Revisited: setState()

In this chapter we revisit the React setState() method. We learn that setState() is asynchronous and not only takes objects as arguments but also takes functions.

`setState()` can take functions as input

So far, we have used React `setState()` to manage your internal component state. We can pass an object to the function where it updates partially the local state.

```
this.setState({ value: 'hello'});
```

But `setState()` doesn't take only an object. In its second version, you can pass a function to update the state.

```
this.setState((prevState, props) => {
  ...
});
```

Use cases of functions as input

There is one crucial case where it makes sense to use a function over an object: when you update the state depending on the previous state or props. If you don't use a function, the local state management can cause bugs. The React `setState()` method is asynchronous. React batches `setState()` calls and executes them eventually. Sometimes, the previous state or props changes between before we can rely on it in our `setState()` call.

```
const { oneCount } = this.state;
const { anotherCount } = this.props;
this.setState({ count: oneCount + anotherCount });
```

Imagine that `oneCount` and `anotherCount`, thus the state or the props, change somewhere else asynchronously when you call `setState()`. In a growing application, you have more than one `setState()` call across your application.

Since `setState()` executes asynchronously, you could rely in the example on stale values.

With the function approach, the function in `setState()` is a callback that operates on the state and props at the time of executing the callback function. Even though `setState()` is asynchronous, with a function it takes the state and props at the time when it is executed.

```
this.setState((prevState, props) => {
  const { oneCount } = prevState;
  const { anotherCount } = props;
  return { count: oneCount + anotherCount };
});
```

In our code, the `setSearchTopStories()` method relies on the previous state, and this is a good example to use a function over an object in `setState()`. Right now, it looks like the following code:

```
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
  });
}
```

Here, we extracted values from the state, but updated the state depending on the previous state asynchronously. Now we'll use the functional approach to prevent bugs from a stale state:

```
setSearchTopStories(result) {
  const { hits, page } = result;

  this.setState(prevState => {

    ...
  });
}
```

We can move the whole block we implemented into the function by directing it to operate on the `prevState` instead of the `this.state`.

```
setSearchTopStories(result) {
  const { hits, page } = result;

  this.setState(prevState => {
    const { searchKey, results } = prevState;

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

    const updatedHits = [
      ...oldHits,
      ...hits
    ];

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

That will fix the issue with a stale state, but there is still one more improvement. Since it is a function, you can extract the function for improved readability. One more advantage to use a function over an object is that function can live outside of the component. We still have to use a higher-order function to pass the result to it since we want to update the state based on the fetched result from the API.

```
setSearchTopStories(result) {
  const { hits, page } = result;
  this.setState(updateSearchTopStoriesState(hits, page));
}
```

The `updateSearchTopStoriesState()` function has to return a function. It is a higher-order function that can be defined outside the App component. Note how the function signature changes slightly now.

```
const updateSearchTopStoriesState = (hits, page) => (prevState) => {
  const { searchKey, results } = prevState;

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

  const updatedHits = [
    ...oldHits,
    ...hits
  ];

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

class App extends Component {
  ...
}
```

The function instead of object approach in `setState()` fixes potential bugs, while increasing the readability and maintainability of your code. Further, it becomes testable outside of the App component. I advise exporting and testing it as practice.

Exercises:

- Export `updateSearchTopStoriesState` from the file. Write a test for it which passes the a payload (hits, page) and a made up previous state and finally expect a new state
- Refactor your `setState()` methods to use a function, but only when it makes sense, because it relies on props or state
- Run your tests again and verify that everything is up to date

Further Reading

- read more about React using state correctly

Quick quiz on setState()!

1

`setState()` runs synchronously.

COMPLETED 0%

1 of 2