CSE 331

Full Stack Apps

Kevin Zatloukal

Review: Stateful React Components

```
type HiProps = {name: string};
type HiState = {greeting: string};

class HiElem extends Component<HiProps, HiState> {
  constructor(props: HiProps) {
    super(props);
    this.state = {greeting: “Hi”};
  }

  render = (): JSX.Element {
    return (<div>
        <p>{this.state.greeting}, {this.props.name}!</p>
        <button onClick={this.makeSpanish}>Espanol</button>
      </div>);
  };

  makeSpanish = (evt: MouseEvent<HTMLButtonElement>) => {
    this.setState({greeting: “Hola”});
  };
```

React Components are Like ADTs

- Components have an invariant like an RI

  $$\text{HTML on screen} = \text{render}(\text{this.state})$$

  – **don't want to be in a state where that is not true**

    unless you like painful debugging!

1. **Do not mutate `this.state` (call `setState`)**

   React will update `this.state` and HTML on screen at the same time

2. **Make sure no data on screen would disappear on re-render**

Mirror All UI State in Component State

- Any state on the screen must be stored in some state
  - text in any INPUT element must be in some state

```
type MyState = {text: string, …};

render = () => {
  … <input type="text" value={this.state.text}
           onChange={this.onTextChanged}></input> …
};

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

  - updated on every character typed!

    this is not slow (typing is very slow)

Example: To-Do List

React Gotchas #1

- Make sure you declare your methods this way

```
onClick = (evt: MouseEvent<HTMLButtonElement>) => { … };
```

  - otherwise, the event handlers won’t work
  - debugging that will be painful

React Gotchas #2

- Note that `setState` is not instant

```
// this.state.x is 2
this.setState({x: 3});
console.log(this.state.x);  // still 2!
```

  - it adds an event that later updates the state
  - (React tries to batch together multiple updates)

React Gotchas #3

- Any state on the screen must be stored in some state
  - text in any INPUT element must be in some state

```
type MyState = {text: string, …};

render = () => {
  … <input type=“text” value={this.state.text}
           onChange={this.onTextChanged}></input> …
};
```

More React Gotchas

- **Never modify anything in render**
  - **should be a pure function**
- **Never modify `this.state` outside of constructor**
  - **use `this.setState` instead**
- **Remember that debugging will suck**
  - **stateful components are inherently complex (Level 3)**
  - **separate anything complex into helper functions**
    - reason through them carefully and test them thoroughly
    - can have helper function that calculates new states, HTML to display, ec.
  - **write code to also check things at run time**

More Events

Events

- Components update their state when events occur
  - event calls a “handler”, which is a method of the class
  - event handler updates state via `setState`
- Some common examples
  - button click, hyperlink click
  - typing in text field
  - check box clicked
  - drop-down changed
  - timers
- See MDN for all possible events…

Button Click Events

```
<button onClick={this.handleClick}>Click Me</button>
```

- **Click results in a call to our method**

```
handleClick = (evt: MouseEvent<HTMLButtonElement>) => {
  console.log("I've been clicked");
};
```

- **Event handlers are passed an event object**
  - **mouse clicks send** `MouseEvent` **objects**

    generic type with a parameter identifying the target of the click

Link Click Events

```
<a href="#" onClick={this.handleClick}>Click Me</a>
```

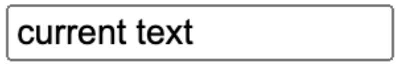
- Click results in a call to our method

```
handleClick = (evt: MouseEvent<HTMLAnchorElement>) => {
  evt.preventDefault();  // don't change the URL
  console.log("I've been clicked");
};
```

- Default action of a link is to go to that URL
  - harmless in this case (just adds "#" to the end of the URL)
  - can stop that with `evt.preventDefault()`

Text Field Events

```
<input type=“text” value=“current text”
       onChange={this.handleChange}></a>
```



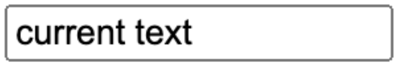
- **Any typing in the text box causes a call to**

```
handleChange = (evt: ChangeEvent<HTMLInputElement>) => {
  console.log(“Text is now: ${evt.target.value}”);
};
```

  - `evt.target` **is the thing that was clicked on**
    has type `HTMInputElement` in this case
  - **“value” attribute of the input text field is changing**
  - **“value” is the text currently shown in the text field**

Text Field Events

```
<input type="text" value="current text"
       onChange={this.handleChange}></a>
```


current text

- Any typing in the text box causes a call to

```
handleChange = (evt: ChangeEvent<HTMLInputElement>) => {
  console.log("Text is now: ${evt.target.value}");
};
```

- This code has a bug! What is it?
  - a re-render would overwrite value!

Text Field Events

```
<input type="text" value={this.state.curText}
       onChange={this.handleChange}></a>
```

- **Any typing in the text field calls our change handler:**

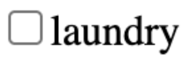
```
handleChange = (evt: ChangeEvent<HTMLInputElement>) => {
  this.setState({curText: evt.target.value});
};
```

- **We update** `curText` **to match the HTML on screen**
  - **restores the invariant:** HTML on screen = render(this.state)
  - **re-render leaves the screen unchanged**

- **Any text field should have state that stores its value**

Check Box Events

