Notes - JavaScript - Proxy

- Dr Nick Hayward

A brief introduction to various options for implementing a *Proxy* in JavaScript.

Contents

Intro We may use a *proxy* to control access to another object - a surrogate relationship between the proxy and the object.

A proxy may commonly be considered akin to a generalised *getter* and *setter*. However, whilst *getters* and *setters* may control access to a single object property, a proxy enables generic handling of interactions. Such interactions may even include method calls relative to an object.

So, we may use a proxy where we might otherwise use a getter and a setter. However, a proxy is considered broader and more powerful in its potential implementation and usage.

For example, a proxy may be used to add profiling support to an object or, perhaps, measure performance, autopopulate code properties, and so on. Suffice to say, there are many potential uses for a proxy in JavaScript.

Creating a proxy To create a proxy in JavaScript, we may use the default, built-in Proxy constructor.

```
// plain object
const planet = {
  name: ['mercury'],
  codes: {
    iau: 'Me',
    unicode: 'U+263F'
  }
};

// proxy for passed target object - target = planet
const planetDetails = new Proxy(planet, {
  get: (target, key) => {
    return key in target ? target[key] :'planet does not exist...';
  },
  set: (target, key, value) => {
    key in target ? target[key].push(value) : 'key not found...';
  }
});

// check proxy access to target property
console.log(planetDetails.name);

// check proxy set against target property
// target = planet, key = name, value = earth
planetDetails.name = 'earth';

console.log(planetDetails.name);
```

For this example, we may access the object and its properties directly, but the proxy gives us extra utility.

For the getter and setter, we may check keys, values, &c. and also control how the object is updated. We may also add basic logging, if necessary.

So, after defining the initial plain object, `planet`, we may then wrap it using the Proxy constructor.

The current proxy includes a getter and setter method, which contains checks for required key in the original object. We may also choose how we would like to compute values, log usage and return, and so on.

proxy traps In the above example, we added a `get` and `set` trap for the defined target object, `planet`.

However, there are other traps we may use with a Proxy. For example,

- `apply` - activated for a function call
  - e.g. measuring performance
- `construct` - activated for `new` keyword
- `enumerate` - activated for `for-in` statements
- `getPrototypeOf` - activated for getting prototype value
- `setPrototypeOf` - activated for setting prototype value

These traps are, naturally, in addition to the existing `get` and `set` traps defined above.

There are also traps that we cannot override using a proxy. For example,

- equality operators - `==` and `===` and not equivalents

- `instanceof` and `typeof`

Equality operators, for example, should always return the same expected pattern of result. This equality check expects a boolean to be returned, and the rules to remain the same. A proxy, therefore, may not be used to modify this expected pattern of condition and return.

The very act of comparison should also prevent access to one of the objects by default return. Again, if equality could be trapped it would cause this side-effect for the comparison.

Likewise, for similar reasons, `instanceof` and `typeof` may not be trapped.

Logging with proxies We commonly use logging in development as a convenient tool for debugging and checking code.

We can output checks, and add debugging statements to various points within our code. However, we may quickly start to add many such logging statements to our code.

A better option, in particular as we consider abstraction and reuse of code, is to use a proxy for such logging.

custom proxy for logging To improve our code reuse and abstraction, we may define a proxy for logging within an app.

For example, we may define a custom function, which accepts a `target` object and returns a new Proxy object with a getter and setter method.

```
// logging with proxy - get and set traps defined
function logger(target) {
  return new Proxy(target, {
    get: (target, property) => {
      console.log(`property read - ${property}`);
      return target[property];
    },
    set: (target, property, value) => {
      console.log(`value '${value}' added to ${property}`);
      target[property] = value;
    }
  });
}
```

This is a custom logger, which wraps the passed target object in a proxy with defined getter and setter methods.

We may then use this custom function as follows,

```
// test object
let planet = {
  name: 'mercury'
};

// new planet object with proxy
planetLog = logger(planet);

// test getting - value for property returned by getter in logger() method...
console.log('default get = ', planetLog.name);

// test setting - value for property set against object
planet.code = 'Me';
```

So, in this example, we define the initial object, and then create a new object with a proxy wrapper. This proxy includes the necessary logger, which we've set for both the setter and getter methods.

