Ajdin Imsirovic

REACT FROM SCRATCH

Book 2: Hooks and Forms

Build your portfolio with
React's hooks and forms

React From Scratch, Book 2: Hooks and forms

Build your porfolio with hooks and forms

Ajdin Imsirovic

This book is for sale at http://leanpub.com/react-from-scratch-book-2-hooks-and-forms

This version was published on 2023-01-19

Contents

Chapter 0: Before we start

0.1 Disclaimer

Before I even begin, note that this disclaimer is exactly the same as the one in the first book in this book series. So, if you've read it in the previous book, you can skip to the next section.

Here goes the not-so-obligatory information that most people skip - but, I kept it really short and sweet - under one page.

Give it a try - I've tried my best to give you some context in this section and some of my thoughts before we go into the topic of the book.

Here goes.

I've tried my best to make this book a success.

But, I'm only another guy, and, like the vast majority of people, I do make mistakes.

I hope that you, dear reader, will be able to appreciate the intention and the effort I have put into writing this book. It is a product of many, many hours spent in front of a computer monitor, either writing about code, thinking about code, debugging some code, or just plain coding. I don't take this effort lightly. It's my source of pride - and the reason why I feel competent to write this book.

I'd say I'm way past the 10 thousand hours of deliberate practice, the concept popularised by Malcolm Gladwell in his book Outliers.

So, who am I and why should you listen to my advice?

Well, I've already sort of answered that question above. I'm just another guy; a self-taught web developer who has spent years mastering my craft.

In some parts of the web development world, I'm a seasoned veteran. In some other parts of it, I'm a newbie.

I'm not a know-it-all, if such a person exists.

But I have dedicated a large part of my professional career to writing tutorials on web development, writing books about it, and thinking about best ways to transfer knowledge. Reading up on it too!

So I'd say I've become competent to teach this subject, based on my experience.

If you find this book not to your liking, luckily, Leanpub has a really generous 60-day return policy. You might say that's as risk-free as it can get. You can't really say that it cost a fortune to begin with, and the return policy is working in your favor anyway. So if you strongly feel that this book is doing you no good - please, by all means, return it. No reason to get worked up about it, either on my side or on yours.

If you have complaints about the book, please write to me at ajdin@codingexercises.com and I'll do my best to deal with the issue.

Also, any comments and kind words, as well as harsh criticisms, are all welcome.

Kind words from readers mean a lot.

Harsh criticisms, while I do find them usually not really neccessary, are a way to point me in the right direction - and ultimately help me become a better technical writer and a better developer.

So I thank you either way.

0.2 Assumptions: Things to have and know before we start

If you're starting this book, I expect you at least have Node.js and NPM installed on your machine.

I also expect that you know everything I've covered in Book 1 of this book series.

Ideally, you'd have gone through the contents of that book and you're ready to continue with this one.

If you haven't gone through the first book, perhaps some chapters in this one might be a bit challenging.

How can you determine if that's going to be the case for you?

Well, here's an overview of things covered in Book 1.

0.2.1 Things covered in *Chapter 1: Simplest app*

In this chapter:

1. I've shown to you how to install React Developer Tools in your browser.
2. I've introduced you to the create-react-app npm package, and I've shown to you how to use this package to build a boilerplate React app.
3. I've explained all the files and folders in that boilerplate app, which goes a long way, because now you'll understand the basic structure of any React app built using this package
4. I've shown how you can update this starter app to change its output

0.2.2 Things covered in *Chapter 2: Serving React from a CDN*

In this chapter, I've shown to you:

1. How to build a React app without a build step (using Codepen.io)

2. How to render multiple heading elements
3. How to use React.createElement
4. The DOM in context of React
5. Improving your app's code with JSX
6. How the app transpiling works
7. The dynamic nature of JSX

0.2.3 Things covered in *Chapter 3: Build React apps locally without webpack*

In this chapter, I've shown to you the following:

1. A more in-depth look at `React.createElement`
2. Understanding `className`

0.2.4 Things covered in *Chapter 4: What is JSX*

In this chapter, I've guided you through:

1. Structuring the return values using JSX
2. Working with Babel
3. Using dynamic (evaluated) values in JSX
4. Working with nested elements in JSX

0.2.5 Things covered in *Chapter 5: Components and props*

In this chapter, we've learned:

1. How to code a component in React
2. How to export a component and how to import it into another component
3. Using a single wrapping element is a must
4. Introduction to props in React
5. Parent-child structure of components in React

0.2.6 Things covered in *Chapter 6: Assets in React*

In this chapter, I've shown to you how to use images and JSON in React apps, and how to loop over the JSON data inside a component.

0.2.7 Things covered in *Chapter 7: JSX and styling*

In this chapter, I've shown to you how to work with styles in JSX. Specifically, I've covered:

1. Importing styles using the `link` element
2. Inline styles with object literals
3. Inline styles with variables
4. Composable styles using the spread operator

0.2.8 Things covered in *Chapter 8: Build a static Bootstrap layout*

In this chapter, I've shown to you how to use the Bootstrap framework with React, without using a custom-made framework like, for example, React-Bootstrap.

I've also shown how to work with the `children` prop.

0.2.9 Things covered in *Chapter 9: Looping over data*

In this chapter, I've explained the topic of looping over data. I've shown you a practical example of mapping over data and for each piece of data, rendering a component. That way, you avoid repetition and make your code easier to reason about.

