

Form & Validation

CS568 – Web Application Development I

Computer Science Department

Maharishi International University

Maharishi International University - Fairfield, Iowa



All rights reserved. No part of this slide presentation may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying or recording, or by any information storage and retrieval system, without permission in writing from Maharishi International University.

Content

- Controlled & Uncontrolled Element
- Form validation
- Refs
- HTTP & AJAX
- Axios

Web form

A webform or HTML form on a web page allows a user to enter data that is sent to a server for processing. Forms can resemble paper or database forms because web users fill out the forms using checkboxes, radio buttons, or text fields.

React forms are slightly different than the HTML form.

Name	Value
Name	<input type="text"/>
Sex	<input type="radio"/> Male <input checked="" type="radio"/> Female
Eye color	<input type="text" value="green"/>
Check all that apply	<input type="checkbox"/> Over 6 feet tall <input type="checkbox"/> Over 200 pounds
Describe your athletic ability: <input type="text"/>	
<input type="button" value="Enter my information"/>	

Controlled Components

In HTML, form elements such as `<input>`, `<textarea>`, and `<select>` typically **maintain their own state** and update it based on user input.

In React, mutable state is typically kept in the state property of components, and **only updated with `setState()`**.

An input form element whose value is controlled by React in this way is called a “**controlled component**”.

Read more: [React forms](#)

```
class NameForm extends React.Component {
  state = {value: ''};

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  handleSubmit(event) {
    event.preventDefault();
    alert('A name was submitted: ' + this.state.value);
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <input type="text" value={this.state.value} onChange={this.handleChange} />
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

The textarea Tag

In HTML, a `<textarea>` element defines its text by its children:

```
<textarea>  
  Hello there, this is some text in a text area  
</textarea>
```

In React, a `<textarea>` uses a value attribute instead

```
<textarea value={this.state.value} onChange={this.handleChange} />
```

The select Tag

In HTML, the default option is selected with selected attribute.

```
<select>  
  <option selected value="coconut">Coconut</option>  
  <option value="mango">Mango</option>  
</select>
```

In React, it uses a value attribute on the root select tag

```
<select value={this.state.value} onChange={this.handleChange}>
```


Form validation

There are three main reasons why we insist on validating forms:

1. We want to get the right data in the right format
2. We want to protect our users
3. We want to protect ourselves' data

We can validate form data on server side or client side.

Before submitting data to the server, it is important to ensure all required form controls are filled out, in the correct format. This is called client-side form validation, and helps ensure data submitted matches the requirements set forth in the various form controls.

Form validation

There are three ways to validate the user input:

1. Writing your own validation library using built-in form validation uses HTML 5 features – You have full control on it. But it will take effort and time to get it done.
2. Using third-party libraries – Saves effort and time to develop validation. But you may not validate some complex fields.
3. Shared library in the company – Reusing the code but you need to ping to the other team to request changes.

You can validate the form on the following events:

- OnChange
- OnClick (of the submit button)

Uncontrolled Components

In most cases, React recommends using controlled components to implement forms. In a controlled component, form data is handled by a React component. The alternative is uncontrolled components, where form data is **handled by the DOM itself**.

To write an uncontrolled component, instead of writing an event handler for every state update, you can use a **Ref** to get form values from the DOM.

Refs

Refs provide a way to access DOM nodes or React elements created in the render method. Similar to `const el = document.getElementById("id");`

It gets the value from the DOM, not from the React state. It is uncontrolled by React.

When to use Refs:

- Need to work with the DOM element directly.
- Managing focus, text selection, or media playback.
- Triggering imperative animations.

Read more: [React ref](#)

```
1  import React, { Component } from 'react'
2
3  class RefsDemo extends Component {
4    constructor(props) {
5      super(props)
6      this.inputRef = React.createRef()
7    }
8
9    componentDidMount() {
10      this.inputRef.current.focus()
11      console.log(this.inputRef)
12    }
13
14    render() {
15      return (
16        <div>
17          <input type="text" ref={this.inputRef} />
18        </div>
19      )
20    }
21  }
```

```
function CustomTextInput(props) {  
  // textInput must be declared here so the ref can refer to it  
  const textInput = useRef(null);  
  
  function handleClick() {  
    textInput.current.focus();  
  }  
  
  return (  
    <div>  
      <input  
        type="text"  
        ref={textInput} />  
      <input  
        type="button"  
        value="Focus the text input"  
        onClick={handleClick}  
      />  
    </div>  
  );  
}
```

Example of Imperative animation

