Watch this chapter's introduction video by scanning this QR code with your phone or going to http://reactquickly.co/videos/ch07.

7 Working with forms in React

This chapter covers

- Defining forms and form elements
- Capturing data changes
- Using references to access data
- Alternative approaches for capturing user-input data from form elements
- Setting default values for form elements

Thus far, you've learned about events, states, component composition, and other important React topics, features, and concepts. But aside from capturing user events, I haven't covered how to capture text input and input via other form elements like `input`, `textarea`, and `option`. Working with them is paramount to web development, because they allow your applications to receive data (such as text) and actions (such as clicks) from users.

This chapter refers to pretty much everything I've covered so far. You'll begin to see how everything fits together.

NOTE The source code for the examples in this chapter is at www.manning.com/books/react-quickly and https://github.com/azat-co/react-quickly/tree/master/ch07 (in the ch07 folder of the GitHub repository https://github.com/azat-co/react-quickly). You can also find some demos at http://reactquickly.co/demos.

7.1 The recommended way to work with forms in React

In regular HTML, when you're working with an input element, the page's DOM maintains that element's value in its DOM node. It's possible to access the value via methods like `document.getElementById('email').value` or by using jQuery methods. In essence, the DOM is your storage.

In React, when you're working with forms or any other user-input fields such as standalone text fields or buttons, you have an interesting problem to solve. The React documentation says, "React components must represent the state of the view at any point in time and not only at initialization time." React is all about keeping things simple by using declarative style to describe UIs. React describes the UI: its end stage, how it should look.

Can you spot a conflict? In traditional HTML form elements, the states of elements change with user input. But React uses a declarative approach to describe UIs. Input needs to be dynamic to reflect the state properly.

Thus, opting *not* to maintain the component state (in JavaScript) and not to sync it with the view adds problems; there may be a situation when the internal state and view are different. React won't know about the changed state. This can lead to all sorts of trouble and bugs, and negates React's simple philosophy. The best practice is to keep React's `render()` as close to the real DOM as possible—and that includes the data in the form elements.

Consider the following example of a text-input field. React must include the new value in its `render()` for that component. Consequently, you need to set the value for the element to a new value using `value`. But if you implement an `<input>` field as in HTML, React will always keep `render()` in sync with the real DOM. React won't allow users to change the value. Try it yourself. It's peculiar, but that's the appropriate behavior for React!

```
render() {
  return <input type="text" name="title" value="Mr." />
}
```

This code represents the view at any state, so the value will *always* be `Mr.`. On the other hand, input fields must change in response to the user clicking or typing. Given these points, let's make the value dynamic. This is a better implementation, because it'll be updated from the state:

```
render() {
  return <input type="text" name="title" value={this.state.title} />
}
```

But what's the value of `state`? React can't know about users typing in the form elements. You need to implement an event handler to capture changes with `onChange`:

```
handleChange(event) {
  this.setState({title: event.target.value})
}
render() {
  return <input type="text" name="title" value={this.state.title}
  ➥ onChange={this.handleChange.bind(this)}/>
}
```

Given these points, the best practice is to implement these things to sync the internal state with the view (see figure 7.1):

1. Define elements in `render()` using values from `state`.
2. Capture changes to a form element as they happen, using `onChange`.
3. Update the internal state in the event handler.
4. New values are saved in `state`, and then the view is updated by a new `render()`.

It may seem like a lot of work at first glance, but I hope that when you've used React more, you'll appreciate this approach. It's called *one-way binding* because the state changes views, and that's it. There's no trip back: only a one-way trip from state to view. With one-way binding, a library won't update the state (or the model) automatically. One of the main benefits of one-way binding is that it removes complexity when you're working with large apps where many views implicitly can update many states (data models) and vice versa (see figure 7.2).

Simple doesn't always mean writing less code. Sometimes, as in this case, you'll have to write extra code to manually set the data from event handlers to the state (which is rendered to the view); but this approach tends to be superior when it comes to complex UIs and single-page applications with myriads of views and states. Simple isn't always easy.

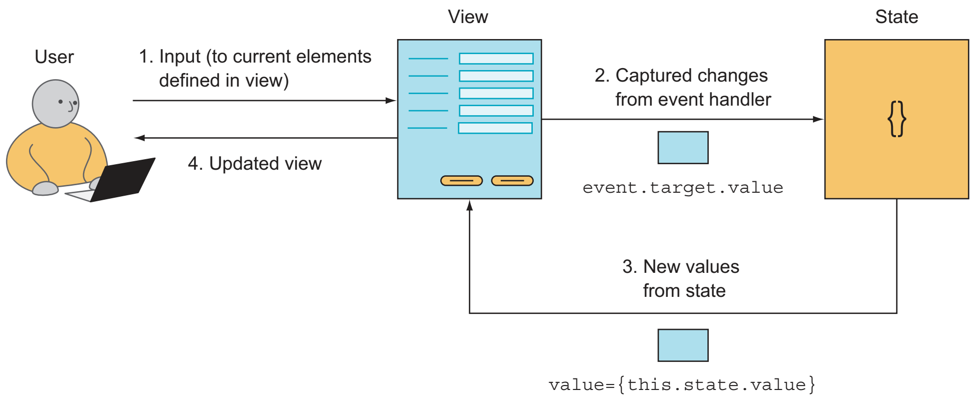


Figure 7.1 The *correct* way to work with form elements: from user input to events, then to the state and the view

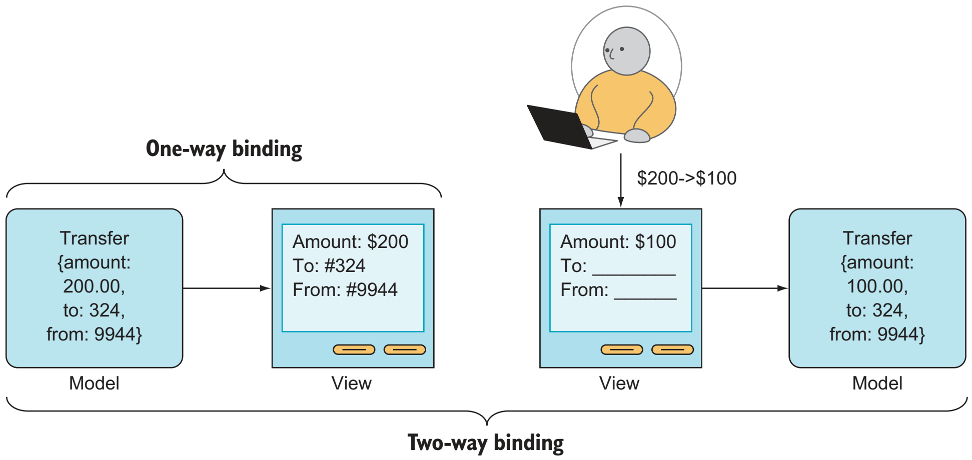


Figure 7.2 One-way binding is responsible for the model-to-view transition. Two-way binding also handles changes from view to model.

