Introducing ES2015 Proxies

By Addy Osmani
Eng Manager, Web Developer Relations

ES2015 Proxies (in Chrome 49 and later) provide JavaScript with an intercession API, enabling us to trap or intercept all of the operations on a target object and modify how this target operates.

Proxies have a large number of uses, including:

- Interception
- Object virtualization
- Resource management
- Profiling or logging for debugging
- Security and access control
- Contracts for object use

The Proxy API contains a **Proxy constructor** that takes a designated target object and a handler object.

```
var target = { /* some properties */ };
var handler = { /* trap functions */ };
var proxy = new Proxy(target, handler);
```

The behavior of a proxy is controlled by the **handler**, which can modify the original behavior of the **target** object in quite a few useful ways. The handler contains optional trap methods (e.g `.get()`, `.set()`, `.apply()`) called when the corresponding operation is performed on the proxy.

Interception

Let's begin by taking a plain object and adding some interception middleware to it using the Proxy API. Remember, the first parameter passed to the constructor is the target (the object being proxied) and the second is the handler (the proxy itself). This is where we can add hooks for our getters, setters, or other behaviour.

```
var target = {};

var superhero = new Proxy(target, {
   get: function(target, name, receiver) {
       console.log('get was called for:', name);
       return target[name];
   }
});

superhero.power = 'Flight';
console.log(superhero.power);
```

Running the above code in Chrome 49 we get the following:

```
get was called for: power
"Flight"
```

As we can see in practice, performing our property get or property set on the proxy object correctly resulted in a meta-level call to the corresponding trap on the handler. Handler operations include property reads, property assignment, and function application, all of which get forwarded to the corresponding trap.

The trap function can, if it chooses, implement an operation arbitrarily (e.g forwarding the operation to the target object). This is indeed what happens by default if a trap doesn't get specified. E.g., here is a no-op forwarding proxy that does just this:

```
var target = {};

var proxy = new Proxy(target, {});
 // operation forwarded to the target
proxy.paul = 'irish';
// 'irish'. The operation has been  forwarded
console.log(target.paul);
```

Note: An example of using all available Proxy handler traps can be found on MDN.

We just looked at proxying plain objects, but we can just as easily proxy a function object, where a function is our target. This time we'll use the **`handler.apply()`** trap:

```
// Proxying a function object
function sum(a, b) {
    return a + b;
}
```

```
var handler = {
    apply: function(target, thisArg, argumentsList) {
        console.log(`Calculate sum: ${argumentsList}`);
        return target.apply(thisArg, argumentsList);
    }
};

var proxy = new Proxy(sum, handler);
proxy(1, 2);
// Calculate sum: 1, 2
// 3
```

Identifying proxies

The identity of a proxy can be observed using the JavaScript equality operators (== and ===). As we know, when applied to two objects these operators compare object identities. The next example demonstrates this behavior. Comparing two distinct proxies returns false despite the underlying targets being the same. In a similar vein, the target object is different from any of its proxies:

```
// Continuing previous example

var proxy2 = new Proxy (sum, handler);
(proxy==proxy2); // false
(proxy==sum); // false
```

Ideally, you shouldn't be able to distinguish a proxy from a non-proxy object so that putting a proxy in place doesn't really affect the outcome of your app. This is one reason the Proxy API doesn't include a way to check if an object is a proxy nor provides traps for all operations on objects.

Use cases

As mentioned, Proxies have a wide array of use cases. Many of those above, such as access control and profiling fall under **Generic wrappers**: proxies that wrap other objects in the same address "space". Virtualization was also mentioned. **Virtual objects** are proxies that emulate other objects without those objects needing to be in the same address space. Examples include remote objects (that emulate objects in other spaces) and transparent futures (emulating results that are not yet computed).

Proxies as Handlers

A pretty common use case for proxy handlers is to perform validation or access control checks before performing an operation on a wrapped object. Only if the check is successful does the operation get forwarded. The below validation example demonstrates this:

```
var validator = {
  set: function(obj, prop, value) {
    if (prop === 'yearOfBirth') {
      if (!Number.isInteger(value)) {
        throw new TypeError('The yearOfBirth is not an integer');
      }

      if (value > 3000) {
        throw new RangeError('The yearOfBirth seems invalid');
      }
    }

    // The default behavior to store the value
    obj[prop] = value;
  }
};

var person = new Proxy({}, validator);

person.yearOfBirth = 1986;
console.log(person.yearOfBirth); // 1986
person.yearOfBirth = 'eighties'; // Throws an exception
person.yearOfBirth = 3030; // Throws an exception
```

More complex examples of this pattern might take into account all of the different operations proxy handlers can intercept. One could imagine an implementation having to duplicate the pattern of access checking and forwarding the operation in each trap.

This can be tricky to easily abstract, given each op may have to be forwarded differently. In a perfect scenario, if all operations could be uniformly funneled through just one trap, the handler would only need to perform the validation check once in the single trap. You could do this by implementing the proxy handler itself as a proxy. This is unfortunately out of scope for this article.

