Interactions with Forms and Events

This lesson will explain how to add search functionality by introducing forms and events in React.

We'll add another interaction to see forms and events in React, a search functionality where the input the search field temporarily filters a list based on the title property of an item.

In the first step, we define a form with an input field in JSX:

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
        <form>
          <input type="text" />
        </form>
        {this.state.list.map(item =>
          ...
        )}
      </div>
    );
  }
}
```

In the following scenario you will type into the input field and filter the list temporarily by the search term that is used in the input field. To filter the list based on the value of the input field, we store the value of the input field in the local state. We use **synthetic events** in React to access a value in an event payload.

Let's define a `onChange` handler for the input field:

```
class App extends Component {

  ...

  render() {
    return (
```

```
      <div className="App">
        <form>
          <input

            type="text"
            onChange={this.onSearchChange}
          />
        </form>
        ...
      </div>
    );
  }
}
```

The function is bound to the component, so it is a class method again. You just need to bind and define the method:

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      list,
    };

    this.onSearchChange = this.onSearchChange.bind(this);
    this.onDismiss = this.onDismiss.bind(this);
  }

  onSearchChange() {
    ...
  }

  ...
}
```

When using a handler in your element, you get access to the synthetic React event in your callback function's signature.

```
class App extends Component {

  ...

  onSearchChange(event) {
    ...
  }

  ...
}
```

The event has the value of the input field in its target object, so you can update the local state with a search term using `this.setState()`.

```
class App extends Component {

  ...

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

  ...
}
```

Don't forget to define the initial state for the `searchTerm` property in the constructor. The input field should be empty in the beginning, so its value is an empty string.

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      list,
      searchTerm: '',
    };

    this.onSearchChange = this.onSearchChange.bind(this);
    this.onDismiss = this.onDismiss.bind(this);
  }

  ...
}
```

We store the input value to the local state every time the value in the input field changes.

We can assume that when we update `searchTerm` with `this.setState()`, the list also needs to be passed to preserve it. React's `this.setState()` is a shallow merge, however, so it preserves the sibling properties in the state object when it updates a property. The list state, though you have already dismissed an item from it, stays the same when updating the `searchTerm` property.

Returning to our application, we see the list isn't filtered yet, based on the input field value stored in the local state. We need to filter the list temporarily based on the `searchTerm`, and we have everything we need to perform this operation. In the `render()` method, before mapping over the list, we apply a

filter to it. The filter will only evaluate if the `searchTerm` matches the title property of the item. We've already used the built-in JavaScript filter functionality, so let's use it again to sneak in the filter function before the map function. The filter function returns a new array, so the map function can be used on it.

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
        <form>
          <input
            type="text"
            onChange={this.onSearchChange}
          />
        </form>
        {this.state.list.filter(...).map(item =>
          ...
        )}
      </div>
    );
  }
}
```

Let's approach the filter function in a different way this time. We want to define the filter argument, which is the function passed to the filter function outside the ES6 class component. We don't have access to the state of the component, so we have no access to the `searchTerm` property to evaluate the filter condition. This means we'll need to pass the `searchTerm` to the filter function, returning a new function to evaluate the condition. This is called a higher-order function.

It makes sense to know about higher-order functions, because React deals with a concept called higher-order components. You will get to know the concept later in the course. Now again, let's focus on the filter functionality.

First, you have to define the higher-order function outside of your App component.

```
function isSearched(searchTerm) {
  return function(item) {
    // some condition which returns true or false
  }
}
```

```
class App extends Component {


  ...

}
```

The function takes the `searchTerm` and returns another function, because the filter function only takes that type as its input. The returned function has access to the item object, because it is the one passed to the filter function.

It will also be used to filter the list based on the condition defined in the function, so let's define the condition:

```
function isSearched(searchTerm) {
  return function(item) {
    return item.title.toLowerCase().includes(searchTerm.toLowerCase());
  }
}

class App extends Component {

  ...

}
```

The condition matches the incoming `searchTerm` pattern with the title property of the item from your list. You can do that with the built-in `includes` JavaScript functionality. When the pattern matches, it returns true and the item stays in the list; when the pattern doesn't match, the item is removed from the list. Don't forget to match the capitalizion on both strings to the letter, as there will be mismatches between the search term 'redux' and an item title 'Redux'. Since we are working on a immutable list and return a new list by using the filter function, the original list in the local state isn't modified at all.

We cheated a bit using JavaScript ES7 features, but these aren't present in ES5. For ES5, use the `indexOf()` function to get the index of the item in the list instead. When the item is in the list, `indexOf()` will return its index in the array.