Conversely, two-way binding makes it possible for views to change states automatically without you explicitly implementing the process. Two-way binding is how Angular 1 works. Interestingly, Angular 2 borrowed the concept of one-way binding from React and made it the default (you can still have two-way binding explicitly).

For this reason, I'll first cover the recommended approach of working with forms. It's called using *controlled components*, and it ensures that the internal component state is always in sync with the view. Controlled form elements are called that because React controls or sets the values. The alternative approach is *uncontrolled components*, which I'll discuss in section 7.2.

You've learned the best practice of working with input fields in React: capturing the change and applying it to the state as shown in figure 7.1 (input to changed view). Next, let's look at how you define a form and its elements.

7.1.1 Defining a form and its events in React

Let's start with the `<form>` element. Typically, you don't want input elements hanging around randomly in the DOM. This situation can turn bad if you have many functionally different sets of inputs. Instead, you wrap input elements that share a common purpose in a `<form></form>` element.

Having a `<form>` wrapper isn't required. It's fine to use form elements by themselves in simple UIs. In more-complex UIs, where you may have multiple groups of elements on a single page, it's wise to use `<form>` for each such group. React's `<form>` is rendered like an HTML `<form>`, so whatever rules you have for the HTML form will apply to React's `<form>` element, too. For example, according to the HTML5 spec, you should *not* nest forms.[1]

[1] The specification says content must be flow content, but with no `<form>` element descendants. See www.w3.org/TR/html5/forms.html#the-form-element.

The `<form>` element can have events. React supports three events for forms in addition to the standard React DOM events (as outlined in table 6.1):

- `onChange`—Fires when there's a change in any of the form's input elements.
- `onInput`—Fires for each change in `<textarea><input>` element values. The React team doesn't recommend using it (see the accompanying sidebar).
- `onSubmit`—Fires when the form is submitted, usually by pressing Enter.

onChange vs. onInput

React's `onChange` fires on every change, in contrast to the DOM's change event (http://mng.bz/lJ37), which may not fire on each value change but fires on lost focus. For example, for `<input type="text">`, a user can be typing with no `onChange`; only after the user presses Tab or clicks away with their mouse to another element (lost focus) is `onChange` fired in HTML (regular browser event). As mentioned earlier, in React, `onChange` fires on each keystroke, not just on lost focus. On the other hand, `onInput` in React is a wrapper for the DOM's `onInput`, which fires on each change.

The bottom line is that React's `onChange` works differently than `onChange` in HTML: it's more consistent and more like HTML's `onInput`. The recommended approach is to use `onChange` in React and to use `onInput` only when you need to access native behavior for the `onInput` event. The reason is that React's `onChange` wrapper behavior provides consistency and thus sanity.

In addition to the three events already listed, `<form>` can have standard React events such as `onKeyUp` and `onClick`. Using form events may come in handy when you need to capture a specific event for the entire form (that is, a group of input elements).

For example, it helps provide a good UX if you allow users to submit data when they press Enter (assuming they're not in a `textarea` field, in which case Enter should create a new line). You can listen to the form-submit event by creating an event listener that triggers `this.handleSubmit()`:

```
handleSubmit(event) {
  ...
}
render() {
  <form onSubmit={this.handleSubmit}>
    <input type="text" name="email" />
  </form>
}
```

NOTE You need to implement the `handleSubmit()` function outside of `render()`, just as you'd do with any other event. React doesn't require a naming convention, so you can name the event handler however you wish as long as the name is understandable and somewhat consistent. This book sticks with the most popular convention: prefixing event handlers with the word `handle` to distinguish them from regular class methods.

NOTE As a reminder, don't invoke a method (don't put parentheses) and don't use double quotes around curly braces (correct: `EVENT={this.METHOD}`) when setting the event handler. For some of you, this is basic JavaScript and straightforward, but you wouldn't believe how many times I've seen errors related to these two misunderstandings in React code: you pass the definition of the function, not its result; and you use curly braces as values of the JSX attributes.

Another way to implement form submission on Enter is to manually listen to the key-up event (`onKeyUp`) and check for the key code (13 for Enter):

```
handleKeyUp(event) {
  if (event.keyCode == 13) return this.sendData()
}
render() {
  return <form onKeyUp={this.handleKeyUp}>
  ...
  </form>
}
```

Note that the `sendData()` method is implemented somewhere else in the class/component. Also, for `this.sendData()` to work, you'll need to use `bind(this)` to bind the context to the event handler in `constructor()`.

To summarize, you can have events on the form element, not just on individual elements in the form. Next, we'll look at how to define form elements.

7.1.2 Defining form elements

You implement almost all input fields in HTML with just four elements: `<input>`, `<textarea>`, `<select>`, and `<option>`. Do you remember that in React, properties are immutable? Well, form elements are special because users need to interact with the elements and change these properties. For all other elements, this is impossible.

React made these elements special by giving them the mutable properties `value`, `checked`, and `selected`. These special mutable properties are also called *interactive properties*.

NOTE React DOM also supports other elements related to building forms, such as `<keygen>`, `<datalist>`, `<fieldset>`, and `<label>`. These elements don't possess superpowers like a mutable `value` attribute/property. They're rendered as the corresponding HTML tags. For this reason, this book focuses only on the four main elements with superpowers.

Here's a list of the interactive properties/fields (ones that can change) you can read from events like `onChange` attached to form elements (covered in section 6.1.3):

- `value`—Applies to `<input>`, `<textarea>`, and `<select>`
- `checked`—Applies to `<input>` with `type="checkbox"` and `type="radio"`
- `selected`—Applies to `<option>` (used with `<select>`)

You can read the values and change them by working with these interactive (mutable) properties. Let's look at some examples of how to define each of the elements.

THE <INPUT> ELEMENT

The `<input>` element renders multiple fields by using different values for its `type` attribute:

- `text`—Plain text-input field.
- `password`—Text-input field with a masked display (for privacy).
- `radio`—Radio button. Use the same name to create a group of radio buttons.
- `checkbox`—Check box element. Use the same name to create a group.
- `button`—Button form element.

The main use case for all `<input>` type elements—except check boxes and radio buttons—is to use `value` as the element's interactive/changeable property. For example, an email input field can use the `email` state and `onChange` event handler:

```
<input
  type="text"
  name="email"
  value={this.state.email}
  onChange={this.handleEmailChange}/>
```

The two exceptions that don't have `value` as their primary mutable attribute are inputs with the types `checkbox` and `radio`. They use `checked` because these two types have one value per HTML element, and thus the value doesn't change, but the state of `checked`/selected does. For example, you can define three radio buttons in one group (`radioGroup`) by defining these three elements, as shown in figure 7.3.

