

O'REILLY®



React Up & Running

BUILDING WEB APPLICATIONS

Stoyan Stefanov

ReactJS Up and Running

Stoyan Stefanov

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'REILLY®

www.it-ebooks.info

ReactJS Up and Running

by Stoyan Stefanov

Copyright © 2010 Stoyan Stefanov. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://safaribooksonline.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editor: Meg Foley

Production Editor: FIX ME!

Copyeditor: FIX ME!

Proofreader: FIX ME!

Indexer: FIX ME!

Cover Designer: Karen Montgomery

Interior Designer: David Futato

Illustrator: Rebecca Demarest

December 2015: First Edition

Revision History for the First Edition:

2015-06-29: Early Release Revision 1

See <http://oreilly.com/catalog/errata.csp?isbn=9781491931820> for release details.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. !!FILL THIS IN!! and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-1-491-93182-0

[?]

Table of Contents

1. Hello world.....	1
Setup	1
Hello React world	2
What just happened?	4
React.DOM.*	5
Special DOM attributes	8
Next: custom components	9
 2. The life of a component.....	 11
Bare minimum	11
Properties	13
propTypes	14
State	18
A stateful textarea component	19
A note on DOM events	23
Event handling in the olden days	23
Event handing in React	25
Props vs State	25
Props in initial state: an anti-pattern	26
Accessing the component from the outside	27
Changing properties mid-flight	29
Lifecycle methods	30
Lifecycle example: log it all	30
Lifecycle example: use a mixin	33
Lifecycle example: with a child component	35
Performance win: prevent component updates	37
PureRenderMixin	40

3. Excel: a fancy table component.....	43
Data first	43
Table headers loop	44
Debugging the console warning	46
Adding <td> content	47
Sorting	49
Sorting UI cues	51
Editing data	53
Editable cell	54
Input field cell	56
Saving	56
Conclusion and virtual DOM diffs	57
Search	58
State and UI	60
Filtering content	62
How can you improve the search?	64
Instant replay	64
How can you improve the replay?	66
Download the table data	66
 4. JSX.....	 69
Hello JSX	69
Transpiling JSX	70
Client-side	70
Working with external files	73
Build process	73
Babel	75
About the JSX transformation	75
JavaScript in JSX	77
Whitespace in JSX	78
Comments in JSX	80
HTML entities	80
Anti-XSS	81
Spread attributes	82
Parent-to-child spread attributes	82
Returning multiple nodes in JSX	84
JSX vs HTML differences	85
No class, what for?	86
style is an object	86
Closing tags	86
camelCase attributes	87
JSX and forms	87

onChange handler	87
value vs defaultValue	87
<textarea> value	88
<select> value	89

Let's get started on the journey to mastering application development using React. In this chapter you will learn how to setup React and write your first "Hello world" application.

Setup

First things first: you need to get a copy of the React library. Luckily this is as simple as it can be. Go to <http://reactjs.com> (which should redirect you to the official GitHub home at <http://facebook.github.io/react/>), then click the "Download" button, then "Download Starter Kit" and you'll get a copy of a zip file. Unzip and copy the directory contained in the download to a location where you'll be able to find it.

For example

```
mkdir ~/reactbook  
mv ~/Downloads/react-0.13.3/ ~/reactbook/react
```

Now your working directory (reactbook) should look like [Figure 1-1](#).

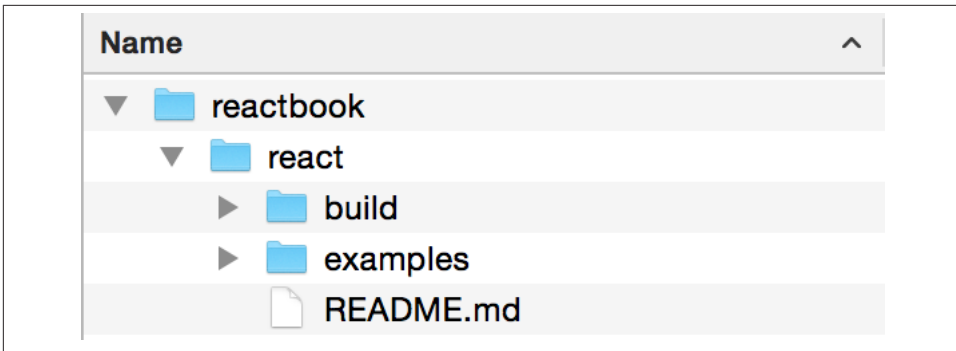


Figure 1-1. Your React directory listing

The only file you need to get started for the time being is `~/reactbook/react/build/react.js`. You'll learn about all the others as you go along.



At the time of writing 0.13.3 is the latest stable version.

Note that React doesn't impose any directory structure, you're free to move to a different directory or rename `react.js` however you see fit.

Hello React world

Let's start with a simple page in your working directory (`~/reactbook/hello.html`).

```
<!DOCTYPE html>
<html>
  <head>
    <title>hello React</title>
    <meta charset="utf-8">
  </head>
  <body>
    <div id="app">
      <!-- my app renders here -->
    </div>
    <script src="react/build/react.js"></script>
    <script>
      // my app's code
    </script>
  </body>
</html>
```

There are only two notable things happening in this file:

- you include the React library (`<script src="react/build/react.js">`)
- you define where your application should be placed on the page (`<div id="app">`)



You can always mix regular HTML content as well as other JavaScript libraries with a React app. You can also have several React apps on the same page. All you need is place in the DOM where you tell React: “do you magic right here”.

Now let’s add the code that says hello. Update `hello.html` and replace `// my app's` code with:

```
React.render(  
  React.DOM.h1(null, "Hello world!"),  
  document.getElementById("app")  
);
```

Load `hello.html` in your browser and you’ll see your new app in action (**Figure 1-2**)

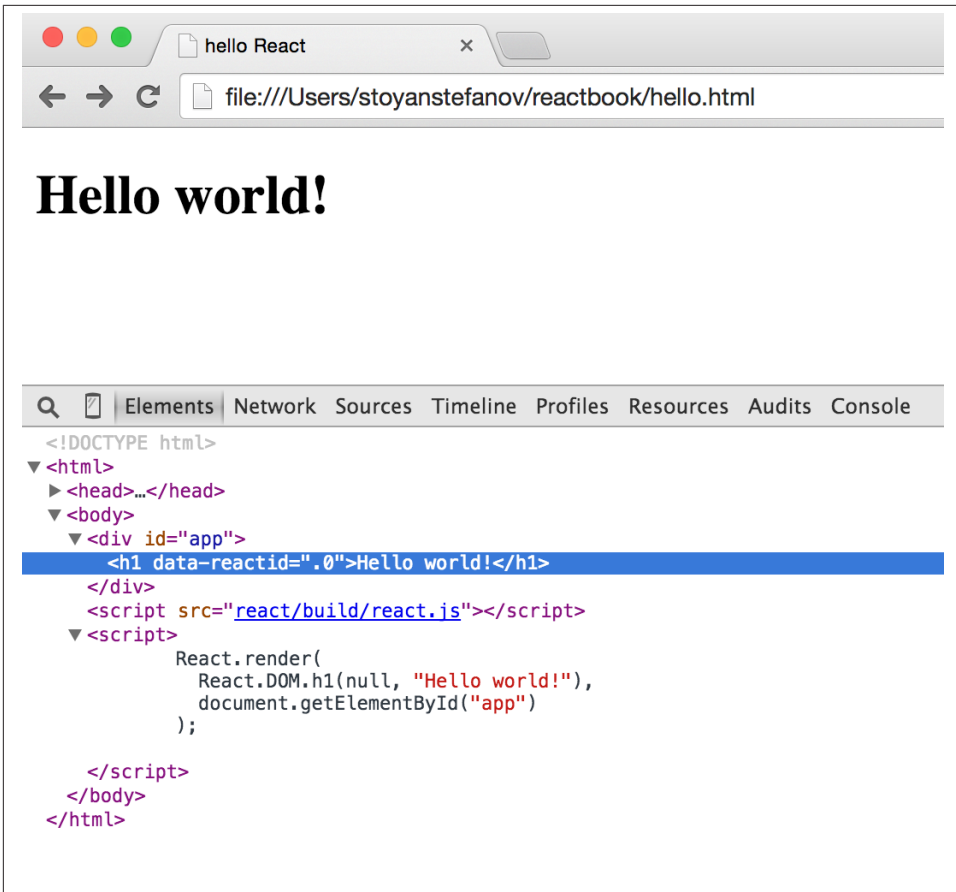


Figure 1-2. Hello World in action

Congratulations, you've just built your first React application!

Figure 1-2 also shows the *generated* code in Chrome Developer Tools where you can see that the contents of `<div id="app">` placeholder was replaced with the contents generated by your React app.

What just happened?

There are few things of interest in the code that made your first app work.

First, you see the global `React` object. All of the APIs available to you are accessible via this object. And not to worry - the API is intentionally minimal, so there are not a lot of method names to remember.

Next, there is the concept of *components*. You build your UI using components and you combine these components in any way you see fit. In your applications you'll end up creating your own custom components, but to get you off the ground, React provides wrappers around HTML DOM elements. You use the wrappers via the `React.DOM` object. In this first example, you can see the use of the `h1` component. It corresponds to the `<h1>` HTML element and is available to you using a call to `React.DOM.h1()`.

Finally, you see good old school `document.getElementById("app")` DOM access. You use this to tell React where the application should be located in the page. This is the bridge crossing over from the DOM manipulation as you know it to the React-land.



Once you cross the bridge from DOM to React, you don't have to worry about DOM manipulation anymore, because React does the translation from components to the underlying platform (browser DOM, canvas, native app). You *don't have to* worry about DOM but that doesn't mean you cannot. React gives you "escape latches" if you want to go back to DOM-land for any reason you may need.

Now that you know what each line does, let's take a look at the big picture. What happened is this: you rendered one React component in a DOM location of your choice. You always render one top-level component and it can have as many children (and grandchildren, and so on) components as you need. In fact even in this simple example, the `h1` component has a child - the "Hello world!" text.

React.DOM.*

AS you know now, you can use a number of HTML elements as React components via `React.DOM` object (Figure 1-3 shows you how to get a full list using your browser console). Let's take a close look at this API.

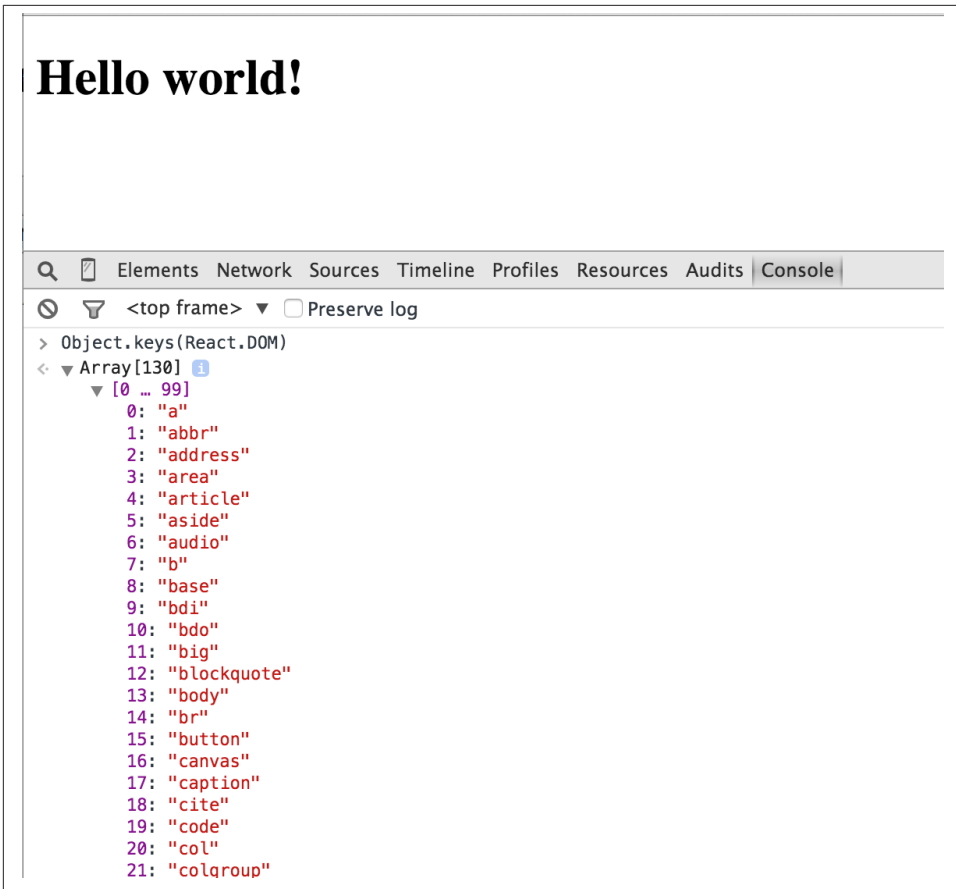


Figure 1-3. List of *React.DOM* properties

Remember the “hello world” app looked like this:

```
React.DOM.h1(null, "Hello world!");
```

The first parameter (`null` in this case) is an object that specifies any properties (think DOM attributes) that you want to pass to your component. For example you can do:

```
React.DOM.h1(  
  {  
    id: "my-heading"  
  },  
  "Hello world!"  
);
```

The HTML generated by this example is shown on [Figure 1-4](#).

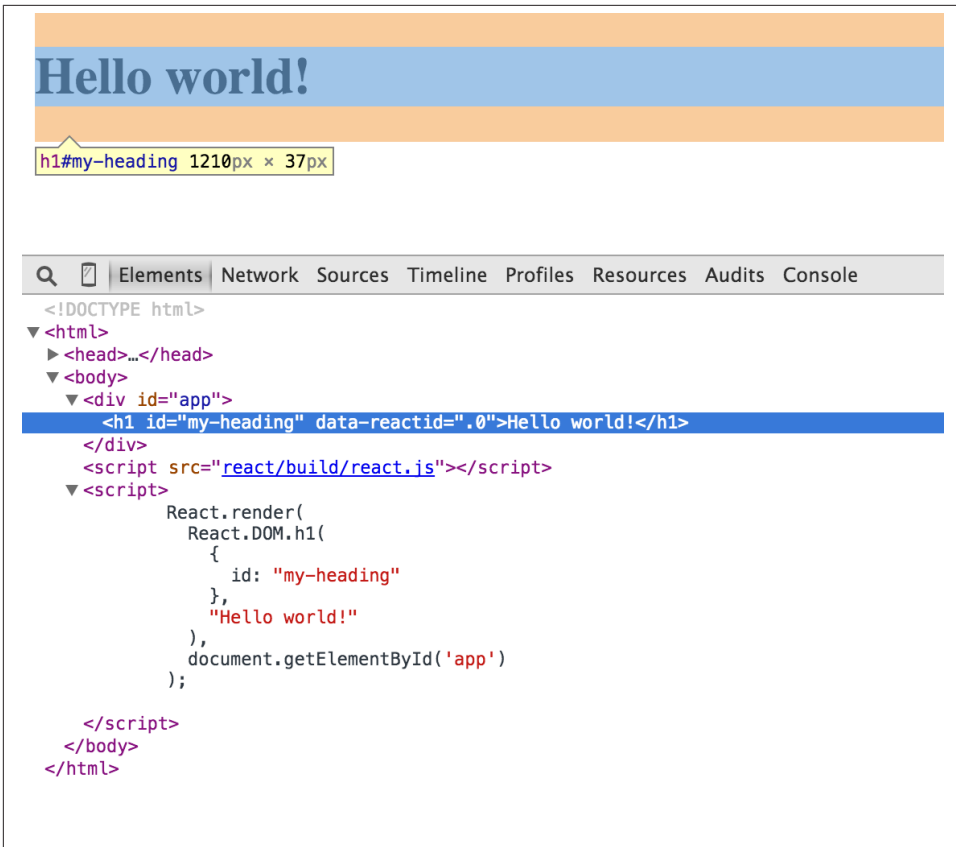


Figure 1-4. HTML generated by a `React.DOM` call

The second parameter ("Hello world!" in this example) defines a child of the component. The simplest case is just a text child (a Text node in DOM-speak) as you see above. But you can have as many nested children as you like and you pass them as additional parameters. For example:

```
React.DOM.h1(
  {id: "my-heading"},
  React.DOM.span(null, "Hello"),
  " world!"
);
```

Another example, this time with nested components:

```
React.DOM.h1(
  {id: "my-heading"},
  React.DOM.span(null,
    React.DOM.em(null, "Hell"),
    "o"
  )
);
```

```

    ),
    " world!"
  );

```



As you can see when you start nesting components, you quickly end up with a lot of function calls and parentheses to keep track of. To make things easier you can use the *JSX syntax*. JSX is a topic of a separate discussion (Chapter 4) and for the time being let's suffer through the pure JavaScript syntax. The reason is that JSX is a little controversial: people often find it repulsive at first sight (ugh, XML!) but indispensable after. Just to give you a taste, here's the previous snippet using JSX syntax:

```

React.render(
  <h1 id="my-heading">
    <span><em>Hell</em></span> world!
  </h1>,
  document.getElementById("app")
);

```

Special DOM attributes

A few special DOM attributes you should be aware of are: `class`, `for` and `style`.

You cannot use `class` and `for` because these are reserved words in JavaScript. Instead you need `className` and `htmlFor`.

```

// COUNTEREXAMPLE
// this doesn't work
React.DOM.h1(
  {
    class: "pretty",
    for: "me"
  },
  "Hello world!"
);

```

```

// PROPER EXAMPLE
// this works
React.DOM.h1(
  {
    className: "pretty",
    htmlFor: "me"
  },
  "Hello world!"
);

```

When it comes to the `style` attribute, you cannot use a string as you normally do in HTML, but you need a JavaScript object instead. Avoiding strings is always a good idea to reduce the risks of XSS (Cross-site scripting) attacks.

```

// COUNTEREXAMPLE
// this doesn't work
React.DOM.h1(
  {
    style: "background: black; color: white; font-family: Verdana"
  },
  "Hello world!"
);

// PROPER EXAMPLE
// this works
React.DOM.h1(
  {
    style: {
      background: "black",
      color: "white",
      fontFamily: "Verdana"
    }
  },
  "Hello world!"
);

```

Also notice that you need to use the JavaScript API names when dealing with CSS properties, in other words use `fontFamily` as opposed to `font-family`.

Next: custom components

And this wraps the barebone “hello world” app. Now you know how to:

- Install, setup and use the React library (it’s really just a question of `<script src="react/build/react.js">`)
- Render a React component in a DOM location of your choice (`React.render(reactWhat, domWhere)`)
- Use built-in components which are wrappers over regular DOM elements (e.g. `React.DOM.div(attributes, children)`)

The real power of React though comes from the use of custom components to build (and update!) the UI of your app. Let’s how to do just that in the next chapter.

The life of a component

Now that you know how to use the ready-made DOM components, it's time to learn how to make some of your own.

Bare minimum

The API to create a new component looks like this:

```
var MyComponent = React.createClass({  
  /* specs */  
});
```

The “specs” is a JavaScript object that has one required method `render()` and a number of optional methods and properties. A bare-bone example could look something like this:

```
var Component = React.createClass({  
  render: function() {  
    return React.DOM.span(null, "I'm so custom");  
  }  
});
```

As you can see, the only thing you *must* do is implement the `render()` method. This method must return a React component, that's why you see the `span` in the snippet above.

Using your component in an application is similar to using the DOM components:

```
React.render(  
  React.createElement(Component),  
  document.getElementById("app")  
);
```

The result of rendering your custom component is shown on [Figure 2-1](#).

