**CS628 Full-Stack Development II**

**HOS04: React State**

12/12/2021 Developed by Jerry Zeyu Chen

School of Technology & Computing (STC)

City University of Seattle (CityU)

Icon

Description automatically generated

**Before You Start**

* Version numbers may not match with the most current version at the time of writing. If given the option to choose between stable release (long-term support) or most recent, please select the **stable release** rather than the beta-testing version.
* This tutorial targets Windows users and MacOS users.
* There might be subtle discrepancies along with the steps. Please **use your best judgment** while going through this cookbook-style tutorial to complete each step.
* For your working directory, use your course number. This tutorial may use a different course number as an example.
* All the steps and concepts in this tutorial are from the textbook, so if you encounter problems in this tutorial, please **try to read and compare the textbook to solve the problem**. If you still can’t solve the problem, please feel free to contact your course TA.
* **Avoid copy-pasting code from the book or the GitHub repository.** Instead, type out the code yourself. Resort to copy-pasting only when you are stuck and find that things are not working as expected.

**Learning Outcomes**

* Understanding setTimeout(): <https://developer.mozilla.org/en-US/docs/Web/API/setTimeout>
* Understanding pure function: <https://en.wikipedia.org/wiki/Pure_function>
* React component document: <https://reactjs.org/docs/react-component.html>

**Resources**

* Subramanian. (2019). *Pro MERN Stack Full Stack Web App Development with Mongo, Express, React, and Node* (2nd ed. 2019.). Apress. https://doi.org/10.1007/978-1-4842-4391-6
  + Ch. 4: React State
    - Initial State
    - Async State Initialization
    - Updating State
    - Lifting State Up
    - Event Handling
    - Stateless Components

1. **Initial State**
   1. For this HOS, the goal is to add a button and append a row to the initial list of issues on click of that button. We’ll add this button in place of the placeholder text in the IssueAdd component. By doing that, you’ll learn about a component’s state, how to manipulate it, how to handle events, and how to communicate between components.
   2. Use vscode to open the folder that you clone on GitHub of the HOS assignment that you accepted this week.
   3. Please copy CS628\_HOS\_yourName/ from last week to the Module04/ folder. Press ctrl + ` in vscode to open the terminal. Type following command to commit the code from the last HOS.

$ git add --all

$ git commit -m "last week HOS work"

* 1. Enter the project directory and activate the watcher with the following command:

$ cd Module04/CS628\_HOS\_yourName

$ npm run watch

* 1. Open a new terminal, go to the project directory, and start the server:

$ cd Module04/CS628\_HOS\_yourName

$ npm start

* 1. The state of a component is captured in a variable called this.state in the component’s class, which should be an object consisting of one or more key-value pairs, where each key is a state variable name and the value is the current value of that variable.
  2. Open src/App.jsx, let’s rename the global array of issues to initialIssues, just to make it explicit that it is only an initial set.

Graphical user interface, text, website

Description automatically generated

* 1. Setting the initial state needs to be done in the constructor of the component. This can be done by simply assigning the variable this.state to the set of state variables and their values. Let’s use the variable initialIssues to initialize the value of the state variable issues like this:

Text

Description automatically generated

* 1. For now, let’s just use an array of issues as the one and only state of the component and use that array to construct the table of issues. Thus, in the render() method of IssueTable, let’s change the loop that creates the set of IssueRows to use the state variable called issues rather than the global array like this:

Text

Description automatically generated

* 1. Now, open your browser and type <http://localhost:3000> in the URL bar.

Running and testing this piece of code should show no change in the application; you will still see a table containing two rows of issues, just as before.

Graphical user interface, text, application

Description automatically generated

* 1. Open a new terminal, enter following command to commit the section 1 work to git.

$ git add --all

$ git commit -m "Section 1"

1. **Async State Initialization**
   1. Although we set the initial state in the constructor, it is highly unlikely that regular SPA components will have the initial state available to them statically. These will typically be fetched from the server. In the case of the Issue Tracker application, even the initial list issues to be displayed would have to be fetched via an API call.

The state can only be assigned a value in the constructor. After that, the state can be modified, but only via a call to React.Component’s this.setState() method . This method takes in one argument, which is an object containing all the changed state variables and their values. The only state variable that we have is the one called issues, which can be set to any list of issues in a call to this.setState().

Let us add a method to the IssueTable to simulate asynchronous calling the API to the server.

A screenshot of a computer

Description automatically generated with medium confidence

In this method, we use JavaScript built-in setTimeout function to delay the execution of the this.setState method by 500 milliseconds to simulate the delayed response of the server.

