Stateful UI Components

James Wilcox and Kevin Zatloukal

May 2023

UI Modularity

The HTML presented in the browser is one large tree. When we write code to generate HTML, however, we would like to split up the work into pieces that can be written separately. React supports this by allowing users to define custom tags, whose HTML content is generated and substituted at run-time by calling a function.

As a simple example, suppose we wanted to have a custom tag that generates UI to say "Hello" to the user. That tag could be used as follows:

```
<div>
  ...
  <SayHi name={"Bob"}/>
  ...
</div>
```

The custom tag `SayHi` can be legally used in React provided that there is a function called `SayHi` in scope, which takes the record type `{name: string}` as an argument and returns `JSX.Element` (HTML).

That function could be defined as follows:

```
type SayHiProps = {name?: string};

const SayHi = (props: SayHiProps): JSX.Element => {
  return <p>Hello, {props.name || "friend"}</p>;
};
```

The record containing the attributes passed to the function is traditionally called "`props`", short for "properties". The usage `<SayHi name={"Bob"}/>` would return the HTML `<p>Hello, Bob</p>` since `props.name` is `"Bob"` (that string evaluates as true, so the `"friend"` part is ignored).

In this case, the `name` field is declared as optional, so it would also be legal to use the custom tag simply as `<SayHi/>`. In that case, `props.name` would be undefined, which means `props.name || "friend"` would evaluate to `"friend"` since `undefined` is false (`A || B` evalutes to `B` when `A` is a false value).

UI with State

The previous example is lacking in one important respect: the HTML is fixed and unchanging — there is no way for the user to interact with it! Any real-world application needs user interaction, so functions like the one above will not be sufficient for creating their user interfaces.

To create a UI that can change its appearance, based on user interaction, we will replace the function above with a stateful component. Components differ from simple functions, first and foremost, because they use two kinds of data to generate the HTML: the "props", which store data passed in by the client (as in a simple function), and the "state", which stores *changeable* data managed internally by the component.

We will create these by extending React's `Component` class. This class is generic, taking two type arguments, one describing the type of props and the other describing the type of state. The only required method of a component is `render`, which produces the HTML. Whereas our simple function above takes

only props as an argument, the render method generates HTML based on both the props and the state, which are fields of the class.

We can replace the custom tag above with a stateful component that takes the same props as before but now also stores internal state that allows the user to change what name is shown.

```
type SayHiState = {curName: string, editing: boolean};
```

Here, the state records the name that is currently being shown in the HTML, `curName`. This will start out as `props.name` but can change if the user so wishes. The `editing` field is set to true when the UI shows a text box allowing the user to change the name and false when we are displaying the regular `SayHi` HTML. We implement that as follows

```
class SayHi extends Component<SayHiProps, SayHiState> {
  constructor(props: SayHiProps) {
    super(props);
    this.state = {curName: props.name || "", editing: false};
  }

  render = (): JSX.Element => {
    if (this.state.editing) {
      return (<p>Hello,
          <input type="text" value={this.state.curName}></input>
          <button type="button">Change</button>
        </p>);
    } else {
      return <p>Hello, {this.state.curName || "friend"}</p>;
    }
  };
}
```

The constructor is required to initialize the state by setting `this.state = ...`. In this case, we declared `this.state.curName` to have type `string`, so it cannot be undefined like `this.props.name`. We handle the undefined value in props by assign in `curName` to the value of the expression `props.name || ""`, which will evaluate to `""` when `props.name` is undefined.

In the `render` method, we look at the value of `this.state.editing` to determine whether to display HTML that lets the user type in a new name or the regular greeting. In the former case, we show a text box where the user can type their preferred name. In the latter case, we present the original HTML, but now using `this.state.curName` rather than `props.name`. Note that the string `""` is considered false, so in that case, the expression evaluates to the string `friend` and the HTML displays "Hello, friend" as before.