As we read a property, the `get` method will log access and return the requested data.

Likewise, as we set data, we log this update, and then update the target.

custom proxy for measuring performance Another appropriate use of a Proxy is to test performance for a given function.

We may wrap a function with a Proxy, and then `apply` a trap. This trap may include a simple timer, or perhaps a detailed series of tests for the pass function.

For example, the following function simply loops through a passed counter and outputs a series of characters for each iteration.

```
// FN: test loop to output to terminal
function loopOutput(counter, marker = '-') {
  if (!counter) {
    return false;
  }
  // loop through passed counter - check number for even...
  for (i = 0; i <= counter; i++) {
    // check for even counter value
    if (i % 2 === 0) {
      process.stdout.write('+');
    } else {
      // console.log(marker);
      process.stdout.write(marker);
    }
  }
  console.log('\n');
  return true;
}
```

We may then wrap this function inside a Proxy, adding a simple timer for the duration of the loop,

```
// wrap function inside custom Proxy
loopTest = new Proxy(loopOutput, {
  // apply simple timer to loop function
  apply: (target, thisArg, args) => {
    console.time("loopTest");
    /* invokes target function - thisArg defines the `this` value
    * if no `thisArg`, undefined will be used instead...
    * thisArg = value to use as `this` when executing a callback
    * args passed to target function loopOutput
    */
    const result = target.apply(thisArg, args);
    console.timeEnd("loopTest");
    return result;
  }
});
```

In this Proxy, the `apply` property trap means the function value will be executed each time the `loopOutput` function is called.

This handler will now be executed on function invocation for `loopTest` .

We can then execute this function with its Proxy,

```
// call function with counter value and custom marker...
loopTest(75, '-');
```

The markers are output to the terminal with a record of the loop's performance in milliseconds.

The benefit of this approach is that we do not need to modify the original function, `loopOutput` , and the return, logic, computation &c. will all remain the same. The customisation in this example, for performance checking using the `apply` trap, does not affect the passed function.

So, the `loopOutput` function is now routed through the custom proxy each time it is executed.

custom proxy for property autopopulate A proxy may also be used to autopopulate properties.

For example, we might need to model a directory structure for a file save. This will require verification of a defined file path, or creation of directories to ensure a path may be completed successfully,

The latter option may be achieved using a custom proxy to create missing directories in a defined path structure.

e.g.

```
// FN: recursive check for dir path and file...
function Directory() {
  return new Proxy({}, {
    get: (target, property) => {
      console.log(`reading property...${property}`);
      // check if property already exists
      if (!(property in target)) {
        // if not - simply add a new directory to target
        target[property] = new Directory();
      }
      // otherwise return property as is from target
      // - write method not implemented for actual directory...
      return target[property];
    }
  });
}

// create new Proxy for function
const rootDir = new Directory();

try {
  // check properties relative to root dir...
  rootDir.testDir.test2Dir.testFile = "test.md";
  console.log('exception not raised...');
} catch (event) {
  // error handling for null exception should be OK due to custom proxy...
  console.log(`exception raised...${event}`);
}
```

In this example, we instantiate a Proxy object for the inital `rootDir` . The Proxy includes a siAn obvious benefit of this Reflect usage is the abstraction of `get` usage from the Proxy getter to a default, re-usable Reflect `get` method. We may use the Proxy getter to check against data, type &c. in the target, and then call the Reflect `get` method if successful.mple getter, which checks for the passed property relative to the

target. If the property does not exist, a recursive call may be executed. This recursion allows the directory to be checked, and then created if necessary.

If we check a directory path for a file, as a child of the instantiated Proxy object, we may then recursively check each child and create any missing structure.

Reflect a Proxy ES6 introduced a complement to Proxy usage with a new built-in object, *Reflect*.

Proxy traps are mapped one-to-one in the Reflect API, allowing an easy combination of Proxy and Reflect usage. So, for each trap there is a matching reflect method.

reflect - get trap For example, we might use `Reflect.get` to define default behaviour for a Proxy getter.

```
const handler = {
    get(target, key) {
        if (key.startsWith('_')) {
           throw new Error(`Property "${ key }" is inaccessible.`)
        }
        return Reflect.get(target, key)
    }
}

const target = {}
const proxy = new Proxy(target, handler)
proxy._secret
```

In the above example, we are now unable to access the `_secret` property.

