTP({ schema: Schema }));

```

Note that we've changed our previous arguments to `app.use` (this replaces the `app.use` from before). There's a bunch of interesting things going on here, but let's skip to the good part first. Start up your server (`node index.js`) and run this cURL command:

```
$ curl -XPOST -H 'content-type:application/graphql' http://localhost:3000/graphql\
1 -d '{ viewer }'
{"data": {"viewer": "viewer!"}}
```

If you see the above output then your server is configured correctly and resolving GraphQL requests accordingly. Now let's walk through how it actually works.

First we import some dependencies from the GraphQL libraries:

```

7 import graphqlHTTP from 'express-graphql';
8 import { GraphQLSchema, GraphQLObjectType, GraphQLString } from 'graphql';

```

The `graphql` library exports many objects and you'll become familiar with them as we write more code. The first two we use are `GraphQLObjectType` and `GraphQLString`:

```

10 const RootQuery = new GraphQLObjectType({
11   name: 'RootQuery',
12   description: 'The root query',
13   fields: {
14     viewer: {
15       type: GraphQLString,
16       resolve() {
17         return 'viewer!';
18       }
19     }
20   }
21 });

```

When you create a new instance of `GraphQLObjectType`, it's analogous to defining a new class. It's required that we give it a `name` and optional (but very helpful for documentation) that we set a `description`.

The `name` field sets the type name in the GraphQL schema. For instance, if want to define a fragment on this type, we would write `... on RootQuery` in our query. If we changed `name` to something

like `AwesomeRootQuery`, we would need to change our fragment to `... on AwesomeRootQuery`, even though the JavaScript variable is still `RootQuery`.

That defines the type, now we need to give it some fields. Each key in the `fields` object defines a new corresponding field, and each field object has some required properties.

We need to give it:

- a type - the GraphQL library exports the basic scalar types, such as `GraphQLString`.
- a `resolve` function to return the `value` of the field - for now, we have the hard-coded value `'viewer!'`.

Next we create an instance of `GraphQLSchema`:

```
23 const Schema = new GraphQLSchema({  
24   query: RootQuery  
25 });
```

Hopefully the naming makes it clear that this is the top-level GraphQL object.

You can only resolve queries once you have an instance of a schema - you can't resolve query strings against object types by themselves.

Schemas have two properties: `query` and `mutation`, which corresponds to the two types of operations we discussed earlier. Both of these take an instance of a GraphQL type, and for now we just set the `query` to `RootQuery`.

One quick note on naming things (one of the Hard Problems of computer science): we generally refer to the top-level query of a schema as the *root*. You'll see many projects that have similar `RootQuery`-named types.

Finally we hook it all up to Express:

```
27 app.use('/graphql', graphqlHTTP({ schema: Schema }));
```

Instead of manually writing a handler function, the `graphqlHTTP` function will generate one using our `Schema`. Internally, this will grab our GraphQL query from the request and hand it off to the main GraphQL library's resolving function.

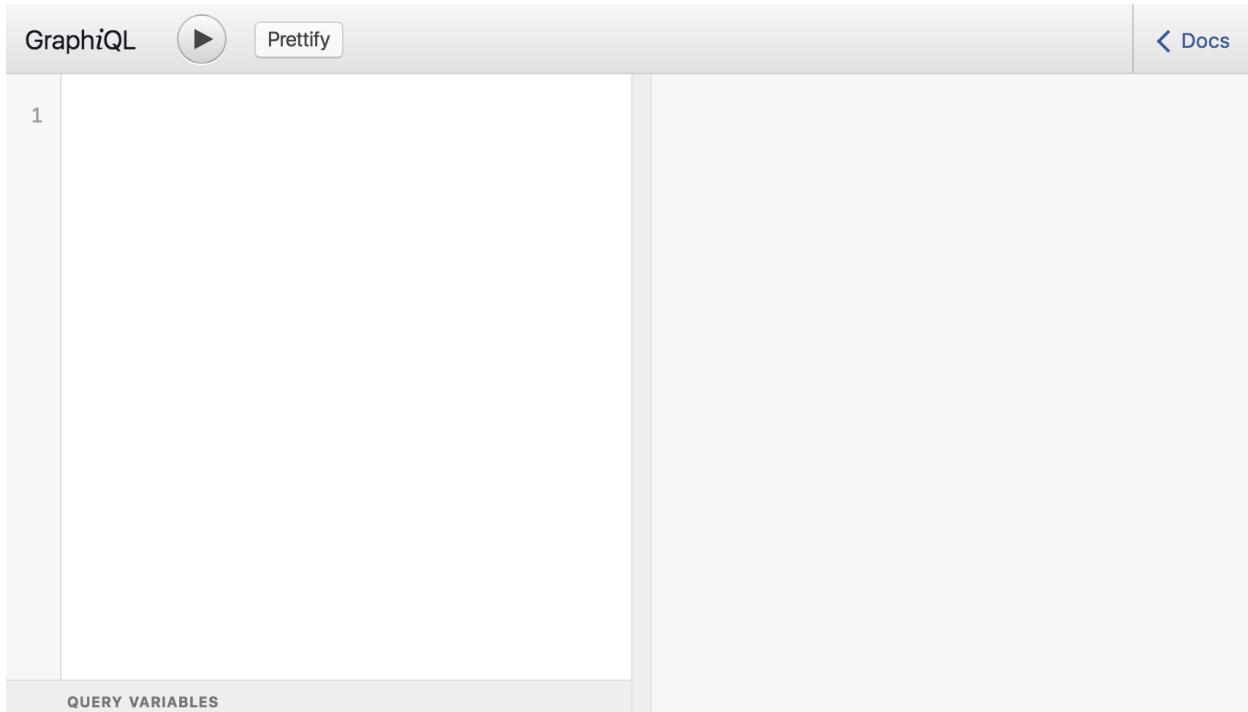
Adding GraphiQL

Earlier we used GraphQLHub's hosted instance of GraphiQL, the GraphQL IDE. What if I told you that you could add GraphiQL to our little GraphQL server with just one change?

Try adding the `graphiql: true` option to `graphqlHTTP`:

```
27 app.use('/graphql', graphqlHTTP({ schema: Schema, graphiql: true }));
```

Restart your server, head to Chrome, and open <http://localhost:3000/graphql>¹¹⁴. You should see something like this:



Empty GraphiQL

If you open the “Docs” sidebar, you’ll see all the information we entered about our Schema - the RootQuery, the description, and it’s viewer field:

¹¹⁴<http://localhost:3000/graphql>

< Schema **RootQuery** X

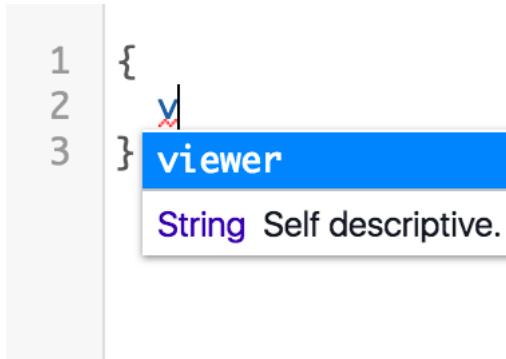
The root query

FIELDS

viewer: String

Server Docs

You'll also get auto-complete for our fields:



A screenshot of a GraphQL schema editor showing auto-complete for the 'viewer' field. The code editor shows lines 1, 2, and 3, with the cursor on line 3 after a brace. A tooltip for 'viewer' is displayed, showing its type as 'String' and the description 'Self descriptive.'

Server Autocomplete

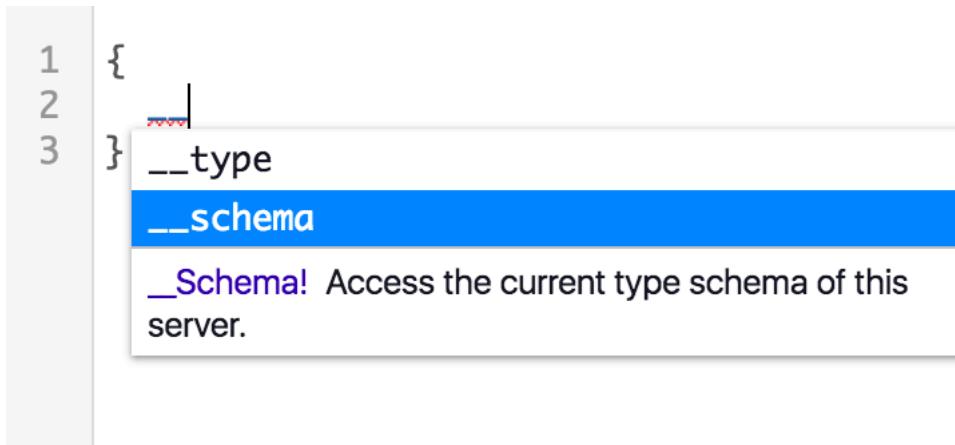
We get all of this goodness for free by using `graphql-express`.



It's also possible to setup GraphiQL if you're using another JavaScript framework or an entirely different language, read [GraphiQL's documentation¹¹⁵](#) for details.

You'll notice that our typeahead suggests some interesting fields like `__schema`, even though we didn't define that. This is something we've alluded to throughout the chapter: GraphQL's introspection features.

¹¹⁵<https://github.com/graphql/graphiql#getting-started>



Server Introspection

Let's dig into it a bit further.

Introspection

Go ahead and run this GraphQL query inside of our server's GraphiQL instance:

```
1 {
2   __schema {
3     queryType {
4       name
5       fields {
6         name
7         type {
8           name
9         }
10      }
11    }
12  }
13 }
```

`__schema` is a “meta” field that automatically exists on every GraphQL root query operation. It has a whole tree of its own types and their fields, which you can read about in the [GraphQL introspection spec¹¹⁶](#). GraphiQL’s auto-complete will also give you helpful information and let you explore the possibilities of `__schema` and the other introspection field, `__type`.

After running that query, you'll get some data back like this:

¹¹⁶<https://facebook.github.io/graphql/#sec-Schema-Introspection>

```

1  {
2    "data": {
3      "__schema": {
4        "queryType": {
5          "name": "RootQuery",
6          "fields": [
7            {
8              "name": "viewer",
9              "type": {
10                "name": "String"
11              }
12            }
13          ]
14        }
15      }
16    }
17  }

```

This is essentially a JSON description of our server's schema. This is how GraphiQL populates it's documentation and auto-complete, by issuing an introspection query as the IDE boots up. Since every GraphQL server is required to support introspection, tooling is often portable across all GraphQL servers (again, regardless of the language or library they were implemented with).

We won't do anything else with introspection for this chapter, but it's good to know that it exists and how it works.

Mutation

So far we've set the root query of our schema, but we mentioned that we can also set the root mutation. Remember from earlier that mutations are the correct place for "writes" to happen in GraphQL - whenever you want to add, update, or remove data, it should occur through a mutation operation. As we'll see, mutations differ very little from queries other than the implicit contract that writes should **not** happen in queries.

To demonstrate mutations, we'll create a simple API to get and set in-memory node objects, similar to the idiomatic Node pattern we described earlier. For now, instead of returning objects that implement a Node interface, we're going to return a string for our nodes.

Let's start by importing some new dependencies from the GraphQL library:

