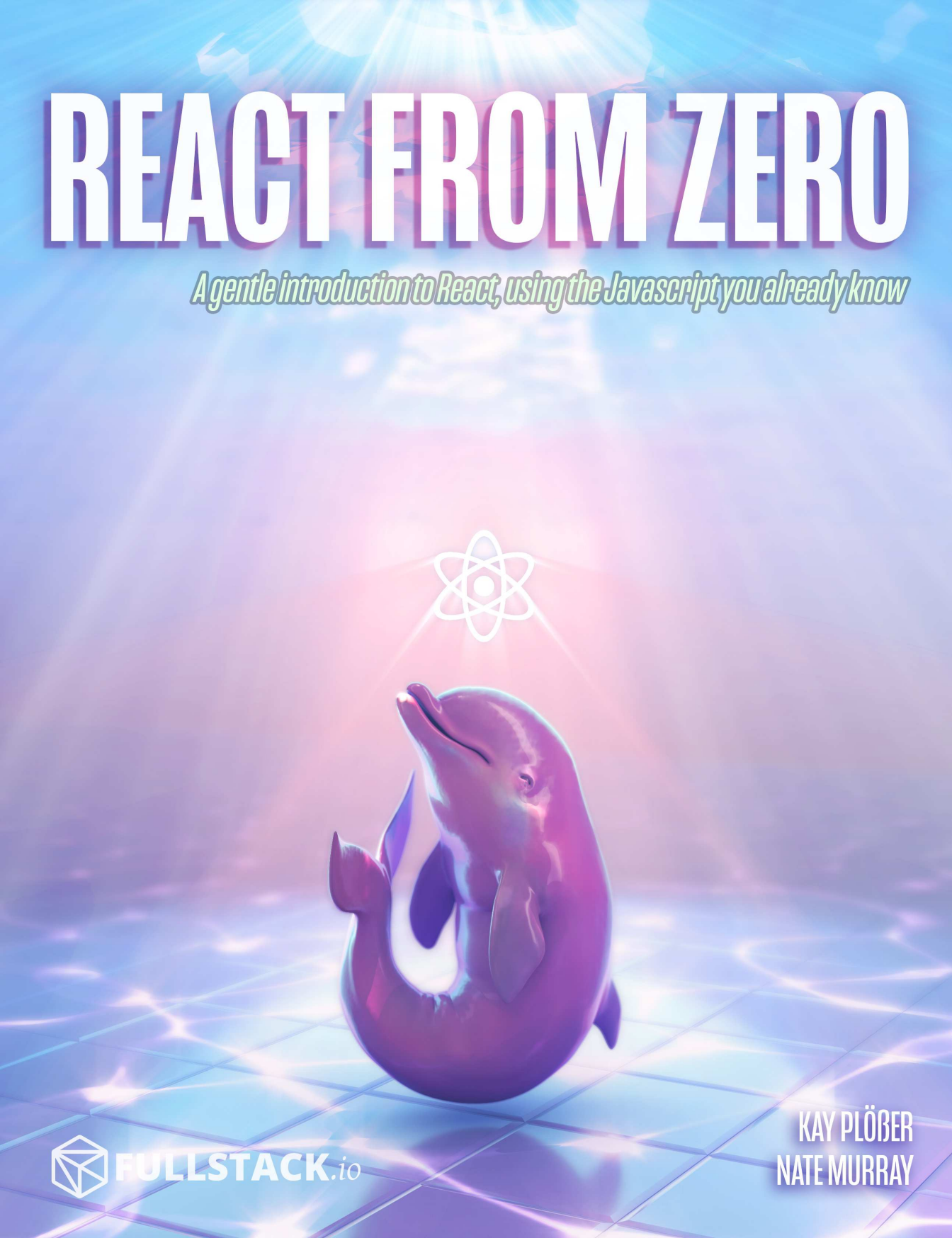
REACT FROM ZERO

A gentle introduction to React, using the Javascript you already know



KAY PLÖßER
NATE MURRAY

FULLSTACK.io

React from Zero

Learn React with the JavaScript you already know

Written by Kay Plößer
Edited by Nate Murray

Published in San Francisco, California by Fullstack.io.

FULLSTACK.io

Contents

Book Revision

Revision 3 - 2018-10-30

Bug Reports

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

Be notified of updates via Twitter

If you'd like to be notified of updates to the book on Twitter, follow us at @fullstackio[1].

We'd love to hear from you!

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

[1] https://twitter.com/fullstackio

How This Book Is Structured

React From Zero is split into two parts, each with multiple chapters. Every chapter tries to teach a single feature of React by presenting code examples and explaining them in detail.

