What is Destructuring?

According to Mozilla, "destructuring assignment syntax is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects into distinct variables." Did you follow that? It basically means that you have a data structure, an array or object, and you want information from it to be its own variable. Before we look at how to achieve that through destructuring let us examine how we might currently perform this task.

Here we have our initial data structures.

*const* person = {

    firstName: 'Bob',

    lastName: 'Marley',

    email: 'bob@marley.com',

    password: 'sekureP@ssw0rd9',

    username: 'barley',

    createdAt: 1543945177623

};

*const* animals = ['horse', 'dog', 'fish', 'cat', 'bird'];copy

Assume we want email from person and the first animal as standalone variables. Before ES6 that might look like this:

// BEFORE ES6

*var* email = person.email;

*var* firstAnimal = animals[0];copy

Let's do that again, but with destructuring assignment:

// AFTER ES6

*const* { email } = person;

*const* [firstAnimal] = animals;

console.log(email);

// => bob@marley.com

console.log(firstAnimal);

// => horsecopy

In order to destructure, our declarations need to match the data type to the right of the equal sign. For objects, list the properties you want to copy into variables and for arrays, create variables to capture content at certain positions. Interesting, but is it really more useful than our pre-ES6 example? What happens to the original content?

Good questions. The former will be best answered with another example. For the latter; the original content is unchanged. Destructuring does not destroy the origin data, it merely makes copies of it and sets new variables.

Let's expand our previous scenario. Now we want email and password from person and the first three strings from animals.

*var* email = person.email;

*var* password = person.password;

*var* firstAnimal = animals[0];

*var* secondAnimal = animals[1];

*var* thirdAnimal = animals[2];copy

While not pretty, it does have the benefit of being verbose. We know exactly where each variable is receiving its information. Now with destructuring:

*const* { email, password } = person;

*const* [firstAnimal, secondAnimal, thirdAnimal] = animals;copy

Destructuring allows us to extract multiple items at once, we get all the same data while being significantly more succinct. This method is just as verbose *if* you understand destructuring syntax.

Name Conflicts

What if my current scope already has a variable of the same name of a property that I'm extracting from an object?

*const* password = '12345';

*const* { password } = person;copy

This will certainly cause conflict. Because password exists before destructing and is a const this will throw an error: SyntaxError: Identifier 'password' has already been declared.

So do we have to fallback to this?

*const* hashedPassword = person.password;copy

Can we destructure into a distinct variable?

*const* { hashedPassword } = person;copy

That first example will work, but it's unecessary. The second example won't throw an error, but it won't have the information you expect either. When destructuring from objects the property names must be matched exactly, so in this example, because hashedPassword does not exist on our person object, it will be undefined.

Not to worry. The JS maintainers already considered this scenario and have a way to specify exactly what you want to destructure, whilst also supplying a new variable name.

*const* password = '12345';

*const* { password: hashedPassword } = person;copy

Specifying the property name, password, will access the value of that property on our object. The colon : after the property name followed by your new variable redirects the value of password to be held in hashedPassword, thereby eliminating the aforementioned identifier conflict.

Nested Destructuring

Often we'll be working with much more complex content and we want to take advantage of destructuring these nested structures. For this next example we'll modify our person object to have an array of addresses.

*const* person = {

firstName: 'Bob',

lastName: 'Marley',

email: 'bob@marley.com',

password: 'sekureP@ssw0rd9',

username: 'barley',

addresses: [

{

address: '1600 Pennsylvania Avenue',

city: 'Washington, D.C.',

zipcode: '20500',

},

{

address: '221B Baker St.',

city: 'London',

zipcode: 'WC2N 5DU',

}

],

createdAt: 1543945177623

};copy

If we want the individual addresses as variables we could certainly destructure addresses, then destructure each address.

*const* { addresses: [whiteHouse, sherlock] } = person;copy

Note that a variable addresses is no longer created, but we can reference each address by the variable name listed at those positions.

What if we want to skip the first address and only want the city of the second, but using an alternate variable name?

*const* { addresses: [ , { city: london }] } = person;

console.log(london);

// => Londoncopy

Leaving an 'empty' first position (just a comma) allows skipping that index and destructuring from the second address. This can be done to any depth, just be sure your content actually exists.

*const* { addresses: [ , , , , { city: london }] } = person;copy

The above example will produce an error. We skipped so many addresses that the final position doesn't have an address object. Attempting to destructure from undefined or null throws: TypeError: Cannot destructure property 'city' of 'undefined' or 'null'.

Destructuring simplifies extracting information from objects and arrays, allowing for a cleaner, more succinct codebase. We're not done learning about destructuring but we need to understand a couple other features before we can fully utilize it.