Callbacks Vs Promises and basics of JS

Abhinav Rai Follow
Jul 3, 2017 · 4 min read

Life for JS developer is not easy without understanding the basics of what callbacks and promises are. There is very small difference between the two.

Lets start with some basics.

1. What is (req,res) => {} notation in JS ?

Well this is ES6 arrow function equivalent to **function(req,res) {}.** Read more on how the behaviour of `this` is inside arrow functions here.

2. What does bind method do in JS ?

The **`bind()`** method creates a new function that, when called, has its `this` keyword set to the provided value, with a given sequence of arguments preceding any provided when the new function is called.

(req, res) => {} also help in binding the context. Read this for its use in react here.

```
var Button = function(content) {
  this.content = content;
};
Button.prototype.click = function() {
  console.log(this.content + ' clicked');
}

var myButton = new Button('OK');
myButton.click();

var looseClick = myButton.click;
looseClick(); // not bound, 'this' is not myButton - it is
the global object

var boundClick = myButton.click.bind(myButton);
boundClick(); // bound, 'this' is myButton
```

Response:

```
OK clicked
undefined clicked
OK clicked
```

3. What is Closure?

A closure is an inner function that has access to the outer (enclosing) function's variables—scope chain. The closure has three scope chains: it has access to its own scope (variables defined between its curly brackets), it has access to the outer function's variables, and it has access to the global variables.

Callbacks are also closures as the passed function is executed inside other function just as if the callback were defined in the containing function. Closures have access to the containing function's scope, so the callback function can access the containing functions' variables, and even the variables from the global scope.

A great read on closure can be found here.

4. What is yarn?

Ever came us with yarn installations? The most popular JavaScript package manager is the npm client, which provides access to more than 300,000 packages in the npm registry. But as the size of codebase and the number of engineers grew, they ran into problems with consistency, security, and performance.

Yarn is a new JavaScript package manager built by Facebook, Google, Exponent and Tilde. As can be read in the official announcement, its purpose is to solve a handful of problems that these teams faced with npm, namely:

- installing packages wasn't fast/consistent enough, and
- there were security concerns, as npm allows packages to run code on installation.

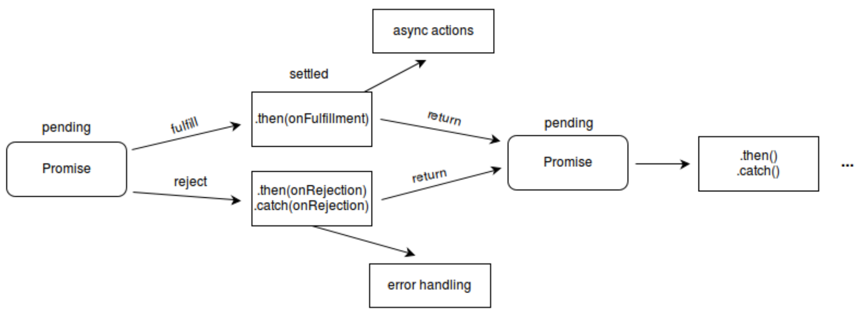
Yarn is only a new CLI client that fetches modules from the npm registry. Nothing about the registry itself will change—you'll still be able to fetch and publish packages as normal.

Callbacks or Higher Order Functions

Now coming to Callbacks. This sounds very disturbing at first but is very easy as we read and understand about it. A great read to start from basics about callback is here. I recommend to read it first.

We can pass multiple parameter to the callback function depending on the function.

Promises



How Promise work

```
// Here's how you create a promise:

var promise = new Promise(function(resolve, reject) {
  // do a thing, possibly async, then…

  if (/* everything turned out fine */) {
    resolve("Stuff worked!");
  }
  else {
    reject(Error("It broke"));
  }
});
```

The promise constructor takes one argument, a callback with two parameters, resolve and reject. Do something within the callback, perhaps async, then call resolve if everything worked, otherwise call reject.

Like `throw` in plain old JavaScript, it's customary, but not required, to reject with an Error object.

Here’s how you use that promise:

```
promise.then(function(result) {
  console.log(result); // "Stuff worked!"
}, function(err) {
  console.log(err); // Error: "It broke"
});
```

We can use chaining and return the objects in chain for synchronous as shown below:

```
getJSON('supplyData.json').then(function(supply) {
  return getJSON(supply[0]);
}).then(function(supply0) {
  console.log("Got supply0!", supply0);
})
```

What’s ‘.error’ then in promises?

As we saw earlier, `then()` takes two arguments, one for success, one for failure (or fulfill and reject, in promises-speak):

```
get('supplyData.json').then(function(response) {
  console.log("Success!", response);
}, function(error) {
  console.log("Failed!", error);
})
```

Now with .catch ()

```
get('supplyData.json').then(function(response) {
  console.log("Success!", response);
}).catch(function(error) {
  console.log("Failed!", error);
})
```

There’s nothing special about `catch()` , it's just sugar for `then(undefined, func)` , but it's more readable. Note that the two code examples above do not behave the same, the latter is equivalent to:

```
get('supplyData.json').then(function(response) {
  console.log("Success!", response);
}).then(undefined, function(error) {
  console.log("Failed!", error);
})
```

Browsers are pretty good at downloading multiple things at once, so we're losing performance by downloading supplydata one after the other. What we want to do is download them all at the same time, then process them when they've all arrived. Thankfully there's an API for this:

```
Promise.all(arrayOfPromises).then(function(arrayOfResults) {
  //...
})
```

In promises, we can only return ONE object. We cannot return multiple arguments in then.

Difference between callback and promises

As explained above, promises are cleaner way for running asynchronous tasks to synchronous and also provide catching mechanism which are not in callbacks. Promises are built over callbacks. Promises are a very mighty abstraction, allow cleaner and better, functional code with less error-prone boilerplate.