Factory Function

For more in-depth: <https://medium.com/javascript-scene/javascript-factory-functions-with-es6-4d224591a8b1>

A factory function is any function which is not a class or constructor that returns a (presumably new) object. In JavaScript, any function can return an object. When it does so without the new keyword, it’s a factory function.

JavaScript provides a very handy object literal syntax. It looks something like this:

const user = {  
 userName: 'echo',  
 avatar: 'echo.png'  
};

Like JSON (which is based on JavaScript’s object literal notation), the left side of the : is the property name, and the right side is the value. You can access props with dot notation:

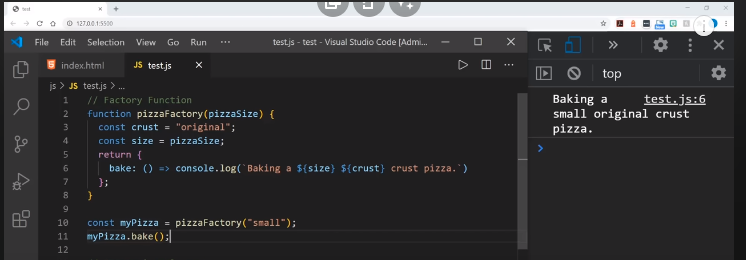
console.log(user.userName); // "echo"

You can access computed property names using square bracket notation:

const key = 'avatar';console.log( user[key] ); // "echo.png"

const userName = 'echo';  
const avatar = 'echo.png';const user = {  
 userName,  
 avatar  
};console.log(user);  
// { "avatar": "echo.png", "userName": "echo" }

example of factory function



Let’s turn our user object into a createUser() factory:

const createUser = ({ userName, avatar }) => ({

userName,

avatar,

setUserName (userName) {

this.userName = userName;

return this;

}

});

console.log(createUser({ userName: 'echo', avatar: 'echo.png' }));

/\*

{

"avatar": "echo.png",

"userName": "echo",

"setUserName": [Function setUserName]

}

\*/

**Returning Objects**

Arrow functions (=>) have an implicit return feature: if the function body consists of a single expression, you can omit the return keyword: () => 'foo' is a function that takes no parameters, and returns the string, "foo".

Be careful when you return object literals. By default, JavaScript assumes you want to create a function body when you use braces, e.g., { broken: true }. If you want to use an implicit return for an object literal, you’ll need to disambiguate by wrapping the object literal in parentheses:

const noop = () => { foo: 'bar' };

console.log(noop()); // undefined

const createFoo = () => ({ foo: 'bar' });

console.log(createFoo()); // { foo: "bar" }

In the first example, foo: is interpreted as a label, and bar is interpreted as an expression that doesn’t get assigned or returned. The function returns undefined.

In the createFoo() example, the parentheses force the braces to be interpreted as an expression to be evaluated, rather than a function body block.

Destructuring

Pay special attention to the function signature:

const createUser = ({ userName, avatar }) => ({

In this line, the braces ({, }) represent object destructuring. This function takes one argument (an object), but destructures two formal parameters from that single argument, userName, and avatar. Those parameters can then be used as variables in the function body scope. You can also destructure arrays:

const swap = ([first, second]) => [second, first];

console.log( swap([1, 2]) ); // [2, 1]

And you can use the rest and spread syntax (...varName) to gather the rest of the values from the array (or a list of arguments), and then spread those array elements back into individual elements:

const rotate = ([first, ...rest]) => [...rest, first];

console.log( rotate([1, 2, 3]) ); // [2, 3, 1]

for more:

<https://medium.com/javascript-scene/javascript-factory-functions-with-es6-4d224591a8b1>