The goal if this book is to demystify React and connect it to "everyday" JavaScript. Unlike many tutorials which use React as a "black box", we're going to start from the bottom and work our way up.

Part I: Basics

Part I of this book is about the **basic usage of React**.

We learn how React works by starting with **JavaScript you already know**.

First we setup a development environment and then we start by digging into the basic JavaScript objects that React components are made of. We'll talk about *why* we use JSX tags when we write React, how to create re-usable *components* and how we can next them to make more complex structures.

In **Part I** we'll teach you the skills that you'll use in 80% of your day-to-day work with React.

Part II: Additions

Part II of this book is about the **advanced features of React**.

We learn about things like refactoring, lifecycle methods, library integration and unit-testing.

Is this book right for me?

We assume that you:

- Have a basic grasp of HTML
- Have a working knowledge of basic JavaScript

I'd guess that you've probably gone through a React tutorial or two before buying this book, but that isn't a requirement.

You **don't** have to be a JavaScript expert and you certainly don't need any existing expertise in React.

We believe this book will be helpful if you've felt like learning React is daunting and you want to understand the core parts in isolation. If you're the type of person who learns by taking things apart and then putting them back together, then I think you'll really like this book.

Getting Started

This book comes with a folder of example code. The code examples load in your browser **without any extra tooling**. You can simply open the `.html` files in your browser and they should run.

Most of the code blocks in the book will show the name of the file in which you can find that code, like this:

react-from-zero/00-object-elements.html

```
var anotherElement = {
  $$typeof: magicValue,
  ref: null,
  type: "p",
  props: {
    children: "A nice text paragraph."
  }
};
```

As you can see in the heading of that block, the full code for this example can be found in `react-from-zero/00-object-elements.html`.

If you get lost at any point reading the book, open up the code example and look at the code to see the whole context.

Get Excited!

This book is designed to be an easy step-by-step tutorial on learning the basics of React. Don't be intimidated by all of the different pieces in React that you've read about on the web.

Over the next lessons, we're going to walk through the basics of React. By the end, you're going to have a firm foundation on which you can learn to build bigger apps.

Without further ado, let's get started!

– Kay

Part I

The first part of this book is mainly about getting up and running.

We will talk about basic React concepts and learn about 80% of the day-to-day work that we will be facing when building applications with React.

In this part you will learn:

- What a virtual DOM is
- What React elements are and how they are structured
- What JSX is and how it relates to React and JavaScript
- What React components are and how they are structured
- How a whole React application is structured
- How to build a complete application with React

The lessons are rather short, most of them can be done in under 1 hour. Let's dive right into the first chapter!

What's the point of React?

When using a new framework, the first question you should ask is, *"Why should I use it?"* so I want to take a step back and tell you about what makes React a better base for application architecture than, let's say, jQuery.

When we're using React, the problem we're trying to solve is how to update our "HTML" view with **dynamic data** and **user interactions**.

React manages the view layer of your application.

That is, you take your **data** and you pass it to **React** and the output is your "HTML" view.

I put "HTML" in quotes, because actually what happens is that our browser converts the HTML into *objects* which render the view we see. We call these objects *the DOM*.

At a high level, you can roughly think of what React does as the equation:

```
reactApp(yourData) => rendered view
```

Meaning, we take our React code (which we'll describe below), combine it with our data, and then a new view is rendered in our browser.

So, before we get to React, what are some historic approaches to handling this problem?

jQuery's approach is basically to manipulate the DOM directly. jQuery gives you, as its name implies, *query* methods that allow to get access to DOM elements.

Once you have a reference to a DOM elements you *imperatively* write text into your DOM nodes, add event listeners and so on. The problem with is that this gets unwieldy fast.

It starts small: you add a few elements to a list, change a `<p>` here and there, but as the application grows and requirements change **you end up with hundreds of elements that all need to be updated dynamically**.

If you're not careful the performance of your app can suffer. You're forced to craft some sophisticated algorithms that don't thrash the DOM and bring the browser to its knees.

Furthermore, it's just hard to keep track of the resulting nest of code – it can be difficult to orchestrate those handlers to all work together.

React tries to solve this problem for you and it does it with **the help of a *virtual* DOM** (VDOM), which is just a fancy name for **plain old JavaScript objects** nested to resemble the structure of a DOM.

The idea is, you write your code as if you would simply overwrite/replace your whole DOM.

However, React has some performance optimizations to make this run really fast in reality.

It's a win-win – This approach allows you to focus on your data and UI and frees you from meddling with the details of DOM updates - all while getting super-fast performance.

