Proxies & Reflect

WE'LL COVER THE FOLLOWING

A Proxy in ES6 is a special type of object that allows us to intercept operations preformed on an Object. Operations like property lookups, assignments and more. Proxies all us to set up a `target` object with a `handler`. The `handler` object is used to override the operations on our target object. Let's first look at the syntax for our Proxy.

```
new Proxy(target,handler);
```

With this in mind, let's create a `person` object with two simple properties on it.

```
const person = new Proxy({
    name: 'Ryan',
    age: 31
},{
    //Handler goes here.
})
```

On this `person` object, we have only two properties `name` and `age`. The `handler` object is used to list 'traps', these trap methods include `get()`, `set()`, and much more. As an example, lets set up a Proxy that has a `get` trap.

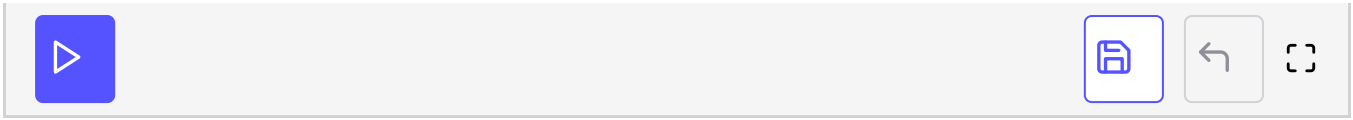
```
const person = new Proxy({
  name: 'Ryan',
  age: 31
}, {
```

```
  get(target, prop) {
    return target[prop]


  }
});
```

Here we implement a `get` trap on our handler, notice that the method gets passed two arguments, `target`, and `prop`. `target` is the object we are trying to access the property from. And `prop` is what `key` we want to access. Calling `person.name` will trigger the `get` trap, as we are trying to access the `name` property. In our trap here we simply return the value, but image that we could alter the values.

Let's change this to return all uppercase strings from our object.

```
const person = new Proxy({
  name: 'Ryan',
  age: 31
}, {
  get(target, prop) {
    const val = target[prop];
    return typeof val === 'string' ? val.toUpperCase() : val;
  }
});

console.log(person.name); //RYAN
```



This opens up a lot of possibilities, because now we have the ability to intercept operations on our objects and validate or alter them. We have had the ability to do this in the past with the `get` and `set` keywords, remember we looked at `get` a little bit back in the chapter on classes. However with those we had to create a `get` and a `set` for each property, here we are able to do that with more programmatically as we are passed all the info we need.

We are just scratching the surface of Proxies, I will list of bunch of great resources at the end of the chapter so a more indepth look into them. But to finish, let's implement a little `Component` function that we can pass some data, and validate the data when we want to change it. Also using the `set` trap we will have it re-render our `Component` when data is `set`.

To start, we will use the `templater` function from chapter 7 to create our HTML

HTML.

```
//From chapter 7
function templater(strings, ...keys) {
    return function(data) {
        let temp = strings.slice();
        keys.forEach((key, i) => {
            temp[i] = temp[i] + data[key];
        });
        return temp.join('');
    }
}
```

Next we need to create our `Component` function, it will take an `options` object and return a slightly altered version of it. Ideally this is how we would like to call a new `Component`.

```
const app = Component({
  model:{
    num:'0'
  },
  template: templater`<h1>Clicked: ${'num'}</h1>`
});
```

The important bit here is the `model` property, this we will convert into a Proxy so we can intercept the `set` operation. Now we will look at the `Component` and talk about what is going on here.

```
function Component(options) {
  let defaults = {
    model: new Proxy(options.model,{
      set(target,prop,value) {
        target[prop] = value;
        defaults.render(target);
      }
    }),
    template: options.template,
    el: options.el || document.body,
    render(data) {
      this.el.innerHTML = this.template(data)
    }
  }
  defaults.render(options.model);
  return defaults;
}
```

First thing you notice is we create a `defaults` object that is used to set us up. We have a `model` key that we pass a `new Proxy` to, that takes the `model` passed from `options` and sets the `set` trap. This trap is provided three arguments,

the `target`, the `prop` and what `value` we want. In the trap we set the value, but also we call the `render` method of the `defaults` object.

