Introduction to ImmutableJS

Learn how to enhance your application with immutability to increase the performance and avoid mutation bugs!

As we discovered in Chapter 4: Redux, immutability is quite helpful when developing applications! It makes it so much easier to reason about what is happening to your data, as nothing can be mutated from somewhere entirely different.

The problem is that JavaScript is by default a mutable language. Other developers that don't have this intricate knowledge of immutability might still mess up, mutate the state and break our app in unexpected ways.

Facebook released a second library called `Immutable.js` that adds immutable data structures to JavaScript! Let's see what this looks like.

ImmutableJS exports this nice little `fromJS` function that allows us to create immutable data structures from your standard JavaScript objects and arrays. (it also adds a `toJS` method to make objects and arrays out of them again) Let's create an immutable object:

```
import { fromJS } from 'immutable';

var immutableObject = fromJS({
    some: 'object'
});
```

If you now tried to do `object.some = 'notobject'`, i.e. tried to change the data inside this object, you won't be able to see those changes as those changes would not be applied on the object. That's the power of immutable data structures, you know exactly what they are.

```
import { fromJS } from 'immutable';

let immutableObject = fromJS({
        some: 'object'
});

immutableObject.some = 'notobject';


console.log(immutableObject); // Doesn't change
```

Now you might be thinking “But then how can we set a property?”. Well, ImmutableJS still let’s us set properties with the `set` and `setIn` methods! Let’s take a look at an example:

```
import { fromJS } from 'immutable';

var immutableObject = fromJS({
        some: 'object'
});

immutableObject.set('some', 'notobject');
console.log(immutableObject);
```

If you run the above code, you’ll still get our initial object again. Why?

Well, since `immutableObject` is immutable, what happens when you `immutableObject.set` is **that a new immutable object is returned with the changes**. No mutation happening, this is kind of like what we did with `Object.assign` for our reducers!

Let’s see if that works:

```
import { fromJS } from 'immutable';

var immutableObject = fromJS({
        some: 'object'
});

newObject = immutableObject.set('some', 'notobject');
console.log(newObject);
```

The changes are there, awesome! `immutableObject` on the other hand still is our old `{ some: 'object' }` without changes.

As I mentioned before, this is kind of what we did in our redux reducer right? So what would happen if we used ImmutableJS there? Let’s try it!