If you're pretty comfortable with JavaScript, at this point you might be interested in reviewing Appendix A: A Virtual DOM Primer. In that section, we'll walk through the code of building a toy Virtual DOM library.

However, if you're relatively new to JavaScript, you don't need to understand that part just yet.

We'll talk more about the VDOM in a later chapter, but for now, know that the virtual dom **takes a tree of objects** and then **renders a view**.

Again, the **virtual DOM** (VDOM), is just a fancy name for **plain old JavaScript objects** nested to resemble the structure of a DOM.

But **what are the objects that this VDOM requires?** What do they look like?

That's the topic of the next chapter. Let's take a look at the fundamental unit of a React app: **an object element**.

Quiz

1. What are the two problems React tries to solve?
2. What is the main difference between React and jQuery?
3. How does React simplify your code?

Lesson 0 - Object Elements

In order for React to render an object into the DOM, we need to describe *what* we need to be rendered.

Our approach here is going to be a little different than what you've seen in other tutorials: instead of showing you how to use React as a black box, we're going to start at the bottom and work our way up.

We're trying to *demystify* React and connect it to "everyday" JavaScript. To that end, we'll start by creating a **basic JavaScript object** that represents an *element* which React can use to render an element on our screen.

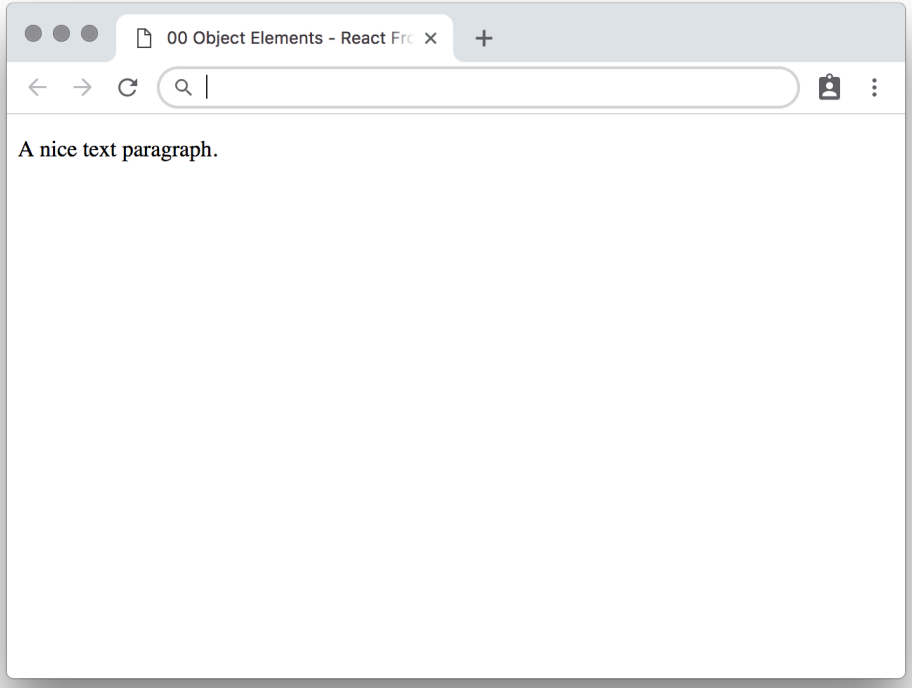
A Simple Element

First I want to show you a simple element object, it only uses the parts that are required by React.

A simple paragraph element implemented as React element would look like this:

react-from-zero/00-object-elements.html

```
var anotherElement = {
  $$typeof: magicValue,
  ref: null,
  type: "p",
  props: {
    children: "A nice text paragraph."
  }
};
```



Simple Element

Remember: the idea here is that we are creating a **basic JavaScript object** that will eventually render into an element like this HTML: `<p>A nice text paragraph.</p>`

For this object, the first part `$$typeof` is used by **React** to differentiate between simple objects and React elements.

If it's missing, React will complain. In the example, it uses a variable called `magicValue` which is either the magic number `0xeac7` or `Symbol.for("react.element")`.

`Symbol` is a relatively new object in JavaScript and not all browsers support it. If you use a browser with a JavaScript version that doesn't support symbols, the magic number is used, but if you have an up to date browser, React uses a symbol to mark objects as React elements.

The idea of *symbols* is that they are globally unique. If someone wants to access an attribute that uses a symbol as an object key or tries to check if something equals a specific symbol, they have to explicitly get that symbol with `Symbol.for()` function. With *String* or *Number* equality could happen by accident.

