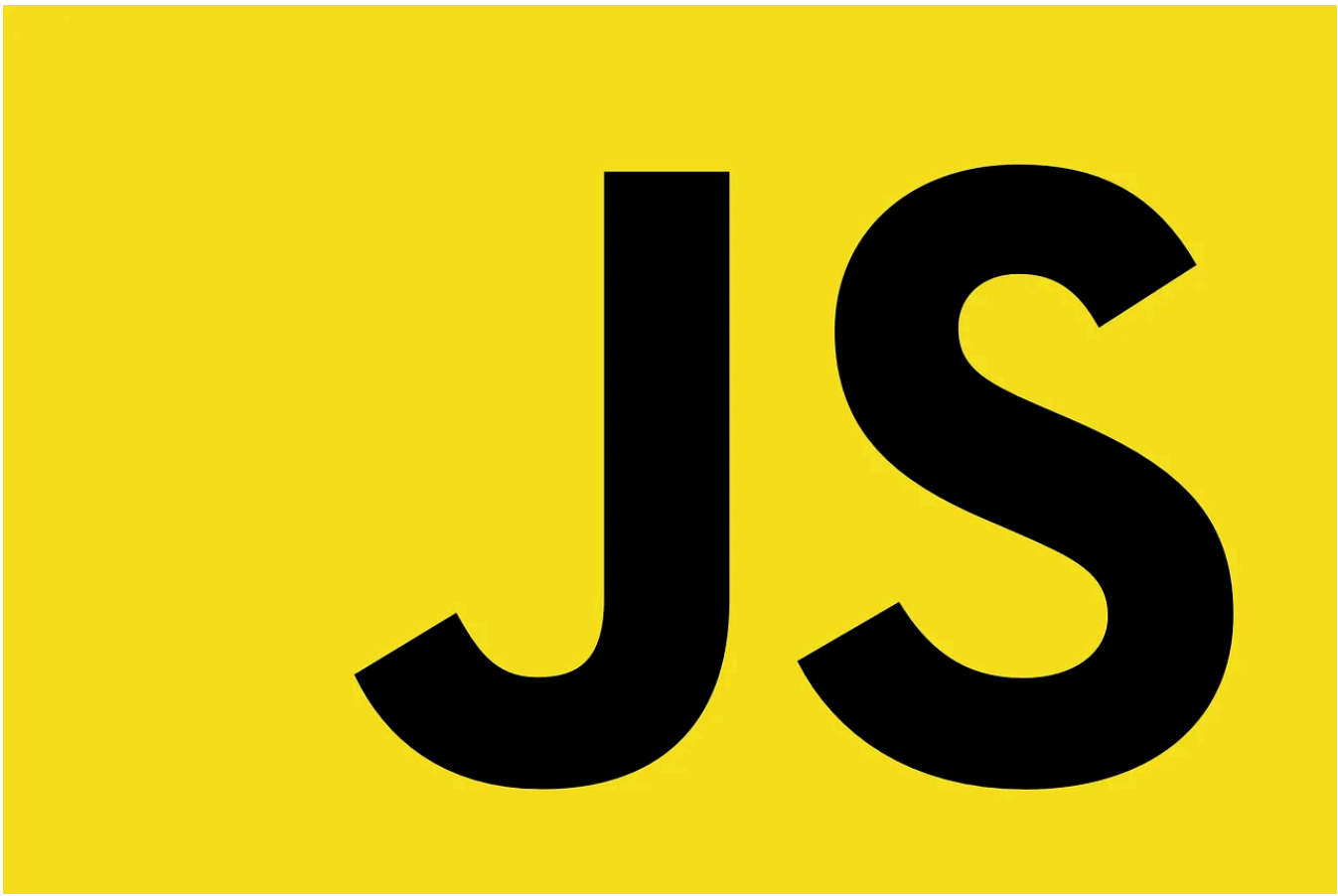
Dev Proto · Follow publication

JavaScript: What's `__proto__`?

