

Learning React

...and Redux!



Course overview

1. Welcome

1. Who am I
2. Course structure
3. The Web platform
4. Approaching the web
5. The JavaScript language
6. Course resources

2. Introducing React

1. Templating
2. Enter React
3. Expressing the UI
4. JSX Basics
5. Installing JSX
6. Updating the UI
7. Nested components
8. Connecting to the DOM
9. React devtools
10. Exercise 1

3. Rendering lessons

1. Faulty render return
2. Close all elements
3. Single VS double braces
4. Reserved names
5. Translation issues
6. Spacing trick
7. Conditional render
8. Component children
9. Component props VS Element props
10. Rendering a list
11. Rendering HTML strings
12. Exercise 2

4. React level 2

1. Class syntax
2. Stateful components
3. Event handling
4. Default properties
5. Lifecycle methods
6. Refs
7. Forms
8. Communication
9. Styling
10. Exercise 3

5. Introducing Redux

1. What Redux is
2. State and actions
3. The reducer
4. The store
5. Action creators
6. Redux resources

6. A Redux example

1. App idea
2. State shape
3. Initial state
4. Action shapes
5. Reducer
6. Store
7. Action creators
8. UI
9. UI updater
10. Interaction
11. Tying it together
12. Exercise 4

7. React and Redux

1. Presenting the app
2. Redux parts
3. React parts
4. Vanilla integration
5. Examining React-Redux
6. React-Redux integration
7. Exercise 5

8. Redux Thunk

1. Store enhancers
2. Middlewares
3. The thunk problem
4. The thunk solution
5. Trying it out

9. React Router

1. The need for navigation
2. Exploring an example
3. Dynamic Routing
4. Route Matching Components
5. Navigation Components
6. Parameters
7. Routers
8. Leveling Up

10. Redux level 2

1. Redux devtools
2. Combined reducers
3. Caching with Reselect
4. Where to put logic
5. Flavours of state

11. Advanced React

1. setState take 2
2. The React Context API
3. Context in React-Redux
4. Firebase
5. Recompose
6. Isomorphic apps
7. Performance
8. Working with the DOM

12. Appendix - ES6 features

1. Versatile object definitions
2. Destructuring and rest
3. Versatile function definitions
4. Spreads
5. Modules
6. Classes
7. Decorators
8. Miscellaneous
9. Exercise - trying it out

13. Appendix - Graveyard

- 1..createClass syntax
2. Declaring props
3. Lifecycle methods
4. React Router v2
5. Mixins

Welcome

to the jungle

Sections in this chapter:

1. Who am I
2. Course structure
3. The Web platform
4. Approaching the web
5. The JavaScript language
6. Course resources

1-1. Who am I

Know thy enemy

Let's start with the most important part - me!

1-1-1

Stephen Lau

1-1-2

- Background in driver and graphics programming for the mobile industry.
- Have spent the last few years working at Edument.
- Cloud Computing and Frontend Projects for large customers
- Lots of travel to China as well as teaching and consulting in Sweden and Germany.

Contact information:

1-1-3

- Email: stephen@edument.se
- Twitter: [lau_stephen](https://twitter.com/lau_stephen)

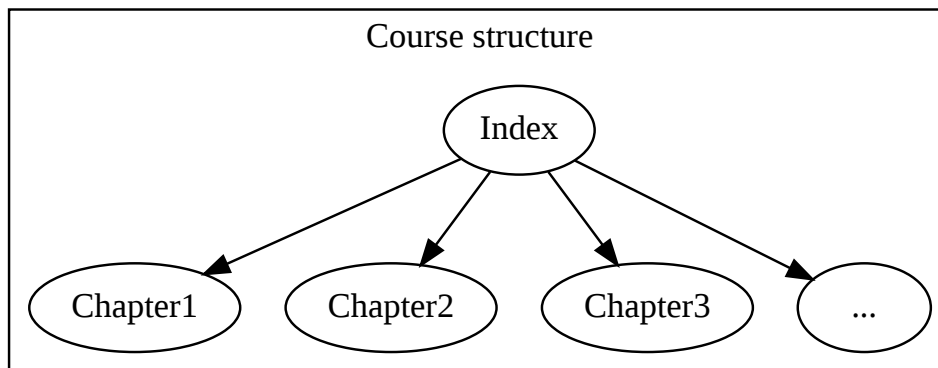
Don't be a stranger! :)

1-2. Course structure

How we'll go about things

The course is divided into several **chapters**, where you're currently in the first.

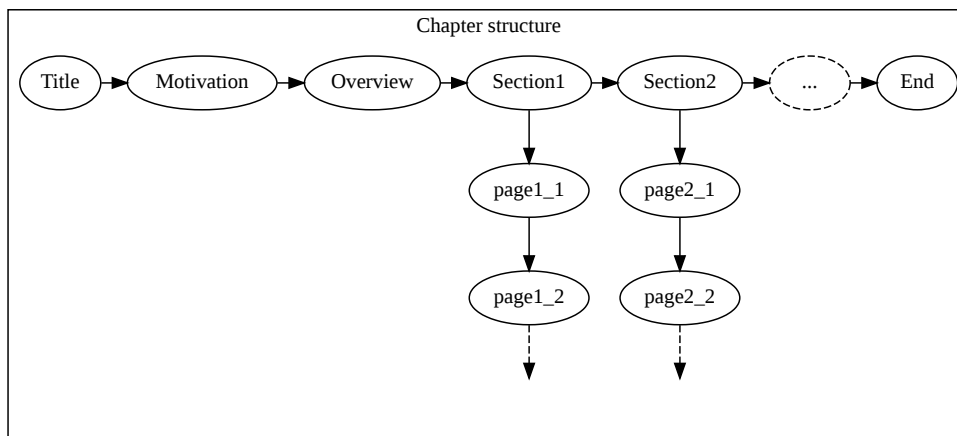
1-2-1



We access the chapters from an **index**, giving you a **birds-eye view** of the entire contents.

Each **chapter** has many **sections**:

1-2-2



In the **printed material** (or PDF) and in the **presentation top-right corner**, the slides are numbered **X-Y-Z** where...

1-2-3

- X is the number of the current **chapter**
- Y is the number of the section **section** within that chapter
- Z is the number of the **slide** within that section

Some sections are **exercises**, where you'll get to internalise what we've been talking about through a **practical task**.

1-2-4

These are all **tied together** in that it is one single app that you will keep tinkering with.

The latest bunch of chapters are **Appendixes**, meant to be used as references as needed.

1-2-5

For example, if you're not comfortable with **new language features** you can check out the **ES6 appendix**!

Finally - it is **forbidden to keep a question to yourself**. Ask away!

1-2-6

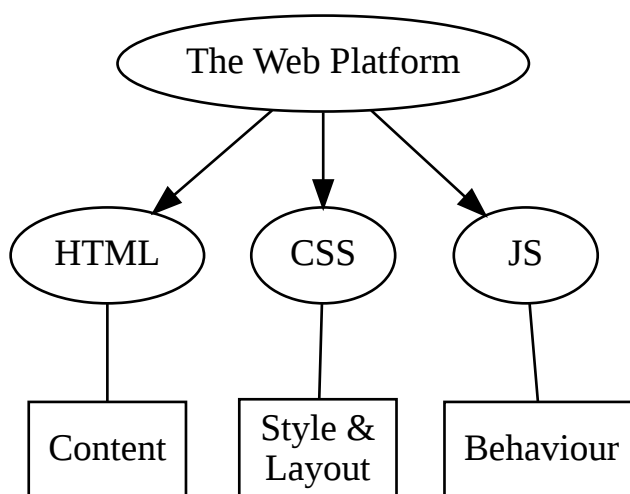
Because, **there are no stupid questions!** ~~Only stupid people.~~

1-3. The Web platform

a.k.a. the holy trinity

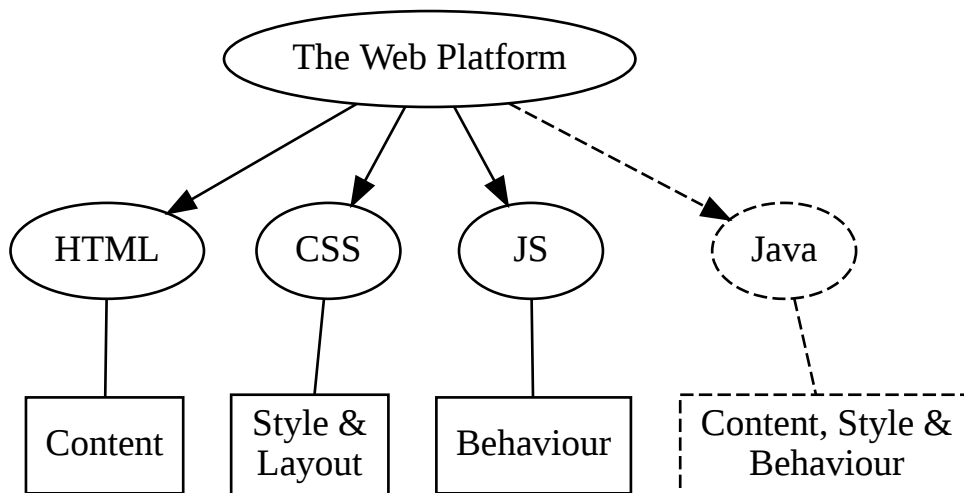
The **Web platform** is really the combination of three separate technologies:

1-3-1



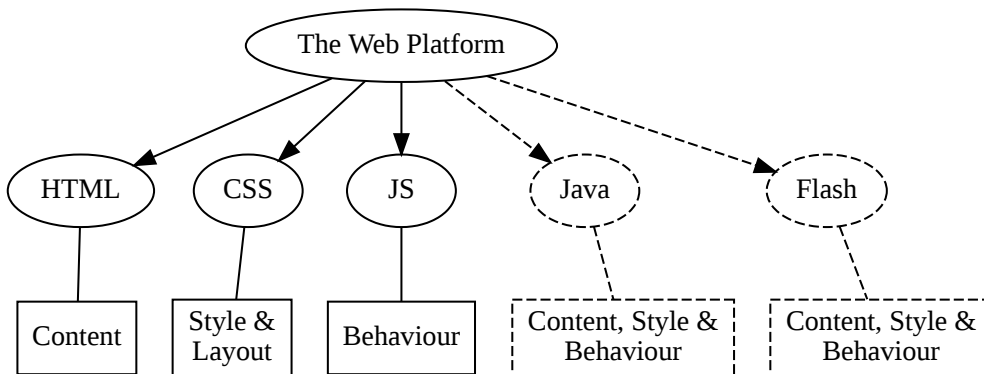
(There was also a dark time when **Java applets** were used in webpages, but we don't talk about that...)

1-3-2



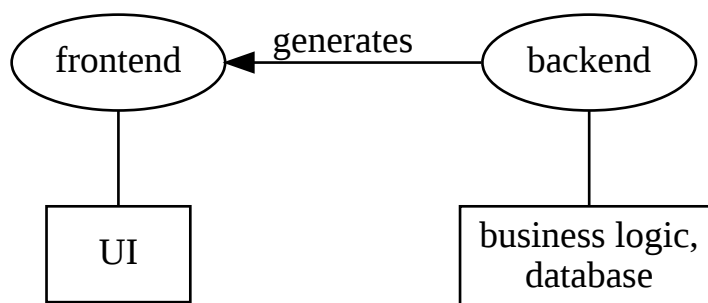
(And we don't acknowledge the existence of **flash** either)

1-3-3



Traditionally, **web apps** looked like this:

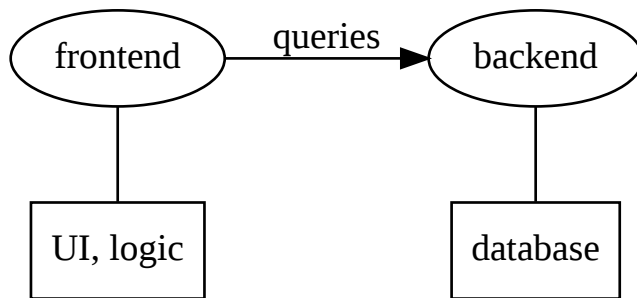
1-3-4



Web tech was **just the UI**. The real coding was done in php / ruby / java / .NET.

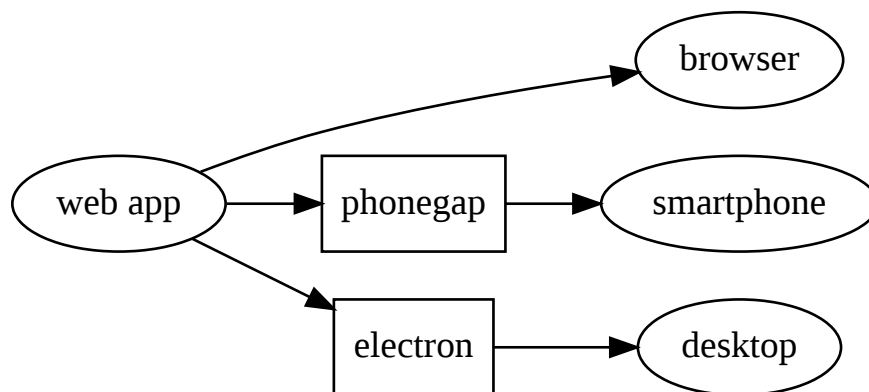
But a **modern SPA** (Single Page Application) is more like this:

1-3-5



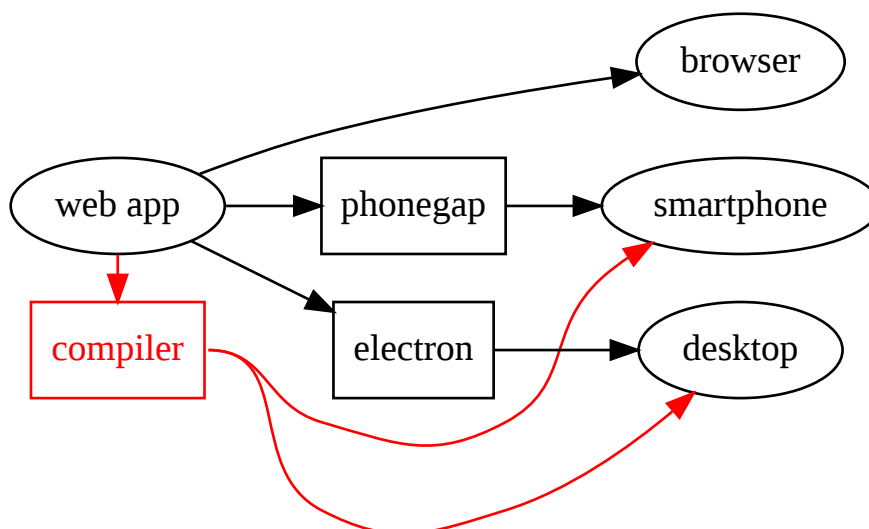
Additionally, web tech has escaped from the browser:

1-3-6



And blurring the lines even further - there are frameworks to **compile** web to native code:

1-3-7



In essence: **web tech is on the (high) rise**, and even if you intend to stay in a sheltered Java garden, knowledge about the web platform will serve you well.

1-3-8

1-4. Approaching the web

this and this and this and

We must admit something up front regarding **learning the web platform**.

1-4-1

It is often **overwhelming to newcomers** that there is an **abundance of choices** to make, for many different aspects;

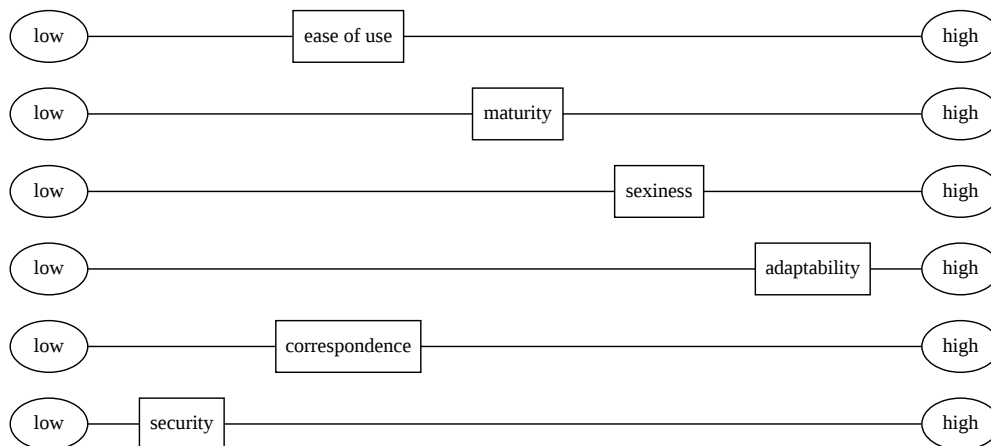
1-4-2

- what **framework** to use
- what **build chain** to use
- which **JS language flavour** to use
- what **CSS preprocessor** to use
- what **style rules** to enforce
- **how to enforce** those rules

...and many more.

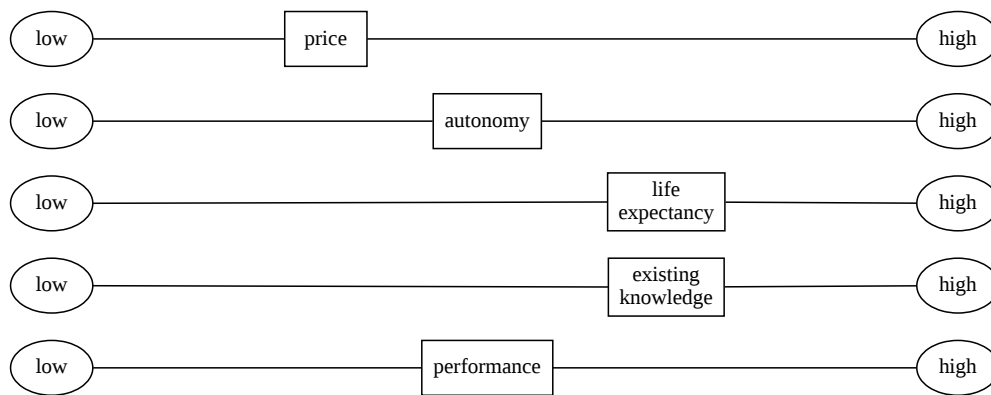
And for **each of those choices**, you must consider;

1-4-3



...and also:

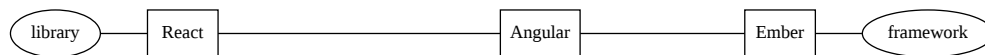
1-4-4



...and likely a few more that we've forgotten.

Regarding the **choice of framework**, the **most important aspect** is perhaps that of **complexity**:

1-4-5



Observe that there is **no right or wrong** in the complexity choice.

1-4-6

The important thing is to **be aware of where on the scale you are**, and **understand the consequences of that**.

To ease into adopting the web, here are some humble pointers!

1-4-7

Don't worry about **having the latest**. That'll be old next week anyway.

1-4-8

Respect the role of the architect. He/she needs to make informed decisions, the rest of us can just relax and use the tools!

1-4-9

If you find a setup that works, **go with it**. You won't find the ideal setup to begin with anyway.

1-4-10

Don't let your pride prevent you from **using starter kits**. For example, in the world of React, [create-react-app](#) is an excellent low-cost entryway!

1-4-11

1-5. The JavaScript language

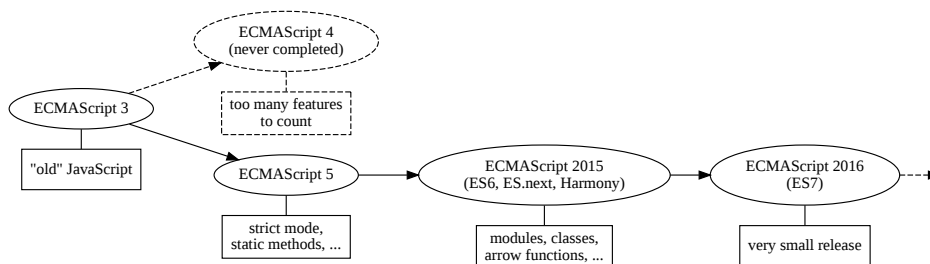
lingua franca of the web

So - **JavaScript**! Let's kick things off with a **chopper view of the language**.

1-5-1

First, here is an overview of the **different JavaScript versions**:

1-5-2



We will **mostly use ES5**, with some more modern stuff sprinkled in.

...as well as some **more stuff beyond the borders of JavaScript**, but that comes later too!

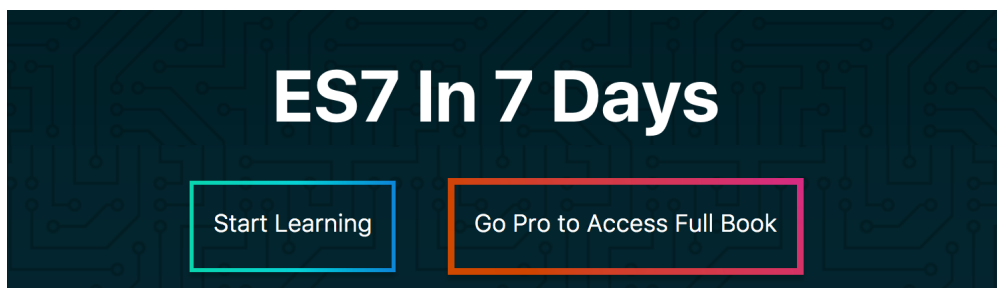
1-5-3

Q So, just **how small is ES7**? Err, I mean, ES2016.

1-5-4

A Well, if you go to <https://getgood.at/js/es7-in-7-days...>

1-5-5



...then you have **3.5 days to learn this:**

1-5-6

```
myArr.includes('foo') // returns true or false
```

...and **3.5 days to learn this:**

```
5 ** 3 // 125, same as Math.pow(5, 3)
```

And ES2017 isn't much bigger; it has 6 features.

1-5-7

What is going on?!

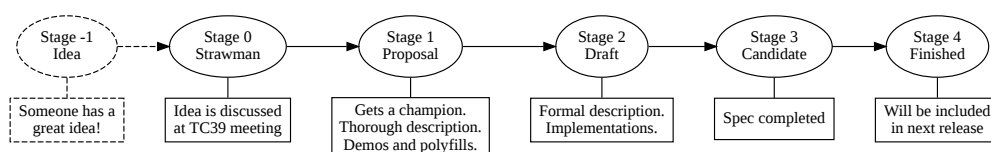
After ES6 (ES2015), TC39 switched to a **yearly release cycle** where **features compete individually**.

1-5-8



The **TC39** process:

1-5-9



Formal description [here](#)

So what's in the pipeline?

1-5-10

- [Stage 0 list](#) (23)
- [Stage 1-3](#) (41, 8, 14)
- [Stage 4](#) (8, not counting ES2016 and ES2017)

(counts valid as of April 2018)

Curious what a **finished feature spec** looks like?

1-5-11

Let's check out the spec for Exponentiation]

(<http://rwaldron.github.io/exponentiation-operator/>)

A **recommended learning resource** for new and upcoming syntaxes are these books by Alex Rauschmayer:

1-5-12

- [Exploring ES6](#)
- [Exploring ES2016 and ES2017](#)

Both are **free online**.

Since we **can't trust all browsers to know the latest JS**, we must **convert our code**. We do this either by...

1-5-13

- a polyfilling
- b transpiling

- a **Polyfilling** means **adding a homebrew version** of a method if a native one isn't available. This **can be done programmatically**, like this:

1-5-14

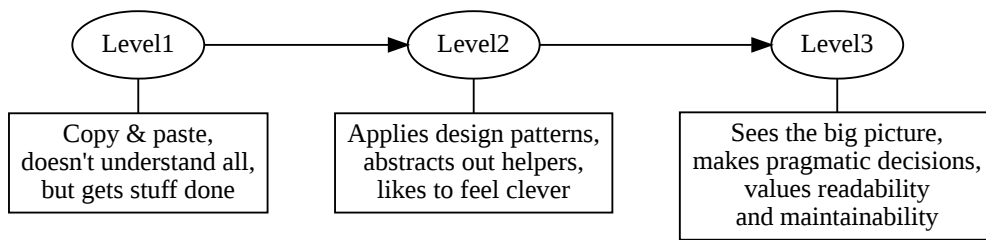
```
Array.isArray = Array.isArray || function(a){  
  return ' ' + a !== a && {}.toString.call(a) == '[object Array]';  
});
```

- b But **syntax changes** requires a **transpilation step** that must be done **prior to running the code**. We **cannot do this programmatically**.

1-5-15

Speaking of JavaScript's character - are you familiar with the **3 levels of programming**?

1-5-16



Q They were interesting, but **what do they have to do with JavaScript**?

1-5-17

A JavaScript's flexible nature **encourages and empowers level 2 behavior**.

1-5-18

So **take care**, and remember:

With great power comes great responsibility

Wielding of that power requires you to **have a grasp JS's quirks**, of which there are many.

1-5-19

These ones are **especially important**:

- **Loose typing** and **truthiness**
- The **implicit this** parameter
- **Lexical scoping**
- **closures**

Here are three **succinct primers** for getting up to speed quickly:

1-5-20

- [MDN:s reintroduction to JS](#)
- [Interactive ES6 features overview](#)
- [JavaScript Garden](#)

For more **general JS references** we recommend [MDN](#) (and **not W3Schools**).

1-6. Course resources

Goodie bag time!

You will be provided with a link to a **course resource** bundle, containing:

1-6-1

- A set of **Demos** - we'll show them off during the course, but this way you can peruse them at your leisure later on
- **Exercises**, which you'll be tasked with accomplishing at set points during the course
- A **PDF** of this very presentation

Introducing React

What's the fuss about?

Sections in this chapter:

1. Templating
2. Enter React
3. Expressing the UI
4. JSX Basics
5. Installing JSX
6. Updating the UI
7. Nested components
8. Connecting to the DOM
9. React devtools
10. Exercise 1

2-1. Templating

Converting data to UI

Here's the **first thing** you should **learn about React**: it is, at its heart, a **templating solution** for dynamic web sites.

2-1-1

That means that React **converts data to UI**. Commonly this step in an application is called the **View layer**, also known as the **V in MVC**.

(...although React is **more than that**, but this is where we start!)

2-1-2

If you have ever used an **application framework** on any platform, you have already used **some sort of templating**.

2-1-3

But even so, let's take a **closer look at the templating problem** before we get started with React.

We'll use the [Handlebars](#) templating library, for no other reason than the fact that it is **very typical**.

Here's a **Handlebars template definition** for showing a **user**. Likely this is harboured in a file called `user.hbt`, where `.hbt` is short for HandleBars Template.

2-1-4

```
<div class="user">
  
  <h4>{{firstname}} {{lastname}}</h4>
  <div>{{description}}</div>
</div>
```

Notice the **double curly braces** which are of course **data placeholders**.

If we send the content of such a file into **Handlebars.compile...**

2-1-5

```
let usertemplate = Handlebars.compile(definition);
```

...then we get back a **template function** that, when called with some data, will **return the corresponding DOM structure**.

So if we have this user object...

2-1-6

```
let data = {
  firstname: 'John',
  lastname: 'Doe',
  imgurl: 'http://some.url',
  description: 'Best imagination EVER.'
};
```

...and feed it to the template function...

```
let DOM = usertemplate(data);
```


...then DOM will be:

2-1-7

```
<div class="user">
  
  <h4>John Doe</h4>
  <div>Best imagination EVER.</div>
</div>
```

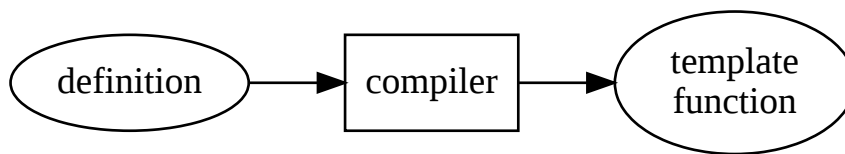
Here's the original template definition for comparison:

```
<div class="user">
  
  <h4>{{firstname}} {{lastname}}</h4>
  <div>{{description}}</div>
</div>
```

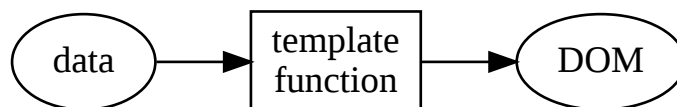
Let's look at the flow zoomed out.