The term *magic number* can have different meanings in programming. In this case, it means a constant numerical or text value used to identify a file format or protocol. Wikipedia[2]

This magic number is arbitrary. It's just a number selected by the React team as an identifier to be used for all React element objects. You won't need to come up with your own magic numbers when using React, it's just an identifier for the React library.

React creates one of these symbols and then checks if objects contain it inside their `$$typeof` attribute.

The second part of `anotherElement` is the `ref`. It can be used to store a reference to the corresponding DOM element after this React element is rendered into the DOM, but we will talk about this in detail, later in another chapter.

Then we have the `type`. It tells React, what kind of DOM element it should create for this object, in this case, a paragraph. (This maps to the HTML `<p>` tag.)

Finally the `props`. Here you see a `children` prop that is a string. `props` is React's way to **pass data down the element tree**. Discussing this has to wait until another chapter, too.

This element `anotherElement` is the simplest possible React element. It's the "hello world" of React elements, if you will.

We could pass this into the `ReactDOM.render()` function (which we'll talk about in the next chapter) and it would render DOM elements like the HTML:

[2] https://en.wikipedia.org/wiki/Magic_number_(programming)

```
<p>A nice text paragraph.</p>
```

Next, let's look at another element, this time with all the optional parts that we left out in this simple example.

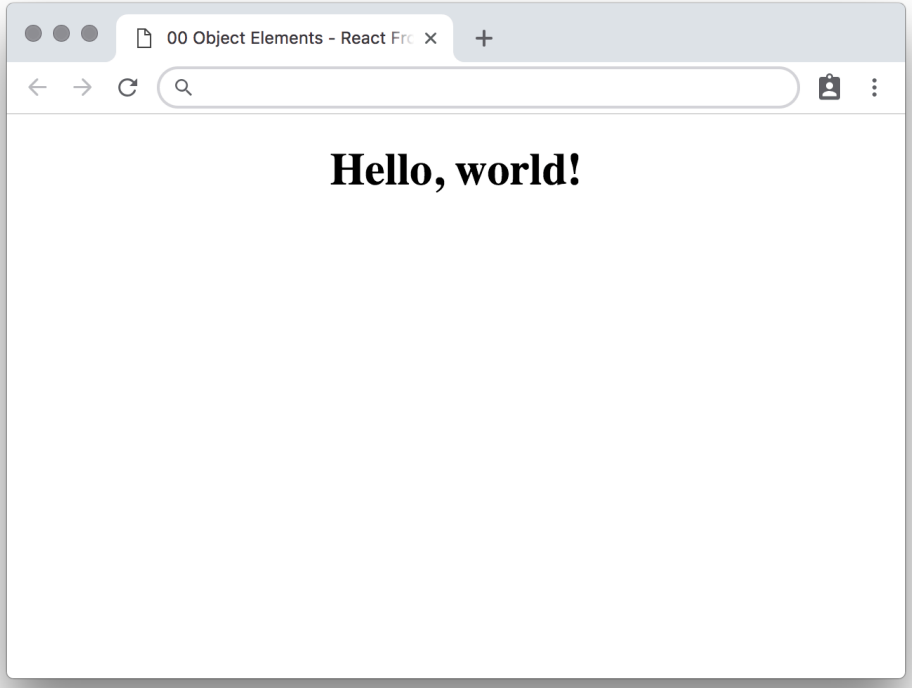
A Complex Element

Here's a more complex element that covers additional aspects of the React element API:

react-from-zero/00-object-elements.html

```
var reactElement = {
  // This special property will be checked by React to ensure this
  // object
  // is a React element and not just some user data
  // React.createElement() sets it for you
  $$typeof: magicValue,

  // This will also be checked by React. We will be talking about
  // references later, but if you're not using them, this has to be
  // set to null and not undefined
  ref: null,

  // This defines the HTML-tag
  type: "h1",

  // This defines the properties that get passed down to the element
  props: {
    // In this example there is just a single text node as child
    children: "Hello, world!",

    // a CSS class
    className: "abc",

    // styles can be passed as object literals
```

```
    // React uses camelCase instead of dashed-case (like CSS/D3 do)
    style: {
      textAlign: "center"
    },

    // event handlers can be added as properties, too
    // React uses synthetic events to try to normalize browser
    // behavior
    onClick: function (notYourRegularEvent) {
      alert("click");
    }
  }
};
```



Complex Element

First we have `$$typeof`, `ref`, `type` and `props` again. Nothing special here, but this time we added `props` other than `children`.