Note that there are **no declared fields** in this class. A component is not a general class that can store any sort of information and provide any sort of operations. Its sole purpose is to display HTML based on the props and state, so it has no need for any other fields. In fact, components that declare fields, especially mutable fields, are almost always wrong.

Handling Events

There is a key problem with the class written above: there is no way to change the state! The `editing` field will remain false forever, and we will never display the `<input>` that allows them to change the name.

The missing parts in the code above are *event handlers*, which are methods that are called when the user interacts with the HTML. We can allow the user to change the name that is displayed by changing the plain text into a hyperlink and register an event handler that will be called when the user clicks on it as follows:

```
const name = this.state.curName || "friend";
return <p>Hello, <a href="#" onClick={this.doNameClick}>{name}</a></p>;
```

The `doNameClick` method can then change the state of the component like this:

```
doNameClick = (evt: MouseEvent<HTMLAnchorElement>): void => {
  evt.preventDefault();
  this.setState({editing: true});
};
```

We change the value of `this.state.editing` to `true` by calling `this.setState`, passing it a record containing the names of the fields we want to change and their new values. Note that we do not need to specify the value of all fields: any field not mentioned is left unchanged. (We also call `evt.preventDefault` here in order to stop the browser from doing its normal action when a link is clicked, namely, navigating to the URL. We want to stay on this page, so we prevent that behavior.)

The call to `setState` does not directly update the state, but rather, it adds an event to the event queue that will, at the same time, update `this.state`, call `render` again, and update the HTML on the screen to match what `render` returned. Each component has an **implicit invariant** stating that the HTML on the screen matches what `render` would return if called right now, with the current value of props and state. We cannot directly mutate the state, as it would break that invariant. Instead, we notify React that we want a state change to be performed. It places an event in the event queue that will be handled by updating the state and the HTML on the screen at the same time. Since that is another event, however, it will not happen until **after** the current event finishes processing.

When our call to `this.setState({editing: true})` above returns, it will still be the case that our field, `this.state.editing`, is false. However, this will cause React to add an event to the queue that, when processed, will update our state and make another call to `render`. After the new event is processed, since `this.state.editing` will then be true, we will return HTML with an `<input>` box and a change `<button>`.

We can add an event handler on the change `<button>` that, when clicked, will change the value of our field, `this.state.editing`, back to false by calling `this.setState({editing: false})`. That will add an event to the queue that, when processed, will update the state and call `render` a third time, now returning the original HTML with a hyperlink rather than an `<input>` box.

Mirroring HTML State

When we try to write the button click event handler, we run into another problem. In addition to updating `this.state.editing`, we also want to update `this.state.curName` to the name that the user typed in. How do we get that text? One idea would be to try to reach into the HTML document to find the input box that the user typed into to grab its current value; however, that is not ever how we want to do this.

As discussed above, an invariant of any component is that calling `render` should produce exactly the HTML that is on the screen, and the HTML that we have written above would not do that. We created the `<input>` box with an attribute of `value={this.state.curName}`, which means that the initial text in the box will be `this.state.curName`. However, the user is free to change that text by typing in the box. The moment they do so, however, the invariant is broken. If `render` were called again, it would generate HTML containing `<input value={this.state.curName}..>`, describing an input box containing the original text, `this.state.curName`, from before the user started typing.

This is a real bug that can often occurs in React code. If we should change some other part of the state, React would call `render` again. We would return HTML telling it to show this input box with the old value of the text, and it would, of course, do that! From the user's perspective, the result would be that the text they typed would disappear, switching back to the text before they changed it.

To avoid this sort of bug, we must ensure that, at any moment, a call to `render` produces *exactly* the HTML that is on the screen. In particular, this means that any state in the HTML elements on the screen must be **mirrored** in fields of `this.state`.

