Framework Training
React
London July 2018
Exercise H-form

- This exercise builds a form with validation in React.

Installation

- Install and run the starter version of the project.

```
npm install
npm start
```

Review the existing project

- The Form component defines a form without any validation.
- We want the state of the form to reflect the component state.
- The first time the form is displayed, it should use values from the component state.
- When the user types into a field, an event handler should capture this value and update component state.
- When component state changes, the render method is called.
- It should update the form to reflect state.
- This approach to keeping form and component state in sync is called a **Controlled Component**.

Validation

- The component will apply **validation** to the form.
- The city field will allow **two or more letters, but no digits**.
- The passport field will only allow exactly **eight digits**.
- **Warning feedback** will be displayed adjacent to each field.
- Field contents will be **styled red** when a field is incorrect.
- **Regular expressions** will be used to style the field.

Form state

- Define state in the constructor for the two form fields.

```
    constructor( props ) {
```

```
    super( props );
    this.state = { city:"", passport:"" };
    }
```

- Listen for change events on the city field as the user types.

```
    <input type="text" name="city"
    onChange={this.changeField} />

    changeField = e => console.log(e);
```

- React passes a **synthetic event** to the function changeField.
- The expression **e.target** points at the form field.
- We can use this to update state for the city field.

```
    changeField = e => {
        let el = e.target;
        this.setState( { city:el.value })
    }
```

- Use React DevTools to confirm that this changes state.

More generic event handlers

- This approach works but we want to avoid writing separate event handlers for each field.
- Expression e.target.name contains the name of each form field.
- We can refactor the event handler to use this expression.

```
    this.setState( { [el.name]:el.value })
```

- We can then call the same method from the passport field.

```
    <input type="text" name="passport"
     onChange={this.changeField} />
```

- Use React DevTools to confirm that this changes state for both fields.

Form values

- We also want to ensure that form fields and state remain in sync, and we may want to initially set form values in the constructor.
- Set the form field values equal to their state.

```
.... value={city}
.... value={passport}
```

Validation

- We will use regular expressions to validate the fields.
- This expression will return true if the city contains two or more letters and no digits.

```
/^[a-zA-Z]{2,}$/.test( city )
```

- This expression returns true if the passport is exactly eight digits.

```
/^[0-9]{8}$/.test( passport )
```

- This validate function creates an object that contains the current validation state of the form.

```
    validate = () => {

        let {city,passport} = this.state;

        return {
            city : {
                test : /^[a-zA-Z]{2,}$/.test( city )
            } ,
            passport: {
                test : /^[0-9]{8}$/.test( passport )
            }
        }
    }
```

- Call the function in the render method.

```
    let valid = this.validate();
    console.log(valid);
```

- Once the form is valid, this object will contain:

```
{ city:{ valid:true }, passport: { valid:true }}
```

Conditional styling

- We can conditionally style form fields using the valid object.

```
className={ valid.city.test ? null : "form-error" }
className={ valid.passport.test ? null : "form-error" }
```

Error spans

- Adjacent to each field is a span containing an error warning message if the field is incorrect.
- We can add warning messages to our validate method.

```
    city : {
    test : /^[a-zA-Z]{2,}$/.test( city ),
    warn : "Only letters"
} ,
passport: {
    test : /^[0-9]{8}$/.test( passport ),
    warn : "8 digits"
}
```

- We can use conditional styling to hide/reveal these messages based on the state of the form.

```
    <span className=
    { valid.city.test ? null : "form-warning" }>{
valid.city.warn }</span>

    <span className=
    { valid.passport.test ? null : "form-warning" }>{
valid.passport.warn }</span>
```

- Test the behaviour of the span messages.

Submit the form

- We need an event handler that is called when the user submits the form.
- The default behaviour of web forms is to refresh the web page on submit. The first task is to turn off this behaviour.

```
<form id="holiday" onSubmit={this.buyHoliday}>

buyHoliday = (e) => {
    e.preventDefault();
    console.log( this.state );
}
```

- This works but the submit button is active all the time, even when the form contains invalid fields.
- We need a boolean function which returns true if every field is valid.

```
  isValid = () => {
    let valid = this.validate();
```

```
    return Object.keys(valid).every( j => valid[j].test );
}
```

- We can use this function to control the submit button state.

```
disabled={!this.isValid()}
```

- Once the form has been submitted, we want to clear the form.

```
this.setState( { city:"", passport:"" } );
```

- We have now created a Controlled component which manages the state of a form.