There is for example `className`. It is the React way to add a CSS `class` attribute to an element, it makes it easy to style your React elements with external CSS.

The `className` is a DOM element property used to set the value of class attribute of that element. React avoided using `class` because it is a reserved word in JavaScript.

Next, there is `style`. If you write HTML, style is normally a string attribute, but for React it has to be an object. Also, it uses `camelCase` instead of `kebab-case` what you

would expect from CSS. This allows for simpler transformation of local styles. So in this case, our `textAlign` key maps to the `text-align` property in CSS.

At the end, we have `onClick`. This is the way to add **event handlers** to objects. The difference from HTML is that they are written in `camelCase` instead of `lowercase` and you can pass functions directly without the need to wrap them in a string.

Another difference to these handlers is that they get a `SyntheticEvent` as an argument, rather than a browser's native event. This is done to **normalize events across all browsers** so that events always look the same to our code.

`SyntheticEvents` also have a `nativeEvent` attribute you can use if you need the *real* event for some reason.

Also, `SyntheticEvents` are re-used when your function returns, so gather all the data you need from them before you make any asynchronous calls.

Wrap Up

In this lesson, we showed how to define React element with basic objects.

I showed you that **there is no tooling required to get started** and that you can even skip some helper methods and JSX if you want to play around with React.

We also talked about some subtle differences on how React does things that may not match your DOM/HTML experience.

You probably also noticed that using this approach, while being easy to understand the basics, is rather cumbersome for a big project.

While components will provide you with mechanisms to reuse this code, it's still not easy to follow, especially for big elements with many props and children.

In the next lesson, we will talk about a little helper method, that eases some of the pain.

Quiz

1. What JavaScript language construct is used to implement virtual DOM elements in React?
2. What field is used in React to set a class for the elements?
3. How are interactions implemented in React?

Lesson 1 - Element Factory

In the previous chapter, we explained how you define React elements with plain JavaScript objects. There isn't much to them, but they are a bit cumbersome to work with, primarily because many of the required keys are often empty or always have the same values.

In this short chapter, we'll learn about a way to get rid of this boilerplate code.

React provides you with a helper function called `React.createElement()`. It adds `$$type` and `ref` attributes to your element definitions every time you call it.

A Simple Element

In the previous lesson, this was how we created an element manually:

react-from-zero/00-object-elements.html

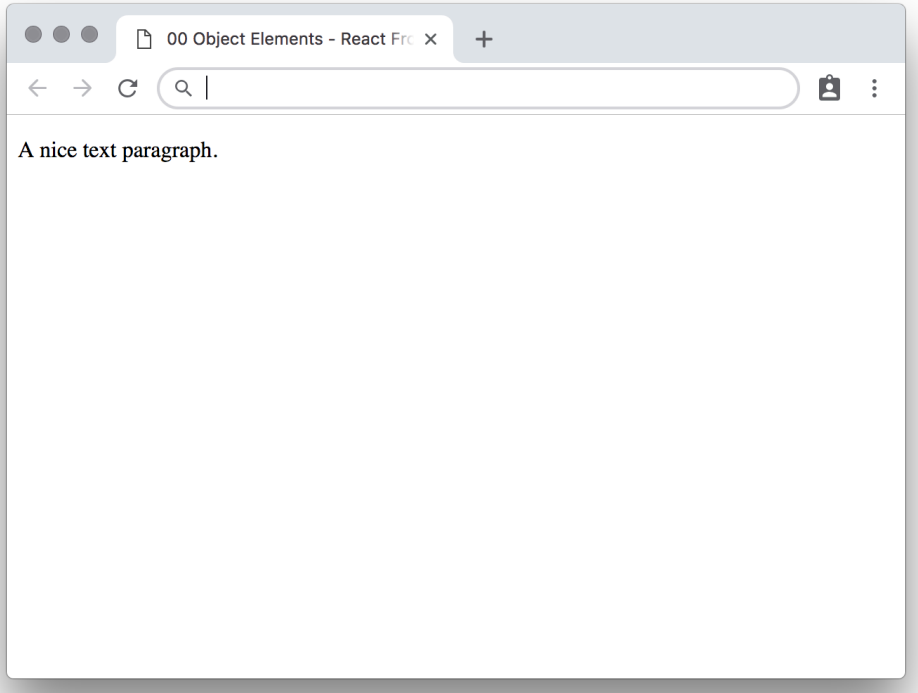
```
var anotherElement = {
  $$typeof: magicValue,
  ref: null,
  type: "p",
  props: {
    children: "A nice text paragraph."
  }
};
```

However, if we used the factory function the code is reduced to this:

react-from-zero/01-element-factory.html

```
var anotherElement = React.createElement(
  "p",
  null,
  "A nice text paragraph."
);
```

This is a more concise (and less error-prone) way to define the same element.