As mentioned earlier, the values (`value`) are hardcoded because you don't need to change them. What changes with user actions is the element's `checked` attribute, as shown in the following listing (ch07/elements/jsx/content.jsx).

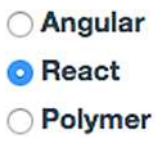


Figure 7.3 Radio button group

Listing 7.1 Rendering radio buttons and handling changes

```
class Content extends React.Component {
  constructor(props) {
    super(props)
    this.handleRadio = this.handleRadio.bind(this)
    ...
    this.state = {
      ...
      radioGroup: {
        angular: false,
        react: true,          <-- Sets the default checked
        polymer: false            radio button in the state
      }
    }
  }
  handleRadio(event) {
    let obj = {}  // erase other radios
    obj[event.target.value] = event.target.checked // true
    this.setState({radioGroup: obj})
  }
  ...
  render() {
    return <form>
       <input type="radio"
        name="radioGroup"
        value='angular'
        checked={this.state.radioGroup['angular']}
        onChange={this.handleRadio}/>
      <input type="radio"
        name="radioGroup"
        value='react'
        checked={this.state.radioGroup['react']}
        onChange={this.handleRadio}/>
      <input type="radio"
        name="radioGroup"
        value='polymer'
        checked={this.state.radioGroup['polymer']}
        onChange={this.handleRadio}/>
      ...
    </form>
  }
}
```

Uses the target.checked attribute to get a Boolean that indicates whether this radio button is selected

Uses an attribute from the state object or any state attribute

Uses the same onChange event handler because you can get the radio button value from target.value

For check boxes, you follow an approach similar to that for radio buttons: using the `checked` attribute and Boolean values for states. Those Booleans can be stored in a `checkboxGroup` state:

```
class Content extends React.Component {
  constructor(props) {
    super(props)
    this.handleCheckbox = this.handleCheckbox.bind(this)
    // ...
    this.state = {
      // ...
      checkboxGroup: {
        node: false,
        react: true,
        express: false,
        mongodb: false
      }
    }
  }
```

Then the event handler (which you bind in the constructor) grabs the current values, adds true or false from `event.target.value`, and sets the state:

```
handleCheckbox(event) {
  let obj = Object.assign(this.state.checkboxGroup)
  obj[event.target.value] = event.target.checked          <-- True or false
  this.setState({checkboxGroup: obj})
}
```

There's no need for the assignment from the state in `radio`, because radio buttons can have only one selected value. Thus, you use an empty object. This isn't the case with check boxes: they can have multiple values selected, so you need a merge, not a replace.

In JavaScript, objects are passed and assigned by references. So in the statement `obj = this.state.checkboxGroup`, `obj` is really a state. As you'll recall, you aren't supposed to change the state directly. To avoid any potential conflicts, it's better to assign by value with `Object.assign()`. This technique is also called *cloning*. Another, less effective and more hacky way to assign by value is to use JSON:

```
clonedData = JSON.parse(JSON.stringify(originalData))
```

When you're using state arrays instead of objects and need to assign by value, use `clonedArray = Array.from(originArray)` or `clonedArray = originArray.slice()`.

You can use the `handleCheckbox()` event handler to get the value from `event.target.value`. The next listing shows `render()` (ch07/elements/jsx/content.jsx), which uses the state values for four check boxes, as shown in figure 7.4.

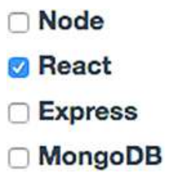


Figure 7.4 Rendering check boxes with React as the preselected option

Listing 7.2 Defining check boxes

```
<input type="checkbox"
  name="checkboxGroup"
  value='node'
  checked={this.state.checkboxGroup['node']}              <-- Uses state as a value. It can
  onChange={this.handleCheckbox}/>                            be an attribute of an object
<input type="checkbox"                                        or just a state attribute.
  name="checkboxGroup"
  value='react'
  checked={this.state.checkboxGroup['react']}
  onChange={this.handleCheckbox}/>                        <-- Uses onChange to
<input type="checkbox"                                        capture actions
  name="checkboxGroup"
  value='express'
  checked={this.state.checkboxGroup.express}              <-- Uses dot notation when
  onChange={this.handleCheckbox}/>                            keys are valid JS names
<input type="checkbox"
  name="checkboxGroup"
  value='mongodb'
  checked={this.state.checkboxGroup['mongodb']}
  onChange={this.handleCheckbox}/>                        <-- No need to bind in the element,
                                                              due to binding in the constructor
                                                              (true for all check boxes)
```

In essence, when you're using check boxes or radio buttons, you can hardcode the value in each individual element and use `checked` as your mutable attribute. Let's see how to work with other input elements.

THE <TEXTAREA> ELEMENT

`<textarea>` elements are for capturing and displaying long text inputs such as notes, blog posts, code snippets, and so on. In regular HTML, `<textarea>` uses inner HTML (that is, children) for the value:

```
<textarea>
  With the right pattern, applications...
</textarea>
```

Figure 7.5 shows an example.

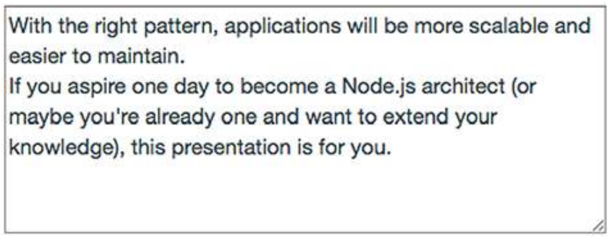

With the right pattern, applications will be more scalable and easier to maintain.
If you aspire one day to become a Node.js architect (or maybe you're already one and want to extend your knowledge), this presentation is for you.

Figure 7.5 Defining and rendering the `<textarea>` element

In contrast, React uses the `value` *attribute*. In view of this, setting a value as inner HTML/text is an antipattern. React will convert any children (if you use them) of `<textarea>` to the default value (more on default values in section 7.2.4):

```
<!-- Anti-pattern: AVOID doing this! -->
<textarea name="description">{this.state.description}</textarea>
```

Instead, it's recommended that you use the `value` attribute (or property) for `<textarea>`:

```
render() {
  return <textarea name="description" value={this.state.description}/>
}
```

To listen for the changes, use `onChange` as you would for `<input>` elements.

THE <SELECT> AND <OPTION> ELEMENTS

Select and option fields are great UX-wise for allowing users to select a single value or multiple values from a prepopulated list of values. The list of values is compactly hidden behind the element until users expand it (in the case of a single select), as shown in figure 7.6.

Figure 7.6 Rendering and preselecting the value of a drop-down

`<select>` is another element whose behavior is different in React compared to regular HTML. For instance, in regular HTML, you might use `selectDOMNode.selectedIndex` to get the index of the selected element, or `selectDOMNode.selectedOptions`. In React, you use `value` for `<select>`, as in the following example (ch07/elements/jsx/content.jsx).

