Lifting State

We have already covered the basics of state management in React in the previous chapters by using React's local state, so this chapter will dig a bit deeper. In this lesson, we'll learn what lifting state in React is and how it is used.

What does 'Lifting State' mean?

Only the App component is a stateful ES6 component in your application. It handles a lot of application state and logic in its class methods. Moreover, we pass a lot of properties to the Table component, most of which are only used in there. It's not important that the App component knows about them, so the sort functionality could be moved into the Table component.

Moving a sub-state from one component to another is known as lifting state. We want to move state that isn't used in the App component into the Table component, down from parent to child component. To deal with state and class methods in the Table component, it has to become an ES6 class component.

Refactoring the Table component

Refactoring from functional stateless components to ES6 class components is simple. Your Table component as a functional stateless component:

```
const Table = ({
  list,
  sortKey,
  isSortReverse,
  onSort,
  onDismiss
}) => {
  const sortedList = SORTS[sortKey](list);
  const reverseSortedList = isSortReverse
    ? sortedList.reverse()
    : sortedList;

  return(
    ...
  );
}
```

Your Table component as an ES6 class component:

```
class Table extends Component {

  render() {
    const {
      list,
      sortKey,
      isSortReverse,
      onSort,
      onDismiss
    } = this.props;

    const sortedList = SORTS[sortKey](list);
    const reverseSortedList = isSortReverse
      ? sortedList.reverse()
      : sortedList;

    return(
      ...
    );
  }
}
```

Since you want to deal with state and methods in your component, you have to add a constructor and initial state.

```
class Table extends Component {

  constructor(props) {
    super(props);

    this.state = {};
  }

  render() {
    ...
  }
}
```

Now you can move state and class methods regarding the sort functionality from your App component down to your Table component.

```
class Table extends Component {
  constructor(props) {
    super(props);

    this.state = {
      sortKey: 'NONE',
      isSortReverse: false,
    };

    this.onSort = this.onSort.bind(this);
  }

  onSort(sortKey) {
    const isSortReverse = this.state.sortKey === sortKey &&
      !this.state.isSortReverse;

    this.setState({ sortKey, isSortReverse });
  }

  render() {
    ...
  }
}
```

Don’t forget to remove the moved state and `onSort()` class method from your App component.

```
class App extends Component {
  _isMounted = false;

  constructor(props) {
    super(props);

    this.state = {
      results: null,
      searchKey: '',
      searchTerm: DEFAULT_QUERY,
      error: null,
      isLoading: false,
    };

    this.setSearchTopStories = this.setSearchTopStories.bind(this);
    this.fetchSearchTopStories = this.fetchSearchTopStories.bind(this);
    this.onDismiss = this.onDismiss.bind(this);
    this.onSearchSubmit = this.onSearchSubmit.bind(this);
    this.onSearchChange = this.onSearchChange.bind(this);
    this.needsToSearchTopStories = this.needsToSearchTopStories.bind(this);
  }

  ...

}
```

Making the Table component more lightweight

To do this, we move props that are passed to it from the App component, because they are handled internally in the Table component.

```
class App extends Component {

  ...

  render() {
    const {
      searchTerm,
      results,
      searchKey,

      error,
      isLoading
    } = this.state;

    ...

    return (
      <div className="page">
        ...
        { error
          ? <div className="interactions">
            <p>Something went wrong.</p>
          </div>
          : <Table
            list={list}
            onDismiss={this.onDismiss}
          />
        }
        ...
      </div>
    );
  }
}
```

Now in your Table component you can use the internal `onSort()` method and the internal Table state.

```
class Table extends Component {

  ...

  render() {
    const {
      list,
      onDismiss
    } = this.props;

    const {
      sortKey,
      isSortReverse,
    } = this.state;

    const sortedList = SORTS[sortKey](list);
    const reverseSortedList = isSortReverse
      ? sortedList.reverse()
      : sortedList;

    return(
      <div className="table">
        <div className="table-header">
          <span style={{ width: '40%' }}>
            <Sort
              sortKey={'TITLE'}
              onSort={this.onSort}
              activeSortKey={sortKey}
            >
              Title
            </Sort>

          </span>
          <span style={{ width: '30%' }}>
            <Sort
              sortKey={'AUTHOR'}
              onSort={this.onSort}
              activeSortKey={sortKey}
            >
              Author
            </Sort>
          </span>
          <span style={{ width: '10%' }}>
            <Sort
              sortKey={'COMMENTS'}
              onSort={this.onSort}
              activeSortKey={sortKey}
            >
              Comments
            </Sort>
          </span>
          <span style={{ width: '10%' }}>
            <Sort
              sortKey={'POINTS'}
              onSort={this.onSort}
              activeSortKey={sortKey}
            >
              Points
            </Sort>
          </span>
          <span style={{ width: '10%' }}>
            Archive
          </span>
        </div>
        { reverseSortedList.map((item) =>
          ...
        )}
      </div>
    );
  }
}
```