Simple Element

The `createElement` function takes three arguments.

1. The `type` of the element we want to create, in this case, a `"p"`.

2. The `props` of the element, things like event handlers or styles. It must be `null` if no `props` are used.
3. The `children` of the element. While children are also `props` React also allows to add them via an extra parameter.

A Complex Element

The more complex example we used in the last chapter would look like this:

react-from-zero/01-element-factory.html

```
var reactElement = React.createElement(
  "h1",
  {
    className: "abc",

    style: {
      textAlign: "center"
    },

    onClick: function () {
      alert("click");
    }
  },
  "Hello, world!"
);
```

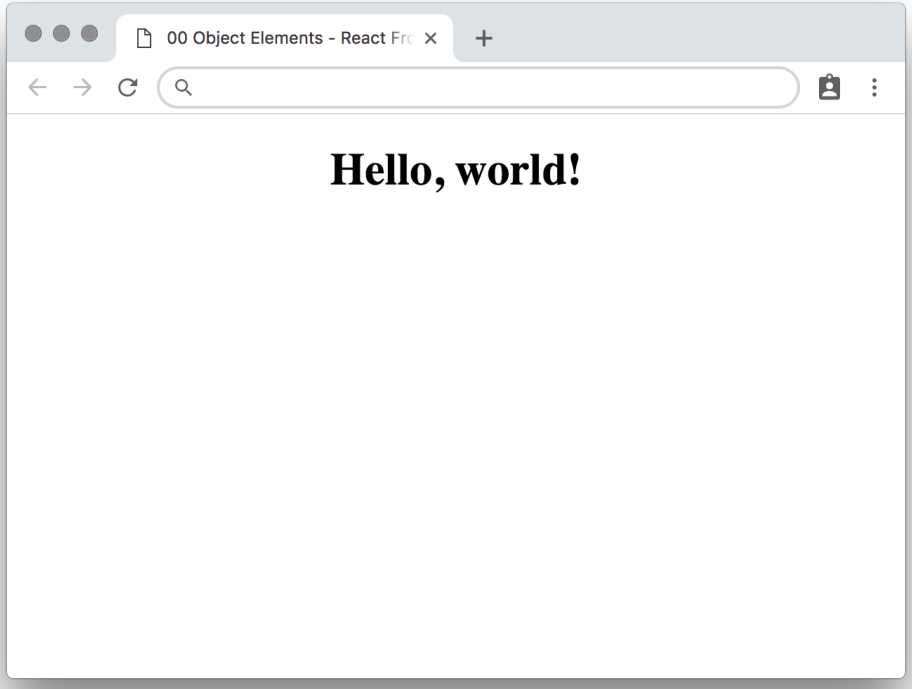
Now we use the `props` parameter to define a `style` a `className` an `onClick` event handler.

And if we were to expand the `reactElement` generated by `React.createElement`, the object looks exactly as before:

react-from-zero/00-object-elements.html

```
var reactElement = {
  // This special property will be checked by React to ensure this
  // object
  // is a React element and not just some user data
  // React.createElement() sets it for you
  $$typeof: magicValue,

  // This will also be checked by React. We will be talking about
  // references later, but if you're not using them, this has to be
  // set to null and not undefined
  ref: null,

  // This defines the HTML-tag
  type: "h1",

  // This defines the properties that get passed down to the element
  props: {
    // In this example there is just a single text node as child
    children: "Hello, world!",

    // a CSS class
    className: "abc",

    // styles can be passed as object literals
    // React uses camelCase instead of dashed-case (like CSS/D3 do)
    style: {
      textAlign: "center"
    },

    // event handlers can be added as properties, too
    // React uses synthetic events to try to normalize browser
    // behavior
    onClick: function (notYourRegularEvent) {
      alert("click");
    }
```

```
    }
};
```



Complex Element

Wrap Up

The factory function `React.createElement()` is a much more concise way to define React element objects. Using `React.createElement` especially shines with simple elements that don't have much configuration (which is the lion's share of most applications).

Some React developers even save `React.createElement()` into a shorter identifier

like `h`, so they can call it directly without much typing and without the need of JSX, which we will talk about in the next chapter.

Quiz

1. What are the three arguments the `React.createElement()` function accepts?
2. What does the `React.createElement()` function return?
3. What does the `React.createElement()` add that we need to do manually when using element objects.

