Promises in JavaScript

CPEN 400A – Winter Term 1 2018

Karthik Pattabiraman

Tuesday, November 5th, 2018

Outline

- **Promises introduction**
- Promises Examples
- Chaining Promises and Error Handling
- Multiple Promises

Call-backs in JavaScript

- JavaScript (esp. Node.js) allows you to have nested functions as call-backs
- Useful for remembering state and keeping track of deferred operations

```
var fs = require("fs");
var length = 0;
var fileName = "sample.txt";
fs.readFile(fileName, function(err, buf) {
if (err) {
        console.log("Error in reading file " + err);
} else {
        length = buf.length;;
        console.log("Number of characters read = " + length); }
} );
```

Call-back hell

- Too many call backs can lead to confusion - also known as call-back hell
- Difficult to keep track of order of call-backs and handle failures (if any)

Pyramid of Doom

```
app.post('/cparse', function(req, res) {
 customParse(req.body, function(er, list) {
    list.forEach(function (row, index) {
       buildScript(row, req.body.format).↩
            extractArgs(row, function (instType) {
          row.forEach(function (arg, i) {
           resolveAliases(instType, arguments0);
 }) }) }) })
   // send response back
})
```

Promises: Introduction

- Allow deferred execution without explicitly using call-backs
- Return a promise object that is either resolved or rejected later
- Promise object can be passed around to functions etc.
  - resolve handler is called if promise is successful
  - reject handler is called if promise fails

Promises: Advantages

- Prevent race conditions between event setup and event firing
  - Exactly once semantics - resolved or rejected
- Easier to compose together (promises returning promises)
- Allow multiple asynchronous events to be handled simultaneously
- Allow multiple handlers to be attached to same event
  - Chaining of handlers
  - Unified handling of errors

Support for Promises

- **Promises were originally supported only in Node.JS**
  - Through custom third-party libraries (e.g., Bluebird, Q, Jasmine)
  - Each had slightly different semantics and implementation
  - Jquery has a completely different view of Promises (best avoided)

- Starting ES6, Promises are part of standard **JavaScript (**not just Node)
  - Support for client-side and server-side code
  - Npm module – promises – included in Node.Js since 2016 at least
  - Supported on client-side by most ES6 compatible browsers
  - We'll only look at Promises in Node.Js on server-side though

Outline

- Promises introduction
- **Promises Examples**
- Chaining Promises and Error Handling
- Multiple Promises

ReadFile Example - 1

```
var fs = require("fs");
if (! fs) process.exit(1);

// This function reads a new file and returns a promise
// It doesn't wait for the read to be complete
function readFile(fileName) {
        return new Promise(function(resolve, reject) {
                console.log("Creating a new promise");
                fs.readFile(fileName, function(err, buf) {
                        if (err) {
                                console.log("Rejecting the promise");
                                reject(err);
                        } else {
                                console.log("Resolving the promise");
                                resolve(buf);
                        }
                }); // End of readFile
        }); // End of promise
};
```

ReadFile Example - 2

```
// Get a new promise when you call readFile
var promise = readFile(fileName);


// Setup the success and failure handlers for the promise
promise.then( function(contents) {
                // fulfilment
                console.log("Read " + contents.length + " bytes");
        }, function(err) {
                // rejection
                console.log("Error reading file " + err);
        }
);


console.log("End of program");
```

Points to Note

- New Promise returns right away and does not actually call resolve and reject functions till the promise is resolved or rejected later
- .then specifies the resolve and reject functions after setup
- Resolve is called if promise is resolved, reject if it’s rejected
- It’s fine to set either (or both) resolve and reject to be null

Multiple .then blocks

```
// Setup the success and failure handlers for the promise
promise.then( function(contents) {
                // fulfilment
                console.log("Read " + contents.length + " bytes");
        }, function(err) {
                // rejection
                console.log("Error reading file " + err);
        }
);

// Setup another set of then handlers
promise.then( function(contents) {
                // fulfilment
                console.log("Another handler for then");
        }, function(err) {
                // rejection
                console.log("Another handler for err: " + err.message);
        }
);
```

.catch block

- Used to catch errors in the promise or when it is rejected
- Can be replaced with then(null, foo())

```
// Setup just an error handler, without a resolution handler
// Equivalent to promise.then(null, function(err) { });
promise.catch( function(err) {
                // rejection
                console.log("Yet another handler for err : " + err.message);
        }
);
```

Settled Promises

- Promises can be rejected or resolved and they stay that way forever
- Sometimes you may want to create a settled promise (resolve/reject)

```
// Comment out the appropriate line of code below
var p = Promise.resolve("hello");
// var p = Promise.reject("hello");


p.then( function(str) {
                console.log("Resolved: " + str);
        }, function(err) {
                console.log("Error: " + err);
        }
);
```

Why create Settled Promises ?