We made a crucial refactoring by moving functionality and state closer into another component, and other components got more lightweight. Again, the component API of the Table got lighter because it deals internally with the sort functionality.

Lifting state can go the other way as well: from child to parent component. It is called as lifting state up. Imagine you were dealing with local state in a child component, and you want to fulfil a requirement to show the state in your parent component as well. You would have to lift up the state to your parent component. Moreover, imagine you want to show the state in a sibling component of your child component. Again, you would lift the state up to your parent component. The parent component deals with the local state, but exposes it to both child components.

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

const DEFAULT_QUERY = 'redux';
const DEFAULT_HPP = '100';

const PATH_BASE = 'https://hn.algolia.com/api/v1';
const PATH_SEARCH = '/search';
const PARAM_SEARCH = 'query=';
const PARAM_PAGE = 'page=';

const SORTS = {
  NONE: list => list,
  TITLE: list => sortBy(list, 'title'),
  AUTHOR: list => sortBy(list, 'author'),
  COMMENTS: list => sortBy(list, 'num_comments').reverse(),
  POINTS: list => sortBy(list, 'points').reverse(),
};

class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      results: null,
      searchKey: '',
      searchTerm: DEFAULT_QUERY,
      error: null,
      isLoading: false,
    };

    this.needsToSearchTopstories = this.needsToSearchTopstories.bind(this);
    this.setSearchTopstories = this.setSearchTopstories.bind(this);
    this.fetchSearchTopstories = this.fetchSearchTopstories.bind(this);
    this.onSearchChange = this.onSearchChange.bind(this);
    this.onSearchSubmit = this.onSearchSubmit.bind(this);
    this.onDismiss = this.onDismiss.bind(this);
  }

  needsToSearchTopstories(searchTerm) {
    return !this.state.results[searchTerm];
  }

  setSearchTopstories(result) {
    const { hits, page } = result;
    const { searchKey, results } = this.state;

    const oldHits = results && results[searchKey]
      ? results[searchKey].hits
      : [];

    const updatedHits = [
      ...oldHits,
      ...hits
    ];
    this.setState({
      results: {

        ...results,
        [searchKey]: { hits: updatedHits, page }
      },
      isLoading: false
    });
  }

  fetchSearchTopstories(searchTerm, page = 0) {
    this.setState({ isLoading: true });

    fetch(`${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}&${PARAM_PAGE}${page}`)
      .then(response => response.json())
      .then(result => this.setSearchTopstories(result))
        .catch(e => this.setState({ error: e }));
  }

  componentDidMount() {
    const { searchTerm } = this.state;
    this.setState({ searchKey: searchTerm });
    this.fetchSearchTopstories(searchTerm);
  }

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

  onSearchSubmit(event) {
    const { searchTerm } = this.state;
    this.setState({ searchKey: searchTerm });

    if (this.needsToSearchTopstories(searchTerm)) {
      this.fetchSearchTopstories(searchTerm);
    }

    event.preventDefault();
  }

  onDismiss(id) {
    const { searchKey, results } = this.state;
    const { hits, page } = results[searchKey];

    const isNotId = item => item.objectID !== id;
    const updatedHits = hits.filter(isNotId);

    this.setState({
      results: {
        ...results,
        [searchKey]: { hits: updatedHits, page }
      }
    });
  }

  render() {
    const {
      searchTerm,
      results,
      searchKey,
      error,
      isLoading
    } = this.state;

    const page = (
      results &&
      results[searchKey] &&
      results[searchKey].page
    ) || 0;