I've also shown to you how to refactor a component so that:

- it uses a single prop, and
- uses object destructuring

0.2.10 Things covered in *Chapter 10: The very basics of data and events in React*

In this chapter, I've covered the following:

1. Data
2. State data
3. Using events to update state data

0.2.11 Things covered in *Chapter 11: Conditional rendering*

In this chapter, I've shown you how to conditionally render:

1. if-else statements
2. classNames using ternary operators
3. a different value than the one in received in the `props` object
4. different components
5. components using the `&&` operator

I've also shown to you how to:

- extract the return values as varaibles
- rendering them using the ternary operator

0.3 What this book is about?

Ok, so now that I have given you an overview of things covered in the previous book, I'd like to give you a quick introduction to things that this book will be discussing.

First of all, since the title of this book series is "React from Scratch", I'm indeed working on introducing you to all the relevant, important concepts, without the learning curve being too steep.

What that means is that, now you understand the very basics of React's syntax and some of the logic behind it, I'll continue with the approach from the previous book: introducing a new concept, isolating it and simplifying it as much as possible, and then giving you an app that shows how that concept works in practice.

This should help you be confident in building your own code that's similar to the one shown.

Furthermore, this approach will allow us to build "multi-page" SPAs in the book that follows this one.

However, we shouldn't get ahead of ourselves. For now, the focus is on understanding hooks, forms, and some related concepts, without which it will be quite difficult to make some more advanced React apps.

Welcome to *Book 2: Hooks and Forms*.

Chapter 1: Understanding the useState hook

1.1 Updating a component's state using the useState hook

Ideally, an app would consist of stateless components through and through.

In other words, an ideal app would be a static app, where components compose together to build an app that has no state.

However, that app would be boring. A user would not be able to interact with it other than read the information on the screen - since that information would never change.

This means that a lot of interactivity in an app is actually user-triggered.

For example, if I want to allow a user to add an item to a cart on a shopping website, I'll have to:

- add a button for the user to press to complete such an action
- react to events triggered by such a user action

Essentially, what this boils down to is that we need to code our React apps to handle user events which will in turn update a given component's state.

That means that I need to:

1. Set up state in a component
2. Set up event handlers
3. Update state inside event handlers

In the first book of this book series, I've shown to you a basic example of how this is done.

As a starting point, here's a quick reminder, A basic add-to-cart example, pt 3*.

The entire app was just a single component - the *App* component:

*https://codesandbox.io/s/a-basic-add-to-cart-example-pt-3-hqq0fr

```
import React from "react";

export default function App() {
  const [cartCount, setCartCount] = React.useState(0);

  const handleClick = () => setCartCount(cartCount + 1);

  return (
    <>
      <h1>Number of items in the cart: {cartCount}</h1>
      <button onClick={handleClick}>Add to cart</button>
    </>
  );
}
```

Now, based on this app, let me build another one.

The app that I'm about to show you will have the following characteristics:

1. It will have the `userName` variable, which will be updated using the `setUserName` variable.
2. It will have the `userAge` variable, which will be updated using the `setUserAge` variable.

How can I do this?

Well, here's a challenge for you: Try coding this app yourself! Here's a tip: you can use two calls to the `useState` hook, where the first state variable will be just a humble string, and the second one just a humble number.

If you'd rather just examine a completed app, here's the completed app named User name and age changer*.

Again, the app is pretty simple, so there's only a single file that I'll be working on - the `App.js` file:

```
import React from "react";

export default function App() {
  const [userName, setUserName] = React.useState('James');
  const [userAge, setUserAge] = React.useState(21);

  const updateUsername = () => {
    setUserName("Mark");
  }

```

*https://codesandbox.io/s/user-name-and-age-changer-el431w

```
  const updateUserAge = () => {
    setUserAge(25);
  }

  return (
    <>
      <h1>User details:</h1>
      <h2>{userName}</h2>
      <h2>{userAge}</h2>
      <button onClick={updateUsername}>Update username</button>
      <button onClick={updateUserAge}>Update user age</button>
    </>
  );
}
```

While this app works, it still has several disadvantages.

1. You can only update the user's name and age once
2. You need to use the useState hook twice.

For now, let's focus on updating the user's name and age multiple times. In other words, making sure that clicking the buttons will update the state forever.

A simple approach would be to check the previous state and act accordingly.

Here's how.

```
import React from "react";

export default function App() {
  const [userName, setUserName] = React.useState("James");
  const [userAge, setUserAge] = React.useState(21);

  const updateUsername = () => {
    userName === "James" ? setUserName("Mark") : setUserName("James");
  };

  const updateUserAge = () => {
    userAge === 21 ? setUserAge(25) : setUserAge(21);
  };

  return (
    <>
```

```
      <h1>User details:</h1>
      <h2>{userName}</h2>
      <h2>{userAge}</h2>
      <button onClick={updateUsername}>Update username</button>
      <button onClick={updateUserAge}>Update user age</button>
    </>
  );
}
```

The way that I'm making the buttons constantly "do something", that is, constantly update the state, is by using a simple ternary operator, which toggles back and forth between the two possible user names, and the two possible user ages.

Of course, this is nothing special, but don't worry, we're building towards something more impressive.

I've named this update to my app User name and age changer, pt 2*.

Now let me tackle the other issue, that is, the issue of having to use the `useState` hook twice.

1.1.1 Using an object to set state with the useState hook

Just like in plain JavaScript I can save multiple primitive values in multiple variables, then update those variables individually, I've done the exact same thing in my example React app so far.

Likewise, just like I can use an object to group related values as properties on that single object, I can do the same thing in React.

Here's my updated app:

```
import React from "react";

export default function App() {
  const [user, setUser] = React.useState({ name: "James", age: 21 });

  const updateUsername = () => {
    setUser({ name: "Mark", age: 25 });
  };

  const updateUserAge = () => {
    setUser({ name: "Mark", age: 25 });
  };

  return (
```

*https://codesandbox.io/s/user-name-and-age-changer-pt-2-jv5wbz

```
15      <>
16        <h1>User details:</h1>
17        <h2>{user.name}</h2>
18        <h2>{user.age}</h2>
19        <button onClick={updateUsername}>Update username</button>
20        <button onClick={updateUserAge}>Update user age</button>
21      </>
22    );
23  }
```

I've named this update to my app User name and age changer, pt 3*.