Listing 7.3 Rendering form elements

```
...
constructor(props) {
  super(props)
  this.state = {selectedValue: 'node'}
}
handleSelectChange(event) {
  this.setState({selectedValue: event.target.value})
}
...
render() {
  return <form>
    <select
      value={this.state.selectedValue}
      onChange={this.handleSelectChange}>
        <option value="ruby">Ruby</option>
        <option value="node">Node</option>
        <option value="python">Python</option>
    </select>
  </form>
}
...
```

This code renders a drop-down menu and preselects the `node` value (which must be set in `constructor()`, as shown in figure 7.6). Yay for Node!

Sometimes you need to use a multiselect element. You can do so in JSX/React by providing the `multiple` attribute without any value (React defaults to true) or with the value `{true}`.

TIP Remember that for consistency, and to avoid confusion, I recommend wrapping all Boolean values in curly braces `{}` and not `""`. Sure, `"true"` and `{true}` produce the same result. But `"false"` will also produce true. This is because the string `"false"` is treated as true in JavaScript (truthy).

To preselect multiple items, you can pass an array of options to `<select>` via its `value` attribute. For example, this code preselects Meteor and React:

```
<select multiple={true} value={['meteor', 'react']}>
  <option value="meteor">Meteor</option>
  <option value="react">React</option>
  <option value="jQuery">jQuery</option>
</select>
```

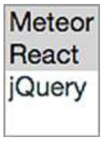


Figure 7.7 Rendering and preselecting multiselect elements

`multiple={true}` renders the multiselect element, and the Meteor and React values are preselected as shown in figure 7.7.

Overall, defining form elements in React isn't much different than doing so in regular HTML, except that you use `value` more often. I like this consistency. But defining is half the work; the other half is capturing the values. You did a little of that in the previous examples. Let's zoom in on event captures.

7.1.3 Capturing form changes

As mentioned earlier, to capture changes to a form element, you set up an `onChange` event listener. This event supersedes the normal DOM's `onInput`. In other words, if you need the regular HTML DOM behavior of `onInput`, you can use React's `onInput`. On the other hand, React's `onChange` isn't exactly the same as the regular DOM `onChange`. The regular DOM `onChange` may be fired only when the element loses focus, whereas React's `onChange` fires on all new input. What triggers `onChange` varies for each element:

- `<input>`, `<textarea>`, *and* `<select>`—`onChange` is triggered by a change in `value`.
- `<input>` *with type* `checkbox` *or* `radio`—`onChange` is triggered by a change in `checked`.

Based on this mapping, the approach to reading the value varies. As an argument of the event handler, you're getting a `SyntheticEvent`. It has a `target` property of `value`, `checked`, or `selected`, depending on the element.

To listen for changes, you define the event handler somewhere in your component (you can define it inline too, meaning in the JSX's `{}`) and create the `onChange` attribute pointing to your event handler. For example, this code captures changes from an email field (ch07/elements/jsx/content.jsx).

Listing 7.4 Rendering form elements and capturing changes

```
handleChange(event) {
  console.log(event.target.value)
}
render() {
  return <input
    type="text"
    onChange={this.handleChange}
    defaultValue="hi@azat.co"/>
}
```

Interestingly, if you don't define `onChange` but provide `value`, React will issue a warning and make the element read-only. If your intention is to have a read-only field, it's better to define it explicitly by providing `readOnly`. This will not only remove the warning, but also ensure that other programmers who read this code know this is a read-only field by design. To set the value explicitly, set the `readOnly` value to `{true}`—that is, `readOnly={true}`—or add the `readOnly` attribute by itself without the value, and React by default will add the value of `true` to the attribute.

Once you capture changes in elements, you can store them in the component's state:

```
handleChange(event) {
  this.setState({emailValue: event.target.value})
}
```

Sooner or later, you'll need to send this information to a server or another component. In this case, you'll have the values neatly organized in the state.

For example, suppose you want to create a loan application form that includes the user's name, address, telephone number, and Social Security number. Each input field handles its own changes. At the bottom of this form, you'll put a Submit button to send the state to the server. The following listing shows the name field with `onChange`, which keeps all input in the state (ch07/elements/jsx/content.jsx).

Listing 7.5 Rendering form elements

```
constructor(props) {
  super(props)
  this.handleInput = this.handleInput.bind(this)
  this.handleSubmit = this.handleSubmit.bind(this)
  ...
}
handleFirstNameChange(event) {
  this.setState({firstName: event.target.value})
}
...
handleSubmit() {
    fetch(this.props['data-url'], {method: 'POST', body:
    ➥ JSON.stringify(this.state)})
      .then((response)=>{return response.json()})
      .then((data)=>{console.log('Submitted: ', data)})
}
render() {
  return <form>
    <input name="firstName"
      onChange={this.handleFirstNameChange}
      type="text"/>
        ...
      <input
        type="button"
        onClick={this.handleSubmit}
        value="Submit"/>
  </form>
}
```

Captures changes to the firstName field by saving them to the state

Sends data to a URL from the data-url property with the Fetch promise-based browser API (experimental as of this writing, but supported by most modern browsers)

Defines an event handler to handle the Submit button

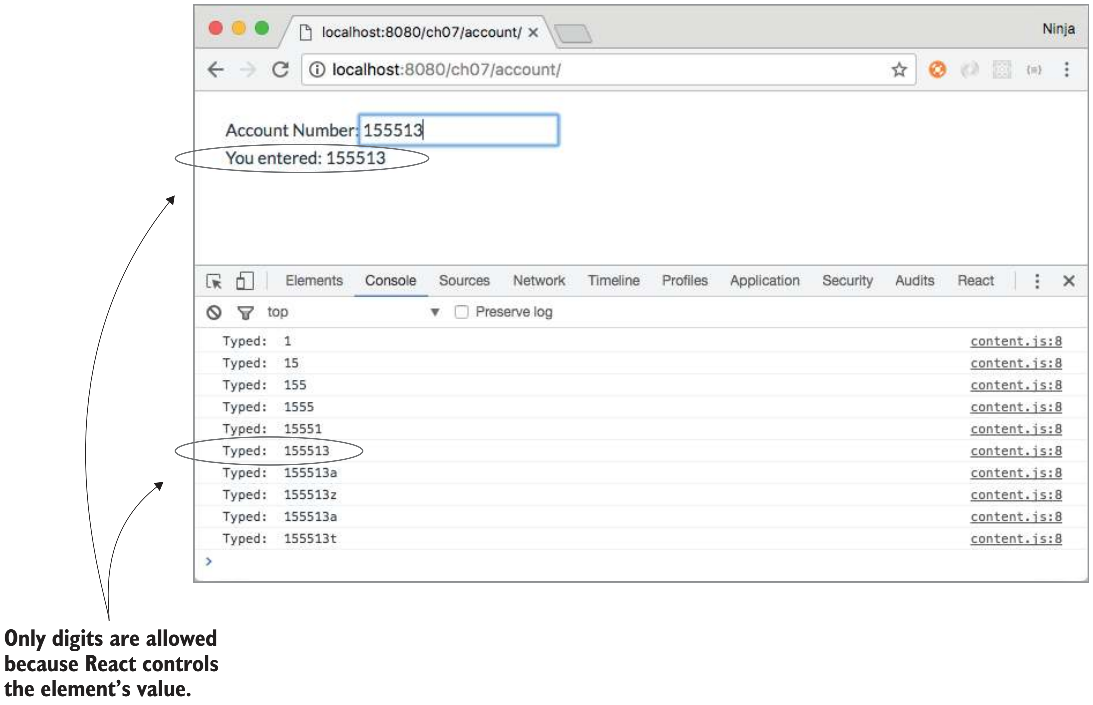
NOTE Fetch is an experimental native browser method to perform promise-based AJAX/XHR requests. You can read about its usage and support (it's supported by most modern browsers as of this writing) at http://mng.bz/mbMe.

