Proxy and Reflect APIs

🪞 `Proxy` and `Reflect` APIs in JavaScript — Explained in Detail

The `Proxy` and `Reflect` APIs were introduced in ES6 to give developers more **control and introspection** over objects. They are especially powerful for **meta-programming**, which means writing code that manipulates or enhances the behavior of other code.

🔶 `Proxy` — Intercept and Customize Object Behavior

✅ What is a Proxy?

A `Proxy` is an object that **wraps another object** and allows you to intercept and customize fundamental operations on it — like reading, writing, or function invocation.

🔶 Syntax:

```
const proxy = new Proxy(target, handler);
```

- `target`: the object you want to wrap
- `handler`: an object that defines **traps** — methods that intercept operations

---

📌 Common Proxy Traps

Trap	Description
`get`	Intercepts property access
`set`	Intercepts property assignment
`has`	Intercepts the `in` operator
`deleteProperty`	Intercepts `delete` operations
`apply`	Intercepts function calls
`construct`	Intercepts `new` operator

✅ Example: Logging All Property Accesses

```
const user = { name: 'Abhi', age: 25 };

const proxy = new Proxy(user, {
  get(target, prop) {
    console.log(`Getting ${prop}`);
    return target[prop];
  },
  set(target, prop, value) {
    console.log(`Setting ${prop} to ${value}`);
    target[prop] = value;
    return true;
  }
});

console.log(proxy.name);    // Logs: Getting name → Abhi
proxy.age = 30;             // Logs: Setting age to 30
```

🧠 Use Cases of Proxy

- **Validation**: Validate property values
- **Logging**: Track changes or access
- **Default values**: Return defaults when a key is missing
- **Access control**: Hide or restrict data
- **Reactive programming** (used in Vue.js)

🔷 `Reflect` API — The Companion to Proxy

The `Reflect` object provides **methods for object operations** — like reading, writing, or applying functions. It is essentially the **"default behavior"** for the traps used in `Proxy`.

🔶 Example:

```
const obj = { name: 'Abhi' };

// Traditional way
obj.name = 'Gurjar';

// Using Reflect
Reflect.set(obj, 'name', 'Gurjar');
console.log(Reflect.get(obj, 'name')); // Gurjar
```

✅ Common `Reflect` Methods

Method	Equivalent Operation
`Reflect.get(obj, prop)`	`obj[prop]`
`Reflect.set(obj, prop, val)`	`obj[prop] = val`
`Reflect.has(obj, prop)`	`prop in obj`
`Reflect.deleteProperty(obj, prop)`	`delete obj[prop]`
`Reflect.ownKeys(obj)`	`Object.getOwnPropertyNames` + `Object.getOwnPropertySymbols`
`Reflect.construct()`	`new` operator
`Reflect.apply()`	Function invocation

🎯 Proxy + Reflect: Best Practice

Use `Reflect` inside your Proxy traps to **preserve default behavior**.

```
const proxy = new Proxy(user, {
  get(target, prop) {
    console.log(`Getting ${prop}`);
    return Reflect.get(target, prop);  // Recommended
  },
  set(target, prop, value) {
    console.log(`Setting ${prop}`);
    return Reflect.set(target, prop, value);  // Recommended
```

```
  }
});
```

🧪 Advanced Use Case: Access Control

```
const secureData = {
  username: 'admin',
  password: '123456'
};

const proxy = new Proxy(secureData, {
  get(target, prop) {
    if (prop === 'password') {
      throw new Error('Access denied');
    }
    return Reflect.get(target, prop);
  }
});

console.log(proxy.username); // admin
console.log(proxy.password); // ❌ Error: Access denied
```

✅ Summary

Feature	`Proxy`	`Reflect`
Purpose	Intercept and customize behavior	Perform default object ops
Usage	Meta-programming, reactivity	Helper for proxy trap defaults
Methods	Traps like `get`, `set`, `apply`	Methods like `get`, `set`, etc.

Let me know if you'd like a real-world use case or project (like form validation or logging object state with Proxies).