2-1-8

First we **compiled a template** from a **definition**:



That **template** can then **transform data to DOM**:



We've now peeked at how **Handlebars** solves the **data-to-UI problem**.

2-1-9

What about React? Let's find out!

2-2. Enter React

There's a new sheriff in town

At [JSConfUS 2013](#), Facebook introduced a new library - React - with a **radically different approach** to the templating problem that **lifted quite a few eyebrows**.

2-2-1

Then at [JSConfEU 2013](#) they gave **more details**, which made people even **more skeptic**.

Both these videos are **well worth watching**, both for their **explanation of React** and their **historical significance**!

2-2-2

So! Remember how a **template function** turns data into UI?

2-2-3



React does the exact **same thing**, just with a couple of **name changes**:

2-2-4



So a React app is made up by **components**. They are **functions** which you **feed data**, called **props** in React lingo, and you **get DOM** back.

2-2-5

Q Ok, so a React component **serves the same purpose** as a handlebars template. Then what is the fuss about? **What is different** about React?

2-2-6

A Many things;

2-2-7

- We **express the UI in JS** instead of HTML
- We **rerender the whole UI on every update** (what?)
- Components are **inherently composable**

We'll **take a look** at these differences, and more, in the upcoming sections!

2-2-8

2-3. Expressing the UI

Dissecting a React component

In Handlebars you **write a definition** in **html** which is then **compiled** into a **template function**

2-3-1

In React there is **no compilation**. Instead you **work with JS** and **write the template function yourself!**

Here's the mandatory [HelloWorld](#) demo!

2-3-2

```
let HelloWorld = props => <div>Hello world!</div>;
```

As you can see the HelloWorld **component** is a **plain JavaScript function**, although with some **weird xml syntax** mixed in! That's called **JSX**, short for **JavaScript XML**.

The previous example was a bit boring as it was **completely static**. Here instead is a [HelloSomeone](#) component which greets whoever you tell it to:

2-3-3

```
let HelloSomeone = props => <div>Hello, {props.who}!</div>;
```

So, **inside JSX** we use **single braces** to **switch back to JavaScript**.



Hang on - how was that better than Handlebars? Wasn't the earlier Handlebars version...

2-3-4

```
<div>Hello, {{who}}!</div>
```

...much easier to both read and write?



For this simple example, yes. But! Just **regular HTML** is **rarely enough** to formulate the view.

2-3-5

A competent templating solution needs to **provide more opportunity to express logic** in connection with describing what the output should look like.

Consider for example a view that is supposed to render a bunch of **links** to a list of **posts**. We need to...

2-3-6

- **repeat** some HTML for every link
- or **conditionally** display an apologetic message if there are no posts yet.

Here's a **Handlebars** solution to the aforementioned problem:

2-3-7

```
<div class="posts">
  <h2>Posts</h2>
  {{#if posts}}
    {{#each posts}}
      <a class="post" href="{{this.url}}">{{this.title}}</a>
    {{/each}}
  {{else}}
    <p>No posts :(</p>
  {{/if}}
</div>
```

As you saw, **Handlebars** - like so many other templating solutions - has **augmented HTML** with **additional logic helpers**, to accommodate for the fact that **views often need logic**.

2-3-8

This invariably means that you **have to learn these augmentations**, which are always **solution-specific**. Your knowledge on Handlebars helpers is **useless outside a Handlebars context**.

2-3-9

Here is a **React** solution to the same problem:

2-3-10

```
let ListOfPosts = props => {
  let posts = (props.posts || []).map(p => (
    <a class="post" href={p.url}>{p.title}</a>
  ));
  return (
    <div className="posts">
      <h2>Posts</h2>
      { props.posts.length > 0 ? posts : <p>No posts @</p> }
    </div>
  );
};
```

Apart from the JSX syntax, everything else in the code is **just pure JavaScript**. Being an actual programming language, it has **no problem with expressing logic**.

2-3-11

HTML is of course **better** at **expressing markup**, since it is a markup language. This is why Facebook added the **JSX syntax** to JavaScript.

2-3-12

In essence: In a view we need to **describe nested data** (HTML) in connection with **logic**. This gives us two options:

2-3-13

1. Use a **nested data language** with **logic helpers**
2. Use a **logic language** with **nested data helpers**

Since **logic** is **much more complex than nested data**, option 2 seems much easier.

2-3-14

Yet, **everyone but React chose option 1!**

2-4. JSX Basics

Waiter, there's HTML in my JS!

As the previous section taught us, **JSX** is a **light syntactic sugar** aiming to make it **easier to express nested data structures** with JavaScript. Let's explore how it works!

2-4-1

Consider this line of JSX code:

2-4-2

```
<div foo="bar">Hello <strong>WORLD</strong>!</div>
```

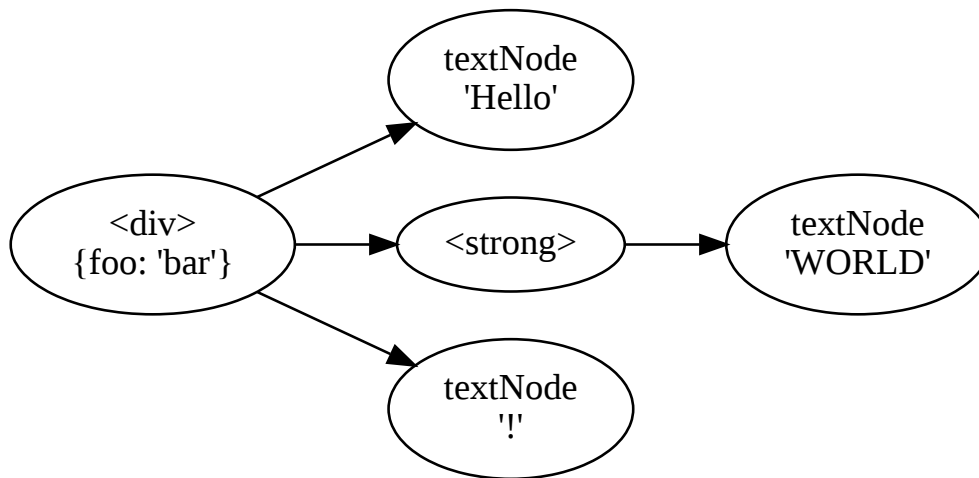
If we **indent** it to show off the **structure**, then it would look like this:

2-4-3

```
<div foo="bar">
  Hello
  <strong>
    WORLD
  </strong>
  !
</div>
```

Here's the same structure as a diagram:

2-4-4



The JSX is simply **transformed** into matching **nested `React.createElement`** calls:

2-4-5

```
React.createElement(  
  "div",           // tagname  
  {foo: "bar"},    // properties  
  "Hello ",        // child 1  
  React.createElement( // child 2  
    "strong",      // tagname  
    null,           // properties  
    "WORLD"         // child 1  
  ),  
  "!"              // child 3  
)
```

Here's the signature for `React.createElement`:

2-4-6

```
React.createElement(tagNameOrReactClass, props, child1, child2,...)
```

Only the **first parameter** is **required**.

Each **child** is either...

2-4-7

- a **string** in which case it is treated as a **text node**
- another **`React.createElement`** call.

Some developers **don't much care for the JSX syntax**. Especially with some **clever aliasing** it is perfectly fine to **make the calls manually**.

2-4-8

If we up top do this **aliasing**...

2-4-9

```
var E = React.createElement, N = null;
```

...then this JSX line...

```
<div foo="bar">Hello <strong>WORLD</strong>!</div>
```

...can be replaced by this:

```
E("div",{foo:"bar"},"Hello ",E("strong",N,"WORLD"),"!")
```

Some even take it a step further and create **methods for all tag names**. So instead of doing this...

2-4-10

```
E("div",{foo:"bar"},"Hello ",E("strong",N,"WORLD"),"!")
```

...they can do this:

```
div({foo:"bar"},"Hello ",strong(N,"WORLD"),"!")
```

The **point** of showing you this isn't that you shouldn't use JSX (because you should), but that **JSX isn't magic**.

2-4-11

It is just a **very small syntax extension** to let us **describe nested data** in a convenient and familiar manner.

We keep saying the **syntax is simple**, but, **how exactly do we use it?**

2-4-12

Well - we **enter JSX mode** simply by **opening a tag**. If you **close the (last) tag** then you are **back in JS mode** again.

```
let License = () => (  
  <p>Steal this and we'll <em>SMACK YOU SILLY</em>!</p>  
);
```

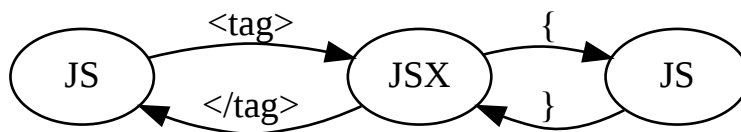
If we want to **go to JS** from **inside JSX** mode, we use curly braces:

2-4-13

```
let msg = <span>Hello, {target || 'mate'}!</span>;
```

Thus the **map** for travelling between JS and JSX could be described like this:

2-4-14



Although note that this can be **nested to any depth**:

2-4-15

```
let output = <div>{<div>{<div>{"Inception!"}</div>}</div>}</div>;
```

2-5. Installing JSX

Who put the HTML in the JS?

Since JSX isn't (yet) a **part of the language**, some kind of **transpilation** must occur.

2-5-1



Otherwise the **browser wouldn't know** how to interpret the code. But **where does this happen?**

The absolute **majority of React developers** have a **build step** that doesn't just take care of JSX, but also allows the use of **modules** and perhaps **other ES6/ES7 features**.

2-5-2

This isn't unique to React, but true for **JavaScript development in general**.

The **most popular tools** for this are [Webpack](#) and [Browserify](#).

2-5-3

They run **JavaScript code on your local machine** using [Node](#), which in turn allows you to **download dependencies** using [npm](#).

There's a lot **more to be said about the build step**, but we'll leave it for now as we'll use an **in-browser solution** to avoid complexity.

2-5-4

What we'll do is use a browser version of [Babel](#), a library that can do all sorts of **transformations on JavaScript code**, including handling JSX.

2-5-5

Our setup is very easy - we **include babel as a script tag** along with everything else we need...

2-5-6

```
<head>
  <title>Playing with React</title>
  <script src="lib/react.min.js"></script>
  <script src="lib/react-dom.min.js"></script>
  <script src="lib/babel.js"></script>
</head>
```

...and then we put our **code in script tags** with **type** set to **text/babel**:

2-5-7

```
<script type="text/babel">
  let MyReactComponent = props => (
    <div>I can use <strong>JSX</strong>!</div>
  );
</script>
```

Babel will **notice the type** and **perform the conversion** to regular JS when the **document loads**.

Note however that this solution is **only for toying and demoing!**

2-5-8

In a **production environment** you should **precompile** in a **build step** as previously discussed.

Q Whether we do it in a build step or in the browser - why isn't **converting JSX a part of React**?

2-5-9

A Two main reasons;

2-5-10

- As shown **JSX isn't mandatory**, you can use React just fine without it.
- Also, **JSX has uses outside of React**. It isn't a **part of React**, but a convenient way to **express nested data**.

As a **non-React example**: here's JSX being used in [CycleJS](#) code:

2-5-11

```
/** @jsx hJSX */ // <-- this points JSX to use hJSX instead

import {hJSX} from '@cycle/dom';

function main(drivers) {
  return {
    DOM: Observable.timer(0, 1000)
      .map(i => <div>Seconds elapsed: {i}</div>)
  };
}
```

2-6. Updating the UI

acting on changes

Let's zoom back out for a bit and consider the problem of **updating our UI** whenever **data changes**.

2-6-1

Handlebars doesn't solve this problem at all, it just provides the **initial DOM**. In a dynamic app we need **additional functionality** to **update** the UI when something happens.

React instead takes a super-simple, brute force approach; it **rerenders everything on every data update**, using the **same function** for the **initial render** and every subsequent update.

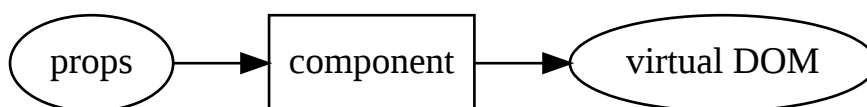
2-6-2

Q Fine, **rerendering everything on every update** makes life easier for us developers, but surely that must have a huge **performance penalty**, lead to **loss of scroll position** and lots of other headaches for the user?

2-6-3

A Nope! And here's why. We actually lied before - a **React component** doesn't output DOM, but **virtual DOM**.

2-6-4



The **virtual DOM** is simply a **JavaScript representation** of a DOM structure. For example, this HTML...

2-6-5

```
<div class="excerpt">
  <h2>Chapter 1</h2>
  <p>When MacGyver learned to fly</p>
</div>
```

...could be **represented in JS** through something like this:

2-6-6

```
let virtualDOM = {
  type: "div",
  attributes: {"class": "excerpt"},
  children: [{
    type: "h2",
    children: ["Chapter 1"]
  }, {
    type: "p",
    children: ["When MacGyver learned to fly"]
  }]
};
```

To see what **virtual DOM** actually looks like, check out the [VirtualDOM](#) demo!

2-6-7

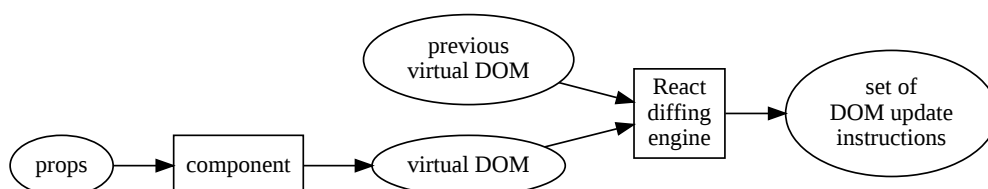
The point of the **virtual DOM** is that it **allows React to compare new output with the previous output**, and figure out what changes needs to be made to the actual DOM.

2-6-8

Only those changes are actually made, so the **complete rerender is just conceptual**.

So here is what **actually happens** when the props of a component updates:

2-6-9



This seemingly simple idea is one of the **biggest advantages of React**, as it **removes lots of complexity** and makes **components easy to reason about**.

2-6-10

Perhaps you've heard of **React Fiber**? That was a **complete rewrite of the diffing engine** for React v16, resulting in **faster diffing and rendering**.

2-6-11

You don't need to know more than that since it is **backwards compatible**, but if you're interested the [detailed explanation](#) of the rewrite is rather fascinating.

2-6-12

2-7. Nested components

Matryoshka time!

It might already be evident, but still worth emphasizing: we can **use components within components**.

2-7-1

This sounds banal, but the **composability** that follows from this is a **huge advantage** of React.

As an example, let's say we've made **components** for the three **sections of a news site**:

2-7-2

```
let Topbar = props => (  
  <h2>The Superhero Tattler</h2>  
)  
;  
  
let Main = props => (  
  return // lots of stuff to generate the main section;  
)  
;  
  
let Footer = props => (  
  return <div>© The Tattler 2016</div>;  
)  
;
```

We can now **use these components** in an all-encompassing **Site component**:

2-7-3

```
let Site = props => (  
  <div>  
    <Topbar/>  
    <Main/>  
    <Footer/>  
  </div>  
);
```

You can try this example in the [Nested components](#) demo.



How can JSX tell the difference between an HTML element and a component?

2-7-4

```
let output1 = <Topbar/>; // Here we mean a component  
let output2 = <div/>;    // Here we mean a regular HTML div element
```



It is ridiculously simple - if the **name is lower case**, it is assumed to be a **DOM tag**,

2-7-5

```
let output1 = <div/>; // --> React.createElement("div");
```

While **upper case** means a **component** which has to be an actual name in your code.

```
let output2 = <Div/>; // --> React.createElement(Div);
```

2-8. Connecting to the DOM

Where does the plug go?

We've mentioned that **React components don't output DOM**, but **virtual DOM** (expressed with JSX). But eventually we do of course want to **turn that into actual DOM**.

2-8-1

How is this done?

Meet **ReactDOM**. This tiny helper library takes **virtual DOM** and puts it into a **target DOM node**.

2-8-2

```
ReactDOM.render(  
  <div>Hello world!</div>,  
  document.getElementById("container")  
);
```

The **virtual DOM input** can of course also be the result of a **component render**:

2-8-3

```
let User = props => <div>Name: {props.name}</div>;  
  
ReactDOM.render(  
  <User name="David"/>,  
  document.getElementById("container")  
);
```

Note that **ReactDOM is sold separately!** We have to **explicitly include it**, as the fox-eyed among you noticed already in our Babel example:

2-8-4

```
<head>  
  <title>Playing with React</title>  
  <script src="lib/react.min.js"></script>  
  <script src="lib/react-dom.min.js"></script>  
  <script src="lib/babel.js"></script>  
</head>
```

There is **much to tell still**, but now you have **enough to say hello to the world!**

2-8-5

2-9. React devtools

The component petri dish

So, hopefully you **remember** the [Nested components](#) from a couple sections seconds back? If we open the demo and check **webkit inspector** we'd see something like this:

2-9-1

```
▼ <div class="comp site">
  <div class="comp top">The Superhero Tattler</div>
  ▼ <div class="comp main">
    ► <div class="comp nav">...</div>
    ► <div class="comp center">...</div>
    ► <div class="comp side">...</div>
  </div>
  <div class="comp footer">© The Tattler 2016</div>
</div>
```

That is **not super useful**, as there is **no way of telling what output comes from what component**.

2-9-2

As React developers it is very convenient to **think in components**, so not being able to do that in the debugger is **very detrimental**!

Enter **React devtools**, a **plugin to Chrome or Firefox**. It provide a **React tab**, which gives the following view instead:

2-9-3

```
▼ <Site> == $r
  ▼ <div className="comp site">
    ▼ <Topbar>
      <div className="comp top">The Superhero Tattler</div>
    </Topbar>
    ▼ <Main>
      ▼ <div className="comp main">
        ► <Navigation>...</Navigation>
        ► <Center>...</Center>
        ► <Sidebar>...</Sidebar>
      </div>
    </Main>
    ► <Footer>...</Footer>
  </div>
</Site>
```

Now we can **easily distinguish between the components**!

It can also **show component details** such as **what props were passed**.

2-9-4

Below we show this for the [HelloSomeone](#) demo:



For more details, check the [React Devtools](#) homepage.

2-9-5

The devtools also do **a lot more**, such as showing **state** and **context**. But you don't know about any of that yet, so, never mind!

2-10. Exercise 1

Dipping our toe in the water

Now you are ready to try your hand at the first exercise! You'll find it in the course resources, named [MyFirstComponent](#).

2-10-1

Rendering lessons

reindeering

Before we turn to **more involved components**, let's **master** the art of **rendering**!

3-0-1

Sections in this chapter:

1. Faulty render return
2. Close all elements
3. Single VS double braces
4. Reserved names
5. Translation issues
6. Spacing trick
7. Conditional render
8. Component children
9. Component props VS Element props
10. Rendering a list
11. Rendering HTML strings
12. Exercise 2

3-1. Faulty render return

There can be only 1

There's a **very common mistake** that'll trip most people up at least once regarding **render output**, which involves doing something like this:

3-1-1

```
return (  
  <h4>Sidebar!</h4>  
  <MyComp somedata={data} />  
  <footer>Copyright</footer>  
);
```

That **won't work**, since **each element corresponds to a `React.createElement` call**.

3-1-2

It makes no sense to return 3 function calls.

If we want to **return 3 things** from a regular function, we have to **put those in an array**.

3-1-3

```
function(someData){
  // ... lots of advanced computations
  return [
    something,
    somethingElse,
    sometOtherThing
  ];
}
```

The equivalent in React prior to version 16 was to **wrap it all in a `div`**:

3-1-4

```
return (
  <div>
    <h4>Sidebar!</h4>
    <MyComp somedata={data} />
    <footer>Copyright</footer>
  </div>
);
```

In essence we had to always **return exactly 1 element** from the render function.

3-1-5

Even if you wanted to return **just a string**, you **had to wrap that in an element**.

But, starting from **React v16**, a component can render:

3-1-6

- an element
- a string (or number)
- an array

No more pointless `div`s and `spans`!

See this in action in the [React16](#) demo.

3-2. Close all elements

lock all doors

In HTML there are several elements which are considered to be **self-closing** (the proper term is **void elements** if you want to get specific).

3-2-1

For example, you **don't need to close an image** element:

```
<header>
  
  <h2>Edument</h2>
</header>
```

But in JSX we **have to close all elements**, since they are translated to function calls.

3-2-2

The **closing tag** becomes the **end parens** of the invocation.

Thus when we want to render an image in JSX, we have to either **self-close** it or provide a **closing tag**.

3-2-3

```

</img>
```

If you forget this then the transpiler will throw a **compile time error**.

3-2-4

In other words, this particular error is **easy to catch** and **cannot cause bugs** in your application.

3-3. Single VS double braces

The more the merrier?

As you've seen, we go from JSX to JS through using a **single pair of curly braces**. In many other templating languages (Angular, Handlebars) we use double braces.

3-3-1

In JSX, **double braces** would mean **switching to JS** (the outer braces) and then **defining an object** (the inner braces).

3-3-2

This is sometimes useful if we want to **pass an object to a prop on a component**:

```
<User data={{name: 'Steve', age: 7}} />
```

A **more clear** but also **verbose** way is to **first define the object**...

3-3-3

```
let user = {name: 'Steve', age: 7}
```

...and then **pass in the variable** instead:

```
<User data={user} />
```



What happens if you forget yourself and **just use single braces** when you **declare an object in a property**?

3-3-4

```
<User data={name: 'Steve', age: 7} />
```



An **error is thrown** when the JSX is being parsed, since this...

3-3-5

```
<User data={name: 'Steve', age: 7} />
```

...means that we try to do this:

```
let dataprop = name: 'Steve', age: 7 // syntax error!
```

3-4. Reserved names

Lost in translation

Because **JSX translates to JavaScript**, words that are **reserved in JS** are **avoided as attribute names**.

3-4-1

Remember how this JSX string...

3-4-2

```
<div foo="bar">Hello</div>
```

...is translated to this JS call:

```
React.createElement("div", {foo: "bar"}, "Hello");
```

The **attribute names** in JSX become **object keys** in JS.

3-4-3

So if we write...

```
<div class="heading">Hello</div>
```

...then the transpiler would output...

```
React.createElement("div", {class: "heading"}, "Hello");
```

...which would **blow up** in older browsers since **class** previously wasn't allowed as a key name.

To work around that we **use className** instead:

3-4-4

```
<div className="heading">Hello</div>
```

React will **take care of making sure** that the resulting DOM node gets the **correct class attribute**.

For the same reason we also **use htmlFor** instead of **for** in form labels.

3-4-5

Forgetting about all this and typing `class` instead of `className` is a **very common mistake** among beginners and veterans alike, so be warned!

3-5. Translation issues

English to French to English shenanigans

When we "send JS into JSX" using curly braces...

3-5-1

```
let output = <div>{someValue}</div>;
```

...what can we send, and **how is it interpreted?**

Here's what!

3-5-2

Strings or numbers end up as **text nodes**, as we've seen many times already:

```
let name = 'Steve', number = 5;
let output = <div>My friend {name} has {number} bottles of beer</div>;
// --> "<div>My friend Steve has 5 bottles of beer</div>"
```

Earlier versions of React put a **span** around the strings, but that **doesn't** happen anymore.

JSX elements are passed straight through:

3-5-3

```
let friend = <strong>Steve</strong>;
let output = <div>My friend {friend} is thirsty</div>;
// --> "<div>My friend <strong>Steve</strong> is thirsty</div>"
```

undefined, null and false don't create anything:

3-5-4

```
let favourite = null;
let output = <div>Steve's favourite meal: {favourite}</div>;
// --> "<div>Steve's favourite meal: </div>"
```

Curiously, **neither does true or functions**.

Items in an **array** will be **interpreted individually** (with no space in-between):

3-5-5

```
let numbers = [1,2,3];
let output = <div>Steve can count: {numbers} bottles of beer</div>
// --> "<div>Steve can count: 123 bottles of beer</div>"
```

And finally, for completion's sake

3-5-6

- an **object** throws an **error**
- NaN renders the string "NaN".

3-6. Spacing trick

Room to breathe

Here's what happens. You define your structure like this:

3-6-1

```
<div>
  <Comp1/>
  <Comp2/>
</div>
```

...but since both Comp1 and Comp2 are **inline elements**, you need a **space between them**. There won't be.

Solve through this little trick:

3-6-2

```
<div>
  <Comp1/>{' '}
  <Comp2/>
</div>
```

The little { ' ' } spacing will result in a **one-space string** at the correct location:

```
<div><Comp1/> <Comp2/></div>
```

3-7. Conditional render

It depends

Let's talk a bit about how to **return different things** depending on the data!

3-7-1

First off, in React, this is a **pure JavaScript problem**. We don't have to look up any specific logic helpers tacked on to the side of HTML, as in Handlebars.

The **simplest solution**, as shown in the [ConditionalReturn](#) demo, is to simply **branch the return statement**:

3-7-2

```
Mirror = props => {  
  if (props.betruthful){  
    return <div>The fairest of all is <strong>Snow-white</strong></div>  
  } else {  
    return <div>The fairest of all is <strong>You</strong></div>;  
  }  
};
```

Often, as in this case, it is more convenient to do the **branching inside the return statement**:

3-7-3

```
Mirror = props => (  
  <div>  
    The fairest of all is  
    <strong>{props.betruthful ? "Snow-white" : "You"}</strong>  
  </div>  
);
```

We show this in the [ConditionalReturn2](#) demo.

Some prefer to **do the branching before returning** by using **variables**, which also works fine:

3-7-4

```
Mirror = props => {  
  let prettiest = props.betruthful ? "Snow-white" : "You";  
  return (  
    <div>  
      The fairest of all is <strong>{prettiest}</strong>  
    </div>  
  );  
};
```


Sometimes we don't need to branch between different output, but instead display **something or nothing**.

3-7-5

A convenient solution is to use the **JavaScript && operator** inside JSX as in the [OnlyIfTrue](#) demo:

```
Mirror = props => (  
  <div>  
    <p>The fairest of all is you!</p>  
    {props.betruthful && <p>...no, actually it's Snow-white.</p>}  
  </div>  
);
```

Q There's a **common related mistake**; can you spot the bug below?

3-7-6

```
return (  
  <div>  
    {generalInfo}  
    {reviews.length && (  
      <p>There are {reviews.length} reviews for you to read!</p>  
    )}  
  </div>  
);
```

A Since...

3-7-7

- **numbers become text nodes**
- the && operator **returns the falsy value**
- the **falsy value is 0** in our case

...the component will **print the number 0** if there are no reviews! See it happen in the [ConditionalFail](#) demo.

3-8. Component children

What happens to the little ones?

It is a **very natural thing** to wrap elements in other elements. That is how we **express the nested structure** of what we want.

3-8-1

```
<ul>
  <li>milk</li>
  <li>toilet paper</li>
  <li>bread</li>
</ul>
```

But what happens when we **wrap elements inside a component**?

