LabKey User Conference 2019

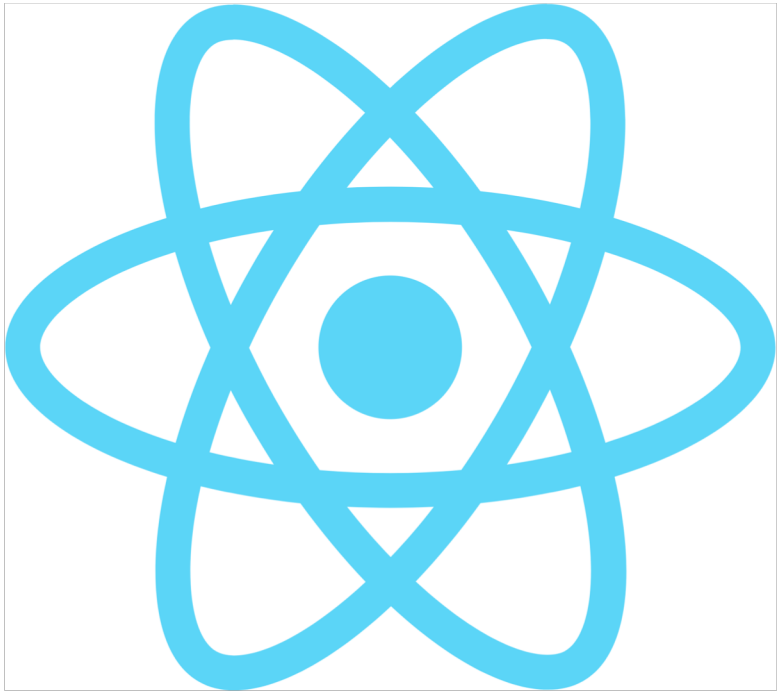
ReactJS Development: Getting Started

October 4, 2019

LabKey
PARTNERS IN SCIENCE

ReactJS Development: Getting Started

- What is React?
- ES6+
- JSX
- Components
- State Management
- Reconciliation
- Example Module
- Further Learning



What is React?

“A JavaScript library for building user interfaces”

Traditional HTML/JS Form

```
<form class="login-form">
   <div class="error-message">
   </div>

   <div class="form-row">
       <label>Sample Name:</label>
       <input type="text" name="name" />
   </div>

   <div class="form-row">
       <label>Sample Date:</label>
       <input type="text" name="date" />
   </div>

   <div class="form-row">
       <label>Sample Description:</label>
       <textarea name="description"></textarea>
   </div>

   <div class="form-row form-buttons">
       <button type="submit" onclick="cancel();">Cancel</button>
       <button type="submit" onclick="saveSample();">Save</button>
   </div>
</form>
```

Form with Components

```
// Build a form with smaller components, reducing boilerplate
<Form className="sample-form" onCancel={props.onCancel} onSave={props.onSave}>
   <Alert type="warning" message={props.error} />
   <Input type="text" name="name" label="Name" value={props.name} />
   <Input type="textarea" name="description" label="Description" value={props.desc} />
   <Input type="date" name="name" label="Date" value={props.date} />
</Form>
```

Re-usable Form

```
// Whole form can be reused. Useful for create/edit scenarios.
<SampleForm onCancel={this.onCancel} onSave={this.onSave} data={preExistingData} />
```

EcmaScript 6+

- Notable Features:
  - let/const
  - Destructuring
  - Spread operator
  - Arrow functions
  - Classes
  - Module system (import / export)
- Requires a build system

Destructuring

```
const exampleData = {
   first: 'Alan',
   last: 'Vezina',
   hobbies: [ 'Skateboarding', 'Painting', 'Cycling'],
};

// Old way, without destructuring.
var first = exampleData.first;
var last = exampleData.last;
var favHobby = exampleData.hobbies[0];
var secondFavHobby = exampleData.hobbies[1];

// With destructuring
const { first, last, hobbies } = exampleData;
const [ favHobby, secondFavHobby ] = hobbies;
```

Spread Operator

```
// Copy objects:
const user = { name: 'alanv', role: 'Developer' };
const newUser = { ...user, role: 'Admin' };

// Merge objects
const person = { first: 'Alan', last: 'Vezina' };
const hobbies = { hobbies: [ 'Skateboarding', 'Painting', 'Cycling'] };
const completeObject = { ...person, ...hobbies };

// Merge arrays
const arrayOne = [1, 2, 3];
const arrayTwo = [5, 6, 7];
const combined = [...arrayOne, 4, ...arrayTwo];

// Pass all contents of object as props
<UserRenderer name={user.name} role={user.role} />
<UserRenderer {...user} />
```

Arrow Functions

```
// Old style
function fullName(user) {
   return `${user.first} ${user.last}`;
}

// Arrow function, with implicit return
const fullName = (user) => `${user.first} ${user.last}`;

// Arrow function, implicit object return. Note: spread operator makes this immutable!
const addFullName = (user) => ({...user, full: `${user.first} ${user.last}`});

// Arrow function, explicit return
const fullName = (user) => {
   const { first, last } = user;
   return `${first} ${last}`;
}
```

Classes

```
class Rectangle {
    constructor(height, width) {
      this.height = height;
      this.width = width;
    }
    // Method
    calcArea() {
      return this.height * this.width;
    }
    // Getter
    get area() {
        return this.calcArea();
    }
 }

const square = new Rectangle(10, 10);
console.log(square.area); // 100
```

