Proxies

Learn how to define custom behaviors for fundamental operations with Proxies.

WE'LL COVER THE FOLLOWING

What is a Proxy?

From MDN:

> The **Proxy** object is used to define custom behavior for fundamental operations (e.g. property lookup, assignment, enumeration, function invocation, etc).

How to use a `Proxy` ?

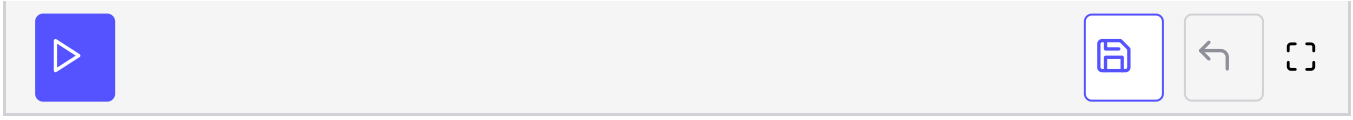
This is how we create a Proxy:

```
var x = new Proxy(target,handler)
```

- our `target` can be anything, from an object, to a function, to another `Proxy`
- a `handler` is an object which will define the behavior of our `Proxy` when an operation is performed on it

```
// our object
const dog = { breed: "German Shephard", age: 5}
```

```
// our Proxy
const dogProxy = new Proxy(dog, {
  get(target,breed){

    return target[breed].toUpperCase();
  },
  set(target, breed, value){
    console.log("changing breed to...");
    target[breed] = value;
  }
});

console.log(dogProxy.breed);
// "GERMAN SHEPHARD"
console.log(dogProxy.breed = "Labrador")
// changing breed to...
// "Labrador"
console.log(dogProxy.breed);
// "LABRADOR"
```

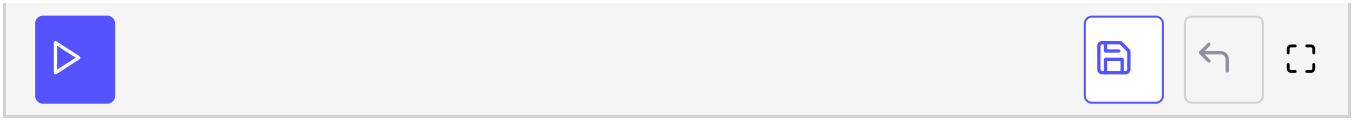


When we call the `get` method, we step inside the normal flow and change the value of the breed to uppercase.

When setting a new value, we step in again and log a short message before setting the value.

Proxies can be very useful. For example, we can use them to validate data.

```
const validateAge = {
  set: function(object,property,value){
    if(property === 'age'){
      if(value < 18){
        throw new Error('you are too young!');
      } else {
        // default behaviour
        object[property] = value;
        return true
      }
    }
  }
}

const user =  new Proxy({},validateAge)

user.age = 17
// Uncaught Error: you are too young!
```

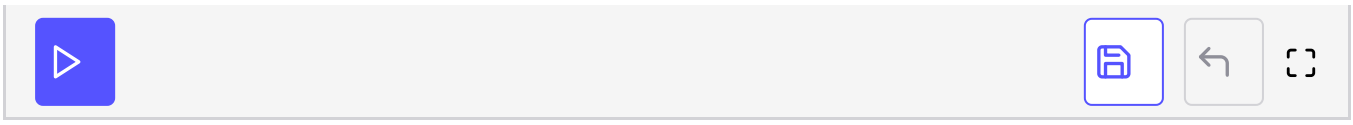


When we set the `age` property of the `user Object`, we pass it through our

`validateAge` function which checks if it is more or less than 18 and throws an error if it's less than 18.