3-8-2

```
<Component>
  <span>What will happen to me?</span>
</Component>
```

Inside a component, we can **access elements** that were passed in like this through **props.children**:

3-8-3

```
Highlighter = props => (
  <div className="highlight">{props.children}</div>;
);

let output = (
  <Highlighter>
    Clean up <strong>NOW</strong>!
  </Highlighter>
);

// output is <div className="highlight">Clean... </div>
```

As shown in the [PropsChildren](#) demo

3-9. Component props VS Element props

You say tomato

When we **render an element**...

3-9-1

```
return <div className='something' data-myvar='sth else'>...</div>;
```

the props will end up on the actual DOM node:

```
<div class='something' data-myvar='sth else'>...</div>
```

But when we render a **component**, the **attributes become properties** that are passed to the component, as we've seen many times already.

3-9-2

```
let User = props => <span>Name: {props.name}</span>;  
  
let output = <User name="David"/>; // ---> Name: David
```

A **common mistake** is to confuse this and type:

3-9-3

```
<MyComp className='someclass'>
```

This **won't put the className on the DOM node**, unless `MyComp.render` explicitly passes it on to its output:

```
<div className={props.className}>
```

3-10. Rendering a list

What makes me special?

It is quite common to want to **render a dynamic array** of something.

3-10-1

```
let chores = ["do dishes", "take out trash", "water plants"];  
  
let output = <Chores chores={chores} />;
```

This can be easily accomplished by **prerendering** the list in a **variable** which is then **referenced** in the **output**.

3-10-2

```
let Chores = props => {
  let renderedChores = props.chores.map(chore => <li>{chore}</li>);
  return (
    <div>
      <h4>My chores:</h4>
      <ul>{renderedChores}</ul>
    </div>
  );
};
```

However, if we run the code on the previous slide, React will give a **warning** in the **console**:

3-10-3

```
► Warning: Each child in an array or iterator should have a unique "key" prop. Check the render method of `Chores`. See https://fb.me/react-warning-keys for more information. browserassets.js:2168
```

Since the map method is called with the **index** as second argument, we can use that as **unique key**:

3-10-4

```
let renderedChores = props.chores.map((chore,n)=> (
  <li key={n}>{chore}</li>; // <-- note the `key` prop
));
```

Now we will no longer be given a warning.

Try this out in the [Renderlist](#) demo.

3-10-5

Q Hey, but, since the code **worked without a key** you might wonder - **what's the big deal?**

3-10-6

A It usually doesn't matter, but if we do **dynamic updates** in a list, then React **must be able to distinguish** between components.

3-10-7

3-11. Rendering HTML strings

microwave meals ftw

Imagine you got a string of HTML from somewhere:

3-11-1

```
let longList = "<ul><li>...</li></ul>";
```

How do we **render that from inside a component?**

Simply **outputting the string doesn't work**, it will just render as plain text.

3-11-2

```
let Static = props => <div>{props.static}</div>;

let output = <Static static="<h2>Hello!</h2>" />;
// ---> "<h2>Hello!</h2>"
```

Here's how to do it: We **put the string in the `__html` prop** of an object:

3-11-3

```
let htmlObj = {__html:longList};
```

...and then we **feed that to the `dangerouslySetInnerHTML` attribute** of a container tag:

```
<div dangerouslySetInnerHTML={htmlObj} />
```

Or we can **do it in one swoop** without going via `htmlObj`:

3-11-4

```
<div dangerouslySetInnerHTML={{__html:longList}} />
```

Note how we end up with **double braces** - the **outer pair** is to **shift from JSX to JavaScript**, and the **inner pair declares an object**.



Wasn't that **needlessly complicated**?

3-11-5

- Ⓐ Yes. This is **by design** as injecting html like this is often a bad (and unsafe) idea.

3-11-6

3-12. Exercise 2

Turning up the heat

Let us now hone our rendering skills by taking on **Exercise 2**!

3-12-1

Find it in the course resources with the name [RenderChallenges](#).

React level 2

Things're getting serious

Armed with the **basics of components** and **JSX**, we're now ready to explore **more powerful APIs!**

4-0-1

Sections in this chapter:

1. Class syntax
2. Stateful components
3. Event handling
4. Default properties
5. Lifecycle methods
6. Refs
7. Forms
8. Communication
9. Styling
10. Exercise 3

4-1. Class syntax

When plain clothes just won't cut it

So far we have **defined components** as **plain functions**:

4-1-1

```
let User = props => <div>Name: {props.name}</div>;
```

But there are actually **two ways to define a component**. The plain function components (or PFC:s) are a **shorthand** for this **full definition**:

4-1-2

```
class User extends React.Component {  
  render() {  
    return <div>Name: {this.props.name}</div>;  
  }  
}
```

Our **initial plain function** ended up as a **render method** in the class definition.

4-1-3

...with **one key difference** however. Did you notice it?

The **props** are **not passed in** to the render method. Instead we **access them using `this.props`**.

4-1-4

```
class User extends React.Component {  
  render() {  
    return <div>Name: {this.props.name}</div>;  
  }  
}
```



This syntax seems **clunkier in every way**. Why would we ever want to use it?

4-1-5



The reason is that for more advanced components, there is **more to the story than the render method**.

4-1-6

If we need to **say more than one thing**, we must have a **definition object** - or a **class** - to encapsulate those things.

There's **more details** on classes in the **ES6 appendix**, but there's one thing we should cover here that will keep coming up: ES6 classes can **only have methods**, and **not other properties**.

4-1-7

We can however **fake properties** by marking a method as a **getter**:

4-1-8

```
class Test() {  
  get someProp() {  
    return 42;  
  }  
}  
  
let test = new Test();  
console.log(test.someProp); // 42
```


Still, this is of course rather cumbersome. We want to be able to simply do this...

4-1-9

```
class Test() {  
  someProp = 42;  
}
```

...and have it work the same way!

The good news is that we will soon be able to - there's a [proposal](#) currently at stage 3.

4-1-10

In the meantime, if you don't want to wait, you can use a [Babel plugin](#) (or TypeScript)!

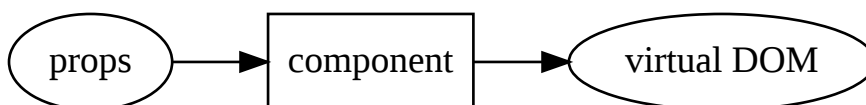
It's worth noting that react-create-app already includes this proposal for us, so if you use that you will end up with the babel plugin automatically!

4-2. Stateful components

I have my secrets

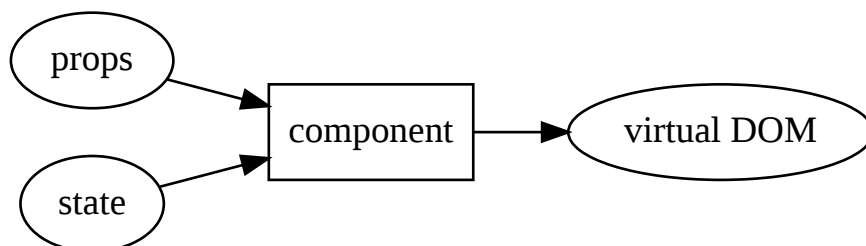
Remember when we said that a **component UI** is the **result of its props**?

4-2-1



That was a **lie**. The UI is a result of **props and state**.

4-2-2



So what is the difference between these two?

4-2-3

- **Props** are passed down from the parent. They are **immutable**, from the perspective of the receiving component.
- **State** is **data** that is **only of internal interest**. It lives only inside the component, and the **component is in control** of it.

The **similarity** however is that when **either of them change**, we **rerun the render** method.

4-2-4

Also, much like props, **state is always an object**.

4-2-5

```
{  
  editing: false,  
  currentValue: 'Batman'  
}
```

We can think of the **individual keys** in that object as **state variables**.

There are **three parts** to component state:

4-2-6

- (a) setting initial state
- (b) reading current state
- (c) updating state

- (a) The **initial state** is set inside the **constructor**:

4-2-7

```
class StateExample extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {data1: 'foo', data2: 'bar'};  
  }  
  // ... more code here ...  
}
```

If you have the ability to use **class properties** we can **assign initial state directly** instead:

4-2-8

```
class StateExample extends React.Component {
  state = {data1: 'foo', data2: 'bar'};
  // ... more code here ...
}
```

A **PFC cannot have state**, since they lack the ability to give information about instances.

4-2-9

(b) We **read from state** simply by accessing **this.state**:

4-2-10

```
render(){
  if (this.state.editing){
    // return editing version
  } else {
    // return view version
  }
}
```

(c) To **update state** we must call **this.setState**. That method expects an object with all parts we want to update:

4-2-11

```
cancel() {
  this.setState({editing: false});
}
```

Note that any **existing state keys not mentioned here** will be **unchanged**.

Also - we might need to do some gymnastics to **make sure this is correct inside the method**:

4-2-12

```
cancel() {
  this.setState({editing: false});
}
```

But we'll address that in the **upcoming section**!

Remember to **never mutate `this.state` directly!**

4-2-13

```
cancel() {  
  this.state.editing = false; // this should use setState instead!  
}
```

The state will be updated, but the **component will not rerender** since React doesn't know something just happened!

Let's internalize the **concepts of state** in a demo: [State](#)

4-2-14

There's **more to say** about `setState`, but we're saving that for a later chapter!

4-3. Event handling

Waiter, there are inline event handlers in my HTML!

Something we haven't seen yet - how do we **attach event handlers** in React?

4-3-1

We can already realize that it is **not done through regular means**, since we **don't have access to the real DOM**.

The answer will feel like **going back in time**: we simply declare **inline event handlers** as props!

4-3-2

```
let ClickableWord = props => {  
  let clickHandler = () => alert("You clicked me!");  
  return <span onClick={clickHandler}>Click me!</span>;  
}
```

When rendering, React will **notice the prop** named `onClick`, and **attach an event handler** accordingly.

4-3-3

This works for **all DOM events**; `onSubmit`, `onFocus`, etc.

Much like jQuery it will pass in a **normalised event object** to the callback.

4-3-4

For the full **list of supported events** and **contents of the event object**, see the **API docs** here: [Events](#)

Q

Aren't we **mixing behavior and visuals** when we declare event handlers in the render function?

4-3-5

A

We are mixing technologies, but **not concerns**. Our component definition encapsulates everything about the component.

4-3-6

When using **Class components** it is customary (but not technically necessary) to **use component methods as event handlers**:

4-3-7

```
class ClickableWord extends React.Component {
  clickHandler() {
    alert("You clicked me!");
  }
  render() {
    return <span onClick={this.clickHandler}>Click me!</span>;
  }
}
```

There's a trap here - the methods are **not autobound to the instance**, so we must be careful when we need to access this. This **won't work**:

4-3-8

```
class ClickableWord extends React.Component {
  showExplanation() {
    alert(this.props.explanation); // BOOM! `this.props` doesnt exist
  }
  render() {
    return <span onClick={this.showExplanation}>{this.props.word}</span>;
  }
}
```

There are **many ways** to solve this. Here are a few common ones:

4-3-9

- a anonymous handler calling the method
- b bind the handler in the constructor
- c define the handler in the constructor
- d assign arrow funcs to class prop
- e using decorators

- a) By calling the handler as a method in an arrow function, this will be correct:

4-3-10

```
class ClickableWord extends React.Component {
  showExplanation(e) {
    alert(this.props.explanation);
  }
  render() {
    let handler = () => this.showExplanation()
    return <span onClick={handler}>{this.props.word}</span>;
  }
}
```

- b) ...or we can bind it in the constructor:

4-3-11

```
class ClickableWord extends React.Component {
  constructor(props){
    super(props); // because we have to, see the ES6 appendix!
    this.showExplanation = this.showExplanation.bind(this);
  }
  showExplanation(e) {
    alert(this.props.explanation);
  }
  render() {
    return <span onClick={this.showExplanation}>{this.props.word}</span>;
  }
}
```

- c) ...or we simply define it as an arrow function in the constructor:

4-3-12

```
class ClickableWord extends React.Component {
  constructor(props){
    super(props);
    this.showExplanation = () => alert(this.props.explanation);
  }
  render() {
    return <span onClick={this.showExplanation}>{this.props.word}</span>;
  }
}
```

- d If you can use **class properties** we can **assign event handlers as arrow functions**:

4-3-13

```
class ClickableWord extends React.Component {
  showExplanation = () => alert(this.props.explanation)
  render() {
    return <span onClick={this.showExplanation}>{this.props.word}</span>;
  }
}
```

The `this` context inside the arrow function will now be the instance, no further work required!

- e Finally, perhaps less commonly used; if we're using Babel to allow **decorators**, we can employ the **AutoBind decorator**:

4-3-14

```
class ClickableWord extends React.Component {
  @autobind
  showExplanation() {
    alert(this.props.explanation);
  }
  render() {
    return <span onClick={this.showExplanation}>{this.props.word}</span>;
  }
}
```

Now `autobind` will make sure that `this` inside `showExplanation` always points to the correct `this`.

You can **explore the examples** in the **Events** demo.

4-3-15

4-4. Default properties

Fallback fun

In some instances it makes sense for a component to provide a **fallback value** if a property wasn't passed in.

4-4-1

We could just do it **manually**, as in this case for the age in the User component:

4-4-2

```
const User = (props) => (  
  <div>Name: {props.name}, age: {props.age || 'unknown'}</div>  
)
```

But it would be nice to explicitly declare our defaults, which is why React provides the defaultProps API:

```
const User = (props) => (  
  <div>Name: {props.name}, age: {props.age}</div>  
)  
  
User.defaultProps = {  
  age: 'unknown'  
}
```

When using a class we put them in a static getter:

4-4-3

```
export class User extends React.Component {  
  static get defaultProps(){  
    return {  
      age: 'unknown'  
    }  
  }  
  render() {  
    return (  
      <div>Name: {props.name}, age: {props.age}</div>  
    )  
  }  
}
```

Or, if we have access to class properties via TypeScript or Babel as mentioned earlier, we can assign them as an object like we did with state:

4-4-4

```
export class User extends React.Component {  
  static defaultProps = {  
    age: 'unknown'  
  }  
  render() {  
    return (  
      <div>Name: {props.name}, age: {props.age}</div>  
    )  
  }  
}
```


4-5. Lifecycle methods

The circle of life

There are several **lifecycle methods** that you can define to hook into certain moments in the life of a component.

4-5-1

When the component is **created**, the following methods are called:

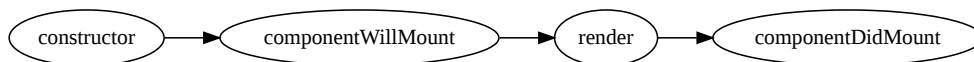
4-5-2



Note that `getDerivedStateFromProps(props, state)` is called before the `render` method, both on initial mount, and on subsequent updates. It should return an object to update the state or, null to update nothing. (Note it's fired on *every* render).

Recently Deprecated:

4-5-3



- `componentWillMount()` is being deprecated and will be removed in 17.0
- `componentWillMount()` renamed to `UNSAFE_componentWillMount()`

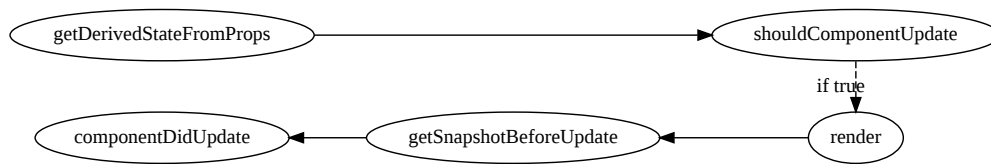
These changes have been made because `componentWillMount()` has typically been misunderstood and is likely to cause issues with async rendering. Commonly issues with error handling, as well as delaying the rendering of components have been caused by using this method improperly.

Any usages should be replaced by a combination of `constructor()`, and `componentDidMount()`. In fact typically most usages of `componentWillMount` actually would be better placed in the `constructor()` or depending on the case in `componentDidMount()`. Of course it is possible to use `getDerivedStateFromProps()` as well, but this is a bit of an anti-pattern.

See [Update on Async Rendering](#) and [React v16.3 release](#)

Then, when a component will update (because **state** or **props changed**), the following are called with the new props and state:

4-5-4

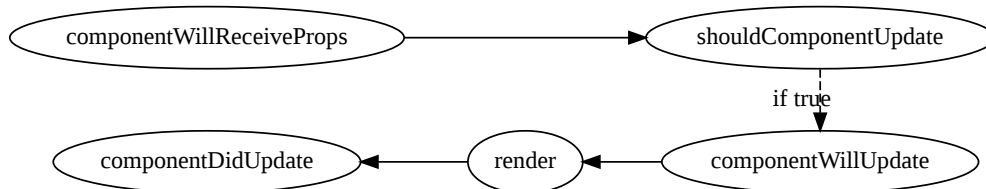


`shouldComponentUpdate()` is perhaps the single most important lifecycle method for improving performance. It allows you to shortcircuit the rendering process.

`getSnapshotBeforeUpdate()` is called before the changes are committed thus you can use it to capture any current values (scroll position and such) before they're potentially changed. Anything returned from `getSnapshotBeforeUpdate()` is passed as a param to `componentDidUpdate()`.

Recently Deprecated:

4-5-5

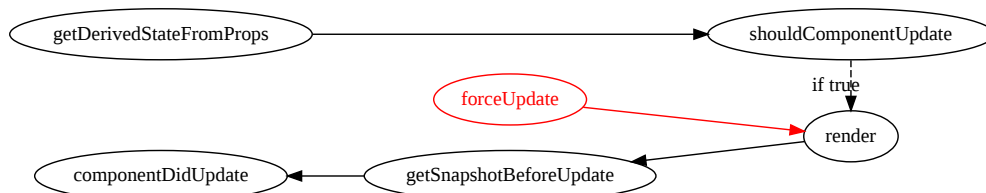


- `componentWillReceiveProps()` is being deprecated and will be removed in 17.0
- `componentWillUpdate()` is being deprecated and will be removed in 17.0
- Both are renamed with `UNSAFE_` prefix

These are also being removed as part of the cleanup of poorly used lifecycle methods. They had a tendency to be misused and so React is moving towards avoiding these commonly misused functions. Eventually they'll be dropped completely.

You can also **forceUpdate**:

4-5-6



But, that is **rarely necessary**, if ever.

Finally, **before** a component is **destroyed**, `componentWillUnmount` is called.

4-5-7

This is a good place to do listener and interval **cleanup**.

Explore these methods in the [Lifecycle](#) demo.

4-5-8



What are the **common use cases** for these methods?

4-5-9



Sometimes we might **kick off something** in **componentDidMount**, and there are times when implementing **shouldComponentUpdate** can **help performance**, but normally you **don't need to bother**.

4-5-10

There's a demo of [ShouldComponentUpdate](#) in action.

In React v16 a **new hook**, **componentDidCatch(error, info)**, is introduced. With that we can **catch errors thrown by children** and grandchildren (but not by the current component).

4-5-11

See it happen in the [ErrorCatching](#) demo.

4-6. Refs

There's a string attached

Sometimes you want to **keep a line open to something you have rendered**. This is accomplished through the **ref syntax**.

4-6-1

It works by **attaching a function to the ref prop** of the output part you are interested in:

4-6-2

```
<Comp ref={refFunc} />
```

Now, when the above is rendered, **refFunc** will be called with **Comp** as a parameter. See this in action in the [Refcomponent](#) demo.



But, why would I want to reference a child component like that? Doesn't it **break uni-directional flow** to do that instead of passing properties?

4-6-3

Ⓐ Absolutely correct! This is a tool for **very special cases**, for example if you are gradually converting a codebase to React. 4-6-4

Refs are much more often used on **regular DOM elements**, when we want to do something with a DOM node. 4-6-5

For example, in the [RefjQuery](#) demo we take a rendered node and execute a (silly) jQuery plugin on it.

A common use case is to **interact with inputs**. If we render this: 4-6-6

```
<input ref={ (el)=> this._input = el }/>
```

...then in `componentDidMount` we could do this...

```
this._input.focus();
```

...and elsewhere we can get the input content like this:

```
this._input.value
```

There'll be more on form shenanigans in the upcoming Forms section, but for now you can try this example in the [RefInput](#) demo. 4-6-7

There is also a **legacy syntax** where you **give ref a string**. The previous example would become: 4-6-8

```
// render
<input ref="_input"/>

// interacting with the node
this.refs._input
```

This is demonstrated in the [RefByString](#) demo.

At first glance the **string syntax** looks easier, it has been deemed **unidiomatic** and **will eventually be deprecated**. 4-6-9

4-7. Forms

Sign here and here and here and

It might not be immediately apparent, but **React isn't a good fit for forms**.

4-7-1

The reason is that the **HTML model** for forms is built on **node mutation**, which doesn't (or at least shouldn't) exist in React!

Consider this vanilla **HTML input element**:

4-7-2

```
<input id="name" type="text" value="John Doe"/>
```

The provided **value prop** will set the **initial value**. When the **user types** in the field, the value prop of the node will be **updated**. So we can **query the current value** like this:

```
var val = document.getElementById("name").value;
```

This **doesn't make sense** in a React setting. The **props are set at render**, and will **never mutate**. If **state or props for the component change**, we simply **rerender** it, but in-between renders nothing should happen!

4-7-3

For example, this means that if we **give an input a value**, that value cannot be changed! See the [PetrifiedInput](#) demo.

4-7-4

So **how do we do it** in React? Well, there are **two different patterns**!

4-7-5

- a Uncontrolled
- b Controlled

- a The first is the one you've already seen in the [RefInput](#) demo, which is called having **uncontrolled inputs**.

4-7-6

This pattern closely resembles what happens in a **regular DOM form**, where we let the DOM node keep its own state.

Having an **uncontrolled input** involves the following:

4-7-7

- We set **initial value** using the **defaultValue** prop
- The form control will **mutate its value prop** when the user makes changes
- We can **read the value** using the **event object** in the **change event**.
- Or we **fetch the value** later by using a **ref**!

Check out the dedicated [Uncontrolled](#) demo.

Q In contrast, the more idiomatic pattern is called having **controlled inputs**. It works like this:

4-7-8

- We set **initial value** using the **value prop** as usual
- That value is **provided by component prop or state**
- We update that value in the **change event**
- This **causes a rerender** which will **update the component**!

Check it out in the [Controlled](#) demo.

Q So, **comparing uncontrolled and controlled inputs** - is it fair to say that...

4-7-9

- with an **uncontrolled input** we store the **state** in the **DOM**?
- with a **controlled input** we store the **state** in the **component state**

A Yes, exactly! :)

4-7-10

Also note that it **never makes sense** to have a **ref on a controlled input** to access its value, since their values are in the state or props of the component. There's no need to read the value from the DOM node.

You can read **all the detail about forms** on the [Forms](#) documentation, but we'll mention **two more things** that are different about using forms in React:

4-7-11

First off: In **regular HTML** you give **default options** a **selected attribute**:

4-7-12

```
<select>
  <option value="U">Urd; that what was</option>
  <option value="V" selected>Verdandi; that which is</option>
  <option value="S">Skuld; That which will be</option>
</select>
```

In **React** we instead set `value` or `defaultValue` on the **select** tag itself (which makes more semantic sense anyway):

4-7-13

```
<select defaultValue="V">
  <option value="U">Urd; that what was</option>
  <option value="V">Verdandi; that which is</option>
  <option value="S">Skuld; That which will be</option>
</select>
```

Try this out in the [Select](#) demo.

Finally the **second difference** we wanted to highlight: in **regular HTML** we provide **initial value** to a **textarea** by passing it as a **child**:

4-7-14

```
<textarea>Enter your comments here</textarea>
```

In React you should instead pass it as a **value** or **defaultValue** prop.

```
<textarea defaultValue="Enter your comments here" />
```

4-8. Communication

Parent-child interaction

A **parent** can **give data to a child** easy enough by **passing props**.

4-8-1

But how does a **child** **give data to a parent**?

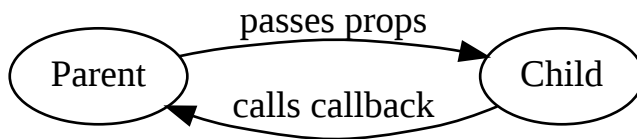
An easy and decoupling pattern is for the **parent** to **pass a callback as a prop**.

4-8-2

When the **child** wants to **give data back to the parent**, it can simply **invoke the callback** with the data.

Thus the **communication cycle** can be described like this:

4-8-3



Look at this **parent** teaching his child to count:

4-8-4

```
class Parent extends React.Component {
  constructor(props){
    super(props);
    this.state = {count:1, feedback: "Let's begin!"};
    this.receiveAnswer = this.receiveAnswer.bind(this);
  }
  receiveAnswer(answer) { /* next slide */}
  render() {
    return (
      <div>
        <strong>Parent:</strong>
        {this.state.feedback} What comes after {this.state.count}?
        <hr/>
        <Child reply={this.receiveAnswer} number={this.state.count}/>
      </div>
    );
  }
};
```

The parent **passes `this.receiveAnswer` as a callback to Child**, along with `this.state.count`. Here's the code for `receiveAnswer`:

4-8-5

```
receiveAnswer(answer) {
  if (answer === this.state.count + 1){
    this.setState({
      count: this.state.count+1,
      feedback: 'Good! Lets try the next one:'
    });
  } else {
    this.setState({
      feedback: 'No, try again!'
    });
  }
}
```


And here's the corresponding **child**, who will **call the callback** as appropriate.

4-8-6

```
let Child = props => {
  let correctAnswer = props.number + 1;
  let cb = props.reply;
  return <div>
    <strong>Child: </strong>
    <button onClick={e=> cb(correctAnswer)}>{correctAnswer}</button>
    <button onClick={e=> cb(correctAnswer+1)}>{correctAnswer+1}</button>
  </div>;
}
```

Try this example in the [Communication](#) demo.

There is also a **more complex** (and **less silly**) example of the same pattern in the [Communication2](#) demo.

4-8-7

And finally, [Communication3](#) is a communication version of the button race in the [State](#) demo.

4-9. Styling

Waiter, there's CSS in my HTML in my JS!

Not only does React make you mix **JS** and **HTML**, it also gives you an opportunity to throw **CSS** into the same messy mix!

4-9-1

The motivation is the same as before - **separating technologies** is **not the same** as **separating concerns**.

4-9-2

If some **styles** are **only of concern to the component**, then they **belong inside the component definition**.

The styles are applied through giving a **style object** to the **style** prop of a tag:

4-9-3

```
let myStyles = {
  backgroundColor: 'blue',
  borderRadius: '5px'
};

let output = <div style={myStyles}>{somecontent}</div>;
```

Note how, much like with the prop names, we must use **camelCasing** instead of **hyphen-connection**.

4-9-4

See it all in action in the [Style](#) demo!

Style objects are a powerful way to really **encapsulate a component**. And since we're just dealing with objects, you can pass in **global style objects** and mix and match as you want, **without** worrying about **specificity clashes**.

4-9-5

But you need to **be mindful** about mixing inline styles like this with regular CSS file, and maintain a good discipline on what to put where.

4-9-6

There are also libraries like [React-Styleable](#) which allows you to **declare your styles in CSS files** but then still **scope to a specific component**.

4-9-7

See the [Styleable](#) demo for an example!

Also, the community is currently exploring **handling CSS from JavaScript space**. This has **immense potential** as there's **no global scope** (unless we want there to be), but the idea is very young and **best practices has yet to settle**.

4-9-8

If you're curious, check out [Aphrodite](#).

4-10. Exercise 3

It's getting hotter still

The next exercise, named [Roster](#), will task you with creating components involving the concepts from this chapter!

4-10-1

Introducing Redux

rewho?

Sections in this chapter:

1. What Redux is
2. State and actions
3. The reducer
4. The store
5. Action creators
6. Redux resources

5-1. What Redux is

Exploring beyond the borders of React

As we said before, **React has a component focus**. It has **no opinions** on how you deal with data beyond component state.