An obvious benefit of this Reflect usage is the abstraction of `get` usage from the Proxy getter to a default, re-usable Reflect `get` method. We may use the Proxy getter, for example, to check against data, type &c. in the target, and then call the Reflect `get` method if successful.

This is a useful option for restricting access to certain properties through a Proxy. We may expose the Proxy instead of the underlying object, setting access privileges according to requirements. If successful, a request will then be handled by the Reflect API method.

So, access must now go through the Proxy, and meet its rules and requirements.

reflect - false return However, returning an error may still be an indication that the `_secret` property exists. An alternative is to return an explicit `false` boolean value for the requested hidden property.

```
const handler = {
    get(target, key) {
        if (key.startsWith('_')) {
           return false;
        }
        return Reflect.get(target, key)
    }
};

const library = {
    archive : 'waldzell',
    curator : 'knechts',
    _secret : true
};
const proxy = new Proxy(library, handler);
console.log(`secret = ${proxy._secret}`);
console.log(`archive = ${proxy.archive}`);
```

A request for underscore value names may still be checked using

```
// _secret is not a private property in object -
console.log(proxy.hasOwnProperty('_secret'))
```

This means *underscore* property names are still not private and remain visible to specific property checks.

reflect - set trap As expected, we may also apply reflection to `set` traps. The reflected `set` method defines behaviour for a setter on a given Proxy object. It is equivalent to the default behaviour for the proxy.

For example,

```
set(target, key, value) {
  return Reflect.set(target, key, value)
}
```

We may also add various checks for the passed key, for example.

So, we may now update our previous example to include a `set` trap with Proxy support.

```
const handler = {
    get(target, key) {
        if (key.startsWith('_')) {
           // return false to show prop doesn't exist...
           return false;
        }
        return Reflect.get(target, key)
    },
    set(target, key, value) {
        return Reflect.set(target, key, value);
    }
};
```

and then test property access using the `get` and `set` traps

```
const library = {};
const proxy = new Proxy(library, handler);
proxy.archive = 'mariafels';
proxy._secret = true;
```

reflect - defaults & checks As we use the Reflect object as the default for traps, we may add checks, updates &c. to the Proxy trap itself.

For example, we might add a conditional check to the Proxy, and then pass a successful update or query to the Reflect method.

A default Reflect method allows abstraction for traps from the Proxy.

For example, we might update each trap in the above example with a call to the following conditional check

```
function keyCheck(key, action) {
    if (key.startsWith('_')) {
        throw new Error(`${action} action is not permitted on '${ key }'`)
     }
}
```

This function is called in each trap before continuing to the Reflect method for `get` or `set` .

Proxy wrapper To ensure we restrict access to a `target` object to the defined proxy and reflect traps, we need to wrap the `target` itself in a Proxy.

In previous examples, the target object may have been accessed directly in certain contexts. Whilst this might be beneficial for an admin mode and access, it is customary to wrap such objects in the Proxy to restrict access to the defined traps and handlers.

For example, we can modify our previous example for `get` and `set` traps

```
function proxyWrapper() {
    const target = {};
    const handler = {
        get(target, key) {
            if (key.startsWith('_')) {
               // return false to show prop doesn't exist...
               return false;
            }
            return Reflect.get(target, key)
        },
        set(target, key, value) {
            return Reflect.set(target, key, value);
        }
    };
    return new Proxy(target, handler);
}
```

`target` may now be accessed and managed using an instantiated proxy,

```
const proxiedObject = proxyWrapper();
// set prop & value on target using proxy set trap
proxiedObject.archive = 'waldzell';
// target accessible using proxy get trap
console.log(`target archive = ${proxiedObject.archive}`);
```

However, `target` may not be accessed directly using standard property access,

```
// target not directly accessible
console.log(`target = ${target}`);
```

pass object to wrapper We may modify this wrapper to also accept an existing object, which may then be returned wrapped in a Proxy.

For example,

```
const archive = {
    name: 'waldzell'
}

const proxiedArchive = proxyWrapper(archive);
```

check object We may also add a further check to ensure we always have a target object to work with, regardless of passed argument value.

