Data Entry

What are we going to cover

Creating data entry forms

Uncontrolled Components

Controlled Components

Validating input data

Creating data entry forms

React uses a **unidirectional data flow**

- Updates to data models are done very explicitly

Results in a better and more **predictable** application

- At the expense of slightly more code

React forms

React form are based on **standard HTML elements**

- <form> <input> <textarea> etc

<form> elements raise onChange, onInput and onSubmit **events**

- The React onChange event behaves different HTML element from because it **fires every change**

Interactive components

- Raise an onChange event when changes are made
- Can either controlled or uncontrolled

Note: <textarea> elements use the **value** property instead of their children

Edit form
Render function

```
class MovieEditForm extends Component {
  // ...
  render() {
    var { movie } = this.state;

    return (
      <form onSubmit={this.onSubmit}>
        <InputText onChange={this.onChange} name="title" value={movie.title}>
          Title
        </InputText>
        <div className="btn-group">
          <button type="submit" className="btn btn-primary">
            Save
          </button>
        </div>
      </form>
    );
  }
}
```

Edit form Remaining

```
class MovieEditForm extends Component {
  state = {
    movie: {
      id: 278,
      title: "The Shawshank Redemption",
    }
  };
  onChange = (name, value) => {
    const movie = { ...this.state.movie, [name]: value };
    this.setState({ movie });
  };
  onSubmit = e => {
    e.preventDefault()
    const { movie } = this.state;
    alert(`Saving: => ${JSON.stringify(movie)}`)
  };
  render() {
    // ...
  }
}
```

Uncontrolled Components

An **uncontrolled component** is an input element **without** a **value** property

- The value is not controlled by the application

Uncontrolled components maintain their own state internally

- Makes them easy to work with

Use the **onChange** event to track changes

- Or use the underlying DOM element value

Initially rendered without a value

- A default value can be supplied using the **defaultValue** property

The value can be updated by manipulating the underlying DOM element

- Use the **ref** callback to retrieve it

Uncontrolled Components

Uncontrolled components are often considered an anti pattern

- But they are not deprecated for a good reason

Can be more performant than controlled components

- Because there is less state management and rendering

An Uncontrolled Component

Note: No value is set in the render() and the value is only retrieved when it is used

```
class SearchForm extends Component {
  input = null;
  onSearch = () => {
    this.props.onSearch(this.input.value);
  };
  render() {
    return (
      <div className="input-group">
        <input type="text" className="form-control" ref={el => (this.input = el)}
        />
        <span className="input-group-btn">
          <button className="btn btn-secondary" onClick={this.onSearch}>
            Go!
          </button>
        </span>
      </div>
    );
  }
}
```

Controlled Components

A **controlled component** is an input element **with** a **value** property

- Only the application controls the value being rendered
- Always render this value
- Always add an **onChange** handler to be notified of changes

The **value** needs to be stored in **state** somewhere

- Either in the component itself, a parent component or externally using Redux or similar

The state is updated using the **onChange** handler

Controlled Components

Controlled components are considered the **best approach**

- But they depend on setting the state after each character typed
- Can cause a lot of reducer code and rendering to run

Can be less performant than uncontrolled components

- Because there is more state management and rendering

Consider using immutable principals with the **PureComponent**

- Specially with large forms or complex edit controls

A Controlled Component

Note: The value is being set and every change updates the state

```
class InputText extends Component {
  onChange = e => this.props.onChange(this.props.name, e.target.value);
  render() {
    const { name, value } = this.props;
    return (
      <div className="form-group">
        <label htmlFor={name + "Input"}>{this.props.children}</label>
        <input
          id={name + "Input"}
          type="text"
          className="form-control"
          value={value}
          onChange={this.onChange}
        />
      </div>
    );
  }
}
```

Validating input data

React has no in-built mechanism for **data validation**

- Use custom code or libraries like moment.js or validator.js to validate data

If validation is simple and fast it can be done in the **render function** or with each **update**

- Validity state is always up to date at the price of performance
- If validation takes too long using a debounce can help

Often a View Model is used with both the data and the result of the validation

- Stored the result in state

A simple validation function

```
exprort default {
  validate(name, value) {
    var errors = [];
    value = value || "";

    switch (name) {
      case "title":
        if (value.trim().length === 0) {
          errors.push("The title is required");
        }
        break;
    }

    return errors;
  }
};
```

Validating the movie after each change

```
onChange = (name, value) => {
  const movie = { ...this.state.movie, [name]: value };
  const errors = {
    ...this.state.errors,
    [name]: validator.validate(name, value)
  };

  this.setState({ movie, errors });
};
```

Useful libraries

Building data entry forms and validation in React requires quite a bit of boilerplate code

- Can easily be abstracted away

Do it yourself or use one the popular libraries as **Formik**

- Or alternatives like react-final-form or redux-form

With **Yup** or similar for form validation

Create a form using Formik

```
class MovieEditor extends Component {
  render() {
    const { isSubmitting, isValid } = this.props;

    return (<Form>
        <div className="form-group">
          <label>Title</label>
          <Field name="title" className="form-control" />
          <ErrorMessage name="title" component="div" className="alert alert-warning"
          />
        </div>
        <div className="btn-group">
          <button type="submit" className="btn btn-primary" disabled={isSubmitting || !isValid}>
            Save
          </button>
        </div>
      </Form>);
}}
```

Create a validation schema using Yup

```
const movieSchema = yup.object().shape({
  title: yup.string().required(),
  synopsis: yup
    .string()
    .required()
    .min(2)
});
```

Wrap the form and validation in a Formik component

```
export default class MovieEditorContainer extends Component {
  onSubmit = (values, actions) => {
    setTimeout(() => {
      alert(JSON.stringify(values));
      actions.setSubmitting(false);
    }, 1000);};
  render() {
    const { movie } = this.props;

    return (<Formik
      initialValues={{ title: movie.title, synopsis: movie.synopsis}}
      validationSchema={movieSchema}
      onSubmit={this.onSubmit}
      render={props => (
        <MovieEditor isSubmitting={props.isSubmitting} isValid={props.isValid} />)}
      />);
  }}
```

Conclusion

Use unidirectional data flow

- When creating data entry forms

Prefer controlled components over uncontrolled components

- Use immutable objects where appropriate

React is very flexible about validating input data

- Use the strategy you prefer