Camosun College

ICS 211 - Web Applications

Lab 7 - Adding a Form to Your Application

Due Date: Lab Demo Due by Nov. 9 @ 2:20 PM

Lab Quiz Due by Nov. 9 @ 11:59 PM

Background and Theory

DOM Events

Events allow us to add two-way data flow in our web applications. Mouse movements, button clicks, and typing text are all examples of actions that can fire events in DOM elements. The DOM Level 0 API is the simplest API to use. Event handlers are attributes of the DOM element. In React and JSX, the event is camelCase and the handler is provided as a function name. Events propagate through the DOM Tree. In the first phase, called *capture*, the event travels from the top (Window) down to the target element that caused the event. In the second phase, called *target*, which is also the beginning of the third *bubbling* phase, the event is at the element that caused it. Finally, in the third *bubbling* phase, the event travels back up to the top. Event Handlers in React have to be *bound* to the instance of the object. That's because they aren't called by the object itself. Events in React are actually an instance of `SyntheticEvent`, a wrapper around native browser events. This is for cross-platform compatibility. **For some examples of DOM Events and Event Handlers, see the corresponding lecture slides**.

React Forms

The challenge with React Forms is to keep everything in sync. The UI, form control, and Component state must all be kept in sync. A Component that does this is called a *Controlled Component*. The key thing to do is to use the `onChange` event to capture any change to a form control and record that change in the Component's `this.state`. Four form controls have special properties in React called *interactive properties*. This means they are mutable within the element. The elements with *interactive properties* are `<input>`, `<textarea>`, `<select>`, and `<option>`. The *interactive properties* themselves are `value` (used with `<input>`, `<textarea>`, and `<select>`), `checked` (used with `<input type="checkbox">` and `<input type="radio">`) and `selected` (used with `<option>` in `<select>`). For Form submission, you use the `onClick` event with a button. **For some examples on React Forms, see the corresponding lecture slides**.

Tasks

Task 1 - Setting Up

1. **Copy** your solution for Lab 6, including your JSON file, into a new *Lab 7* folder. Rename your Lab 6 solution (ex. *lab7.html*). You will be adding on to your solution from Lab 6 in this new file.

2. Make a copy of your *lab6.json* file and call it *lab7.json*. **Add a new key-value pair to each object in the sites array**. The key should be something like `checked` and the value initially set to `false`. **Keep this file open in something like Notepad++ and have the Replace feature set up to change the values back to `false` for testing.**

3. Start the JSON Server in your *Lab 7* folder: `json-server --watch lab7.json`

Task 2 - Using the New `checked` key-value Pair

1. In your `FavSites` Component's `render()` method's `return()` statement, pass in the new prop to your `Site` Component that you are now getting from your JSON file (the `checked`).

2. In your `Site` Component, check this prop. If it's `true`, return the site as before. If it's `false`, return a `null`. This means that only those JSON objects with the `checked` key set to `true` will be rendered.

Task 3 - Adding a New Form Component: The Constructor

1. Create a new *Stateful* Component called *FavSitesForm*. In this Component, you want to add some multi-select checkboxes that will allow a user to select one or multiple favourite sites. **Refer to the lecture notes on React Event and Forms for an example.**

2. In the constructor, like the example from the lecture notes, you will need to add a `this.state` key-value pair for the checkboxes. In the lecture notes example, this was a JavaScript object but you may find it easier in this case to use an array. For an array, add the following to `this.state` in the *FavSites* constructor:

   ```
   checkboxGroup: [ false, false, false ]
   ```

   This creates an array and inits each element to `false`. There is a `false` for every site that will be in the `checkboxGroup`.

3. Also in the constructor, add the necessary bind statements that you need for your two handlers (one handler when one of the checkboxes are checked, and another for the submit button).

4. Finally, add a `showForm` boolean to `this.state`. Initially set it to `true`.

Task 4 - Create the Form in FavSitesForm

1. You are going to need to add an `if...else` statement in the `render()` method of *FavSitesForm*:

```
if ( this.state.showForm ) {
  return (
  // Put your form code here
  );
} else {
  return <FavSites />
}
```

If the state variable `showForm` is `true`, show the form. If not, show your previously made `<FavSites />` component. This means *FavSitesForm* is a parent of *FavSites*.