Object Extension

Another common use case for proxies is extending or redefining the semantics of operations on objects. You might for example want a handler to log operations, notify observers, throw exceptions instead of returning undefined, or redirect operations to different targets for

storage. In these cases, using a proxy might lead to a very different outcome than using the target object.

```
function extend(sup,base) {

  var descriptor = Object.getOwnPropertyDescriptor(base.prototype,"constructor");

  base.prototype = Object.create(sup.prototype);

  var handler = {
    construct: function(target, args) {
      var obj = Object.create(base.prototype);
      this.apply(target,obj, args);
      return obj;
    },

    apply: function(target, that, args) {
      sup.apply(that,args);
      base.apply(that,args);
    }
  };

  var proxy = new Proxy(base, handler);
  descriptor.value = proxy;
  Object.defineProperty(base.prototype, "constructor", descriptor);
  return proxy;
}

var Vehicle = function(name){
  this.name = name;
};

var Car = extend(Vehicle, function(name, year) {
  this.year = year;
});

Car.prototype.style = "Saloon";

var Tesla = new Car("Model S", 2016);

console.log(Tesla.style); // "Saloon"
console.log(Tesla.name); // "Model S"
console.log(Tesla.year);  // 2016
```

Access Control

Access control is another good use case for Proxies. Rather than passing a target object to a piece of untrusted code, one could pass its proxy wrapped in a sort of protective membrane. Once the app deems that the untrusted code has completed a particular task, it can revoke the reference which detaches the proxy from its target. The membrane would extend this detachment recursively to all objects reachable from the original target that was defined.

Using reflection with proxies

Reflect is a new built-in object that provides methods for interceptable JavaScript operations, very much useful for working with Proxies. In fact, Reflect methods are the same as those of proxy handlers.

Statically typed languages like Python or C# have long offered a reflection API, but JavaScript hasn't really needed one being a dynamic language. One can argue ES5 already has quite a few reflection features, such as `Array.isArray()` or `Object.getOwnPropertyDescriptor()` which would be considered reflection in other languages. ES2015 introduces a Reflection API which will house future methods for this category, making them easier to reason about. This makes sense as Object is meant to be a base prototype rather than a bucket for reflection methods.

Using Reflect, we can improve on our earlier Superhero example for proper field interception on our get and set traps as follows:

```
// Field interception with Proxy and the Reflect API

var pioneer = new Proxy({}, {
  get: function(target, name, receiver) {
      console.log(`get called for field: ${name}`);
      return Reflect.get(target, name, receiver);
  },

  set: function(target, name, value, receiver) {
      console.log(`set called for field: ${name} and value: ${value}`);
      return Reflect.set(target, name, value, receiver);
  }
});

pioneer.firstName = 'Grace';
pioneer.secondName = 'Hopper';
// Grace
pioneer.firstName
```

Which outputs:

```
set called for field: firstName and value: Grace
set called for field: secondName and value: Hopper
get called for field: firstName
```

Another example is where one might want to:

- Wrap a proxy definition inside a custom constructor to avoid manually creating a new proxy each time we want to work with specific logic.
- Add the ability to 'save' changes, but only if data has actually been modified (hypothetically due to the save operation being very expensive).

```
function Customer() {

  var proxy = new Proxy({
    save: function(){
      if (!this.dirty){
        return console.log('Not saving, object still clean');
      }
      console.log('Trying an expensive saving operation: ', this.changedPropertie
    },

  }, {

    set: function(target, name, value, receiver) {
      target.dirty = true;
      target.changedProperties = target.changedProperties || [];

      if(target.changedProperties.indexOf(name) == -1){
        target.changedProperties.push(name);
      }
      return Reflect.set(target, name, value, receiver);
    }

  });

  return proxy;
}

var customer = new Customer();

customer.name = 'seth';
customer.surname = 'thompson';
// Trying an expensive saving operation:  ["name", "surname"]
customer.save();
```

For more Reflect API examples, see ES6 Proxies by Tagtree.

Polyfilling Object.observe()

Although we're saying goodbye to `Object.observe()`, it's now possible to polyfill them using ES2015 Proxies. Simon Blackwell wrote a Proxy-based Object.observe() shim recently that's worth checking out. Erik Arvidsson also wrote a fairly spec complete version all the way back in 2012.

Browser support

ES2015 Proxies are supported in Chrome 49, Opera, Microsoft Edge and Firefox. Safari have had mixed public signals towards the feature but we remain optimistic. Reflect is in Chrome, Opera, and Firefox and is in-development for Microsoft Edge.

Google has released a limited polyfill for Proxy. This can only be used for **generic wrappers**, as it can only proxy properties known at the time a Proxy is created.

Further reading

- ES6 in depth: Proxies and Reflect
- MDN: ES6 Proxies
- ES6 Proxies and Reflect on TagTree
- MDN: The Reflect Object
- ES6 Reflection in depth
- Proxies: Design Principles for Robust Object-oriented Intercession APIs
- 2ality: Metaprogramming with ES6
- Metaprogramming in ES6 using Reflect
- ES6 everyday Reflect

Last updated July 2, 2018.