In this case, the text that the user has typed can be mirrored in `this.state.curName`. To do that, we return HTML that includes a handler for a change event on the input box like this:

```
return (<p>Hello,
    <input type="text" placeholder="your name"
           value={this.state.curName} onChange={this.doNameChange}></input>
    <button type="button" onClick={this.doChangeClick}>Change</button>
  </p>);
```

The browser will then call our `doNameChange` method each time the text in the box is changed. We can implement that method as follows:

```
doNameChange = (evt: ChangeEvent<HTMLInputElement>): void => {
  this.setState({curName: evt.target.value});
};
```

This updates `this.state.curName` to stay in sync with the text that is in the input box: `evt.target` is always the HTML element that fired the event, so in this case, it is the input box, and `value` stores the text in that box. (Note that the event handler will be called after each character is typed into the text box. If you are worried that this will be slow, remember that the 50+ milliseconds between keystrokes is an eternity in computer time.)

Event Handlers

Our example component has no fields. It has only a constructor, a `render` method, and three event handler methods. This is as it should be. Components should normally contain just a constructor that initializes `this.state`, a `render` method that returns HTML for the current value of props and state, and some number of event handlers.

In this class, we will use a convention of naming our event handlers `doXY`, where `X` is a name for the HTML element that fires the event and `Y` is the event in question such as a click or change. Above, our event handlers were `doNameChange` (input box text change), `doChangeClick` (button click), and `doNameClick` (hyperlink click). While the names of these event handlers are arbitrary, sticking to a convention makes it easier for others to read your code. As a more practical benefit, it means that most event handlers will not require overview comments because the name alone communicates what it is for.

There are many other kinds of useful HTML elements, including for example `<select>`, which allows the user to choose an element from a drop-down. The Mozilla Developer Network (MDN) documentation is the best source of information on these elements and their events. Timers are another event that is sometimes useful. Those are created by calling `setTimeout(this.doMyTimeout, timeInMs)`.

One final type of event that we will need in the course are those dealing with Network requests. See the notes on "Client-Server Communication" for full details on how to create network requests using `fetch`. After you have read them, note that each network request requires three different event handlers: (1) handling a successful response, (2) handling the JSON content of the response, and (3) handling a failure of those.

Lifecycle Methods

React includes another set of handler methods that are not for events on individual HTML elements but rather for the component itself. React will invoke these methods to tell the component that it has been mounted (made visible on the screen), updated (had a change to its props or state), and unmounted. Of these, only the first is likely to be useful in this course.

The `componentDidMount` method is called when the component becomes visible for the first time. In general, this method is not needed and often, when people want to use this method, it is a result of poor design. However, there is one important case where it will be necessary for us, namely, when we need to make a `fetch` in order retrieve the data that will be shown in the HTML.

For various reasons, it is not a good idea to perform a `fetch` inside the constructor. (There are React-specific methods for this, but simply in terms of class design, constructors should not do anything other than initialize the state.) Instead, the proper place to launch these initial requests for data is in `componentDidMount`.

Note that, since `fetch` will take some time to return, `render` will be called before that data is available. We will need to make sure that `render` can detect from the state that it does not yet have the data, and it will need to return some HTML in that case, although a blank page is probably fine given how quickly the data should be returned.

Component Modularity

At the beginning, we saw how functions provide modularity in code that generates UI. Stateful components provide the same facility, but the fact that their HTML can change enables new forms of modularity not seen with functions. While functions allow us to write parts of the UI that are on the screen at the same time in different functions, stateful components allow us to write parts of the UI that are on the screen at different times in different functions / components.

A common user interface pattern where this arises is when the UI displays different pages that user at the same place in the screen, with each page having some elements that allow you to switch to different pages. We produce that effect by having a component at that part of the screen whose render method changes which page is shown based on its state:

```
render = (): JSX.Element => {
  if (/* showing page A */) {
    return <A .../>;
  } else if (/* showing page B */) {
    return <B .../>;
  } else {
    return <C .../>;
  }
};
```

To make this work, the component needs to keep track not only of which page to display but also the arguments to pass to that component. One way to do so is to declare a union type that stores this information:

```
type Page = "A" | {kind: "B", x: number} | {kind: "C", y: string};

type PagerState = {page: Page};
```

Here, our component shows either the page A, with no arguments, or the page B, with a number argument `x`, or the page C, with a string argument `y`. The full render method then looks like this:

```
render = (): JSX.Element => {
  if (this.state.page === "A") {
    return <A/>;
  } else if (this.state.page.kind === "B") {
    return <B x={this.state.page.x}/>;
  } else {
    return <C y={this.state.page.y}/>;
  }
};
```

Due to the union type and type narrowing, TypeScript can see that each of these lines is valid.