```

8 import { GraphQLSchema, GraphQLObjectType, GraphQLString,
9 GraphQLNonNull, GraphQLID } from 'graphql';

```

GraphQLID is the JavaScript analog to the ID scalar type, while GraphQLNonNull is something we haven't quite covered yet. It turns out that GraphQL's type system not only tracks the types and interfaces of fields, but also whether or not they can be null. This is especially handy for field arguments, which we'll get to in a second.

Next we need to create a type for our mutation. Take a second to think about what that code will look like, given that it should be fairly similar to our RootQuery type. What kinds of invocations and properties will we need?

After you've given it some thought, compare it to the actual implementation:

```
35 let inMemoryStore = {};
36 const RootMutation = new GraphQLObjectType({
37   name: 'RootMutation',
38   description: 'The root mutation',
39   fields: {
40     setNode: {
41       type: GraphQLString,
42       args: {
43         id: {
44           type: new GraphQLNonNull(GraphQLID)
45         },
46         value: {
47           type: new GraphQLNonNull(GraphQLString),
48         }
49       },
50       resolve(source, args) {
51         inMemoryStore[args.key] = args.value;
52         return inMemoryStore[args.key];
53       }
54     }
55   }
56});
```

At the very top we initialize a "store" for our nodes. This will only live in-memory, so whenever you restart the server it will clear the data - but you can start to imagine this being a key-value service like Redis or Memcached. Creating an instance of GraphQLObjectType, setting the name, description, and fields should all look familiar.

One new thing here is dealing with field arguments using the args property. Similar to how we set the names of fields with the fields property, the keys of the args object are the names of the arguments allowed by the field.

We specify each argument's type, wrapped as an instance of GraphQLNonNull. For arguments, this means that the query **must** specify a non-null value for each argument. Our resolve function

takes this into account - `resolve` gets passed several arguments, the second of which is an object containing the field arguments (we'll touch on `source` later on).

When we set a value in `inMemoryStore`, that is the “write” of our mutation. We also return a value in the `resolve` function, to cooperate with the type of the `setNode` field. In this case it happens to be a string, but you can imagine it being a complex type like `User` or something bespoke for the mutation.

Now that we have our mutation type, we add it to our schema:

```
58 const Schema = new GraphQLSchema({
59   query: RootQuery,
60   mutation: RootMutation,
61 });
```

For one last bit of housekeeping, we'll go ahead and add a `node` field to our query:

```
14   fields: {
15     viewer: {
16       type: GraphQLString,
17       resolve() {
18         return 'viewer!';
19       }
20     },
21     node: {
22       type: GraphQLString,
23       args: {
24         id: {
25           type: new GraphQLNonNull(GraphQLID)
26         }
27       },
28       resolve(source, args) {
29         return inMemoryStore[args.key];
30       }
31     }
32 }
```

Restart your server and give this query a run in GraphiQL:

```
1 mutation {
2   setNode(id: "id", value: "a value!")
3 }
```

Notice that we have to explicitly state the `mutation` operation type, since GraphQL assumes query otherwise. Running the mutation should return the new value:

```
1 {
2   "data": {
3     "node": "a value!"
4   }
5 }
```

You can then confirm that the mutation worked by running a fresh query:

```
1 query {
2   node(id: "id")
3 }
```

Mutations aren't too conceptually complicated, and most apps will need them to write data back to the server eventually. Relay has a slightly more rigorous pattern to defining mutations, which we'll get to in due time.

This mutation changed an in-memory data structure, but most products' data lives in datastores like Postgres or MySQL. It's time to explore how we work with those environments.

Rich Schemas and SQL

As we mentioned earlier, we're going to build a small Facebook clone using SQLite. It's going to show how to communicate with a database, how to handle authorization and permissions, and some performance tips to make it runs smoothly.

Before we dive into the code, here's what our database is going to look like:

- A `users` table, which has `id`, `name`, and `about` columns
- A `users_friends` table, which has `user_id_a`, `user_id_b`, and `level` columns
- A `posts` column, which has `body`, `user_id`, `level`, and `created_at` columns

The `level` columns are going to represent a hierarchical “privacy” setting for friendships and posts. The possible levels we'll use are `top`, `friend`, `acquaintance`, and `public`. If a post has a `level` of `acquaintance`, then only friends with that level or “higher” can see it. `public` posts can be seen by anyone, even if the user isn't a friend.

To implement this in NodeJS, we're going to use the [node-sql¹¹⁷](#) and [sqlite3¹¹⁸](#) packages. There are many options when working with database in NodeJS, and you should do your own research before using any mission-critical libraries in production, but these should suffice for our learning.

Setting Up The Database

Time to install some more libraries! Run this to add them to our project:

```
$ npm install sqlite@0.0.4 sqlite3@3.1.3 --save --save-exact
$ mkdir src
$ touch src/tables.js src/database.js src/seedData.js
```

Now let's create a database. SQLite databases exist in files, so there's no need to start a separate process or install more dependencies. For legibility, we're going to start splitting our code into multiple files, which is the set of files we created with touch.

First we define our tables - you can read node-sql's documentation for specifics, but it's pretty legible. Open up `tables.js` and add these definitions:

```
1 import sql from 'sql';
2
3 sql.setDialect('sqlite');
4
5 export const users = sql.define({
6   name: 'users',
7   columns: [
8     {
9       name: 'id',
10      dataType: 'INTEGER',
11      primaryKey: true
12    },
13    {
14      name: 'name',
15      dataType: 'text'
16    },
17    {
18      name: 'about',
19      dataType: 'text'
20    }
21  });
22
23
24 export const usersFriends = sql.define({
25   name: 'users_friends',
```

¹¹⁷<https://github.com/brianc/node-sql>

¹¹⁸<https://github.com/mapbox/node-sqlite3>

```
22  columns: [{  
23    name: 'user_id_a',  
24    dataType: 'int',  
25  }, {  
26    name: 'user_id_b',  
27    dataType: 'int',  
28  }, {  
29    name: 'level',  
30    dataType: 'text',  
31  }]  
32});  
33  
34 export const posts = sql.define({  
35  name: 'posts',  
36  columns: [{  
37    name: 'id',  
38    dataType: 'INTEGER',  
39    primaryKey: true  
40  }, {  
41    name: 'user_id',  
42    dataType: 'int'  
43  }, {  
44    name: 'body',  
45    dataType: 'text'  
46  }, {  
47    name: 'level',  
48    dataType: 'text'  
49  }, {  
50    name: 'created_at',  
51    dataType: 'datetime'  
52  }]  
53});
```

node-sql lets us craft and manipulate SQL queries using JavaScript objects, similar to how the GraphQL JavaScript library enables us to work with GraphQL. There's nothing "happening" in this particular file, but we'll consume the objects it exports soon.

Now we need to create the database and load it with some data. Included with the materials for this course, inside the `graphql-server/src` directory, is a `data.json` file. Go ahead and copy that into the `src` directory of the project we're working on. You can also come up with your own data, but the examples we'll work through will assume you're using the included data file.

To copy the data from `data.json` into the database, we need to write a bit of code. First open up `src/database.js` and define some simple exports:

```
1 import sqlite3 from 'sqlite3';
2
3 import * as tables from './tables';
4
5 export const db = new sqlite3.Database('./db.sqlite');
6
7 export const getSql = (query) => {
8     return new Promise((resolve, reject) => {
9         console.log(query.text);
10        console.log(query.values);
11        db.all(query.text, query.values, (err, rows) => {
12            if (err) {
13                reject(err);
14            } else {
15                resolve(rows);
16            }
17        });
18    });
19}
```

First we define our database file and export it so other code can use it. We also export a `getSql` function, which will run our queries as asynchronous promises. The GraphQL JS library uses promises to handle asynchronous resolving, which we'll leverage soon.

Then open up the `src/seedData.js` file we created earlier and start by adding this `createDatabase` function:

```
1 import * as data from './data';
2 import * as tables from './tables';
3 import * as database from './database';
4
5 const sequencePromises = function (promises) {
6     return promises.reduce((promise, promiseFunction) => {
7         return promise.then(() => {
8             return promiseFunction();
9         });
10    }, Promise.resolve());
11 };
12
13 const createDatabase = () => {
14     let promises = [tables.users, tables.usersFriends, tables.posts].map((table) =>
15     {
16         return () => database.getSql(table.create().toQuery());
17     });
18}
```

```
18
19     return sequencePromises(promises);
20 };
```

Recall that the exports of `tables.js` are the objects created by `node-sql`. When we want to create a database query with `node-sql`, we use the `.toQuery()` function and then pass it into the `getSql` function we just wrote in `database.js`. In plain-English, our new `createDatabase` function runs queries that create each table. We use `Promise.all` to make sure all of the tables are created before moving on to any next steps in the promise chain.

After the tables are setup, we need to add our data from `data.json`. Write this new `insertData` function next:

```
21 const insertData = () => {
22     let { users, posts, usersFriends } = data;
23
24     let queries = [
25         tables.users.insert(users).toQuery(),
26         tables.posts.insert(posts).toQuery(),
27         tables.usersFriends.insert(usersFriends).toQuery(),
28     ];
29
30     let promises = queries.map((query) => {
31         return () => database.getSql(query);
32     });
33     return sequencePromises(promises);
34 };
```

Similar to `createDatabase`, we create queries using `toQuery` and then execute them using `getSql`. Finally we tie it all together at the end of the file by invoking our two functions:

```
36 createDatabase().then(() => {
37     return insertData();
38 }).then(() => {
39     console.log({ done: true });
40 });
```

To get this code to run, you can invoke this shell command:

```
$ node -e 'require("babel-register"); require("./src/seedData");'  
{ done: true }
```

You should now have a `db.sqlite` file at the top of your project! You can use many graphical tools to explore your SQLite database, such as [DBeaver¹¹⁹](#), or you can verify that it has some data with this one-line command:

```
$ sqlite3 ./db.sqlite "select count(*) from users"  
5
```

Now it's time to hook up the GraphQL schema to our newly created database.

Schema Design

In our earlier examples, the `resolve` functions in our GraphQL schema would just return constant or in-memory values, but now they need to return data from our database. We also need corresponding GraphQL types for our `users` and `posts` table, and to add the corresponding fields to our original root query.

Before we dive into the code, we should take a minute to consider how our final GraphQL queries are going to look. Generally it's a good practice to start with the `viewer`, since most of our data will flow through that field.

In this case, the `viewer` will be the current user. A user has `friends`, so we'll need a list for that, as well as a connection for the `posts` authored by that user. The `viewer` also has a `newsfeed`, which is a connection to posts authored by other users. We'll want all of these connections to be idiomatic GraphQL and include features like proper pagination, in case the entire newsfeed or friends list gets too long to compute or send in one response.

Given all of that information, we expect our `viewer` to enable queries like this:

```
1  {  
2    viewer {  
3      friends {  
4        # connection fields for Users  
5      }  
6  
7      posts {  
8        # connection fields for Posts  
9      }  
10     newsfeed {
```

¹¹⁹<http://dbeaver.jkiss.org/>

```
12      # connection fields for Posts
13  }
14 }
15 }
```

Remember that we want all of our objects to also be nodes in a graph, in accordance with idiomatic GraphQL. Eventually, all of the data returned in `newsfeed` will need to be authorization-aware, taking into account the friendship level between the author of a given post and the viewer.

We should add support for queries that fetch arbitrary nodes using a top-level `node(id:)` field like this:

```
1 {
2   node(id: "123") {
3     ... on User {
4       friends {
5         # friends
6       }
7     }
8
9   ... on Post {
10    author {
11      posts {
12        # connection fields
13      }
14    }
15
16    body
17  }
18 }
19 }
```

As we mentioned in an earlier section, this field is valuable to help front-end code re-fetch the current state of any node without knowing its position in the hierarchy.

This may be surprising, but the GraphQL convention is to fetch lots of different types of objects from the same top-level `node` field. SQL databases usually have a different table for each type, and REST APIs use distinct endpoints per type, but idiomatic GraphQL fetches all objects the same way as long as you have an identifier. This also means that IDs need to be globally unique, or else a GraphQL server can't tell the difference between a `User` with `id: 1` and a `Post` with `id: 1`.

Object and Scalar Types

To start making these queries possible, create a new file to hold these types called `types.js`:

```
$ touch src/types.js
```

The first type we need to define is the `Node` interface. Recall that for a type to be a valid `Node`, it needs to have a globally-unique `id` field. To implement that in JavaScript, we start by importing some APIs and creating an instance of `GraphQLInterfaceType`:

```
1 import {
2   GraphQLInterfaceType,
3   GraphQLObjectType,
4   GraphQLID,
5   GraphQLString,
6   GraphQLNonNull,
7   GraphQLList,
8 } from 'graphql';
9
10 import * as tables from './tables';
11
12 export const NodeInterface = new GraphQLInterfaceType({
13   name: 'Node',
14   fields: {
15     id: {
16       type: new GraphQLNonNull(GraphQLID)
17     }
18   }
19 }
```

Aside from using a new class, everything looks familiar to how we create instance of `GraphQLObjectType`. Notice that the `id` field does *not* have a `resolve` function. Fields in interfaces are not expected to have default implementations of `resolve`, and even if you do provide one it will be ignored. Instead, each object type that implements `Node` should define its own `resolve` (we'll get to that in a second).

In addition to defining `fields`, `GraphQLInterfaceType` instances must also define a `resolveType` function. Remember that our top-level `node(id:)` field only guarantees that some kind of `Node` will be returned, it does not make any guarantees about which specific type. In order for GraphQL to do further resolution (such as using partial fragments, i.e. `... on User`), we need to inform the GraphQL engine of the concrete type for a particular object.

We can implement it like this:

```

11 export const NodeInterface = new GraphQLInterfaceType({
12   name: 'Node',
13   fields: {
14     id: {
15       type: new GraphQLNonNull(GraphQLID)
16     }
17   },
18   resolveType: (source) => {
19     if (source.__tableName === tables.users.getName()) {
20       return UserType;
21     }
22     return PostType;
23   }
24 });

```

The `resolveType` function takes in raw data as its first argument (in this case, `source` will be the data returned directly from the database), and is expected to return an instance of `GraphQLObjectType` that implement the interface. We use the `__tableName` property of `source`, which isn't an actual column in the database - we'll see how that gets injected later.

We return some objects, `UserType` and `PostType`, that we haven't defined quite yet. Add their definitions below `resolveType`:

```

26 const resolveId = (source) => {
27   return tables.dbIdToNodeId(source.id, source.__tableName);
28 };
29
30 export const UserType = new GraphQLObjectType({
31   name: 'User',
32   interfaces: [ NodeInterface ],
33   fields: {
34     id: {
35       type: new GraphQLNonNull(GraphQLID),
36       resolve: resolveId
37     },
38     name: {
39       type: new GraphQLNonNull(GraphQLString)
40     },
41     about: {
42       type: new GraphQLNonNull(GraphQLString)
43     }
44   }
45 });

```

```

46
47 export const PostType = new GraphQLObjectType({
48   name: 'Post',
49   interfaces: [ NodeInterface ],
50   fields: {
51     id: {
52       type: new GraphQLNonNull(GraphQLID),
53       resolve: resolveId
54     },
55     createdAt: {
56       type: new GraphQLNonNull(GraphQLString),
57     },
58     body: {
59       type: new GraphQLNonNull(GraphQLString)
60     }
61   }

```

Most of the code should look similar to everything we've been working with so far. We define new instances of `GraphQLObjectType`, add `NodeInterface` to their `interfaces` property, and implement their `fields`. Some of the `fields` are wrapped in `GraphQLNonNull`, which enforces that they must exist. If you don't provide an implementation of `resolve` to a field, it will do a simple property lookup on the underlying source data - for example, the `name` field will invoke `source.name`.

We do provide an implementation of `resolve` for `id` on both of these types, which both use the same `resolveId` function. Even though our `NodeInterface` code couldn't provide a default `resolve`, we can still share code by referencing the same variable.

Our implementation of `resolveId` uses `tables.dbIdToNodeId`, which we haven't defined in `tables.js`. Open up `tables.js` and add these two new exported functions at the bottom:

```

55 export const dbIdToNodeId = (dbId, tableName) => {
56   return `${tableName}:${dbId}`;
57 };
58
59 export const splitNodeId = (nodeId) => {
60   const [tableName, dbId] = nodeId.split(':');
61   return { tableName, dbId };
62 };

```

Earlier we mentioned that Node IDs must be globally unique, but looking at our `data.json` we have some row ID collisions. This is not uncommon in relational databases like Postgres and MySQL, so we have to write some logic which coerces the row integer ID into a unique string. In a production application, you may want to obfuscate IDs to leak less information about your database, but using the raw table name will make it easier to debug for now.

We're almost ready to run a GraphQL query! Head to `server.js`, import some of the types we just authored, and change the `RootQuery` to have only the new node field that we want:

```
11 import {
12   NodeInterface,
13   UserType,
14   PostType
15 } from './src/types';
16
17 import * as loaders from './src/loaders';
18
19 const RootQuery = new GraphQLObjectType({
20   name: 'RootQuery',
21   description: 'The root query',
22   fields: {
23     node: {
24       type: NodeInterface,
25       args: {
26         id: {
27           type: new GraphQLNonNull(GraphQLID)
28         }
29       },
30       resolve(source, args) {
31         return loaders.getNodeById(args.id);
32       }
33     }
34   }
35});
```

We also imported a new file, `src/loaders.js`, which we need to edit. Its purpose will be to expose APIs that load data from the source - we don't want to clutter our server or top-level schema code with code that directly accesses the database.

Create that `src/loaders.js` file and add this small function:

```
1 import * as database from './database';
2 import * as tables from './tables';
3
4 export const getNodeById = (nodeId) => {
5   const { tableName, dbId } = tables.splitNodeId(nodeId);
6
7   const table = tables[tableName];
8   const query = table
9     .select(table.star())
10    .where(table.id.equals(dbId))
11    .limit(1)
12    .toQuery();
13
14  return database.getSql(query).then((rows) => {
15    if (rows[0]) {
16      rows[0].__tableName = tableName;
17    }
18    return rows[0];
19  });
20};
```

This should look similar to the code we wrote in `seedData` - we use the `node-sql` APIs to construct a SQL query based on the `nodeId` provided. Remember that this `nodeId` is the globally-unique node ID, not a row ID, so we extract the database-specific information with `tables.splitNodeId`.

One trick we perform is attaching the `__tableName` property. This helps us in our earlier `resolveType` function, and anywhere else we may need to tie an object back to its underlying table. It's safe to add this because we don't expose the `__tableName` property explicitly in the GraphQL schema, so malicious consumers can't access it.

A very subtle but important thing happening with all of this code is that `loaders.getNodeById` returns a Promise (which is why we can attach the `__tableName` using `.then`), and ultimately that promise is returned in `resolve`. Promises are how GraphQL handles asynchronous field resolution, which usually occurs with database queries and any third-party API calls. If you're using an API that doesn't natively support promises, you can refer to the [Promise API¹²⁰](#) to convert callback-based APIs to promises.

One last step, we need to tell our `GraphQLSchema` about all the possible types in our schema. You have to do this if you use interfaces, so GraphQL can compute the list of types that implement any interfaces.

¹²⁰https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise#Creating_a_Promise

```

60 const Schema = new GraphQLSchema({
61   types: [UserType, PostType],
62   query: RootQuery,
63   mutation: RootMutation,
64 });

```

And that's it! Restart your server, open up GraphiQL (still available at <http://localhost:3000/graphql>¹²¹), and try this query:

```

1 {
2   node(id:"users:4") {
3     id
4     ... on User {
5       name
6     }
7   }
8 }

```

You should see this data returned:

```

1 {
2   "data": {
3     "node": {
4       "id": "users:4",
5       "name": "Roger"
6     }
7   }
8 }

```

You can play around with other node IDs, such as "posts:4" and . . . on Post. Before we move on, reflect on what's going on under the hood: we request a particular node ID, which invokes a database call, and then each field gets resolve'd against the database source data.

We could write a very simple front-end app against our current server at this point, but we have more of our schema to fill-out. Let's work on some of those friends list and posts connections.

Lists

We mentioned earlier that the friends field should return a list of User types. Let's take baby steps towards that goal and return a simple list of IDs for now.

First let's edit our UserType. Open up `types.js` and a new friends field:

¹²¹<http://localhost:3000/graphql>

```

43     about: {
44       type: new GraphQLNonNull(GraphQLString)
45     },
46     friends: {
47       type: new GraphQList(GraphQLID),
48       resolve(source) {
49         return loaders.getFriendIdsForUser(source).then((rows) => {
50           return rows.map((row) => {
51             return tables.dbIdToNodeId(row.user_id_b, row.__tableName);
52           });
53         })
54       }
55     }

```

We set the `friends` field to return a `GraphQList` of IDs. `GraphQList` works similarly to the `GraphQLNonNull` we saw earlier, wrapping an inner type in its constructor. Inside `resolve` we invoke a new loader (to-be-written), and then coerce its results to the globally-unique IDs we expect.

Add an `import` at the top of the file to prepare for our new loader:

```

10 import * as tables from './tables';
11 import * as loaders from './loaders';

```

Edit `loaders.js` with that new `getFriendIdsForUser` function:

```

21 export const getFriendIdsForUser = (userSource) => {
22   const table = tables.usersFriends;
23   const query = table
24     .select(table.user_id_b)
25     .where(table.user_id_a.equals(userSource.id))
26     .toQuery();
27
28   return database.getSql(query).then((rows) => {
29     rows.forEach((row) => {
30       row.__tableName = tables.users.getName();
31     });
32     return rows;
33   });
34 };

```

That's all the new code we have to write - fire up GraphiQL and run this query:

```
1 {
2   node(id:"users:4") {
3     id
4     ... on User {
5       name
6       friends
7     }
8   }
9 }
```

Confirm that your results are equal to this:

```
1 {
2   "data": {
3     "node": {
4       "id": "users:4",
5       "name": "Roger",
6       "friends": [
7         "users:1",
8         "users:3",
9         "users:2"
10      ]
11    }
12  }
13 }
```

Performance: Look-Ahead Optimizations

This is great, but there's some sneaky business going on under-the-hood that we should explore. When we resolve the original node field, that executes one database query (`loaders.getNodeById`). Then after that node is resolved, we execute *another* database query (`loaders.getFriendIdsForUser`). If you extend this pattern to a larger application, you can imagine lots of database queries getting fired - at worst, one per field. But we know that in SQL it's possible to express these two queries with one efficient SQL query.

It turns out the GraphQL library provides a way of doing these sort of optimizations, where we want to "look ahead" at the rest of the GraphQL query and perform more efficient resolution calls. It may not be appropriate to do this in every case, but certain products and workloads may benefit tremendously.

Open up `server.js` and let's do a bit of work on the `node` field's `resolve` function. Aside from `source` and `args`, GraphQL passes two more variables to `resolve`: `context` and `info`. We'll use `context` later to help with authentication and authorization; `info` is a bag of objects, including an *abstract syntax tree (AST)* of the entire GraphQL query. We'll do a simple traversal of the AST and determine if we should run a more efficient loader:

```
23     node: {
24       type: NodeInterface,
25       args: {
26         id: {
27           type: new GraphQLNonNull(GraphQLID)
28         }
29       },
30       resolve(source, args, context, info) {
31         let includeFriends = false;
32
33         const selectionFragments = info.fieldASTs[0].selectionSet.selections;
34         const userSelections = selectionFragments.filter((selection) => {
35           return selection.kind === 'InlineFragment' && selection.typeCondition.\
36 name.value === 'User';
37         })
38
39         userSelections.forEach((selection) => {
40           selection.selectionSet.selections.forEach((innerSelection) => {
41             if (innerSelection.name.value === 'friends') {
42               includeFriends = true;
43             }
44           });
45         });
46
47         if (includeFriends) {
48           return loaders.getUserNodeWithFriends(args.id);
49         }
50         else {
51           return loaders.getNodeById(args.id);
52         }
53       }
54     }
```

There's a lot happening as we traverse the AST, and we won't go into detail on most of the specifics. If you end up performing these look-ahead optimizations in your code, you can `console.log` each level of the tree and determine what information you can access.

Essentially we look for `User` fragments on the `node` field's selection set, and determine if the fragment accesses the `friends` field. If it does, then we run a new loader; else, we fall back to the original loader.

Now let's take a look at the new `loaders.getUserNodeWithFriends` function:

```
37 export const getNodeWithFriends = (nodeId) => {
38   const { tableName, dbId } = tables.splitNodeId(nodeId);
39
40   const query = tables.users
41     .select(tables.usersFriends.user_id_b, tables.users.star())
42     .from(
43       tables.users.leftJoin(tables.usersFriends)
44         .on(tables.usersFriends.user_id_a.equals(tables.users.id))
45     )
46     .where(tables.users.id.equals(dbId))
47     .toQuery();
48
49
50   return database.getSql(query).then((rows) => {
51     if (!rows[0]) {
52       return undefined;
53     }
54
55     const __friends = rows.map((row) => {
56       return {
57         user_id_b: row.user_id_b,
58         __tableName: tables.users.getName()
59       }
60     });
61
62     const source = {
63       id: rows[0].id,
64       name: rows[0].name,
65       about: rows[0].about,
66       __tableName: tableName,
67       __friends: __friends
68     };
69     return source;
70   });
71 }
```

This starts getting a bit complicated, and very specific to this product and the frameworks we chose - which is a common pattern when working on performance optimizations. Our SQL query now grabs all of the friends and the user's profile simultaneously, eliminating a database round-trip. We then load those friends into a `__friends` property (we've chosen to continue the `__` prefix for "private" properties), and can access it inside of `types.js`:

```

46     friends: {
47       type: new GraphQLList(GraphQLID),
48       resolve(source) {
49         if (source.__friends) {
50           return source.__friends.map((row) => {
51             return tables.dbIdToNodeId(row.user_id_b, row.__tableName);
52           });
53         }
54
55         return loaders.getFriendIdsForUser(source).then((rows) => {
56           return rows.map((row) => {
57             return tables.dbIdToNodeId(row.user_id_b, row.__tableName);
58           });
59         })
60       }
61     }

```

Other applications might perform look-ahead optimizations differently - instead of making globbing multiple SQL queries into one, they might warm a cache in the background. The important takeaway is that `resolve` accepts many arguments which let you short-circuit the usual recursive resolution flow.

Lists Continued

Our `friends` field returns a complete list of IDs (performatly, might I add), but what we really want is a list to their full `User` types. For large lists, we'd probably want to use an idiomatic GraphQL connection (which we'll implement soon) but we'll allow the `friends` field to return all entries on each query.

We'll start by removing the logic we added for performance optimization. Since our application is about to change a bit, we can revisit performance once its capabilities have stabilized.

```

30       resolve(source, args, context, info) {
31         return loaders.getNodeById(args.id);
32       }

```

Next we need to change the type returned by our `friends` field to a list of `User`. Since we already have a loader for loading arbitrary nodes by ID, there's not much code we need to write:

```
32 export const UserType = new GraphQLObjectType({
33   name: 'User',
34   interfaces: [ NodeInterface ],
35   // Note that this is now a function
36   fields: () => {
37     return {
38       id: {
39         type: new GraphQLNonNull(GraphQLID),
40         resolve: resolveId
41       },
42       name: { type: new GraphQLNonNull(GraphQLString) },
43       about: { type: new GraphQLNonNull(GraphQLString) },
44       friends: {
45         type: new GraphQList(UserType),
46         resolve(source) {
47           return loaders.getFriendIdsForUser(source).then((rows) => {
48             const promises = rows.map((row) => {
49               const friendNodeId = tables.dbIdToNodeId(row.user_id_b, row.__tabl\
50 eName);
51               return loaders.getNodeById(friendNodeId);
52             });
53             return Promise.all(promises);
54           })
55         }
56       };
57     };
58   }
59 });
```

We now set the `friends` type to `GraphQList(UserType)`. Because of how JavaScript variable hoisting works, we have to change the `fields` property to a function instead of an object in order to pick up a “recursive” type definition (where a type returns a field of itself). We invoke `loaders.getNodeById` on all of the IDs we previously retrieved and voila! Restart your server and execute this kind of query in GraphiQL:

```
1  {
2    node(id:"users:4") {
3      ... on User {
4        friends {
5          id
6          about
7          name
8        }
9      }
10    }
11 }
```

Which should return this data:

```
1  {
2    "data": {
3      "node": {
4        "friends": [
5          {
6            "id": "users:1",
7            "about": "Sports!",
8            "name": "Harry"
9          },
10         {
11           "id": "users:3",
12           "about": "Love books",
13           "name": "Hannah"
14         },
15         {
16           "id": "users:2",
17           "about": "I'm the best",
18           "name": "David"
19         }
20       ]
21     }
22   }
23 }
```

You can even go a step further and query the `friends of friends!`

Connections

Now we want to implement idiomatic connection fields. Instead of returning a simple list, we're going to return a more complicated (but powerful) structure. Although there is additional work, connections fields are most appropriate for lists that would otherwise be large or unbounded. It might be prohibitive or wasteful to return a huge list in one query, so GraphQL schemas prefer to break up these fields into smaller paginated chunks.

Instead of using literal page numbers, recall from the last chapter that idiomatic GraphQL uses opaque strings called *cursors*. Cursors are more resilient to real-time changes to your data, which might lead to duplicates in simple page-based systems. The `pageInfo` field of a connection gives metadata to help with making new requests, while the `edges` field will hold the actual data for each item.

Instead of the previous query which used lists for `friends`, we want something like this for `posts`:

```
1  {
2    node(id:"users:1") {
3      ... on User {
4        posts(first: 1) {
5          pageInfo {
6            hasNextPage
7            hasPreviousPage
8            startCursor
9            endCursor
10           }
11          edges {
12            cursor
13            node {
14              id
15              body
16            }
17          }
18        }
19      }
20    }
21 }
```

Instead of just returning a list of `PostType`, the `posts` field will now return a `PostsConnection` type.

Define the `PageInfo`, `PostEdge`, and `PostsConnection` types in your `types.js`, in addition to importing more types from the `graphql` library:

```
1 import {
2   GraphQLInterfaceType,
3   GraphQLObjectType,
4   GraphQLID,
5   GraphQLString,
6   GraphQLNonNull,
7   GraphQLList,
8   GraphQLBoolean,
9   GraphQLInt,
10 } from 'graphql';
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34 const PageInfoType = new GraphQLObjectType({
35   name: 'PageInfo',
36   fields: {
37     hasNextPage: {
38       type: new GraphQLNonNull(GraphQLBoolean)
39     },
40     hasPreviousPage: {
41       type: new GraphQLNonNull(GraphQLBoolean)
42     },
43     startCursor: {
44       type: GraphQLString,
45     },
46     endCursor: {
47       type: GraphQLString,
48     }
49   }
50 });
51
52 const PostEdgeType = new GraphQLObjectType({
53   name: 'PostEdge',
54   fields: () => {
55     return {
56       cursor: {
57         type: new GraphQLNonNull(GraphQLString)
58       },
59       node: {
60         type: new GraphQLNonNull(PostType)
61       }
62     }
63   }
64 });
```

```

65
66 const PostsConnectionType = new GraphQLObjectType({
67   name: 'PostsConnection',
68   fields: {
69     pageInfo: {
70       type: new GraphQLNonNull(PageInfoType)
71     },
72     edges: {
73       type: new GraphQLList(PostEdgeType)
74     }
75   }

```

These are mostly just type definitions with no inherent `resolve` functions for now. Different applications will have different ways and patterns for resolving connections, so don't consider some of the implementation details here as the gospel for your own work.

Now we need to hook up our `UserType` to the new types we created and actually create the `posts` field.

```

100      }
101    },
102    posts: {
103      type: PostsConnectionType,
104      args: {
105        after: {
106          type: GraphQLString
107        },
108        first: {
109          type: GraphQLInt
110        },
111      },
112      resolve(source, args) {
113        return loaders.getPostIdsForUser(source, args).then(({ rows, pageInfo \
114 }) => {
115          const promises = rows.map((row) => {
116            const postNodeId = tables.dbIdToNodeId(row.id, row.__tableName);
117            return loaders.getNodeById(postNodeId).then((node) => {
118              const edge = {
119                node,
120                cursor: row.__cursor,
121              };
122              return edge;
123            });

```

```

124     });
125
126     return Promise.all(promises).then((edges) => {
127       return {
128         edges,
129         pageInfo
130       }
131     });
132
133   })
134 }
135 }
136 };
137 }
138 });

```

This should look familiar to how we implement our `friends` field, aside from the new arguments. Remember that this field does not return a list of `PostType` - it returns a `PostsConnectionType`, which is an object with `pageInfo` and `edges` keys.

We use a new `loader` method, `getPostIdsForUser`, and pass it the `args` to our resolver. We'll implement this loader very soon, and it will not only return the associated `rows` but also a `pageInfo` structure that corresponds to the `PageInfoType`. We then load the nodes for each of the identifiers and create the wrap them into a `PostEdgeType` with the row's cursor.

There are ways to make this more efficient at the JavaScript and database levels, but for now let's focus on making our code work by implementing `getPostIdsForUser`.

This loader will determine what data to fetch based on the pagination arguments and what cursors to assign the rows returned. The algorithm for slicing and pagination through your data based on the arguments is rather complex when supporting all possibilities, and you can read about it in detail in the [Relay specification¹²²](#). For brevity, we're only going to support the `after` and `first` arguments.

We start by defining the new function and parsing the arguments:

```

74 export const getPostIdsForUser = (userSource, args) => {
75   let { after, first } = args;
76   if (!first) {
77     first = 2;
78   }

```

In other words, if the user does not supply an argument for `first`, then we will return two posts. Then we start to construct our SQL query:

¹²²<https://facebook.github.io/relay/graphql/connections.htm#sec-Pagination-algorithm>

