305CDE Lab 5

JavaScript Promises

Colin Stephen

October 2014

Overview

Dealing with asynchronous code.

- Callback Hell
- "Future-Facing" Objects: Promises

Callback Hell

Callbacks

Remember that functions are *first class objects* in JS. So they can be used as parameter values ("callbacks") for input to other functions.

```
function some_function(param, callback) {
  // async code here - may take a while
  callback(param);
}
some_function("yay", function(value) {
  console.log("callback called! " + value);
});
```

- Second input to `some_function()` is a callback function

Nested Callbacks

But what if `some_function()`'s *callback* uses a callback as one of its arguments?

- We get a *nested callback*
- Unfortunately this pattern can continue for several layers

```
some_function(param, function(err, res) {
  some_function2(param, function(err, res) {
    some_function3(param, function(err, res) {
      some_function4(param, function(err, res) {
        some_function5(param, function(err, res) {
          some_function6(param, function(err, res) {
            // do something useful
          });
        });
      });
    });
  });
});
```

The "Problem"

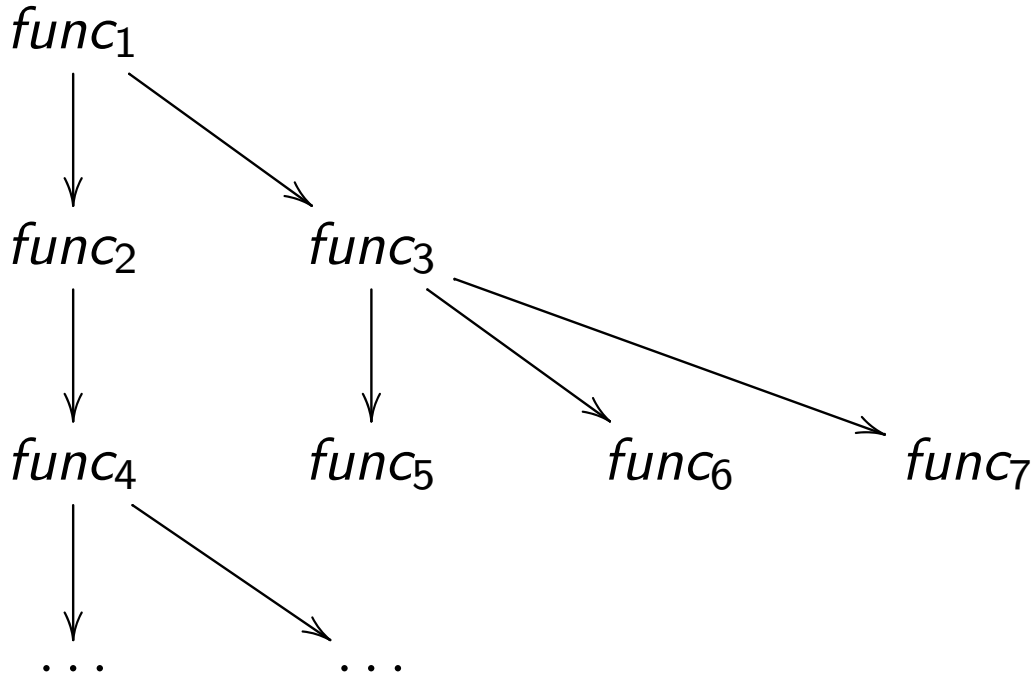
- JS is an event-based language
- So even in moderately complex programs various *chains of events* need to be handled
- Using callbacks to do this (with nesting) makes the code very hard to read
  - Code that is hard to read is hard to debug
  - Code that is hard to read is hard to refactor
  - Code that is hard to read is hard to collaborate on with colleagues

Best case: the function calls are *linear* down the chain - i.e. each function has at most one callback.

$$func_1 \rightarrow func_2 \rightarrow func_3 \rightarrow \ldots \rightarrow func_n$$

The Real Problem

Common case: the function calls are *branched* down the chain - i.e. at least one function has two or more callbacks.



Branching Callbacks

Real problems!

- If the callbacks branch, we can't know the order of the function calls
- It can be very complex to "reassemble" data returned from the various branches correctly
- Scoping becomes a challenge

Possible solutions?

1. Avoid *anonymous* function callbacks. Replace them with (un-nested) named functions defined in their own blocks. But:
   - It is easier but still difficult to read the "meaning" or "intention" of the code
   - Branching and scoping are still challenges
2. Use event listeners when possible

```
var img1 = document.querySelector('.img-1');
img1.addEventListener('load', function() {
  // woo yey image loaded
});
```

- But what about events that happen before binding?
- What about *combinations of events* happening??