Now let's try passing a different value:

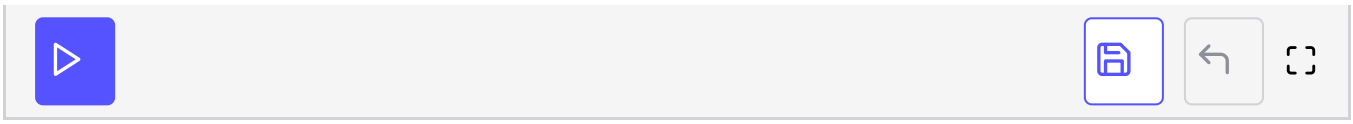
```
const validateAge = {
  set: function(object,property,value){
    if(property === 'age'){
      if(value < 18){
        throw new Error('you are too young!');
      } else {
        // default behaviour
        object[property] = value;
        return true
      }
    }
  }
}

const user =  new Proxy({},validateAge)

user.age = 21
console.log(user.age);
// 21
```



Proxies can be very useful if we have many properties that would require a **getter** and **setter** each. We need to define only one **getter** and one **setter** when using a `Proxy`. Let's look at this example:

```
const dog  = {
  _name: 'pup',
  _age: 7,

  get name() {
    console.log(this._name)
  },
  get age(){
    console.log(this._age)
  },

  set name(newName){
    this._name = newName;
    console.log(this._name)
  },
  set age(newAge){
    this._age = newAge;
    console.log(this._age)
  }
}

dog.name;
// pup
```

```
dog.age;
// 7
dog.breed;

// undefined
dog.name = 'Max';
// Max
dog.age = 8;
// 8
```



Notice that i'm writing `_name` instead of `name` etc..., the `_` symbol is used in `JavaScript` convention to define **Private** properties, meaning properties that should not be accessed by instances of the same class. That is not something that `JavaScript` enforces, it's just for developers to quickly identify **Private** properties.The reason why I'm using it here is because if I were to call:

```
set name(newName){
  this.name = newName;
}
```

This would cause an infinite loop as `this.name =` would call the setter again and again. By putting the underscore in front of it it, I can achieve the same result that I would get by renaming the setter to something else For example:

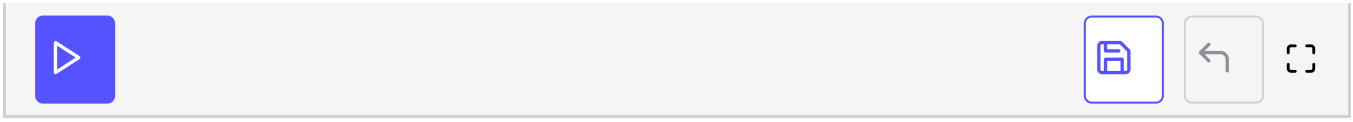
```
set rename(newName){
  this.name = newName;
}
```

As you can see we had three properties: `name`, `age` and `breed`. For each of them we had to create a **getter** and a **setter**. In the case of `breed` we only created a **getter** so when we tried to access the **setter** nothing happened and the property did not change.

We can simplify the code with a `Proxy` by writing the following:

```
const dog  = {
  name: 'pup',
  age: 7
}
const handler = {
  get: (target, property) => {
    property in target ? console.log(target[property]) : console.log('property not found');
  },
  set: (target, property, value) => {
```

```
  set: (target, property, value) => {
    target[property] = value;
    console.log(target[property])
  }
}

const dogProxy = new Proxy(dog, handler);

dogProxy.name;
// pup
dogProxy.age;
// 7
dogProxy.breed;
// property not found
dogProxy.name = 'Max';
// Max
dogProxy.age = 8;
// 8
```



First, we created our `dog` Object but this time we did not set any **getter** or **setter** inside of it. We created our `handler` that will handle each possible property with only one **getter** and **setter**. What we are doing with the **getter** is checking if the `property` is available on the `target` Object. If it is, we log it, otherwise we log a custom message. The setter takes three arguments, the `target` object, the `property` name, and the `value`. Nothing special happens here, we set the property to the new value and we log it.

As you can see, by using a `Proxy` we achieved two things:

- shorter, cleaner code
- we are logging a custom message if we try to access a property that is not available.

Up next is... you guessed it.