```
// Make an instance of two and place it on the page.
var elem = document.getElementById('draw-shapes').children[0];
var params = { width: 280, height: 200 };
var two = new Two(params).appendTo(elem);
// two has convenience methods to create shapes.
var circle = two.makeCircle(72, 100, 50);
var rect = two.makeRectangle(150, 50, 100, 100);
// The object returned has many stylable properties:
circle.fill = '#FF8000';
circle.stroke = 'orangered'; // Accepts all valid css color
circle.linewidth = 5;
rect.fill = 'rgb(0, 200, 255)';
rect.opacity = 0.75;
rect.noStroke();
// Don't forget to tell two to render everything
// to the screen
two.update();
```

Example of Declarative animation

As you can see declarative description in this case is much shorter and is very explicit. In fact you don't even need any library.

Learn more: [Imperative vs Declarative drawing API](#)

```
<div id='#draw-shapes'>
  <svg width="280" height="200">
    <circle cx="72" cy="100" r="50" fill="#FF8000"
      stroke="orangered" stroke-width="5" />

    <rect x="150" y="50" width="100" height="100"
      fill="rgb(0, 200, 255)" opacity="0.75">
  </svg>
</div>
```


Declarative vs imperative programming languages

Declarative programming is a paradigm describing WHAT the program does, without explicitly specifying its control flow. Imperative programming is a paradigm describing HOW the program should do something by explicitly specifying each instruction (or statement) step by step, which mutate the program's state.

SQL, HTML – Declarative

JS – Both (array `map()` is declarative and there is a for-loop)

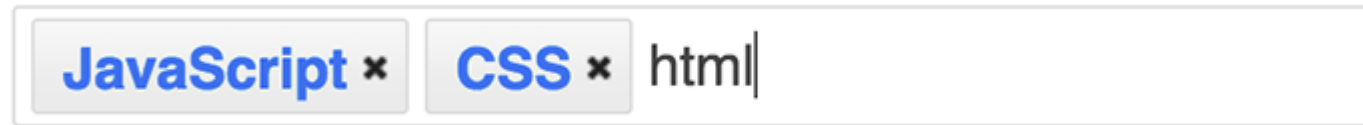
Java – Mostly imperative. Since Java 8 Lambda, it has also become a bit declarative.

Integrating React with JQuery library via Ref

The third party JQuery library is [tag-it](#). transforms an unordered list to input field for managing tags:

```
<ul>
  <li>JavaScript</li>
  <li>CSS</li>
</ul>
```

To:



```
$('#<dom element selector>').tagit();
```

To make it work we have to include jQuery, jQuery UI and the tag-it plugin code. It works like that. [Source](#)

Integrating React with JQuery library – App.jsx

```
class App extends React.Component {  
  constructor(props) {  
    super(props);  
  
    this.state = { tags: ['JavaScript', 'CSS' ] };  
  }  
  render() {  
    return (  
      <div>  
        <Tags tags={ this.state.tags } />  
      </div>  
    );  
  }  
}
```

Integrating React with JQuery library – Tags.jsx

```
class Tags extends React.Component {
  componentDidMount() {
    // initialize tagit
    $(this.refs.list).tagit();
  }
  render() {
    return (
      <ul ref="list">
        {
          this.props.tags.map(
            (tag, i) => <li key={ i }>{ tag } </li>
          )
        }
      </ul>
    );
  }
};
```

Hypertext Transport Protocol (HTTP)

HTTP is the underlying protocol used by the World Wide Web and this **protocol** defines how messages are formatted, and what actions Web servers and browsers should take in response to various commands.

HTTP Verbs

- GET: Retrieves data from the server.
- POST: Submits data to the server.
- PUT: Replace data on the server.
- PATCH: Partially update a certain data on the server.
- DELETE: Delete data from the server.

AJAX

Asynchronous JavaScript and XML

- Not a programming language, but another asynchronous JavaScript API
- Downloads data from a server in the background
- Allows dynamically updating a page without making the user wait
- Avoids the "click-wait-refresh" pattern

Axios

Axios is a promise-based HTTP Client for node.js and the browser.

- Make **XMLHttpRequests** from the browser and **node.js**
- By default, axios **serializes** JavaScript objects to **JSON**
- Supports the Promise API **Intercept** request and response
- Client-side support for protecting against XSRF

<https://axios-http.com/docs/intro>

Sending Get Request

```
//Blog.js

componentDidMount() {
  axios.get('https://jsonplaceholder.typicode.com/posts')
    .then((response) => {
      console.log(response);
    });
}
```

Response Schema

```
{  
  data: {},  
  status: 200,  
  statusText: 'OK',  
  // headers from the server  
  headers: {},  
  // config that was provided to `axios` for the request  
  config: {},  
  // request that generated this response  
  request: {}  
}
```

Config

You can specify config defaults that will be applied to every request.

Learn more about Axios [request config](#).