You've learned how to define elements, capture changes with events, and update the state (which you use to display values). The next section walks through an example.

7.1.4 Account field example

Continuing with the loan application scenario, once the loan is approved, users need to be able to type in the number of the account to which they want their loan money transferred. Let's implement an account field component using your new skills. This is a controlled element, which is the best practice when it comes to working with forms in React.

In the component shown in listing 7.6 (ch07/account/jsx/content.jsx), you have an account number input field that needs to accept numbers only (see figure 7.8). To limit the input to a number (0–9), you can use a controlled component to weed out all non-numeric values. The event handler sets state only after filtering the input.



Account Number: 155513
You entered: 155513

```
Typed:  1                                    content.js:8
Typed:  15                                   content.js:8
Typed:  155                                  content.js:8
Typed:  1555                                 content.js:8
Typed:  15551                                content.js:8
Typed:  155513                               content.js:8
Typed:  155513a                              content.js:8
Typed:  155513z                              content.js:8
Typed:  155513a                              content.js:8
Typed:  155513t                              content.js:8
```

Only digits are allowed because React controls the element's value.

Figure 7.8 You can type anything you want, as shown in the console. But only digits are allowed as the value and in the view, because this element is controlled.

Listing 7.6 Implementing a controlled component

```
class Content extends React.Component {
  constructor(props) {
    super(props)
    this.handleChange = this.handleChange.bind(this)
    this.state = {accountNumber: ''}
  }
```

Sets the initial value of the account number to an empty string

```
  handleChange(event) {
    console.log('Typed: ', event.target.value)
    this.setState({accountNumber: event.target.value.replace(/[^0-9]/ig,
    ➥ '')})
  }
  render() {
    return <div>
      Account Number:
      <input
        type="text"
        onChange={this.handleChange}
        placeholder="123456"
        value={this.state.accountNumber}/>
      <br/>
      <span>{this.state.accountNumber.length > 0 ? 'You entered: ' +
      ➥ this.state.accountNumber: ''}</span>
    </div>
  }
})
```

Outputs the unfiltered value as it was typed

Filters the value and updates the state

Captures changes

Controls the element by assigning value to state

Prints the account number if it's not empty. "length" is a string property that returns the number of characters. If the value is empty, you print nothing.

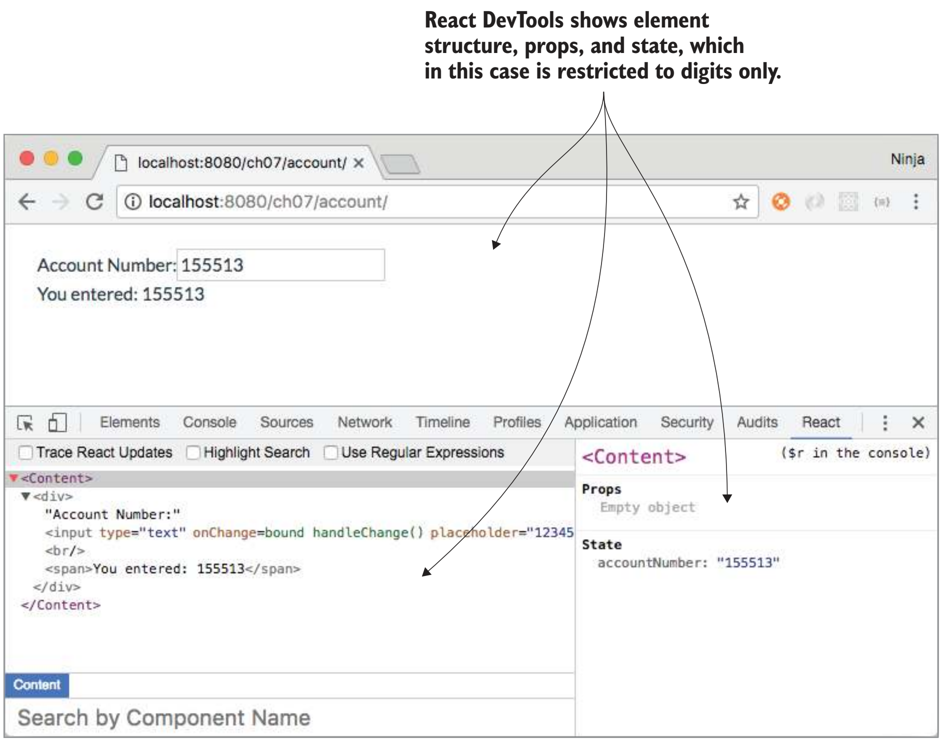
You use a regular expression (http://mng.bz/r7sq), `/[^0-9]/ig`, and the string function `replace` (http://mng.bz/2Qon) to remove all non-digits. `replace(/[^0-9]/ig, '')` is an uncomplicated regular expression function that replaces anything but numbers with an empty space. `ig` stands for case insensitive and global (in other words, find all matches).

`render()` has the input field, which is a controlled component because `value={this.state.accountNumber}`. When you try this example, you'll be able to type in only numbers because React sets the new state to the filtered number-only value (see figure 7.9).

By following React's best practice for working with input elements and forms, you can implement validation and enforce that the representation is what the app wants it to be.

> **NOTE** Obviously, in the account component, you're implementing a front-end validation, which won't prevent a hacker from inputting malicious data into your XHR request sent to the server. Therefore, make sure you have proper validation on the back-end/server and/or business layer, such as ORM/ODM (https://en.wikipedia.org/wiki/Object-relational_mapping).

So far, you've learned about the best practice for working with forms: creating controlled components. Let's cover some alternatives.


React DevTools shows element structure, props, and state, which in this case is restricted to digits only.