For example, we may add a check to the `proxyWrapper` function to ensure target is always an object

```
// check object & return empty object if necessary...
function checkTarget(original) {
    // check for existing target object
    if (original.typeof !== 'object' || original === undefined) {
        console.log('not object...');
        const target = {};
        return target;
    } else {
        const target = original;
        return target;
    }
}
```

If we pass a string, for example, instead of a target object, we can now create a proxy wrapper with an empty object.

```
const proxiedArchive = proxyWrapper('archives');
// set prop & value on target using proxy set trap
proxiedArchive.admin = 'knechts';
proxiedArchive._secret = '1235813';
```

The properties for `admin` and `_secret` may now be set against an empty object due to the passed `archives` string.

We can call this function at the top of the `proxyWrapper` function,

```
function proxyWrapper(original) {
    // check target for proxy wrapper - original must be object
  const target = checkTarget(original);
  ...
}
```

update property access check We may also abstract the initial check for property access using a defined character delimiter.

For example,

```
// check property access using defined char delimiter
function checkDelimiter(key, char) {
    // check key relative to specified char delimiter
    if (key.startsWith(char)) {
        // return false to show prop not available
        return true;
     }
}
```

This will simply check the defined delimiter character relative to the passed property key. This may then be called in the `proxyWrapper` function,

```
if (checkDelimiter(key, '_')){
  return false;
}
```

restricting access In the above examples, we define the `target` object both inside and outside the `proxyWrapper` function.

However, both may be effective options for restricting object access depending upon the context.

Internal object declaration for `target` restricts full access to the Proxy object. Any traps for the object will only be accessible using the Proxy object. The consumer must use the instantiated Proxy object to read, write, query &c.

An external `target` object may still be useful after it has been wrapped by a Proxy object. Restricted access is controlled by only exposing the target as a Proxy object. Again, if we exposed the target as an access point for a pubic API, the proxy object will be exposed and not the original target object.

Proxy & schema validation Objects may be defined for a specific purpose or context, which requires control over stored properties and values.

Validation allows us define the structure of an object, its properties, types, permitted values &c. For example, we might define validation for a user form, which is then saved to a custom object.

Validation may be defined and controlled using various options in an application. We may use a third party module or, perhaps, a custom function, which returns an error for invalid input and data. However, we'd still need to ensure that the object storing the input data is restricted to authorised access both internal and external to the app.

Another option is to use a Proxy with validation of the object. This proxy object may then be used to provide access to the model object for validation. A further benefit of a proxy with validation is the separation of concerns. The data object remains separate from the validation.

A consumer never accesses the input object directly. Instead, they are given a proxy object with validation checks and balances. The original input object remains a plain object due to the nature of the Proxy object usage.

Any defined proxy handlers for validation &c. may also be referenced and reused across multiple Proxies.

proxy and validator We may create an initial validator using a Proxy, a map, and defined handlers for required object properties.

For example, as a property is set through a proxy object, its key may be checked against the map. If there is a rule for the key, its *handler* value will be executed to check that the property is valid.

```
// MAP - validation rules for properties
const validationMap = new Map();

// TRAPS - define traps for proxy
const validator = {
    // set trap
    set(target, key, value) {
        // check map for matching handler
        if (validationMap.has(key)) {
            // return handler function if available...pass value as parameter
            return validationMap.get(key)(value);
        }

        // else - default reflect set method for proxy
        return Reflect.set(target, key, value);
    }
};
```

The value may be passed as a parameter to the handler function stored in the map for the requested key. This function may include a validation, check &c.

```
// RULES - define executable rules for permitted object properties
// e.g. log, update state, get state, broadcast, subscribe...
// e.g. sample validation for text to log
function validateLog(text) {
    if (typeof text === 'string') {
        console.log(`logger = ${text}`);
    } else {
        throw new TypeError(`logger requires text input...`);
    }
}
```

We may then use this proxy and map as follows,

```
// set key and handler function in map
validationMap.set('logger', validateLog);
// empty object to wrap with proxy
const process = {};
// instantiate proxy object
const proxyProcess = new Proxy(process, validator);

// string set using handler for logger
proxyProcess.logger = 'test string = hello proxy...';
// number will not be set - fails validation
proxyProcess.logger = 96;
```

Resources

- MDN - JS Proxy