Module System

```
// Old way:
<script src="/path/to/myLibrary.js" />

<script>
    // Assume library is on the page
    myLibrary.doTheThing();
</script>
```

```
// New way:
import { doTheThing } from 'my-library';

doTheThing();
```

JSX

- JavaScript with HTML embedded into it
- Custom tags supported (via components)
- Not natively supported, but used everywhere
  - transpiled via Webpack or other build systems

Components

- Functional
  - Stateless
  - Pure
- Classes
  - Stateful, or stateless
  - Lifecycle methods give you more control

Function Component

```
function RectangleStats(props) {
    const { rectangle } = props;
    const { width, height, area } = rectangle;

    return (
        <div className="rectangle-stats">
            <div>Width: {width}</div>
            <div>Height: {height}</div>
            <div>Area: {area}</div>
        </div>
    );
}

// In a block of JSX:
<RectangleStats rectangle={myRectangle} />
```

Class Component

```
class Timer extends React.Component {
   constructor(props) {
     super(props);
     this.state = { seconds: 0 };
   }
   tick() {
     this.setState(state => ({ seconds: state.seconds + 1 }));
   };
   componentDidMount() {
     this.interval = setInterval(() => this.tick(), 1000);
   }
   componentWillUnmount() {
     clearInterval(this.interval);
   }
   render() {
     return <div>Seconds: {this.state.seconds}</div>;
   }
 }
```

State Management

- Define and initialize state in your constructor
- State is updated via setState
  - Object form: this.setState({count: 1})
  - Callback form: this.setState((state) => {...state, count: state.count + 1})
  - Prefer callback form
  - Calls to setState are batched
- Don’t directly mutate data
  - Copy data first, then make changes to the copy
  - Spread operator is your friend

State Management cont.

- Keep state in an easy to manipulate format
  - Prefer flat structures to deeply nested ones
- Separate state changes and event handlers
  - Event handlers should call state change methods
  - Makes for easier testing

Stateful Components

```
class Input extends React.Component {
 constructor(props) {
   super(props);
   this.state = { value: "" };
 }

 onChange = event => {
   event.stopPropagation();
   event.preventDefault();
   const value = event.target.value;
   this.setState({ value });
 };

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

Stateful Components - better

```
class Input extends React.Component {
 constructor(props) {
   super(props);
   this.state = { value: "" };
 }

 updateValue = value => this.setState({ value });

 onChange = event => {
   event.stopPropagation();
   event.preventDefault();
   const value = event.target.value;
   this.updateValue(value);
 };

 render() {
   const { value } = this.state;
   return <input type="text" onChange={this.onChange} value={value} />;
 }
}
```

State Management cont.

- Start loading state in componentDidMount
  - Don’t do it in the constructor
  - loading flag, error string, data
- Split components into stateful/stateless
  - stateful component knows how to fetch and manipulate data
  - stateless component knows how to render the data

Stateless Components - rendering data

```
class ToDoList extends React.Component {
 render() {
   if (this.props.loading === true) {
     return <div className="loading">Loading...</div>;
   }

   return (
     <ul>
       {this.props.todos.map(todo => (
         <li key={todo.id}>{todo.text}</li>
       ))}
     </ul>
   );
 }
}
```

Stateful Components - fetching data

```
class StateFulToDoList extends React.Component {
 constructor(props) {
   super(props);
   this.state = { todos: null, loading: false };
 }

 loadData = () => {
   this.setState({ loading: true });
   api.loadData().then(data => this.setState({ loading: false, todos: data }));
 };

 componentDidMount() {
   this.loadData();
 }

 render() {
   return <ToDoList todos={this.state.todos} loading={this.state.loading} />;
 }
}
```

Reconciliation

- “Optimizing Performance”
  - https://reactjs.org/docs/optimizing-performance.html
- “Reconciliation”
  - https://reactjs.org/docs/reconciliation.html
- PureComponent is your best friend
  - Use it by default
  - Implement shouldComponentUpdate if all else fails

Reconciliation

```
class RectangleForm extends React.Component {
 constructor(props) {
   super(props);
   this.state = { width: "", height: "" };
 }

 onChange = (name, value) => this.setState({ [name]: value });

 render() {
   const { width, height } = this.state;
   const { onCancel, onSave } = this.props;
   return (
     <Form className="sample-form" onCancel={onCancel} onSave={onSave}>
       <Input label="Width" value={width} onChange={this.onChange} />
       <Input label="Height" value={height} onChange={this.onChange} />
     </Form>
   );
 }
}
```

Example LabKey Module

https://github.com/LabKey/tutorialModules/tree/develop/demo

Conclusion

- React is powerful, but deceptively simple
- Be careful with State Management
  - Bubble state changes upward
  - Keep state changes immutable
  - Use stateless components wrapped in stateful components
- Use reconciliation to your advantage
  - Use PureComponent by default
- www.labkey.org/reactJS.url

Further Learning

- TypeScript
  - JavaScript, but with types
- React Router
  - Needed for single page apps
- Redux
  - State management for single page applications
- Immutable JS
  - Easier immutable transformations
- Jest / Enzyme
  - Unit tests for your components

Further Learning

- ESLint
  - Automatically alerts you of questionable code
  - Can automatically fix some types of issues
- Prettier
  - Formats your code for you, so you never need to think about that again
  - A good idea, even if you don’t agree with all of the formatting choices