PWT React labs

Table of Contents

Exercises

EXERCISE 1 - say hello

Let's create a simple component. Add this file next to **`App.js`**:

Create a file named SayHello.js with this code:

```
import React from 'react';

const name = 'Ken';
const SayHello = () => <p>Hello, {name}!</p>;    ①

export default SayHello;
```

① All dynamically evaluated JSX expressions are surrounded with single curly braces

EXERCISE 2 - names

Let's render a collection of names. We'll need to use some JSX magic but first, let's see what it looks like as a simple stringified array. Yuck!

Change the **`SayHello`** *function to output the names*

```
import React from 'react';

const names = ['Ken', 'Sujan', 'Alameda'];
const SayHello = () => <p>Hello, {names}!</p>;

export default SayHello;
```

EXERCISE 3 - map to pretty print

Change the component to generate a list of names as list items

```
import React from 'react';

const names = ['Ken', 'Sujan', 'Alameda'];
const SayHello = () => {

  const renderedNames = names.map(name => <li>Hello, {name}</li>);

  return <ul>{renderedNames}</ul>;      ①
}

export default SayHello;
```

① Note the rendering of the mapped JSX component array as a single entry.

You might be tempted to create a list of regular objects and render them with curly braces. You'll get an error; only valid JSX expressions can be rendered from a variable, and an *array* of JSX markup elements is legitimate, not an array of plain old Javascript variables.

EXERCISE 4 - add key

Keys are used to reduce the complexity of DOM updates (it can locate something to change quickly).

Add the key to the mapping operation (it is the 2nd parameter)

```
import React from 'react';

const names = ['Ken', 'Sujan', 'Alameda'];
const SayHello = () => {

  const renderedNames = names.map((name, key) => <li key={key}>Hello, {name}</li>);

  return <ul>{renderedNames}</ul>;
}

export default SayHello;
```

Keys *should be* something that is generated with the data of your list, like a database **primary key**. In this case it's a simple generated list, but you can save yourself lots of issues by using something the entry has already, as long as it is unique.

EXERCISE 5 - use props!

Setting props in your component allows you to pass data inward.

Edit the **App.js** *file which launches our* **SayHello** *component, and pass it some names*

```
function App() {
  return (
    <SayHello names={['Ken', 'Sujan', 'Alameda']}/>.     ①
  );
}

export default App;
```

① Props are passed as HTML attributes, and dynamic data is bound by curly braces. In this case we're passing an array of strings *into* the **SayHello** component as the **names** prop.

and

Edit the **Sayhello.js** *component and access the names from the* **props** *which are passed to each component*

```
import React from 'react';

const SayHello = (props) => {
  const renderedNames = props.names.map((name, key) => <li key={key}>Hello, {name}</li>);
  return <ul>{renderedNames}</ul>;
}

export default SayHello;
```

EXERCISE 6 - RENDERING and formatting

Now let's create a new component. We'll build the **DisplayRandomNumber** component in a bit, but let's mount it in the **App.js** file first the way we'd like it to be used.

Add the bolded lines to main App.js - for now we'll keep the **`SayHello`** *component mounted.*

```
function App() {
    return (
     <div>
      <SayHello ... />
      <DisplayRandomNumber number={23423423} />
     </div>
    );
}
```

React doesn't allow you to put more than one element in a render method without containing it, such as with a **`div`**, **`p`** or **`ul/li`** structure. You can also use **`Fragment`** if you import it from **`react`**.

create DisplayRandomNumber.js and set its contents to this

```
import React from 'react';
const DisplayRandomNumber = (props) =>
  <span>
    {props.number.toLocaleString()}              ①
  </span>;

export default DisplayRandomNumber;
```

① You can execute functions on objects (beware of undefined!)

Exercise 7 - Adding CSS styles

Add a style to the application-level stylesheet

Add to App.css:

```
.bigNumber {
  font-size: 5em;
}
```

Now, update DisplayRandomNumber to use the **`className`** *prop to attach the class*

```
const DisplayRandomNumber = (props) =>
  <span className="bigNumber">                ①
    {props.number.toLocaleString()}
  </span>;
```

① React uses **`className`** rather than **`class`** because **`class`** is a reserved JavaScript word.

This is a *global style* as anything added to a normal CSS stylesheet has no concept of locality. It's just added to the styles in the page. There is a way around this, actually several, and we'll look at two of them.