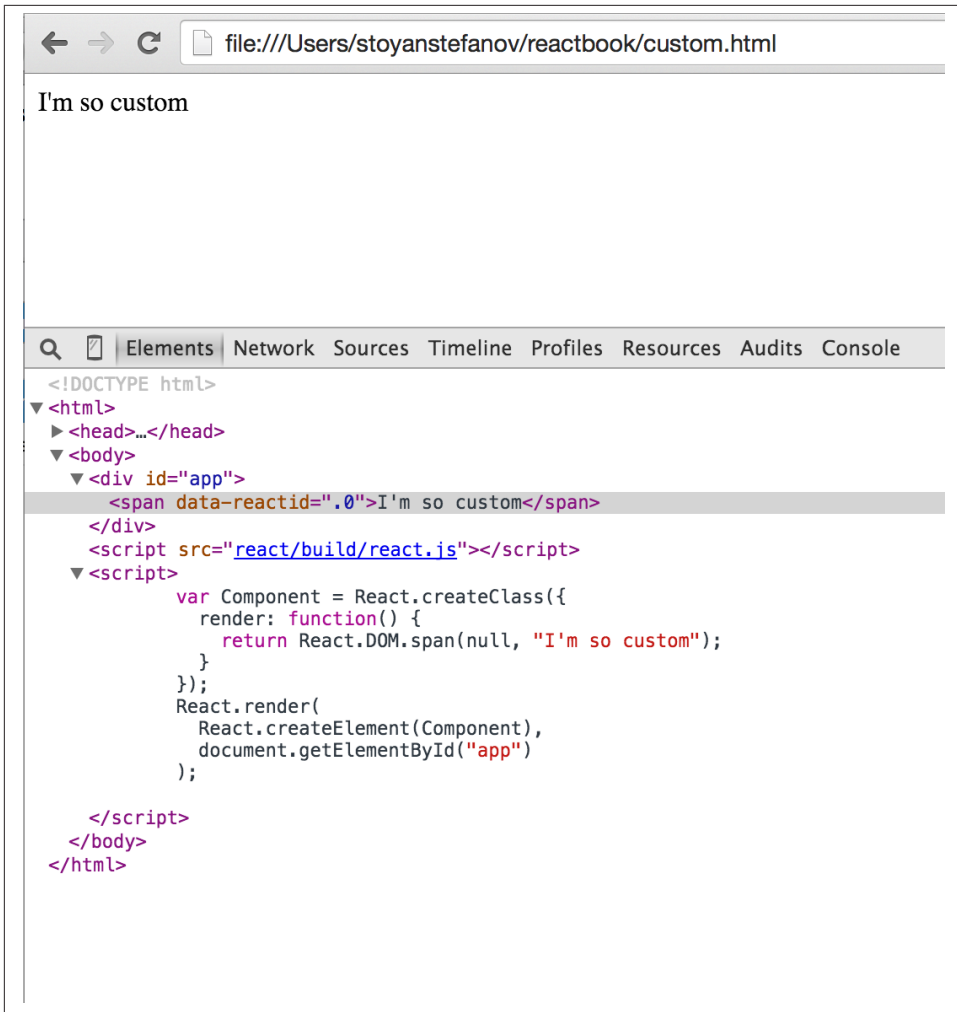


Figure 2-1. Your first custom component

The `React.createElement()` is one way to create an “instance” of your component. Another way, if you’ll be creating several instances, is to create a factory:

```
var ComponentFactory = React.createFactory(Component);

React.render(
  ComponentFactory(),
  document.getElementById("app")
);
```

Note that the `React.DOM.*` methods you already know of are actually just convenience wrappers around `React.createElement()`. In other words, this code also works with DOM components:

```
React.render(  
  React.createElement("span", null, "Hello"),  
  document.getElementById("app")  
);
```

As you can see, the DOM elements are defined as strings as opposed to JavaScript functions as is the case with custom components.

Properties

Your components can take properties and render or behave differently, depending on the values of the properties. All properties are available via `this.props` object. Let's see an example.

```
var Component = React.createClass({  
  render: function() {  
    return React.DOM.span(null, "My name is " + this.props.name);  
  }  
});
```

Passing the property when rendering the component looks like:

```
React.render(  
  React.createElement(Component, {  
    name: "Bob"  
  }),  
  document.getElementById("app")  
);
```

The result is shown on [Figure 2-2](#).

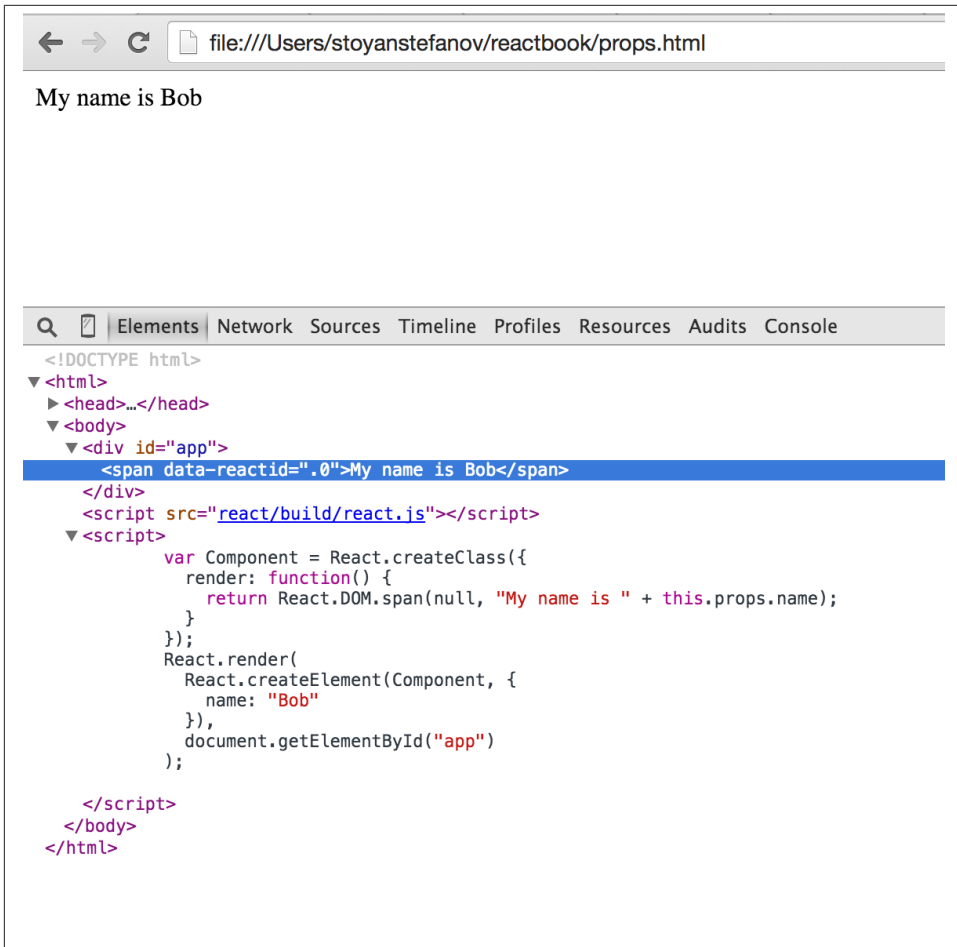


Figure 2-2. Using component properties



Think of `this.props` as read-only. Properties are useful to carry on configuration from parent components to children, so if you break this convention, it will make the application state hard to debug. Just use additional variables, or properties of your component if you feel tempted to set a property of `this.props`.

propTypes

In your components you can add a property called `propTypes` to declare the list of properties that your component accepts and their types. Here's an example:

```

var Component = React.createClass({
  propTypes: {
    name: React.PropTypes.string.isRequired
  },
  render: function() {
    return React.DOM.span(null, "My name is " + this.props.name);
  }
});

```

Using propTypes is optional but it's beneficial in two ways:

- You declare upfront what properties your component expects. Users of your component don't need to look around the (potentially long) source code of the `render()` function to tell which properties they can use to configure the component.
- React does validation of the property values at runtime, so you can write your `render()` function without being defensive (or even paranoid) about the data your components are receiving.

Let's see the validation in action. It's pretty clear that `name: React.PropTypes.string.isRequired` asks for non-optional string value of the `name` property. If you forget to pass the value, you get a warning in the console (Figure 2-3)

```

React.render(
  React.createElement(Component, {
    // name: "Bob"
  }),
  document.getElementById("app")
);

```

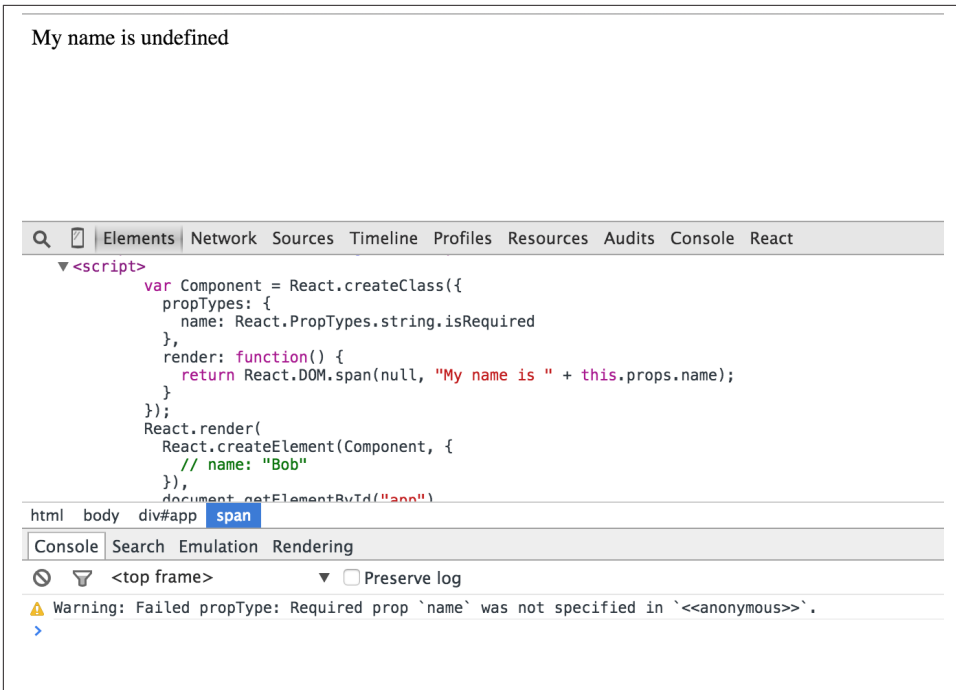


Figure 2-3. Warning when failing to provide a required property

Same if you provide a value of invalid type, say an integer (Figure 2-4).

```
React.createElement(Component, {
  name: 123
})
```

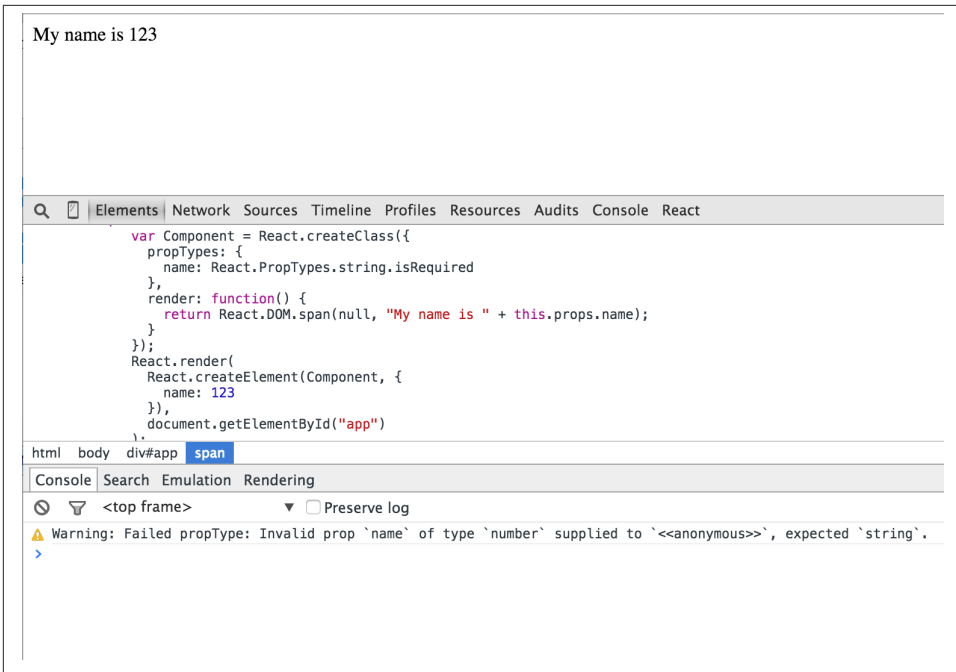


Figure 2-4. Warning when providing an invalid type

Appendix A is a detailed description of your options when it comes to using `React.PropTypes`, but [Figure 2-5](#) can give you a taste.

```
Console Search Emulation Rendering
<top frame> [x] Preserve log
> Object.keys(React.PropTypes).join('\n')
< "array
  bool
  func
  number
  object
  string
  any
  arrayOf
  element
  instanceOf
  node
  objectOf
  oneOf
  oneOfType
  shape"
> |
```

Figure 2-5. Listing all `React.PropTypes`



Declaring `propTypes` in your components is optional, which also means that you can have some, but not all, properties listed in there. You can tell it's a bad idea to not declare all properties, but bear in mind it's possible when you debug other people's code.

State

The examples so far were pretty static (or “stateless”). The goal was just to give you an idea of the building blocks when it comes to composing your UI. But where React really shines (and where old-school browser DOM manipulation and maintenance gets complicated) is when the data in your application changes. React has the concept of *state* which is the data your component uses to render itself. When state changes, React rebuilds the UI without you having to do anything. So all you care about after initially building the UI is updating the data and not worry about UI changes at all. After all, your `render()` method has already provided the blueprint of what the component should look like.

Similarly to how properties work, you access the state via `this.state` object. To update the state you use `this.setState()`. When `this.setState()` is called, React calls your `render()` method and updates the UI.



The UI updates after calling `setState()` are done using a queuing mechanism that efficiently batches changes, so updating `this.state` directly can have unexpected behavior and you shouldn't do it. Just like with `this.props`, consider the `this.state` object read-only, not only because it's semantically a bad idea, but because it can act in ways you don't expect.

A stateful textarea component

Let's build a new component - a textarea that keeps count of the number of characters typed in (Figure 2-6).



Figure 2-6. The end result of the custom textarea component

You (as well as other consumers of this reusable component) can use the new component like so:

```
React.render(  
  React.createElement(TextAreaCounter, {  
    text: "Bob"  
  }),  
  document.getElementById("app")  
);
```

Now, let's implement the component. Start first by creating a "stateless" version that doesn't handle updates, since this is no different than the previous examples.

```
var TextAreaCounter = React.createClass({  
  propTypes: {  
    text: React.PropTypes.string  
  },  
  render: function() {  
    return React.DOM.div(null,  
      React.DOM.textarea({  
        defaultValue: this.props.text  
      }),  
    );  
  },  
});
```

```
    React.DOM.h3(null, this.props.text.length)
  );
}
```



You may have noticed that the textarea in the snippet above takes a `defaultValue` property, as opposed to a text child like you're used to in regular HTML. This is because there are some slight differences between React and old-school HTML forms. These are discussed in chapter 4 and rest assured there are not a lot of them and they tend to make sense and make your life as a developer better.

As you can see, the component takes an optional text string property and renders a textarea with the given value, as well as an `<h3>` element that simply displays the string's length (Figure 2-7).

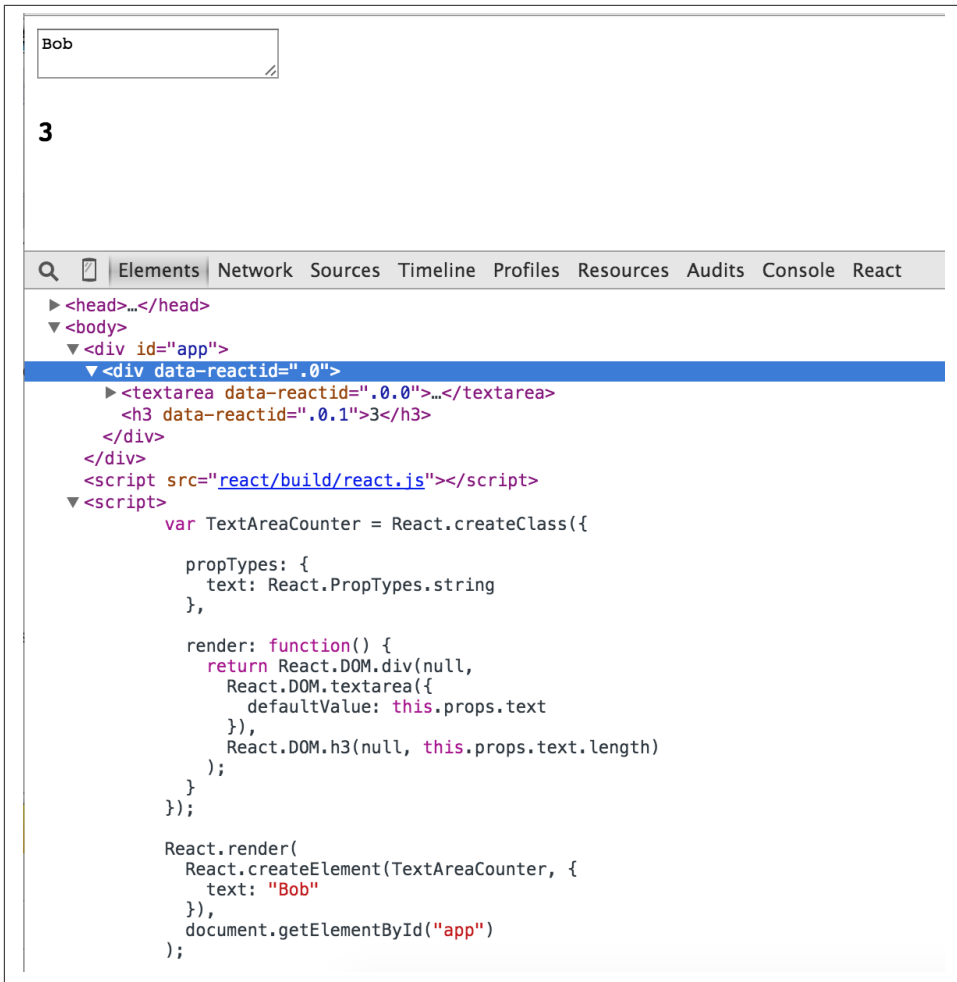


Figure 2-7. *TextAreaCounter* component in action

Next step is to turn this *stateless* component into a *stateful* one. In other words let's have the component maintain some data (state) and use this data to render itself initially and later on update itself (rerender) when data changes.

First step is to implement a method in your component called `getInitialState()` so you're sure you always work with sane data.

```
getInitialState: function() {  
  return {  
    text: this.props.text,  
  };  
},
```

The data this component will maintain is simply the text of the textarea, so the state has only one property called `text` and accessible via `this.state.text`. Initially you just copy the `text` property. Later when data changes (user is typing in the textarea) the component updates its state using a helper method.

```
  _textChange: function(ev) {
    this.setState({
      text: ev.target.value
    });
  },
```

You always update the state with `this.setState()` which takes an object and merges it with the already existing data in `this.state`. As you can probably guess, `_textChange()` is an event listener that takes an event `ev` object and reaches into it to get the text of the textarea input.

The last thing left is to update the `render()` method to use `this.state` instead of `this.props` and to setup the event listener.

```
  render: function() {
    return React.DOM.div(null,
      React.DOM.textarea({
        value: this.state.text,
        onChange: this._textChange
      }),
      React.DOM.h3(null, this.state.text.length)
    );
  }
```

Now whenever the user types into the textarea, the value of the counter updates to reflect the contents (Figure 2-8).