```
axios.defaults.baseURL = 'https://api.example.com';  
axios.defaults.headers.common['Authorization'] = AUTH_TOKEN;
```

Interceptors

You can intercept requests or responses before they are handled by then or catch.

Interceptors are common in programming. It obstructs requests or responses to prevent them from continuing to a destination. For example, you can implement an interceptor that stops malicious requests.

In Axios, the interceptors can be used to add common headers like authorization header or to log the requests etc.

Logging requests with Interceptors

```
//index.js
axios.interceptors.request.use(request => {
  console.log(request);
  return request;
}, error => {
  console.log(error);
  return Promise.reject(error);
});
```

Rendering Data to the Screen

```
class Blog extends Component {
  state = {
    posts: []
  }
  componentDidMount() {
    axios.get('https://jsonplaceholder.typicode.com/posts')
      .then((response) => {
        this.setState({ posts: response.data });
      });
  }
  render() {
    const posts = this.state.posts.map(item => {
      return <Post
        key={item.id}
        title={item.title}>
      </Post>

    });
  }
}
```

```
return (
  <div>
    <section className="Posts">
      {posts}
    </section>
    <section>
      <FullPost />
    </section>
    <section>
      <NewPost />
    </section>
  </div>
);
}
```

Posting Data to Server

```
//NewPost.js
class NewPost extends Component {
  state = {
    title: '',
    content: '',
    author: 'Umur'
  }
  postDataHandler =()=>{
    const post = {
      title:this.state.title,
      body:this.state.content,
      author: this.state.author
    } axios.post('https://jsonplaceholder.typicode.com/posts',post)
    .then(response =>{
      console.log(response)
    });
  }
  render () {
    return (
```

Posting Data to Server

```
div className="NewPost">
  <h1>Add a Post</h1>
  <label>Title</label>
  <input type="text" value={this.state.title} onChange={(event) => this.setState({title: event.target.value})} />
  <label>Content</label>
  <textarea rows="4" value={this.state.content} onChange={(event) => this.setState({content:
event.target.value})} />
  <label>Author</label>
  <select value={this.state.author} onChange={(event) => this.setState({author: event.target.value})}>
    <option value="Umur">Umur</option>
    <option value="Aynur">Aynur</option>
  </select>
  <button onClick={this.postDataHandler}>Add Post</button>
</div>

);
}
```


Extra – Implementing forms dynamically by reading the config from the JSON file

Creating Custom Dynamic Input Component

```
//input.js
const input = (props) => {
  let inputElement = null;
  switch (props.elementType) {
    case ('input'):
      inputElement = <input {...props.elementConfig} value={props.value}/>
      break;
    case ('textarea'):
      inputElement = <textarea {...props.elementConfig} value={props.value}/>
      break;
    default:
      inputElement = <input {...props.elementConfig} value={props.value}/>
  }
  return (
    <div>
      <label>{props.label}</label>
      {inputElement}
    </div>
  )
}
```

JS Config for the Form

```
// app.js
state = {
  registerForm: {
    fname: {
      elementType: 'input',
      elementConfig: {
        type: 'text',
        placeholder: 'Name'
      },
      value: ''
    },
    lname: {
      elementType: 'input',
      elementConfig: {
        type: 'text',
        placeholder: 'Name'
      },
      value: ''
    },
  },
}
```

```
email: {
  elementType: 'input',
  elementConfig: {
    type: 'text',
    placeholder: 'Name'
  },
  value: ''
}
}
```

Dynamically Create Input based on JS Config

Creating an array from the state object to iterate over it.

```
const formElementsArray = [];  
for (let key in this.state.registerForm) {  
  formElementsArray.push({  
    id: key,  
    config: this.state.registerForm[key]  
  });  
}
```

Dynamically Create Input based on JS Config

```
let form = (  
  <form>  
    {  
      formElementsArray.map(item => {  
        return (  
          <Input key={item.id}  
            elementType={item.config.elementType}  
            elementConfig={item.config.elementConfig}  
            value={item.config.value} />  
        )  
      })  
    }  
    <button  
      onClick={ (event) => { this.formSubmitHandler(event) } }>  
      Register  
    </button>  
  </form>  
)
```

Custom Form Validation

```
fname: {
  elementType: 'input',
  elementConfig: {
    type: 'text',
    placeholder: 'Name'
  },
  value: '',
  validation: {
    required: true
  },
  valid: false
}
```

```
validate = (value, rules) => {
  let isValid = false;
  if (rules.required) {
    isValid = value.trim() !== '';
  }
  return isValid;
}

inputChangedEventHandler = (event, inputId) => {
  //console.log(event.target.value);
  const copyOfRegisterForm = { ...this.state.registerForm };
  const copyOfElement = { ...copyOfRegisterForm[inputId] };
  copyOfElement.value = event.target.value;
  copyOfElement.valid =
this.validate(copyOfElement.value, copyOfElement.validation);
  copyOfRegisterForm[inputId] = copyOfElement;
  this.setState({ registerForm: copyOfRegisterForm });
}
```