5-1-1

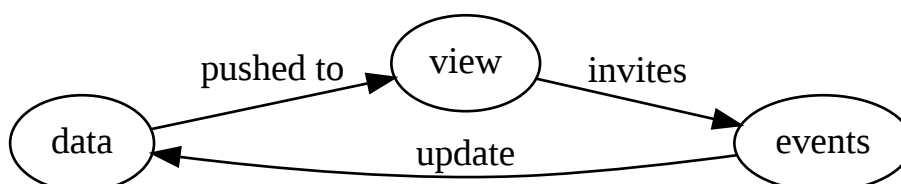
But surely they must give us **some pointers** on how to do it?

Facebook provided a "pattern" to follow which they call **Flux**. This isn't a library, but an idea on how to implement **uni-directional data flow**.

5-1-2

Uni-directional data flow is the idea that data flows **one way** through the app:

5-1-3



Since Facebook preached Flux there has been a **gazillion flux libraries**, and the internet is littered with articles like "comparing the top 10 flux libraries".

5-1-4

Fortunately for you, now **Redux** has come along and **ended the war** by virtue of being the **simplest** and **most powerful** solution!

5-1-5

In spite of being a **tiny** library (less than 100 lines if we remove some warnings and dev fluff), Redux holds some **big ideas** that simply blew the competition out of the water.

5-1-6

5-2. State and actions

How Redux handles data and events

In Redux your **entire app state** is stored in **one single object**.

5-2-1

```
{
  users: {
    id1: { name: "David", ... }
  },
  posts: {
    id1: { title: "My opinion on NEXO Knights", ... }
  },
  ...
}
```

This is in direct **contrast to most other Flux libraries**, who often advocate "thematic stores"; a userStore, a postStore, a commentStore, etc.

5-2-2

Redux' seemingly simple one-store idea holds a surprising amount of power:

5-2-3

- It makes **hydration** easy
- It **lowers complexity**

The twin to that idea is that **every event** that affects that data should be **represented by an action object**:

5-2-4

```
{type: 'ADDPST', authorid: 'user1', content: '...' }  
{type: 'RECEIVEDATA', data: '...'}  
{type: 'DELETEPOST', postid: 'post47'}
```

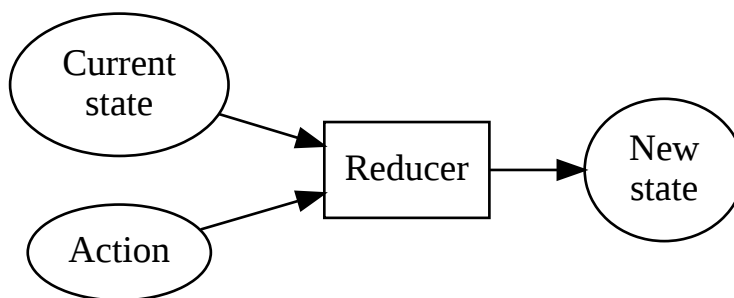
This object has a **type** property and an **optional data** payload.

5-3. The reducer

Deriving updated state

The reducer is so simple it is hard to explain: it is a **pure function** that **takes the state and an action**, and **returns a new state**:

5-3-1



Traditionally the reducer is coded as a **switch** statement with **one case per action type**:

5-3-2

```
let reducer = (currentstate,action)=> {  
  switch(action.type){  
    case C.RECEIVE_QUOTES_DATA:      // ..  
    case C.AWAIT_NEW_QUOTE_RESPONSE: // ..  
    case C.RECEIVE_NEW_QUOTE_RESPONSE: // ..  
    case C.START_QUOTE_EDIT:         // ..  
    case C.FINISH_QUOTE_EDIT:        // ..  
    case C.SUBMIT_QUOTE_EDIT:        // ..  
    default: return currentstate;  
  }  
}
```

In each case we do some computation depending on the action contents, and then we **return a new state**.

5-3-3

Note that we must **not mutate the current state** - it is important that the reducer stays pure and has **no side effects**.

To prevent this from becoming unwieldy we can make **separate reducers** for different parts of the app state, and then *combine* them into one big reducer. We'll take a look at that later.

5-3-4

Perhaps you noticed the **default** path that just passes the current state on?

5-3-5

```
let reducer = (currentstate,action)=> {  
  switch(action.type){  
    // ..lots of cases..  
    default: return currentstate;  
  }  
}
```

This is important as Redux will fire "native" actions that your reducer doesn't know about.

Up until this point you still haven't seen **any Redux code**. All we have talked about are **ideas**, which again goes to show the potency of the Redux model.

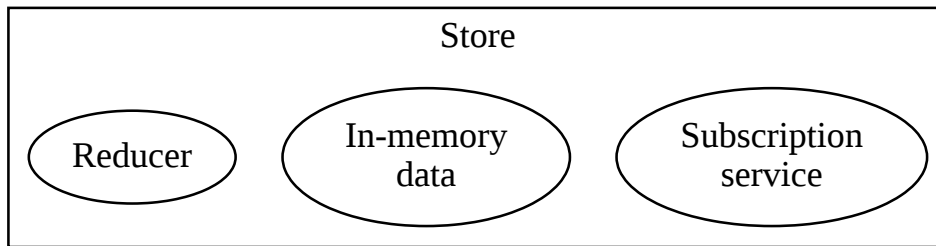
5-3-6

5-4. The store

A house for our data

A **Redux store** consists of three things:

5-4-1



- A **reducer** to calculate new state
- An in-memory **data state** which is updated using the reducer when we pass in an action to the store
- A **subscription service** which notifies interested parties when the in-memory data is updated

We **initialize a store** using a **reducer** and an (optional) **initial state**:

5-4-2

```
let store = Redux.createStore(reducer,initialstate);
```

Look, our first line of Redux code!

We can then **query the state** from the store...

5-4-3

```
let currentstate = store.getState();
```

...and also **update the state** by giving it an action:

```
store.dispatch(action);  
let updatedstate = store.getState();
```

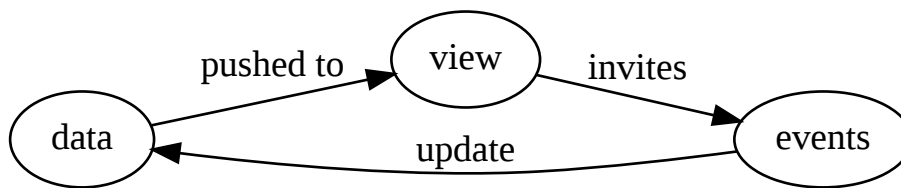
From what we've seen so far, the **store** could be **implemented** like this:

5-4-4

```
Redux.createStore: (reducer,initialstate)=> {  
  let _state = initialstate;  
  return {  
    dispatch(action) {  
      _state = reducer(_state,action);  
    },  
    getState() {  
      return _state;  
    }  
  }  
};
```


In reality **components don't often use `getState`** to query the store, instead the **store notifies the components** whenever there is a change. Remember the **flux flow**:

5-4-5



This is accomplished through a **subscription service**.

The store has a **`subscribe`** method...

5-4-6

```
store.subscribe(myCallback);
```

...which will store all listeners in an array.

```
subscribe(cb) {  
  _subscribers.push(cb);  
},
```

These will then be **triggered on every dispatch**:

5-4-7

```
dispatch(action) {  
  _state = reducer(_state,action);  
  _subscribers.forEach(cb => cb());  
},
```

Here's our **full conceptual code** again:

5-4-8

```
Redux.createStore = (reducer,initialstate)=> {  
  let _state = initialstate, _subscribers = [];  
  return {  
    subscribe(cb) { _subscribers.push(cb); },  
    dispatch(action) {  
      _state = reducer(_state,action);  
      _subscribers.forEach(cb => cb());  
    },  
    getState() { return _state; }  
  }  
};
```

In reality the store is **somewhat more complex** since it allows cancellation of subscriptions, checks for faulty API usage and some other stuff.

5-4-9

But **all of the important stuff** fit onto the previous slide, which yet again shows the succinctness of Redux' approach.

5-5. Action creators

Where do actions come from?

Action creators aren't part of the Redux API, but a useful idea that **views** **shouldn't create actions** themselves.

5-5-1

```
// a bad idea
myViewClickEventHandler(e) {
  let newText = this.field.value;
  let action = {
    type: 'ADDQUOTE',
    text: newText
  };
  store.dispatch(action);
}
```

Instead we make action creators that **convert input to actions**...

5-5-2

```
let actionCreators = {
  addQuote(newText) {
    return {
      type: 'ADDQUOTE',
      text: newText
    }
  }
}
```

...and use these in our views:

5-5-3

```
// a better idea, but still not optimal
myViewClickEventHandler(e) {
  let newText = this.field.value;
  action = actionCreators.addQuote(newText);
  store.dispatch(action);
}
```

Slightly cleaner, but it is still **irksome** to have to **reference the store** in our view. Let's make **bound variants** which automatically dispatches!

5-5-4

```
let boundActionCreators = {};  
for(let action in actionCreators){  
  // using rest params and array spread, see the ES6 appendix!  
  boundActionCreators[action] = (...args) => {  
    let action = actionCreators[action](...args);  
    store.dispatch(action);  
  }  
}
```

Now our views can consume the bound action creators instead, which means they can be **completely unaware of Redux!**

5-5-5

```
// the best idea  
myViewClickEventHandler(e) {  
  let newText = this.field.value;  
  boundActionCreators.addQuote(newText);  
}
```

That was nice, but **creating the bound variants** was a bit of an **effort!** Redux therefore **provides a util to do it more easily:**

5-5-6

```
let boundActionCreators = Redux.bindActionCreators(  
  actionCreators,  
  store.dispatch  
);
```

Having action creators isn't just about **avoiding store references in our views**. Another advantage of action creators is that we now have **every data-affecting event** defined in **one single place**.

5-5-7

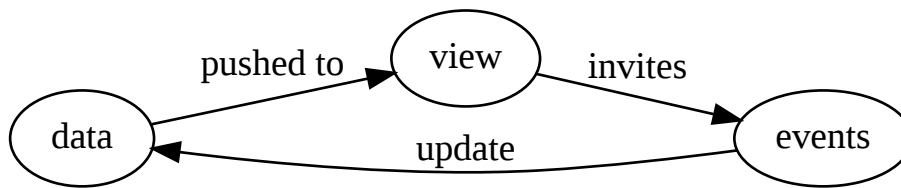
Later we'll...

5-5-8

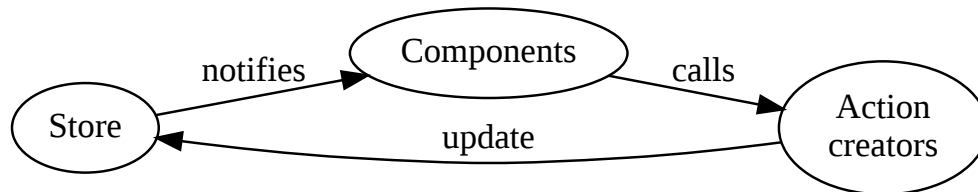
- find out how to do **asynchronous action creators** (by using **Redux-Thunk**)
- look at convenient ways to **marry Redux to React** (by using **React-Redux**).

Zooming out, we can now **translate** the **flux flow**...

5-5-9



...to use **Redux** terms:



5-6. Redux resources

I want moar!

The contents of this chapter will be solidified in the upcoming Redux example chapter, but before we go there we'll give you some **pointers on where to go for more Redux content** after this course.

5-6-1

The already linked-to [Redux](#) official homepage should be your first stop.

5-6-2

It doesn't just contain the **API docs**, but also a **thorough explanation** of the rationale behind the library.

After that you'd do well to devour the free [Egghead Redux course](#) made by Redux' creator Dan Abramov. The bite-sized and to-the-point episodes make the course **very accessible**.

5-6-3

The course does **not involve React** at all, opting to focus on the **essentials of Redux**.

For a meatier but just-as-free video course, check out [Learn Redux](#) by prolific developer and teacher Wes Bos.

5-6-4

This is a **more practical course** throughout which you will be building an application with React and Redux.

...and React Router, which you have yet to meet! The main focus of the course, however, is on Redux.

Beyond these resources a good starting point is [Awesome Redux](#), a **curated collection of links** to various Redux-related content around the internetz.

5-6-5

A Redux example

Learning by doing

We'll now **walk through** a simple **Redux example app** in order to internalize the **basic Redux concepts**.

6-0-1

We won't use React or any other dependencies.

Sections in this chapter:

1. App idea
2. State shape
3. Initial state
4. Action shapes
5. Reducer
6. Store
7. Action creators
8. UI
9. UI updater
10. Interaction
11. Tying it together
12. Exercise 4

6-1. App idea

What are we going to build?

Let's make a simple **counter** app. The counter increases whenever the user clicks a **button**, by an amount controlled by an **input**. It should look something like this:

6-1-1

Counter is now at 8.

Increase by

As we'll find this makes for a very tiny app, yet it will still touch on all the various aspects of Redux.

6-1-2

6-2. State shape

What should our data look like?

Now that we have an **app idea**, let's decide on the **state shape** for our app!

6-2-1

In Redux, all the **app state** is contained in **one single object**. One of the first things we need to do is therefore to design the shape of that state.

6-2-2

However, in our simple counter app, we only need to track one single thing - the current count. Thus the state for our app could simply be a **single number**!

6-2-3

But to make it somewhat less artificial, let's make it into an **object** like this:

6-2-4

```
{count: 7}
```

6-3. Initial state

Kicking things off

Having decided on a **state shape**, however simple, we can now define an **initial state**.

6-3-1

For our little app that only entails **one single decision** - at what number should we start the count?

6-3-2

Let's go with 0!

Now we can write our **first line of code**:

6-3-3

```
let initialState = {count: 0};
```

6-4. Action shapes

What can happen?

Let's design the **actions** needed for our **app idea**!

6-4-1

In a Redux app, everything that **affects app state** is represented by an **action object** with...

6-4-2

- a mandatory **type** property
- and eventual further **data** payload props.

Incrementing the counter is the only thing that happens in our app. Let's decide that the **action** describing this event looks like this:

6-4-3

```
{
  type: 'INCREMENT',
  by: 3
}
```

A side note - Normally we would **isolate all string constants** such as "INCREMENT" to a **separate file**, but in this simple example we'll live with having a single magic string like this.

6-4-4

6-5. Reducer

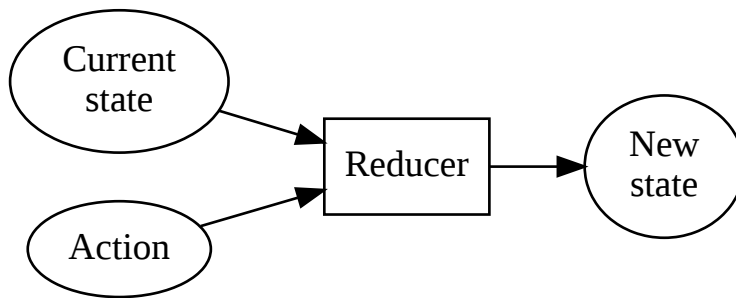
Introducing the heart

Now that we have designed the **action shapes** and the **state shape** we can move on to define our **reducer**!

6-5-1

Remember: a **reducer** in Redux lingo is simply a function that **takes the current state and an action**, and **returns a new state**:

6-5-2



The reducer in our app just needs to **deal with a single action**, namely INCREMENT.

6-5-3

When that action happens we **create a new state** with count increased by `action.by`:

```
let reducer = (state, action) => {
  switch(action.type){
    case 'INCREMENT': return {count: state.count + action.by};
    default: return state;
  }
};
```

Two small **comments** to this code:

6-5-4

- Remember that we must **always have a default case** returning the current state, to deal with internal Redux startup actions.
- Since our reducer deals with just a single action it would be **better phrased as an if-else**. We only used a switch because that's what a reducer normally looks like.

6-6. Store

A house for the data

With the **initial state** and **reducer** in place, we can now create a **store** for our app state!

6-6-1

A store is **instantiated** using **Redux.createStore**:

6-6-2

```
let store = Redux.createStore(reducer, initialState);
```

An **alternative** to providing an `initialState` on store creation is to have a **default return value** inside the **reducer**.

6-6-3

However it can be **valuable** to **explicitly define** the full **initial state** in a single place.

6-7. Action creators

How are actions spawned?

Knowing the shape of the **actions** lets us define **action creators** to be consumed by our **UI**.

6-7-1

Only one single action creator is needed in our app:

6-7-2

```
let actionCreators = {  
  increment(amount) {  
    return {type: 'INCREMENT', by: amount};  
  }  
};
```

We pass in an **amount**, and return an **action** shaped according to our previous decision.

Now we create a version of our action creators that is **bound to our store**, so they're easier to consume in our views:

6-7-3

```
let boundActionCreators = Redux.bindActionCreators(  
  actionCreators,  
  store.dispatch  
);
```

6-8. UI

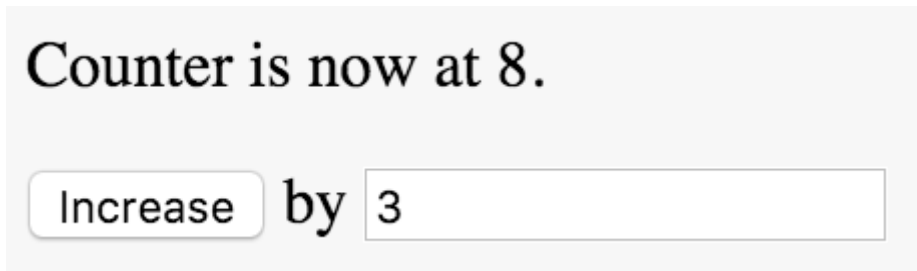
Designing the interface

Let's design the **UI** needed for our **app idea**!

6-8-1

To create our UI we translate our **mental screenshot**...

6-8-2



...to this **html**:

6-8-3

```
<body>
  <p>Counter is now at <span id="num"></span>.</p>
  <p>
    <button id="incbtn">Increase</button>
    <span> by </span>
    <input type="number" id="amount" value=1 />
  </p>
</body>
```

6-9. UI updater

data templating

Now that we know the **state shape** and the **UI**, we can define a **UI updater** function!

6-9-1

We need to update the **UI** depending on the current **state** which is passed in as an argument:

6-9-2

```
let updateUI = state => {  
  let element = document.getElementById("num");  
  element.innerHTML = state.count;  
};
```

All we need to do is update the `#num` element with `state.count`.

In a React app this would rather be a **render** function that **defined the whole UI** instead of mutating existing DOM.

6-9-3

6-10. Interaction

connecting the user

Now we can introduce **interaction** through hooking up the **action creators** to the **UI** and dispatch the result to the **store**!

6-10-1

So what needs to happen when the user clicks the button?

6-10-2

1. We must **collect the increase value** from the input field
2. Then we **create an action** by passing the increase value to the **Increment action creator**
3. We use a **bound action creator** so that the action is **passed to the store**

Here's the code:

6-10-3

```
let button = document.getElementById("incbtn");  
button.addEventListener("click", () => {  
  let input = document.getElementById("amount"),  
      increase = parseInt(input.value);  
  boundActionCreators.increment(increase);  
});
```

6-11. Tying it together

Ship it!

We now have all the pieces we need to finalize our app!

6-11-1

There's only two things remaining to do:

6-11-2

- Make our **UI updater** run whenever the **store** is updated
- Run an **initial rendering**

It sounds easy, and it is easy! Here's the code:

6-11-3

```
store.subscribe(() => { // <-- not called with state!
  updateUI(store.getState());
});

updateUI(store.getState());
```

Note how the **callback** is **not called with state**, which you might expect! You have to **explicitly** get it if you need it.

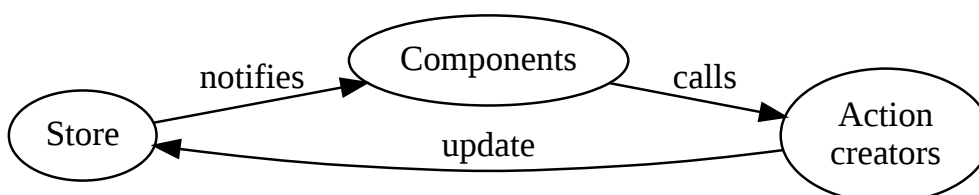
You can **play with this example** in the [Redux](#) demo.

6-11-4

..wait, were we **already done**? Yep, **that was it!**

Note how already this simple example app **demonstrates** the **unidirectional data flow**:

6-11-5



6-12. Exercise 4

It's getting hot in here

You probably saw this coming - you now get to **create your own Redux application!**

6-12-1

Look for the [Generator](#) exercise in the course resources!

6-12-2

React and Redux

sitting in a tree

We have looked at **React** and **Redux** in **isolation**. The time has now come to **marry the two**!

7-0-1

Sections in this chapter:

1. Presenting the app
2. Redux parts
3. React parts
4. Vanilla integration
5. Examining React-Redux
6. React-Redux integration
7. Exercise 5

7-1. Presenting the app

Prepare to be dazzled!

Here's what we're going to build:

7-1-1

Words of Wisdom

- Carpe diem
- Carpe noctem

The functionality is very simple; there is a **list of quotes**, and the user can also **add new quotes**.

7-1-2

In the upcoming sections we will...

7-1-3

- first build the **Redux parts**,
- then the **React parts**, and finally we'll
- look at how to **merge the two together**.

7-2. Redux parts

Installing the plumbing

Our **app state** object has a single **quotes** key, will contain an **array of strings**.

7-2-1

Let's make our **initial state** prepopulated with a quote to set the tone:

```
let initial = {  
  quotes: ['Carpe diem']  
};
```

The single **action** that can happen, **adding a quote**, should look like this:

7-2-2

```
{  
  type: 'ADD',  
  text: 'Do unto others etc etc'  
}
```

Here's the **reducer** to handle this setup:

7-2-3

```
let reducer = (state, action) => {  
  switch(action.type){  
    case 'ADD': return {  
      ...state, // only needed if we had other keys in the state  
      quotes: [...state.quotes, action.text]  
    };  
    default: return state;  
  }  
};
```


We **instantiate the store** as per usual:

7-2-4

```
let store = Redux.createStore(reducer,initial);
```

We only need **one action creator** to interact with our store:

7-2-5

```
let actionCreators = {  
  addQuote(text) {  
    return {type: 'ADD',text:text};  
  }  
};
```

Now **all the Redux parts** needed to support our functionality **are in place!**

7-2-6

7-3. React parts

Dressing the doll

Let's say your team has prepared a **couple of React components** for the app:

7-3-1

- a) QuoteList to **display the quotes**
- b) QuoteForm to hold the **form for new quotes**

Because they didn't know what data handling solution the app would use, they've made the components **portable**.

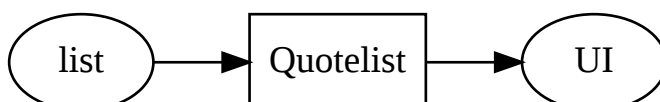
7-3-2

This of course is **good practice** in any scenario, as it makes the code **easier to test** and **less coupled**.

- a) Let's check out **QuoteList**, responsible for **rendering the list of quotes**.

7-3-3

It **expects to receive an array of quotes** as a property.



Here's the **code** for the QuoteList component:

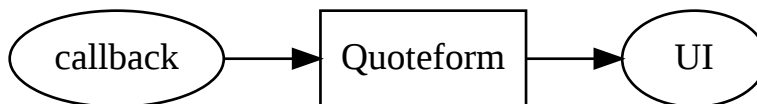
7-3-4

```
let QuoteList = props => {
  let list = props.quotes.map((q,n) => <li key={n}>{q}</li>);
  return (
    <div className="quoteslist">
      <h3>Words of Wisdom</h3>
      <ul>{list}</ul>
    </div>
  );
};
```

- ⓑ And then we have **QuoteForm**, which **renders the form** where the user enters a new quote.

7-3-5

This components **expects to receive a callback** which will be invoked with the text of the new quote when the user clicks the button.



Here's the source code:

7-3-6

```
class QuoteForm extends React.Component {
  submit(e){
    this.props.callback(this.field.value);
    this.field.value = '';
    e.preventDefault();
  }
  render(){
    return (
      <form onSubmit={e=> this.submit(e)}>
        <input ref={i=> this.field = i} />
        <button type='submit'>Add</button>
      </form>
    );
  }
}
```

7-4. Vanilla integration

Marriage attempt, take 1

It is customary to create **container components** around **portable components** like we have now, in order to **provide them what they need**. So let's do that for our two components:

7-4-1

- a) QuoteListContainer
- b) QuoteFormContainer

- a) First we have QuoteList, which expects to be given an **array of quotes** as a prop. Since we must also **update when the data changes**, we need to create a component that **has Redux data as state**.

7-4-2

7-4-3

```
class QuoteListContainer extends React.Component {
  constructor(props) {
    super(props);
    this.state = {quotes: []};
  }
  componentDidMount() {
    store.subscribe(() =>
      this.setState({quotes: store.getState().quotes})
    );
  }
  render() {
    return <QuoteList quotes={this.state.quotes} />;
  }
}
```

In the componentDidMount lifecycle hook we **initiate a store subscription**, which will **update the component state** on every change.

7-4-4

- b) Moving on to QuoteForm - that component needs to be given a **callback** which it **invokes with new quotes**. This should of course be **passed along to the store**.

7-4-5

First we create **bound action creators** for our component to use:

7-4-6

```
let boundActionCreators = Redux.bindActionCreators(
  actionCreators,
  store.dispatch
);
```

We then make a container that passes in the bound action creator:

7-4-7

```
let QuoteFormContainer = props => (  
  <QuoteForm callback={boundActionCreators.addQuote}/>;  
);
```

Initialising this app is pretty straightforward:

7-4-8

```
ReactDOM.render(  
  <div>  
    <QuoteListContainer />  
    <QuoteFormContainer />  
  </div>,  
  document.getElementById("container")  
);  
store.dispatch({type: 'BOGUSEVENT'}); // triggers initial render
```

Try it in a demo here: [Vanilla](#)

This vanilla solution **works just fine**. The only downside is that we had to **access our store to create our containers**;

7-4-9

- **QuoteListContainer** needed **store.subscribe** and **store.getState**.
- **QuoteFormContainer** needed **store.dispatch**.

7-5. Examining React-Redux

Meeting the matchmaker

The [React-Redux](#) provide **official bindings** that **generates wrapper components for us** for us, saving us from having to create our own and spread around references to the store.

7-5-1

React-Redux works through a **.connect** method with the following signature:

7-5-2

```
ReactRedux.connect(  
  mapStateToProps, // connecting to store state  
  mapDispatchToProps, // connecting to store dispatch  
  mergeProps // baking all props together  
) // the component to be wrapped
```

Let's **explore the three parameters** one at a time!

7-5-3

- a) `mapStateToProps`
- b) `mapDispatchToProps`
- c) `mergeProps`

- a) First, **`mapStateToProps`**. It is a **function** which will be **invoked with the store state**, and what you **return** from the method will become **additional props on the component**.

7-5-4

```
let mapStateToProps = appstate => ({
  numberOfPosts: appstate.posts.length
});
```

- b) Then **`mapDispatchToProps`** It can be an object or a function.

7-5-5

If it is an **object** it is assumed to **contain action creators**. They'll be made **available as props on the component** which will **automatically pipe** what **they're returning to the store dispatch**.

If `mapDispatchToProps` is a **function** then it'll be **invoked with the store dispatch**, and you must manually return props just like with `mapStateToProps`.