However, this update is not really good.

While I did manage to have the `useState` hook use an object, I haven't really succeeded in handling button clicks properly.

There are two issues here:

1. First, I'm hard-coding the updates to state using `setUser`, which is not the React way of doing it
2. Additionally, in my hard-coded updates, I'm updating the complete object, rather than the specific property, that is, either the name, or the age

Let's fix both.

The way that I'll fix both issues is by:

1. Returning a new object by **cloning** the previous value of the state object (using the **spread** operator)
2. Updating a given **specific property** on the newly returned state object - based on which button is pressed

Here's the updated code:

*https://codesandbox.io/s/user-name-and-age-changer-pt-3-o3ekpf

```
import React from "react";

export default function App() {
  const [user, setUser] = React.useState({ name: "James", age: 21 });

  const updateUsername = () => {
    setUser({ ...user, name: "Mark" });
  };

  const updateUserAge = () => {
    setUser({ ...user, age: 25 });
  };

  return (
    <>
      <h1>User details:</h1>
      <h2>{user.name}</h2>
      <h2>{user.age}</h2>
      <button onClick={updateUsername}>Update username</button>
      <button onClick={updateUserAge}>Update user age</button>
    </>
  );
}
```

I've named this update to my app User name and age changer, pt 4*.

Again, similar to what we've had when the user data was saved as primitive values, again, I have the issue of the buttons updating state only once.

So again, as an intermediate step in the development of this app, I'll update it so that every button click will update the state.

Here's the newest update.

```
import React from "react";

export default function App() {
  const [user, setUser] = React.useState({ name: "James", age: 21 });

  const updateUsername = () => {
    user.name === "James"
      ? setUser({ ...user, name: "Mark" })
      : setUser({ ...user, name: "James" });
```

*https://codesandbox.io/s/user-name-and-age-changer-pt-4-iiwzbz

```
  };

  const updateUserAge = () => {
    user.age === 21
      ? setUser({ ...user, age: 25 })
      : setUser({ ...user, age: 21 });
  };

  return (
    <>
      <h1>User details:</h1>
      <h2>{user.name}</h2>
      <h2>{user.age}</h2>
      <button onClick={updateUsername}>Update username</button>
      <button onClick={updateUserAge}>Update user age</button>
    </>
  );
}
```

I've named this update to my app User name and age changer, pt 5*.

This is the improved version of the app that showcases the `useState` hook in use.

Of course, this app can be improved even further, using forms. Specifically, this app would work much better if instead of a button that a user can press, I had an input field where the user could type a name in.

I'll show you how to do exactly that in the next chapter.

*https://codesandbox.io/s/user-name-and-age-changer-pt-5-4kj5e6

Chapter 2: Forms in React

How do forms work in React? This is the chapter where I'll introduce you to the topic.

2.1 Forms in React

Contrary to some other ways of dealing with forms on the web, with React, forms are put into a state variable, and then React watches for any form-related event, with every such event updating the form's state.

This means that React's sort of constantly watching your form for any changes.

2.2 Watch for input changes in React

In this section, I'll give you a simple app with a single input element.

```
function App() {
  return (
    <form>
      <input type="text" placeholder="Enter User Name" />
    </form>
  );
}
export default App;
```

I've named this update to my app Input element in React*.

While this app shows valid code, I'm not keeping any state in this component. In other words, the component is stateless.

As I've emphasized earlier, while a stateless component is a "perfect scenario" because it's as simple as things can get, in React, a stateless form is kind of pointless.

So let's detect and handle the `onChange` event on this form's input element, before adding state to handle this change.

2.3 Detecting and handling the `onChange` event on an input

I'll also add the `onChange` event attribute, so that I can call the `changeHandler` function:

*https://codesandbox.io/s/input-element-in-react-pit95j

```
function App() {
  return (
    <form>
      <input
        type="text"
        placeholder="Enter User Name"
        onChange={changeHandler}
      />
    </form>
  );
}
export default App;
```

Next, I need to define the `changeHandler` function:

```
function App() {
  function changeHandler() {
    alert('Change event fired!')
  }

  return (
    <form>
      <input
        type="text"
        placeholder="Enter User Name"
        onChange={changeHandler}
      />
    </form>
  );
}
export default App;
```

I've named this update to my app Input element in React, pt 2*.

What the app now does is, it shows an alert for any change detected on the form. Next, let me add the built-in Event object's instance. I'll explain why in the next section.

2.4 Adding the built-in event object

To begin, let me get hold of the instance of the built-in Event object.

I can name this instance anything I want, so I'll use the lowercased letter `e`, implying the word "event".

*https://codesandbox.io/s/input-element-in-react-pt-2-lb9y7z

```
function App() {
  function changeHandler(e) {
    alert(e.target);
  }

  return (
    <form>
      <input
        type="text"
        placeholder="Enter User Name"
        onChange={changeHandler}
      />
    </form>
  );
}
export default App;
```

So, the `e` object instance of the built-in Event object is readily available. I then access the `target` property of this object instance.

I've named this update to my app Input element in React, pt 3*.

This time, whenever a change is detected, I get the following output in the alert:

```
[object HTMLInputElement]
```

What this means is that my event's target is an object of the `HTMLInputElement` type - in other words, the input element itself - or, more specifically, the DOM object representing the input element.

The target property is also an object, which means that I can access properties that exist on it. One of those properties is the `value` property, so let me update my app by alerting `e.target.value`.