Prove it by adding the same style to App.js

```
<div>
  <p>This is a <span className="bigNumber">BIG number</span></p>   ①
  <SayHello... />
...
```

① This span will use the same globally available style, **`bigNumber`**.

Global styling may be what you want. But there is a way to localize styles per component by namespacing them. One technique is by creating a CSS module.

EXERCISE 8 - Using CSS Modules

Do the following tasks to set up for using CSS modules:

1. Create DisplayRandomNumber.module.css and copy .bigNumber style to it
2. Remove .bigNumber style from .App
3. Change DisplayRandomNumber:

```
import * as styles from './DisplayRandomNumber.module.css';   ①

const DisplayRandomNumber = (props) =>
  <span className={styles.bigNumber}>                         ②
    {props.number.toLocaleString()}
  </span>;
```

① Special CSS import enabled via **`Create React App`** pulls all styles in as properties of an object named **`styles`**. The file name **must** be suffixed as **`.module.css`**. This also works with SASS if you've installed **`node-sass`**.

② Styles are applied by using the dynamic binding of **`styles.styleName`** which also automatically adds a randomized attribute to the component and the style when rendered. This style will *not* bleed out to other components with different classes or function names.

Refresh, inspect with chrome and see the CSS being namespaced appropriately. Try accessing the same style className in the **`App.js`** component and you won't be able to mount it without actually importing the CSS file.

Use Style modules to prevent novel styles from bleeding out into the HTML page outside of your component. Also install the **`classnames`** NPM module if you'd like to mix these names with other styles.

EXERCISE 9 - A class-based component

Now we'll switch to building a class-based component, **`TaskList`**, which will ultimately manage a list of tasks.

1. Remove all JSX markup in the render method of App.js (we'll place the TaskList component in their place). Just leave the return statement with a String like 'NOTHING TO SEE HERE'.
2. Create a new component, **`TaskList.js`**, in **`src`**:

   Create **`TaskList.js`** *and fill it with the following code:*

```
import React, {Component} from 'react';                ①

export default class TaskList extends Component {      ②

  render() {                                           ③
    return (
      <div className="taskList">
        <h2>Task List</h2>
        <hr />
        <ul>
          <li>Task One</li>
          <li>Task Two</li>
        </ul>
      </div>
    );
  }
}
```

   ① Import the **`Component`** named export from the **`React`** module.

   ② All React components extend the **`Component`** class defined in the **`react`** module. This gives them a set of lifecycle methods.

   ③ All React components *must* implement a **`render`** function.

The function-based components essentially define the **`render`** method of an anonymous component. When they are compiled they are turned into a class component.

3. Now, Import and mount your task list:

Change your **App.js** *component to look like this:*

```
import React from 'react';
import TaskList from './TaskList';
import './App.css';

function App() {
  return <TaskList />;
}

export default App;
```

EXERCISE 10 - Introducing State

In this sample, we'll add a special member variable that only stateful components possess, the **state** attribute. The primary reason for a class-based component is to hold and manage state.

Modify your **`TaskList.js`** *component with the following changes in bold*

```
import React, { Component } from 'react';
import * as styles from './TaskList.module.css';

export default class TaskList extends Component {

  state = {
    tasks: [
      { description: 'Wash the floors', complete: false },
      { description: 'Do the dishes', complete: false },
      { description: 'Clean your clothes', complete: false },
      { description: 'Hang out at the mall', complete: true }
    ]
  };

  render() {
    const checkText = <span>&#10003;</span>;
    const taskList = this.state.tasks.map((task, idx) =>
      return (
        <div className={styles.task} key={idx}>
          <span className={styles.completed}>{task.complete ? checkText : ' '}</span>
          <span className={styles.description}>{task.description}</span>
        </div>
      );
    });
    return (
      <div className="taskList">
        <h2>Task List</h2>
        <hr />
        { !!taskList && taskList }
      </div>
    );
  }
}
```

We've done away with the UL/LI setup. DIVs and SPANs are a better way to go generally. They can be styled in dozens of different ways.

EXERCISE 11 - Update State

Now we'll do something simple with the state to modify it: complete a task. We'll add a button to the left of each task if the task is not complete yet. Once the button is clicked, we'll remove the button and show the checkmark by modifying the state. Once the state is modified, we re-render automatically.