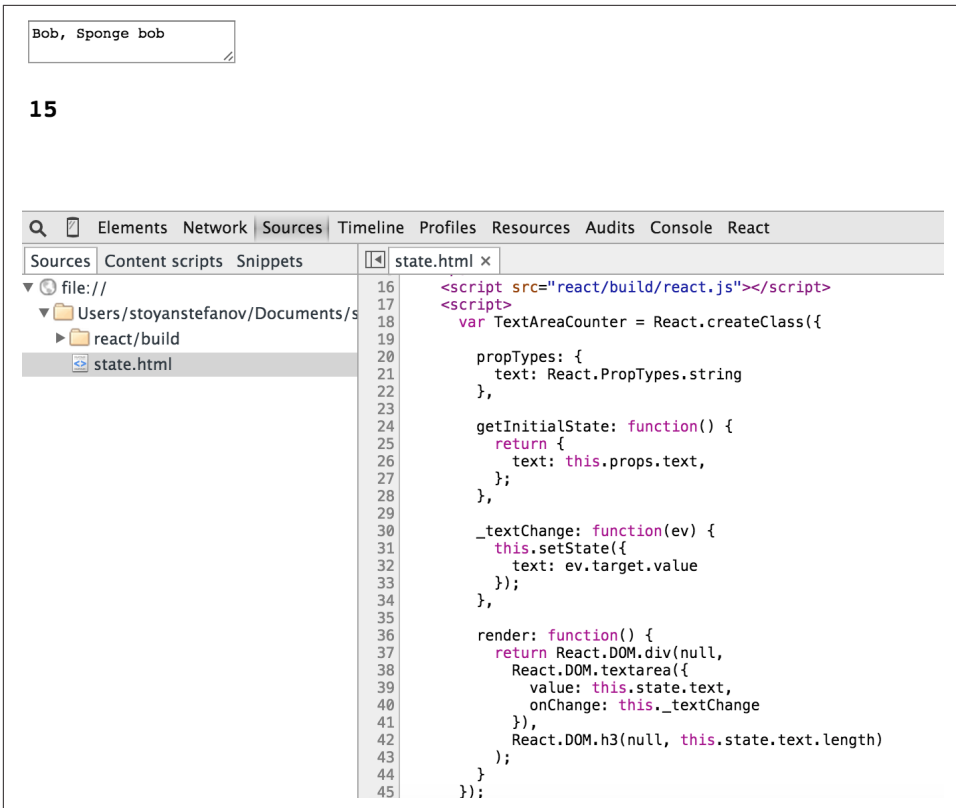


Figure 2-8. Typing in the textarea

A note on DOM events

To avoid any confusion, a few clarifications are in order regarding the line `onChange: this._textChange`.

React uses its own *synthetic* events system for performance as well as convenience and sanity reasons. To help understand why, you need to consider how things are done in the pure DOM world.

Event handling in the olden days

It's very convenient to use *inline* event handlers to do things like this:

```
<button onclick="doStuff">
```

While convenient and easy to read (the event listener is right there with the UI), it's inefficient to have too many event listeners scattered like this. It's also hard to have more than one listener on the same button, especially if the said button is in somebody else's

“component” or library and you don’t want to go in there and “fix” or fork their code. That’s why in the DOM world people use `element.addEventListener` to setup listeners (which now leads to having code in two places or more) and *event delegation* (to address the performance issues). Event delegation means you listen to events at some parent node, say a `<div>` that contains many `<button>`’s and setup one listener for all the buttons.

With event delegation you do something like

```
<div id="parent">
  <button id="ok">OK</button>
  <button id="cancel">Cancel</button>
</div>

<script>
document.getElementById('parent').addEventListener('click', function(event) {
  var button = event.target;

  // do different things based on which button was clicked
  switch (button.id) {
    case 'ok':
      console.log('OK!');
      break;
    case 'cancel':
      console.log('Cancel');
      break;
    default:
      new Error('Unexpected button ID');
  };
});
</script>
```

This works and performs fine, but there are drawbacks: * declaring the listener is further away from the UI component which makes it harder to find and debug * using delegation and always ‘switch’ing creates unnecessary boilerplate code even before you get to do the actual work (responding to a button click in this case) * browser inconsistencies (omitted here) actually require this code to be longer

Unfortunately, when it comes to taking this code live in front of real users, you need a few more additions in order to support all browsers: * You need `attachEvent` in addition to `addEventListener`. * You need `var event = event || window.event;` at the top of the listener * You need `var button = event.target || event.srcElement;`

All of these are necessary and annoying enough that you end up using an event library of some sorts. But why add another library (and study more APIs), when React comes bundled with a solution to the event handling nightmares.

Event handing in React

React uses synthetic events in order to wrap and normalize the browser events, so this means no more browser inconsistencies. You can always rely that `event.target` is available to you in all browsers. That's why in the `TextAreaCounter` snippet you only need `ev.target.value` and it just works. It also means the API to cancel events is the same in all browsers, in other words `event.stopPropagation()` and `event.preventDefault()` work even in old IEs.

The syntax makes it easy to have the UI and the event listener together. It looks like old-school inline event handlers, but behind the scenes it's not. Behind the scenes, React uses event delegation for performance reasons, but you don't need to worry about this.

React uses camelCase syntax for the event handlers, so you do `onClick` instead of `onclick`.

If you need the original browser event for whatever reason, it's available to you as `event.nativeEvent`, but it's unlikely you ever need to go there.

And one more thing: the `onChange` event (as used in the `textarea` example above) behaves as you'd expect. It fires when a person types, as opposed to when they're done typing and navigate away from the field, which is the behavior in plain DOM.

Props vs State

Now you know that you have access to `this.props` and `this.state` when it comes to displaying your component in your `render()` method. You may be asking when you should use one and not the other.

Properties are a mechanism for the outside world (users of the component) to configure your component. State is your internal data maintenance. So `this.props` is like public properties in object-oriented parlance and `this.state` is a bag of your private properties.

You can also choose to keep a certain property in sync with the state, if this is useful to the callers, by updating the property with `this.setProps()` in addition to `this.setState()`.

The state vs props distinction is more of a maintainability concern when building large apps with lots of components, rather than a programming concern. In fact, there's nothing that will prevent you from always using `this.props` to maintain state as your component will rerender if you use `this.setProps()` too. Here's the component again, rewritten to only use props and no state:

```
var TextAreaCounter = React.createClass({
```

```

propTypes: {
  value: React.PropTypes.string
},

_textChange: function(ev) {
  this.setProps({
    value: ev.target.value
  });
},

render: function() {
  return React.DOM.div(null,
    React.DOM.textarea({
      value: this.props.value,
      onChange: this._textChange
    }),
    React.DOM.h3(null, this.props.value.length)
  );
}
});

```

Props in initial state: an anti-pattern

Previously you saw an example of using `this.props` inside of the `getInitialState()`:

```

getInitialState: function() {
  return {
    text: this.props.text,
  };
},

```

This is actually considered an anti-pattern. Ideally you use any combination of `this.state` and `this.props` as you see fit to build your UI in your `render()` method. But sometimes you want to take a value passed to your component and use it to construct the initial state. There's nothing wrong with this, only that the callers of your component may expect the property (`text` in the example above) to always have the latest value and the example violated this expectation. To set expectation straight, a simple naming change is sufficient, e.g. calling the property something like `defaultText` or `initialValue` instead of just `text`.

```

propTypes: {
  defaultValue: React.PropTypes.string
},

getInitialState: function() {
  return {
    text: this.props.defaultValue,
  };
},

```



Chapter 4 illustrates how React solves this for its own implementation of inputs and textareas where people may have expectations coming from their prior HTML knowledge.

Accessing the component from the outside

You don't always have the luxury of starting a brand new React app from scratch. Sometimes you need to hook into an existing application or a website and migrate to React one piece at a time. React was designed to work with any pre-existing codebase you might have. After all, the original creators of React couldn't stop the world or rewrite an entire huge application (Facebook) completely from scratch.

You can, for example, get a reference to a component you render with `React.render()` and use from outside of the component:

```
var myTextAreaCounter = React.render(  
  React.createElement(TextAreaCounter, {  
    defaultValue: "Bob"  
  }),  
  document.getElementById("app")  
);
```

Now you can use `myTextAreaCounter` to access the same methods and properties you normally access with `this` when inside the component. You can even play with the component using your JavaScript console ([Figure 2-9](#)).

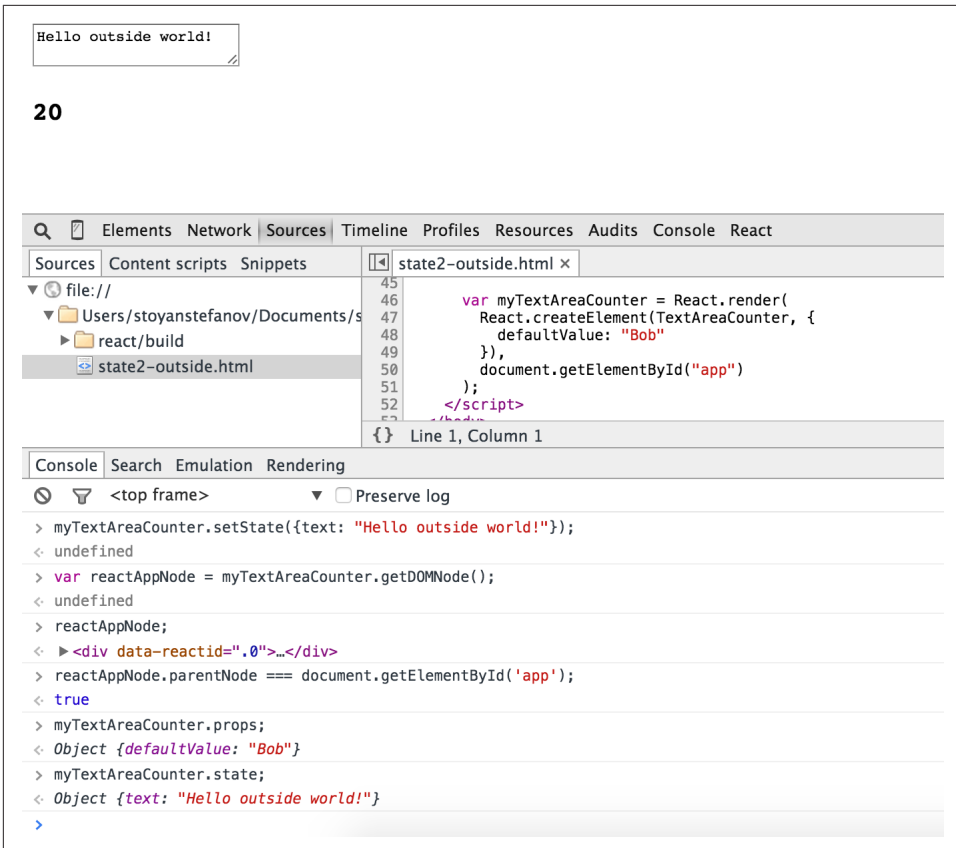


Figure 2-9. Accessing the rendered component by keeping a reference

Setting some new state:

```
myTextAreaCounter.setState({text: "Hello outside world!"});
```

Getting reference to the main parent DOM node that React created:

```
var reactAppNode = myTextAreaCounter.getDOMNode();
```

This is the first child of the `<div id="app">` which is where you told React to do its magic:

```
reactAppNode.parentNode === document.getElementById('app'); // true
```

Properties and state:

```
myTextAreaCounter.props; // Object { defaultValue: "Bob"}
myTextAreaCounter.state; // Object { text: "Hello outside world!"}
```



You have access to all of the component API from outside of your component. But you should use your new superpowers sparingly, if at all. Maybe `getDOMNode()` if you need to get the dimensions of the node to make sure it fits your overall page. Or more rarely `setProps()` if you need to update the app based on new user input received from outside the application. But not much else really. It may be tempting to fiddle with the state of components you don't own and "fix" them, but you'll be violating expectations and cause bugs down the road as the component doesn't expect such intrusions.

Changing properties mid-flight

As you already know, properties are a way to configure a component. So using `setProps()` from the outside after the component has been created can be justified. But your component should be prepared to handle this scenario.

If you take a look at the `render()` method from the previous examples, it only uses `this.state`:

```
render: function() {  
  return React.DOM.div(null,  
    React.DOM.textarea({  
      value: this.state.text,  
      onChange: this._textChange  
    }),  
    React.DOM.h3(null, this.state.text.length)  
  );  
}
```

If you change the properties from the outside of the component, this will have no rendering effect. In other words the textarea contents will not change after you do:

```
myTextAreaCounter.setProps({defaultValue: 'Hello'});
```

The contents of `this.props` will change (but the UI will not):

```
myTextAreaCounter.props; // Object { defaultValue="Hello"}
```



Setting the state *will* update the UI:

```
// COUNTEREXAMPLE  
myTextAreaCounter.setState({text: 'Hello'});
```

But this is a bad idea because it may result in inconsistent state in more complicate components, for example mess up internal counters, boolean flags, event listeners and so on.

If you want to handle this outside intrusion gracefully, you can prepare by implementing a method called `componentWillReceiveProps()`:

```
componentWillReceiveProps: function(newProps) {
  this.setState({
    text: newProps.defaultValue
  });
},
```

As you see this method receives the new props object and you can set the `state` accordingly, as well as do any other work when required to keep the component in a sane state.

Lifecycle methods

The method `componentWillReceiveProps` from the previous snippet is one of the so called *lifecycle* methods that React offers. You can use the lifecycle methods to listen to changes such as property updates (as in the example above). Other lifecycle methods you can implement include:

- `componentWillUpdate()` - executed before the `render()` method of your component is called again (as a result to changes to the properties or state)
- `componentDidUpdate()` - executed after the `render()` method is done and the new changes to the underlying DOM have been applied
- `componentWillMount()` - executed before the node is inserted into the DOM
- `componentDidMount()` - after the node is inserted into the DOM
- `componentWillUnmount()` - right before the component is removed from the DOM
- `shouldComponentUpdate(newProps, newState)` - this method is called before `componentWillUpdate()` and gives you a chance to return `false`; and cancel the update. It's useful in performance-critical areas of the app when you think nothing interesting changed and no rerendering is necessary. You make this decision based on comparing the `newState` argument with the existing `this.state` or comparing `newProps` with `this.props` or just simply knowing that this component is static and doesn't change. (You'll see an example shortly.)

Lifecycle example: log it all

To better understand the life of a component, let's add some logging in the `TextAreaCounter` component. Simply implement all of these lifecycle methods to log to the console when they are invoked, together with any arguments.

```
var TextAreaCounter = React.createClass({
  _log: function(methodName, args) {
    console.log(methodName, args);
```

```

    },
    componentWillUpdate: function() {this._log('componentWillUpdate', arguments);},
    componentDidUpdate: function() {this._log('componentDidUpdate', arguments);},
    componentWillMount: function() {this._log('componentWillMount', arguments);},
    componentDidMount: function() {this._log('componentDidMount', arguments);},
    componentWillUnmount: function() {this._log('componentWillUnmount', arguments);},

    // ...
    // more implementation, render() etc....

  };

```

Figure 2-10 shows what happens after you load the page.

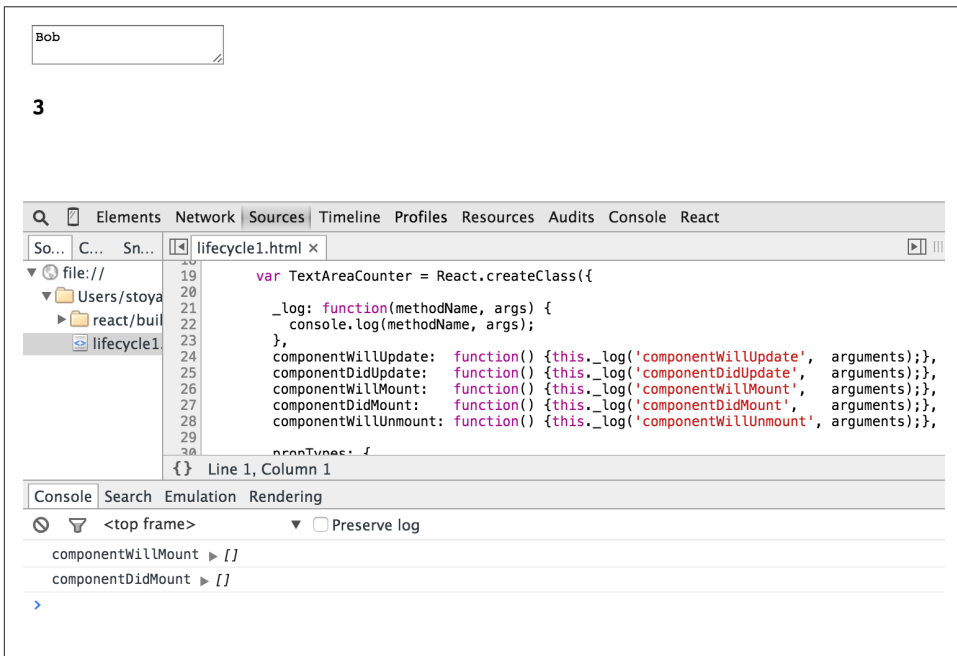


Figure 2-10. Mounting the component

As you see, two methods were called without any arguments. The method `componentDidMount()` is usually the more interesting of the two. You can get access to the freshly mounted DOM node with `this.getDOMNode()` if you need, for example, to get the dimensions of the component. You can also do any sort of initialization work for your component now that your component is alive.

Next, what happens when you type “s” to make the text “Bobs”? (Figure 2-11)

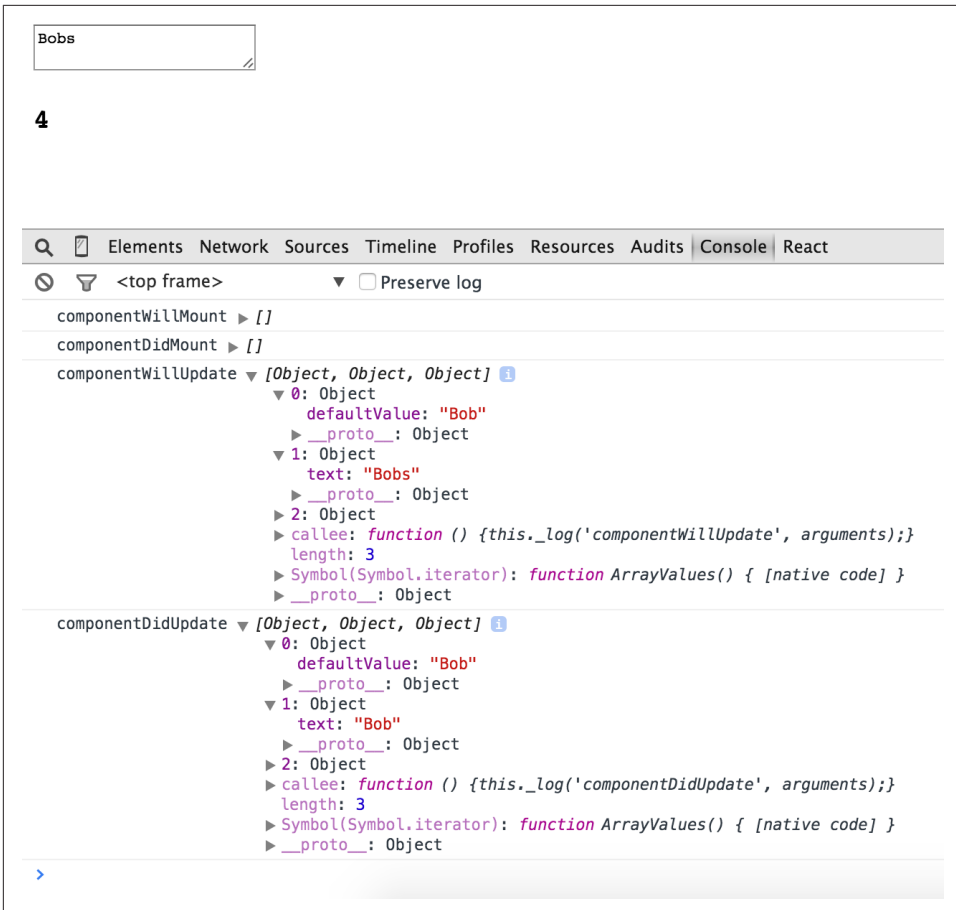


Figure 2-11. Updating the component

The method `componentWillUpdate(nextProps, nextState)` is called with the new data that will be used to rerender the component. The first argument is the new future value of `this.props` (which doesn't change in this example), the second is the future value of the new `this.state`. The third is context which is not that interesting at this stage. You can compare the arguments e.g. `newProps` with the current `this.props` and decide whether to act on it.

After `componentWillUpdate()`, next you see that `componentDidUpdate(oldProps, oldState)` is called, passing the values of what props and state used to be before the change. This is an opportunity to do something after the change. You can use `this.setState()` here, which you cannot do in `componentWillUpdate()`.