Account Number: 155513
You entered: 155513

```
▼<Content>
  ▼<div>
     "Account Number:"
     <input type="text" onChange=bound handleChange() placeholder="12345
     <br/>
     <span>You entered: 155513</span>
   </div>
 </Content>
```

```
<Content>                    ($r in the console)
Props
  Empty object
State
  accountNumber: "155513"
```

Figure 7.9 The controlled element filters input by setting `state` to digits only.

7.2 Alternative ways to work with forms

Using controlled form elements is best practice. But as you've seen, this approach requires additional work, because you need to manually capture changes and update states. In essence, if you define the value of the attributes `value`, `checked`, and `selected` using strings, properties, or states, then an element is controlled (by React).

At the same time, form elements can be uncontrolled when the `value` attributes aren't set (neither to a state nor to a static value). Even though this is discouraged for the reasons listed at the beginning of this chapter (the view's DOM state may be different than React's internal state), uncontrolled elements can be useful when you're building a simple form that will be submitted to the server. In other words, consider using the uncontrolled pattern when you're not building a complex UI element with a lot of mutations and user actions; it's a hack that you should avoid most of the time.

Typically, to use uncontrolled components, you define a form-submit event, which is typically `onClick` on a button and/or `onSubmit` on a form. Once you have this event handler, you have two options:

- Capture changes as you do with controlled elements, and use state for submission but not for values (it's an uncontrolled approach, after all!).
- Don't capture changes.

The first approach is straightforward. It's about having the same event listeners and updating the states. That's too much coding if you're using the state only at the final stage (for form submission).

> **WARNING** React is still relatively new, and the best practices are still being formed through real-life experiences of not just writing but also maintaining apps. Recommendations may change based on a few years of maintaining a large React app. The topic of uncontrolled components is a grey area for which there's no clear consensus. You may hear that this is an antipattern and should be avoided completely. I don't take sides but present you with enough information to make your own judgment. I do so because I believe you should have all the available knowledge and are smart enough to act on it. The bottom line is this: consider the rest of the chapter optional reading—a tool you may or may not use.

7.2.1 Uncontrolled elements with change capturing

As you've seen, in React, an *uncontrolled component* means the `value` property isn't set by the React library. When this happens, the component's internal value (or state) may differ from the value in the component's representation (or view). Basically, there's a dissonance between internal state and representation. The component state can have some logic (such as validation); and with an uncontrolled component pattern, your view will accept any user input in a form element, thus creating the disparity between view and state.

For example, this text-input field is uncontrolled because React doesn't set the value:

```
render() {
  return <input type="text" />
}
```

Any user input will be immediately rendered in the view. Is this good or bad? Bear with me; I'll walk you through this scenario.

To capture changes in an uncontrolled component, you use `onChange`. For example, the input field in figure 7.10 has an `onChange` event handler (`this.handleChange`), a reference (`textbook`), and a placeholder, which yields a grey text box when the field is empty.

Here's the `handleChange()` method that prints the values in the console and updates the state using `event.target.value` (ch07/uncontrolled/jsx/content.jsx).

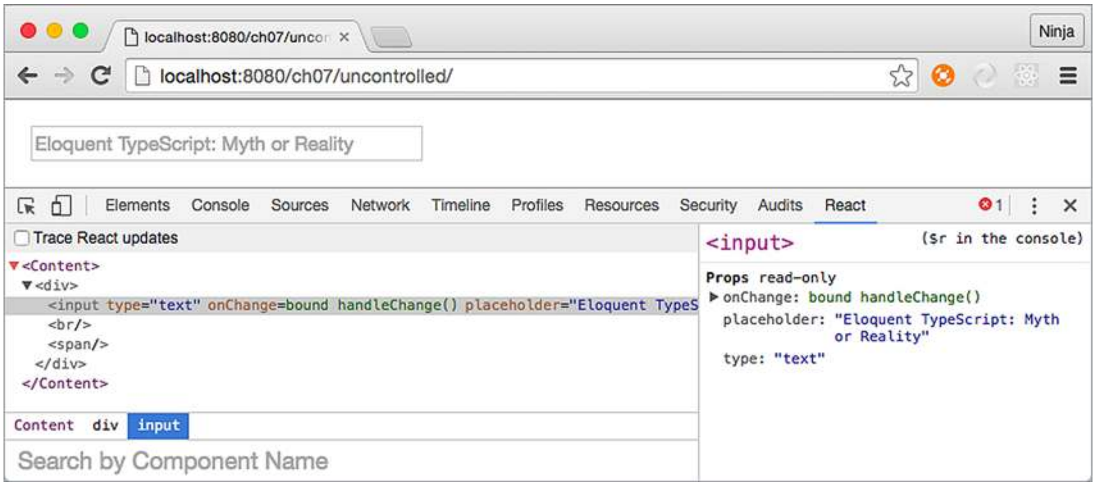
Listing 7.7 Uncontrolled element that captures changes

```
class Content extends React.Component {
  constructor(props){
    super(props)
    this.state = {textbook: ''}          <-- Sets the initial value
                                             to an empty string
    }
    handleChange(event) {
      console.log(event.target.value)
      this.setState({textbook: event.target.value})
    }
    render() {
      return <div>
        <input
          type="text"
          onChange={this.handleChange}
          placeholder="Eloquent TypeScript: Myth or Reality" />
        <br/>
        <span>{this.state.textbook}</span>
      </div>
    }
  }
```

Updates the state on each change in the input field

Doesn't set the value for input, only the event listener

Uses <span> to output the state variable, which you'll set in the handleChange() method



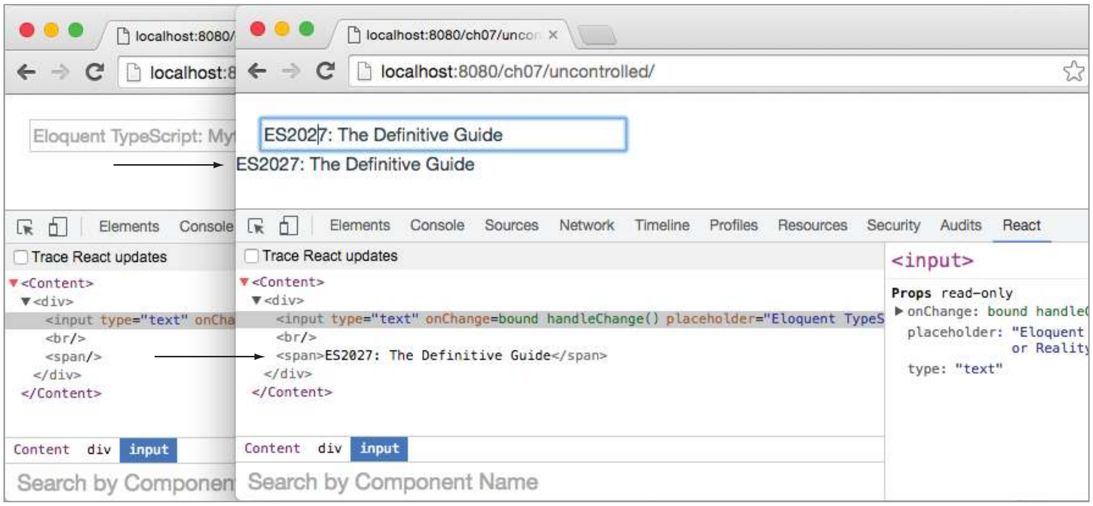
```
▼<Content>
  ▼<div>
     <input type="text" onChange=bound handleChange() placeholder="Eloquent TypeS
     <br/>
     <span/>
   </div>
 </Content>
```

```
<input>                        ($r in the console)
Props read-only
▶onChange: bound handleChange()
  placeholder: "Eloquent TypeScript: Myth
               or Reality"
  type: "text"
```

Figure 7.10 This uncontrolled component has no value set by the application.