Lesson 2 - JSX

When I hear people complain about React, it's mainly due to **JSX**, the non-standard syntax extension created for writing markup in JavaScript.

If you've used other frameworks like Vue or Ember, you are probably used to template languages like HTMLBars. These are languages with their own semantics and syntax embedded into JavaScript with strings.

The React team went in a different direction: **they tried to use JavaScript directly** to create markup, but also added a JSX as **syntax extension** to the mix. So you can still use your JavaScript know-how, but also use a syntax that more closely resembles HTML.

JSX Compilation

The drawback here is that **it needs an extra compile step** to run in a browser. Which is the second frequent complaint about React: people think they can't use it without an extra build step.

While this is technically true (if you're going to use JSX), there are ways to use JSX compilers like Babel, directly in the browser with a `<script>` tag.

Babel is a **JavaScript compiler**; it is used to compile *new JavaScript features* down to an older version of JavaScript, so that the new features can be used in older browsers that don't implement them natively.

In this case, it **compiles down JSX to regular JavaScript function calls** (using the `React.createElement()` function we discussed in the last chapter).

Because Babel itself is JavaScript (which runs in the browser) we can include the Babel compiler into our app by adding a `<script>` tag and then instructing Babel to compile our JSX/JavaScript code.

Here we see the Babel stand-alone version, which is created to run in browsers, included via a `<script>` tag from the *Unpkg* content delivery network:

react-from-zero/02-jsx.html

```
<script src="https://unpkg.com/@babel/standalone/babel.min.js">
```

A content delivery network, short CDN, is a service that provides hosting for content. Unpkg, for example, mirrors the NPM registry and makes the NPM packages available via HTTP, so no extra tooling is required to use them.

If you use the stand-alone version of Babel in the browser, you have to write your JSX and JavaScript inside a `<script type="text/babel">` tag. By specifying the `type` of `text/babel` we're telling the Babel compiler that we want it to compile this code for us (which means we'll gain the benefit of the extra features it supports, i.e. JSX).

A Simple Element

So lets start with our simple element example again:

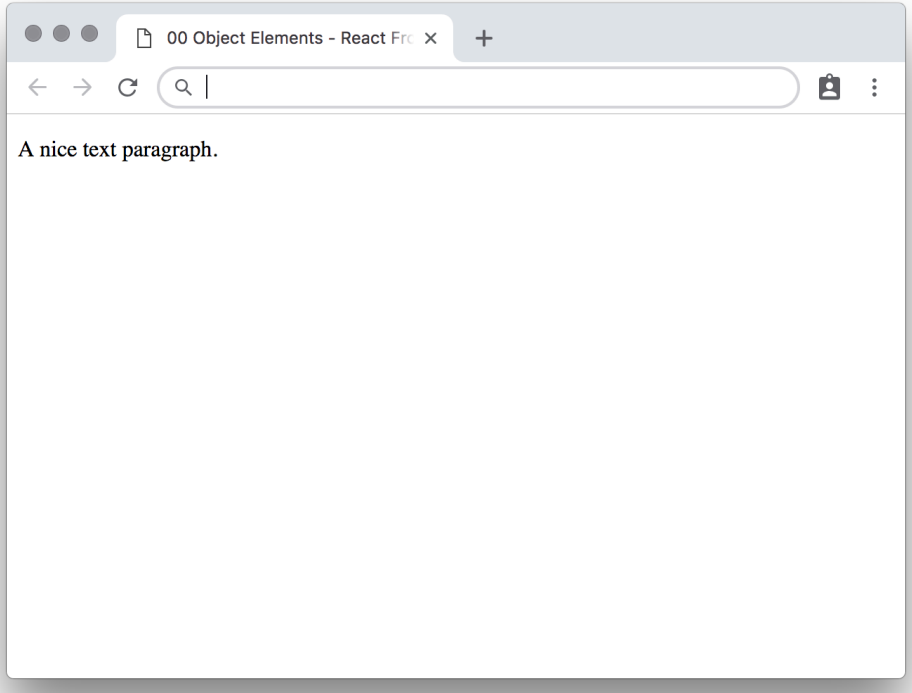
react-from-zero/02-jsx.html

```
var anotherElement = <p>A nice text paragraph.</p>;
```

This would be compiled down to our simple element example from the factory element lesson.

react-from-zero/02-jsx.html

```
anotherElement = React.createElement(
  "p",
  null,
  "A nice text paragraph."
);
```



Simple Element

Take a look at the top paragraph code example again - we have `<p>` tags in our JavaScript code! This syntax is supported by the Babel compiler.