Say you want to restrict the number of characters to be typed in the textarea. You should do this in the event handler `_textChange()` which is called as the user types. But what if someone (a younger, more naive you?) calls `setState()` from the outside of the component? Which, as mentioned earlier, is a bad idea. Can you still protect the consistency and well-being of your component? Sure. You can do the same validation in `componentDidUpdate()` and if the number of characters is greater than allowed, revert the state back to what it was. Something like:

```
componentDidUpdate: function(oldProps, oldState) {  
  if (this.state.text.length > 3) {  
    this.replaceState(oldState);  
  }  
},
```

This may seem overly paranoid, but it's still possible to do.

Note the use of `replaceState()` instead of `setState()`. While `setState(obj)` merges the properties of `obj` with these of `this.state`, `replaceState()` completely overwrites everything.

Lifecycle example: use a mixin

In the example above you saw four out of the five lifecycle method calls being logged. The fifth, `componentWillUnmount()`, is best demonstrated when you have children components that are removed by a parent. But since the child and the parent both want to log the same method calls, let's introduce a new concept for reusing code - a mixin.

A mixin is a JavaScript object that contains a collection of methods and properties. A mixin is not meant to be used on its own, but included (mixed-in) into another object's properties. In the logging example, a mixin can look like so:

```
var logMixin = {  
  _log: function(methodName, args) {  
    console.log(this.name + '::' + methodName, args);  
  },  
  componentWillUpdate: function() {this._log('componentWillUpdate', arguments);},  
  componentDidUpdate: function() {this._log('componentDidUpdate', arguments);},  
  componentWillMount: function() {this._log('componentWillMount', arguments);},  
  componentDidMount: function() {this._log('componentDidMount', arguments);},  
  componentWillUnmount: function() {this._log('componentWillUnmount', arguments);},  
};
```

In non-React world you can loop with `for-in` and copy all properties into your new object and this way gaining the mixin's functionality. In React world, you have a shortcut - the `mixins` property. It looks like so:

```
var MyComponent = React.createClass({
```

```

mixins: [obj1, obj2, obj3],

// the rest of the methods ...

};

```

You assign an array of JavaScript objects to the `mixins` property and React takes care of the rest. Including the `logMixin` into your component looks like:

```

var TextAreaCounter = React.createClass({
  name: 'TextAreaCounter',
  mixins: [logMixin],
  // all the rest..
});

```

As you see, the snippet also adds a convenience `name` property to identify the caller.

If you run the example with the mixin, you can see the logging in action ([Figure 2-12](#))

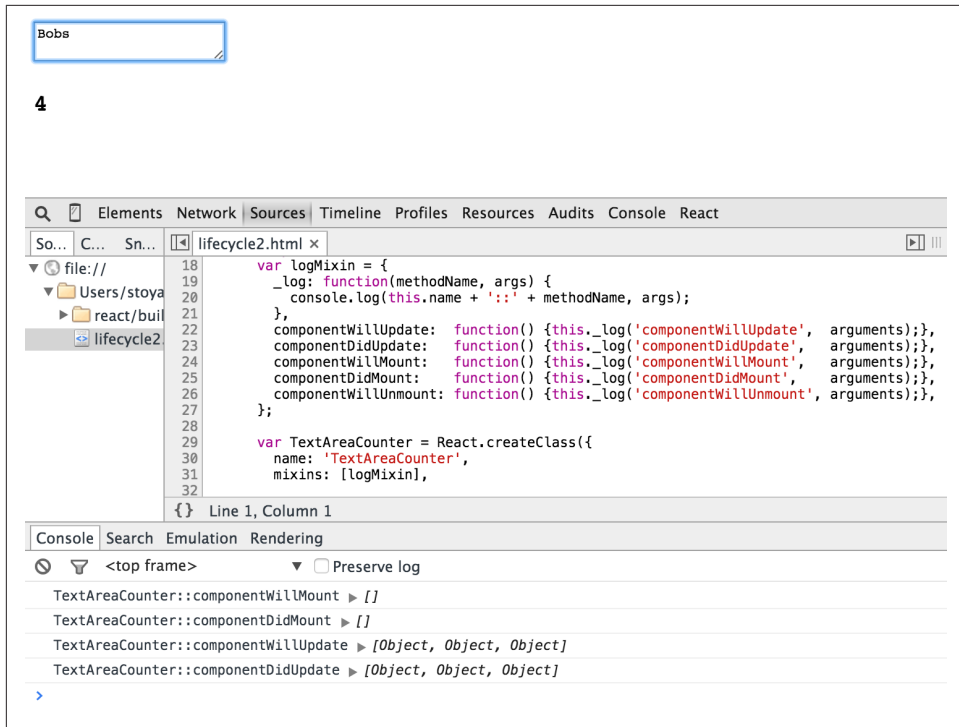


Figure 2-12. Using a mixin and identifying the component

Lifecycle example: with a child component

You know you can mix and nest React components as you see fit. So far you've only seen `React.DOM` components (as opposed to custom ones) in the `render()` methods. Let's take a look at a simple custom component used as a child.

You can isolate the counter part into its own component:

```
var Counter = React.createClass({
  name: 'Counter',
  mixins: [LogMixin],
  propTypes: {
    count: React.PropTypes.number.isRequired,
  },
  render: function() {
    return React.DOM.span(null, this.props.count);
  }
});
```

This component is just the count part, it renders a `<span>` and doesn't maintain state, just displays the count property given by the parent. It also mixes-in the `LogMixin` to log when the lifecycle methods are being called.

Now let's update the `render()` method of the parent `TextAreaCounter` component. It should use the `Counter` component conditionally: if the count is 0, don't even show a number.

```
render: function() {
  var counter = null;
  if (this.state.text.length > 0) {
    counter = React.DOM.h3(null,
      React.createElement(Counter, {
        count: this.state.text.length
      })
    );
  }
  return React.DOM.div(null,
    React.DOM.textarea({
      value: this.state.text,
      onChange: this._textChange
    }),
    counter
  );
}
```

The counter variable is `null` when the textarea is empty. When there is some text, the counter variable contains the part of the UI responsible for showing the number of characters. You see that all the UI doesn't need to be inline as arguments to the main `React.DOM.div` component. You can assign UI bits and pieces to variables and use them conditionally.

You can now observe the lifecycle methods being logged for both components. **Figure 2-13** shows what happens when you load the page and then change the contents of the textarea.

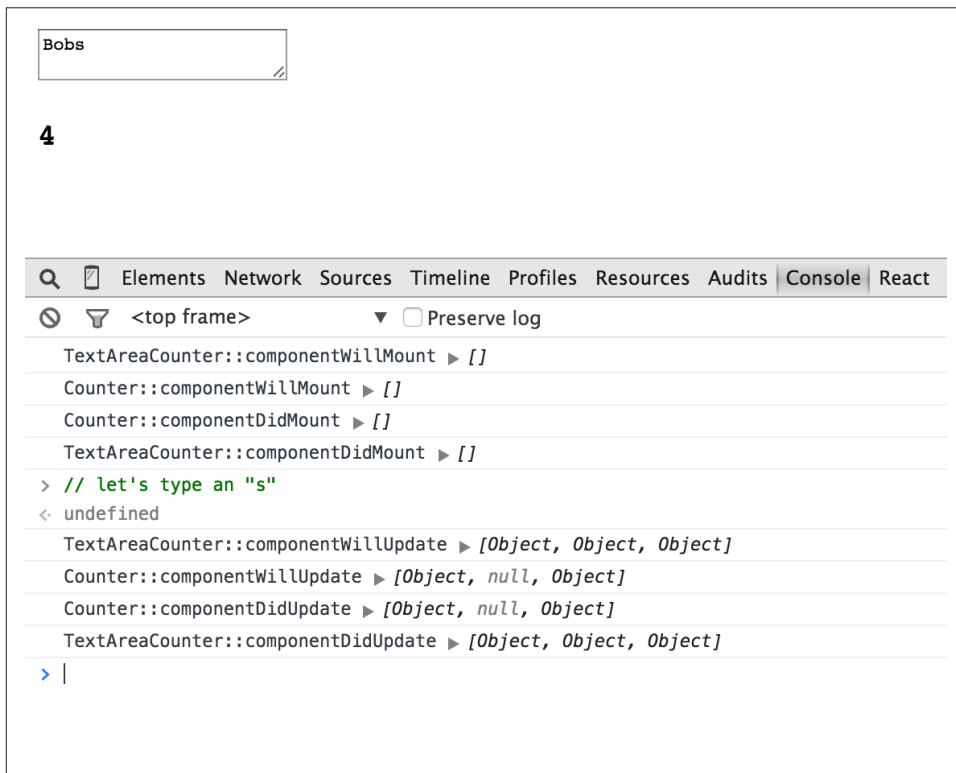


Figure 2-13. Mounting and updating two components

You can see how the child component is mounted and updated before the parent.

Figure 2-14 shows what happens after you delete the text in the textarea and count becomes 0. In this case the Counter child becomes null and its DOM node is removed from the DOM tree, after notifying you via the `componentWillUnmount` callback.

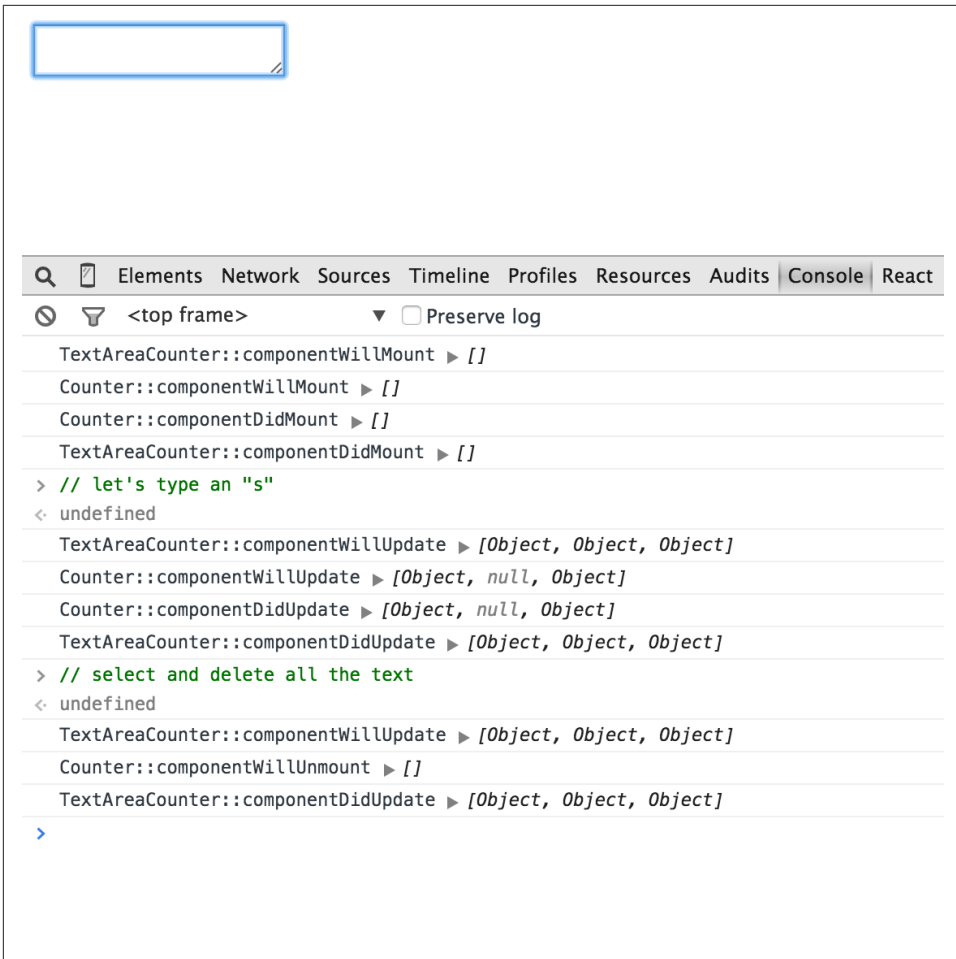


Figure 2-14. Unmounting the counter component

Performance win: prevent component updates

The last lifecycle method you should know about, especially when building performance critical parts of your app, is the method `shouldComponentUpdate(next Props, nextState)`. It's invoked before `componentWillUpdate()` and gives you a chance to cancel the update if you decide it's not necessary.

There is a class of components which only use `this.props` and `this.state` in their `render()` methods and no additional function calls. These components are called “pure” components. They can implement `shouldComponentUpdate()` and compare the state and the properties before and after and if there aren't any changes, return `false` and

save some processing power. Additionally, there can be pure static components that don't use neither props, nor state. These can straight out return false.

Let's explore what happens with the calls to `render()` methods and implement `shouldComponentUpdate()` to win on the performance front.

First, take the new `Counter` component. Remove the logging mixin and instead, log to the console any time the `render()` method is invoked.

```
var Counter = React.createClass({
  name: 'Counter',
  // mixins: [logMixin],
  propTypes: {
    count: React.PropTypes.number.isRequired,
  },
  render() {
    console.log(this.name + '::
```

Do the same in the `TextAreaCounter`.

```
var TextAreaCounter = React.createClass({
  name: 'TextAreaCounter',
  // mixins: [logMixin],

  // all other methods...

  render: function() {
    console.log(this.name + '::
```

Now when you load the page and paste the string “LOL” replacing “Bob”, you can see the result shown in **Figure 2-15**

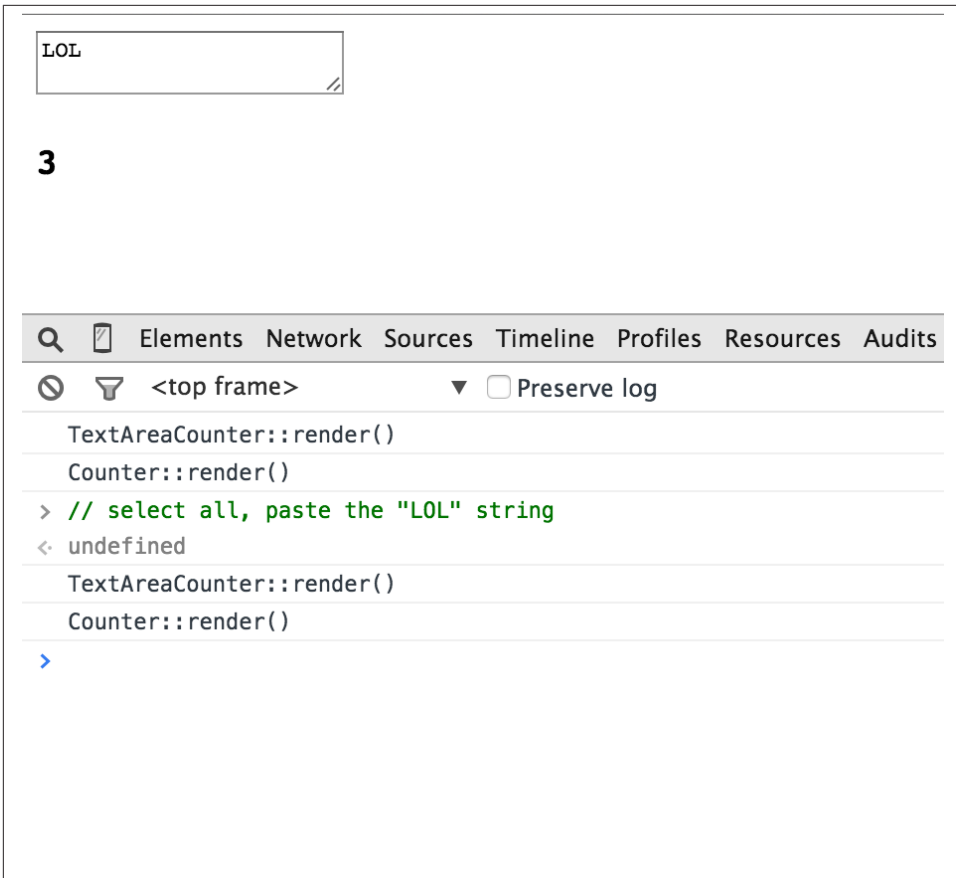


Figure 2-15. Re-rendering both components

You see that updating the text results in calling `TextAreaCounter`'s `render()` method which in turn calls `Counter`'s `render()`. When replacing "Bob" with "LOL", the number of characters before and after the update is the same, so there's no change in the UI of the counter and calling `Counter`'s `render()` is not necessary. You can help React optimize this case by implementing `shouldComponentUpdate()` and returning `false` when no further rendering is necessary. The method receives the future values of `props` and `state` (`state` is not needed in this component) and inside of it you compare the current and the next values.

```
shouldComponentUpdate(nextProps, nextState_ignore) {  
  return nextProps.count !== this.props.count;  
},
```

Doing the same "Bob" to "LOL" update doesn't cause the `Counter` to be rerendered anymore (Figure 2-16)

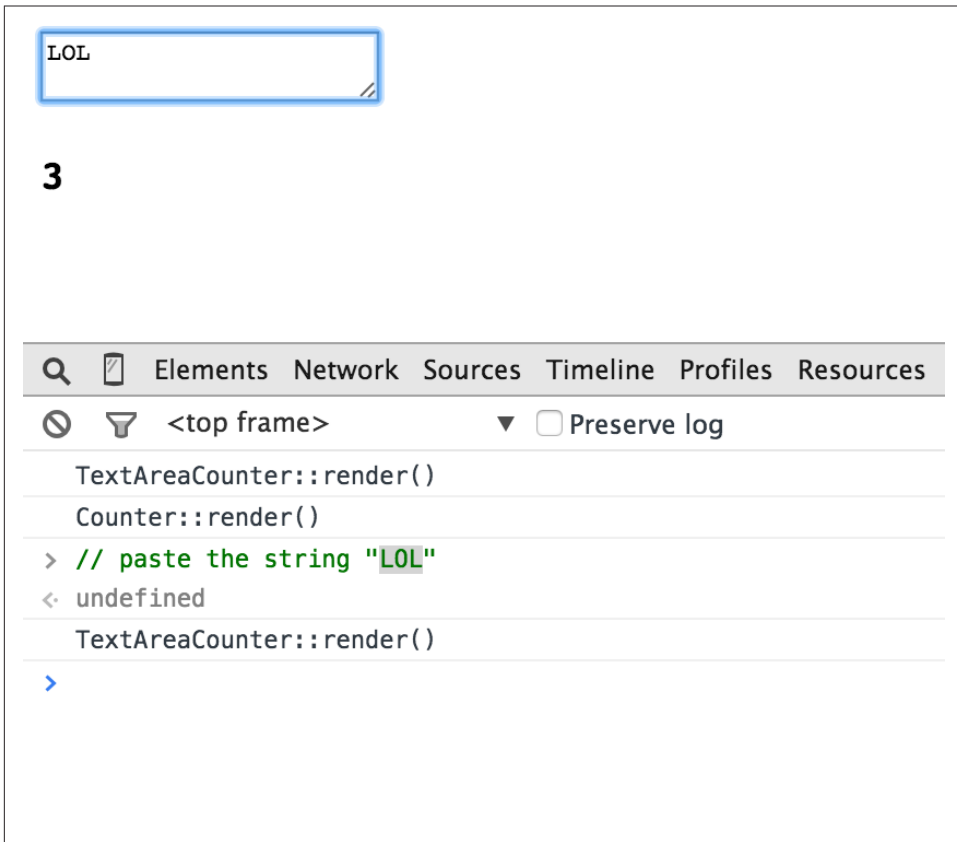


Figure 2-16. Performance win: saving one rerendering cycle

PureRenderMixin

The implementation of `shouldComponentUpdate()` above is pretty simple. And it's not a big stretch to make this implementation generic, since you always compare `this.props` with `nextProps` and `this.state` with `nextState`. React provides one such generic implementation in the form of a mixin you can simply include in any component.

Here's how:

```
<script src="react/build/react-with-addons.js"></script>
<script>

var Counter = React.createClass({
  name: 'Counter',
  mixins: [React.addons.PureRenderMixin],
```

```

    propTypes: {
      count: React.PropTypes.number.isRequired,
    },
    render: function() {
      console.log(this.name + '::render()');
      return React.DOM.span(null, this.props.count);
    }
  });

  // ....
</script>

```

And the result (Figure 2-17) is the same - Counter's render() is not called when there's no change in the number of characters.

