ECE 448/528
Application Software Design

Lecture 24. The React JavaScript Library
Spring 2025

Won-Jae Yi, Ph.D.

Department of Electrical and Computer Engineering
Illinois Institute of Technology

UI Inputs and UI States

UI Inputs

- Now it is time to add the last row to our members table so that new groups can be created.
- For such purpose, two text inputs are used.
  - As we discussed in Lectures 19 and 20.
- It is quite intuitive for users to use the text inputs.
  - As users type, new letters appear.
- It is quite intuitive for browsers to manage the content of those inputs so developers can retrieve theirs values later.
  - As we have done in Lecture 19.
- Question: from the perspective of MVC pattern, is the content of a text input part of the model or not?

UI States

- In MVC pattern, view should only depend on the model.
  - Not on what users have typed.
- The content of a text input should be part of the model.
  - UI states: models that do not depend on a data model from the server backend.
- As a consequence, we cannot let browsers manage the content of the text inputs as users type.
  - This is counter-intuitive but is required to ensure the consistency of the MVC pattern.
- Note that we have already done so for the checkboxes.
  - Browsers will not check or uncheck the checkboxes when users click.
  - We handle it with the click event, send a RESTful request, and update the model (thus views) as response returns.

Handling UI Inputs via UI States in React

- Instead of letting browsers manage the contents of text inputs, React uses the following sequence to manage UI inputs.

1. Capture user input events
2. Update state/models in the controller
3. Then, Views are rendered automatically, with the text inputs showing relevant UI states as the content.

- These are simpler than handling UI inputs that are related to backend RESTful data models like the checkboxes.
    - Though in those cases, the needs to use models are more intuitive than these cases where there is no corresponding backend data model.

Adding UI States

```
class Members extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      members: create_members_model([]),
      inputName: "",
      inputMembers: "",
    };
  }
  ...
}
```

- In `public/web/members.js` under branch `lec24-input`
- UI states for the two text inputs are introduced as part of the controller state.
- Initial states determine what users will see first.
- Hierarchy may be introduced in state if more complex UI states need to be included.

Handling UI States Changes

```
class Members extends React.Component {
  ...
  onInputNameChange = value => {
    this.setState({inputName: value});
  }
  onInputMembersChange = value => {
   this.setState({inputMembers: value});
  }
  ...
```

- Views will need to associate onInputNameChange and onInputMembersChange with the text inputs and retrieve what the user has typed.
- Don’t forget to use the setState method.

Passing States and Handlers

```
class Members extends React.Component {
  ...
  render() {
    return (<MembersTable members={this.state.members}
      inputName={this.state.inputName} inputMembers={this.state.inputMembers}
      ...
      onInputNameChange={this.onInputNameChange}
      onInputMembersChange={this.onInputMembersChange}
      ...
      />);
  }
  ...
```

- States and handlers can be passed to views as usual.

Capturing User Inputs

```
function AddGroup(props) {
  var onChangeName = event => props.onInputNameChange(event.target.value);
  var onChangeMembers = event => props.onInputMembersChange(event.target.value)
  return (<div>
    <label>Group Name</label>
    <input type="text" onChange={onChangeName} value={props.inputName}/>
    <label>Members</label>
    <input type="text" onChange={onChangeMembers} value={props.inputMembers}
      ...
```

- For each UI element that is controlled by a UI state,
  - Use the UI state to set its content – for text inputs, this is done by setting the value attribute.
  - Associate the onChange handler to capture user inputs – for text inputs, the onChange handler can make use of the parameter event to obtain what users' type.
- Note that for render or stateless component you can only return a single tag.
  - Surround multiple tags with div if you need to return them.

Making Use of UI States

```
class Members extends React.Component {
  ...
  onAddGroup = () => {
    var name = this.state.inputName;
    var members = this.state.inputMembers.split(',');
    this.createGroup(name, members);
  }
  ...
}
```

- onAddGroup will handle the event when users click the Add/Replace button to add or replace a group.
- There is no need to retrieve the values from the text inputs like Lecture 19.
  - The state/models already have all information available.
- Pass onAddGroup to the views and associate it with the button.

Advanced UI Features

Handling Multiple RESTful Requests

```
class Members extends React.Component {
  ...
  onAddMemberToAllGroups = memberName => {
    var groups = [];
    for (var groupName of this.state.members.get_group_names()) {
      ...
    }
    this.createManyGroups(groups);
  }
  ...
}
```

- Multiple groups may need to be updated if users click the member name to add that member to all groups.
- How would you implement createManyGroups to send multiple RESTful requests to update those groups?
  - Use a for loop to update each group with createGroup.
  - However, doing so may trigger multiple getGroup's as each createGroup will trigger one.
- Ideally, we should wait all create group requests to finish before we send the get-all-groups request once.

Working with Multiple Promises

```
class Members extends React.Component {
  ...
  createManyGroups = groups => {
    var pendingReqs = groups.map(group => {
      var postReq = {...};
      return fetch("api/groups/"+group.name, postReq);
    });
    Promise.all(pendingReqs).then(() => this.getGroups())
      .catch(err => console.error("Members: createManyGroup", err));
  }
  ...
}
```

- The asynchronous nature of HTTP requests makes it impossible to determine which RESTful request completes the last.
- Since each fetch returns a Promise, we should use Promise.all to wait for all Promises to complete.
  - Then we can send the get groups request.

Distributed Web Applications

- Web applications are distributed applications.
  - Multiple web frontends may access the server backend at the same time.
  - One user may interact with one web frontend to update a RESTful data model at the server backend.
  - At this moment, if the server backend can broadcast the updates to all web frontends, then users of other web frontends may see the interaction.
- RESTful requests are always initiated by the web frontend; the server backend needs another mechanism to notify the web frontend.
- WebSocket: a protocol compatible with HTTP that allows server backends to send information to web frontends.
  - But what if WebSocket is not available for some reason?

Polling

```
class Members extends React.Component {
  ...
  componentDidMount() {
    this.getGroups();
    setInterval(this.getGroups, 1000);
  }
  ...
}
```

- We may set up a timer for the web front to periodically poll the server backend for any updates.
- Trade-offs
  - How difficult is it to introduce new services like WebSocket?
  - How frequently are the updates?
  - How many users may connect at the same time?