If the general shape of the `Page` type looks familiar, that is because we have seen types like this throughout the course: it is an inductive type! Specifically, it falls in to the "enum-like" family of inductive types. This example shows that, despite not using any recursive type arguments (the way that lists and trees do), enum-like inductive types are quite useful and arise often in practice.

Keys

React allows any HTML element to have a "`key`" attribute, which it uses to distinguish elements in lists of the same type from one another. As a simple example, suppose that we have the following bulleted list:

```
<ul>
  <li>One</li>
```

```
  <li>Two</li>
  <li>Three</li>
</ul>
```

Suppose that this was returned by the first call to `render` and that the second call returns the following:

```
<ul>
  <li>One</li>
  <li>One Point Five</li>
  <li>Two</li>
  <li>Three</li>
</ul>
```

In order to update the HTML on the screen, React needs to understand what has changed. In this case, there are two competing explanations for what happened here. The first is that the second item in the list changed from `Two` to `One Point Five`, the third item in the list from `Three` to `Two`, and a new item `Three` was added at the end. The second explanation is that a new item, `One Point Five`, was inserted between the first and second items.

Obviously, of the second explanation is more parsimonious. However, the calculation necessary for React to find this explanation is potentially expensive and, even more importantly, the first explanation might actually be the correct one! Rather than trying to figure this out itself, React asks us to identify the elements that are the same by including a `key={..}` attribute on each item.

In our example, if we returned this HTML initially

```
<ul>
  <li key="1">One</li>
  <li key="2">Two</li>
  <li key="3">Three</li>
</ul>
```

and then this HTML the second time

```
<ul>
  <li key="1">One</li>
  <li key="1.5">One Point Five</li>
  <li key="2">Two</li>
  <li key="3">Three</li>
</ul>
```

Then, React would see that a single new item was changed. On the other hand, if we returned this HTML

```
<ul>
  <li key="1">One</li>
  <li key="2">One Point Five</li>
  <li key="3">Two</li>
  <li key="4">Three</li>
</ul>
```

Then, React would see that items `2` and `3` were changed and item `4` was added.

React will give warnings whenever lists of items without `key` attributes are placed in the HTML. For the reasons just described, it is important to fix those errors by adding keys. However, there are other cases where keys are useful.

In our example from the previous section, we supported pages `A`, `B`, and `C`, with the latter two accepting arguments. It is possible that page could change from page `B` with argument `x={1}` to page `B` with argument `x={2}`. In that case, rather than creating a new instance of the class `B`, React would reuse the existing component, **changing its props** and calling `render` again.

Changing props often causes bugs in components. One case this would likely happen, for example, is when we are performing a `fetch` in `componentDidMount` to retrieve data to display. Almost certainly, that

fetch would include arguments from props, so when props changes, we would very likely need to perform another fetch to get new data. It is possible to detect a change to props in `componentDidUpdate` and handle it by making another fetch. However, that approach is difficult and error-prone.

A simpler solution is to prevent React from reusing the existing component. We can do so by making it interpret our intention not as a change to the properties of the `B` element returned but rather as a change that removed the old `B` and added a new one `B`. As we just saw, we communicate that interpretation by including a `key` attribute on the element.

In this example, we could change the HTML returned by our paging component to return include a `key` for `B` pages as follows:

```
const x = this.state.page.x;
return <B x={x} key={`B-${x}`}/>;
```

Note that `key` must be a string, so we incorporate `x` into a string here using a template literal. With this choice of key, `B` pages with different `x` arguments will have different keys, so React will never reuse one when displaying new HTML.