If we look at `render` we can see it takes some data, our object, and used the `el` property(which is `body` by default) and renders our template on it. The really neat thing here is that if we create a new `Component` and then have an event listener that every time the `body` is clicked we update the value on `num`, it will re render our view!

```
const app = Component({
  model:{
    num:'0'
  },
  template: templater`<h1>Clicked: ${'num'}</h1>`
});

let counter = 0;

document.addEventListener('click',() => {
    counter++;
    app.model.num = counter;
});
```

Kind of fun right? I mean this is no React, but it gets you thinking about how intercepting object operations can help us create intuitive API's. This concept is called meta programming, and in the *Additional Resources* section I have added a link from Axel Rauschmayer on Reflection and Meta Programming for a further read. There are also many more traps we can add, to see the whole list, check out the MDN article listed below.

One step further with proxies

Let's take this idea one step further and implement type validation to our `model`. Suppose we want to make sure the user is always setting a certain type for data model, like a number for example.

```
const app = Component({
  model:{
    num: '`number` 0'
  },
  template: templater`<h1>Clicked: ${'num'}</h1>`
});
```

We can use the initial setup of our `Component` to create an object that can be used for validation.

```
const props = Object.assign({},options.model);
const model = {};
for(let key in props) {
 let type = /`(.*)`/gi.exec(props[key])[1];
 let value = /^.*`\s(.*)/gi.exec(props[key])[1];
 if(type === 'number') value = parseFloat(value)
 props[key] = {
   type,
   value
 }
 options.model[key] = value
}
```

There is some regExp here that looks for anything in the `` and will use that as the type, and the it will set the value back on the options `model`. In our `set` trap we can add a condition that checks the `value` that comes in, if it is not right it will throw an error.

```
set(target,prop,value) {
  if(typeof value !== props[prop].type) {
    throw new Error(`Invalid type set on property ${prop}`);
  }
  target[prop] = value;
  defaults.render(target);
}
```

Below you can find the whole code.

```
function templater(strings, ...keys) {
    return function(data) {
        let temp = strings.slice();
        keys.forEach((key, i) => {
            temp[i] = temp[i] + data[key];
        });
        return temp.join('');
    }
}

function Component(options) {
  const props = Object.assign({},options.model);
  const model = {};
  for(let key in props) {
    let type = /`(.*)`/gi.exec(props[key])[1];
    let value = /^.*`\s(.*)/gi.exec(props[key])[1];
    if(type === 'number') value = parseFloat(value)
    props[key] = {
      type,
      value
    }
    options.model[key] = value
```

```
  }
  let defaults = {
    model: new Proxy(options.model,{

      set(target,prop,value) {
        if(typeof value !== props[prop].type) {
          throw new Error("Not the right type")
        }
        target[prop] = value;
        defaults.render(target);
      }
    }),
    template: options.template,
    el: options.el || document.body,
    render(data) {
      this.el.innerHTML = this.template(data)
    }
  }
  defaults.render(options.model);
  return defaults;
}


const app = Component({
  model:{
    num: '`number` 0'
  },
  template: templater`<h1>Clicked: ${'num'}</h1>`
});

let counter = 0;

document.addEventListener('click',() => {
    counter++;
    app.model.num = counter;
});
```

Reflect

Reflect is a new object in JavaScript that contains most of the methods that the Object has on it, however it allows us to use these methods with more reliability. When looking to find good examples of Reflect, I stumbled across a Stack Overflow that was great, you can find it in the list of additional resources below.

The goal of `Reflect` is to provide static methods that match with the proxy traps we just went over. The idea is that we can use the `Reflect.apply` method with out having to worry if our object that we want to use it on has implemented its own `.apply` method. It also allows us to work with our objects with more ease.

Checking for a key

In JavaScript if we want to check for a key on an Object we typically do something like this `key in myObject`, now we can is the `Reflect.has()` method that will return `true` or `false`.

```
const person = {
  name: "Ryan",
  age: 31
};

if('age' in person) {
  console.log('Age!')
}

if(Reflect.has(person, 'age')) {
  console.log('Relect age!')
}
```

There are a lot of other use cases for the `Reflect` object, however I do not want to rehash some of the great information out there on the Internet, so listed in the additional resources, I have added a bunch of great links.

Additional Resources

- https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Proxy
- http://www.2ality.com/2011/01/reflection-and-meta-programming-in.html
- https://ponyfoo.com/articles/es6-reflection-in-depth
- http://stackoverflow.com/questions/25421903/what-does-the-reflect-object-do-in-javascript
- https://github.com/tvcutsem/harmony-reflect/wiki