This is the only update.

```
function App() {
  function changeHandler(e) {
    alert(e.target.value);
  }

  return (
    <form>
      <input
        type="text"
```

*https://codesandbox.io/s/input-element-in-react-pt-3-lgc43i

```
        placeholder="Enter User Name"
        onChange={changeHandler}
      />
    </form>
  );
}
export default App;
```

I've named this update to my app Input element in React, pt 4*.

You probably understand now where I'm going with this. I now have a function which outputs the internal form's state as its change gets detected and handled by my `changeHandler` function.

However, I'm still not working with state the React way.

So now, I simply need to set my forms state, then update it based on the value held in `e.target.value`.

2.5 Adding state to the form input

To begin, let's import `React` as an object so that we can access the `useState` method on this object using the dot operator.

Additionally, I'll add the `user` state variable and set it to an object.

```
import React from "react";

function App() {
  const [user, setUser] = React.useState({ name: "James", age: 21 });

  function changeHandler(e) {
    alert(e.target.value);
  }

  return (
    <form>
      <input
        type="text"
        placeholder="Enter User Name"
        onChange={changeHandler}
      />
    </form>
  );
```

*https://codesandbox.io/s/input-element-in-react-pt-4-h9ghk9

```
}
export default App;
```

Now that I have state at my disposal, I can update the code in the body of the `changeHandler` function declaration to the following:

```
  function changeHandler(e) {
    setUser({...user, name: e.target.value})
  }
```

Now, the app's full code is as follows:

```
import React from "react";

function App() {
  const [user, setUser] = React.useState({ name: "James", age: 21 });

  function changeHandler(e) {
    setUser({ ...user, name: e.target.value });
  }

  return (
    <div>
      <h1>
        Name: {user.name}, age: {user.age}
      </h1>
      <form>
        <input
          type="text"
          placeholder="Enter User Name"
          onChange={changeHandler}
        />
      </form>
    </div>
  );
}
export default App;
```

Notice that I've added a wrapping div to the return, as well as an h1 that evaluates the value of the `user.name` and the `user.age` variables as their text nodes.

I've named this update to my app Input element in React, pt 5*.

*https://codesandbox.io/s/input-element-in-react-pt-5-f6mbk6

This looks like a good opportunity to show you an example of a potential issue that you might be facing as you work with objects, state, and forms in React.

2.5.1 Error: Objects are not valid as a React child

Consider the following example:

```
import React from "react";

function App() {
  const [user, setUser] = React.useState({ name: "James", age: 21 });

  function changeHandler(e) {
    setUser({ ...user, name: e.target.value });
  }

  return (
    <div>
      <h1>
        {user}
      </h1>
      <form>
        <input
          type="text"
          placeholder="Enter User Name"
          onChange={changeHandler}
        />
      </form>
    </div>
  );
}
export default App;
```

I've named this update to my app Input element in React, pt 6*.

If you open the link to the app, in the served app's preview, instead of the app being rendered, you'll be greeted with the following error:

```
Error
Objects are not valid as a React child (found: object with keys {name, age}). If you\
 meant to render a collection of children, use an array instead.
```

*https://codesandbox.io/s/input-element-in-react-pt-6-7u3f1i

Why is this error appearing?

Actually, the explanation is really, really simple. If you try to render an object, it will confuse React because it will not know what to render. Should it render `user.name` or `user.age`?

That's essentially what this error is all about.

Next, let me introduce you to the proper way of updating this form object in the event-handling function.

2.6 The proper way of updating the state object in React

An important caveat of updating the state object has to do with copying the previous state into a new object, then returning that copied object with additional changes as needed.

A common and easy way of copying (aka "cloning") an object in JavaScript is with the use of the spread operator.

Here's the full code of the improved app:

```
import React from "react";

function App() {
  const [user, setUser] = React.useState({ name: "James", age: 21 });

  function changeHandler(e) {
    const newStateObject = {...user};
    newStateObject.name = e.target.value;
    setUser(newStateObject);
  }

  return (
    <div>
      <h1>{user.name}, {user.age}</h1>
      <form>
        <input
          type="text"
          placeholder="Enter User Name"
          onChange={changeHandler}
        />
      </form>
    </div>
  );
```

```
}
export default App;
```

Let's discuss the `changeHandler` function declaration:

```
function changeHandler(e) {
    const newStateObject = {...user};
    newStateObject.name = e.target.value;
    setUser(newStateObject);
}
```

So, this function works with an instance `e` of the Event object triggered by the app's user typing into the input.

I declare a `newStateObject` variable, and assign to it the cloned `user` state variable's object.

Then I update the `name` value on the cloned object so that it receives the `e.target.value` string.

Finally, I use the `setUser` function to update the state variable with the value of the `newStateObject`.

I've named this update to my app Input element in React, pt 7*.

While this code is working, and in addition I am updating the old object's state with the new object - which is a general rule of how to properly update state, there is still something missing here.

Specifically, the usual way of updating state involves using arrow functions.

So now I'll update my app using arrow functions to update state, in addition to using an arrow function to save my event-handling function as a function expression.

First, I'll update the `changeHandler` function so that the `setUser` function call inside of it uses an arrow function, as follows.

```
  function changeHandler(e) {
    setUser(oldState => {
      return {...oldState, name: e.target.value}
    });
  }
```

Note the `oldState` parameter - which is the previous value of the `user` object. Notice how I can name it anything I want. I'm not really introducing anything new in this code. It's pretty much the same as what I had in the previous example.

Let's further improve the `changeHandler` function by refactoring it as a named function expression.

Here's the update:

*https://codesandbox.io/s/input-element-in-react-pt-7-he5fyp

```
  const changeHandler = e => {
    setUser(oldState => {
      return {...oldState, name: e.target.value}
    });
  }
```

Again, the way my code works hasn't really changed in this latest update. The goal here was just to show you a more common way of how code is written in React.