- Test cases where you want to test all handlers of a promise
- Sometimes a library expects a promise as argument, but you already know the results of the call are not going to succeed
- For composing promises together, some of which may have their results known in advance but not others

Outline

- Promises introduction
- Promises Examples
- **Chaining Promises and Error Handling**
- Multiple Promises

Chaining Promises

- Promises can be chained together (i.e., executed one after another)
- Simulates multiple asynchronous handlers executing in sequence
- Each promise can be handled by a separate reject handler or a generic catch handler at the end of the chain
- Values can be passed down the chain from one handler to the next

Example of Chaining

```
var randomPromise = function(threshold) {
        console.log("Calling randomPromise");
        var r = Math.random();
        console.log( "Random " + r);
        return (r > threshold) ? Promise.resolve() : Promise.reject();
};

var p = randomPromise(0.5);
var foo = function() {
        console.log("Resolved");
}
var bar = function() {
        console.log("Rejected");
}
// This is how  you'd chain promises
// Can  you predict the output of this sequence ?
p.then(foo, bar).then(foo, bar).then(foo, bar);
```

Example of Chaining and Value Passing

```
var p = randomPromise(0.5, 100);

var foo = function(val) {
        console.log("Resolved: " + val);
        return val + 1;
}

var bar = function(val) {
        console.log("Rejected: " + val);
        return val + 1;
}

// This is how  you'd chain promises
// Can  you predict the output of this sequence ?
p.then(foo, bar).then(foo, bar).then(foo, bar).then(foo, bar);
```

Error Handling

- Catch handler at end can handle errors in the original promise or its preceding then handlers – sort of like a “catch all” block in try-catch

```
var p = errorPromise("Original");

// The same handler can handle both errors in the
// original promise and in any resolution handlers
p.then( function() {
                    console.log("resolved");
          }
).then( function() {
                    throw new Error("Resolution");
          }
).catch( function(err) {
                    console.log("Error : " + err.message);
          }
);
```

Outline

- Promises introduction
- Promises Examples
- Chaining Promises and Error Handling
- **Multiple Promises**

Multiple Promises: Sequential Execution

- A promise handler can itself return promises for downstream handlers

```
var delayedPromise = function(delay) {
        return new Promise( function(resolve, reject) {
                console.log("Delayed promise = " + delay);
                setTimeout(resolve, delay);
        });
}
var p = delayedPromise(1000);
p.then( function() {
        console.log("First promise");
        return delayedPromise(2000);
}).then( function() {
        console.log("Second promise");
        return delayedPromise(3000);
}).then( function() {
        console.log("Done");
});
```

Multiple Promises: Parallel Execution (all)

- Multiple promises can be issued in parallel and “joined” by Promise.all
  - Resolved when **all** the promises are resolved (rejected if even one is rejected)

```
var promises = []
for (var i=0; i<5; i++) {
        promises.push( valuePromise(i) );
}
// Wait for all the promises to be resolved
var result = Promise.all( promises );
// Add a resolution function to get the values of each promise
result.then( function(values) {
        console.log("All promises resolved");
        for (var j=0; j<values.length; j++)
                console.log( "Promise " + j + " returned " + values[j] );
        } ).catch( function(value) {
                console.log("Promise not resolved " + value);
        }
);
```

Multiple Promises: Parallel Execution (Race)

- Multiple promises can be issued in parallel and “joined” by Promise.race
  - Resolved (or rejected) when **any of** the promises are resolved (or rejected)

```
var promises = []
// Initialize the promises array with 'n' promises
for (var i=0; i<n; i++) {
        promises.push( valuePromise(i) );
}
// Wait for any of the promises to be resolved
var result = Promise.race( promises );
// Add a resolution function to get the values of each promise
result.then( function(value) {
        console.log( "Promise resolved : " + value );
} ).catch( function(value) {
        console.log("None of the promises resolved");
});
```

Class Activity

- Write a node.js program to read from two different text files and concatenate their contents using Promises. After both reads are complete, you should write the contents of the two files to a third file. You can assume that the order of reads is not important. You should not block for file read, nor read the files sequentially.

- How will you modify the above program if you wanted to write to the third file without waiting for both files to complete reading, again using promises ? Make sure that you follow the same constraints.

Outline

- Promises introduction
- Promises Examples
- Chaining Promises and Error Handling
- Multiple Promises