```

80  const table = tables.posts;
81  let query = table
82    .select(table.id, table.created_at)
83    .where(table.user_id.equals(userSource.id))
84    .order(table.created_at.asc)
85    .limit(first + 1);

```

We grab `first + 1` rows as a cheap method to determine if there are any more rows beyond what the user wants. Our query is ordered by `created_at ASC`, which is important in order to get deterministic data upon successive queries.

Next we account for an `after` cursor that may be passed:

```

87  if (after) {
88    // parse cursor
89    const [id, created_at] = after.split(':');
90    query = query
91      .where(table.created_at.gt(after))
92      .where(table.id.gt(id));

```

Our cursors in this example are strings composed of row IDs and row dates. Generally cursors will be based upon some date in most systems, since keeping IDs as incrementing integers is less common when working with high scale data.

We can finally execute our database query:

```

94  return database.getSql(query.toQuery()).then((allRows) => {
95    const rows = allRows.slice(0, first);
96
97    rows.forEach((row) => {
98      row.__tableName = tables.posts.getName();
99      row.__cursor = row.id + ':' + row.created_at;

```

Remember that we actually queried one more row than the user requested, which is why we have to slice the rows returned. We also construct the cursor for each row and set the `__tableName` property so that our future JOIN queries will work.

Now that we have our rows, we execute it and start to create our `pageInfo` object:

```
102     const hasNextPage = allRows.length > first;
103     const hasPreviousPage = false;
104
105     const pageInfo = {
106         hasNextPage,
107         hasPreviousPage,
108     };
109
110     if (rows.length > 0) {
111         pageInfo.startCursor = rows[0].__cursor;
112         pageInfo.endCursor = rows[rows.length - 1].__cursor;
113     }
}
```

Keeping a reference to `allRows` lets us calculate `hasNextPage`; because we don't support the `before` and `last` arguments, we always set `hasPreviousPage` to false. Setting `startCursor` and `endCursor` is as simple as grabbing the first and last elements of our `rows` array.

We return both the `rows` and `pageInfo` objects to finish the loader - finally! Restart your server and try the query we described at the start of the section:

```
1  {
2      node(id:"users:1") {
3          ... on User {
4              posts(first: 1) {
5                  pageInfo {
6                      hasNextPage
7                      hasPreviousPage
8                      startCursor
9                      endCursor
10                 }
11                 edges {
12                     cursor
13                     node {
14                         id
15                         body
16                     }
17                 }
18             }
19         }
20     }
21 }
```

In response you should get the first post by this user:

```
1  {
2      "data": {
3          "node": {
4              "posts": {
5                  "pageInfo": {
6                      "hasNextPage": true,
7                      "hasPreviousPage": false,
8                      "startCursor": "1:2016-04-01",
9                      "endCursor": "1:2016-04-01"
10                 },
11                 "edges": [
12                     {
13                         "cursor": "1:2016-04-01",
14                         "node": {
15                             "id": "posts:1",
16                             "body": "The team played a great game today!"
17                         }
18                     }
19                 ]
20             }
21         }
22     }
23 }
```

See that endCursor? Now try running a query with that cursor as the after value:

```
1  {
2      node(id:"users:1") {
3          ... on User {
4              posts(first: 1, after:"1:2016-04-01") {
5                  pageInfo {
6                      hasNextPage
7                      hasPreviousPage
8                      startCursor
9                      endCursor
10                 }
11                 edges {
12                     cursor
13                     node {
14                         id
15                         body
16                     }
17                 }
18             }
19         }
20     }
21 }
```

```

17      }
18    }
19  }
20}
21}

```

This returns the next (and final, judging from `hasNextPage`) post in the series:

```

1 {
2   "data": {
3     "node": {
4       "posts": {
5         "pageInfo": {
6           "hasNextPage": false,
7           "hasPreviousPage": false,
8           "startCursor": "2:2016-04-02",
9           "endCursor": "2:2016-04-02"
10        },
11        "edges": [
12          {
13            "cursor": "2:2016-04-02",
14            "node": {
15              "id": "posts:2",
16              "body": "Honestly I didn't do so well at yesterday's game, but eve\\
17  ryone else did."
18            }
19          }
20        ]
21      }
22    }
23  }
24}

```

Congratulations, you've now implemented cursor-based pagination! You might be able to use simple lists for static data, but using cursors prevents all kinds of frontend bugs and complexity for data that updates relatively often. It also allows you to leverage Relay's understanding of pagination and build paginated or infinite-scrolling UIs very quickly.

Authentication

Earlier we noted that in our social network the friendships have “levels,” which posts should respect. For example, if a post has a level of `friend`, then only my friends with a level of `friend` or higher (instead of `acquaintance` or a lower level) should see it.

This topic is generally referred to as *authorization*. GraphQL has no inherit notion or opinions on authorization, which makes it quite flexible for implementing controls on who can see what data in your schemas. This also means you need to take care to ensure that you're not accidentally exposing data that should be hidden to the user.

We're going to add a small *authentication* layer to our server, which verifies that the GraphQL query is allowed to be processed, as well as the authorization logic to control who can see the different posts. The techniques we'll use are definitely not the only ways to implement these features with GraphQL, but should spark some ideas that might apply to your product.

For authentication, we're going to use [HTTP basic authentication¹²³](#). There are a myriad of protocols for authentication, such as OAuth, JSON web tokens, and cookies, and the choice is ultimately very unique to each product. HTTP basic auth is fairly simple to add to our current NodeJS server, which is the primary reason in this case.

First, install the `basic-auth-connect` package, which provides a very simple API to allow certain credentials access:

```
$ npm i basic-auth-connect@1.0.0 --save --save-exact
```

Then in our server code, import the module:

```
3 import express from 'express';
4 import basicAuth from 'basic-auth-connect';
5
6 const app = express();
```

Right before we add our GraphQL endpoint, add a new call to `app.use`. Remember that Express will trigger each `app.use` function in the order they are added - if we put the new `basicAuth` function *after* our `graphqlHTTP` function, the ordering would be incorrect.

```
67 app.use(basicAuth(function(user, pass) {
68   return user === 'harry' && pass === 'mypassword1';
69 }));
70
71 app.use('/graphql', graphqlHTTP({ schema: Schema, graphiql: true }));
```

For now, we'll allow any user with the right password. Restart your server and try running this cURL command to test a simple query:

¹²³https://en.wikipedia.org/wiki/Basic_access_authentication

```
$ curl -XPOST -H 'content-type:application/graphql' http://localhost:3000/graphql \
1 -d '{ node(id:"users:4") { id } }'
Unauthorized
```

Since we didn't specify a username or password, our query fails. Try this next command to correctly pass our credentials:

```
$ curl -XPOST -H 'content-type:application/graphql' --user 1:mypassword1 http://\
localhost:3000/graphql -d '{ node(id:"users:4") { id } }'
{"data": {"node": {"id": "users:4"} }}
```

Great, now our authentication is working. You can also try this in Chrome and Firefox, which allow a GUI for entering the username and password.

The important concept here is that authentication is generally decoupled from a GraphQL schema. It's definitely possible to pass the username and password into a GraphQL query to authenticate the user (over HTTPS of course), but idiomatic GraphQL tends to separate the concerns.

Authorization

Now, onto tackling authorization. Remember in the last chapter we brought up the idea of a `viewer` field, which represents the logged-in users node in the data graph. We're going to add that field to our schema and allow our resolution code to be aware of the viewer's permissions.

By using `basic-auth-connect`, we can access to a `user` property on every Express request. The specifics on how you determine the user making each request will vary depending on your authentication library, but we simple need to take that `request.user` property and forward to our GraphQL resolver:

```
77 app.use('/graphql', graphqlHTTP((req) => {
78   const context = 'users:' + req.user;
79   return { schema: Schema, graphiql: true, context, pretty: true };
80 }));
```

Instead of always returning the same `schema` and `graphiql` settings for all requests, we now return a different configuration object for each GraphQL query. This new configuration has the `context` property set to the username, like the `user1` from the example earlier.

The next question is how do we access that `context` inside our GraphQL fields? Recall from earlier in the chapter that each `resolve` function gets passed some arguments. We've become very familiar with the `args` argument, but it turns out the `context` is also passed.

Here's how we add the `viewer` field with that knowledge:

```

20 const RootQuery = new GraphQLObjectType({
21   name: 'RootQuery',
22   description: 'The root query',
23   fields: {
24     viewer: {
25       type: NodeInterface,
26       resolve(source, args, context) {
27         return loaders.getNodeById(context);
28       }
29     },

```

If you restart your server, you should be able to cURL the endpoint like so:

```

$ curl -XPOST -H 'content-type:application/graphql' --user 1:mypassword1 http://\
localhost:3000/graphql -d '{ viewer { id } }'
{
  "data": {
    "viewer": {
      "id": "users:1"
    }
  }
}

```

You can explore using the `... on User` inline fragment to query more properties. We are able to provide this sort of consistent API with minimal code changes because we modeled our application data as a graph - neat!

Not only can top-level `viewer` field access the context, but *all* `resolve` functions have access, regardless of their depth in the hierarchy. This makes it very simple to add authorization checks to our `posts` field.

We start by forwarding the `context` argument in our `resolve` function:

```

112   resolve(source, args, context) {
113     return loaders.getPostIdsForUser(source, args, context).then(({ rows, \
114     pageInfo }) => {

```

GraphQL schemas should not handle authorization logic directly, which is likely to duplicate logic from your main codebase. Instead, that responsibility should fall into the underlying data loading libraries or services, as we show here.

Inside our `getPostIdsForUser`, we need to load each post's level from the database before we can use it. All we have to do is add it to our `select` arguments:

```

106 let query = table
107   .select(table.id, table.created_at, table.level)
108   .where(table.user_id.equals(userSource.id))
109   .order(table.created_at.asc)
110   .limit(first + 10);

```

In addition to running the database query to get all of the posts, we're also going to run another query to get all of the user access levels for our context. We'll use that list of levels to filter down the results our database query.

```

120 return Promise.all([
121   database.getSql(query.toQuery()),
122   getFriendshipLevels(context)
123 ]).then(([allRows, friendshipLevels]) => {
124   allRows = allRows.filter((row) => {
125     return canAccessLevel(friendshipLevels[userSource.id], row.level);
126   });
127   const rows = allRows.slice(0, first);

```

We're referencing two new functions that have yet to be implemented, `getFriendshipLevels` and `canAccessLevel`. Before we get to that, note that this does potentially introduce a bug into our system. We were calculating `hasNextPage` based on the length of `allRows`, but now `allRows` can be truncated depending on privacy settings. This highlights some of the complexity of systems that are highly aware of authorization; a naive mitigation of this is to just read more rows eagerly from the database, which we changed above (`first + 10`).

The `getFriendshipLevels` definition is similar to our other queries:

```

74 const getFriendshipLevels = (nodeId) => {
75   const { dbId } = tables.splitNodeId(nodeId);
76
77   const table = tables.usersFriends;
78   let query = table
79     .select(table.star())
80     .where(table.user_id_a.equals(dbId));
81
82   return database.getSql(query.toQuery()).then((rows) => {
83     const levelMap = {};
84     rows.forEach((row) => {
85       levelMap[row.user_id_b] = row.level;
86     });
87     return levelMap;
88   });
89 };

```

At the end we transform the array of rows into an object for a slightly more efficient API (you can also implement this transformation using a single `reduce` function if you'd like).

The last piece is `canAccessLevel`. Because our privacy settings are totally linear, we can represent the settings as an array and use the indices as a simple comparison:

```
91 const canAccessLevel = (viewerLevel, contentLevel) => {
92   const levels = ['public', 'acquaintance', 'friend', 'top'];
93   const viewerLevelIndex = levels.indexOf(viewerLevel);
94   const contentLevelIndex = levels.indexOf(contentLevel);
95
96   return viewerLevelIndex >= contentLevelIndex;
97 };
```

That all wasn't too bad, was it? We can test this out with some queries. Login as user 1 (so username 1 and password `mypassword1`) and run this query:

```
1  {
2    node(id:"users:2") {
3      ... on User {
4        posts {
5          edges {
6            node {
7              id
8              ... on Post {
9                body
10               }
11             }
12           }
13         }
14       }
15     }
16 }
```

In return, you'll see no posts. This is because our context (user 1) is not friends with the node we're accessing (user 2), and their posts have a level of `friend`.

Now open an incognito window, login as user 5, run that same query. You'll see a post!