The idea is that users can enter whatever they want because React has no control over the value of the input field. All React is doing is capturing new values (`onChange`) and setting the state. The change in state will, in turn, update `<span>` (see figure 7.11).

In this approach, you implement an event handler for the input field. Can you skip capturing events completely?



```
localhost:8080/
Eloquent TypeScript: My

Trace React updates
<Content>
  <div>
    <input type="text" onCha
    <br/>
    <span/>
  </div>
</Content>
Content  div  input
Search by Componen
```

```
localhost:8080/ch07/uncontrolled/
ES2027: The Definitive Guide
ES2027: The Definitive Guide

Trace React updates
<Content>
  <div>
    <input type="text" onChange=bound handleChange() placeholder="Eloquent TypeS
    <br/>
    <span>ES2027: The Definitive Guide</span>
  </div>
</Content>
Content  div  input
Search by Component Name

<input>
Props read-only
▶onChange: bound handle(
 placeholder: "Eloquent
              or Reality
 type: "text"
```

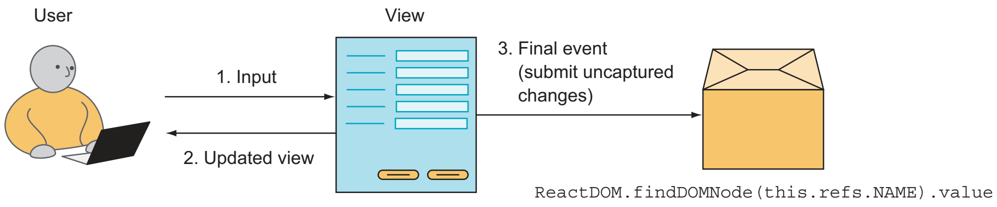
Figure 7.11 Typing updates the state due to capturing changes, but the value of the DOM text-input element isn't controlled.

7.2.2 Uncontrolled elements without capturing changes

Let's look at a second approach. There's a problem with having all the values ready when you want to use them (on form submit, for example). In the approach with change capturing, you have all the data in states. When you opt to not capture changes with uncontrolled elements, the data is still in the DOM. To get the data into a JavaScript object, the solution is to use references, as shown in figure 7.12. Contrast how uncontrolled elements work in figure 7.12 with the controlled elements flow in figure 7.1, which shows how controlled elements function.

NOTE When you're working with controlled components or with uncontrolled components that capture data, the data is in the state all the time. This isn't the case with the approach discussed in this subsection.

To sum up, in order for the approach of using uncontrolled elements without capturing changes to work, you need a way to access other elements to get data from them.



```
ReactDOM.findDOMNode(this.refs.NAME).value
```

Figure 7.12 Using an uncontrolled element without capturing changes and instead accessing values via references

7.2.3 *Using references to access values*

You use references to access values when working with uncontrolled components that don't capture events, such as `onChange`, but the references aren't exclusive to this particular pattern. You can use references in any other scenario you see fit, although using references is frowned on as an antipattern. The reason is that when React elements are defined properly, with each element using internal state in sync with the view's state (DOM), the need for references is almost nonexistent. But you need to understand references, so I'll cover them here.

With references, you can get the DOM element (or a node) of a React.js component. This comes in handy when you need to get form element values, but you don't capture changes in the elements.

To use a reference, you need to do two things:

- Make sure the element in the render's return has the `ref` attribute with a camelCase name (for example, `email: <input ref="userEmail" />`).
- Access the DOM instance with the named reference in some other method. For example, in the event handler, `this.refs.NAME` becomes `this.refs.userEmail`.

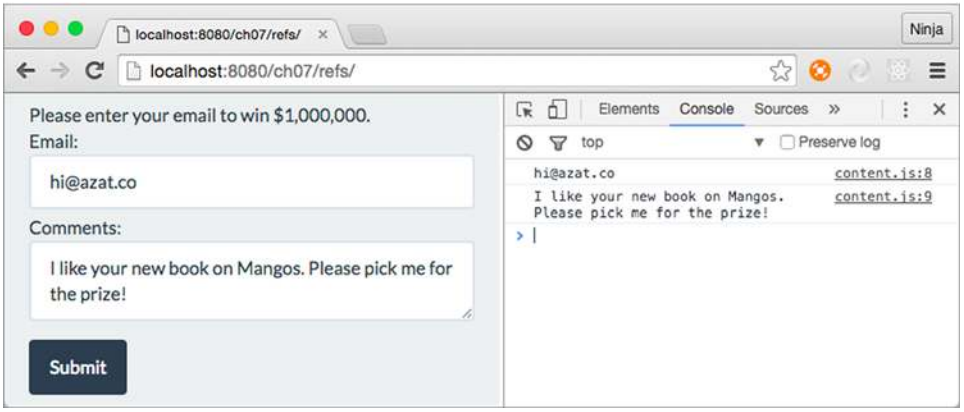
`this.refs.NAME` will give you an instance of a React component, but how do you get the value? It's more useful to have the DOM node! You can access the component's DOM node by calling `ReactDOM.findDOMNode(this.refs.NAME)`:

```
let emailNode = ReactDOM.findDOMNode(this.refs.email)
let email = emailNode.value
```

I find this method a bit clunky to write (too lengthy), so with this in mind you can use an alias:

```
let fD = ReactDOM.findDOMNode
let email = fD(this.refs.email).value
```

Consider the example shown in figure 7.13, which captures user email addresses and comments. The values are output to the browser console.



Please enter your email to win $1,000,000.
Email:
hi@azat.co
Comments:
I like your new book on Mangos. Please pick me for the prize!

```
hi@azat.co                                      content.js:8
I like your new book on Mangos.                 content.js:9
Please pick me for the prize!
```

Figure 7.13 Uncontrolled form that gets data from two fields and prints it in logs

The project structure is very different from other project structures. It looks like this:

```
/email
  /css
    bootstrap.css
  /js
    content.js
    react.js
    react-dom.js
    script.js
  /jsx
    content.jsx
    script.jsx
  index.html
```

Compiled script with the main component

ReactDOM.render() statement in JSX

When the Submit button is clicked, you can access the `emailAddress` and `comments` references and output the values to two logs, as shown next (ch07/email/jsx/content.jsx).