7-5-6

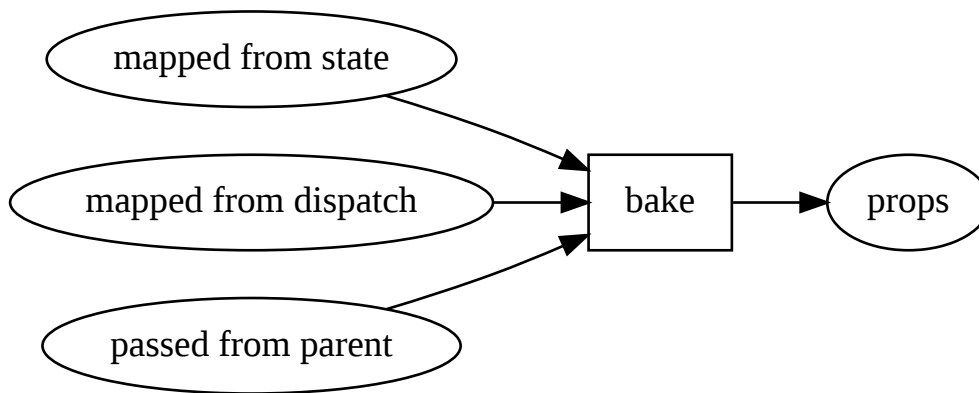
```
let mapDispatchToProps = dispatch => ({
  addPost(text) {
    let action = actionCreators.addPost(text);
    dispatch(action);
  }
});
```

Note how **`mapDispatchToProps`** serves the exact **same purpose** as **`bindActionCreators`**, namely **passing actions** from the view to the store **without needing a store reference**!

7-5-7

- c Finally the **mergeProps** function handles the fact that a connected component **receives props from 3 sources** which must somehow be **baked together**:

7-5-8



If **mergeProps** isn't supplied then ReactRedux will do the following by default:

7-5-9

```
props = Object.assign({}, parentProps, stateProps, dispatchProps)
```

For the majority of cases this is fine.

But if you want control, **provide mergeProps** and do your own baking. It is **called with all three sources** like this:

7-5-10

```
let mergeProps = (fromState, fromDispatch, fromParent)=> {  
  // do your own baking and return it  
  return myBakedProps;  
}
```

The protocol of the **.connect** method and the mapping functions is actually **more versatile** than we have detailed here. See the [React-Redux](#) homepage for the full details.

7-5-11

7-6. React-Redux integration

Employing the matchmaker

Now we must again create...

7-6-1

- a **QuoteListContainer**
- b **QuoteFormContainer**

- a We start with **making the QuoteListContainer**.

7-6-2

Remember, **QuoteList** expected the array of **quotes as a props**, which must also be **kept live** as the data updates.

Thus we'll **need to use the mapStateToProps** function.

That function will be called with the **entire state**, and should return a props object containing the array of quotes:

7-6-3

```
let mapStateToQuoteListProps = appState => ({quotes: appState.quotes});
```

Now we **create the container** using the **.connect** method:

7-6-4

```
let QuoteListContainer = ReactRedux.connect(  
  mapStateToQuoteListProps  
  // we don't need `mapDispatch` or a custom `mergeProps`  
)(QuoteList);
```

This QuoteListContainer will **act exactly like the vanilla version**, including keeping the data updated.

- b Now for **QuoteFormContainer**.

7-6-5

It doesn't need to map state, but it **expects the callback prop** to be a method whose **return value should be dispatched**.

Thus it suffices to simply **rename the addQuote action creator** to "callback":

7-6-6

```
let QuoteFormContainer = ReactRedux.connect(  
  null, // don't need state  
  {callback: actionCreators.addQuote}  
)(QuoteForm);
```

If we had been **allowed to edit** the code of QuoteForm to **use** **props.addQuote** instead of **props.callback**, then we could have **passed in the** **actionCreators** object directly:

7-6-7

```
let QuoteFormContainer = ReactRedux.connect(
  null,
  actionCreators
)(QuoteForm);
```

But, wait - now there's **no single reference to the store** anywhere in our code. How does **.connect** **actually connect** our components to the store?

7-6-8

Here's the **final piece** of the puzzle: we must **wrap our entire app in a** **Provider** component which is **given a store reference**:

7-6-9

```
let Provider = ReactRedux.Provider;

ReactDOM.render(
  <Provider store={store}>
    <div>
      <QuoteListContainer />
      <QuoteFormContainer />
    </div>
  </Provider>,
  document.getElementById("container")
);
```

Behind the scenes the Provider component uses something called **Context** to supply the store to interested children.

7-6-10

We'll **take a closer look at Context** in the **Advanced React** chapter.

Here's a demo with this **library-helped integration** version of the app:
[Usinglib](#)

7-6-11

7-7. Exercise 5

The final boss

As the final test of your new 1337 skillz - **return to the solution you made for exercise 4**, and:

7-7-1

- **make it use React for UI** instead of the jQuery/vanilla solution you have now
- **use ReactRedux** as a bridge between React and Redux!

Redux Thunk

slow motion actions

Redux-Thunk has the following lineage:

8-0-1



So to understand what Redux-thunk is, we will now travel this chain backwards!

Sections in this chapter:

1. Store enhancers
2. Middlewares
3. The thunk problem
4. The thunk solution
5. Trying it out

8-1. Store enhancers

Can't leave well enough alone

Store enhancers are a function that we can **optionally** send to createStore in order to **change the store's behavior**:

8-1-1

```
Redux.createStore(reducer, initialState, enhancer);
```



Wait up - wasn't **initialState** also optional? How then would Redux know the difference here?

8-1-2

```
Redux.createStore(reducer, initialState);  
Redux.createStore(reducer, enhancer);
```

A

Easy - the **initialState** is an **object** while the **enhancer** is a **function**.
Redux can figure out which signature was used through **duck typing**:

8-1-3

```
function createStore(reducer,initial,enhancer){
  if (arguments.length === 2 && typeof initial === "function"){
    enhancer = initial;
    initial = undefined;
  }
  // now initial and reducer are guaranteed to be correct
}
```

Back to what **enhancers actually do**: they are the way in which **third party developers can change the behaviour of Redux**.

8-1-4

Think of enhancers as a **plugin API**!

If you want to send in **multiple enhancers** you must first **combine** them:

8-1-5

```
let enhancer = Redux.combine(devTools,middleWare);
let store = Redux.createStore(reducer,enhancer);
```

Now you may have also noticed the **two most common enhancers**:

8-1-6

- **Redux Dev Tools**, which allows us to **undo actions** and **redo them again, hot reloading** and other things. More on that later!
- **middlewares**, which is what we actually want to know more about right now!

8-2. Middlewares

The meddling man in the middle

They really should be called **action middlewares**, because that's what they do; they **process actions before they reach the store**.

8-2-1

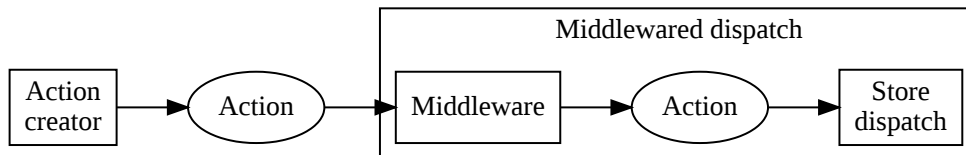
Without a middleware, the action from the action creator is passed straight to `store.dispatch`:

8-2-2



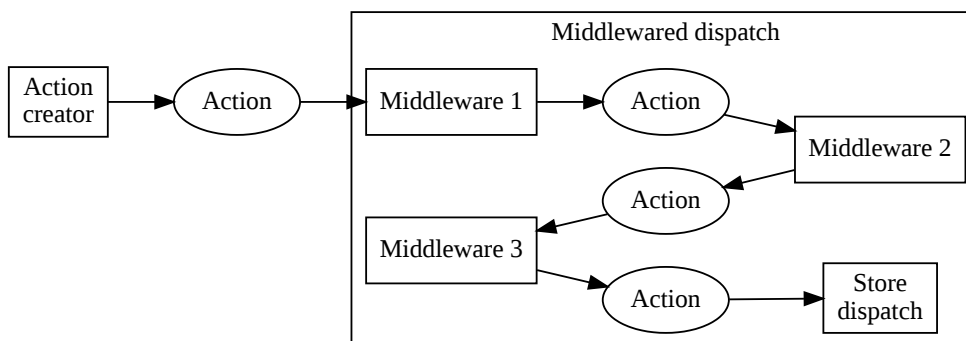
The **middleware sits in-between**, processing the action before (maybe) passing it along to `store.dispatch`

8-2-3



We can have **multiple middlewares**, in which case they'll act as a chain in the order they were applied:

8-2-4



From the outside we **can't tell any difference** - we're just dispatching an action to the store as we always do:

8-2-5

```
store.dispatch(action);
```

To get further familiarised with the **structure of middlewares**, let's now see what they look like!

8-2-6

Here's a middleware that **does nothing**:

8-2-7

```
function noopMiddleware(API){ // has `.dispatch` and `.getState`  
  return function(next){    // `next` is the following middleware  
    return function(action){ // here we receive the `action`  
      return next(action);   // passing the action along!  
    }  
  }  
}
```

If we **convert it to ES6**, here is what we get:

8-2-8

```
let noopMiddleware = API => next => action => {  
  next(action);  
}
```

Here's a **logger** middleware which logs actions and the resulting state to the console.

8-2-9

```
let logger = middlewareAPI => next => action => {  
  console.log("Dispatching action:",action);  
  let result = next(action);  
  let newstate = middlewareAPI.getState();  
  console.log("State after dispatch:",newstate);  
  return result;  
};
```

We'll be using this middleware in the upcoming experiments!

Here's a **Deaf** middleware that only **passes the action along a third of the time**:

8-2-10

```
let deaf = middlewareAPI => {  
  let i = 0;  
  return next => action => {  
    if (!(i++%3)) {  
      next(action);  
    }  
  }  
};
```

Here's a **Nervous** middleware that **executes actions twice** just to be sure:

8-2-11

```
let nervous = middlewareAPI => next => action => {  
  next(action);  
  next(action);  
};
```

Here's an **Impatient** middleware that **multiplies amount by 5**:

8-2-12

```
let impatient = middlewareAPI => next => action => {  
  action = Object.assign({},action); // copy  
  action.by *= 5;  
  next(action);  
};
```

What would happen if we applied **deaf**, **nervous** and **impatient** all at the **same time**?

8-2-13

```
let middlewares = Redux.applyMiddleware(deaf,nervous,impatient);  
let store = Redux.createStore(reducer,initialstate,middlewares);
```

We'd get a store that only acts **a third of the time**, but when it does act it'll **increase the amount by 2*5**. Try this in the **Chaos** demo!

As you've seen, middlewares have **many potential uses**. We can do **flow control** for actions, **manipulate them** and provide **developer utilities**.

8-2-14

8-3. The thunk problem

Eyes on the prize

So now we know what middlewares are, and that Redux Thunk is a middleware, but **what problem is it meant to solve**?

8-3-1

Take a look at this flow again:

8-3-2



We want to be able to **take what action creators return** and **dispatch it to the store**.

This has the benefit that the **action creator doesn't require a store reference**. We can have some outer infrastructure that ties this together for us, which **separates concerns** in a nice way.

8-3-3

But what if we for example need to do some **asynchronous** stuff in our action creators?

8-3-4

Let's say we **make a network request**, and we want stuff to happen both when the **request is made** and when the **data comes back**.

This is **easy to accomplish** if we have access to the store:

8-3-5

```
let loadUrl = url => {
  store.dispatch({type: 'LOADING'});
  myBackend.request(url, data => {
    store.dispatch({
      type: 'RECEIVEDATA',
      data: data
    });
  });
}
```

But, again, giving the action creators access to the store **ties things together** more tightly than is necessary.

8-3-6

Still, we **must be able** to do async stuff like we just saw!

And this is exactly what Redux Thunk gives us: a **way to have complex action creators** but **without having to pass the store around**!

8-3-7

8-4. The thunk solution

Again, we want to...

8-4-1

- always **pass what's returned from the action creators to `store.dispatch`**
- be able to **do advanced stuff in the action creators**
- **not having to pass around the store**

Redux Thunk's answer to this is as simple as it is elegant: it allows us to optionally **return functions** ("thunks") from the action creators **instead of plain action objects!**

8-4-2

Here's what Redux Thunk does:

8-4-3

- Sit at the **beginning** of the middleware chain
- **Examine** all incoming actions
- If they're a function, **invoke** them with `dispatch` and `getState`
- Otherwise **pass them along** to next as usual

Sounds simple, right?

It is **simple**. Here's the **full source code**:

8-4-4

```
let thunk = API => next => action => {  
  if (typeof action === 'function'){  
    return action(API.dispatch,API.getState);  
  }  
  return next(action);  
};
```

Find out more on the [Redux-Thunk](#) homepage.

8-4-5

(including the fact that I lied on the last slide, since a [new API](#) was added last year)

8-5. Trying it out

Put the pedal to the metal

We'll again use our venerable counter app. The action creator for increment looked like this:

8-5-1

```
let increment = function(amount){
  return {type: 'INCREMENT', by: amount};
};
```

The action creator **returns an action object** containing the **amount**.

And here's the **event listener code** again, showing that what is returned from increment is immediately dispatched to the store.

8-5-2

```
button.addEventListener("click",function(){
  let input = document.getElementById("amount"),
      increase = parseInt(input.value),
      action = actionCreators.increment(increase);
  store.dispatch(action);
});
```

Let's now use Redux Thunk and **remodel the action creator** to make this happen when the button is clicked:

8-5-3

- We **immediately dispatch** a WARNING action, letting the user know that the counter is about to be incremented.
- Then **3 seconds later** we'll dispatch the actual INCREMENT action.

Here's the updated action creator to accomplish this:

8-5-4

```
let increment = amount => (dispatch,getState) => {
  dispatch({
    type: 'WARNING',
    msg: 'Now '+getState()+', will add '+amount+'!'
  });
  setTimeout(() => {
    dispatch({
      type: 'INCREMENT',
      by: amount
    });
  },3000);
};
```

Provided that **Redux Thunk sits at the beginning** of the chain, it will **notice that the action is a function**, and so it will **invoke** it with dispatch and getState.

8-5-5

Try this out, together with a **logger** so you can see the 'WARNING' message, in the [Thunk](#) demo.

8-5-6

Should you feel the urge, a more **in-depth version** of this walkthrough can be found here: [Middlewares](#)

8-5-7

React Router

completing the trinity

So we introduced **Redux** to deal with the **data** in our **React** app. But, what about **navigation**?

9-0-1

Sections in this chapter:

1. The need for navigation
2. Exploring an example
3. Dynamic Routing
4. Route Matching Components
5. Navigation Components
6. Parameters
7. Routers
8. Leveling Up

9-1. The need for navigation

Where are we going?

SPA stands for **Single Page Application**. This describes the fact that the page **never reloads** in the traditional sense.

9-1-1

We're dealing with **one single front-end webapp** throughout the session lifetime.

But unless we're building something very simple, we still **need to provide navigation**.

9-1-2

In a SPA we have **two options** for doing so.

Option 1 is to **handle our own navigation**. Whenever the user clicks something to go to a different view, we catch that event and **repopulate the screen** with whatever the user asked for.

9-1-3

The advantage of this approach is that it is **easy**. We're in **total control**, and navigation added **no external dependencies**.

9-1-4

There's a **huge downside**, however: **browser navigation won't work** in our app, meaning:

9-1-5

- The user **cannot bookmark** a position in the app
- Hitting the **back button means leaving the app** altogether

For a serious webapp, these are **dealbreakers**.

9-1-6

Which is why **all frameworks** go with **option 2**, namely to **hook into the browser navigation**. Angular, Ember, Meteor, Aurora - they all have their own **built-in routing** solution.

But **React is not a full framework**. It deals mostly with the view, and doesn't care how you solve the navigation problem.

9-1-7

If you want to hook into the browser navigation, you have to **do it yourself**.

...or, you can include a **companion library** where someone else has already solved the problem! Which is exactly what **React Router** is.

9-1-8

You can get it via:

9-1-9

- `npm install react-router-dom --` DOM bindings
- `npm install react-router-native --` React Native bindings

Don't install react-router directly, instead take one of the above.

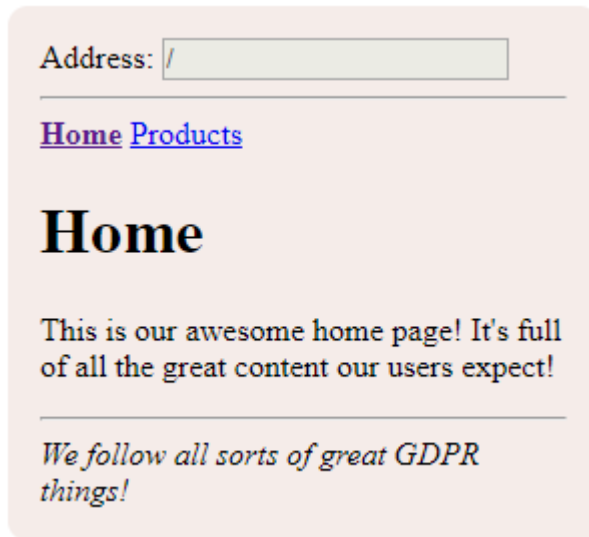
9-2. Exploring an example

Routing in the real world

Before we peek under the hood, we'll explore a simple example app that uses React Router.

9-2-1

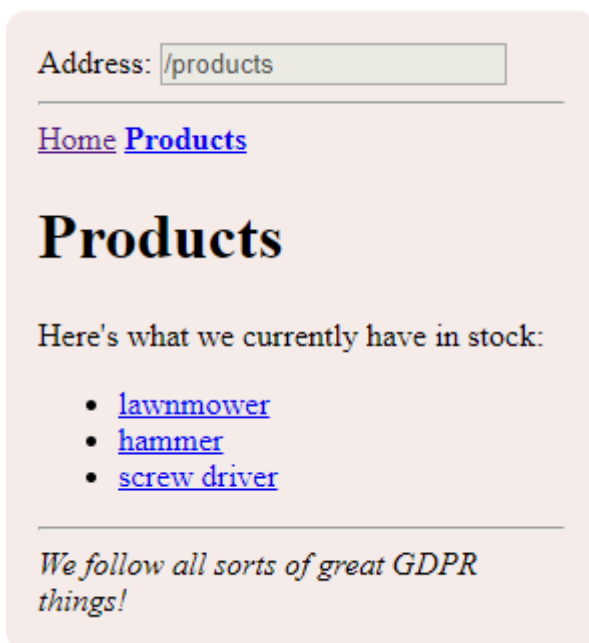
Here's what the **home screen** looks like:



Note how the **home** link in the navbar is active.

Clicking on **products** takes us to this list:

9-2-2



Now the **Products** link in the nav menu is active.

Finally here's the screen for a **specific product**:

9-2-3



Note how the **Products** link is still active, since we're still considered to be in the same section.

As stated a very simple app, yet still enough to catch the most common routing needs.

9-2-4

We'll be walking through it together, but you can try the demo here: [RouterV4](#)

9-3. Dynamic Routing

Where to go

Traditional routing solutions tend to be **static routing** solutions.

9-3-1

- Express
- Ember
- Angular
- React Router (prior to v4)

Routes are defined up front, before your app renders.

9-3-2

```
Router.map(function() {
  this.route('home');
  this.route('about');
  this.route('products', function() {
    this.route('show', { path: '/:product_id' });
  });
});

export default Router
```

example from Ember

At the heart of an app using React Router v4+ is the idea of **dynamic routing**

9-3-3

Routing takes place as the app is rendering

This in turn closely matches the **component architecture** of React

9-3-4

We use a Route higher order component:

9-3-5

```
<Route exact path="/" component={Home}/>
<Route exact path="/products" component={Products}/>
<Route path="/products/:product" component={Product}/>
```

This will render either the component (if the path matches) or nothing:

9-3-6

```
▼ <Route exact=true path="/" component=Home()> == $r
  ▼ <Home match={path: "/", url: "/", isExact: true, ...} location={pathna
    ► <div>...</div>
    </Home>
  </Route>
  <Route exact=true path="/products" component=Products()></Route>
  <Route path="/products/:product" component=Product()></Route>
  ► <Footer>...</Footer>
```

We can nest these however we like:

9-3-7

```
//In App:
<Route path="products" component={Products}/>

const Products = props => (
  <div>
    Something about all products
    <Route path="/products/:product" component={Product}/>
  </div>
);
```

Even without fully understanding every detail, we can see that the route component gives us a powerful new way of defining routes.

9-3-8

React Router components are divided into 3 types

9-3-9

- Route Matching Components
- Navigation Components
- Routers

We'll explore each of these in the following sections.

9-4. Route Matching Components

Building blocks

We've already seen our first **Route Matching Component**:

9-4-1

- Route
- Switch

Route can be included **anywhere** you want to render content based on the location.

9-4-2

You may have noticed the use of the `exact` prop to determine how matching works.

9-4-3

```
<Route exact path="/" component={Home}/>
<Route path="/about" component={About}/>
```

Here the `Home` component needs to have `exact` so it isn't shown at the same time as the `About` Component

Here `Products` will be rendered for all of the following routes:

9-4-4

```
<Route path="/products" component={Products}/>
<Route path="/products/:product_id" component={SingleProduct}/>
```

- `/products`
- `/products/saw`
- `/products/blanket`

`Switch` is used to group `Routes` together

9-4-5

```
<Switch>
  <Route exact path="/" component={Home}/>
  <Route path="/about" component={About}/>
  <Route path="/" component={NotFound}/>
</Switch>
```

Only the first **match** is rendered.

This can be quite handy for dealing with 404 or **not found** type errors.

9-4-6

9-5. Navigation Components

Links, Links, Links!

We'll now take a look at how to implement **links** in an app using `React Router`.

9-5-1

As you perhaps noticed in the [RouterV4](#) source, we can use the **Link** component from React Router:

9-5-2

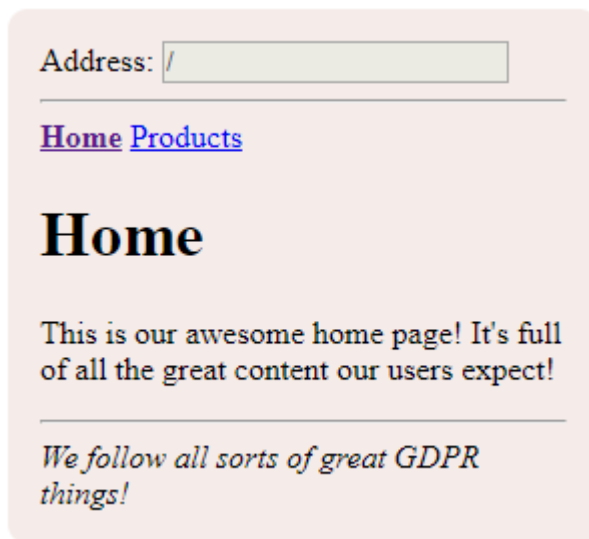
```
import { Link } from 'react-router-dom';

let l = <Link to="/about">About</Link>;
```

When clicked, the link `l` will navigate to the route `/about`.

Remember how the navbar links in our example app were **highlighted** if the route they linked to was currently active?

9-5-3



We accomplish that by using `NavLink`s instead.

9-5-4

Then we can set the `activeClassName` property:

```
<NavLink to="/products" activeClassName="active">Products</NavLink>
```

This link will be rendered with the CSS class **active** if our current route matches `/about`.

Alternatively we can provide `styles` object directly:

9-5-5

```
const activeStyle= {
  fontWeight: 'bold'
}

//...

<NavLink exact to="/products" activeStyle={activeStyle}>Products</NavLink>
```

However, this clashes when we link to / since **that route will always match**.

9-5-6

```
<NavLink exact to="/" activeStyle={activeStyle}>Home</NavLink>
```

That's why we use an exact when linking to such a route, to only have it active if we are literally at that route and no other.

Here are the navbar links from the example app demonstrating this:

9-5-7

```
<NavLink exact to="/" activeStyle={activeStyle}>Home</NavLink>  
<NavLink to="/products" activeStyle={activeStyle}>Products</NavLink>
```

9-6. Parameters

There's no party without them!

You saw them flash by in our example app for the **product item page**. Here's that route definition again:

9-6-1

```
<Route path="/products/:product" component={Product}/>
```

The **colon** makes the last part of the path into a **parameter**.

Which means that if we...

9-6-2

- **navigate** to **/products/dishwasher**
- then the route **/products/:productid** will **match**
- and the parameter **productid** will equal **dishwasher**.

Observe that the **colon** is only used in the **routes definition**.

9-6-3

We do **not** use it **in our URL:s** when we navigate or link.

We can see this in action in the **source code for the Product component**:

9-6-4

```
export const Product = props => (  
  <div>  
    So you want to buy a {props.match.params.product}, do you?  
  </div>  
);
```

Note how the **parameters are available** on `props.match.params`. This is React Router's doing.

9-7. Routers

Hooking things up

Now the time has come to zoom out and see how to **configure** and **initialize** all this!

9-7-1

Here's the relevant code from the example app:

9-7-2

```
import { MemoryRouter as Router, Route } from 'react-router-dom';  
  
ReactDOM.render(  
  <Router>  
    <div>  
      //...  
    </div>  
  </Router>,  
  document.getElementById("container")  
);
```

As you can see we use a Router component as a **root component**.

Note the difference between Router and Route!

9-7-3

- A Route represents a **single route**, so we have many of those.
- Router is the **root component**, which receives all Routes as sub-children.
- A Router can have exactly 1 child.

What is the **MemoryRouter** type? That dictates **how** React Router should **hook up paths to the browser**.

9-7-4

There are 4 different implementations built in;

- **MemoryRouter**
- **HashRouter**
- **BrowserRouter**
- **StaticRouter**

We'll now take a quick look at each of them.

MemoryRouter doesn't connect to the browser at all, but instead **handles the navigation state in memory**. Much like **Option 1** for SPA:s that we mentioned initially as an example of what not to do.

9-7-5

So why would we want to use MemoryRouter? Three main reasons:

9-7-6

- For **small apps that are living inside a larger app**, for example a JSBin demo
- For rendering our apps **server side** where there is no browser
- For **testing**
- For **React Native**

Our example falls under the first point.

If we used **HashRouter** then the app **path lives in the hash of the URL**:

9-7-7



blog.krawaller.se/react-router-demo/#/products/lampshade?_k=hl9mxa

The meaningless stuff at the end, `?_k=hl9mxa`, is an unfortunate artifact necessary to reliably track unique state.

If we instead used **BrowserRouter** then we'd get a clean, "regular" URL with no hashes or artifacts.

9-7-8

But this **requires server-side configuration** to handle the case when the user starts somewhere else other than the root, which is why the example app doesn't use it.

StaticRouter is a Router that never changes location.

9-7-9

9-8. Leveling Up

FAQ time!

We introduced Route using the component setup:

9-8-1

```
<Route exact path="/" component={Home}/>
```

Q So one question you might have is how to pass **props** to a component?

9-8-2

A Use the render type of Route instead of component:

9-8-3

```
<Route exact path="/products" render={ (props) => (  
  <Products {...props} products={products}/>  
)} />
```

We can see this in the [RouterV4Render](#) demo.

A couple of guidelines:

9-8-4

- Don't use both component **and** render
- Don't pass a lambda into component (will cause mount/un-mount on every render)

Handling Authenticated Routes

9-8-5

There is one other route matching component: Redirect

```
<Redirect to="/somewhere/else" />
```

Using this we can create a ProtectedRoute component:

9-8-6

```
let isLoggedIn = true;

const ProtectedRoute = ({ component: Component, ...rest }) => {
  return <Route {...rest}
    render={() =>
      props =>
        isLoggedIn
          ? <Component {...props}/>
          : <Redirect to="/login" />
    } />
};
```

As seen in the [RouterV4Auth](#) demo.

Accessing history and location

9-8-7

Sometimes we need to access the current location, or the history from a component.