```
1  {
2      "data": {
3          "node": {
4              "posts": {
5                  "edges": [
6                      {
7                          "node": {
8                              "id": "posts:3",
9                              "body": "Hard at work studying for finals..."
10                         }
11                     }
12                 ]
13             }
14         }
15     }
16 }
```

This is because user 5 is actually a friend of user 2.

This is a simple example, but highlights a few points about GraphQL.

- GraphQL server libraries typically allow you to forward on some kind of query-level context
- Your GraphQL schema code should not concern itself with authorization logic, instead deferring to your underlying data code

We've been reading data from our server for a bit, but now we should try out changing some of it with mutations.

Rich Mutations

There's only so much your application can do if it can only read data from the server - more often than not, we have to upload some new data. Recall from the last chapter that in GraphQL, we call these updates *mutations*.

We're going to add a mutation to our schema which creates a new post. The field will have a string arguments for the post body and a friendship privacy level, and it will allow us to query more information about the resulting post object.

Earlier in this chapter we added a simple key-value mutation in our `server.js` file. Let's update that definition to use the arguments and types we expect for creating a new post:

```
9 import { GraphQLSchema, GraphQLObjectType, GraphQLString,
10   GraphQLNonNull, GraphQLID, GraphQLEnumType } from 'graphql';
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44 const LevelEnum = new GraphQLObjectType({
45   name: 'PrivacyLevel',
46   values: {
47     PUBLIC: {
48       value: 'public'
49     },
50     ACQUAINTANCE: {
51       value: 'acquaintance'
52     },
53     FRIEND: {
54       value: 'friend'
55     },
56     TOP: {
57       value: 'top'
58     }
59   }
60 });
61
62 const RootMutation = new GraphQLObjectType({
63   name: 'RootMutation',
64   description: 'The root mutation',
65   fields: {
66     createPost: {
67       type: PostType,
68       args: {
69         body: {
70           type: new GraphQLNonNull(GraphQLString)
71         },
72         level: {
73           type: new GraphQLNonNull(LevelEnum),
74         }
75       },
76       resolve(source, args, context) {
77         return loaders.createPost(args.body, args.level, context).then((nodeId) \
78 => {
79           return loaders.getNodeById(nodeId);
80         });
81       }
82     }
83   }
84 })
```

```
83     }
84 });

```

First we instantiate a new kind of object, a `GraphQLEnumType`. We only briefly mentioned the `Enum` GraphQL type in previous chapter, but it works similarly to how enums work in many programming languages. Because our `level` argument should only be one of a fixed amount of options, we enforce that contract at the schema level with an enum. By convention, enums in GraphQL are ALL_CAPS.

After creating our enum, we use it in the `args` property of our new `createPost` mutation. Note that in addition to the arguments, `createPost` has a type of `PostType`, which means we eventually need to return a post object after performing our mutating code. That work is actually deferred to a new `createPost` loader, which looks like this:

```
151 export const createPost = (body, level, context) => {
152   const { dbId } = tables.splitNodeId(context);
153   const created_at = new Date().toISOString().split('T')[0];
154   const posts = [{ body, level, created_at, user_id: dbId }];
155
156   let query = tables.posts.insert(posts).toQuery();
157   return database.getSql(query).then(() => {
158     return database.getSql({ text: 'SELECT last_insert_rowid() AS id FROM posts' \
159   });
160   }).then((ids) => {
161     return tables.dbIdToNodeId(ids[0].id, tables.posts.getName());
162   });
163 };

```

This is mostly specific to our SQLite database, but you can imagine how this would work in other frameworks or data stores. We construct our database row, insert it, and then retrieve the newly inserted ID.

If you open up GraphiQL, you should be able to give this mutation a try:

```
1 mutation {
2   createPost(body:"First post!", level:PUBLIC) {
3     id
4     body
5   }
6 }
```

In the wild, you may run into more complex scenarios for updating data such as uploading files. The specifics will depend on what server language and library you use, but it's supported with

Relay and GraphQL-JS. The [Relay documentation](#)¹²⁴ discusses how files are handled, and you can find [examples elsewhere](#)¹²⁵ of how to leverage those within your GraphQL schema.

Relay and GraphQL

The “Facebook-lite” schema we’ve developed may be small, but it should give you an idea of how to structure common operations in a GraphQL server. It also happens to be compatible with [Relay](#)¹²⁶, Facebook’s frontend React library for working with a GraphQL server.

In addition to publishing Relay itself, Facebook publishes a library to help you more easily construct a Relay-compatible GraphQL server with Node. This [GraphQL-Relay-JS](#)¹²⁷ package reduces much of the boilerplate we’ve experienced, especially for some of Relay’s more powerful features.

You should read over the docs for all the details, but we’re briefly going to convert some of our code to use this library. First we need to install it via NPM:

```
$ npm install graphql-relay@0.4.1 --save --save-exact
```

GraphQL-Relay is particularly helpful with connection fields. Although we only had one proper connection field in our schema (`posts`), you can imagine that repeating the types and code for each connection in a larger app would become tiresome. Luckily, all we need is a quick import:

```
15 import {
16   connectionDefinitions
17 } from 'graphql-relay';
```

Then delete all of our existing connection types, so that we skip straight to the `UserType`:

```
34 const resolveId = (source) => {
35   return tables.dbIdToNodeId(source.id, source.__tableName);
36 };
37
38 export const UserType = new GraphQLObjectType({
39   name: 'User',
```

And at the very bottom, add this one-liner to define `PostsConnectionType`:

¹²⁴<https://facebook.github.io/relay/docs/api-reference-relay-mutation.html#getfiles>

¹²⁵<http://stackoverflow.com/a/35585482>

¹²⁶<https://facebook.github.io/relay/>

¹²⁷<https://github.com/graphql/graphql-relay-js>

```
116 const { connectionType: PostsConnectionType } = connectionDefinitions({ nodeType\\
117 : PostType });
```

Internally, this generates all the types we crafted by hand - PageInfo, PostEdge, and PostConnection. You can confirm this if you load up the GraphQL documentation:

A connection to a list of items.

FIELDS

pageInfo: PageInfo!

edges: [PostEdge]

GraphQL Relay

In addition to connections, the GraphQL-Relay library also has functions for simplifying how [node types are structured¹²⁸](#) and [working with Relay-compatible mutations¹²⁹](#). We'll explore this more in the upcoming Relay chapter, but Relay imposes some rules on how mutations work, similar to the way that certain types and arguments required for connections.

None of the GraphQL server changes required for Relay are specific to GraphQL-JS or JavaScript in general. Any language GraphQL server can be made compatible with Relay, and hopefully this chapter has made you more familiar with the basic building blocks of any GraphQL schema.

Performance: N+1 Queries

Now that our schema has settled, we can reconsider performance. Before we dive in, remember that the performance needs of different products can be incredibly disparate. Engineers should carefully consider the costs and benefits of writing code that is more performant at the risk of additional complexity.

Let's consider a query like this:

¹²⁸<https://github.com/graphql/graphql-relay-js#object-identification>

¹²⁹<https://github.com/graphql/graphql-relay-js#mutations>

```

1  {
2    node(id:"users:4") {
3      ... on User {
4        friends {
5          edges {
6            node {
7              id
8              about
9              name
10             }
11           }
12         }
13       }
14     }
15 }
```

Under the hood, our current GraphQL resolution code will trigger one database query to get the node for "users:4", one query to get the list of friend IDs, and N database queries for each of the friends edges. This is commonly referred to as the [N+1 Query Problem¹³⁰](#) and can easily occur using any web framework or ORM. You can imagine that for a slow database or a large number of edges, this will cause degraded performance. We can visualize this with the following raw SQL:

```

1 SELECT "users".* FROM "users" WHERE ("users"."id" = 4) LIMIT 1
2 SELECT "users_friends"."user_id_b" FROM "users_friends" WHERE ("users_friends"."\  
user_id_a" = 4)
3 SELECT "users".* FROM "users" WHERE ("users"."id" = 1) LIMIT 1
4 SELECT "users".* FROM "users" WHERE ("users"."id" = 3) LIMIT 1
5 SELECT "users".* FROM "users" WHERE ("users"."id" = 2) LIMIT 1
```

In a perfect world, we would only need two database queries: one to retrieve the initial node, and then one to retrieve all of the friends in one query (or within whatever paging limits we need) - in other words, we batch all of the queries for friends. Something more like this would not:

```

1 SELECT "users".* FROM "users" WHERE ("users"."id" = 4) LIMIT 1
2 SELECT "users_friends"."user_id_b" FROM "users_friends" WHERE ("users_friends"."\  
user_id_a" = 4)
3 SELECT "users".* FROM "users" WHERE ("users"."id" in (1, 3, 2)) LIMIT 3
```

Consider how loading a user works: it's a call to `loaders.getNodeById`. Currently that function immediately triggers a database query, but what if we could "wait" for some small amount of

¹³⁰https://secure.phabricator.com/book/phabcontrib/article/n_plus_one/

time, collect node IDs that need to be loaded, and then trigger a database query like the one above? GraphQL and JavaScript enable intuitive techniques for batching alike queries, which we'll implement.

Facebook maintains a library called [DataLoader¹³¹](#) to help, which is a generic JavaScript library independent of React or GraphQL. You use the library to create *loaders*, which are objects that automatically batch fetching of similar data. For example, you would instantiate a `UserLoader` to load users from the database:

```
1 const UserLoader = new DataLoader((userIds) => {
2   const query = table
3     .select(table.star())
4     .where(table.id.in(userIds))
5     .toQuery();
6
7   return database.getSql(query.toQuery());
8 });
9
10 // elsewhere, loading a single user
11 function resolveUser(userId) {
12   return UserLoader.load(userId);
13 }
```

Notice how `UserLoader.load` takes a single `userId` as an argument, but the argument to its internal function is an array of `userIds`. This means if we call `UserLoader.load` from multiple places in rapid succession, we have the option of crafting a more efficient database query.

In our app, most of our code touches `loaders.getNodeById` which makes it a strong candidate for automatic batching. None of the code that invokes `getNodeById` has to change; instead, we're going to internally batch node fetches using `DataLoader`.

First, install `DataLoader` from NPM:

```
$ npm install dataloader@1.2.0 --save --save-exact
```

Now onto our changes to `loaders.js`. We're going to make one data loader per table, which is a reasonable way to start optimizing. The code starts like this:

¹³¹<https://github.com/facebook/dataloader>