The completed app now has the following code:

```
import React from "react";

function App() {
  const [user, setUser] = React.useState({ name: "James", age: 21 });

  const changeHandler = (e) => {
    setUser((oldState) => {
      return { ...oldState, name: e.target.value };
    });
  };

  return (
    <div>
      <h1>
        {user.name}, {user.age}
      </h1>
      <form>
        <input
          type="text"
          placeholder="Enter User Name"
          onChange={changeHandler}
        />
      </form>
    </div>
  );
}
export default App;
```

I've named this update to my app Input element in React, pt 8*.

Next, let's add more than just a single input to our app.

*https://codesandbox.io/s/input-element-in-react-pt-8-57v5j0

2.6 Adding the age input

In this improved version of the app I'm working on, I'll add the age input.

```
import React from "react";

function App() {
  const [user, setUser] = React.useState({ name: "James", age: 21 });

  const changeName = (e) => {
    setUser((oldState) => {
      return { ...oldState, name: e.target.value };
    });
  };

  const changeAge = (e) => {
    setUser((oldState) => {
      return { ...oldState, age: e.target.value };
    });
  };

  return (
    <div>
      <h1>
        {user.name}, {user.age}
      </h1>
      <form>
        <input
          type="text"
          placeholder="Enter User Name"
          onChange={changeName}
        />
        <input
          type="number"
          min="18"
          max="130"
          placeholder="Enter User Age"
          onChange={changeAge}
        />
      </form>
    </div>
  );
```

```
}
export default App;
```

I've named this update to my app Building a form React, pt 1*.

The app works, but there is an issue that has potential to make it a lot more complex than it needs to be.

Basically, with the current setup, every new form element I add will require its own function.

If my form had, say, 10 inputs, I'd need to code 10 functions, where each function would have the following code:

```
setUser((oldState) => {
    return { ...oldState, age: e.target.value };
});
```

Essentially, this is the same function. The only change that it gets is in the key. In this example, the key is `age`, and for the previous input field, it would be `name`.

Luckily, JavaScript comes to the rescue here, and allows me to use its dynamic nature to keep my code from being repetitive.

This solution takes a bit of time to wrap your head around it if you haven't come across it before.

Luckily, I've already covered it in detail in my other book.

The text that follows is an excerpt from my book titled: A Better Way to Learn JavaScript, Book 1: The Basics†.

2.6.1 Function expressions as shorthand methods with the computed properties syntax

To understand how shorthand ES6 methods work with the computed properties syntax, we'll work with our `car` object again. This time, there's a twist: we'll return it from a function.

Note that while the below function is a function declaration (not a function expression!), we'll build up to having shorthand computed properties methods inside of it.

But let's not get ahead of ourselves, and instead, let's take things one step at a time.

First, we'll write a function declaration, and call it `getACar`. This function will return an object:

*https://codesandbox.io/s/building-a-form-react-pt-1-xoild4
†https://leanpub.com/a-better-way-to-learn-javascript/

```
function getACar() {
    return {
        lights: "off",
        lightsOn() {
            this.lights = "on";
        },
        lightsOff() {
            this.lights = "off";
        },
        lightsStatus() {
            console.log(`The lights are ${this.lights}`);
        }
    }
}
```

So far so good, we're just returning our `car` object, which is exactly the same object like we had before. We can now call the `getACar` function, and it will return our `car` object.

```
getACar();
```

We can even call a method on our returned object, like this:

```
getACar().lightsStatus(); // returns: "The lights are off";
```

Now let's extend the returned `car` object, like this:

```
function getACar(carType, color, year) {
    return {
        carType: carType,
        color: color,
        year: year,
        lights: "off",
        lightsOn() {
            this.lights = "on";
        },
        lightsOff() {
            this.lights = "off";
        },
        lightsStatus() {
            console.log(`The lights are ${this.lights}`);
        },
        getType() {
```

```
            return this.carType
        },
        getColor() {
            return this.color
        },
        getYear() {
            return this.year
        }
    }
}
getACar("sports car", "red", "2020").getType(); // returns: "sports car"
```

Now comes the fun part: to access a property in an object, we can either use the dot notation, or the brackets notation.

This means that the following syntax is completely legitimate:

```
let aCarObj = {
    [year]: "2020",
    [color]: "red",
    [carType]: "sports car"
}
aCarObj.carType; // returns: Uncaught ReferenceError: year is not defined
```

The returned error is completely expected, since we haven't declared the year variable anywhere. Let's fix the error, like this:

```
let year = "2020";
let color = "red";
let carType = "sports car";

let aCarObj = {
    [year]: "2020",
    [color]: "red",
    [carType]: "sports car"
}
aCarObj.carType; // returns: undefined
```

Now, trying to access the `carType` property returns `undefined` instead of `"sports car"`. Just why that is so, can be seen if we just inspect the entire object:

```
aCarObj;
```

The returned object looks like this:

```
{2020: "2020", red: "red", sports car: "sports car"}
```

This brings us to an a-ha moment: **the brackets notation makes it possible to run expressions inside of them**. Actually, this is the regular behavior of the brackets notation.

What that means is that the values of the **computed properties are indeed computed**, which means that the values of the variables are evaluated inside brackets and **used as object properties' keys**.

Let's rewrite the `aCarObj`:

```
let a = "year";
let b = "color";
let c = "carType";

let aCarObj = {
    [a]: "2020",
    [b]: "red",
    [c]: "sports car"
}
aCarObj.carType; // returns: "sports car"
```

In the above code, the value of `[a]` was evaluated to `year`; the value of `[b]` was evaluated to `color`, and the value of `[c]` was evaluated to `carType`.