* 1. Now, it is very tempting to call loadData() within the constructor of IssueTable. It may even seem to work, but the fact is that the constructor only constructs the component (i.e., does all the initialization of the object in memory) and does not render the UI. The rendering happens later, when the component needs to be shown on the screen. If this.setState() gets called before the component is ready to be rendered, things will go awry.

React provides many other methods called *lifecycle methods* to cater to this and other situations where something needs to be done depending on the stage, or changes in the status of the component. The best place to initiate the loading of data in this case is the componentDidMount() method. At this point in time, the DOM is guaranteed to be ready, and setState() can be called to re-render the component. Let’s add the componentDidMount() method in IssueTable and load the data within this method:

Text

Description automatically generated

* 1. Now, open your browser and type <http://localhost:3000> in the URL bar.

You will find that the list of issues is displayed as it used to be in the previous steps. But you will also see that for a fraction of a second after the page is loaded, the table is empty.

A screenshot of a computer

Description automatically generated

* 1. Please feel free to explore React's official documentation to learn about other lifecycle methods, and answer why not use componentDitUpdate() method in this section, but componentDidMount()?

Documentation: <https://reactjs.org/docs/react-component.html>

* 1. Open a new terminal, enter following command to commit the section 2 work to git.

$ git add --all

$ git commit -m "Section 2"

1. **Updating State**
   1. In this section, let’s make a minor change to the state rather than set a completely new value to it. Let’s add a new issue and thus change, not the complete state, but only a portion of it.

To start, let’s add a method in IssueTable to add a new issue. This can take in as an argument an issue object, to which we’ll assign a new ID and set the creation date. The new ID can be calculated from the existing length of the array.

Text

Description automatically generated

Note that in line 56, we used the slice method that comes with JavaScript's Array to make a shallow copy of the this.state.issues array. Please do some personal research or read textbooks to answer why we need to do this? Some useful information: <https://reactjs.org/docs/react-component.html#setstate>

* 1. Let’s first declare this hard-coded sample issue object globally, right after the global initialIssues:

A screenshot of a computer

Description automatically generated with medium confidence

* 1. Let’s use this object in a call to createIssue() , after the expiry of a timer of two seconds, in the constructor of IssueTable:

Text

Description automatically generated

* 1. Now, open your browser and type <http://localhost:3000> in the URL bar.

On running this set of changes and refreshing the browser, you’ll see that there are two rows of issues to start with. After two seconds, a third row is added with a newly generated ID and the contents of the sample issue.

A screenshot of a computer

Description automatically generated

* 1. Note that we did not explicitly call a setState() on the IssueRow components. React automatically propagates any changes to child components that depend on the parent component’s state. Further, we did not have to write any code for inserting a row into the DOM. React calculated the changes to the virtual DOM and inserted a new row.

At this point, the hierarchy of the components and the data flow can be visually depicted, as shown in below figure.

Diagram

Description automatically generated

* 1. Open a new terminal, enter following command to commit the section 3 work to git.

$ git add --all

$ git commit -m "Section 3"

1. **Lifting State Up**
   1. Let's refactor the existing code to make it look more reasonable.

Before we add user interface elements to create new issues, let’s move the initiation of the creation to where it really belongs: in the IssueAdd component.

There is no straightforward way to communicate between siblings in React. Only parents can pass information down to children; horizontal communication seems hard, if not impossible.

The way around this is to have the common parent contain the state and all the methods that deal with this state. By lifting the state up on level to IssueList, information can be propagated down to IssueAdd as well as to IssueTable.

Let’s start by moving the state to IssueList and the methods to load the initial state. The constructor of IssueTable had both the state initialization as well as the timer, of which only the state initialization needs to move (the timer will move to IssueAdd). The other methods that deal with the state are componentDidMount(), loadData(), and createIssue(). Let’s move these also to the IssueList class.

Cut the code from the IssueTable component to the appropriate component:

A screenshot of a computer

Description automatically generated with medium confidence

* 1. Now, IssueTable doesn’t have a state to construct the IssueRow components from. But you have already seen how data can be passed in from a parent to a child in the form of props. Let’s use that strategy and pass the array of issues from the state within IssueList to IssueTable via props. And, within IssueTable, instead of referring to the state variable issues, we’ll need to get the same data from props:

Graphical user interface, text

Description automatically generated

Text

Description automatically generated

* 1. As for IssueAdd, we need to move the timer into the constructor of this class and trigger the addition of a new issue from within this component. But we don’t have the createIssue() method available here. Fortunately, since a parent component can pass information down to a child component, we’ll pass the method itself as part of the props to IssueAdd from IssueList, so that it can be called from IssueAdd. Here’s the changed instantiation of the IssueAdd component within IssueList:

Text

Description automatically generated

* 1. This lets us make a call to createIssue() from IssueAdd using this.props.createIssue() as part of the timer callback. So let’s create a constructor in IssueAdd and move the timer set up with a minor change to use the createIssue callback passed in via props like this:

Text

Description automatically generated

* 1. To make this keyword in the createIssue method all point to the IssueList instance, let's bind it in the constructor of IssueList.