Listing 7.8 Beginning of the email form

```
class Content extends React.Component {
  constructor(props) {
    super(props)
    this.submit = this.submit.bind(this)
    this.prompt = 'Please enter your email to win $1,000,000.'
  }
  submit(event) {
    let emailAddress = this.refs.emailAddress
    let comments = this.refs.comments
    console.log(ReactDOM.findDOMNode(emailAddress).value)
    console.log(ReactDOM.findDOMNode(comments).value)
  }
```

Defines a class attribute

Accesses and prints the value for the email address using a reference

Next, you have the mandatory `render()` function, which uses the Twitter Bootstrap classes to style the intake form (ch07/email/jsx/content.jsx). Remember to use `className` for the `class` attribute!

Listing 7.9 `render()` method of the email form

```
render: function() {
    return (
      <div className="well">
        <p>{this.prompt}</p>
        <div className="form-group">
          Email: <input ref="emailAddress" className="form-control"
          ➥ type="text" placeholder="hi@azat.co"/>
        </div>
        <div className="form-group">
          Comments: <textarea ref="comments" className="form-control"
          ➥ placeholder="I like your website!"/>
        </div>
        <div className="form-group">
```

Prints the value of the prompt attribute of the Content component

Implements the input field for the email, which has a placeholder element attribute. A placeholder property is a visual aid to show an example of what to enter. Uses the className and ref element attributes.

```
          <a className="btn btn-primary" value="Submit"
          ➥ onClick={this.submit}>Submit</a>   <-
        </div>
      </div>
    )
  }
})
```

Code the Submit button with the onClick event that calls this.submit.

A regular HTML DOM node for `<textarea>` uses `innerHTML` as its value. As mentioned earlier, in React you can use `value` for this element:

```
ReactDOM.findDOMNode(comments).value
```

This is because React implements the `value` property. It's just one of the nice features you get with a more consistent API for form elements. At the same time, because the `ReactDOM.findDOMNode()` method returns a DOM node, you have access to other regular HTML attributes (like `innerHTML`) and methods (like `getAttribute()`).

Now you know how to access elements and their values from pretty much any component method, not just from an event handler for that particular element. Again, references are only for the rare cases when you use uncontrolled elements. The overuse of references is frowned on as a bad practice. Most of the time, you won't need to use references with controlled elements, because you can use component states instead.

It's also possible to assign a function to the `ref` attribute in JSX. This function is called just once, on the mounting of the element. In the function, you can save the DOM node in an instance attribute `this.emailInput`:

```
<input ref={(input) => { this.emailInput = input }}
  className="form-control"
  type="text"
  placeholder="hi@azat.co"/>
```

Uncontrolled components require less coding (state updates and capturing changes are optional), but they raise another issue: you can't set values to states or hardcoded values because then you'll have controlled elements (for example, you can't use `value={this.state.email}`). How do you set the initial value? Let's say the loan application has been partly filled out and saved, and the user resumes the session. You need to show the information that has already been filled in, but you can't use the `value` attribute. Let's look at how you set default values.

7.2.4 Default values

Suppose you want the example loan application to prepopulate certain fields with existing data. In normal HTML, you define a form field with `value`, and users can modify the element on a page. But React uses `value`, `checked`, and `selected` to maintain consistency between the view and the internal state of elements. In React, if you hardcode the value like

```
<input type="text" name="new-book-title" value="Node: The Best Parts"/>
```

it'll be a read-only input field. That isn't what you need in most cases. Therefore, in React, the special attribute `defaultValue` sets the value and lets users modify form elements.

For example, assume the form was saved earlier, and you want to fill in the `<input>` field for the user. In this case, you need to use the `defaultValue` property for the form elements. You can set the initial value of the input field like this:

```
<input type="text" name="new-book-
    title" defaultValue="Node: The Best Parts"/>
```

If you use the `value` attribute (`value="JSX"`) instead of `defaultValue`, this element becomes read-only. Not only will it be controlled, but the value won't change when the user types in the `<input>` element, as shown in figure 7.14. This is because the value is hardcoded, and React will maintain that value. Probably not what you want. Obviously, in real-life applications, you get values programmatically, which in React means using properties (`this.props.name`)

JSX

Figure 7.14 The value of an `<input>` element appears frozen (unchangeable) on a web page when you set the value to a string.

```
<input type="text" name="new-book-title" defaultValue={this.props.title}/>
```

or states:

```
<input type="text" name="new-book-title" defaultValue={this.state.title}/>
```

The `defaultValue` React feature is most often used with uncontrolled components; but, as with references, default values can be used with controlled components or in any other scenario. You don't need default values as much in controlled components because you can define those values in the state in the constructor; for example, `this.state = { defaultName: 'Abe Lincoln'}`.

As you've seen, most UI work is done in handy form elements. You need to make them beautiful, yet easy to understand and use. And you must also have user-friendly error messages, front-end validation, and other nontrivial things like tooltips, scalable radio buttons, default values, and placeholders. Building a UI can be complicated and can quickly spiral out of control! Fortunately, React makes your job easier by letting you use a cross-browser API for form elements.

7.3 Quiz

1. An uncontrolled component sets a value, and a controlled component doesn't. True or false?
2. The correct syntax for default values is which of the following? `default-value`, `defaultValue`, or `defVal`
3. The React team recommends using `onChange` over `onInput`. True or false?

4 You set a value for the text area with which of the following? Children, inner HTML, or `value`
5 In a form, `selected` applies to which of the following? `<input>`, `<textarea>`, or `<option>`
6 Which of the following is the best way to extract the DOM node by reference? `React.findDomNode(this.refs.email)`, `this.refs.email`, `this.refs.email.getDOMNode`, `ReactDOM.findDOMNode(this.refs.email)`, or `this.refs.email.getDomNode`

7.4 Summary

- The preferred approach for forms is to use controlled components with event listeners capturing and storing data in the state.
- Using uncontrolled components with or without capturing changes is a hack and should be avoided.
- References and default values can be used with any elements but usually aren't needed when components are controlled.
- React's `<textarea>` uses a `value` attribute, not inner content.
- `this.refs.NAME` is a way to access class references.
- `defaultValue` allows you to set the initial view (DOM) value for an element.
- `ref="NAME"` is how you define references.

7.5 Quiz answers

1 False. The definition of a controlled component/element is that it sets the value.
2 `defaultValue`. The other options are invalid names.
3 True. In regular HTML, `onChange` might not fire on every change, but in React it always does.
4 In React, you set a value with `value` for consistency. But in vanilla HTML, you use inner HTML.
5 `<option>`.
6 Use `ReactDOM.findDOMNode(reference)` or a callback (not listed as an answer).