This brings us to the point of this section. Just like we can use computed properties in the above use case, we can use them on shorthand object methods, like this:

```
let lightsOnMethod = "lightsOn";
let lightsOffMethod = "lightsOff";
let lightsStatusMethod = "lightsStatus";
let getTypeMethod = "getType";
let getColorMethod = "getColor";
let getYearMethod = "getYear";

function getACar(carType, color, year) {
    return {
        carType: carType,
        color: color,
        year: year,
        lights: "off",
```

```
14          [lightsOnMethod]() {
15              this.lights = "on";
16          },
17          [lightsOffMethod]() {
18              this.lights = "off";
19          },
20          [lightsStatusMethod]() {
21              console.log(`The lights are ${this.lights}`);
22          },
23          [getTypeMethod]() {
24              return this.carType
25          },
26          [getColorMethod]() {
27              return this.color
28          },
29          [getYearMethod]() {
30              return this.year
31          }
32      }
33  }
34  getACar("sports car", "red", "2020").getType(); // returns: "sports car"
```

This is the end of the excerpt from my book titled: A Better Way to Learn JavaScript, Book 1: The Basics*.

2.6.2 Using computed properties to simplify the event-handling function in forms

Now that you hopefully understand this concept, let's implement it so that it doesn't follow the WET rule ("Write Everything Thrice"), and instead use the DRY rule ("Don't Repeat Yourself).

So, I ended up with my app having the following code:

*https://leanpub.com/a-better-way-to-learn-javascript/

```
import React from "react";

function App() {
  const [user, setUser] = React.useState({ name: "James", age: 21 });

  const changeName = (e) => {
    setUser((oldState) => {
      return { ...oldState, name: e.target.value };
    });
  };

  const changeAge = (e) => {
    setUser((oldState) => {
      return { ...oldState, age: e.target.value };
    });
  };

  return (
    <div>
      <h1>
        {user.name}, {user.age}
      </h1>
      <form>
        <input
          type="text"
          placeholder="Enter User Name"
          onChange={changeName}
        />
        <input
          type="number"
          min="18"
          max="130"
          placeholder="Enter User Age"
          onChange={changeAge}
        />
      </form>
    </div>
  );
}
export default App;
```

And the issue that I was having were the duplicate event-handling functions - specifically, one event-handling function for each element in the form.

Now, based on the possibilities of making dynamic keys in our objects, I'll update my code to the following:

```
import React from "react";

function App() {
  const [user, setUser] = React.useState({ name: "James", age: 21 });

  function changeHandler(e) {
    setUser({ ...user, [e.target.name]: e.target.value });
  }

  return (
    <div>
      <h1>
        {user.name}, {user.age}
      </h1>
      <form>
        <input
          type="text"
          placeholder="Enter User Name"
          onChange={changeHandler}
          name="name"
        />
        <input
          type="number"
          min="18"
          max="130"
          placeholder="Enter User Age"
          onChange={changeHandler}
          name="age"
        />
      </form>
    </div>
  );
}
export default App;
```

Notice that I've used another little trick here, which is the use of the `name` attribute in my inputs.

I've named this update to my app Building a form React, pt 2*.

*https://codesandbox.io/s/building-a-form-react-pt-2-qhq4xd

So, I've used **computed properties** to dynamically set the key based on the event instance object, that is, based on `e.target.name`.

Forms are probably one of the more difficult beginner-level concepts, because in HTML, they keep their own state. Next, I'll discuss this topic and how it relates to something known as "controlled components".

2.7 Controlled components in React

The issue with form elements in React is that each of them, in plain HTML, keeps its state. Then, on top of that, we have React state to handle state changes.

So, to keep things working the way they should in React, we need to make our form component a controlled component.

This term, "controlled component", just refers to the fact that the state of my form is controlled by React.

Doing this is extremely simple: you just add a `value` attribute to your input, and set it equal to `user` (as the state object), followed by the dot, and then the correct state object's property. For name, it will be `user.name`, and for age, it will be `user.age`.

Here's the updated app's code:

```
import React from "react";

function App() {
  const [user, setUser] = React.useState({ name: "James", age: 21 });

  function changeHandler(e) {
    setUser({ ...user, [e.target.name]: e.target.value });
  }

  return (
    <div>
      <h1>
        {user.name}, {user.age}
      </h1>
      <form>
        <input
          type="text"
          placeholder="Enter User Name"
          onChange={changeHandler}
          name="name"
          value={user.name}
```

```
22          />
23          <input
24            type="number"
25            min="18"
26            max="130"
27            placeholder="Enter User Age"
28            onChange={changeHandler}
29            name="age"
30            value={user.age}
31          />
32        </form>
33      </div>
34    );
35  }
36  export default App;
```

I've named this update to my app Building a form React, pt 3*.

Now, all the individual form elements are **controlled by React** - hence the term "controlled compoentns".

2.8 Other form inputs: radio buttons, checkboxes, textareas, and selects

Forms in React can feel a bit off when compared to what you're likely familiar with - that is, those same elements in plain HTML.

That's why in this section I'll discuss some common form elements and I'll show you how they work in React.

2.8.1 Radio buttons

Let's discuss radio buttons.

Radio buttons allow the user to choose one of the available options.

In this example, I'm adding four options.

*https://codesandbox.io/s/building-a-form-react-pt-3-bg2p29

```
<div style={{ marginBottom: "20px" }}>
    <input type="radio" name="preferences" id="cats" />
    <label htmlFor="cats">I'm a cats person</label>

    <input type="radio" name="preferences" id="dogs" />
    <label htmlFor="dogs">I'm a dogs person</label>

    <input type="radio" name="preferences" id="parrots" />
    <label htmlFor="parrots">I'm a parrots person</label>

    <input type="radio" name="preferences" id="fish" />
    <label htmlFor="fish">I like fish</label>
</div>
```

So, a working radio button is actually two elements working as a single element.

