Asynchronous JavaScript

Overview

- callbacks : example setTimeout
- asynchronous IO : ajax
- from callbacks to Promises
- async await

pdf

Problem statement

Problem statement :

1. do something
2. ask for processing that will take too long to wait and return a value
3. do something with that value

Extended problem :

1. do something
2. ask for processing that will take too long to wait and return a value
3. do something else that is independant
4. do something with that value

Extended extended problems :

- do this with multiple async actions
- wait for multiple results

Callbacks

- JavaScript has features that help :
  - the function to help with capturing code to be executed later : a function is a set of coherent instructions that can be triggered, e.g. at the end of a wait
  - the function can be defined inside another function to have a specific variable context (closure)
- Example : code to load a file and find the last occurrence of a string in it then call function fun on this information

```
function f(fileName, searchString, fun) {
    fs.readFile(fileName, function(err, data) {
        if (err) {
            // do something to process the error
        }
        if (data) {
            fun(data.lastIndexOf(searchString));
        }
    });
}
```

Warnings

- returning a value is not possible, you can only provide a function to consume the value when ready.
- order of execution is not obvious

```
function f(fileName, searchString, fun) {
    fs.readFile(fileName, function(err, data) {
        if (err) {
            // do something to process the error
        }
        if (data) {
            fun(data.lastIndexOf(searchString));
        }
    });
}
```

Another example : animations

```
function animate() {
    // do something
    ...
    setTimeout(animate, 100); // call myself in 100ms
}
```

This function is called every 100ms after the first time, which you have to call to start.

Another example : Ajax

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "test.txt");
xhr.onload = function() {
  alert(this.responseText);
}
xhr.send();
```

1. create the xhr object
2. position the method and url
3. define the callback
4. start executing the background code

Consequences

- if result comes later, errors also come later, in a context they may not be understood : context needs to be kept not only for processing results
- debugging is a mess as the order of execution depends on things out of my control
- errors are not always predictable
- if you did not provide error recovery, errors can be really hard to understand
- you have to create your code in a way resistant to out-of-order results
- you may not have thought of all the actual dependencies

Promises

- Promises are a way to structure asynchronous code that is convenient/readable
- Promises are still new and you may find Promise code that is “legacy” (behaves differently)
- Using promises :

```
// f is a function that returns a promise when all its stuff i
f()
  .then(resultProcessing)
  .catch(errorProcessing)
  .finally(doItAnyWay)

function resultProcessing(result) {...}
function errorProcessing(error) {...}
function doItAnyWay() {...}
```

Why does this notation work ?

```
f().then(resultProcessing).catch(errorProcessing).finally(doI
```

- the function f is called and returns a pending promise
- then is called and returns a pending promise (same object as above)
- catch is called and returns a pending promise (same object as above)
- finally is called and returns a pending promise (same object as above)
- the code after that is executed until there is a thread break . . . After that, either :
- resultProcessing is called with the value passed to resolve, followed by doItAnyWay()
- or errorProcessing is called with the value passed to reject, followed by doItAnyWay()

Creating promises

- A Promise needs one function with two parameters :
  - a function `resolve` called when the processing is successful, to pass the result of the processing on
  - a function `reject` called when there is an error
- A Promise is in one of three states :
  - pending (in progress)
  - fulfilled (resolve has been called)
  - rejected (reject has been called)
- To create a Promise :

```
let p = new Promise(function (resolve, reject) {...}) ;
// the anonymous function should call resolve with the result
// the processing OR call reject with the reason (error)
```

Rewrite Ajax as Promise

```
function get(url) {
    return new Promise((resolve, reject) => {
        var xhr = new XMLHttpRequest();
        xhr.open("GET", url);
        xhr.onload = () => resolve(xhr);
        xhr.onerror = () => reject(xhr);
        xhr.send();
    });
}
```

Combining promises

Promise.all(iterable).then(. . .)

I have used this function on an array of promises (the `iterable`) to wait for the completion of all the promises in the array

Promise.any(iterable).then(. . .)

This is the opposite of Promise.all, and then is executed with the value of the first promise that is resolved in the `iterable`

More on Promises

More on Promises

Async / await

- Async/await is actually just syntax sugar built on top of promises. It cannot be used with plain callbacks or node callbacks.
- Async/await is, like promises, non-blocking.
- Async/await makes asynchronous code look and behave a little more like synchronous code. This is where all its power lies.

Promise code :

```
const makeRequest = () =>
 getJSON()
   .then(data => {
     console.log(data)
     return "done"
   })

makeRequest()
```

Discussion of async/await vs Promise

- async/await is only relevant for code USING Promises, it seems you need to learn the Promises anyway if you need to write code that creates Promises. . .
- It is only syntax, but the syntax seems to be simpler in more complex cases, including the cases where multiple Promises are involved
- If you do not know Promises syntax, you may be better off learning the async syntax directly
- More complex cases seem to be A LOT simpler with async than with Promises
  - especially debug
- It looks like on of those `des gouts et des couleurs...` cases
  - One thing is clear : learn async/await and Promises
- async cannot be used at the top level, so the top level code has to be in Promise form

More

More on Async/await

More on asynchronous JS