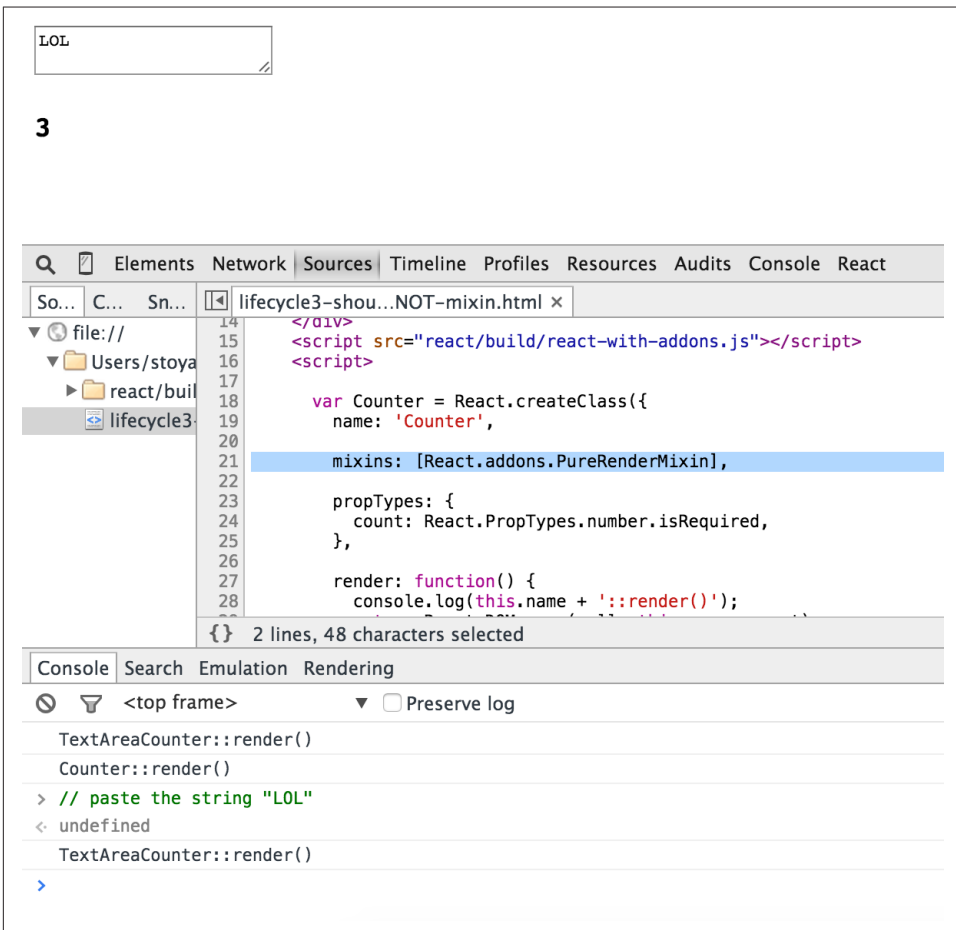


Figure 2-17. Easy perf win: mix in the PureRenderMixin

Note that `PureRenderMixin` is not part of React core, but is part of an extended version of React + addons. So in order to gain access to it, you include `react/build/react-with-addons.js` as opposed to `react/build/react.js`. This gives you a new namespace `React.addons` and that's where you can find the `PureRenderMixin` as well as other add-on goodies.

If you don't want addons or want to implement your own version of the mixin, feel free to peek into the implementation. It's pretty simple and straightforward, just a shallow (non-recursive) check for equality.

```
var ReactComponentWithPureRenderMixin = {  
  shouldComponentUpdate: function(nextProps, nextState) {  
    return !shallowEqual(this.props, nextProps) ||  
           !shallowEqual(this.state, nextState);  
  }  
};
```

Excel: a fancy table component

Now you know how to:

- create custom react components
- compose (render) UI using generic DOM ones as well as your own custom components
- set properties
- maintain state
- hook into the lifecycle of a component
- optimize performance by not rerendering when not necessary

Let's put all of this together (and learn more about React while at it) by creating a more interesting component - a data table. Something like Microsoft Excel v.0.1.beta that lets you edit the content of a table, and also sort, search (filter), and export the content as CSV or JSON.

Data first

Tables are all about the data, so the fancy table component (why not call it Excel?) should take an array of data and an array of headers. For testing, let's grab a list of bestselling books from Wikipedia (http://en.wikipedia.org/wiki/List_of_best-selling_books).

```
var headers = [  
  "Book", "Author", "Language", "Published", "Sales"  
];  
  
var data = [  
  ["The Lord of the Rings", "J. R. R. Tolkien",  
   "English", "1954-1955", "150 million"],  
  ["Le Petit Prince (The Little Prince)", "Antoine de Saint-Exupéry",
```

```

    "French", "1943", "140 million"],
    ["Harry Potter and the Philosopher's Stone", "J. K. Rowling",
     "English", "1997", "107 million"],
    ["And Then There Were None", "Agatha Christie",
     "English", "1939", "100 million"],
    ["Dream of the Red Chamber", "Cao Xueqin",
     "Chinese", "1754-1791", "100 million"],
    ["The Hobbit", "J. R. R. Tolkien",
     "English", "1937", "100 million"],
    ["She: A History of Adventure", "H. Rider Haggard",
     "English", "1887", "100 million"]
  ];

```

Table headers loop

First step, just to take off the ground, is to display only the headers. Here's what a bare-bone implementation may look like:

```

var Excel = React.createClass({
  render: function() {
    return (
      React.DOM.table(null,
        React.DOM.thead(null,
          React.DOM.tr(null,
            this.props.headers.map(function(title) {
              return React.DOM.th(null, title);
            })
          )
        )
      )
    );
  }
});

```

Using the component:

```

React.render(
  React.createElement(Excel, {
    headers: headers,
    initData: data
  }),
  document.getElementById("app")
);

```

The result of this get-off-the-ground example is shown on [Figure 3-1](#)

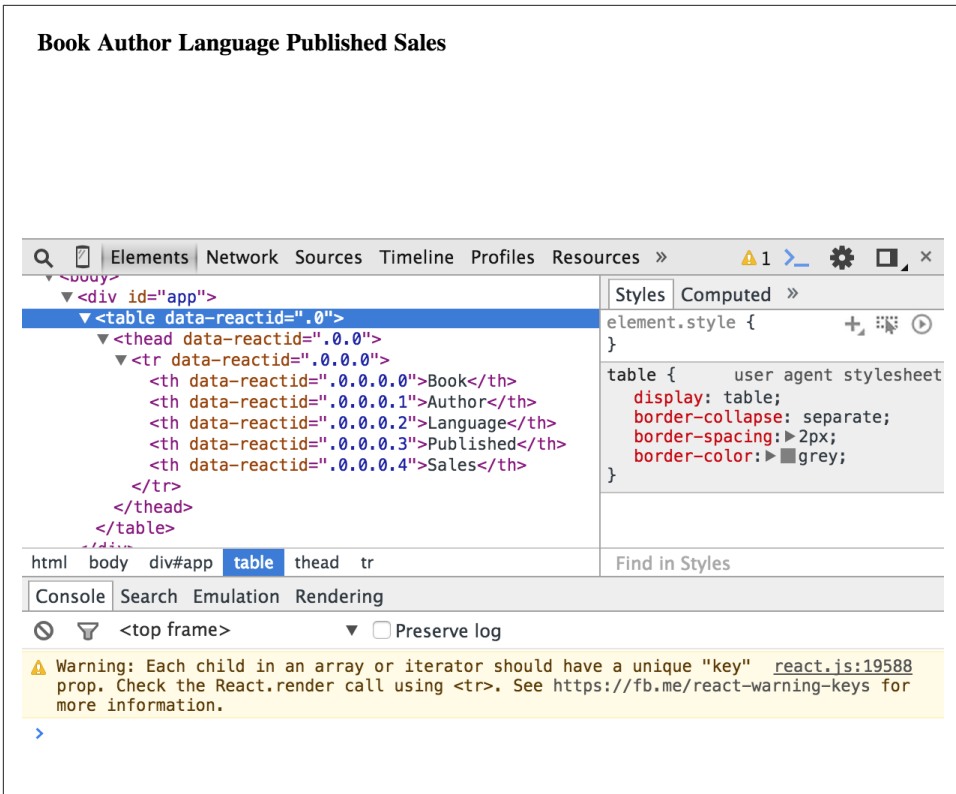


Figure 3-1. Rendering table headers

There's something new here - the array's `map()` method used to return an array of children components. The `map()` method of an array takes each element of the array (the headers array in this case) and passes it to a callback function. In this case the callback function creates a new `<th>` component and returns it.

This is part of the beauty of React - you use JavaScript to create your UI and all the power of JavaScript is available to you. Loops and conditions all work as usual and you don't need to learn another "templating" language or syntax to build the UI.



You can pass children to a component as a single array argument instead of what you've seen so far which was: passing each child as separate argument. So these both work:

```

// separate arguments
React.DOM.ul(null, React.DOM.li(null, 'one'), React.DOM.li(null, 'two') );
// array
React.DOM.ul(null, [React.DOM.li(null, 'one'), React.DOM.li(null, 'two')]);

```

Debugging the console warning

Now let's talk about the warning you see in the screenshot and what to do about it. The warning says "Warning: Each child in an array or iterator should have a unique "key" prop. Check the React.render call using <tr>". Since there's only one component in this app, it's not a stretch to conclude that the problem is there, but potentially you may have a lot of components that create <tr>'s. `Excel` is just a variable that is assigned a React component outside of React's world, so React can't figure out a name for this component. You can help by declaring a `displayName` property.

```
var Excel = React.createClass({
  displayName: 'Excel',
  render: function() {
    // ...
  }
});
```

Now React can identify where the problem is and warn you that: "Each child in an array should have a unique "key" prop. Check the render method of Excel." Much better. But still there's a warning. To fix it, you simply do as the warning says, now that you know which `render()` is to blame.

```
this.props.headers.map(function(title, idx) {
  return React.DOM.th({key: idx}, title);
})
```

What happened here? The callback functions passed to the `Array.prototype.map()` method are supplied with three arguments - the array value, its index (0, 1, 2, etc.) and the whole array too. For the keys property you can use the index (`idx`) and be done with it. The keys only need to be unique inside of this array, not unique in the whole React application.

Now with the keys fixed and with the help of a little CSS you can enjoy version 0.0.1 of your new component - pretty and warning-free ([Figure 3-2](#))

Book	Author	Language	Published	Sales
------	--------	----------	-----------	-------

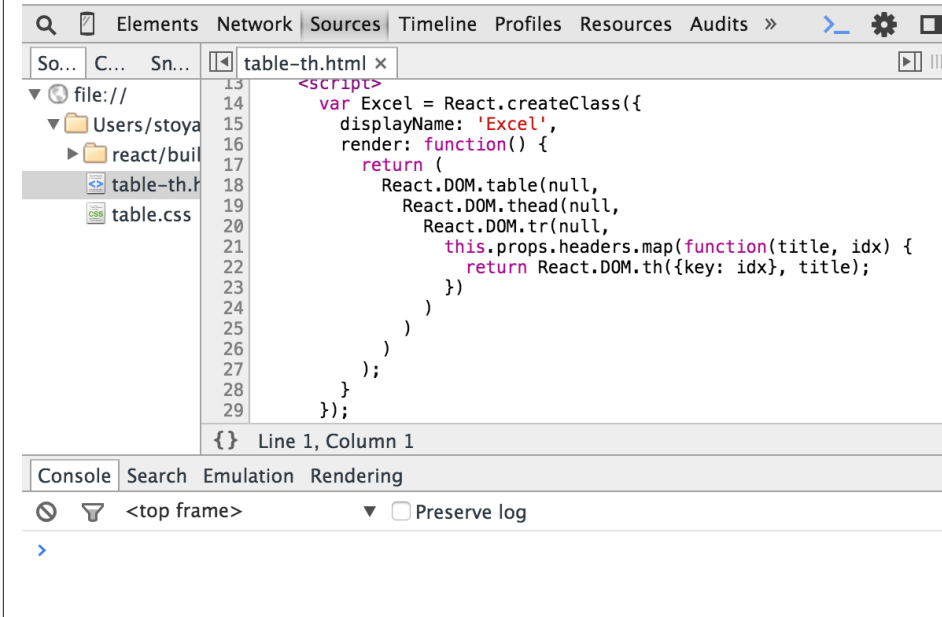


Figure 3-2. Rendering warning-free table headers



Adding `displayName` just to debug may look like a hassle, but there's a cure: when using JSX (next Chapter) you don't need define this property as the name is derived automatically.

Adding `<td>` content

Now that you have a pretty table head, time to add the body. The header content is a one-dimensional array (single row), but the data is two-dimensional. So you'll need two loops - one that goes through rows and one that goes through the data (cells) for each row. This can be accomplished using the same `.map()` loops you know already.

```

data.map(function (row) {
  return (

```

```

        React.DOM.tr(null,
            row.map(function (cell) {
                return React.DOM.td(null, cell);
            })
        )
    );
})

```

One more thing to consider is the contents of the data variable: where does it come from and how does it change? The caller of your Excel component should be able to pass data to initialize the table. But later, as the table lives on, the data will change, because the user should be able to sort, edit and so on. In other words the *state* of the component will change. So let's use `this.state.data` to keep track of the changes and use `this.props.initialData` to let the caller initialize the component. Now a complete implementation could look like this (result shown in [Link to Come]):

```

getInitialState: function() {
    return {data: this.props.initialData};
},

render: function() {
    return (
        React.DOM.table(null,
            React.DOM.thead(null,
                React.DOM.tr(null,
                    this.props.headers.map(function(title, idx) {
                        return React.DOM.th({key: idx}, title);
                    })
                )
            ),
            React.DOM.tbody(null,
                this.state.data.map(function (row, idx) {
                    return (
                        React.DOM.tr({key: idx},
                            row.map(function (cell, idx) {
                                return React.DOM.td({key: idx}, cell);
                            })
                        )
                    );
                })
            )
        )
    );
}

```

Rendering the whole table: [Figure to Come]

You see the repeating `{key: idx}` to give each child in an array of components a unique key. Although all the `.map()` loops start from index 0, this is not a problem as the keys only need to be unique in the current loop, not for the whole application.



The `render()` function is getting already a little bit harder to follow, especially keeping track of the closing `}`'s and ```'s. Fear not - JSX is coming to alleviate the pain!

The code snippet above is missing the optional, but often a good idea, `propTypes` property. It serves as both data validation and component documentation. Let's get really specific and try as hard as possible to reduce the probability of someone supplying junk data to the beautiful `Excel` component. `React.PropTypes` offers an array validator to make sure the property is always an array. But it goes further with `arrayOf` where you can specify the type of array elements. In this case, let's only accept strings for header titles and for data.

```
propTypes: {
  headers: React.PropTypes.arrayOf(
    React.PropTypes.string
  ),
  initialData: React.PropTypes.arrayOf(
    React.PropTypes.arrayOf(
      React.PropTypes.string
    )
  ),
},
```

Now that's strict!

Sorting

How many times you've seen a table on a web page that you wish was sorted differently? Luckily, it's trivial to do this with React. Actually this is an example where React shines because all you need to do is sort the data array and all of the UI updates are handled for you.

First, adding a click handler to the header row:

```
React.DOM.table(null,
  React.DOM.thead({onClick: this._sort},
    React.DOM.tr(null,
      // ...
```

Now let's implement the `_sort` function. You need to know which column to sort by, which can conveniently be retrieved by using the `cellIndex` property of the event target (the event target is a table header `<th>`):



You may have rarely seen `cellIndex` but it's a property defined as early as DOM Level 1 as "The index of this cell in the row" and later on made read-only in DOM Level 2. You may wonder if it's working in all browsers, but since you're using React none of this is your concern - you're now working with a consistent normalized API across browsers.

```
var column = e.target.cellIndex;
```

You also need a copy of the data to be sorted. Otherwise if you use the array's `sort()` method directly, it modifies the array, meaning `this.state.data.sort()` will modify `this.state`. As you know already `this.state` should not be modified directly, but only through `setState()`.

```
// copy the data  
var data = this.state.data.slice();
```

Now the actual sorting done via a callback to the `sort()` method:

```
data.sort(function(a, b) {  
  return a[column] > b[column];  
});
```

And finally, setting the state with the new, sorted data:

```
this.setState({  
  data: data  
});
```

Now, when you click a header, the contents gets sorted alphabetically (Figure 3-3)

Book	Author	Language	Published	Sales
And Then There Were None	Agatha Christie	English	1939	100 million
Dream of the Red Chamber	Cao Xueqin	Chinese	1754-1791	100 million
Harry Potter and the Philosopher's Stone	J. K. Rowling	English	1997	107 million
Le Petit Prince (The Little Prince)	Antoine de Saint-Exupéry	French	1943	140 million
She: A History of Adventure	H. Rider Haggard	English	1887	100 million
The Hobbit	J. R. R. Tolkien	English	1937	100 million
The Lord of the Rings	J. R. R. Tolkien	English	1954-1955	150 million

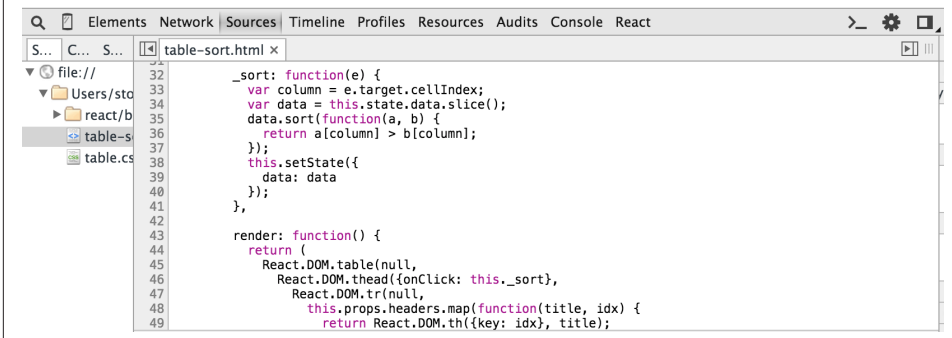


Figure 3-3. Sorting by book title

And this is it, you don't have to touch the UI rendering, in the `render()` method you've already defined once and for all how the component should look like given some data. When the data changes, so does the UI but this is none of your concern anymore.



This is pretty simple sorting, just enough to be relevant to the React discussion. You can go as fancy as you need, parsing the content to see if the values are numeric, with or without a unit of measure and so on.

Sorting UI cues

The table is nicely sorted, but it's not quite clear which column it's sorted by. Let's update the UI to show arrows based on the column being sorted. And while at it, let's add descending sorting too.