1. the self-closing `input` element
2. the `label` element, which has an opening and a closing tag, with the text node in between being clickable by the user in the rendered app

The `input` element's state is controlled using the `name` attribute, and the `label` has a React-specific `htmlFor` attribute which conncets the `input` element's `id` attribute with the `label` element's `htmlFor` attribute. In other words, the `id` of "fish" in the `input` matches the `htmlFor` of "fish" in the `label` element.

Notice also that each of the radio buttons has the exact same `name` attribute. This is so that only one is clickable - that is, a user can select a single radio button, which excludes all the others at any given moment.

Now that I have the structure set up, I still need to update my local state, and add the `value` attribute to make it trully controlled.

```
import React from "react";

function App() {
  const [user, setUser] = React.useState({
    name: "James",
    age: 21,
    preferences: "dogs"
  });

  function changeHandler(e) {
    setUser({ ...user, [e.target.name]: e.target.value });
```

```
  }

  return (
    <div>
      <h1>
        {user.name}, {user.age}, likes {user.preferences}
      </h1>
      <form>
        <div style={{ marginBottom: "20px" }}>
          <input
            type="text"
            placeholder="Enter User Name"
            onChange={changeHandler}
            name="name"
            value={user.name}
          />
        </div>

        <div style={{ marginBottom: "20px" }}>
          <input
            type="number"
            min="18"
            max="130"
            placeholder="Enter User Age"
            onChange={changeHandler}
            name="age"
            value={user.age}
          />
        </div>

        <div style={{ marginBottom: "20px" }}>
          <input
            type="radio"
            name="preferences"
            id="cats"
            value="cats"
            onChange={changeHandler}
          />
          <label htmlFor="cats">I'm a cats person</label>

          <input
            type="radio"
            name="preferences"
```

```
55              id="dogs"
56              value="dogs"
57              onChange={changeHandler}
58            />
59            <label htmlFor="dogs">I'm a dogs person</label>
60
61            <input
62              type="radio"
63              name="preferences"
64              id="parrots"
65              value="parrots"
66              onChange={changeHandler}
67            />
68            <label htmlFor="parrots">I'm a parrots person</label>
69
70            <input
71              type="radio"
72              name="preferences"
73              id="fish"
74              value="fish"
75              onChange={changeHandler}
76            />
77            <label htmlFor="fish">I like fish</label>
78          </div>
79        </form>
80      </div>
81    );
82  }
83  export default App;
```

Notice that although I've added the `preferences` property on my state object, to make it work properly and reflect the radio button clicks, I also needed to add the `onChange={changeHandler}` to each of the radio button `input` elements.

I've named this update to my app Building a form React, pt 4*.

Next, let's discuss checkboxes.

2.8.2 Checkboxes

The default HTML behavior of checkboxes, contrary to radio buttons, is that you can select multiple checkboxes at the same time.

*https://codesandbox.io/s/building-a-form-react-pt-4-78vnds

Checkboxes are essentially a true/false state, implemented as an `input` element. What do I mean by that?

Here's an example:

```
<input type="checkbox" checked>
```

The type attribute on the `input` element makes this a checkbox. The `checked` attribute is the true/false state.

Additionally, the `id` attributes ties in with a `label` element, similar to how it works in radio buttons.

```
<input type="checkbox" id="olives" checked />
<label htmlFor="olives">Olives side dish</label>
```

As far as keeping state goes, this is where things diverge from what I've showed you for other input elements.

Before, we were controlling state using the `value` attribute. For checkboxes, we'll need to use the `checked` property in the state object, and we'll need to set this property to a value that evaluates either to `true` or `false`.

Here's my update to state:

```
const [user, setUser] = React.useState({
    name: "James",
    age: 21,
    preferences: "dogs",
    olives: true
});
```

And here's the update to the `return` statement:

```
<div>
    <input type="checkbox" id="olives" onChange={changeHandler} checked />
    <label htmlFor="olives">Olives side dish</label>
</div>
```

And now I need to update my `changeHandler` function so that it can work with the `checked` attribute as well:

```
  function changeHandler(e) {
    const name = e.target.name;
    const value =
      e.target.type === "checkbox" ? e.target.checked : e.target.value;

    setUser({ ...user, [name]: value });
  }
```

The fully updated code now looks as follows:

```
import React from "react";

function App() {
  const [user, setUser] = React.useState({
    name: "James",
    age: 21,
    preferences: "dogs",
    olives: true
  });

  function changeHandler(e) {
    const name = e.target.name;
    const value =
      e.target.type === "checkbox" ? e.target.checked : e.target.value;

    setUser({ ...user, [name]: value });
  }

  return (
    <div>
      <h1>
        {user.name}, {user.age}, likes {user.preferences}
      </h1>
      <h2>Favorite side dish: olives - {user.olives ? "yes" : "no"}</h2>
      <form>
        <div style={{ marginBottom: "20px" }}>
          <input
            type="text"
            placeholder="Enter User Name"
            onChange={changeHandler}
            name="name"
            value={user.name}
          />
```

```
34          </div>
35
36          <div style={{ marginBottom: "20px" }}>
37            <input
38              type="number"
39              min="18"
40              max="130"
41              placeholder="Enter User Age"
42              onChange={changeHandler}
43              name="age"
44              value={user.age}
45            />
46          </div>
47
48          <div style={{ marginBottom: "20px" }}>
49            <input
50              type="radio"
51              name="preferences"
52              id="cats"
53              value="cats"
54              onChange={changeHandler}
55            />
56            <label htmlFor="cats">I'm a cats person</label>
57
58            <input
59              type="radio"
60              name="preferences"
61              id="dogs"
62              value="dogs"
63              onChange={changeHandler}
64            />
65            <label htmlFor="dogs">I'm a dogs person</label>
66
67            <input
68              type="radio"
69              name="preferences"
70              id="parrots"
71              value="parrots"
72              onChange={changeHandler}
73            />
74            <label htmlFor="parrots">I'm a parrots person</label>
75
76            <input
```

```
77              type="radio"
78              name="preferences"
79              id="fish"
80              value="fish"
81              onChange={changeHandler}
82            />
83            <label htmlFor="fish">I like fish</label>
84          </div>
85
86          <div>
87            <input type="checkbox" id="olives" onChange={changeHandler} checked />
88            <label htmlFor="olives">Olives side dish</label>
89          </div>
90        </form>
91      </div>
92    );
93  }
94  export default App;
```