```

1 import * as database from './database';
2 import * as tables from './tables';
3
4 import DataLoader from 'dataloader';
5
6 const createNodeLoader = (table) => {
7   return new DataLoader((ids) => {
8     const query = table
9       .select(table.star())
10      .where(table.id.in(ids))
11      .toQuery();
12
13   return database.getSql(query).then((rows) => {
14     rows.forEach((row) => {
15       row.__tableName = table.getName();
16     });
17     return rows;
18   });
19 });
20 };

```

Our `createNodeLoader` is a factory function which returns a new instance of `DataLoader`. We craft a query of the form `SELECT * FROM $TABLE WHERE ID IN($IDS)`, which lets us select multiple nodes with a single query.

Now we need to invoke our factory function, which we'll store in a constant:

```

22 const nodeLoaders = {
23   users: createNodeLoader(tables.users),
24   posts: createNodeLoader(tables.posts),
25   usersFriends: createNodeLoader(tables.usersFriends),
26 };

```

Finally, we change our definition of `getNodeById` to use the appropriate loader:

```

28 export const getNodeById = (nodeId) => {
29   const { tableName, dbId } = tables.splitNodeId(nodeId);
30   return nodeLoaders[tableName].load(dbId);
31 };

```

If you open up GraphiQL and try this query, you'll notice the SQL logs in the server console are appropriately batching our database fetches:

```

1  {
2    user3: node(id:"users:3") {
3      id
4    }
5    user4: node(id:"users:4") {
6      id
7    }
8  }

```

```

$ node index.js
{ starting: true }
{ running: true }
SELECT "users".* FROM "users" WHERE ("users"."id" IN ($1, $2))
[ '3', '4' ]

```

Note that our higher-level GraphQL schema code is totally unaware of this optimization and didn't have to change at all. In general, you should prefer keeping optimizations at the loader and data service level so all consumers can enjoy the benefits.

DataLoader is simple but powerful tool. Although we showed off its batching abilities, it can also act as a cache - if you'd like to learn more, check out its [documentation¹³²](#) and consider watching [this talk by its maintainer¹³³](#).

Summary

We covered a lot of ground in this chapter. We designed a schema, created a GraphQL server from scratch, connected it to a relational database, and explored some performance optimizations. Regardless of your production language and stack, these concepts will apply to all GraphQL server implementations. Additionally, this should give you some more understanding and context whenever you connect to a GraphQL server from the frontend.

We used the Facebook-maintained GraphQL-JS library in this chapter, but the GraphQL ecosystem is exploding. Here some more popular options and technologies you might want to explore:

- [GraphQL¹³⁴](#) for Ruby
- [Graphene¹³⁵](#) for Python
- [Sangria¹³⁶](#) for Scala

¹³²<https://github.com/facebook/dataloader#getting-started>

¹³³<https://www.youtube.com/watch?v=OQTnXNCDywA>

¹³⁴<https://github.com/rmosolgo/graphql-ruby>

¹³⁵<https://github.com/graphql-python/graphene>

¹³⁶<https://github.com/sangria-graphql/sangria>

- Apollo Server¹³⁷ for Node
- Graffiti-Mongoose¹³⁸ for Node and MongoDB
- Services like Reindex¹³⁹ and Graphcool¹⁴⁰ for hosting GraphQL servers

Now that we've learned about consuming GraphQL and authoring a GraphQL server, it's time to put everything we've covered thus far and learn about Relay.

¹³⁷<http://docs.apollostack.com/apollo-server/tools.html>

¹³⁸<https://github.com/RisingStack/graffiti-mongoose>

¹³⁹<https://www.reindex.io/>

¹⁴⁰<https://graph.cool/>

Appendix A: PropTypes

PropTypes are a way to validate the values that are passed in through our props. Well-defined interfaces provide us with a layer of safety at the run time of our apps. They also provide a layer of documentation to the consumer of our components.

We define PropTypes by passing them as an option to `createClass()`:

```
1 const Component = React.createClass({
2   propTypes: {
3     // propType definitions go here
4   },
5   render: function() {}
6 });
```



Classical style PropTypes

Defining `propTypes` in a class-based component using ES6 syntax is slightly different as it needs to be defined as a `class` method on the component. For example:

```
1 class Component extends React.Component {
2   render() {}
3 }
4 Component.propTypes = { /* definition goes here */};
```

The key of the `propTypes` object defines the name of the prop we are validating, while the value is the types defined by the `React.PropTypes` object (which we discuss below) or a custom one through a custom function.

The `React.PropTypes` object exports a lot of validators that cover the most of the cases we'll encounter. For the not-so common cases, React allows us to define our own PropType validators as well.

Validators

The `React.PropTypes` object contains a list of common validators (but we can also define our own. More on that later).

When a prop is passed in with an invalid type or fails the prop type, a warning is passed into the JavaScript console. These warnings will *only* be shown in development mode, so if we accidentally deploy our app into production with an improper use of a component, our users won't see the warning.

The built in validators are:

- `string`
- `number`
- `boolean`
- `function`
- `object`
- `shape`
- `oneOf`
- `instanceOf`
- `array`
- `arrayOf`
- `node`
- `element`
- `any`
- `required`

string

To define a prop as a string, we can use `React.PropTypes.string`.

```
1 const Component = React.createClass({  
2   propTypes: {  
3     name: React.PropTypes.string  
4   },  
5   // ...  
6 })
```

To pass a string as a prop we can either just pass the string itself as an attribute or use the {} braces to define a string variable. These are all *functionally* equivalent:

```
1 <Component name={"Ari"} />  
2 <Component name="Ari" />
```

number

To specify a prop should be a number, we can use `React.PropTypes.number`.

```

1 const Component = React.createClass({
2   propTypes: {
3     totalCount: React.PropTypes.number
4   },
5   // ...
6 })

```

Passing in a number, we must pass it as a JavaScript value or the value of a variable using braces:

```

1 var x = 20;
2
3 <Component totalCount={20} />
4 <Component totalCount={x} />

```

boolean

To specify a prop should be a boolean (true or false), we can use `React.PropTypes.bool`.

```

1 const Component = React.createClass({
2   propTypes: {
3     on: React.PropTypes.bool
4   },
5   // ...
6 })

```

To use a boolean in a JSX expression, we can pass it as a JavaScript value.

```

1 var isOn = true;
2
3 <Component isOn={true} />
4 <Component isOn={false} />
5 <Component on={isOn} />

```



A trick for using booleans to show or hide content is to use the `&&` expression.

For instance, inside our `Component.render()` function, if we want to show content only when the `on` propotype is true, we can do so like: `javascript render: function() { return (<div> {this.props.on && <p>This component is on</p>} </div>) }`

function

We can pass a function as a prop as well. To define a prop to be a function, we can use `React.PropTypes.func`. Often times when we write a Form component, we'll pass a function as a prop to be called when the form is submitted (i.e. `onSubmit()`). It's common to define a prop as a required function on a component.

```
1 const Component = React.createClass({  
2   propTypes: {  
3     onComplete: React.PropTypes.func  
4   },  
5   // ...  
6 })
```

We can pass a function as a prop by using the JavaScript expression syntax, like so:

```
1 const x = function(name) {};  
2 const fn = value => alert("Value: " + value);  
3  
4 <Component onComplete={x} />  
5 <Component onComplete={fn} />
```

object

We can require a prop should be a JavaScript object through the `React.PropTypes.object`:

```
1 const Component = React.createClass({  
2   propTypes: {  
3     user: React.PropTypes.object  
4   },  
5   // ...  
6 })
```

Sending an object through as a prop, we'll need to use the JavaScript expression `{}` syntax:

```

1 const user = {
2   name: 'Ari'
3 }
4
5 <Component user={user} />
6 <Component user={{name: 'Anthony'}} />

```

object shape

React allows us to define the shape of an object we expect to receive by using `React.PropTypes.shape()`. The `React.PropTypes.shape()` function accepts an object with a list of key-value pairs that dictate the keys an object is expected to have as well as the value type:

```

1 const Component = React.createClass({
2   propTypes: {
3     user: React.PropTypes.shape({
4       name: React.PropTypes.string,
5       friends: React.PropTypes.arrayOf(React.PropTypes.object),
6       age: React.PropTypes.number
7     })
8   },
9   // ...
10 })

```

multiple types

Sometimes we don't know in advance what kind a particular prop will be, but we can accept one or other type. React gives us the `propTypes` of `oneOf()` and `oneOfType()` for these situations.

Using `oneOf()` requires that the `propType` be a discrete value of values, for instance to require a component to specify a log level value:

```

1 const Component = React.createClass({
2   propTypes: {
3     level: React.PropTypes.oneOf(['debug', 'info', 'warning', 'error'])
4   },
5   // ...
6 })

```

Using `oneOfType()` says that a prop can be one of any number of types. For instance, a phone number may either be passed to a component as a string or an integer:

```
1 const Component = React.createClass({  
2   propTypes: {  
3     phoneNumber: React.PropTypes.oneOfType([  
4       React.PropTypes.number,  
5       React.PropTypes.string  
6     ])  
7   },  
8   // ...  
9 })
```

instanceOf

We can dictate that a component *must* be an instance of a JavaScript class using `React.PropTypes.instanceOf()` as the value of the propType:

```
1 const Component = React.createClass({  
2   propTypes: {  
3     user: React.PropTypes.instanceOf(User)  
4   },  
5   // ...  
6 })
```

We'll use the JavaScript expression syntax to pass in a particular prop.

```
1 const User = function(name) {  
2   this.name = name;  
3   return this;  
4 };  
5  
6 const ari = new User('Ari');  
7  
8 <Component user={ari} />
```

array

On occasion we'll want to send in an array as a prop. To set an array, we'll use the `React.PropTypes.array` as the value.

```
1 const Component = React.createClass({  
2   propTypes: {  
3     authors: React.PropTypes.array  
4   },  
5   // ...  
6 })
```

Sending an object through as a prop, we'll need to use the JavaScript expression {} syntax:

```
1 const users = [  
2   {name: 'Ari'}  
3   {name: 'Anthony'}  
4 ];  
5  
6  
7 <Component authors={{name: 'Anthony'}} />  
8 <Component authors={users} />
```

array of type

React allows us to dictate the type of values each member of an array should be using `React.PropTypes.arrayOf()`.

```
1 const Component = React.createClass({  
2   propTypes: {  
3     authors: React.PropTypes.arrayOf(React.PropTypes.object)  
4   },  
5   // ...  
6 })
```

We'll use the JavaScript expression syntax {} to pass in an array:

```
1 const users = [
2   {name: 'Ari'}
3   {name: 'Anthony'}
4 ];
5
6
7 <Component authors={[{name: 'Anthony'}]} />
8 <Component authors={users} />
```

node

We can also pass anything that can be rendered, such as numbers, string, DOM elements, arrays, or fragments that contain them using the `React.PropTypes.node`.

```
1 const Component = React.createClass({
2   propTypes: {
3     icon: React.PropTypes.node
4   },
5   // ...
6 })
```

Passing a *node* as a prop is straightforward as well. Passing a node as a value is often useful when requiring a component to have children or setting a custom element. For instance, if we want to allow our user to pass in either the name of an icon or a custom component, we can use the `node` `propType`.