To keep track of the new state you'll need two new properties: `this.state.sortby` (the index of the column currently being sorted) and `this.state.descending` (a boolean to determine ascending vs. descending sorting).

```
getInitialState: function() {
  return {
```

```

    data: this.props.initialData,
    sortBy: null,
    descending: false
  };
},

```

In the `_sort()` function you have to figure out which way to sort. Default is ascending, unless the index of the new column is the same as the currently sort-by column and the sorting is not already descending:

```

var descending = this.state.sortby === column && !this.state.descending;

```

A tweak to the sorting callback is in order:

```

data.sort(function(a, b) {
  return descending
    ? a[column] < b[column]
    : a[column] > b[column];
});

```

And finally setting the new state:

```

this.setState({
  data: data,
  sortBy: column,
  descending: descending
});

```

The only thing left is to update the `render()` function to show an arrow symbol. For the currently sorted column let's just add an arrow symbol to the title.

```

this.props.headers.map(function(title, idx) {
  if (this.state.sortby === idx) {
    title += this.state.descending ? ' \u2191' : ' \u2193'
  }
  return React.DOM.th({key: idx}, title);
}).bind(this))

```

Now the sorting is feature-complete, people can sort by any column, they can click once for ascending and one more for descending ordering and the UI updates with the visual cue (Figure 3-4)

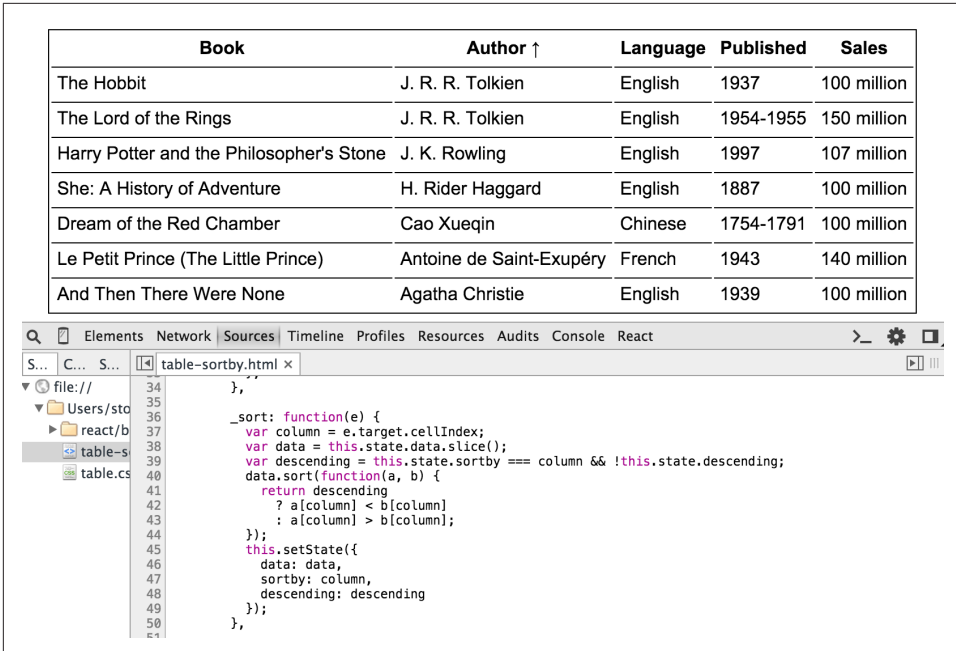


Figure 3-4. Ascending/descending sorting

Editing data

Next step for the Excel component is to give people the option to edit data in the table. One solution could work like so:

- Person double-clicks a cell. Excel figures out which cell was that and turns the cell content from simple text into an input field pre-filled with the content (Figure 3-5).
- Person edits the content (Figure 3-6).
- Then hits Enter. The input field is gone, the table is updated with the new text (Figure 3-7).

Book	Author	Language	Published	Sales
The Lord of the Rings	J. R. R. Tolkien	English	1954-1955	150 million
Le Petit Prince (The Little Prince)	Antoine de Saint-Exupéry	French	1943	140 million

Figure 3-5. Table cell turns into an input field

Book	Author	Language	Published	Sales
The Lord of the Rings	J. R. R. Tolkien	English	1954-1955	200 million
Le Petit Prince (The Little Prince)	Antoine de Saint-Exupéry	French	1943	140 million

Figure 3-6. Edit the content

Book	Author	Language	Published	Sales
The Lord of the Rings	J. R. R. Tolkien	English	1954-1955	200 million
Le Petit Prince (The Little Prince)	Antoine de Saint-Exupéry	French	1943	140 million

Figure 3-7. Content updated on pressing Enter

Editable cell

First thing to do is set up a simple event handler. On double-click, the component “remembers” the selected cell.

```
React.DOM.tbody({onDoubleClick: self._showEditor}, ...)
```



Prior to this line, at the top of the render() function, there’s the famous:

```
var self = this;
```

- i. which helps reduce the amount of .bind(this) added to all callback functions. Later you’ll see how React (via JSX) helps you solve this workaround and still have good-looking functions.

Let’s see how _showEditor looks like:

```
_showEditor: function(e) {
  this.setState({edit: {
    row: parseInt(e.target.dataset.row, 10),
    cell: e.target.cellIndex
  }});
},
```

What’s happening here?

- The function sets the edit property of this.state. This property is null when there’s no editing going on and then turns into an object with properties row and cell which contain the row index and the cell index of the cell being edited. So if you double-click the very first cell, this.state.edit gets the value {row: 0, cell: 0}

- To figure out the cell index you use the same `e.target.cellIndex` as before, where `e.target` is the `<td>` that was double-clicked
- There's no `rowIndex` coming for free in the DOM, so you need to do-it-yourself via a `data-` attribute. Each cell should have a `data-row` attribute with the row index which you can `parseInt()` to get the index back.

A few more clarifications and prerequisites...

The `edit` property didn't exist before and should also be initialized in the `getInitialState()` method, which now should look like so:

```
getInitialState: function() {
  return {
    data: this.props.initialData,
    sortBy: null,
    descending: false,
    edit: null, // {row: index, cell: index}
  };
},
```

The property `data-row` is something you need so you can keep track of row and cell indexes. Let's see how the whole `tbody()` construction looks like:

```
React.DOM.tbody({onDoubleClick: self._showEditor},
  self.state.data.map(function (row, rowidx) {
    return (
      React.DOM.tr({key: rowidx},
        row.map(function (cell, idx) {
          var content = cell;

          // TODO - turn `content` into an input
          // if the `idx` and the `rowidx` match the one being edited
          // otherwise just show the text content

          return React.DOM.td({
            key: idx,
            'data-row': rowidx
          }, content);
        })
      )
    );
  })
);
```

Now the last thing left to do is to do what the `TODO` says and make an input field. The whole `render()` function is called again just because of the `setState()` call that sets the `edit` property. React will rerender the table which gives you chance to update the table cell that was double-clicked.

Input field cell

Remembering the edit state:

```
var edit = self.state.edit;
```

Check if the edit is set and if so, is this the exact cell being edited:

```
if (edit && edit.row === rowidx && edit.cell === idx) {  
  // ...  
}
```

If this is the target cell, let's make an input field with the content of the cell:

```
var content = React.DOM.form({onSubmit: self._save},  
  React.DOM.input({  
    type: 'text',  
    defaultValue: content  
  })  
);
```

As you see it's a form with a single input and the input is pre-filled with the text of the cell. Submitting the form will be trapped in the private `_save()` method.

Saving

Last piece of the editing puzzle is saving the content changes after the person is done typing and they have submitted the form (via the Enter button).

```
_save: function(e) {  
  e.preventDefault();  
  // ... do the save  
},
```

After preventing the default behavior (so the page doesn't reload), you need to get a reference to the input field:

```
var input = e.target.firstChild;
```

Clone the data, so you don't manipulate `this.state` directly:

```
var data = this.state.data.slice();
```

Update the piece of data given the new value and the cell and row indices stored in the `edit` property of the state:

```
data[this.state.edit.row][this.state.edit.cell] = input.value;
```

Finally, set the state, which causes rerendering of the UI

```
this.setState({  
  edit: null, // done editing  
  data: data  
});
```

Conclusion and virtual DOM diffs

At this point the editing feature is complete. It didn't take too much code. All you needed was to:

- keep track of which cell to edit via `this.state.edit`
- render an input field when displaying the table if the row and cell indices match the cell the person double-clicked
- update the data array with the new value from the input field

As soon as you `setState()` with the new data, React calls the component's `render()` method and the UI magically updates. It may look like it won't be particularly efficient to render the whole table for just one cell's content change. And in fact, React updates only a single cell.

If you open your browser's dev tools you can see which parts of the DOM tree are updated as you interact with your application. On [Figure 3-8](#) you can see the dev tools highlighting the DOM change after changing "Lord of the ring"'s language from English to English.

Book	Author	Language	Published	Sales
The Lord of the Rings	J. R. R. Tolkien	English	1954-1955	150 million
Le Petit Prince (The Little Prince)	Antoine de Saint-Exupéry	French	1943	140 million
Harry Potter and the Philosopher's Stone	J. K. Rowling	English	1997	107 million
And Then There Were None	Agatha Christie	English	1939	100 million
Dream of the Red Chamber	Cao Xueqin	Chinese	1754-1791	100 million
The Hobbit	J. R. R. Tolkien	English	1937	100 million
She: A History of Adventure	H. Rider Haggard	English	1887	100 million

Elements Network Sources Timeline Profiles Resources Audits Console React

```
<!DOCTYPE html>
<html>
  <head>...</head>
  <body>
    <div id="app">
      <table data-reactid=".0">
        <thead data-reactid=".0.0">...</thead>
        <tbody data-reactid=".0.1">
          <tr data-reactid=".0.1.0">
            <td data-row="0" data-reactid=".0.1.$0">The Lord of the Rings</td>
            <td data-row="0" data-reactid=".0.1.$0.1">J. R. R. Tolkien</td>
            <td data-row="0" data-reactid=".0.1.$0.2">English</td>
            <td data-row="0" data-reactid=".0.1.$0.3">1954-1955</td>
            <td data-row="0" data-reactid=".0.1.$0.4">150 million</td>
          </tr>
          <tr data-reactid=".0.1.1">...</tr>
          <tr data-reactid=".0.1.2">...</tr>
          <tr data-reactid=".0.1.3">...</tr>
        </tbody>
      </table>
    </div>
  </body>
</html>
```

Figure 3-8. Highlighting DOM changes

Behind the scenes React calls your `render()` and creates a lightweight tree representation of the desired DOM result. This is known as a *virtual DOM tree*. When the `render()` method is called again (after a call to `setState()` for example) React takes the virtual tree before and after and computes a diff. Based on this diff, React figures out the minimum required DOM operations (e.g. `appendChild()`, `textContent`, etc) to carry on that change into the browser's DOM.

In [Figure 3-8](#) for example, there is only one change required to the cell and it's not necessary to rerender the whole table. By computing the minimum set of changes and batching DOM operations, React “touches” the DOM lightly as it's a known problem that DOM operations are slow (compared to pure JavaScript operations, function calls, etc) and are often the bottleneck in rich web applications' rendering performance.

Long story short: React has your back when it comes to performance and updating the UI by: * touching the DOM lightly * using event delegation for user interactions

Search

Next, let's add a search feature to the Excel component that allows people to filter the contents of the table. Here's the plan:

- Add a button to trigger the new feature ([Figure 3-9](#))
- If search is on, add a row of inputs where each one searches in the corresponding column ([Figure 3-10](#))
- As a person types in an input box, filter the array of `state.data` to only show the matching content ([Figure 3-11](#))



Figure 3-9. Search button

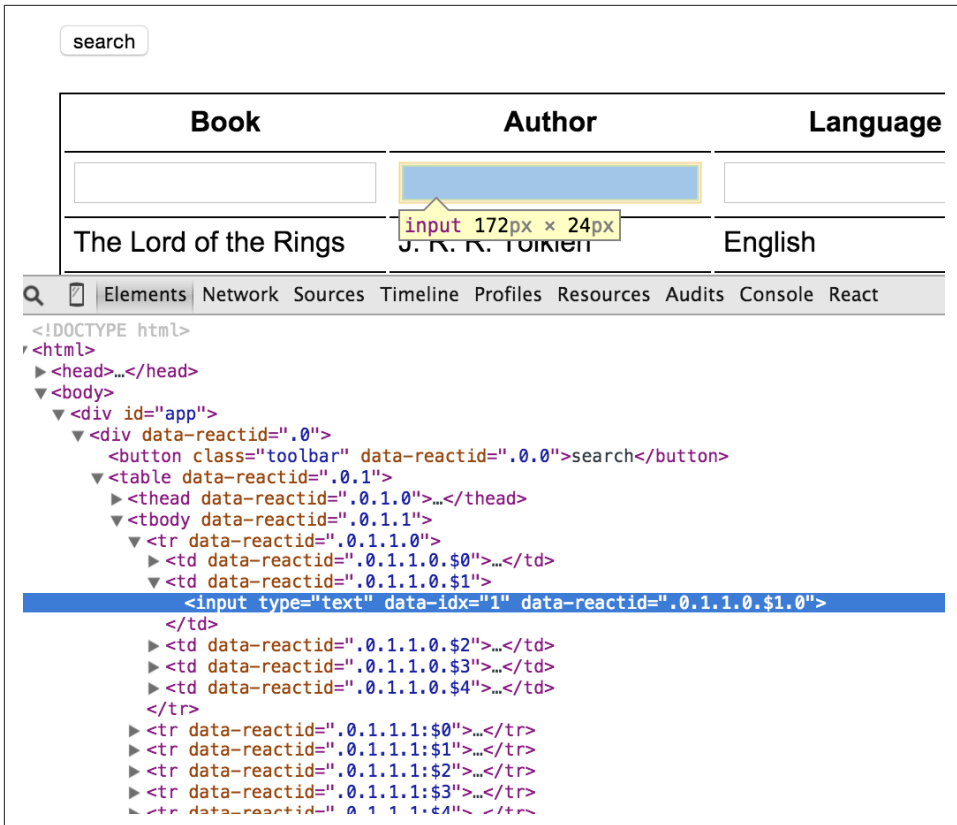


Figure 3-10. Row of search/filter inputs

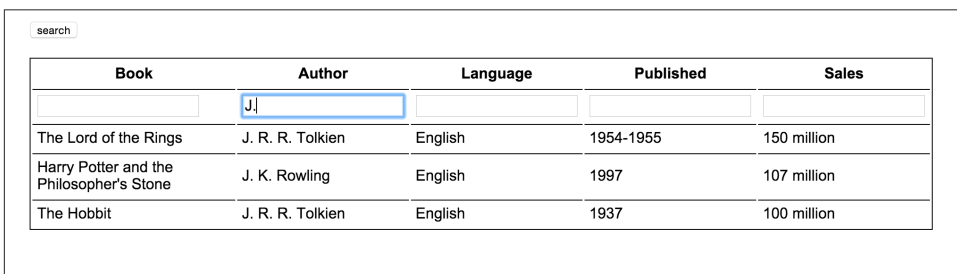


Figure 3-11. Search results

State and UI

The first thing to do is add a search property to the `this.state` object to keep track whether the search feature is on:

```

getInitialState: function() {
  return {
    data: this.props.initialData,
    sortBy: null,
    descending: false,
    edit: null, // [row index, cell index],
    search: false
  };
},

```

Next comes updating the UI. To keep things more manageable, let's split up the `render()` function into smaller dedicated chunks. So far the `render()` function was only rendering a table. So let's rename it to `_renderTable()`. Next, the search button will soon be a part of a whole toolbar (you'll be adding an "export" feature), so let's render it as part of a `_renderToolbar()` function.

The result looks like so:

```

render: function() {
  return (
    React.DOM.div(null,
      this._renderToolbar(),
      this._renderTable()
    )
  );
},

_renderToolbar: function() {
  // TODO
},

_renderTable: function() {
  // same as the function formerly known as `render()`
},

```

As you see, the new `render()` function returns a container `div` with two children: the toolbar and the table. You already know what the table rendering looks like, and the toolbar is just a button for now:

```

_renderToolbar: function() {
  return React.DOM.button(
    {
      onClick: this._toggleSearch,
      className: 'toolbar'
    },
    'search'
  );
},

```

If the search is on (meaning `this.state.search` is true), you need a new table row full of inputs. Let's have a `_renderSearch()` function take care of this.

```

_renderSearch: function() {
  if (!this.state.search) {
    return null;
  }
  return (
    React.DOM.tr({onChange: this._search},
      this.props.headers.map(function(_ignore, idx) {
        return React.DOM.td({key: idx},
          React.DOM.input({
            type: 'text',
            'data-idx': idx,
          })
        );
      })
    );
  );
},

```

As you see, if the search feature is not on, the function doesn't need to render anything, so it returns `null`. Another option is, of course, to have the caller of this function make the decision and not call it at all if the search is not on. But the example above helps the already busy `_renderTable()` function be ever so slightly simpler. Here's what `_renderTable()` needs to do:

Before:

```

React.DOM.tbody({onDoubleClick: self._showEditor},
  self.state.data.map(function (row, rowidx) { // ...

```

After:

```

React.DOM.tbody({onDoubleClick: self._showEditor},
  this._renderSearch(),
  self.state.data.map(function (row, rowidx) { // ...

```

So the search inputs are just another child before the main `data` loop (the one that creates all the table rows and cells). When `_renderSearch()` returns `null`, React simply doesn't render anything.

At this point, that's all for the UI updates. Let's take a look at the meat of the feature, the "business logic" if you will: the actual search.

Filtering content

The search feature is going to be fairly simple - take the array of data, call the `Array.prototype.filter()` method on it and return a filtered array with the elements that match the search string.

The UI still uses `this.state.data` to do the rendering, but `this.state.data` will be a reduced version of itself.

You'll need a copy of the data before the search, so that you don't lose the data forever. This allows the person to go back to full table or change the search string to get different matches. Let's call this copy (actually a reference) `_preSearchData`:

```
var Excel = React.createClass({
  // stuff..

  _preSearchData: null,

  // more stuff...
});
```

When the person clicks the “search” button, the `_toggleSearch()` function is onvoked. This function's task is to turn the search feature on and off. It does its task by: * setting the `this.state.search` to true or false accordingly * when enabling the search, “re-membering” the old data * when disabling the search, reverting to the old data

Here's what this function can look like:

```
_toggleSearch: function() {
  if (this.state.search) {
    this.setState({
      data: this._preSearchData,
      search: false
    });
    this._preSearchData = null;
  } else {
    this._preSearchData = this.state.data;
    this.setState({
      search: true
    });
  }
},
```

The last thing to do is implement the `_search()` function which is called every time something in the search row changes, meaning the person is typing in one of the inputs. Here's the complete implementation, followed by some more details:

```
_search: function(e) {
  var needle = e.target.value.toLowerCase();
  if (!needle) {
    this.setState({data: this._preSearchData});
    return;
  }
  var idx = e.target.dataset.idx;
  var searchdata = this._preSearchData.filter(function (row) {
    return row[idx].toString().toLowerCase().indexOf(needle) > -1;
  });
  this.setState({data: searchdata});
},
```

You get the search string from the change event's target (which is the input box):

```
var needle = e.target.value.toLowerCase();
```

If there's no search string (person erased what they typed), the function takes the original, cached data and this data becomes the new state:

```
if (!needle) {  
  this.setState({data: this._preSearchData});  
  return;  
}
```

If there is a search string, filter the original data and set the filtered results as the new state of the data.

```
var idx = e.target.dataset.idx;  
var searchdata = this._preSearchData.filter(function (row) {  
  return row[idx].toString().toLowerCase().indexOf(needle) > -1;  
});  
this.setState({data: searchdata});
```

And with this, the search feature is complete. All you needed to do to implements is: * Add search UI * Show/hide it upon request * The actual “business logic” which is a simple array filter.

Nothing in the original table rendering really needed to change. You only worry about the state of your data and let React take care of rendering (and all the grunt DOM work associated) whenever data state changes.

How can you improve the search?

This was a simple working example. But how can you improve the feature? One this to do is have an *additive search* in multiple boxes. Meaning filter the already filtered data. If the person types “Eng” in the language row and then searches using a different search box, why not only search in the search results of the previous search only? How would you implement this feature?

Instant replay

As you know now, your components worry about their state and let React render whenever appropriate. Which means that given the same data, the application will look exactly the same, no matter what changed before of after this particular data state. This gives you a great debugging-in-the-wild opportunity.

When a person using your app encounters a bug, they can click a button to report the bug and they don't need to explain what happened when they can just send you a copy of this.state and this.props and you should be able to recreate the exact application state.

Also an “undo” implementation becomes trivial: you just need to go back to the previous state.

Let's take that idea a bit further, just for fun. Let's record each state change in the Excel component and then replay it. It's fascinating to watch all your actions played in front of you.

In terms of implementation, let's not concern with the question of *when* did the change occur and just “play” the app state changes at 1 second intervals. All you need to do to implement this feature is add a `_logSetState()` method and search/replace all calls to `setState()` with calls to the new function.

So all calls to...

```
this.setState(newSate);
```

i. become

```
this._logSetState(newState);
```

The `_logSetState` needs to do two things: log the new state and then pass it over to `setState()`. Here's one example implementation where you make a deep copy of the state and append it to `this._log`:

```
var Excel = React.createClass({
  _log: [],
  _logSetState: function (newState) {
    // remember the old state in a clone
    if (this._log.length === 0) {
      this._log.push(JSON.parse(JSON.stringify(this.state)));
    }
    this._log.push(JSON.parse(JSON.stringify(newState)));
    this.setState(newState);
  },
  // ....
});
```

Now that all state changes are logged, let's play them back. To trigger the playback let's add a simple event listener that captures keyboard actions and invokes the `_replay()` function.

```
componentDidMount: function() {
  document.onkeydown = function(e) {
    if (e.altKey && e.shiftKey && e.which === 82) { // ALT+SHIFT+R(eplay)
      this._replay();
    }
  }.bind(this);
},
```

Finally the `_replay()`. It uses `setInterval()` and once a second it reads the next object from the log and passes it to `setState()`.

```
_replay: function() {  
  if (this._log.length === 0) {  
    console.warn('No state to replay yet');  
    return;  
  }  
  var idx = -1;  
  var interval = setInterval(function() {  
    idx++;  
    if (idx === this._log.length - 1) { // the end  
      clearInterval(interval);  
    }  
    this.setState(this._log[idx]);  
  }.bind(this), 1000);  
},
```

How can you improve the replay?