5 min read · Aug 1, 2020

Chidume Nnamdi 🔥💻🎵🎮 Follow

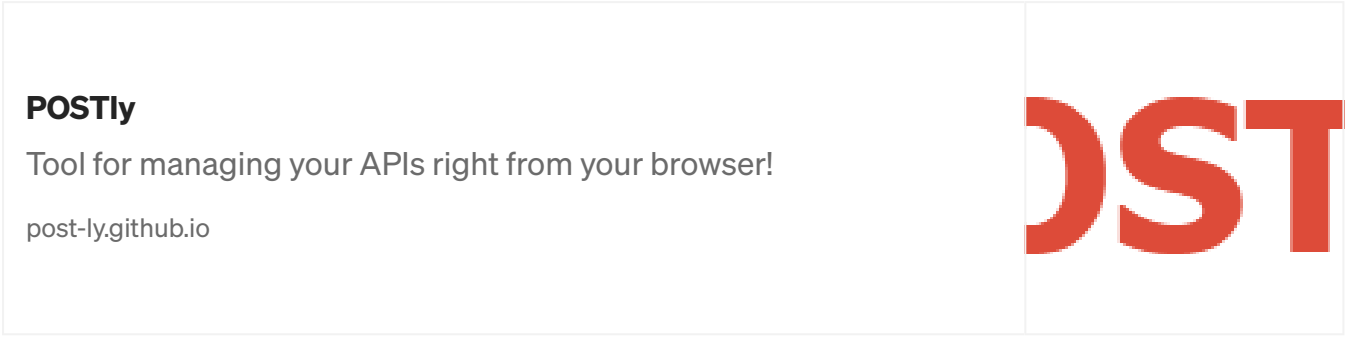
Listen Share More



`__proto__` is a way to inherit properties from an object in JavaScript. `__proto__` a property of Object.prototype is an accessor property that exposes the [[Prototype]] of the object through which it is accessed.

. . .

POSTly is a web-based API tool that allows for fast testing of your APIs (REST, GraphQL). Check it out here:



. . .

This `__proto__` sets all properties of the object set in its [[Prototype]] to the target object.

Let's look at an example:

```
const l = console.log

const obj = {
    method: function() {
        l("method in obj")
    }
}
const obj2 = {}

obj2.__proto__ = obj
obj2.method()
```

We have two object literals: obj and obj2. obj has a method property, method. obj2 is an empty object literal i.e it has no properties.

Moving down, we access the `__proto__` of obj2 and set it to obj. This will copy all the properties of the obj accessible via Object.prototype to obj2. That's why we can call the `method` on obj2 without getting an error despite not being defined there.

```
node proto
method in obj
```

obj2 has inherited the properties of obj, so the `method` method property will be available in its properties.

proto is used on Objects e.g object literal, Object, Array, Function, Date, RegEx, Number, Boolean, String.

Using proto is the same thing as using the `extends` keyword in OOP languages.

Let's add multiple properties on obj2 to see what proto would do.

```
const l = console.log
const obj = {
    method: function() {
        l("method in obj")
    },
    method2: function() {
        l("method2 in obj")
    },
    prop: 90
}

const obj2 = {}
obj2.__proto__ = obj
```

We have a second method `method2` set on obj and a property variable `prop` set with value 90.

Now, obj2 with its proto set to obj, it will inherit all the properties on obj: method, method2 and prop. So we can access method plus method2 and prop properties on obj2.

```
obj2.method()
obj2.method2()
l(obj2.prop)

$ node prop
method in obj
method2 in obj
90
```

Usage on JavaScript `class`

`__proto__` can be used to inherit properties from an object of JS classes defined with `class`.

```
class C {}

class D {
    meth() {
        l("method on D")
    }
}

const d = new D()
const c = new C()

c.__proto__ = d
c.meth()
```

We have two classes C and D. class C has no properties, its empty. class D has one property, meth, a property method. We create instances of C and D and set them to c and d respectively. Then the proto on c is set to d. All the properties of class D are set to c.