Debugging a Component

As with any set of functions that use shared state, there is a much increased likelihood that debugging will be required. With network requests, the built-in Network tab provides most of the information needed to debug, but that is rarely the case for UI components. Instead, we will often need to add information to the console ourselves that will help us debug the problems that occur.

Our typical pattern for doing so starts with declaring a constant, outside the class, that can be used to turn debugging information on or off. Usually, we will want it off, but when we need to debug, we can flip this to true.

```
const DEBUG: boolean = false;  // off by default
```

Next, when debugging, we will want to write to the console indicating that rendering occurred and showing the current value of the state.

```
render = (): JSX.Element => {
  if (DEBUG) console.log("rendering", this.state);
  ...
};
```

We will also want to write to the console indicating each event that occurs. If the event includes information that we use in the code, it is also useful to print those out. For example, with the change event on our input box above, we could describe the event and the new value of the text as follows:

```
doNameChange = (evt: ChangeEvent<HTMLInputElement>): void => {
  if (DEBUG) console.log("name change", evt.target.value);
  this.setState({curName: evt.target.value});
};
```

If you have multiple UI components that need to be debugged simultaneously, you may also want to include the component name in these messages so that you can distinguish where the messages are from.

The following are some common symptoms of bugs and their typical causes.

1. **The event handler is not called when the event occurs.**

   Make sure that you are passing the method rather than calling the method when you construct the HTML. That is, you want to return HTML containing `onClick={this.doInputClick}` rather than `onClick={this.doInputClick()}`. The latter is calling handleInput click in `render` and passing its return value as the value of `onClick`.

2. **No render occurs after an event.**

   Make sure the code is calling `setState`. If you call it, then React will add an event that it processes by calling `render`.

   Make sure that you are not directly mutating records or arrays inside of `this.state`. You must call `setState` for React to initiate an update. (Also, this mutation breaks the implicit invariant.)

3. **The UI updates only after the second time the event occurs**

   Often this happens when we forget that `setState` does not instantly change `this.state`. If your event handler is using `this.state`, its value will remain the original value throughout your event handler. It's only the second time that event handler is called, after a render, that it will hold the value that was passed to `setState`.

4. **Crashes with error "`state does not exist on undefined`".**

   If the line of code in question has the expression `this.state`, then the value of `this` was undefined! Make sure you declared your event handler with "`doXY = () => { .. }`" syntax. If you instead declared it as "`doXY() { .. }`", then you will see this error.

5. **Not displaying the latest data that was written to the server.**

   When the client writes data to the server, it will typically be designed to read that data back again before displaying it. If the new data is not being displayed, then the first thing to check is that it is actually being requested. You should see a request for that data in the Network tab.

   If you don't see a request for it and that request would normally be made in `componentDidMount`, then the problem is likely that the component is being reused instead of re-created. You can fix this by adding a `key` attribute whose value will be different when the data changes, causing React to create a new element, which will request the new data in its `componentDidMount` method.

Why the Legacy API?

The part of the React API we are using here, by extending `Component`, is now called the "Legacy API". There is a newer API that uses "hooks" inside of functions to achieve state. I do not use the new API personally, and there are a couple of important reasons why it is not a good fit for this class.

First, the new API tries to pretend that stateful components are not stateful by dressing them up like functional components. As we have said repeatedly in this course, it is important to know the complexity level of the code you are writing. Disguising complex code as less complex code does not make it less complex, and in fact, it increases the odds that some will mistakenly treat it as if it were at a lower complexity level, not taking the all the required steps to ensure that it is correct.

Second, the mechanisms used by the new API look like they were created to win a bet about who could create the worst debugging humanly possible. When they do not work as intended, I would expect to lose a weekend to debugging, at the very minimum, whereas, in this class, it is a goal to avoid painful debugging.