```
1 const icon = <FontAwesomeIcon name="user" />
2
3
4 <Component icon={icon} />
5 <Component icon={"fa fa-cog"} />
```

element

React's flexibility allows us to pass another React element in as a prop as well by using the `React.PropTypes.element`:

```
1 const Component = React.createClass({
2   propTypes: {
3     customHeader: React.PropTypes.element
4   },
5   // ...
6 })
```

We can build our components so that the interface they allow our users to specify a custom component. For instance, we might have a `<List />` component who's responsibility is to output a list of elements. Without custom components, we would have to build a separate `<List />` React component for each type of list we want to render (which might be appropriate, depending on the behavior of the element). By passing a component type, we can reuse the `<List />` component.

For instance, a list component might look like:

```
1 const List = React.createClass({
2   propTypes: {
3     listComponent: PropTypes.element,
4     list: PropTypes.array
5   },
6   renderListItem: function(item, i) {
7     const Component = this.props.listComponent || "li";
8     return React.createElement(Component, this.props, item)
9   },
10  render: function() {
11    const list = this.props.list;
12    return (
13      <ul>
14        {list.map(this.renderListItem)}
15      </ul>
16    )
17  }
18});
```

We can use this list component with or without specifying a custom component:

```

1 const Item = React.createClass({
2   render: function() {
3     return (
4       <div>{this.props.children}</div>
5     )
6   }
7 })
8
9 <List list={[1, 2, 3]} />
10 <List list={[1, 2, 3]} listComponent={Item} />
```

any type

React also allows us to specify that a prop must be present, regardless of its type. We can do this by using the `React.PropTypes.any` validator.

```

1 const Component = React.createClass({
2   propTypes: {
3     mustBePresent: React.PropTypes.any
4   },
5   // ...
6 })
```

Optional & required props

All props are considered optional unless otherwise specified. To require a prop be passed to a component and validated, we can append every propType validation with `.isRequired`.

For instance, if we must have a function to get called after some action when a `Loading` component has completed, we can specify it like this:

```

1 const Loading = React.createClass({
2   propTypes: {
3     // Optional props:
4     onStart: React.PropTypes.func,
5     // Required props:
6     onComplete: React.PropTypes.func.isRequired,
7     name: React.PropTypes.string.isRequired
8   },
9   // ...
10 })
```

custom validator

React allows us to specify a custom validation function for all of the other situations where the default validation functions don't cover it. In order to run a write a custom validation we'll specify a function that accepts 3 arguments:

1. The props passed to the component
2. The propName being validated
3. The componentName we're validating against

If our validation passes, we can run through the function and return anything. The validation function will only fail if an Error object is raised (i.e. `new Error()`).

For instance, if we have a loader that accepts validated users, we can run a custom function against the prop.

```
1 const Component = React.createClass({
2   propTypes: {
3     user: function(props, propName, componentName) {
4       const user = props[propName];
5       if (!user.isValid()) {
6         return new Error('Invalid user');
7       }
8     }
9   },
10  // ...
11 })
```

Appendix B: Tools

It takes an ecosystem to build a web app. This appendix describes a few tools that we use when developing our React apps.

Curl

cURL¹⁴¹ is a standard, popular tool for performing HTTP requests on the commandline. It comes pre-installed on many systems. Being able to use cURL effectively is an important skill to have as a web developer. Curl is often the first tool many people turn to for debugging or automating HTTP requests.



In order to use curl you will need some understanding of the HTTP protocol. If you're unfamiliar with HTTP, checkout [A Beginner's Guide to HTTP and REST¹⁴²](#) and [The HttpWatch Guide to HTTP¹⁴³](#)

A GET Request

If you have curl installed on your system, you can make a basic request like this:

```
1 curl http://google.com
```

A POST Request

You can also use cURL to mimic submitting a form by using a POST request.

Here is how you can submit a POST request to [GraphQL hub¹⁴⁴](#):

```
1 curl -H 'Content-Type:application/graphql' -XPOST https://www.graphqlhub.com\ 
2 /graphql?pretty=true -d '{ hn { topStories(limit: 2) { title url } } }'
```

Notice that there are three options here, passed as “flags”:

¹⁴¹<https://curl.haxx.se/>

¹⁴²<http://code.tutsplus.com/tutorials/a-beginners-guide-to-http-and-rest--net-16340>

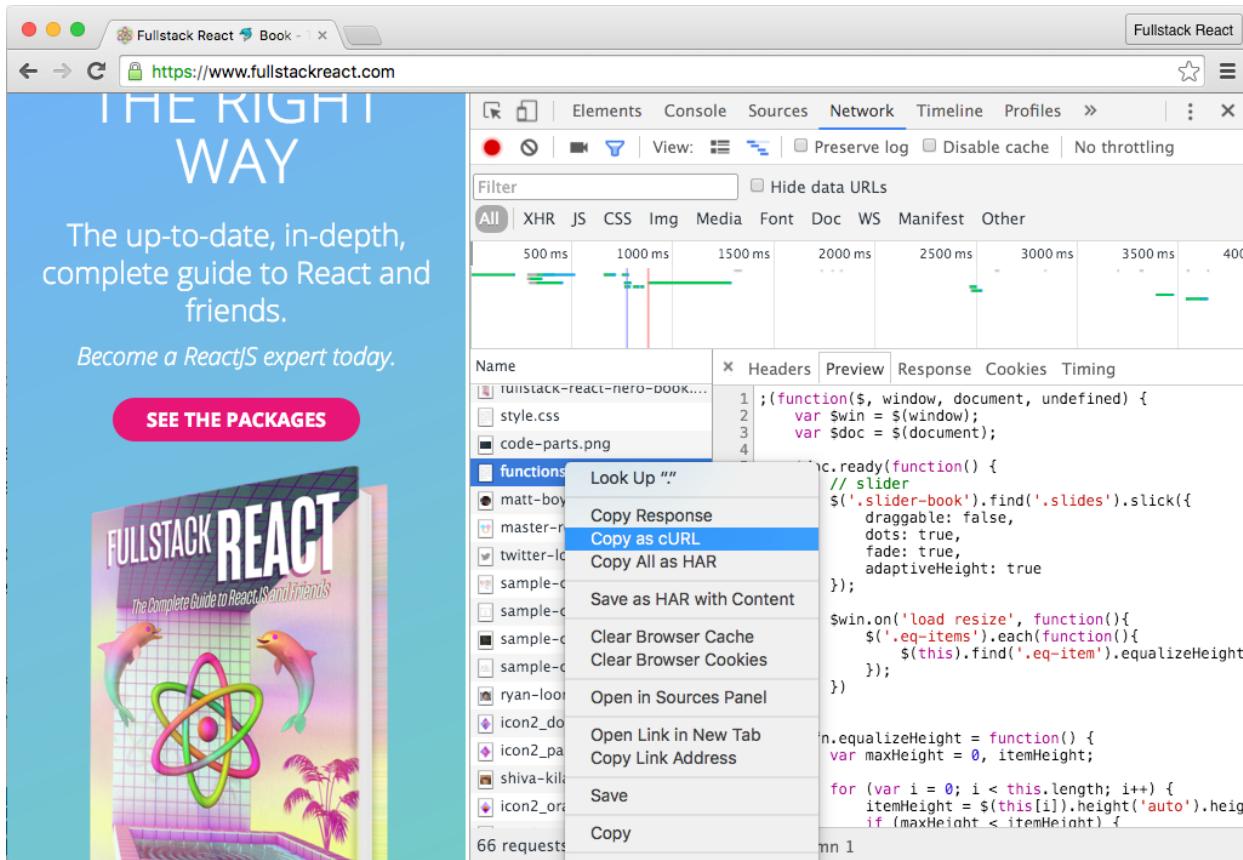
¹⁴³<https://www.httpwatch.com/httpgallery/introduction/>

¹⁴⁴<https://www.graphqlhub.com>

- -H - states that we want to pass a *header*. In this case, we're passing the HTTP header Content-Type as application/graphql
- -XPOST - states that we want to send a POST request
- -d - states that the string that follows is the POST body. In this case, it is a [GraphQL query](#) that fetches the top 2 posts from Hacker News.

Chrome “Copy as cURL”

Chrome has a powerful feature that lets you copy any HTTP operation a cURL request. To do this:



Copy as cURL in Chrome

1. Open up the Chrome Developer Tools
2. Click on the “Network” Tab
3. Navigate with your browser / perform the operation you want to copy
4. Locate your HTTP request in the list of requests
5. Right click the request and pick “Copy as cURL”
6. Paste your request into your shell

When you paste the command, you'll notice that it has a lot of "noise". The browser sends quite a lot of headers! Not all of them are always necessary for your request to succeed, but it's a good starting point. This can be especially helpful when you have authentication cookies stored in your browser and you need to debug a protected request.

```
1 curl 'https://www.fullstackreact.com/assets/vendor/functions.js' -H 'if-none\\
2 -match: W/"5dd95509e78d11579fc427e9f41889d6"' -H 'accept-encoding: gzip, deflate\\
3 , sdch' -H 'accept-language: en-US,en;q=0.8' -H 'user-agent: Mozilla/5.0 (Macintosh;\\
4 Intel Mac OS X 10_11_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/49.0.2623.75\\
5 Safari/537.36' -H 'accept: */*' -H 'cache-control: max-age=0' -H 'authority:\\
6 www.fullstackreact.com' -H 'cookie: __cfduid=d309889bd8e027d878c2e16e0176e7\\
7 64d1463427161; _ga=GA1.2.135748265.1463427163; _gat=1' -H 'if-modified-since: Tu\\
8 e, 15 Mar 2016 03:37:56 GMT' -H 'referer: https://www.fullstackreact.com/' --compressed
```

Though not technically identical, the essential, trimmed down version of this request could be:

```
1 curl 'https://www.fullstackreact.com/assets/vendor/functions.js'
```

More Resources

If you learn more about cURL, checkout:

- The official curl tutorial¹⁴⁵
- This Github API curl tutorial¹⁴⁶

¹⁴⁵<https://curl.haxx.se/docs/httpscripting.html>

¹⁴⁶<https://gist.github.com/joyrexus/85bf6b02979d8a7b0308>

Changelog

Revision 14 - 2016-08-26

- Added “Forms” Chapter!

Revision 13 - 2016-08-02

- Updated code to react-15.3.0

Revision 12 - 2016-07-26

- Updated code to react-15.2.1

Revision 11 - 2016-07-08

- Updated code to react-15.2.0

Revision 10 - 2016-06-24

- Updated code to react-15.1.0

Revision 9 - 2016-06-21

- Added “Writing a GraphQL Server” Chapter

Revision 8 - 2016-06-02

- Added “Intermediate Redux” Chapter
- Added “Using Presentational and Container Components with Redux” Chapter

Revision 7 - 2016-05-13

- Added “Using GraphQL” chapter
- Updated all code to react-15.0.2

Revision 6 - 2016-05-13

- Added “Configuring Components” chapter
- Added “PropTypes” appendix

Revision 5 - 2016-04-25

- Added “Intro to Flux and Redux” chapter

Revision 4 - 2016-04-22

- Added “JSX and the Virtual DOM” chapter

Revision 3 - 2016-04-08

- Upgraded all code to react-15.0.1

Revision 2 - 2016-03-16

- Significant rewrites to Ch 1. “First React Application” to make it clearer / better flow of thought
- Various bugfixes
- Improvements to code style
- Updated diagrams

Revision 1 - 2016-02-14

Initial version of the book. Contains:

- Ch 1. Your first React Web Application
- Ch 2. Components
- Ch 3. Components & Servers