The c.meth() call on c will go through without error:

```
$ node proto
method on D
```

We see the `method on D` logged in the terminal. c has no properties but with `__proto__` it inherited all the properties in class D.

static properties cannot be inherited

As we said, the properties on Object.prototype can be inherited, static properties cannot be inherited.

For example in our class D, if we set a static method on it:

```
class C {}
class D {
    static stMeth() {
```

```
        l("static method stMeth on D")
    }

    meth() {
        l("method on D")
    }
}

const d = new D()
const c = new C()
c.__proto__ = d
c.meth()
c.stMeth()
```

The `c.stMeth` will throw a TypeError ststing c.stMeth is not a function. `__proto__` doesn't set static properties from the target object to the desired object.

Also if we set a static member variable to class D, c will inherit it:

```
class C {}
class D {
    static stMeth() {
        l("static method stMeth on D")
    }

    meth() {
        l("method on D")
    }
    static staticProp = 78
}
const d = new D()
const c = new C()

c.__proto__ = d

l(c.staticProp)
```

A static variable `staticProp` is set to class D. The `c.__proto__ = d` will not set the static property.

Prototype and `__proto__`

Using prototypes is the ideal way in JS to define OOP standards, the `class` we used above is only syntactic sugar for Prototype.

This:

```
class C {}
```

is transpiled to this:

```
function C() {}
```

and class D:

```
class D {
    static stMeth() {
        l("static method stMeth on D")
    }

    meth() {
        l("method on D")
    }
    static staticProp = 78
}
```

is transpiled to this:

```
function D() {}

D.prototype.meth = function () {
        l("method on D")
}

D.stMEth = function() {
        l("static method stMeth on D")
}

D.staticProp = 78
```

`__proto___` picks the properties in the prototype and set it to the target objects prototype property.

So when we did this `c.__proto__ = d`

`meth` property in `D`'s prototype is set to `c` object.

The prototype in Object is set to the proto.

`__proto__` can be set inside the object

`__proto__` is a property in an object. So `__proto__` can be set inside an object literal like this:

```
const D = function() {}
D.prototype.method = function() {
    l("method on D")
}

const obj = {
    __proto__: D.prototype
}
obj.method()
```

See, the `__proto__` property is set in object literal obj. We set it to point to `D.prototype`.

Whenever JS creates an object, it adds `__proto__` property to the object.

```
function D() {}
const d = new D()
```

object d an instance of D will have a `__proto__` property set to `D {}`.

```
l(d.__proto__)

D {}
```

If we set d.proto to function C:

```
function C() {}

function D() {}

const c = new C()
const d = new D()
```

```
d.__proto__= c
l(d.__proto__)
```

The property `__proto__` property on object d will point to `C {}`.

```
node proto
C {}
```

When a property in an object is accessed, the property is searched through its `__proto__` object, before searching the instance of the object.

```
class C {
    meth() {
        l("meth method in C")
    }
}

class D {
    static stMeth() {
        l("static method stMeth on D")
    }

    meth() {
        l("method on D")
    }
    static staticProp = 78
}
const d = new D()
const c = new C()

c.__proto__ = d
c.meth()
```

class C and D both have the `meth` method. c inherits d. When meth is called on c, the `__proto__` is searched first which will be properties on D before C.

See the result:

```
method on D
```

Warning

`__proto__` was never part of the initial ECMAScript spec, it was just added by browsers and got popularised over the years, so it was eventually added to 2015 ECMAScript spec. But it was still discouraged to use it, it is advisable to use `Object.setPrototypeOf` and `Object.getPrototypeOf` instead.

Conclusion

`__proto__` is very powerful. Its ability to collect properties for us is unparalleled.

If you have any question regarding this or anything I should add, correct or remove, feel free to comment, email, or DM me

Thanks !!!

Get my eBook

I have written an eBook that explains a lot of JavaScript concepts in simpler terms with reference to the EcmaSpec as a guide:

- Understanding JavaScript

Thanks for stopping by my little corner of the web. I think you'll love my email newsletter about programming advice, tutoring, tech, programming and software development. Just sign up below:

Follow me on Twitter.