I've named this temporary update to my app Building a form React, pt 5*.

2.8.3 Improving radio buttons and checkboxes

Although the form app works, you might have noticed a few things that could be improved.

1. The label on the checkbox input does not work. When you click it, nothing happens.
2. The radio buttons work, but initially, none are selected

Let's fix the first issue first.

So, clicking the label for the checkbox doesn't toggle the checkbox. This is because the `label` element is not connected to the `input` element.

The reason why it's not connected is because I've omitted the `id` attribute on the `input` element.

Let's fix that:

*https://codesandbox.io/s/building-a-form-react-pt-5-xprhh4

```
<input
  type="checkbox"
  name="olives"
  id="olives"
  onChange={changeHandler}
  checked={user.olives}
/>
<label htmlFor="olives">Olives side dish</label>
```

I've done this on purpose, so that I can emphazize the importance of the `id` and `name` attributes.

The `id` attribute is there to connect the `label` element to the `input` element.

The `name` attribute is there to connect the `input` element to the `state` object.

The `id` attribute is not required for the `input` element to work, but it is required for the `label` element to work.

Secondly, let's fix the radio buttons.

I'll fix them by adding the `checked` attribute to each of the radio buttons, and I'll add some JSX evaluation so that it properly returns either `true` or `false`:

```
import React from "react";

function App() {
  const [user, setUser] = React.useState({
    name: "James",
    age: 21,
    preferences: "dogs",
    olives: true
  });

  function changeHandler(e) {
    const name = e.target.name;
    const value =
      e.target.type === "checkbox" ? e.target.checked : e.target.value;

    setUser({ ...user, [name]: value });
  }

  return (
    <div>
      <h1>
        {user.name}, {user.age}, likes {user.preferences}
      </h1>
```

```
24        <h2>Favorite side dish: olives - {user.olives ? "yes" : "no"}</h2>
25        <form>
26          <div style={{ marginBottom: "20px" }}>
27            <input
28              type="text"
29              placeholder="Enter User Name"
30              onChange={changeHandler}
31              name="name"
32              value={user.name}
33            />
34          </div>
35
36          <div style={{ marginBottom: "20px" }}>
37            <input
38              type="number"
39              min="18"
40              max="130"
41              placeholder="Enter User Age"
42              onChange={changeHandler}
43              name="age"
44              value={user.age}
45            />
46          </div>
47
48          <div style={{ marginBottom: "20px" }}>
49            <input
50              type="radio"
51              name="preferences"
52              id="cats"
53              value="cats"
54              checked={user.preferences === "cats"}
55              onChange={changeHandler}
56            />
57            <label htmlFor="cats">I'm a cats person</label>
58
59            <input
60              type="radio"
61              name="preferences"
62              id="dogs"
63              value="dogs"
64              checked={user.preferences === "dogs"}
65              onChange={changeHandler}
66            />
```

```
67            <label htmlFor="dogs">I'm a dogs person</label>
68
69            <input
70              type="radio"
71              name="preferences"
72              id="parrots"
73              value="parrots"
74              checked={user.preferences === "parrots"}
75              onChange={changeHandler}
76            />
77            <label htmlFor="parrots">I'm a parrots person</label>
78
79            <input
80              type="radio"
81              name="preferences"
82              id="fish"
83              value="fish"
84              checked={user.preferences === "fish"}
85              onChange={changeHandler}
86            />
87            <label htmlFor="fish">I like fish</label>
88          </div>
89
90          <div>
91            <input
92              type="checkbox"
93              name="olives"
94              id="olives"
95              checked={user.olives}
96              onChange={changeHandler}
97            />
98            <label htmlFor="olives">Olives side dish</label>
99          </div>
100       </form>
101     </div>
102   );
103 }
104 export default App;
```

I've named this update to my app Building a form React, pt 6*.

*https://codesandbox.io/s/building-a-form-react-pt-6-z2ju76

2.8.3 Textareas

In plain HTML, the `textarea` form element has an opening and a closing tag, with the starting state value in between these tags, as a text node.

```
<textarea>Starting state value</textarea>
```

In React, effort was made to make this element as close to others as possible:

```
<textarea value={"Starting state value"} />
```

Let's now extend our previous app example's form with a textarea.

2.8.4 Selects

asdf

2.8.5 ???

asdfa

2.9 Form submission and preventDefault

asdf

2.10 ???

asdf

Chapter 3: Revising components and JSX

3.1 asdf

asdf

Chapter 4: Build a todo app

4.1 asdf

asdf

Chapter 5: Popular form libraries

5.1 asdf

asdf

Chapter 6: Popular component libraries

6.1 asdf

asdf

Chapter 7: Build a portfolio app

7.1 asdf

asdf

Chapter 8: Deploy a portfolio app to Netlify

8.1 asdf

asdf