A Better Solution

- Use callbacks in very simple (one or two nested layers) situations
- Use listeners mainly for events that can happen *multiple times on the same object*:
  - `keyup`
  - `click`
  - etc.
- BUT otherwise use *JavaScript Promises* to handle multiple asynchronous event chains
  - in particular success/failure chains arising in AJAX calls!

JavaScript Promises

What Is A JS Promise?

The core idea behind promises is that a **promise object** represents the result of an asynchronous operation. The promise object can be in one of three different "states":

1. pending - The initial state of a promise.
2. fulfilled / resolved - The state of a promise representing a successful operation.
3. rejected - The state of a promise representing a failed operation.

Together, the last two are also referred to as *settled*. Once a promise is fulfilled or rejected, it can never change again.

Making Promises

To create a promise object when you need to deal with some async chaining, you just construct a `new Promise()` object:

```
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"));
  }
});
```

Note that you specify *under what conditions* to `resolve` or `reject` the promise object.

The `Promise` Constructor

How does the constructor work?

- Takes one argument - a callback function!!!
- The callback has two *function* parameters: `resolve` and `reject`

Inside the callback:

1. Run your async code
2. If it works, invoke `resolve()`
3. If it fails, invoke `reject()`

The latter two invocations pass their values "down the promise chain" to be used later...

Using Your Promise Object

Usually, you will want to do something once the promise code completes, which depends on the result of the promise (e.g. the data passed back, or an error occurring).

To do this you can use the `then()` method on promises.

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

Note that `then()` takes *two* callback arguments:

1. A function to call when `resolve()` is invoked
2. A function to call when `reject()` is invoked

These are passed the values given during the invocations.

Chaining With Promises

Transforming Values

You can attach multiple `then`'s to a promise object to transform returned values.

```
var promise = new Promise(function(resolve, reject) {
  resolve(1);
});

promise.then(function(val) {
  console.log(val); // 1
  return val + 2;
}).then(function(val) {
  console.log(val); // 3
});
```

For Example: Transforming Returned Data

Transforming values with multiple `then`'s is handy when data needs to be transformed.

- Suppose we have a `get(url)` function that returns a promise.
- The promise resolves when the data is retreived from the URL.

```
get('story.json')
.then(function(response) {
  return JSON.parse(response);
})
.then(function(response) {
  console.log("Yey JSON!", response);
});
```

- The first `then` transforms `response` to JSON
- The JSON is passed to the second `then` as *its* `response`

Queuing Async Actions

You can attach multiple `then`'s to a promise object to run asynchronous actions in sequence.

- Useful when one async action depends on the outcome of another!

How do you do this?

- Don't return a "value" from the `then`
  - This is what you did to transform values
- Instead, return a new `Promise()` object from the `then`

For Example: Multiple AJAX Calls

Say you make an AJAX call that returns some data. Part of that data is another URL to be called using AJAX. How would you chain the AJAX calls? Return promises!

- Suppose we have a `getJSON(url)` function that returns a promise.
- The promise resolves when the data is retreived from the URL.

```
getJSON('story.json')
.then(function(story) {
  // return another promise!
  return getJSON(story.chapterUrls[0]);
})
.then(function(chapter1) {
  console.log("Got chapter 1!", chapter1);
});
```

Catching Errors

Recall: the second callback to the `then()` method on promises is invoked if and when the promise is *rejected*, otherwise the first callback is invoked.

What if you just want to do something when an error occurs? Two possibilities:

1. `myPromiseObject.then(undefined, errorCallback);`
2. `myPromiseObject.catch(errorCallback);`

These have slightly different behaviour: see the HTML5Rocks Promises Blog Post for details.

Bonus Material

Additional Promise Methods

There are a couple of additional `Promise` methods that you will find useful, both of which take *arrays* as input:

`Promise.all()`

- Returns a promise that resolves when *all* of the promises in the input array have resolved.

`Promise.race()`

- Returns a promise that resolves or rejects as soon as one of the promises in the input array resolves or rejects, with the value or reason from that promise

More Useful Array Methods

Functional programming in JS is often made easier with a few key array methods, which are useful to learn about. Look them up: map, reduce, and filter.

`Array.prototype.map()`

- Creates a new array with the results of calling a provided function on every element in this array.

`Array.prototype.reduce()`

- Apply a function against an accumulator and each value of the array (from left-to-right) so as to reduce it to a single value.

`Array.prototype.filter()`

- Creates a new array with all of the elements of this array for which the provided filtering function returns true.