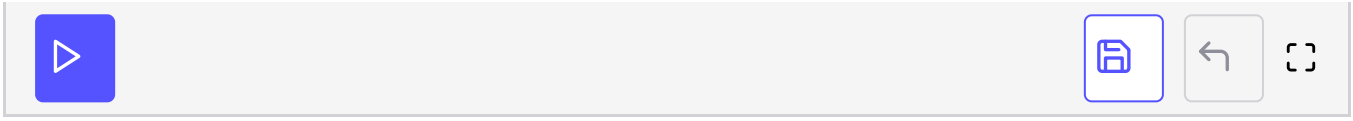
```
// comment either ES5 or ES6 code

var string = 'Robin';
var pattern = 'Rob';
```

```
// ES5
var result = string.indexOf(pattern) !== -1;


// ES6
// var result = string.includes(pattern);

console.log(result);
```



Another neat refactoring can be done with an ES6 arrow function again. It makes the function more concise:

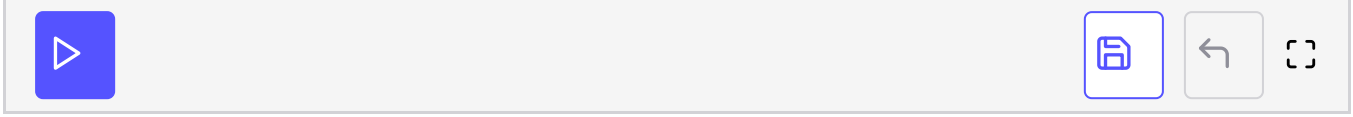
```
// comment either ES5 or ES6 code

// ES5
function isSearched(searchTerm) {
  return function(item) {
    return !searchTerm || item.title.toLowerCase().includes(searchTerm.toLowerCase());
  }
}

// ES6
// const isSearched = (searchTerm) => (item) =>
//   !searchTerm || item.title.toLowerCase().includes(searchTerm.toLowerCase());

var searchTerm = 'rea';
var item = { title: 'React' };

console.log(isSearched(searchTerm)(item));
```



The React ecosystem uses a lot of functional programming concepts, often using functions that return functions (the concept is called high-order functions) to pass information. JavaScript ES6 lets us express these even more concisely with arrow functions.

Last but not least, use the defined `isSearched()` function to filter lists. We pass it the `searchTerm` property from the local state, so that it returns the filter's input function and filters your list based on the filter condition. After that it maps over the filtered list to display an element for each list item.

```
class App extends Component {

  ...
```

```
  render() {
    return (

      <div className="App">
        <form>
          <input
            type="text"
            onChange={this.onSearchChange}
          />
        </form>
        {this.state.list.filter(isSearched(this.state.searchTerm)).map(item =>
          ...
        )}
      </div>
    );
  }
}
```

The search functionality should work now. Try it.

```
import React, { Component } from 'react';
require('./App.css');

const list = [
  {
    title: 'React',
    url: 'https://reactjs.org/',
    author: 'Jordan Walke',
    num_comments: 3,
    points: 4,
    objectID: 0,
  },
  {
    title: 'Redux',
    url: 'https://redux.js.org/',
    author: 'Dan Abramov, Andrew Clark',
    num_comments: 2,
    points: 5,
    objectID: 1,
  },
];

const isSearched = (searchTerm) => (item) =>
  item.title.toLowerCase().includes(searchTerm.toLowerCase());

class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      list,
      searchTerm: '',
    };

    this.onSearchChange = this.onSearchChange.bind(this);
    this.onDismiss = this.onDismiss.bind(this);
  }
```

```
  onSearchChange(event) {
    this.setState({ searchTerm: event.target.value });
  }

  onDismiss(id) {
    const isNotId = item => item.objectID !== id;
    const updatedList = this.state.list.filter(isNotId);
    this.setState({ list: updatedList });
  }

  render() {
    return (
      <div className="App">
        <form>
          <input
            type="text"
            onChange={this.onSearchChange}
          />
        </form>
        {this.state.list.filter(isSearched(this.state.searchTerm)).map(item =>
          <div key={item.objectID}>
            <span>
              <a href={item.url}>{item.title}</a>
            </span>
            <span>{item.author}</span>
            <span>{item.num_comments}</span>
            <span>{item.points}</span>
            <span>
              <button
                onClick={() => this.onDismiss(item.objectID)}
                type="button"
              >
                Dismiss
              </button>
            </span>
          </div>
        )}
      </div>
    );
  }
}

export default App;
```

Further Reading:

- Read more about React events
- Read more about higher-order functions