2. Next, make your form. **Use the example from the lecture notes as a guide**. Note that if you are using an array for `checkboxGroup`, you need to use array notation in the `checked` attribute of your `<input>` tags. For example: `this.state.checkboxGroup[0]`. The `value` attribute's value has to match the property name in the `checkboxGroup` state variable. For example, if you used an array for `checkboxGroup`, the first checkbox's value attribute would be `value='0'`. Don't forget to add a submit button after your checkboxes.

3. Either style your Form with some new CSS or move your styles from *FavSites* to *FavSitesForm*. You can then pass these styles into *FavSites* as props. You could then add some extra styles just for the Form.

Task 5 - The Checkboxes Handler

1. This handler is almost exactly like the handler shown in the lecture notes. The only difference being that if you used an array for `checkboxGroup` replace:

   ```
   let checkboxes = Object.assign(this.state.checkboxGroup)
   ```

   with:

   ```
   let checkboxes = this.state.checkboxGroup.slice()
   ```

   `slice()` is for copying arrays

Task 6 - The Submit Button Handler

1. This is the hardest part!! You're going to need to write to the JSON Server. Basically, you need to find out which checkboxes were checked and update those values. You will need to start with a `for` loop that will iterate through the `this.state.checkboxGroup` array. If the value at `i` is `true`, you need to change the corresponding JSON object's checked value also to `true`. Recall that supplying a JSON object's `id` at the end of the `http://localhost:3000/sites` endpoint, causes the corresponding individual JSON object to be fetched. Since `id`'s start at `1` and indexes in our array start at `0`, you only need to add `1` to `i` in the loop to get the `id`.

2. Use ES6 template literals to create the `fetch()` endpoint. Strings that use template literals must use a backtick (`) around it:

   ```
   fetch(`http://localhost:3000/sites/${id}`, {
     // JSON object for options
   }) // closes fetch call
   ```

   The second argument to `fetch()` is a JSON object with some options we can supply. We didn't need this for the simple GET Request that we did last lab but we need to set some options for the PATCH Request we are going to make to update the JSON.

3. Replace the comment from the above step with the following code:

   ```
   method: 'PATCH',
   headers: {
   ```

```
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({
    "checked": true
  })
```

The HTTP Verb we are sending in this Request is PATCH. The RESTful API (our JSON Server) recognizes that this verb means you are going to update an individual value in the corresponding JSON object without changing any other values or JSON objects. The `application/json` `Content-Type` header is required according to the JSON Server Documentation. The `body` is a JSON object with the key/value pair we want modified. We call `stringify()` here to ensure a correctly formatted JSON string.

4. After the closing `})` for the `fetch()` call, add the following:

```
.then(response=> this.handleHTTPErrors(response))
.then(result=> {
  this.setState({
    showForm: false
  });
})
.catch(error=> {
  console.log(error);
});
```

This is very similar to your `fetch()` in *FavSites*. You will need to copy the `handleHTTPErrors(response)` method from *FavSites* and add it to this Component. The `this.setState()` statement will be in the loop when we really only need to set it once. But we need it in the Promise Chain to ensure that `showForm` doesn't get set to `false` before all the `fetch()` updates have happened.

Task 7 - Modify the ReactDOM.render() Call

Fairly easy, change `ReactDOM.render()` to render the *FavSitesForm* Component instead of *FavSites*.

Task 8 - Testing and Adding More Sites

Test to make sure your Application works. Use the Developer Tools in your browser to assist you in Debugging. In the React Tab, ensure that your checkboxes `checked` attribute changes correctly. Use the Network Tab to ensure your `fetch()` API calls are what you expect. Add some more sites to your JSON file and checkboxes in your Form to give users some more options!!

Lab Submission (see top of lab for due date)

1. Demo to the Lab Instructor (15 marks):
    - That you have both `http-server` and `json-server` running
    - That your website STILL works!
    - That you have added a Stateful *FavSitesForm* Component.
    - Show that your JSON data gets modified by your Form when Submitted.
    - Show that your Form has been styled (this will be rated between 0 to 3)
    - Show that you have at least **five** Sites to choose from.

- For full marks, all have to be implemented correctly.

2. Do the Lab 7 Quiz in D2L (5 marks).