How about implementing an Undo/Redo feature? Say when the person uses ALT+Z keyboard combination, you go back one step in the state log and on ALT+SHIFT+Z you go forward.

Download the table data

After all the sorting, editing and searching, the person is finally happy with the state of the data in the table. It would be nice to be able to download the result of all the labour.

Luckily, there's nothing easier in React. All you need to do is grab the current `this.state.data` and give it back - in JSON or CSV format.

Figure 3-12 shows the end result when a person clicks “Export CSV”, downloads the file called `data.csv` (see the bottom left of the browser window) and opens this file in MS Excel.

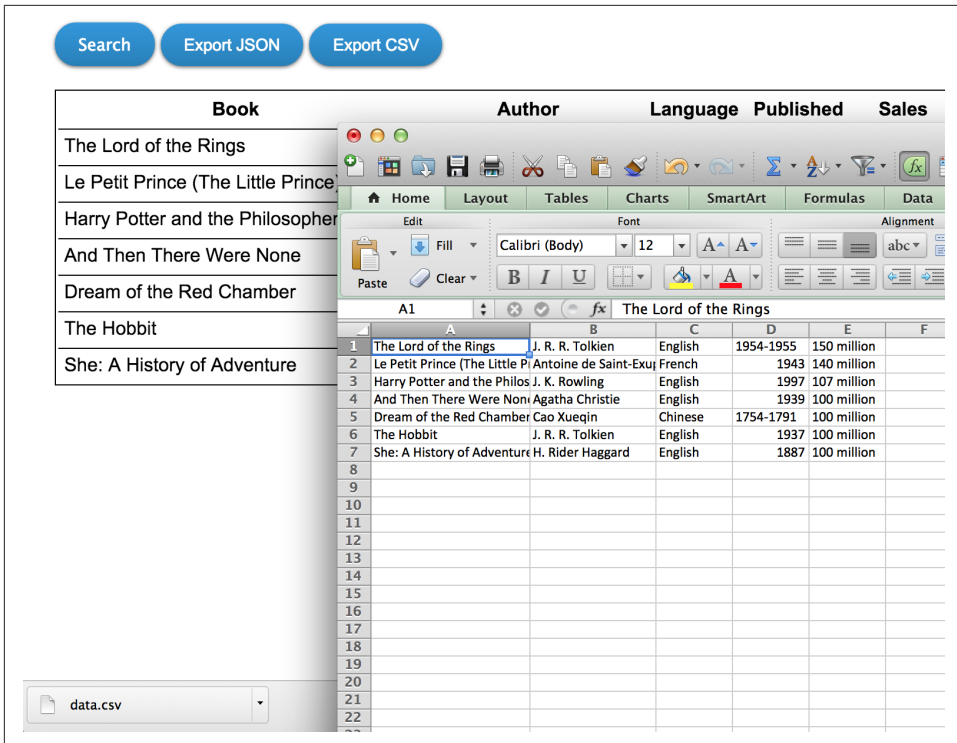


Figure 3-12. Export table data to MS Excel via CSV

First thing is to add new options to the toolbar. Let's use some HTML5 magic that forces `<a>` links to trigger file downloads, so the new "buttons" have to be links disguised as buttons with some CSS

```
_renderToolbar: function() {
  return React.DOM.div({className: 'toolbar'},
    React.DOM.button({
      onClick: this._toggleSearch
    }, 'Search'),
    React.DOM.a({
      onClick: this._download.bind(this, 'json'),
      href: 'data.json'
    }, 'Export JSON'),
    React.DOM.a({
      onClick: this._download.bind(this, 'csv'),
      href: 'data.csv'
    }, 'Export CSV')
  );
},
```

Now for the `_download()` function. While exporting to JSON is trivial, CSV needs a little bit more work. In essence it's just a loop over all rows and all cells in a row and producing a long string. Once this is done, the function initiates the downloads via `download` attribute and the `href` blob created by `window.URL`

```
_download: function (format, ev) {  
  var contents = format === 'json'  
    ? JSON.stringify(this.state.data)  
    : this.state.data.reduce(function(result, row) {  
      return result  
        + row.reduce(function(rowresult, cell, idx) {  
          return rowresult  
            + ' "'  
            + cell.replace(/"/g, '""')  
            + '"'  
            + (idx < row.length - 1 ? ',' : '');  
        }, '')  
        + "\n";  
    }, '');  
  
  var URL = window.URL || window.webkitURL;  
  var blob = new Blob([contents], {type: 'text/' + format});  
  ev.target.href = URL.createObjectURL(blob);  
  ev.target.download = 'data.' + format;  
},
```

So far in the book you've seen how you user interfaces are defined in the `render()` functions using calls to `React.createElement()` and the `React.DOM.*` family, e.g. `React.DOM.span()`. One inconvenience with this many function calls is that it's a little hard to keep up with all these parentheses and curly braces you need to close. There's an easier way - JSX.

JSX is a separate technology from React and completely optional. As you see, the first three chapters didn't even use JSX. You can opt into not using JSX at all. But it's very likely that once you try it, you won't go back to function calls.



It's not quite clear what the acronym JSX stands for, most likely JavaScriptXML or JavaScript (syntax) eXtension. The official home of the open-source project is <http://facebook.github.io/jsx/>

Hello JSX

Let's go back to Chapter 1's last of the "hello world" examples:

```
<script src="react/build/react.js"></script>
<script>
  React.render(
    React.DOM.h1(
      {id: "my-heading"},
      React.DOM.span(null,
        React.DOM.em(null, "Hell"),
        "o"
      ),
      " world!"
    ),
    document.getElementById('app')
```

```
);  
</script>
```

There are quite a few function calls in the `render()` function. Using JSX makes it simpler:

```
React.render(  
  <h1 id="my-heading">  
    <span><em>Hell</em></span> world!  
  </h1>,  
  document.getElementById('app')  
>);
```

This syntax looks just like HTML and you already know HTML. The only thing is, since it's not valid JavaScript syntax, it cannot run in the browser as-is. You need to transform (*transpile*) this code into pure JavaScript that the browser can run.

Transpiling JSX

The process of transpilation is a process of taking source code and rewriting it to work the same but using syntax that's understood by older browsers. It's different than using *polyfills*. An example of a polyfill is adding a method to `Array.prototype` such as `map()` which was introduced in ECMAScript5 and making it work in browsers that support ECMAScript3, like so:

```
if (!Array.prototype.map) {  
  Array.prototype.map = function() {  
    // implement the method  
  };  
}
```

A polyfill is a solution in pure JavaScript-land. It's a good solution when adding new methods to existing objects or implementing new objects (such as JSON). But it's not sufficient when new syntax is introduced into the language. New syntax, such as making the keyword `class` work, is just invalid syntax that throws a parse error in browsers without `class` support and there's no way to polyfill it. New syntax therefore requires a compilation (transpilation) step so it's transformed *before* it's served to the browser.

Transpiling JavaScript is getting more and more common as people want to use features in ECMAScript6 (a.k.a. ECMAScript2015) and beyond and not have to wait for browsers to support them. If you already have a build process set up (that does e.g. minification or ECMAScript6-to-5 transpilation), you can simply add the JSX transformation step to it. But let's assume you don't have a build process and go through the steps of setting up one.

Client-side

The easiest way to get started is to do the transpiling on the client side, in the browser. It's not recommended that you ship applications to real users this way for performance

reasons, but it's easy to get your feet wet or simply to prototype and experiment without setting up a whole build process.

There are two things you can do in your page to make JSX work: * Include the client JSX transpilation script * Markup the scripts tags that use JSX

All the examples in the book so far include React library like so:

```
<script src="react/build/react.js"></script>
```

Now in addition to that you need to include JSXTransformer like so:

```
<script src="react/build/react.js"></script>
<script src="react/build/JSXTransformer.js"></script>
```

This JSXTransformer.js is already in your /react/build/ directory you've setup in Chapter 1.

The second step is to add an invalid type attribute to the <script> tags that require transformation.

Before:

```
<script>
  React.render(/*...*/);
</script>
```

After:

```
<script type="text/jsx">
  React.render(/*...*/);
</script>
```

When you load the page, the JSXTransformer will kick in, find all the text/jsx scripts and transform their content into something the browsers can use.

Figure 4-1 shows what happens in Chrome when you try to run a script with JSX syntax as is. You get a syntax error, just as expected. On Figure 4-2 you can see that the page works fine after the JSXTransformer transpiles the script blocks with type="text/jsx". Finally on Figure 4-3 you can see how the transformer rewrites the JSX into valid JavaScript and auto-inserts it into the <head> of the document. The generated script also includes a source map (a mapping of before/after the transformation) which can prove helpful while debugging.

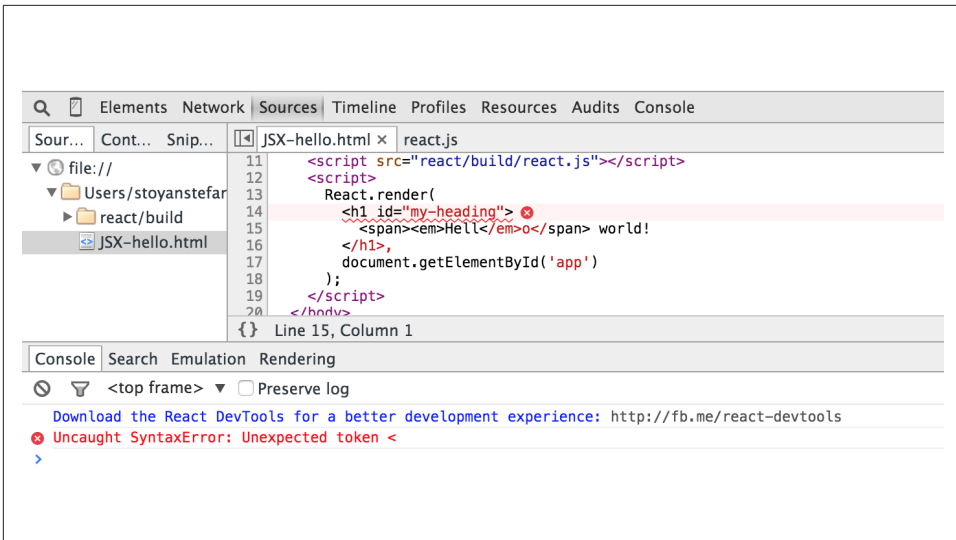


Figure 4-1. Browsers don't understand JSX syntax

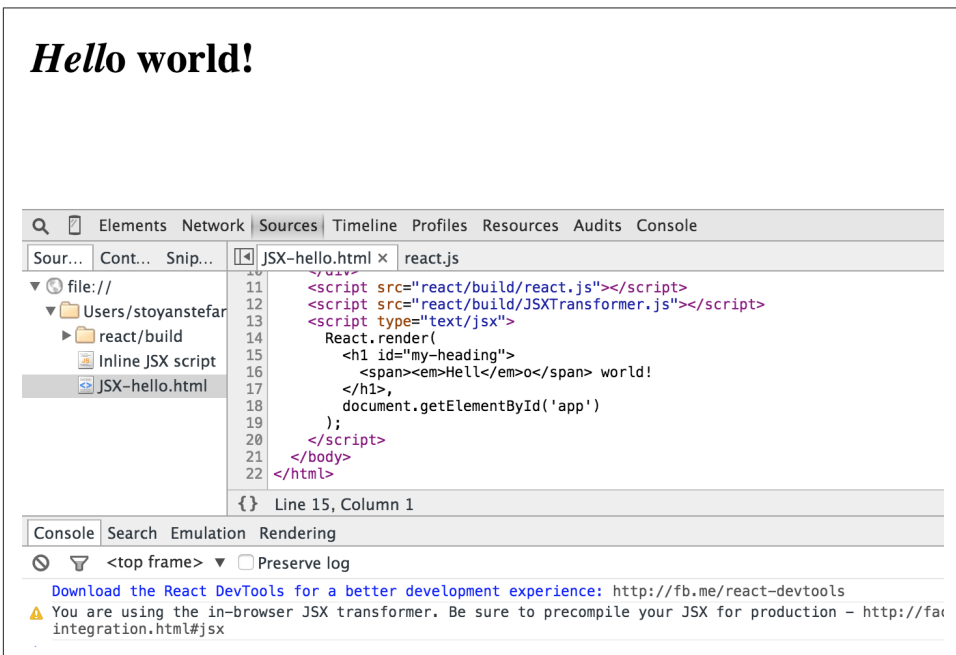


Figure 4-2. JSXTransformer and text/jsx content type

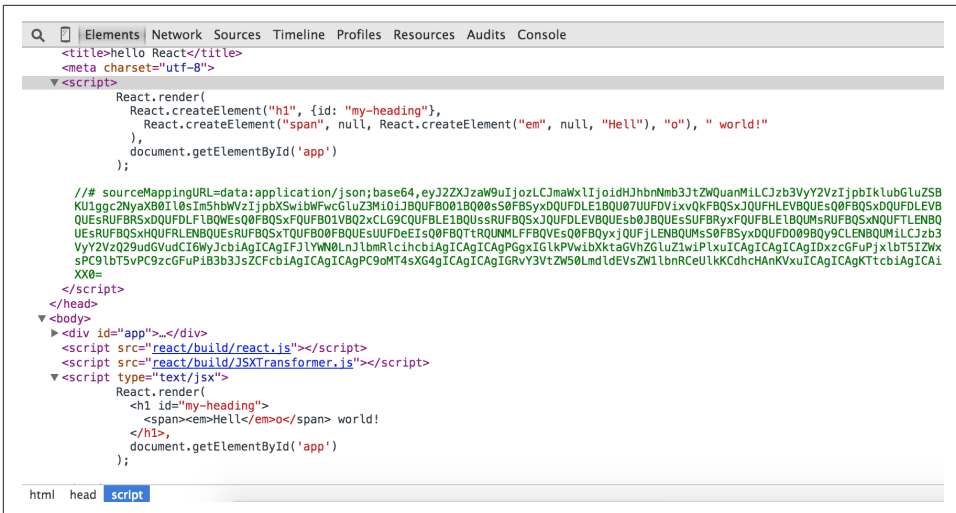


Figure 4-3. Transpiled JSX inserted in the <head>

Working with external files

In the example above you see how JSXTransformer makes inline scripts work. It can do the same with external scripts too. In other words it can make code like this work:

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

In order for this to work though you need to host your application on a server (even development server such as <http://localhost/>). This is required because the JSXTransformer needs to download and transform the external JavaScript via XMLHttpRequest which is subject to same-origin restrictions imposed by the browsers.

Build process

For any serious development and deployment outside of prototyping and testing JSX you need to setup a build process. If you already have one, you just need to add the JSX-to-JS transformation to it.

To get a command-line version of the JSXTransformer you install the `react-tools` NPM package:

```
$ npm install -g react-tools
```

Testing a successful installation:

```
$ jsx -h
```

```
Usage: jsx [options] <source directory> <output directory> ...
```

```
...
```

Now let's recreate the "Hello world" example by setting up for development. Let's have all source files into a `source/` directory and all the results from the transformation into a `build/` directory.

```
$ cd ~/codez/myapp
$ mkdir source
$ mkdir build
$ touch JSX-hello-build.html
$ touch source/hello.js
```

The `.html` file should include the `hello.js` from the `build/` directory.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello JSX React</title>
    <meta charset="utf-8">
  </head>
  <body>
    <div id="app">
      <!-- my app renders here -->
    </div>
    <script src="react/build/react.js"></script>
    <script src="build/hello.js"></script>
  </body>
</html>
```

Setting auto-transformation whenever anything in `source/` changes:

```
$ jsx --watch source/ build/

built Module("hello")
["hello"]
hello.js changed; rebuilding...
built Module("hello")
["hello"]
```

Loading `JSX-hello-build.html` in the browser now reads the transformed `build/hello.js` and works as expected (Figure 4-4)

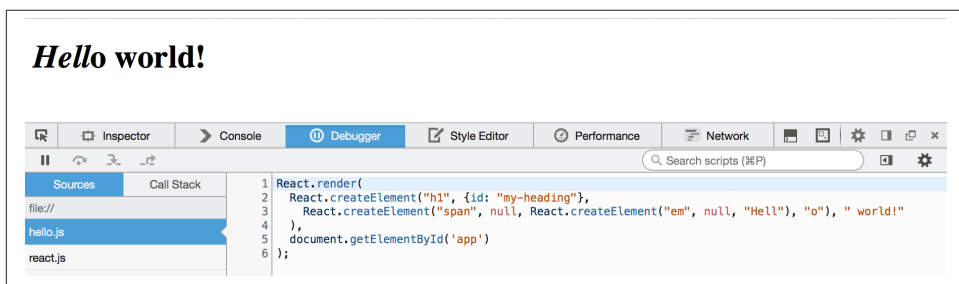


Figure 4-4. Rendering transformed JSX

Babel

This book is written with React v0.13 in mind as it's the latest at the time of writing. However, there's an upcoming change you should know about. Starting with v0.14 the JSXTransformer is deprecated and won't be part of the React releases. It's deprecated in favor of another open-source project called Babel (formerly 6to5). Babel is a transpiler that supports the latest JavaScript features and also includes JSX support.

The switch from JSXTransformer to Babel is relatively painless.

Installing:

```
$ npm install --global babel
```

Transpiling on the fly as soon as you change a file:

```
$ babel source/ --watch --out-dir build/
```

This was previously:

```
$ jsx --watch source/ build/
```

For in-browser transformations, you need the file called `browser.js`, which you can find in your `/node_modules` directory. Copy to your working directory, for example:

```
cp /usr/local/lib/node_modules/babel/node_modules/babel-core/browser.js ~/reactbook/react/babel/br
```

Include the in-browser transformer (replace the `JSXTransformer.js`):

```
<script src="babel/browser.js"></script>
```

And finally, instead of `type="text/jsx"`, you need `text/babel`, so to put it all together:

```
<script src="react/build/react.js"></script>
<script src="babel/browser.js"></script>
<script type="text/babel">
  React.render(
    <h1 id="my-heading">
      <span><em>Hell</em></span> world!
    </h1>,
    document.getElementById('app')
  );
</script>
```

About the JSX transformation

As you can see on [Figure 4-4](#) the JSX source becomes a series of function calls, using the same functional syntax you've seen in the book prior to this chapter. It's just JavaScript so it's easy to read and understand.

The JSX transform is therefore pretty lightweight and simple. It even preserves the line numbers so should an error occur, you can trace back to the offending line in the original JSX source.

To experiment and get familiar with the JSX transforms, you can play with the live editor at <https://facebook.github.io/react/jsx-compiler.html> (Figure 4-5). Note how the transformed code matches the lines of the original.

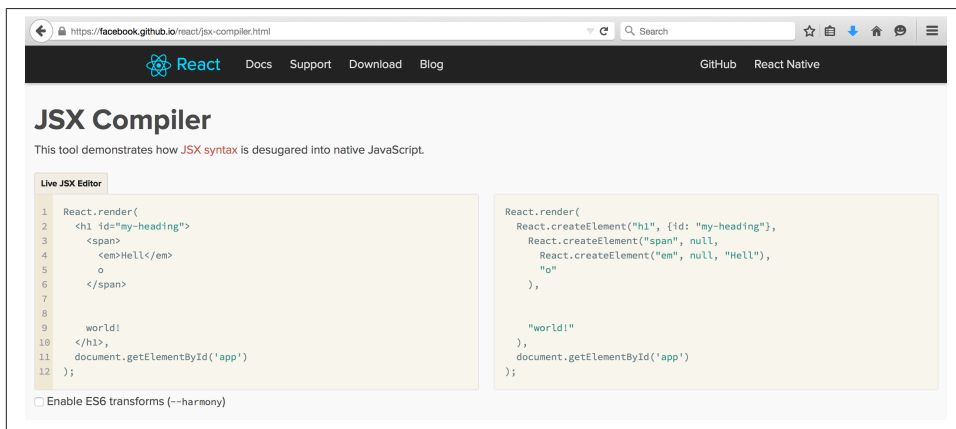


Figure 4-5. Live JSX transformation tool

There's another online tool you may find helpful when learning JSX or transitioning an app's markup from HTML: an HTML-to-JSX transformer (<https://facebook.github.io/react/html-jsx.html>) (Figure 4-6)

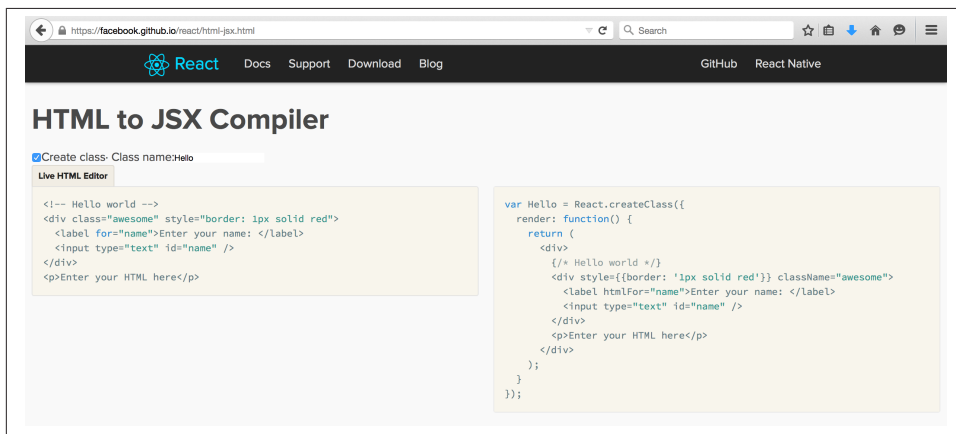


Figure 4-6. HTML-to-JSX tool

JavaScript in JSX

Often when building the UI you need variables, conditions and loops. Instead of making up yet another templating syntax, JSX lets you write JavaScript inside of JSX. All you need to do is wrap your JavaScript code in curly braces.

Take for example one of Excel examples from the last chapter. To replace the functional syntax with JSX, you may end up with something like this:

```
var Excel = React.createClass({

  /* snip... */

  render: function() {
    var state = this.state;
    return (
      <table>
        <thead onClick={this._sort}>
          <tr>
            {this.props.headers.map(function(title, idx) {
              if (state.sortby === idx) {
                title += state.descending ? ' \u2191' : ' \u2193';
              }
              return <th key={idx}>{title}</th>;
            })}
          </tr>
        </thead>
        <tbody>
          {state.data.map(function (row, idx) {
            return (
              <tr key={idx}>
                {row.map(function (cell, idx) {
                  return <td key={idx}>{cell}</td>;
                })}
              </tr>
            );
          })}
        </tbody>
      </table>
    );
  }
});
```

As you can see, to use variables, you do:

```
<th key={idx}>{title}</th>
```

For loops you can wrap `Array#map` calls in curly braces:

```
<tr key={idx}>
  {row.map(function (cell, idx) {
    return <td key={idx}>{cell}</td>;
  })}
</tr>
```

```
    }}}
  </tr>

```

And as you see, you can have JSX in JavaScript in JSX as deeply as you need. You can think of JSX as JavaScript (after a light transformation), but with familiar HTML syntax. Even members of your team who are not as well versed in JavaScript as yourself, but who know HTML, can write JSX. And they can learn just enough JavaScript to use variables and loops to build the UI with live data.

In the Excel example above there is an if-condition in a `map()` callback. Although it's a nested condition, with a little formatting help you can make even that a readable one-liner:

```
return (<th key={idx}>
  {
    state.sortby === idx
      ? state.descending
        ? title + ' \u2191'
        : title + ' \u2193'
      : title
  }
</th>);

```



Notice the repeating `title` in this last example? You can get rid of it:

```
return (<th key={idx}>{title}
  {
    state.sortby === idx
      ? state.descending
        ? ' \u2191'
        : ' \u2193'
      : null
  }
</th>);

```

However in this case you need to modify the sorting function in the example. The sorting function assumes a person clicks on an `<th>` and uses `cellIndex` to figure out which `<th>`. But when you have adjacent `{}` blocks in JSX, you'll get `<span>` tags to differentiate the two. So `<th>{1}{2}</th>` will turn into DOM as if it was `<th><span>1</span><span>2</span></th>`.

Whitespace in JSX

Whitespace in JSX is similar to HTML but not quite.

```
<h1>
  {1} plus {2} is {3}
</h1>

```

i. results in

```
<h1>
  <span>1</span><span> plus </span><span>2</span><span> is </span><span>3</span>
</h1>
```

i. which renders as “1 plus 2 is 3”, exactly as you’d expect in HTML. Multiple spaces become one.

However in this example:

```
<h1>
  {1}
  plus
  {2}
  is
  {3}
</h1>
```

i. you end up with

```
<h1>
  <span>1</span><span>plus</span><span>2</span><span>is</span><span>3</span>
</h1>
```

As you can see all the whitespace is trimmed, so and the end result is “1plus2is3”.

As a workaround, you can always add space where you need it with { ' ' } (which will produce more `<span>`s) or make the literal strings into expressions and add the space there. In other words any of these will work:

```
<h1>
  { /* space expressions */ }
  {1}
  { ' ' }plus{ ' ' }
  {2}
  { ' ' }is{ ' ' }
  {3}
</h1>

<h1>
  { /* space glued to string expressions */ }
  {1}
  { ' plus ' }
  {2}
  { ' is ' }
  {3}
</h1>
```

Comments in JSX

In the examples above you can see how a new concept sneaked in - adding comments to JSX markup.

Since the `{}` expressions are just JavaScript, you can easily add multi-line comments using `/* comment */`. You can also add single-line comments using `// comment`, however you have to make sure the closing `}` of the expression is on a separate line than `//` so it's not considered part of the comment.

```
<h1>
  { /* multi line comment */ }
  { /*
    multi
    line
    comment
    */ }
  {
    // single line
  }
  Hello
</h1>
```

Since `{// comment}` is not working (`}` is now commented out), there's little benefit of using single-line comments, so you can keep your comments consistent and always use multi-line comments.

HTML entities

You can use HTML entities in JSX like so:

```
<h2>
  More info &raquo;
</h2>
```

This examples produces a “right-angle quote” as shown on [Figure 4-7](#)



Figure 4-7. HTML entity in JSX

However if you use the entity as part of an expression, you will run into double-encoding issues. In this example...

```
<h2>
  {"More info &raquo;}
</h2>
```

- i. the HTML gets encoded and you see the result in [Figure 4-8](#)



More info »

Figure 4-8. Double-encoded HTML entity

To prevent the double-encoding issue, you can use the unicode version of the HTML entity which in this case is `\u00bb` (see <http://dev.w3.org/html5/html-author/charref>).

```
<h2>
  {"More info \u00bb"}
</h2>
```

For convenience you can define a constant-like variable somewhere at the top of your module, together with any common spacing, e.g.

```
var RAQUO = ' \u00bb';
```

Then use the convenient variable like:

```
<h2>
  {"More info" + RAQUO}
</h2>
```

Anti-XSS

You may be wondering why do you have to jump through hoops to use HTML entities. There's a good reason that outweighs the drawbacks and the reason is *XSS prevention*.

React escapes all strings in order to prevent a class of XSS (*Cross-site scripting*) attacks. So when you ask the user to give you some input and they provide a malicious string, React protects you. Take this user input for example:

```
var firstname = 'John<scr'+<script> src="http://evil.co.js"></scr'+<script>';
```

If you naively write this input into the DOM, like:

```
document.getElementById('app').innerHTML = firstname;
```

- i. then this is a disaster because the page will say “John”, but the `<script>` tag will load a malicious JavaScript and compromise your app.

React protects you from cases like this out of the box. If you do:

```
React.render(  
  <h2>  
    Hello {firstname}!  
  </h2>,  
  document.getElementById('app')  
)
```

- i. then React escapes the content of `firstname` and the page displays “Hello John<script src=“http://evil.co.js”></script>!”

Spread attributes

JSX borrows a useful feature from ECMAScript6, called the *spread operator* and adopts it as a convenience when defining properties. Let’s consider an example.

Imagine you have a collection of attributes you want to pass to an `<a>` component:

```
var attr = {  
  href: 'http://example.org',  
  target: '_blank'  
};
```

You can always do it like so:

```
return (  
  <a  
    href={attr.href}  
    target={attr.target}>  
    Hello  
  </a>  
)
```

But this feels like a lot of boilerplate code. Using *spread attributes* you do the same in just one line:

```
return <a {...attr}>Hello</a>;
```

In the example above you have an object of attributes you want to define (maybe conditionally) ahead of time. This is useful in itself, but a more common use is when you get this object of attributes from outside - often from a parent component. Let’s see how that case plays out.

Parent-to-child spread attributes

Imagine you’re building a `FancyLink` component which uses a regular `<a>` behind the scenes. You want your component to accept all the attributes that `<a>` does (`href`, `style`, `target`, etc) plus some more (say `size`). So people can use your component like so:

```

<FancyLink
  href="http://example.org"
  style={{color: "red"}}
  target="_blank"
  size="medium">
  Hello
</FancyLink>

```

How can your `render()` function take advantage of spread attributes and avoid redefining all `<a>`'s properties?

```

var FancyLink = React.createClass({
  render: function() {

    switch(this.props.size) {
      // do something based on the `size` prop
    }

    return <a {...this.props}>{this.props.children}</a>;
  }
});

```



Did you notice the use of `this.props.children`? This is a simple and convenient method to compose any number of children passed over to your component.

In the snippet above you do your custom work based on the value of the `size` property, then simply carry over all the properties to `<a>`. This includes the `size` property. `React.DOM.a` has no concept of `size`, so it silently ignores it while using all the other properties.

You can do a little better by not passing properties that cannot be used by doing something like:

```

var FancyLink = React.createClass({
  render: function() {

    switch(this.props.size) {
      // do something based on the `size` prop
    }

    var attribs = Object.assign({}, this.props); // clone
    delete attribs.size;

    return <a {...attribs}>{this.props.children}</a>;
  }
});

```



Using ECMAScript7-proposed syntax this can be even easier without any cloning:

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

    var {size, ...attrs} = this.props;

    switch (size) {
      // do something based on the `size` prop
    }

    return <a {...attrs}>{this.props.children}</a>;
  }
});
```

Returning multiple nodes in JSX

You always have to return a single node from your `render()` function. Returning two nodes is not allowed. In other words this is an error:

```
var Example = React.createClass({
  render: function() {
    return (
      <span>
        Hello
      </span>
      <span>
        World
      </span>
    );
  }
});
```

The fix is easy - just wrap in another component, say a `<div>`:

```
var Example = React.createClass({
  render: function() {
    return (
      <div>
        <span>
          Hello
        </span>
        <span>
          World
        </span>
      </div>
    );
  }
});
```

While you cannot return an array of nodes from your `render()` function, you can use arrays during composition, as long as the nodes in the array have proper key attributes:

```

var Example = React.createClass({
  render: function() {

    var greeting = [
      <span key="greet">Hello</span>,
      ' ',
      <span key="world">World</span>,
      '!'
    ];

    return (
      <div>
        {greeting}
      </div>
    );
  }
});

```

Notice how you can also sneak in whitespace and other strings in the array (these don't need a key).

In a way this is similar to accepting any number of children passed from the parent and propagating them over in your `render()` function:

```

var Example = React.createClass({
  render: function() {
    console.log(this.props.children.length); // 4
    return (
      <div>
        {this.props.children}
      </div>
    );
  }
});

React.render(
  <Example>
    <span key="greet">Hello</span>
    { ' ' }
    <span key="world">World</span>
    !
  </Example>,
  document.getElementById('app')
);

```

JSX vs HTML differences

As you see JSX looks very familiar. It's just like HTML with the benefit of an easy way to add dynamic values, loops and conditions (just wrap wrap them in `{}`). To start with JSX, you can always use the HTML-to-JSX tool (<https://facebook.github.io/react/html-jsx.html>) but the sooner you start typing your very own JSX, the better. Let's consider

the few differences between HTML and JSX which may surprise you at the beginning as you're learning.

Some of these differences were described in Chapter 1, but let's quickly go over them again.

No `class`, what for?

Instead of the `class` and `for` attributes (both reserved words in ECMAScript), you need to use `className` and `htmlFor`

```
// NO-NO!  
var em = <em class="important" />;  
var label = <label for="thatInput" />;  
  
// OK  
var em = <em className="important" />;  
var label = <label htmlFor="thatInput" />;
```

`style` is an object

The `style` attribute takes an object value, not a `;`-separated string. And the names of the CSS properties are `camelCase`, not dash-delimited.

```
NO-NO!  
var em = <em style="font-size: 2em; line-height: 1.6" />;  
  
// OK  
var styles = {  
  fontSize: '2em',  
  lineHeight: '1.6'  
};  
var em = <em style={styles} />;  
  
// inline is also OK  
// note the double {{}} - one for dynamic value in JSX, one for JS object  
var em = <em style={{fontSize: '2em', lineHeight: '1.6'}} />;
```

Closing tags

In HTML some tags don't need to be closed, in JSX (XML) they do.

```
// NO-NO, unclosed tags, though fine in HTML  
var gimmeabreak = <br>;  
var list = <ul><li>item</ul>;  
var meta = <meta charset="utf-8">;  
  
// OK  
var gimmeabreak = <br />;  
var list = <ul><li>item</li></ul>;
```

```
var meta = <meta charSet="utf-8" />;

// or
var meta = <meta charSet="utf-8"></meta>;
```

camelCase attributes

Did you spot the `charSet` vs `charSet` in the snippet above? All attributes in JSX need to be `camelCase`. That means in the beginning a common source of confusion may be that you're typing `onClick` and nothing happens until you go back and change it to `onClick`.

```
// NO-NO!
var a = <a onclick="reticulateSplines()" />;

// OK
var a = <a onClick={reticulateSplines} />;
```

An exception to this rule are all `data-` and `aria-` prefixed attributes, these are just like in HTML.

JSX and forms

There are some differences in JSX vs HTML when working with forms. Let's take a look.

onChange handler

When using form elements, the user changes values when interacting with them. You subscribe to such changes with `onChange` attribute. This is much more consistent than using `checked` value for radio buttons and checkboxes, and `selected` in `<select>` options. Also when typing in textareas and `<input type="text">` fields, `onChange` fires as the user types which is much more useful than firing when the element loses focus. This means no more subscribing to all sorts of mouse and keyboard events just to monitor typing changes.

value vs defaultValue

In HTML if you have an `<input id="i" value="hello" />` and then change the value by typing "bye", then...

```
i.value; // "bye"
i.getAttribute('value'); // "hello"
```

In React, the `value` property will always have the up-to-date content of the text input. If you want to specify a default, you can use `defaultValue`.

In the following snippet you have an `<input>` component with a pre-filled “hello” content and `onChange` handler. Deleting the last “o” in “hello” will result in value being “hell” and `defaultValue` remaining “hello”.

```
function log(event) {
  console.log(event.target.value); // "hell"
  console.log(event.target.defaultValue); // "hello"
}
React.render(
  <input defaultValue="hello" onChange={log} />,
  document.getElementById('app')
);
```



This is a pattern you should use in your own components: if you accept a property that hints that it should be up-to-date (e.g. `value`, `data`), then keep it current. If not, call it `initialData` (as you saw in Chapter 3) or `defaultValue` or similar to keep the expectations straight.

`<textarea>` value

For consistency with text inputs, React’s version of `<textarea>` takes `value` and `defaultValue` properties. It keeps `value` up-to-date while `defaultValue` remains the original. If you go HTML-style and use a child of the `textarea` as `value` (not recommended), it will be treated as if it was a `defaultValue`.

The whole reason HTML `<textarea>`’s take a child as their `value` is so that developers can use new lines in the input. React, being all JavaScript, doesn’t suffer from this limitation. When you need a new line, you just use ``\\n``.

Consider the following examples and their results shown on [Figure 4-9](#)

```
function log(event) {
  console.log(event.target.value);
  console.log(event.target.defaultValue);
}

React.render(
  <textarea defaultValue="hello\\nworld" onChange={log} />,
  document.getElementById('app1')
);
React.render(
  <textarea defaultValue={"hello\\nworld"} onChange={log} />,
  document.getElementById('app2')
);
React.render(
  <textarea onChange={log}>hello
```

```

world
</textarea>,
document.getElementById('app3')
);
React.render(
  <textarea onChange={log}>{"hello\n\
world"}
</textarea>,
  document.getElementById('app4')
);

```

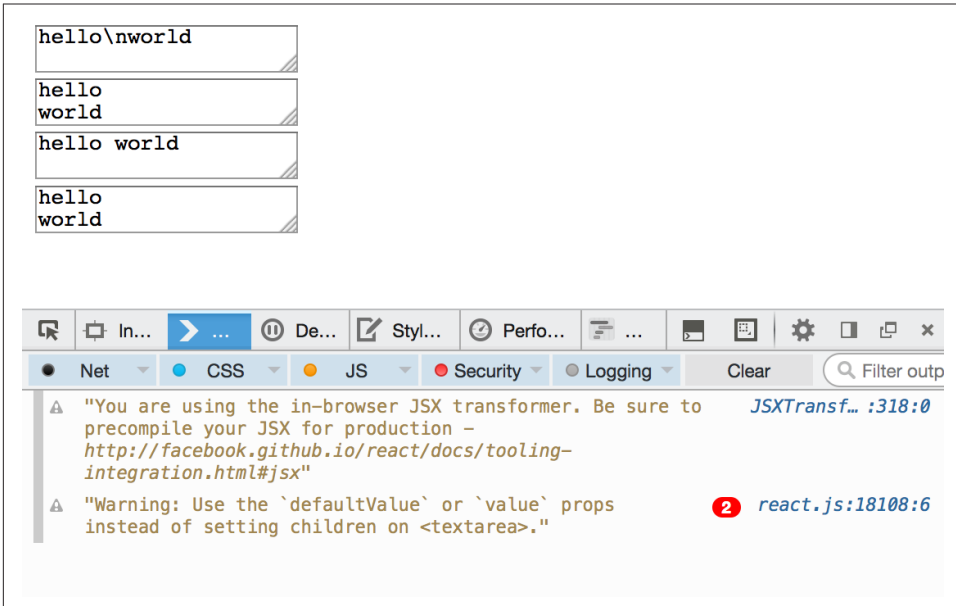


Figure 4-9. New lines in textareas

Note the differences between using a literal string "hello\nworld" as a property value vs. using the JavaScript string {"hello\nworld"}. Also note how a multiline string in JavaScript needs to be escaped with a \ (4th example).

<select> value

When you use a <select> input in HTML, you specify pre-selected entires using <option selected>, like so:

```

<!-- old school HTML -->
<select>
  <option value="stay">Should I stay</option>
  <option value="move" selected>or should I go</option>
</select>

```

In React, you specify value, or better yet, defaultValue on the <select> element.

```
// React/JSX
<select defaultValue="move">
  <option value="stay">Should I stay</option>
  <option value="move">or should I go</option>
</select>
```

Same applies when you have multi-select, only you provide an array of pre-selected values:

```
<select defaultValue={['stay', 'move']} multiple={true}>
  <option value="stay">Should I stay</option>
  <option value="move">or should I go</option>
  <option value="trouble">If I stay it will be trouble</option>
</select>
```



React will warn you if you get mixed up and set the selected attribute of an <option>.

Using <select value> instead of <select defaultValue> is also allowed, although not recommended, as it requires you to take care of updating the value that the user sees. Otherwise when the user selects a different option, the <select> stays the same. In other words you need something like:

```
var MySelect = React.createClass({
  getInitialState: function() {
    return {value: 'move'};
  },
  _onChange: function(event) {
    this.setState({value: event.target.value});
  },
  render: function() {
    return (
      <select value={this.state.value} onChange={this._onChange}>
        <option value="stay">Should I stay</option>
        <option value="move">or should I go</option>
        <option value="trouble">If I stay it will be trouble</option>
      </select>
    );
  }
});
```