Text

Description automatically generated

* 1. Now, open your browser and type <http://localhost:3000> in the URL bar.

The effect of these changes will not be seen in the user interface. The application will behave as it used to. On refreshing the browser, you will see an empty table to start with, which will soon be populated with two issues and after two seconds, another issue will be added.

* 1. Open a new terminal, enter following command to commit the section 4 work to git.

$ git add --all

$ git commit -m "Section 4"

1. **Event Handling**
   1. Let’s now add an issue interactively, on the click of a button rather than use a timer to do this. We’ll create a form with two text inputs and use the values that the user enters in them to add a new issue. An Add button will trigger the addition.

Let’s start by creating the form with two text inputs in the render() method of IssueAdd in place of the placeholder div. At this point, we can remove the timer that creates an issue from the constructor.

Text

Description automatically generated

* 1. To handle events such as onclick and onsubmit, the properties that we need to supply to the elements are, simply, onClick and onSubmit. As in plain HTML and JavaScript, these properties take functions as values. We’ll create a class method called handleSubmit() to receive the submit event from the form when the Add button is clicked. Within this method, we’ll need a handle to the form, so as in regular HTML, let’s give the form a name, say, issueAdd which can then be referred to in JavaScript using document.forms.issueAdd.

So, let’s rewrite the form declaration with a name and an onSubmit handler like this.

Text

Description automatically generated

* 1. Now, we can implement the method handleSubmit() in IssueAdd. This method receives the event that triggered the submit as an argument. In order to prevent the form from being submitted when the Add button is clicked, we need to call the preventDefault() function on the event. Then, using the form handle via documents.forms.issueAdd, we can get the values of the text input fields. Using these, we’ll create a new issue by calling createIssue(). After the call to createIssue(), let’s keep the form ready for the next set of inputs by clearing the text input fields.

A screenshot of a computer

Description automatically generated with medium confidence

* 1. Since handleSubmit will be called from an event, the context, or this will be set to the object generating the event, which is typically the window object. As you saw in the previous section, to let this method have access to the object variables via this, we need to bind it to this in the constructor:

Text

Description automatically generated

* 1. The global object sampleIssue is no longer required, so we can get rid of it.

A screenshot of a computer

Description automatically generated with medium confidence

* 1. Now, open your browser and type <http://localhost:3000> in the URL bar.

You can now test the changes by entering some values in the owner and title fields and clicking Add.

Graphical user interface, application

Description automatically generated

* 1. At the end of all this, we have been able to encapsulate and initiate the creation of a new issue from the IssueAdd component itself. To do this, we “lifted the state up” to the least common ancestor, so that all children have access to it directly via passed-in props or via callbacks that can modify the state. This new UI hierarchy data and function flow is depicted in the below figure. Compare this with the situation where the state was maintained in IssueTable, as in the figure in Section 3 Steps 5.

Diagram

Description automatically generated

* 1. Open a new terminal, enter following command to commit the section 5 work to git.

$ git add --all

$ git commit -m "Section 5"

1. **Stateless Components**
   1. IssueList has lots of methods, a state, initialization of the state, and functions that modify the state. In comparison, IssueAdd has some interactivity, but no state. But, if you notice, IssueRow and IssueTable have nothing but a render() method. For performance reasons and for clarity of code, it is recommended that such components are written as functions rather than classes: a function that takes in props and just renders based on it. It’s as if the component’s view is a pure function of its props, and it is stateless. The render() function itself can be the component.

If a component does not depend on props, it can be written as a simple function whose name is the component name.

Since IssueRow and IssueTable are stateless components, let’s change them to pure functions.

A screenshot of a computer

Description automatically generated with medium confidence

Text

Description automatically generated

* 1. Now, open your browser and type <http://localhost:3000> in the URL bar.

You can see the same output and function as in the previous section.

1. **Pushing your work to GitHub**
   1. Open the PowerShell/Command-line or terminal, and make sure you are in …/cs628-(in or on)-winter-2022-hos04-yourGitHubUserName/
   2. Type “git status”, press entry, and check the output to make sure the following file is in your repository:
      1. HOP04 - Submission.pdf Include all your answers and screenshots
      2. CS628\_HOS\_yourName/ Your HOS work
   3. Type the following command:

$ git add --all

$ git commit -m "Submission for Module 04 - Your Name"

$ git push