Modify bolded lines below in **`TaskList.js`**.

```
import React, { Component } from 'react';
import * as styles from './TaskList.module.css';
export default class TaskList extends Component {

  state = {
    tasks: [
      { description: 'Wash the floors', complete: false },
      { description: 'Do the dishes', complete: false },
      { description: 'Clean your clothes', complete: false },
      { description: 'Hang out at the mall', complete: true }
    ]
  };

  render() {
    const checkText = <span>&#10003;</span>;
    const taskList = this.state.tasks.map((task, idx) => {
      return (
        <div className={styles.task} key={idx}>
          {
            !task.complete &&                                    ①
            <button onClick={() => {                             ②
              this.completeTask(idx);                            ③
            }}>
              Complete task
            </button>
          }
          <span className={styles.completed}>
           {task.complete ? checkText : ' '}
          </span>
          <span className={styles.description}>{task.description}</span>
        </div>
      );
    });
    return (
      <div className="taskList">
        <h2>Task List</h2>
        <hr />
        { !!taskList && taskList }
      </div>
    );
  }

  completeTask = (idx) => {                                      ④
    this.setState((state, props) => {
      return {
```

```
        tasks: state.tasks.map((task, targetIdx) => {          ⑤
          return idx === targetIdx ? { ...task, complete: !task.complete } : task
        })
      };
    })
  }
}
```

① A conditional rendering expression begins with a boolean assertion. The assertion is joined with the JSX expression to render with an **&&** (AND) operation. In this case, if the condition is true, the button is rendered.

② React has **Synthetic Events**, which are attached to the reactivity and lifecycle of a React component. They are camel-cased, and should be used to bind functions to events.

③ We'll call the **completeTask** function in the component. The **this** keyword is automatically available based on our arrow function syntax.

④ An *arrow method* is an arrow function bound as a method name inside of a component. These arrow methods are a way to make sure nothing steals the **this** keyword between the click of the event and evaluating the function's script. This is a relatively new feature of React, and replaces the old-style **bind** function you may see in older tutorials.

⑤ The state assignment example above may seem a bit complex. We'll break it down this way: update the tasks based on their current state in the component (see the **this.setState((state, props)** arrow method on the next line). For each task, either return it as it currently exists, or if the index equals the index we want to change, replace it with a *new object* containing its current values, overwriting the **complete** flag by inverting it. This completes the uncompleted task.

Once the state is updated, the component redraws automatically.

Use arrow methods like **completeTask** above to ensure the **this** method is always bound when calling methods from events.

EXERCISE 12 - Rein in the complexity with functional components

Now we'll tie together functional components (for display) and class-based components (for state management).

1. First, rename **TaskList.module.css** as **TaskItem.module.css** so we can use the styles in a new functional component.
2. Next, create a new file: **TaskItem.js**. We'll use that to extract the details of rendering and managing each task.
3. Now, define the contents of the **TaskItem.js** component. Feel free to cut and paste.

Fill in the contents of **`TaskList.js`***:*

```
import React from 'react';
import * as styles from './TaskItem.module.css';

const TaskItem = ({task, index, onCompleteTask}) => {.        ①
  const checkText = <span>&#10003;</span>;
  return (
    <div className={styles.task}>
          {!task.complete &&
            <button onClick={() => {
              onCompleteTask(index);
            }}>
              Complete task
            </button>
          }
          <span className={styles.completed}>
           {task.complete ? checkText : ' '}
          </span>
          <span className={styles.description}>{task.description}</span>
    </div>
  );
}

export default TaskItem;
```

① Destructure **`props`** into the fields we're interested in.

What is destructuring, anyway?

Destructuring is a way of automatically creating local reference variables from properties of the passed argument to a function. The render method in a functional component is passed the **`props`** as the only argument. We destructure that into the three props we are going to pass from the parent: **`task`** (the current task), **`index`** (the index / key of the current task to report back when we complete it), and **`onCompleteTask`**, a prop passed that represents the event we call when we complete a task. See the next snippet for how these are passed.

4. Finally, edit TaskList to remove inner list and replace with references to TaskItem

Replace in-line task item code with new component

```
import React, { Component } from 'react';
import TaskItem from './TaskItem';
export default class TaskList extends Component {

  state = {
    tasks: [
      { description: 'Wash the floors', complete: false },
      { description: 'Do the dishes', complete: false },
      { description: 'Clean your clothes', complete: false },
      { description: 'Hang out at the mall', complete: true }
    ]
  };

  render() {
   const taskList = this.state.tasks.map((task, idx) => {
      return (
        <TaskItem                                          ①
          onCompleteTask={this.completeTask}               ②
          task={task}                                      ③
          key={idx}                                        ④
          index={idx} />                                   ⑤
      );
    });
    return (
      <div>
        <h2>Task List</h2>
        <hr />
        { !!taskList && taskList }
      </div>
    );
  }

  completeTask = (idx) => {
    this.setState((state, props) => {
      return {
        tasks: state.tasks.map((task, targetIdx) => {
          return idx === targetIdx ? { ...task, complete: !task.complete } : task
        })
      };
    })
  }
}
```

① Our new **`TaskItem`** component greatly simplifies the outer **`TaskList`** component.