    const list = (
      results &&
      results[searchKey] &&
      results[searchKey].hits
    ) || [];

    return (
      <div className="page">
        <div className="interactions">
          <Search
            value={searchTerm}
            onChange={this.onSearchChange}
            onSubmit={this.onSearchSubmit}
          >
            Search
          </Search>
        </div>
        { error
          ? <div className="interactions">
            <p>Something went wrong.</p>
          </div>
          : <Table
                  list={list}
                  onDismiss={this.onDismiss}
                />
        }
        <div className="interactions">
          <ButtonWithLoading
            isLoading={isLoading}
            onClick={() => this.fetchSearchTopstories(searchKey, page + 1)}>
            More
          </ButtonWithLoading>
        </div>
      </div>
    );
  }
}

const Search = ({
  value,
  onChange,
  onSubmit,
  children
}) =>
  <form onSubmit={onSubmit}>
    <input
      type="text"
      value={value}
      onChange={onChange}
    />
    <button type="submit">
      {children}
    </button>
  </form>
```

```
class Table extends Component {

  constructor(props) {
    super(props);

    this.state = {
      sortKey: 'NONE',
      isSortReverse: false,
    };

    this.onSort = this.onSort.bind(this);
  }

  onSort(sortKey) {
    const isSortReverse = this.state.sortKey === sortKey && !this.state.isSortReverse;
    this.setState({ sortKey, isSortReverse });
  }

  render() {
    const {
      list,
      onDismiss
    } = this.props;

    const {
      sortKey,
      isSortReverse,
    } = this.state;

    const sortedList = SORTS[sortKey](list);
    const reverseSortedList = isSortReverse
      ? sortedList.reverse()
      : sortedList;

    return(
      <div className="table">
        <div className="table-header">
          <span style={{ width: '40%' }}>
            <Sort
              sortKey={'TITLE'}
              onSort={this.onSort}
              activeSortKey={sortKey}
            >
              Title
            </Sort>
          </span>
          <span style={{ width: '30%' }}>
            <Sort
              sortKey={'AUTHOR'}
              onSort={this.onSort}
              activeSortKey={sortKey}
            >
              Author
            </Sort>
          </span>
          <span style={{ width: '10%' }}>
            <Sort
              sortKey={'COMMENTS'}
              onSort={this.onSort}
              activeSortKey={sortKey}
            >
```

```
              Comments
            </Sort>
          </span>
          <span style={{ width: '10%' }}>
            <Sort
              sortKey={'POINTS'}
              onSort={this.onSort}
              activeSortKey={sortKey}
            >
              Points
            </Sort>
          </span>
          <span style={{ width: '10%' }}>
            Archive
          </span>
        </div>
        { reverseSortedList.map(item =>
          <div key={item.objectID} className="table-row">
            <span style={{ width: '40%' }}>
              <a href={item.url}>{item.title}</a>
            </span>
            <span style={{ width: '30%' }}>
              {item.author}
            </span>
            <span style={{ width: '10%' }}>
              {item.num_comments}
            </span>
            <span style={{ width: '10%' }}>
              {item.points}
            </span>
            <span style={{ width: '10%' }}>
              <Button
                onClick={() => onDismiss(item.objectID)}
                className="button-inline"
              >
                Dismiss
              </Button>
            </span>
          </div>
        )}
      </div>
    );
  }
}

const Button = ({ onClick, className = '', children }) =>
  <button
    onClick={onClick}
    className={className}
    type="button"
  >
    {children}
  </button>

const Loading = () =>
  <div>Loading ...</div>

const withLoading = (Component) => ({ isLoading, ...rest }) =>
  isLoading ? <Loading /> : <Component { ...rest } />

const ButtonWithLoading = withLoading(Button);
const Sort = ({
  sortKey,
  activeSortKey,
  onSort,
  children
}) => {
  const sortClass = classNames(
    'button-inline',
    { 'button-active': sortKey === activeSortKey }
  );

  return (
    <Button
      onClick={() => onSort(sortKey)}
      className={sortClass}
    >
      {children}
    </Button>
  );
}

export default App;

export {
  Button,
  Search,
  Table,
};
```

Exercises:

- read more about lifting state in React
- read more about lifting state in learn React before using Redux