The **JSX tags will be converted to JavaScript function calls** to `React.createElement()`.

The key idea of what's going on here is that **we are simply using the JSX syntax as a shorthand for the JavaScript `React.createElement()` call**.

Technically, the default configuration of JSX is to call the `React.createElement()` function, however this could also be configured to call a different function if you wanted to.

The *tag name* will become the first, or `type`, parameter of the function, so different tags don't call different functions here, they just supply different `type` parameters to the same function.

One of the powerful features of JSX is that we can compose, not only "primitive" HTML elements, but we can also create our *own* component elements. In order for JSX to support that, there are two cases for the element *tag name*:

If the *tag name* is `lowercase` it will be **passed as a string to the function call**, the function will then try to create a regular VDOM element (as in our example, with the `p` element).

If the *tag name* is `UpperCase` it will be **passed as a *variable* to the function call**, a variable of the same name as the *tag name*. If no such variable exists in the current scope, you get an error. This feature is used to create elements from custom *components*, one major feature about which we will talk about in a later lesson.

The *content* of the element can be another element or, in our case, text. This will be passed into the third, or `children`, a parameter of the function. This allows you to create nested elements like we are used to in HTML. I will tell you about nested elements in the following lesson.

In this simple example, we didn't use `props` so the second parameter of the call to the `React.createElement()` function will be `null`.

A Complex element

The more complex example from the last chapter with many `props`, would look like this in JSX:

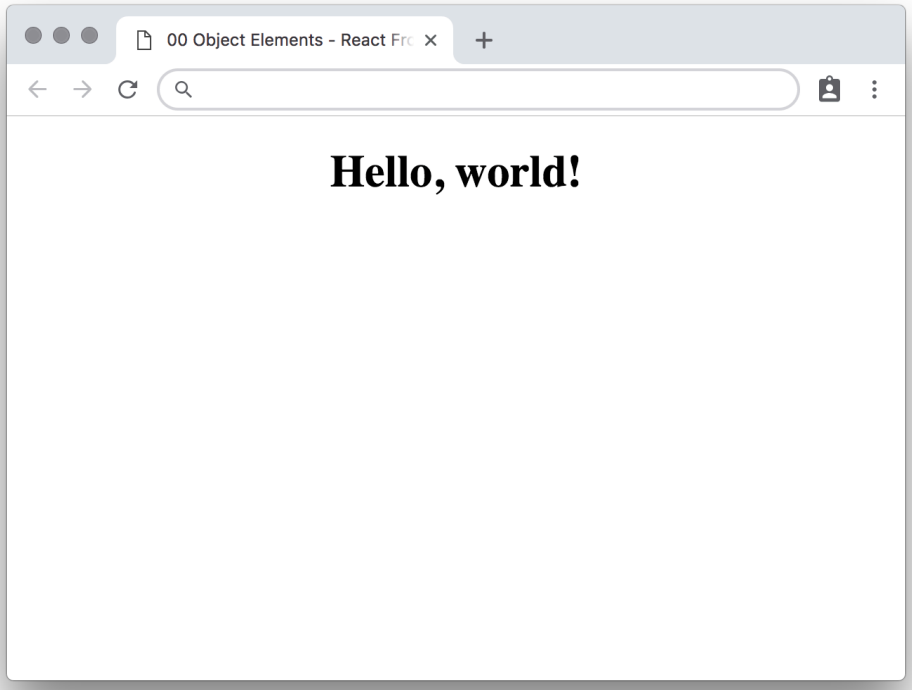
react-from-zero/02-jsx.html

```
var reactElement = (
  <h1
    className="abc"
    style={{ textAlign: "center" }}
    onClick={function() {
      alert("click");
    }}
  >
    Hello, world!
  </h1>
);
```

And it would be compiled to this:

react-from-zero/02-jsx.html

```
reactElement = React.createElement(
  "h1",
  {
    className: "abc",
    style: { textAlign: "center" },
    onClick: function() {
      alert("click");
    }
  },
  "Hello, world!"
);
```



Complex Element

Here you can see that the `props` were converted to a `props` object and placed into the second parameter of the `React.createElement()` function call.

You should also notice **how curly braces are used to embed JavaScript directly into JSX**.

For example, look at the `style` attribute above:

```
style={{textAlign: "center"}}
```

Notice that we have two curly braces. Why is that?

In this case, we have an inner, plain JavaScript object:

GET THE FULL BOOK

This is the end of the preview!

Head over to:

https://fullstackreact.com/react-from-zero

to get the full version!

Featuring:

- 20 lessons
- 105+ pages
- Complete code

GET IT NOW