② Here we create our own synthetic event, naming it **`onCompleteTask`**, which makes it available in the **`TaskItem`** component as a **`prop`** with the same name. We can call it as a function. We bind it to the arrow method we defined before, **`completeTask`**.

③ The **`task`** data is passed downward to the task item component.

④ The key is used as before.

⑤ The index is also passed. This is how we can tell the parent *which* task we clicked on.

You can't access the **`key`** prop of any component within it. Hence we pass the index from our mapping operation to another prop, **`index`**, so we can access it in **`TaskItem`**.

Exercise 13: Getting data in and out with Ajax

Now we'll install a "database" server (actually the awesome and extremely useful **`json-server`**) and load and save our tasks to it.

1. First, install some dependencies:

```
npm install --save-dev json-server concurrently
```

`concurrently` runs two or more processes together from a single NPM script, and **`json-server`** runs a web server that exposes a JSON document as if it was an actual database.

2. Create a file at the root of the project. Name it **`db.json`** and copy the **`tasks`** collection into it. Quote all property names with double-quotes, and replace the single quotes with double quotes. Finally, wrap the entire thing as an object. This is a JSON-formatted file:

   The new **`db.json`** *database file for* **`json-server`**. *You must add an* **`id`** *field and double-quote all strings and non-numeric properties:*

```
{
  "tasks": [
    { "id": 1, "description": "Wash the floors", "complete": false },
    { "id": 2, "description": "Do the dishes", "complete": false },
    { "id": 3, "description": "Clean your clothes", "complete": false },
    { "id": 4, "description": "Hang out at the mall", "complete": true }
  ]
}
```

3. Edit **`package.json`** and add a script to start up the **`json-server`** engine, rename the current **`start`** method as 'start-app', and weld both processes together with a new **`start`** script:

Update the scripts entry to this:

```
  "scripts": {
    "start": "concurrently \"npm run start-json\" \"npm run start-app\"",
    "start-json": "json-server --port=3001 db.json",
    "start-app": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
```

Now when you issue a **`npm start`** command, the app server runs on port 3000, and the database REST web server runs on port 3001.

4. Next, add the proxy entry. This establishes a tunnel to the **`json-server`** when an ajax call is made:

   Add the following entry to the **`package.json`** *file to define a proxy host*

```
{
  "private": true,
  "proxy": "http://localhost:3001",       <-- add this entry
  "dependencies": {
    ...
  }
  ...
}
```

5. Add an AJAX library. We recommend **`axios`** if no better choice exists:

   Install the **`axios`** *library from the command line with:*

```
npm install axios
```

6. Wire up the GET call in the **`componentDidMount`** lifecycle method of the **`TaskList`** like so:

Add the **axios** *import and the* **componentDidMount** *method to* **TaskList.js**

```
...
import * as axios from 'axios';

...

  // Add this method to the body of the class
  async componentDidMount() {
    try {
      const response = await axios.get('/tasks');
      this.setState({
        tasks: response.data
    });
    } catch (e) {
      alert(Error fetching tasks. ${JSON.stringify(e)});
    }
  }
```

Axios provides all of the major REST APIs such as **GET**, **PUT**, **PATCH**, **HEAD**, **POST** and **DELETE**.

7. Adjust the **key** and **index** props to use the primary key of each **task**. This is a more consistent way to manage the data for updates.

   Adjust the **key** *and* **index** *props in* **TaskList.js** *changing the bolded lines:*

```
  render() {
   const taskList = this.state.tasks.map(task => {
      return (
        <TaskItem
          onCompleteTask={this.completeTask}
          task={task}
          key={task.id}
          index={task.id} />
      );
    });
  ...
```

8. Fix up the **completeTask** method

Replace the code in **`completeTask`** *with the bolded code in this listing:*

```
completeTask = (id) => {
  this.setState((state, props) => {
    return {
      tasks: state.tasks.map(task => {
        return id === task.id ? { ...task, complete: !task.complete } : task
      })
    };
  })
}
```

----- END OF STRUCTURED LAB -----