```
<input type="checkbox" id="myCheckBox"
       onChange={this.handleChange}/>
<label htmlFor="myCheckBox">laundry</label>
```



- **Clicking inside the box**

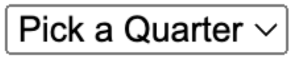
```
handleChange = (evt: ChangeEvent<HTMLInputElement>) => {
  console.log("Checked? ${evt.target.checked}");
};
```

  – `evt.target.checked` **is true / false**

- **Label contains the text to show next to the check box**
  – `htmlFor` **is useful for screen readers**

Drop-Downs

```
<select>
  <option value=“NA”>Pick a Quarter</option>
  <option value=“20au”>Fall 2020</option>
  <option value=“21sp”>Spring 2021</option>
</select>
```



- **HTML** `select` **element creates a drop-down**
  - **one option for each choice**
  - **text in between** `<option>` **and** `</option>` **is shown**
  - **“**`value`**” is used by event handlers…**

Drop-Downs (HTML Select)

```
<select onChange={this.handleChange}>
  {options}
</select>
```

- **Picking an option causes an onChange**

```
handleChange = (evt: ChangeEvent<HTMLSelectElement>) => {
  console.log(“Picked option: ${evt.target.value}”);
};
```

  - `evt.target.value` **is the “**`value`**” from the** `option` **chosen**
  - **“**`value`**” has type** `string`

Timers

```
setTimeout(this.handleTimer, 500);
```

- Calls the handler after 500 milliseconds

```
handleTimer = () => {
  console.log("Timer went off!");
};
```

  - no arguments provided

Arguments to Event Handlers

- Often want to pass arguments to event handlers
  - can do so like this:

```
setTimeout(() => this.handleTimer(“egg”), 500);

handleTimer = (name: string) => {
  console.log(“${name} timer went off!”);
};
```

  - creates a new function on the spot
  - when called, that function calls `handleTimer` with the arg

Arguments to Event Handlers

- **The same thing applies to all other event handlers, e.g.**

```
<input type="checkbox" id="myCheckBox"
       onChange={(evt) => this.handleChange(evt, "laundry")}/>
<label htmlFor="myCheckBox">laundry</label>



handleChange = (evt: ChangeEvent<HTMLInputElement>,
                name: string) => {
  console.log("Done with ${name}? ${evt.target.checked}");
};
```

  - **event handler takes the event and an argument**

    setTimeout, in contrast, does not pass an event objecvt

Example: To-Do List

Client & Server

Making HTTP Requests

- Send / receive data from the server with fetch

```
fetch("/add?name=laundry")
    .then(this.handleServerResponse)
    .catch(this.handleServerError)
```

  - then handler is called if the request can be made
  - catch handler is called if it cannot be
    - only if it could not connect to the server at all
    - status 400 still calls then handler

- Fetch returns a "Promise" object
  - has then/catch methods
  - then/catch methods return the object again
    - allows method calls to be chained in one expression like this

Making HTTP Requests

- **Still need to check for a 200 status code**

```
handleServerResponse = (res: Response) => {
  if (res.status === 200) {
    console.log(“it worked!”);
  } else {
    this.handleServerError(res);  // it failed
  }
};

handleServerError = (res: Response) => {
  console.log(“something bad happened”);
};
```

  - **(need to tell users about errors with some UI…)**

Handling HTTP Responses

- Response has methods to get data returned by server
  - `res.json()` if the server returned JSON (a record)
  - `res.text()` if the server returned text (a string)
  - sadly, these methods do not return record / string…

- Server response could be HUGE (gigabytes)
  - may take a long time to download it all

- Methods above return Promises to get those things
  - use then to add a handler that is called with the data

Making HTTP Requests

```
handleServerResponse = (res: Response) => {
  if (res.status === 200) {
    res.json().then(this.handleServerData);
              .catch(this.handleServerError);
  } else {
    this.handleServerError(res); // it failed
  }
};
```

- Second promise can also fail
  - e.g., fails to parse as valid JSON, fails to download

- Important to catch every error
  - painful debugging if an error occurs and you don’t see it!

Making HTTP Requests

```
handleServerResponse = (res: Response) => {
  if (res.status === 200) {
    res.json().then(this.handleServerData);
              .catch(this.handleServerError);
  } else {
    this.handleServerError(res); // it failed
  }
};
```

- **type of returned data is unknown**
- **to be safe, we should write code to check that it looks right**

  check that the expected fields are present

  check that the field values have the right types

HTTP GET vs POST

- When you type in a URL, browser makes “GET” request
  - request to read something from the server
- Clients often want to write to the server also
  - this is typically done with a “POST” request
    - ensure writes don’t happen just by normal browsing
- POST requests also send data to the server
  - GET only sends data via query parameters
  - limited to a few kilobytes of data
  - POST requests can send arbitrary amounts of data

HTTP GET vs POST

- Extra parameter to fetch changes request type

```
fetch(”/add?name=laundry”, {method: ”POST”})
```

- Can optionally pass data to the server this way

```
fetch(”/add”, {
  method: ”POST”,
  body: JSON.stringify({”name”: ”laundry”})
})
```

  – may also need another field:

```
headers: {”Content-Type”: “application/json”}
```

Example: To-Do List 2.0