We can use the withRouter() higher-order component to add the following props:

9-8-8

- match
- location
- history

9-8-9

```
import { withRouter } from 'react-router-dom';

const AddressBar = props => (
  <div>
    Address: <input type="text" value={props.location.pathname}/>
    <hr/>
  </div>
);
export const AddressBarWithRouter = withRouter(AddressBar);
```

RouterV4 Address Bar

Redux level 2

Redux redux

Sections in this chapter:

1. Redux devtools
2. Combined reducers
3. Caching with Reselect
4. Where to put logic
5. Flavours of state

10-1. Redux devtools

Activating the flux capacitor

We hinted earlier that **the Redux model allows time travel**, which can be a **powerful debug tool**.

10-1-1

This is implemented in the [Redux Dev Tools](#), which are available as a **library** or a **chrome plugin**.

Hot module replacement means that you can **make changes to your code and reload the app *without* losing state**.

10-1-2

Time travel is very powerful for **tracking down erroneous behavior**. It also **looks really cool!**

10-1-3

If you **install the devtools as a Chrome extension**, you can see for yourself in the [ReduxDevtools](#) demo. There we've simply **added the dev tools to the Quotes app**.

10-2. Combined reducers

The more the merrier

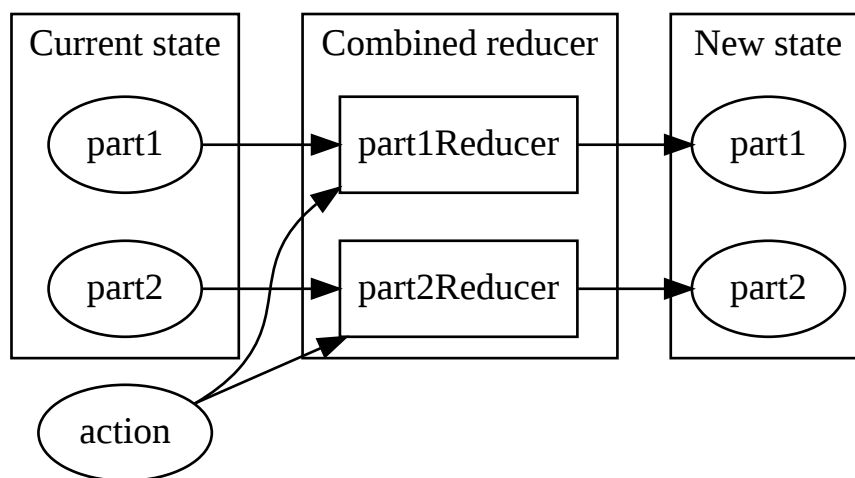
In an **app with lots going on**, the switch in our reducer can quickly **grow out of control**.

10-2-1

The remedy is to have **one reducer per key in our state**, and then **combine these** into a single, app-wide reducer!

That means we'll get a **structure** like this:

10-2-2



We will explore that by **adding more state** to our previous **Quotes app example**, which has now received a footer with **passed time**:

10-2-3

Words of Wisdom

- Carpe diem

App has been running for 15 seconds

The **updated app state** looks like this:

10-2-4

```
let initialState = {
  quotes: ['Carpe diem'],
  time: 0
};
```

The quotes-related data lives in `state.quotes`, and the time-related data lives in `state.time`. We'll now make **one reducer for each of these keys**.

Each of these reducers will **act only on their part of the state**, which means that we can reuse the previous reducer as a `quotesReducer`:

10-2-5

```
let quotesReducer = (state, action) => {
  switch(action.type){
    case 'ADD': return [...state, action.text];
    default: return state || initialState.quotes;
  }
};
```

The only change is the **default `initialstate` return**, which is **customary in combined reducers**.

Now we must add a **`timeReducer`**, which will be very simple:

10-2-6

```
let timeReducer = (state, action) => {
  switch(action.type){
    case 'TICK': return state + 1;
    default: return state || initialState.time;
  }
}
```

Finally these reducers are **combined** and passed to the store:

10-2-7

```
let reducer = Redux.combineReducers({
  quotes: quotesReducer,
  time: timeReducer
});

let store = Redux.createStore(reducer);
```

Note that we **no longer need to pass `initialstate` to `createStore`**, since we dealt with that in the separate reducers.

You can try the augmented Quotes app in the [CombinedReducers](#) demo.

10-2-8

Of course, for this simple app, it would have been perfectly fine to have one reducer work on the entire state object. But as things grow more complex, **having separate reducers will really increase code readability.**

10-3. Caching with Reselect

Memoizing the bindings

A popular **companion library** to Redux is [Reselect](#), which helps with:

10-3-1

- computing **derived data** so that you don't need to store it in Redux
- **memoizing** expensive lookups to make Redux more efficient

We will **explore using Reselect for derived data** by adding the **average seconds per quote** to the footer of our Quotes app:

10-3-2

Words of Wisdom

- Carpe diem
- Carpe noctem

Add

App has been running for 12 seconds
Writing a quote takes 6 seconds on average

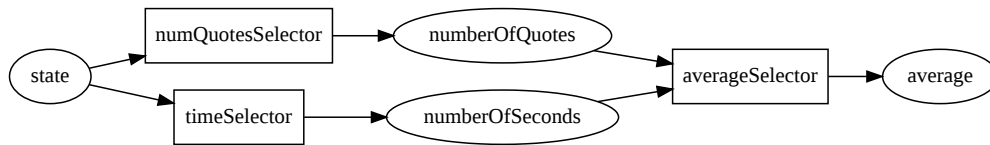
Here are the **main ideas** of Reselect:

10-3-3

- We work with the concept of **selectors**.
- You make a selector for **each piece of data you need from the state**.
- The selectors can then be **combined**, and these combinations will be memoized.

In our case we need to select **the number of quotes and seconds passed**, from which we can then select **average time per quote** which we show in the app.

10-3-4



Here's the **code for numQuotesSelector**. It is a **plain function** that is called with the app state, and then **returns the length of state.quotes**:

10-3-5

```
let numQuotesSelector = state => state.quotes.length;
```

And **timeSelector** is similarly simple:

10-3-6

```
let timeSelector = state => state.time;
```

The interesting part comes now when we create **averageSelector**:

10-3-7

```
let averageSelector = Reselect.createSelector(  
  [numQuotesSelector, timeSelector],  
  (numquotes, secs) => secs / numquotes  
);
```

Reselect.createSelector takes an **array of selectors** and a **constructor function** which is **invoked with the results from the passed selectors**.

We can now **use averageSelector inside mapStateToProps** when we create a container for the footer:

10-3-8

```
let mapStateToFooterProps = state => ({  
  time: state.time,  
  avg: averageSelector(state)  
});
```

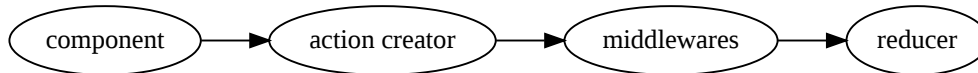
See the whole thing running in the [Reselect demo](#)!

10-4. Where to put logic

app or action creators or reducer?

Events pass through a Redux app like this:

10-4-1



...which means that we often have to ask us **where to put a piece of logic**

10-4-2

-

- (a) in our **components**?
- (b) in our **action creators**?
- (c) in our **middlewares**?
- (d) in our **reducers**?

- (a) The **components** are rarely a good answer, unless the logic is **only relevant to this particular component**.

10-4-3

- (b) The **action creators** are often a good answer. They can be **full of side effects**, and give rise to different actions depending on the circumstances.

10-4-4

- (c) The **middlewares** are only useful in a very special circumstance - we have logic that wants to deal with **every action**.

10-4-5

But in that particular case, they are a perfect fit!

- (d) The **reducers must be pure**, but even apart from that, it is often advantageous to have **complex logic in the action creator instead**.

10-4-6

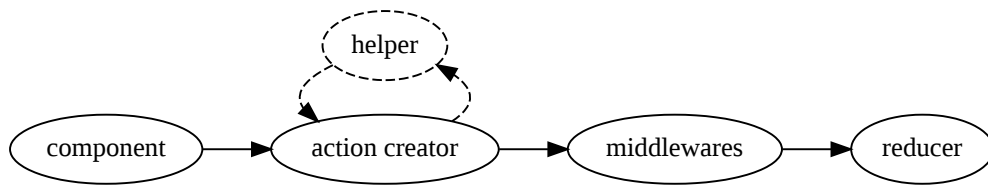
Especially **beware of if-else constructs** in reducers - it is often more flexible to **move the branching to the action creators** and letting that fire different actions for each branch.

10-4-7

So **don't be afraid of fat action creators**, even if it means skinny reducers!

- e There is also a fifth option, namely **somewhere else** entirely.

10-4-8



Imagine a board game app; instead of having an action for attacking with each unit type, have a single attacking action and let a helper library do the heavy computing.

10-4-9

10-5. Flavours of state

Lime and chocolate don't mix

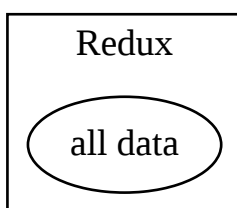
It's not immediately obvious, but when we introduce Redux to an app, we **need to make choices on where to store what**. There are three approaches:

10-5-1

- a Everything in Redux
- b Some in React
- c Some also in a router

- a The easiest to define is to store **everything in Redux**. We **don't use `setState` anywhere in our app**, except for subscribing to Redux state.

10-5-2



Even trivial UI state, such as a flag for whether a table cell is being edited or not, goes into Redux.

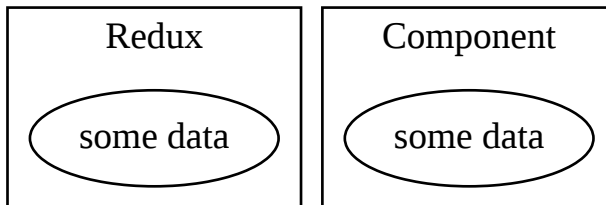
An advantage of this is that it **allows full time travel** and logging.

10-5-3

A disadvantage is that it now means you must be very careful not to **mix UI and app state**. They must be clearly separated within the state object.

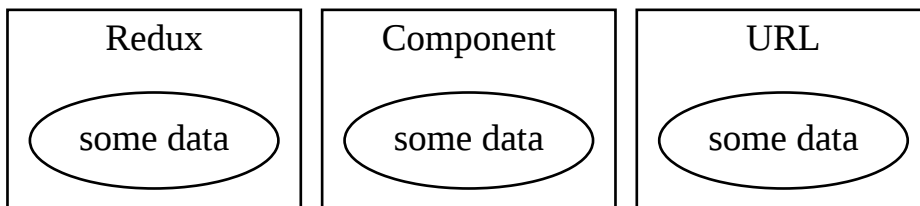
- (b) If we do allow for **some state inside components**, the previous truths are flipped - we **can no longer do (full) time travel**, but app and UI state separation is free.

10-5-4



- (c) Finally, if you have a router with URL parameters, you must be wary of the fact that those **URL parameters are state**.

10-5-5



There are libraries to **mirror parameters into Redux**. The best option today is probably [React-Router-Redux](#), which has been semi-adopted by React-Router and will support the new v4 router.

10-5-6

Advanced React

The final final boss

As a **final challenge**, let's look at **leveling up our React fu** even further!

11-0-1

Sections in this chapter:

1. setState take 2
2. The React Context API
3. Context in React-Redux
4. Firebase
5. Recompose
6. Isomorphic apps
7. Performance
8. Working with the DOM

11-1. setState take 2

First blood 2

There's something we didn't tell you about setState; React will **batch all calls** that happen in the same tick.

11-1-1

In other words, **setState** is asynchronous.



In other words - **what will this handler do?**

11-1-2

```
increaseTwice() {  
  this.setState({counter: this.state.counter + 1});  
  this.setState({counter: this.state.counter + 1});  
}
```


Ⓐ It will actually just **increase by 1**.

11-1-3

See it happen in the [StateFail](#) demo.

So, how can we **safely make several state update calls in the same tick?**

11-1-4

The `setState` method offers **two special syntaxes for this**:

- Ⓐ callback
- Ⓑ transactional

Ⓐ We can **pass a callback as a second argument**. This will only be invoked **after the update is complete**:

11-1-5

```
increaseTwice() {  
  this.setState({counter: this.state.counter + 1}, () => {  
    this.setState({counter: this.state.counter + 1});  
  });  
}
```

We use this in the [StateFailFixCallback](#) demo.

Ⓑ We can also provide a **function instead of an object**. That function will be **passed the current state**, and expected to **return a change object**:

11-1-6

```
increaseTwice() {  
  this.setState(state => ({counter: state.counter + 1}));  
  this.setState(state => ({counter: state.counter + 1}));  
}
```

Even though the calls are batched together and performed in the same tick, the function will be passed the **previously updated state** and it will work as expected.

11-1-7

See it happen in the [StateFailFixFunc](#) demo.

This last pattern is frequently used for **transactional updates to a database**, from which you might recognize it.

11-1-8

11-2. The React Context API

low-level globality

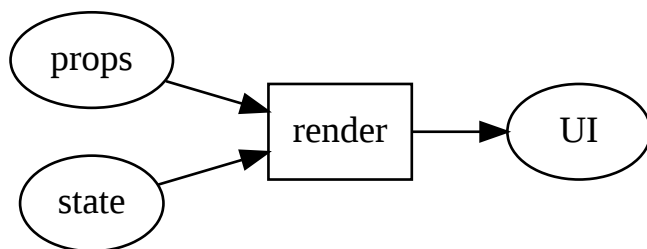
Remember how we first said that a **React component UI** was the **result of its props**...

11-2-1



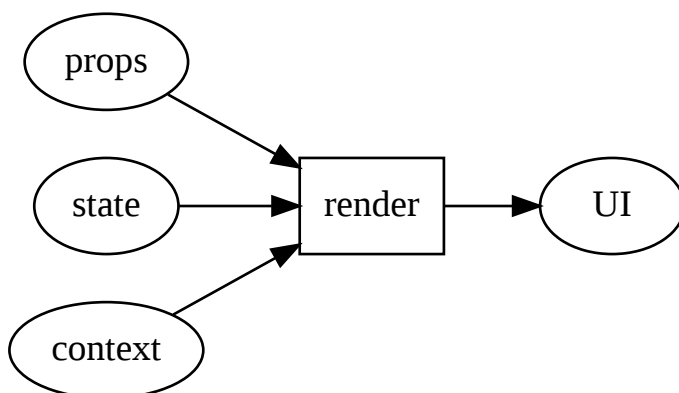
...and then admitted that we lied, and that the UI is in fact derived from **props and state**?

11-2-2



That was actually a lie too - the UI is derive from props, state **and context**:

11-2-3



So here's the proper **signatures for the life cycle methods**:

11-2-4

```
componentWillReceiveProps(nextProps, nextContext)
```

```
shouldComponentUpdate(nextProps, nextState, nextContext)
```

```
componentWillUpdate(nextProps, nextState, nextContext)
```

```
componentDidUpdate(prevProps, prevState, prevContext)
```

There are **very few times** when you actually **need to use context**. It corresponds to **global state**, which is mostly a **bad idea**.

11-2-5

A hint to that is that up until recently the **API wasn't documented** at all!

So, **what then is the point** of context?

11-2-6

Context is mostly **used in libraries**. Later we'll peek at how React-Redux uses context to do their magic, but first we should explore the API!

Ok, enough of the why - on to the how! What does the **API** for Context look like?

11-2-7

The **concept behind Context** builds on the idea of **providers** and **consumers**:

11-2-8

- a A **provider** component can **provide** context...
- b which **consumer** components positioned **further up the tree** can **consume**.

Thus providers are often found at the **root** of the app.

- a We'll first look at how to **define a Provider**. To provide context to **downstream components**, a **Provider must do two things**:

11-2-9

- Define the **shape of the context** in `childContextTypes`, using the same syntax as for `PropTypes` definitions.
- Define a `getChildContext` method which should **return a context matching that shape**.

Here is an **example Provider** which will expose a title to all children as `this.context.title`:

11-2-10

```
class ContextProvider extends React.Component {
  static get childContextTypes() {
    return {
      title: React.PropTypes.string
    };
  }
  getChildContext() {
    return {title: this.props.headline};
  }
  render(){
    return this.props.children;
  }
}
```

The **childContextTypes** definition object is served as a **static property**, exactly like `propTypes` and `defaultProps`.

11-2-11

- ⓑ And now the **Consumers**! Since we **don't want to spam every component** with context provided by upstream providers, Consumers have to **actively opt in** to gain access to the context data.

11-2-12

This is done through defining `contextTypes`, again in the same way as we did for `propTypes`.

Here is an **example Consumer** to catch the context from our previous Provider:

11-2-13

```
let ContextConsumer = (props, context) => <h4>{context.title}</h4>;

ContextConsumer.contextTypes = {
  title: React.PropTypes.string
};
```

Our example components could be rendered something like this:

11-2-14

```
<Provider title="Of global importance">
  <div>
    <div>
      <Consumer/>
    </div>
  </div>
</Provider>
```

The important thing is that the **Consumer must be positioned inside the Provider**.

You can **play around** with the above example in the [Context](#) demo.

11-2-15

More details are in the **official documentation** for the Context API: [Context](#)

11-2-16

As said before, **Context is a bit experimental** (even though it is now officially documented).

11-2-17

See this [Context blog post](#) for a closer look as to **why context is dangerous**.

A final note; context is an **excellent place for style objects** if you are using inline styles as in the [Style](#) demo!

11-2-18

11-3. Context in React-Redux

A real-life case study

We'll now walk through implementing a **home-made version of React-Redux**, in order to understand how it works.

11-3-1

In other words, we'll make our own **Provider component** and **connect function**!

As you might suspect, the top-level **Provider** that we fed the **store** simply **exposes this to the context**:

11-3-2

```
class Provider extends React.Component {
  static get childContextTypes(){
    return { store: React.PropTypes.object };
  }
  getChildContext(){
    return { store: this.props.store };
  }
  render(){
    return this.props.children;
  }
}
```

That was easy enough - connect will be slightly harder!

11-3-3

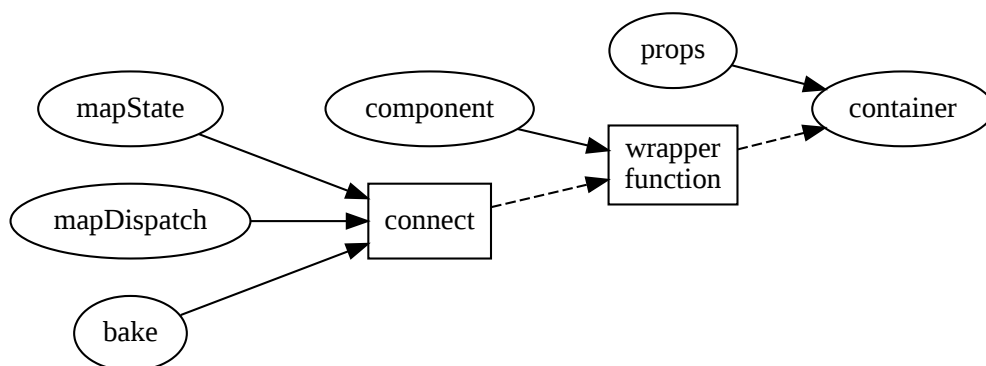
Remember;

11-3-4

1. the **connect** function is called with **mapState**, **mapDispatch** and **bakeProps**.
2. That **returns** a **wrapper function** which is called with the component we want to wrap...
3. ...which finally **returns** the generated **container class** which will be called with some props from the parent.

Here's a **diagram** of this flow:

11-3-5



Here's the **same chain in code format**:

11-3-6

```
let connect = (mapState, mapDispatch, bake) => Component => class Container extends
  // container definition here
}
```

We need to do a **number of things** in there;

11-3-7

- a catch store from the context
- b subscribe to store data
- c render the child

- a First off, the definition will **use the Context API** to fetch a **reference to the store** passed down from the **Provider**:

11-3-8

```
class Container extends React.Component {  
  static get contextTypes: { store: React.PropTypes.object },  
  
  // rest of definition can now access `this.context.store`  
}
```

Side note; you've now seen **all Context-related code**. But we'll follow through anyway since that will help **dispel eventual magic** and **further our understanding of React itself**!

11-3-9

- b We'll use the **same approach** as in our **vanilla Redux integration** - we make our wrapper **save the store data in its state** every time the store is updated:

11-3-10

```
class Container extends React.Component {  
  constructor(props){  
    super(props);  
    this.state = {childprops:{}};  
  }  
  componentDidMount() {  
    let store = this.context.store;  
    let callback = function(){ ... }; // <-- shown soon  
    store.subscribe(callback);  
    callback();  
  }  
  // rest removed  
}
```

Here's the **callback** we run when the store updates!

11-3-11

```
let callback = () => {
  let childprops = bake(
    mapState(store.getState()),
    mapDispatch(store.dispatch),
    this.props
  );
  this.setState({childprops:childprops});
};
```

It simply **bakes the new child props** and **saves them to state!**

- Ⓒ Now we can simply **pass on** `this.state.childprops` as properties to 'Component' in the render method:

11-3-12

```
class Container extends React.Component {
  // ...rest removed ...
  render() {
    return <Component {...this.state.childprops} />;
  }
}
```

Now we've **fully implemented React-Redux!**

11-3-13

Well, except that we **always** expect connect to be **called like this:**

```
connect(mapStateFunction,mapDispatchFunction,bakeFunction)(Component);
```

You can **try out** this homemade React-Redux version applied to our old quotes app in the [Homemade](#) demo.

11-3-14

11-4. Firebase

Live updates from the cloud



Firestore is a **cloud-based database**. It stores data in a **JSON format** and offers **realtime updates**, which makes it an **ideal match with React!**

The **basic idea is simple**: After having **set up our database** at the **Firestore homepage**, we use their web lib to **initialize a local connection**:

11-4-2

```
firebase.initializeApp({
  apiKey: "<mysecretkey>",
  databaseURL: "https://<databasename>.firebaseio.com"
});
```

We then **create a reference** to a spot in our database. Given data that looks like this:

11-4-3

```
{
  blog: {
    comments: {
      // lots of stuff
    }
  },
  // and below lots of other stuffs
}
```

...we could connect to `blog.comments` like this:

```
let ref = firebase.database().ref("blog/comments")
```

With that reference we can **add a change listener**:

11-4-4

```
DB.on('value', snapshot => {
  let data = snapshot.val();
  // now do something with `data`
});
```

Now our **callback will be called whenever new data is pushed from the Firestore servers**.

We can also **add new data** at the reference point:

11-4-5

```
ref.push({  
  author: 'John Doe',  
  msg: 'What trickery is this?!'  
});
```

And of course there are similar functionality for **editing** and **deleting**.

As previously hinted, **Firestore is a very good fit for React**. To highlight this, see how easy it is to build a **chat application** in the [Firestore](#) demo.

11-4-6

There is also a **more advanced version**, [Firestore2](#), which lets you pick a username and chat in different rooms.

A final warning: after **Google bought Firestore** in the **spring of 2016** there's been some **major API updates**, so be aware that material found online **might be out of date**.

11-4-7

11-5. Recompose

A React utility belt

[Recompose](#) is branded as a "lodash but for React".

11-5-1

The primary use case is to **transform components defined as plain functions**.

The end result is that you get **advanced components** that **would normally require the class** syntax, but without abandoning the convenient plain functions.

11-5-2

As an example, take a look at this **simple Clicker demo app**:

11-5-3

```
class Clicker extends React.Component {
  constructor (props) {
    super(props);
    this.state = {count: 3};
    this.more = () => this.setState({count: this.state.count + 1});
  }
  render () {
    return (
      <div>
        <p>{this.state.count} bottles of beer on the wall</p>
        <button onClick={this.more}>Buy more</button>
      </div>
    );
  }
}
```

This can be **expressed in Recompose** like this instead:

11-5-4

```
let enhance = withState('count', 'more', 3);
let Clicker = enhance(props => (
  <div>
    <p>{props.count} bottles of beer on the wall</p>
    <button onClick={() => props.more(n => n + 1)}>Buy more</button>
  </div>
));
```

As another example, let's say we have a **User** plain function component with a **very expensive render**:

11-5-5

```
let User = props => <div>User: {props.name}</div>;
```

Normally, to make sure User doesn't rerender when non-vital props are changed, we would have to use a class and implement **shouldComponentUpdate**.

11-5-6

But using **Recompose** we can **instead do this**:

11-5-7

```
let PerformantUser = Recompose.onlyUpdateForKeys(['name'])(User);
```

Although the "double invocation" can look weird the **functionality is clear**, and we can keep the clean definition of User.

The usefulness of Recompose can be discussed, but if nothing else, using it will **improve your understanding** for how components are defined!

11-5-8

There are some more examples in the [Recompose](#) demo.

11-6. Isomorphic apps

because buzzword

First off; by **isomorphic apps**, or **universal apps**, we mean apps that **can be rendered outside the browser**.

11-6-1

Specifically this often means **rendering on the server**, which means that the **initial rendering is much faster**.

With React this is easy, since **rendering outputs a virtual tree** describing the UI.

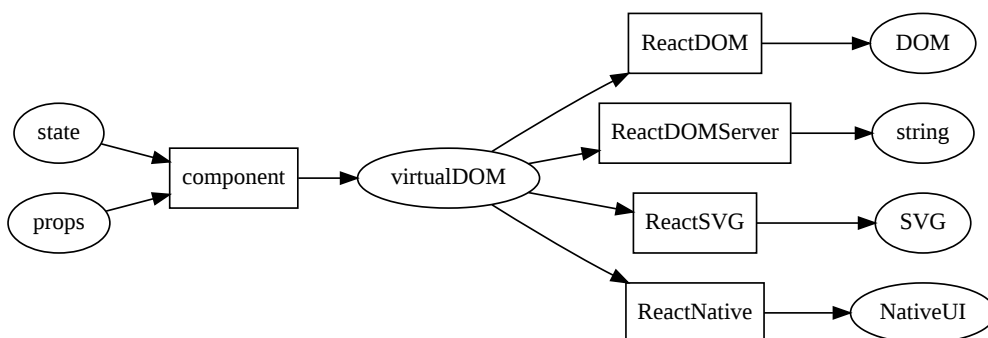
11-6-2

This tree can then be **digested by whatever system**. So far we've usually used **ReactDOM**, which takes the virtual tree and created DOM nodes:



But we can also use **other rendering targets**:

11-6-3



As you can see, [ReactDOMServer](#) outputs a **string**, which makes it very useful on the server.

Building on the idea of being able to **render on the server** is [NextJS](#), a very interesting mini-framework for **isomorphic apps** built on top of React.

11-6-4

The creator, Guillermo Rauch, made a [good overview of NextJS](#) at ReactConf 2017.

11-7. Performance

because the fastest app wins

Let's go over some libraries and concepts that **help improve React performance!**

11-7-1

- ☐ a) the **shouldComponentUpdate** method
- ☐ b) the **React-Perimeter** library
- ☐ c) the **React-virtualized** library
- ☐ d) the **React Fiber** rewrite
- ☐ e) the **React-Canvas** library

- ☐ a) We've already mentioned **shouldComponentUpdate** as a **lifecycle method**, which lets us **short-circuit the rerender**.

11-7-2

But we repeat it here since it is the **single most important thing** you can do to improve app performance.

The logic behind when to short-circuit is **very context specific**, but React contains a ready-made [PureComponent class to inherit from](#) (or a [PureRender mixin](#) if you're not using classes) that implements a **good default**: it returns false if state and props are the same.

11-7-3

Note that the **comparison is shallow**, so for complex data structures we **might need additional logic**.

11-7-4

Also consider using something like [ImmutableJS](#) which **helps make identical objects have the same reference**.

This **saves you from having to do recursive comparisons**, and can be a significant performance boost.

- b) The [React-Perimeter library](#) can further help us to increase perceived performance. 11-7-5
- It lets us **execute logic** when the **cursor breaches the perimeter** of a given component.
- This can be used for a lot of things, but it is especially interesting for **performance enhancements**.
- Here's a **simple example** from the official page: 11-7-6
- ```
import perimeter from 'react-perimeter'

const LoadMoreButton = (
 <Perimeter
 onBreach={this.prefetch}
 padding={80}>
 <button onClick={this.fetch}>Load more</button>
 </Perimeter>
);
```
- There is also a [Perimeter](#) demo. 11-7-7
- Do note how React-Perimeter is a **prime example of the power of the component abstraction!**
- c) The [React-Virtualized library](#) contains **ready-made components** for dealing with **huge lists**, which is a common performance bottleneck. 11-7-8
- It accomplishes this through **windowing**, in essence not rendering stuff that isn't on screen.
- d) For React v16, the team did a **complete rewrite of the core reconciliation engine**. This rewrite was called **React Fiber**, and is in essence an **implementation of the call stack in a render sense**. 11-7-9
- We don't have to change a thing to capitalize on this - just by **upgrading to v16**, everything will be slightly faster!
- The inner workings of the Fiber rewrite are rather fascinating - for an entrypoint, see [Lin Clark's talk](#) from ReactConf 2017. 11-7-10

- e The [React-Canvas](#) library is really just **another rendering target**, much like React-SVG and the others we mentioned previously.

11-7-11

However, for **complex components**, using a canvas element can mean a **performance boost**.

11-7-12

This is not an app-wide decision - you can **render a particularly heavy component onto a canvas**, but leave the surrounding app as a regular DOM app.

## 11-8. Working with the DOM

*DOM DOM DOM*

We've already touched on it several times during the course: since **React wants ownership of the DOM**, it can be awkward when **others want to play with the DOM too**.

11-8-1

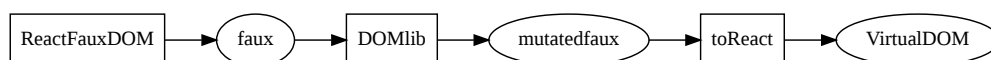
This can be **especially problematic** when you want to **include DOM-operating libraries in a React app**.

11-8-2

For example, the [D3 animation library](#), which expects to be fed a DOM element to render the beautiful graphs into. How do we fit this into React?

A good, general case solution is to use [React-faux-DOM](#). It **creates a faux DOM element** which you can give to a DOM manipulation library.

11-8-3



The fake node has a `.toReact` method which **converts it to Virtual DOM**, ready to be rendered by React!

# Appendix - ES6 features

*The nitty gritty*

---

## Sections in this chapter:

1. Versatile object definitions
2. Destructuring and rest
3. Versatile function definitions
4. Spreads
5. Modules
6. Classes
7. Decorators
8. Miscellaneous
9. Exercise - trying it out

### 12-1. Versatile object definitions

*defining objects like a boss*

In ES2015 we got five small but nice features for **defining objects in a smoother way**:

12-1-1

- a dynamic keys
- b automatic same-key-value
- c method shorthand
- d getters
- e setters

- a If we wanted to create an **object with a dynamic key** we had to go about it in a roundabout way before:

12-1-2

```
var obj = {};
obj[dynamicKey] = someValue;
```



Now, instead, we can use the **dynamic key syntax** by wrapping it in brackets:

12-1-3

```
var obj = {[dynamicKey]: someValue};
```

- (b) Also, if our value is in a **variable with the same name as the intended key**, like here:

12-1-4

```
var person = {
 name: name,
 age: age
};
```

...ES2015 introduces a **shorthand syntax**:

12-1-5

```
var person = {name, age};
```

- (c) And if we define an **object with a method**:

12-1-6

```
var obj = {
 method: function(arg1,arg2){
 // do advanced stuff
 }
};
```

...ES2015 lets us be less verbose by using the **method shorthand syntax**:

12-1-7

```
var obj = {
 method(arg1,arg2){
 // do advanced stuff
 }
};
```

This can also be **combined with the dynamic key syntax**:

12-1-8

```
var obj = {
 [methodName](arg1,arg2){
 // do advanced stuff
 }
};
```

④ Finally, ES2015 also introduced **getters and setters**.

12-1-9

Let's look at **getters** first. They are very useful for dealing with **computed properties**.

Say we're working with **user objects** like this:

12-1-10

```
var user = {
 firstName: "John",
 lastName: "Doe"
};
```

Now we want to implement a **computed property fullName**.

Here's an **ES3 solution** doing it as a **method**:

12-1-11

```
var user = {
 firstName: "John",
 lastName: "Doe",
 fullName: function(){
 return this.firstName + ' ' + this.lastName;
 }
}

user.fullName(); // John Doe
```

By using an **ES2015 getter** we can access the computed property normally instead:

12-1-12

```
var user = {
 firstName: "John",
 lastName: "Doe",
 get fullName() {
 return this.firstName + ' ' + this.lastName;
 }
}

user.fullName; // John Doe, without invocation!
```

Cf. [uniform access principle](#).

ⓔ A **setter** let's you **act upon prop mutation**, for example **logging**...

12-1-13

```
var user = {
 set userName(str) {
 log(this._userName + " changed name to " + str);
 this._userName = str;
 }
}

user.userName = "Steve"; // Bob changed name to Steve
```

...or **validation**:

12-1-14

```
var user = {
 set userName(str) {
 if (str.match(/^[^a-z]/)){
 throw "Name can only contain lowercase letters!";
 }
 this._userName = str;
 }
}

user.userName = "Bob the 1 and only"; // Name can only contain..
```

Ⓚ Did you note that we used a **different property name** inside the setter?  
The setter was for `userName`, but inside it we instead set `_userName`.

12-1-15

Why do you think that is?

Ⓐ If we mutated the same property inside the setter then that would trigger the setter to be called, which would mutate the property, which would trigger the setter, etc. We would end up in an **infinite loop**.

12-1-16

## 12-2. Destructuring and rest

*cherry-picking the raisins from the cookie*

**Destructuring** is a way to pick values from nested structures without having to do the manual digging.

12-2-1

Let's say we have an **array of contenders**, each represented by an object.

12-2-2

```
let contenders = [
 {name: "David", age: 37},
 {name: "Carl", age: 38}
 /* and a few others */
];
```

They are **sorted by position** so the first contender won, etc.

If we wanted the **name of the winner** we would do something like this in ES5:

12-2-3

```
let winnersName = contenders[0].name;
```

With **destructuring**, we can instead do this:

12-2-4

```
let [{name: winnersName}] = contenders;
```

Or, combined with the **same-key-value shorthand**:

```
let [{name}] = contenders;
```

Destructuring also allows us to use the powerful **rest** element which can **lump up many array elements into one**, making for some very succinct code:

12-2-5

```
let [winner, ...losers] = contenders;
```

Note that the rest element **has to be the last one in the array**, so this wouldn't work:

12-2-6

```
let [...others, superloser] = contenders; // syntax error
```



Wait.. Theoretically, **the rest could be placed *anywhere***, as long as there's just one. The parser should still be able to figure out what's what!

12-2-7

Right?

- Ⓐ True. But that would **require lookahead**, which is **complex and more taxing**. And so the choice was made to only allow the rest element in the last position.

12-2-8

### 12-3. Versatile function definitions

*defining function like a boss*

ES2015 provides **several neat features for defining functions**:

12-3-1

- Ⓐ default parameter values
- Ⓑ rest parameters
- Ⓒ destructuring parameters
- Ⓓ arrow functions

- Ⓐ **Default parameter values** exist in many languages, and was popularized in JS through [CoffeeScript](#).

12-3-2

The idea is to **handle optional parameters** in a smoother way.

Creating a **function with an optional parameter** in ES3 meant we had to do a sometimes tedious dance of initialization:

12-3-3

```
function makePerson(name, age) {
 var age = age || 'unknown';
 // do complex stuff
}
```

With **default parameter values** we can instead do this:

12-3-4

```
function makePerson(name, age = 'unknown') {
 // do complex stuff
}
```

- ② The second new feature, **rest parameters**, is a way of capturing multiple arguments into a single variable like a rest element in a destructuring. 12-3-5

This can often save us from having to do awkward stuff with the not-quite-an-array arguments object.

Imagine a competition function that is called with all contenders one by one: 12-3-6

```
function competition() {
 var contenders = Array.prototype.slice.call(arguments);
 var winner = contenders[0];
 var losers = contenders.slice(1);
 // do something with winner and losers
}
```

Using rest parameters, this function simply becomes: 12-3-7

```
function competition(winner, ...losers) {
 // do something with winner and losers
}
```

Note that the rest parameter **has to be the last parameter**, just like the rest element, and for the same reason. 12-3-8

- ③ Remember **destructuring**? We can use that in signatures: 12-3-9

```
function introduce({name, age}) {
 console.log(name, "is", age, "years old");
}
var me = {name: "David", age: 37};
introduce(me); // David is 37 years old
```

- ④ Finally - know how **defining anonymous functions** in JS is rather verbose? 12-3-10

```
var mcboatify = function(arg) {
 return "Boaty Mc"+arg+"Face";
};
```

With **arrow functions** things feel less heavy:

12-3-11

```
var mcboatify = (arg) => {
 return "Boaty Mc" + arg + "Face";
};
```

They can become smaller still - if we have **exactly one parameter**, we can **omit the parenthesis** in the signature:

12-3-12

```
var mcboatify = arg => {
 return "Boaty Mc" + arg + "Face";
};
```

Finally, if you just want to return an expression, we can **skip brackets and the return keyword**:

12-3-13

```
var mcboatify = arg => "Boaty Mc" + arg + "Face";
```

Now the function body consists of a single expression, which will be implicitly returned.

Note however that if you want to use the **single expression form with an object literal**, we have to **wrap it in parenthesis** to distinguish it from a regular function block:

12-3-14

```
var createUser = (name,age)=> ({name,age})
```

Arrow functions are not only less heavy to write, they are also lighter for the interpreter since they **don't get an implicit context parameter**.

12-3-15

Which means that if you refer to this inside an arrow function, it is the **same this as on the outside**.

12-3-16

```
var me = this;
setTimeout(() => {
 console.log(this === me); // true
}, 10);
setTimeout(function(){
 console.log(this === me); // false
}, 10);
```

As a final note; arrow functions can beautifully describe the flow for **nested higher order callbacks**. Take this convoluted old code:

12-3-17

```
function multiplier(func,times){
 return function(){
 for(var i = 0; i < times; i = i + 1){
 func();
 }
 };
}
```

With arrow functions, that becomes:

12-3-18

```
var multiplier = (func,times)=> ()=> {
 for(var i = 0; i < times; i = i + 1){
 func();
 }
}
```

## 12-4. Spreads

*the dark side of rests*

You have already seen how we use **rest** element/parameter to capture several array elements into a single variable:

12-4-1

```
var [winner, ...losers] = competitors;
```

Now imagine **the opposite scenario** - we have the winner and losers variables, and want to define competitors. In ES3 this is done like this:

12-4-2

```
var competitors = [winner].concat(losers);
```



ES2015 gives us a new options - **spreads**! It looks exactly like rest, but we use it on the *right side* instead (or when we *call* a function as opposed to when we define it):

12-4-3

```
var competitors = [winner, ...losers];
```

We say that we *spread* the contents of the expression into the outer array.

Spreads gives us a less verbose way to **copy an object and add properties to it**, which is otherwise done like this:

12-4-4

```
var augmentedObj = Object.assign({}, oldObj, newProps);
```

With spreads we can instead do this:

12-4-5

```
var augmentedObj = {...oldObj, ...newProps};
```

Note that while spreads and rests *with arrays* are in the spec for ES2015, **object spread is still a Stage 3 proposal** (November 2017).

12-4-6

It is **expected to be accepted into an upcoming release** of the language, and is already supported by Babel and the like.

## 12-5. Modules

*getting into the import/export business*

**Node gave us modules** through the `require` and `module.exports` globals it provides.

12-5-1

But with ES2015, we got **native modules** for the very first time!

While **Node modules** followed the **CommonJS module standard**, what was implemented in the language follows **another syntax**, named **ES modules**.

12-5-2

But the **concepts are the same**. While you would do this in **CommonJS**...

```
// file1.js
module.exports = {...};

//file2.js
var lib = require("./file1.js");
```

...you would do this with **ES modules**:

12-5-3

```
// file1.js
export const lib = {...};

//file2.js
import lib from './file1.js'
```

We have to **name our exports** here, otherwise things are **pretty similar**.

There are **other differences too**, so for the full scope you should **check the MDN docs** for **import** and **export**

12-5-4

Note that even though this is now **part of the language**, there are **no browsers that implement the functionality yet**.

12-5-5

This is mainly because it **wouldn't be practical** - we'd get a **gazillion http requests for small files**.

And since we **likely have a build step anyway** to do minification and transpiling and similar, you can easily **bundle your code into a single file**, too.

12-5-6

But, with the advent of HTTP2, **who knows what the future will hold!**

## 12-6. Classes

*Waiter, there are classes in my JS!*

Before ES2015, JavaScript used to famously **lack classes**.

12-6-1

This was **not an oversight**. Consider what **classes are normally used for**:

- **reusing functionality** and
- **setting up hierarchies**

In **JavaScript** this is addressed by

12-6-2

- simply **grabbing methods** and/or **mixing objects**
- **prototypal "inheritance"**, which should really be called delegation

This means that **classes didn't really serve a purpose**. Yet they were **still frequently used**, through the weird, bolted-on **new** syntax which **makes functions behave like constructors**:

12-6-3

```
var user = new User("David", 1979);
```

But to really make this **behave like normal classes**...

12-6-4

```
var lucas = new Dog("Lucas");
lucas instanceof Dog; // true
lucas instanceof Animal; // true
lucas.bark(); // Lucas goes woof!
```

...then lots of **jumping through hoops** had to be done:

12-6-5

```
Dog.prototype = new Animal();
Dog.prototype.constructor = Animal;
Dog.prototype.bark = function() {
 console.log(this.name, "goes woof!");
}
```

To facilitate "class" use in JavaScript, ES2015 introduced the **class** syntax:

12-6-6

```
class Dog extends Animal {
 bark() {
 console.log(this.name, "goes woof!");
 }
}
```

Note how **method shorthands** are available in class declarations too!

But it is important to note that this does **not mean that JavaScript has actual classes**.

12-6-7

Under the hood the same weird prototype and constructor dance happens.

Still, since the **syntax hides the mismatch**, it can be a **convenient way to package functionality**. Specifically:

12-6-8

- a constructor
- b methods
- c properties

a First off, what **used to go in the fake constructor**...

12-6-9

```
function Animal(name) {
 this.name = name;
}
```

...is now placed in a literal **constructor method** in the class declaration:

12-6-10

```
class Animal {
 constructor(name) {
 this.name = name;
 }
}
```

If you want the **inherited constructor to be invoked too**, you must **do so yourself** with the new **super** keyword:

12-6-11

```
class Dog extends Animal {
 constructor(name) {
 super(name);
 this.nickname = name + 'y boy';
 }
}
```

Since **React components inherit from React.Component**, that means that if we have a constructor in our components we **must always call super**.

12-6-12

**(b)** And you've **already seen methods**:

12-6-13

```
class Dog extends Animal {
 constructor() { ... }
 bark() {
 console.log(this.name, "goes woof!");
 }
}
```

Similar to object methods, **this** (normally) **points to the instance**.

**(c)** Finally, as you saw, **properties are normally initialized in the constructor**:

12-6-14

```
class Animal {
 constructor(name) {
 this.name = name;
 }
}
```

...but when we use **TypeScript** - more on that soon - we can also **initialise properties directly on the class declaration**:

12-6-15

```
class Dog {
 numberOfLegs = 4;
}
```

This is **likely to become a part of JavaScript syntax too**.

So, to recap:

12-6-16

- classes are just a **light syntactic sugar** introduced in ES2015
- we normally **don't need them in JavaScript**
- but they are a **convenient way to bundle related functionality**

## 12-7. Decorators

*dewhatnow?*

The situation around decorators is rather confusing;

12-7-1

- There is a [proposal](#) to add it as a language feature
- There is a slightly different [implementation in TypeScript](#)
- There is the parallel idea in **Reflect**
- There is [disagreement](#) on whether decorators are a good idea at all
- There is (was?) something called [annotations](#) that is sort of the same... yet not

Focusing on the **TypeScript implementation** which is the most common, decorators are a way of decorating a class...

12-7-2

- **declaration**
- **property**
- **getter or setter**
- **method**
- **method parameter**

As a simple example, imagine that we have a **debounce function** that **throttles other functions**:

12-7-3

```
function debounce(fn) {
 // ... create a throttled version of `fn`...
 return throttledFn;
}
```

And then we have a class with a **method that is very expensive** to call:

12-7-4

```
class myClass {
 myExpensiveMethod: function() {
 // lots of heavy lifting here
 }
}
```

**Without decorators** we would do this:

12-7-5

```
class myClass {
 myExpensiveMethod: debounce(function() {
 // lots of heavy lifting here
 })
}
```

**With decorators**, instead, we use the @ syntax:

12-7-6

```
class myClass {
 @debounce
 myExpensiveMethod: function() {
 // lots of heavy lifting here
 }
}
```

The **end result is the same thing**.

We can also have **decorators that take additional arguments**. For instance `debounce` could accept a **minimum number of milliseconds**:

12-7-7

```
class myClass {
 @debounce(300)
 myExpensiveMethod: function() {
 // lots of heavy lifting here
 }
}
```

In other words, **decorators are just a light syntax sugar**.

If you want to **dig deeper into decorators**, check out...

12-7-8

- The [decorator section](#) of the TypeScript handbook
- This [concise and clear explanation](#) with examples and interactive links

## 12-8. Miscellaneous

*odds and ends*

There's three more things worth mentioning:

12-8-1

- a) declaring variables with **let**
- b) declaring variables with **const**
- c) **template strings**

- a) Variables in JavaScript have **functional scope**.

12-8-2

Even if you declare them inside an **if-block in the middle of a function**, the variable is still **visible throughout the entire function**.

So when you write this...

12-8-3

```
function myFunc(arg,lib){
 if (arg === 42){
 var ret = lib.method() + 7;
 return ret;
 }
 // do sth else
}
```

...this is what (conceptually) happens:

12-8-4

```
function myFunc(arg,lib){
 var ret;
 if (arg === 42){
 ret = lib.method() + 7;
 return ret;
 }
 // do sth else
}
```

In other words, the **declaration is hoisted to the top**.



This is generally considered a **design mistake**, and can give rise to **weird bugs**.

12-8-5

ES6 therefore introduces **let** as an alternative to **var** for declaring variables, and the **only difference** is that **let** has block scope.

- ⓑ In most languages there's a way to **define constants**, meaning a **variable that cannot change**.

12-8-6

This is **missing from JavaScript**.

A common "hack" is to **name constants in all capitals**:

12-8-7

```
var SOME_CONST = 42;
```

But this has **no technical significance**, it is just a hint.

ES6 therefore introduces **const** as another alternative to **var**, and the **only difference** is that you **cannot reassign the value**.

12-8-8

```
const answer = 42;
answer = 43; // throws an error
```

- ⓒ Finally, **template strings**!

12-8-9

```
let userTempl = `
 First name: ${user.fname}
 Last name: ${user.lname}
`;
```

As you saw, template strings...

12-8-10

- are **defined inside two backticks**
- can **contain linebreaks**
- allow **interpolation inside \${}**

There's also a **semi-secret way to invoke functions with templates**. Here's an example from [Choo](#):

12-8-11

```
html`
 <main class="app">
 Count: ${state.counter.count}
 <button onclick=${() => send('counter:increment')}>+</button>
 </main>`
```

The **html** function is invoked with the templates and interpolated values.

### 12-9. Exercise - trying it out

*Toe into the water*

As a **light-weight exercise**, let's try some of the ES6 stuff out!

12-9-1

We'll do this in **two different parts**:

12-9-2

- a) writing and **running ES6 code**
- b) checking **how it translates to ES5**

- a) An easy way to **run ES6 code** is by using [es6fiddle.net](#). Write **ES6 code in the left column**, and see the **output to the right**.

12-9-3

CODE	CONSOLE
<pre>1 let square = x =&gt; x * x; 2 let add = (a, b) =&gt; a + b; 3 let pi = () =&gt; 3.1415; 4 5 console.log(square(5)); 6 console.log(add(3, 4)); 7 console.log(pi()); 8</pre>	<pre>25 7 3.1415</pre>

Through a dropdown you can access a number of **ready-made examples** which are a good starting point for experimenting.

12-9-4

b

To see the ES5 equivalent, we'll use [babeljs.io/repl](https://babeljs.io/repl). The right column here shows ES5 translation instead of output.

12-9-5

<pre>1 const add = (a, b) =&gt; a + b; 2 let nums = [5, 4]; 3 console.log(add(...nums));</pre>	<pre>1 "use strict"; 2 3 var add = function add(a, b) { 4   return a + b; 5 }; 6 var nums = [5, 4]; 7 console.log(add.apply(undefined, nums));</pre>
------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------

A good way to get started is to copy the ES6fiddle examples and see what they translate to!

12-9-6

# Appendix - Graveyard

*Where old syntax goes to die*

---

In this appendix chapter you'll find some React syntaxes that have been **deprecated**, for one reason or another.

13-0-1

## Sections in this chapter:

1. `createClass` syntax
2. Declaring props
3. Lifecycle methods
4. React Router v2
5. Mixins

### 13-1. `createClass` syntax

*Pass-down garments*

Remember how we first showed you the **plain function** syntax for defining components...

13-1-1

```
let User = props => <div>Name: {props.name}</div>;

User.propTypes = {
 name: React.PropTypes.string.isRequired
};
```

...and then admitted that there are in fact **two definition syntaxes**?

13-1-2

```
class User extends React.Component {
 render() {
 return <div>Name: {this.props.name}</div>;
 }
 static get propTypes(){
 return {
 name: React.PropTypes.string.isRequired
 }
 }
}
```

We lied again - there are **three syntaxes**. We can also use the `createClass` constructor:

13-1-3

```
let User = React.createClass({
 propTypes: { name: React.PropTypes.string.isRequired },
 render() {
 return <div>Name: {this.props.name}</div>;
 }
});
```

This was used for non-PFC:s before ES6 introduced classes and made the class syntax possible.

It is **pretty similar** to real classes, with a few differences:

13-1-4

- There's no constructor, so we **provide starting state from a `getInitialState` method**
- Methods are **autobound to the instance**. Convenient but can have performance impact
- Some **inconsistencies with static stuff**

See them in the [createClass](#) demo

As alluded to earlier; this syntax was **used before classes were available**.

13-1-5

For a long time **both approaches existed side by side**, but `React.createClass` **was removed from React** and put into a separate module.

## 13-2. Declaring props

*Dear santa, I want...*

React provides components with a way to **declare what props they expect**. This serves two purposes:

13-2-1

- To **safeguard** against faulty use (in dev mode) by **throwing an error** (or give a warning depending on settings) if something is missing or of the wrong type
- To **document** how the component is supposed to be used

This declaration, called **propTypes**, is an **object** with constants describing the needed props:

13-2-2

```
{
 name: React.PropTypes.string.isRequired,
 role: React.PropTypes.oneOf(['admin', 'contributor', 'user'])
}
```

See the **full documentation** of this powerful API here: [Proptypes](#)

These declarations are then added as a **static property** to the class. There are **three different syntaxes** for doing that:

13-2-3

- a** as a prop on the class itself
- b** as a static getter
- c** as a static prop (experimental syntax)

- a** We can add the declaration directly onto the class:

13-2-4

```
class User extends React.Component {
 render() {
 return <div>Name: {this.props.name}</div>;
 }
}
User.propTypes = {
 name: React.PropTypes.string.isRequired
}
```

This **also works for PFC:s** (pure function components).

- ⓑ Alternatively we can return it from a **class method** if we use **static** and **get**:

13-2-5

```
class User extends React.Component {
 render() {
 return <div>Name: {this.props.name}</div>;
 }
 static get propTypes(){
 return {
 name: React.PropTypes.string.isRequired
 }
 }
}
```

- ⓒ There is also a [proposal](#) to allow declaring **static props**:

13-2-6

```
class User extends React.Component {
 render() {
 return <div>Name: {this.props.name}</div>;
 }
 static propTypes = {
 name: React.PropTypes.string.isRequired
 }
}
```

To use this we need a [babel plugin](#) (or Typescript).

We can also provide a **defaultProps** object where the **keys are prop names** and the **values are defaults**:

13-2-7

```
{
 name: 'Anonymous',
 age: 'Unknown'
}
```

The **defaultProps** is **attached as a static prop** exactly like **propTypes**.

Try this all out in the [Proptypes](#) demo.

13-2-8

A final word; the **PropTypes** API is somewhat **falling out of grace**. It used to live inside **react**, but since **v15.5** you have to get it from a **separate module**:

13-2-9

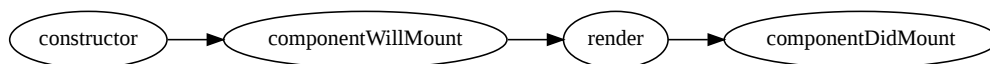
```
import PropTypes from 'prop-types';
```

The reason is simple; **static analyzers with typechecking** have grown in popularity, and if we're using such a solution, it makes more sense to **declare our props using that**.

13-2-10

Most common of these are [Flow](#) and [TypeScript](#).

### 13-3. Lifecycle methods



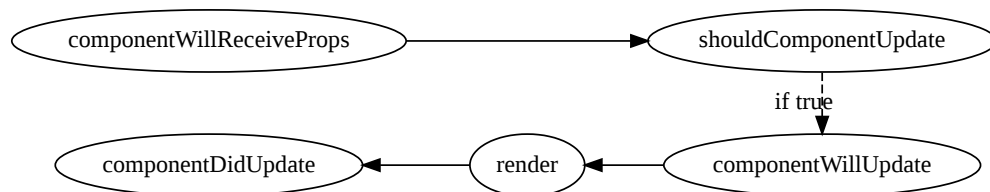
13-3-1

- `componentWillMount()` is being deprecated and will be removed in 17.0
- `componentWillMount()` renamed to `UNSAFE_componentWillMount()`

These changes have been made because `componentWillMount()` has typically been misunderstood and is likely to cause issues with async rendering. Commonly issues with error handling, as well as delaying the rendering of components have been caused by using this method improperly.

Any usages should be replaced by a combination of `constructor()`, and `componentDidMount()`. In fact typically most usages of `componentWillMount` actually would be better placed in the `constructor()` or depending on the case in `componentDidMount()`. Of course it is possible to use `getDerivedStateFromProps()` as well, but this is a bit of an anti-pattern.

See [Update on Async Rendering](#) and [React v16.3 release](#)



13-3-2

- `componentWillReceiveProps()` is being deprecated and will be removed in 17.0
- `componentWillUpdate()` is being deprecated and will be removed in 17.0
- Both are renamed with `UNSAFE_` prefix

These are also being removed as part of the cleanup of poorly used lifecycle methods. They had a tendency to be misused and so React is moving towards avoiding these commonly misused functions. Eventually they'll be dropped completely.



## 13-4. React Router v2

*completing the trinity*

So we introduced **Redux** to deal with the **data** in our **React** app. But, what about **navigation**?

13-4-1

## React-Router-v2

13-4-2

Where are we going?

**SPA** stands for **Single Page Application**. This describes the fact that the page **never reloads** in the traditional sense.

13-4-3

We're dealing with **one single front-end webapp** throughout the session lifetime.

But unless we're building something very simple, we still **need to provide navigation**.

13-4-4

In a SPA we have **two options** for doing so.

**Option 1** is to **handle our own navigation**. Whenever the user clicks something to go to a different view, we catch that event and **repopulate the screen** with whatever the user asked for.

13-4-5

The advantage of this approach is that it is **easy**. We're in **total control**, and navigation added **no external dependencies**.

13-4-6

There's a **huge downside**, however: **browser navigation won't work** in our app, meaning:

13-4-7

- The user **cannot bookmark** a position in the app
- Hitting the **back button** means **leaving the app** altogether

For a serious webapp, these are **dealbreakers**.

13-4-8

Which is why **all frameworks** go with **option 2**, namely to **hook into the browser navigation**. Angular, Ember, Meteor, Aurora - they all have their own **built-in routing** solution.

But **React is not a full framework**. It deals mostly with the view, and doesn't care how you solve the navigation problem.

13-4-9

If you want to hook into the browser navigation, you have to **do it yourself**.

...or, you can include a **companion library** where someone else has already solved the problem! Which is exactly what [React Router](#) is.

13-4-10

Note that we'll be talking about **version 2** of React Router, which is still the most common in the wild. However, there is a **version 4 released** which is **very different!**

13-4-11

Version 3? Never really happened. :)

## Exploring an example

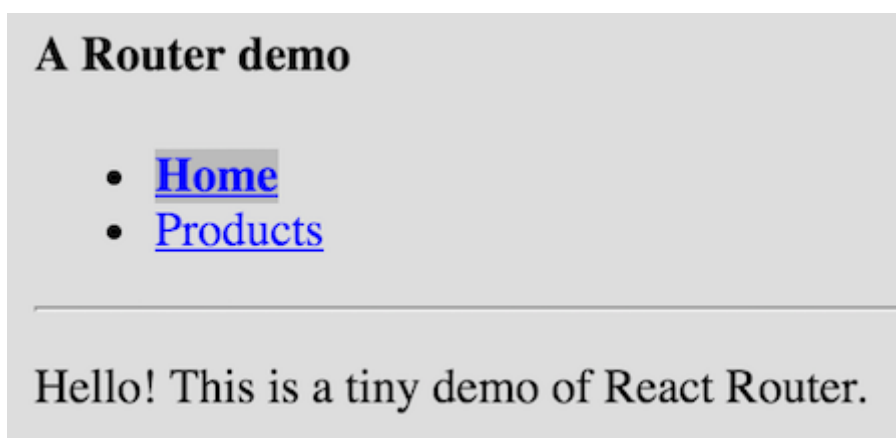
13-4-12

Routing in the real world

Before we peek under the hood, we'll explore a simple example app that uses React Router.

13-4-13

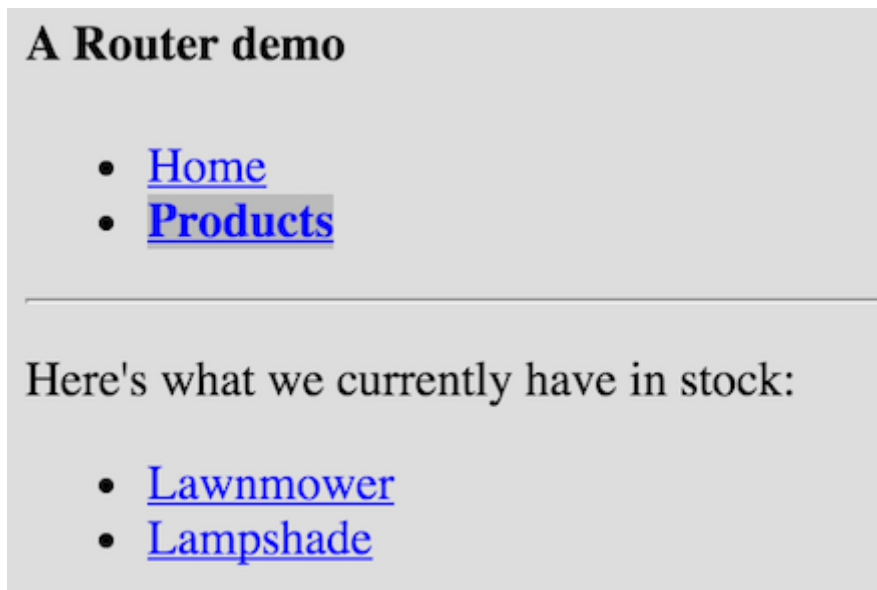
Here's what the **home screen** looks like:



Note how the **home** link in the navbar is active.

Clicking on **products** takes us to this list:

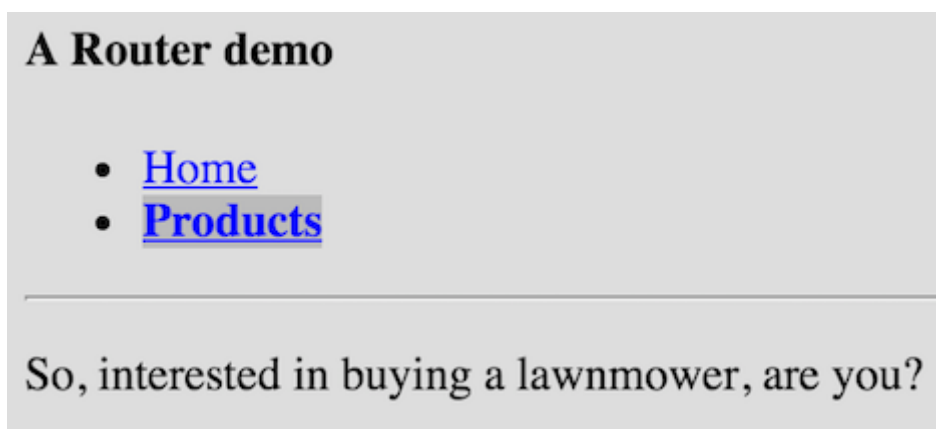
13-4-14



Now the **Products** link in the nav menu is active.

Finally here's the screen for a **specific product**:

13-4-15



Note how the **Products** link is still active, since we're still considered to be in the same section.

As stated a very simple app, yet still enough to catch the most common routing needs.

13-4-16

We'll be walking through it together, but you can try the demo here: [Router](#)

## The route map

13-4-17

Where to go

At the heart of an app using React Router is the **routes definition**.  
Inspired by Ember, it is a **nested declaration of all routes** in your app.

13-4-18

Here's a **conceptual sitemap** of our example:

13-4-19

```
routes
 home
 products
 list
 productitem
```

**Home** is just a single page. **Products** is a section with two pages; a **list** and a **product item**.

Thinking about it, this is a **nested structure** much like the DOM. We've already seen how convenient it is to describe the DOM with the **JSX syntax**, so why not use that for the routes too?

13-4-20

That's exactly the approach React Router takes.

Here's the **routes definition** for our **example app**:

13-4-21

```
const routes = (
 <Route path="/" component={Wrapper}>
 <IndexRoute component={Home} />
 <Route path="/products">
 <IndexRoute component={ProductList} />
 <Route path="/products/:productid" component={Product} />
 </Route>
 </Route>
);
```

Even without fully understanding every detail, we can see that the routes definition serves **three powerful purposes** at the same time:

13-4-22

- **Connecting paths to components**
- **Providing a templating solution**
- **Showing an overview of your entire app**

The rest of the details will hopefully clear when we take a closer look at the **Wrapper**, **index routes**, **links** and **parameters** respectively.

13-4-23

# The Wrapper component

13-4-24

## Router templating

Now we'll take a closer look at the **Wrapper** component, which functions as a **master template** in our app:

13-4-25

```
const routes = (
 <Route path="/" component={Wrapper}>
 // ...lots of sub routes...
 </Route>
)
```

This definition means that **Wrapper** will be rendered for the path **/** and **all child paths** too. If a child path is matched, then the component tied to that route will be **passed as a child to Wrapper**.

Here's the code for **Wrapper**:

13-4-26

```
// Abbreviations: ILink is IndexLink and acn is activeClassName
let Wrapper = props => (
 <div>
 <h4>A Router demo</h4>
 <ul class="nav">
 <ILink to="/" acn="now">Home</ILink>
 <Link to="/products" acn="now">Products</Link>

 <hr/>
 <div class="content">{props.children}</div>
 </div>
)
```

Note how we **render matched child routes** into content.

In the example app we only have one wrapper, but if we wanted to we could **nest wrappers**.

13-4-27

Let's for example say that we want all pages in the **products** section to have a distinct look.

That's as easy as adding a component, let's call it ProductSection, to the /products route:

13-4-28

```
const routes = (
 <Route path="/" component={Wrapper}>
 <IndexRoute component={Home} />
 <Route path="/products" component={ProductSection}>
 <IndexRoute component={ProductList} />
 <Route path="/products/:productid" component={Product} />
 </Route>
 </Route>
)
);
```

And ProductSection wouldn't need to be more complicated than this:

13-4-29

```
let ProductSection = props => (
 <div className="productsection">
 {props.children}
 </div>
)
);
```

In other words, **wrapping routes** and **wrapping components** are just two sides of the same coin!

13-4-30

## Index routes

13-4-31

You say potato

Let's wrap our brains around the difference between a Route and an IndexRoute.

13-4-32

Take a look at our routes again:

13-4-33

```
<Route path="/" component={Wrapper}>
 <IndexRoute component={Home} />
 <Route path="/products" component={ProductSection}>
 <IndexRoute component={ProductList} />
 <Route path="/products/:productid" component={Product} />
 </Route>
</Route>
```

Can you guess what the functions of IndexRoutes are?

An `IndexRoute` is like a **default route** - it is rendered if no sibling route is matched. This means that an `IndexRoute`...

13-4-34

- must be the child of a **Route** with a **path**
- cannot have a **path** of its own
- can never have another **IndexRoute** as a sibling

## Links

13-4-35

Getting a highway pass

We'll now take a look at how to implement **links** in an app using React Router.

13-4-36

As you perhaps noticed in the Wrapper source, we can use the **Link component** from React Router:

13-4-37

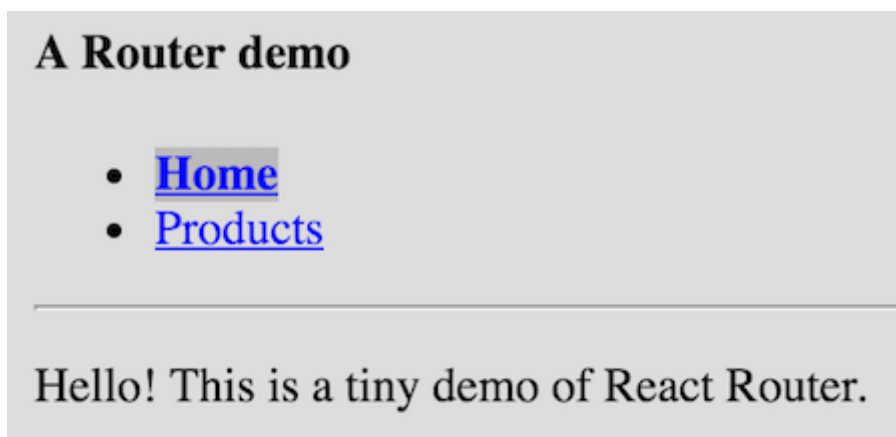
```
let Link = ReactRouter.Link;

let l = <Link to="/about">About</Link>;
```

When clicked, the link `l` will navigate to the route `/about`.

Remember how the navbar links in our example app were **highlighted** if the route they linked to was currently active?

13-4-38



We accomplish that through the `activeClassName` property:

13-4-39

```
<Link to="/about" activeClassName="active">About</Link>
```

This link will be rendered with the CSS class **active** if our current route matches `/about`.

However, this clashes when we link to an `IndexRoute` since **that route will always match**.

13-4-40

That's why we use an `IndexLink` when linking to such a route, to only have it active if we are literally at that route and no other.

Here are the navbar links from the example app demonstrating this:

13-4-41

```
<IndexLink to="/" activeClassName="active">Home</IndexLink>
<Link to="/products" activeClassName="active">Products</Link>
```

## Parameters

13-4-42

There's no party without them!

You saw them flash by in our example app for the **product item page**. Here's that route definition again:

13-4-43

```
<Route path="/products/:productid" component={Product} />
```

The **colon** makes the last part of the path into a **parameter**.

Which means that if we...

13-4-44

- **navigate** to `/products/dishwasher`
- then the route `/products/:productid` will **match**
- and the parameter **productid** will equal **dishwasher**.

Observe that the **colon** is only used in the **routes definition**.

13-4-45

We do **not** use it **in our URL:s** when we navigate or link.



We can see this in action in the **source code for the Product component**:

13-4-46

```
let Product = props => (
 <div>
 <p>Interested in buying a {props.params.productid}?</p>
 </div>
);
```

Note how the **parameters are available** on `props.params`. This is React Router's doing.

## Route configuration

13-4-47

Drawing the map

Now the time has come to zoom out and see how to **configure** and **initialize** all this!

13-4-48

Here's the relevant code from the example app:

13-4-49

```
let Router = ReactRouter.Router,
 hashHistory = ReactRouter.hashHistory;

ReactDOM.render(
 <Router routes={routes} history={hashHistory} />,
 document.getElementById("container")
);
```

As you can see we use a Router component as a **root component**, feeding it our **routes** and a **history** engine.

**Note the difference** between Router and Route!

13-4-50

- A Route represents a **single route**, so we have many of those.
- Router is the **root component**, which receives all Routes as children.

Finally, what about the **history** parameter? That dictates **how** React Router should **hook up paths to the browser**.

13-4-51

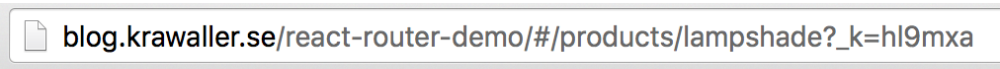
There are 3 different implementations built in;

- **hashHistory**
- **browserHistory**
- **createMemoryHistory**

We'll now take a quick look at each of them.

In the example app we used **hashHistory**. This means that the app **path lives in the hash of the URL**:

13-4-52



The meaningless stuff at the end, `?_k=hl9mxa`, is an unfortunate artifact necessary to reliably track unique state.

If we instead used **browserHistory** then we'd get a clean, "regular" URL with no hashes or artifacts.

13-4-53

But this **requires server-side configuration** to handle the case when the user starts somewhere else other than the root, which is why the example app doesn't use it.

Finally **createMemoryHistory** doesn't connect to the browser at all, but instead **handles the navigation state in memory**. Much like **Option 1** for SPA:s that we mentioned initially as an example of what not to do.

13-4-54

So why would we want to use memoryHistory? Three main reasons:

13-4-55

- For **small apps that are living inside a larger app**, for example a JSBin demo
- For rendering our apps **server side** where there is no browser
- For **testing**

You can read more about the histories, and especially how to configure your server for browser histories, here: [Histories](#)

13-4-56

# Acting on route change

13-4-57

Employing a bouncer

As a final piece to the puzzle, React Router **exposes hooks to act on transitions**. There are three different events:

13-4-58

- **onEnter**: Hmm, you're not on the guest list!
- **onLeave**: Are you SURE you want to leave?
- **onChange**: fired also for query or child route changes

The events are placed as **event listeners on the Route elements** in the config, making us feel right at home!

13-4-59

```
<Route path="/dashboard" onEnter={checkAuth}/>
```

In the function supplied to the hook you can

13-4-60

- do **async** stuff
- **inspect** current / upcoming route data
- **redirect**

As an example, check out the [RouterHook](#) demo where we've added a VIP gatekeeper to our product section!

## 13-5. Mixins

*Mixing it up*

**Mixins** are **objects** with **pre-made functionality** that we can **mix into components**.

13-5-1

This API **only works with the React.createClass syntax**, and not with the newer class components!

We use mixins by supplying them to the **mixins** property in the **component definition**:

13-5-2

```
let Component = React.createClass({
 mixins: [amixin, anothermix, athirdmixin],
 // rest of component follows
});
```

Now **all methods** in the mixins will be available on instances of **Component**.

Let's make a **ticker** mixin which...

13-5-3

- **inits** a **state variable** called **tick** to **0**
- **increases** that variable **every second**

Here's the **code** to accomplish this:

13-5-4

```
let ticker = {
 getInitialState() { return {tick:0}; },
 tick() {
 this.setState({tick:this.state.tick + 1});
 },
 componentDidMount() {
 this._ticker = setInterval(this.tick,1000);
 },
 componentWillUnmount() {
 clearInterval(this._ticker);
 }
};
```

We can now easily make a **Timer** component by **mixing in** ticker:

13-5-5

```
let Timer = React.createClass({
 mixins: [ticker],
 render() {
 return <div>
 Seconds since start: {this.state.tick}
 </div>;
 }
});
```

Try **this** out in the [Mixin](#) demo.



But, hang on - what is **different** from simply **merging the objects ourselves**?

13-5-6

```
let Component = React.createClass(Object.assign(
 {}, amixin, anothermixin, athirdmixin, {
 // rest of component follows
 }));
```



The React **mixin API** takes care of **merging lifecycle methods**.

13-5-7

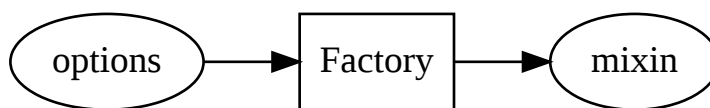
Recall that **ticker** needed to use both **getInitialState** and **componentDidMount**, as well as **componentWillUnmount** for cleanup.

If our component used any of those and we **merged the objects ourselves**, there would be a **clash**!

A **powerful pattern** that allows for **customisable mixins** is to use **mixin factories**.

13-5-8

While a **regular mixin** is an **object**, a **mixin factory** is a **function** that takes **option parameters** and **returns** a corresponding **mixin object**.



As an example, here's a **version of ticker** that let's us **specify the interval**:

13-5-9

```
let MakeTicker = (interval)=> ({
 getInitialState(){ return {tick:0}; },
 tick() {
 this.setState({tick:this.state.tick + 1});
 },
 componentDidMount() {
 this._ticker = setInterval(this.tick,interval);
 },
 componentWillUnmount() {
 clearInterval(this._ticker);
 }
});
```

You can **try** the customisable ticker in the [Mixin Factory](#) demo.

13-5-10

Also, remember the [Lifecycle](#) demo? We remade that as a **more advanced mixin factory demo** called [Mixin Factory 2](#).

It is **rarely useful** to make your own mixins. The **most common use case** is to have **library-supplied mixins** that act as an API bridge for the component.

13-5-11

Dan Abramov makes the same point in [this official React blog post](#) where he **advised against mixin usage**.

13-5-12

# External links

- 10-1-1 Redux Dev Tools: <https://github.com/gaearon/redux-devtools/>
- 10-3-1 Reselect: <https://github.com/reactjs/reselect>
- 10-5-6 React-Router-Redux: <https://github.com/reactjs/react-router-redux>
- 1-1-3 lau\_stephen: [https://twitter.com/Lau\\_Stephen](https://twitter.com/Lau_Stephen)
- 1-4-11 create-react-app: <https://github.com/facebookincubator/create-react-app>
- 1-5-9 here: <https://tc39.github.io/process-document/>
- 1-5-10 Stage 0 list: <https://github.com/tc39/proposals/blob/master/stage-0-proposals.md>
- 1-5-10 Stage 1-3: <https://github.com/tc39/proposals>
- 1-5-10 Stage 4: <https://github.com/tc39/proposals/blob/master/finished-proposals.md>
- 1-5-12 Exploring ES2016 and ES2017: <https://leanpub.com/exploring-es2016-es2017>
- 1-5-12 Exploring ES6: <https://leanpub.com/exploring-es6>
- 1-5-20 Interactive ES6 features overview: <http://es6-features.org/>
- 1-5-20 JavaScript Garden: <http://bonsaiden.github.io/JavaScript-Garden/>
- 1-5-20 MDN: <https://developer.mozilla.org/en-US/>
- 1-5-20 MDN:s reintroduction to JS: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/A\\_re-introduction\\_to\\_JavaScript](https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript)
- 11-2-16 Context: <https://facebook.github.io/react/docs/context.html>
- 11-2-17 Context blog post: <https://medium.com/react-ecosystem/how-to-handle-react-context-a7592dfdcb>
- 11-4-1 Firebase: <https://firebase.google.com/>
- 11-5-1 Recompose: <https://facebook.github.io/react-native/>
- 11-6-3 ReactDOMServer: <https://facebook.github.io/react/docs/react-dom-server.html>
- 11-6-4 NextJS: <https://zeit.co/blog/next>
- 11-6-4 good overview of NextJS: <https://www.youtube.com/watch?v=evaMpdSiZKk>
- 11-7-3 PureComponent class to inherit from: <https://facebook.github.io/react/docs/react-api.html#react.purecomponent>
- 11-7-3 PureRender mixin: <https://facebook.github.io/react/docs/pure-render-mixin.html>
- 11-7-4 ImmutableJS: <https://facebook.github.io/immutable-js/>
- 11-7-5 React-Perimeter library: <https://github.com/aweary/react-perimeter>
- 11-7-10 Lin Clark's talk: <https://www.youtube.com/watch?v=ZCuYPiUIONs&t=6s>
- 11-7-11 React-Canvas: <https://github.com/Flipboard/react-canvas>
- 11-8-2 D3 animation library: <https://d3js.org/>
- 11-8-3 React-faux-DOM: <https://github.com/Olical/react-faux-dom>
- 12-1-7 method shorthand syntax: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Method\\_definitions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Method_definitions)
- 12-1-12 uniform access principle: [https://en.wikipedia.org/wiki/Uniform\\_access\\_principle](https://en.wikipedia.org/wiki/Uniform_access_principle)
- 12-2-1 Destructuring: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring\\_assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment)
- 12-5-4 export: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export>
- 12-5-4 import: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import>
- 12-7-1 annotations: <http://blog.thoughttram.io/angular/2015/05/03/the-difference-between-annotations-and-decorators.html>
- 12-7-1 disagreement: <https://github.com/wycats/javascript-decorators/issues/48>
- 12-7-1 implementation in TypeScript: <https://www.typescriptlang.org/docs/handbook/decorators.html>
- 12-7-1 proposal: <http://tc39.github.io/proposal-decorators/>
- 12-7-8 concise and clear explanation: <https://github.com/arolson101/typescript-decorators>

12-7-8 decorator section: <https://www.typescriptlang.org/docs/handbook/decorators.html>

12-8-11 Choo: <https://github.com/yoshuawuyts/choo>

12-9-3 es6fiddle.net: <http://es6fiddle.net/>

12-9-5 babeljs.io/repl: <https://babeljs.io/repl>

13-1-5 was removed from React: <https://github.com/facebook/react/pull/9232>

13-2-2 Proptypes: <https://facebook.github.io/react/docs/reusable-components.html#prop-validation>

13-2-6 babel plugin: <https://www.npmjs.com/package/babel-plugin-transform-class-properties>

13-2-6 proposal: <https://github.com/tc39/proposal-class-public-fields>

13-2-10 Flow: <https://flow.org/>

13-2-10 TypeScript: <https://www.typescriptlang.org/>

13-3-1 React v16.3 release: <https://reactjs.org/blog/2018/03/29/react-v-16-3.html>

13-3-1 Update on Async Rendering: <https://reactjs.org/blog/2018/03/27/update-on-async-rendering.html>

13-4-10 React Router: <https://github.com/reactjs/react-router>

13-4-56 Histories: <https://github.com/reactjs/react-router/blob/master/docs/guides/Histories.md>

13-5-12 this official React blog post: <https://facebook.github.io/react/blog/2016/07/13/mixins-considered-harmful.html>

2-1-3 Handlebars: <http://handlebarsjs.com>

2-2-1 JSConfEU 2013: <https://www.youtube.com/watch?v=x7cQ3mrcKaY>

2-2-1 JSConfUS 2013: <https://www.youtube.com/watch?v=GW0rj4sNH2w>

2-5-3 Browserify: <http://browserify.org/>

2-5-3 Node: <https://nodejs.org/en/>

2-5-3 Webpack: <https://webpack.github.io/>

2-5-3 npm: <https://www.npmjs.com/>

2-5-5 Babel: <https://babeljs.io/>

2-5-11 CycleJS: <http://cycle.js.org/>

2-6-12 detailed explanation: <https://github.com/acdlite/react-fiber-architecture>

2-9-5 React Devtools: <https://github.com/facebook/react-devtools>

4-1-10 Babel plugin: <https://www.npmjs.com/package/babel-plugin-transform-class-properties>

4-1-10 proposal: <https://github.com/tc39/proposal-class-fields>

4-3-4 Events: <https://facebook.github.io/react/docs/events.html>

4-3-14 AutoBind decorator: <https://www.npmjs.com/package/autobind-decorator>

4-5-3 React v16.3 release: <https://reactjs.org/blog/2018/03/29/react-v-16-3.html>

4-5-3 Update on Async Rendering: <https://reactjs.org/blog/2018/03/27/update-on-async-rendering.html>

4-7-11 Forms: <https://facebook.github.io/react/docs/forms.html>

4-9-7 React-Styleable: <https://github.com/pluralsight/react-styleable>

4-9-8 Aphrodite: <https://github.com/Khan/aphrodite>

5-1-2 Flux: <https://facebook.github.io/flux/>

5-1-5 Redux: <http://redux.js.org/>

5-6-2 Redux: <http://redux.js.org/>

5-6-3 Egghead Redux course: <https://egghead.io/series/getting-started-with-redux>

5-6-4 Learn Redux: <https://learnredux.com/>

5-6-5 Awesome Redux: <https://github.com/xgrommx/awesome-redux>

7-5-1 React-Redux: <https://github.com/reactjs/react-redux>

7-5-11 React-Redux: <https://github.com/reactjs/react-redux>

8-1-6 Redux Dev Tools: <https://github.com/gaearon/redux-devtools/>

8-4-5 Redux-Thunk: <https://github.com/gaearon/redux-thunk>

8-4-5 new API: [https://twitter.com/dan\\_abramov/status/730053481919303685?lang=en](https://twitter.com/dan_abramov/status/730053481919303685?lang=en)

8-5-7 Middlewares: <http://blog.krawaller.se/posts/exploring-redux-middlewares>

9-1-8 React Router: <https://github.com/reactjs/react-router>



# Demo references

chaos	8-2-13
combinedreducers	10-2-8
communication	4-8-6
communication2	4-8-7
communication3	4-8-7
conditionalreturn	3-7-2
conditionalreturn2	3-7-3
conditionalreturnfail	3-7-7
context	11-2-15
controlled	4-7-8
createClass	13-1-4
deaf	8-2-10
defaultprops	4-4-5
errorcatching	4-5-11
events	4-3-15
firebase	11-4-6
firebase2	11-4-6
hellosomeone	2-3-3, 2-9-4
helloworld	2-3-2
homemade	11-3-14
impatient	8-2-12
lifecycle	4-5-8, 13-5-10
mixin	13-5-5
mixinfactory	13-5-10
mixinfactory2	13-5-10
nervous	8-2-11
nestedcomponents	2-7-3, 2-9-1
onlyiftrue	3-7-5
perimeter	11-7-7
petrifiedinput	4-7-4
propschildren	3-8-3
proptypes	13-2-8
reactl6	3-1-6
recompose	11-5-8
redux	6-11-4
reduxdevtools	10-1-3
refbystring	4-6-8
refcomponent	4-6-2
refinput	4-6-7, 4-7-6
refquery	4-6-5
renderlist	3-10-5
reselect	10-3-8
router	13-4-16
routerhook	13-4-60
routernv4	9-2-4, 9-5-2, 9-8-9

routernv4auth	9-8-6
routernv4render	9-8-3
select	4-7-13
shouldcomponentupdate	4-5-10
state	4-2-14, 4-8-7
statefail	11-1-3
statefailfixcallback	11-1-5
statefailfixfunc	11-1-7
style	4-9-4, 11-2-18
styleable	4-9-7
thunk	8-5-6
uncontrolled	4-7-7
usinglib	7-6-11
vanilla	7-4-8
virtualdom	2-6-7