
ES6
callbacks
promises
generators

Promises

- Promises provides us a subtle way for Asynchronous programming in Javascript
- It is one of most Powerful features in Javascript combined with Generators ES6 becomes Truly Harmony and Symphonical
- Think it like promise to eventually having a value at some indeterminable point in the future.
- It's kind of like a way of making multiple potentially long tasks operate within a synchronous set of instructions.
- A promise is like that receipt. It's an object that stands in for a value that is not ready yet, but will be ready later—in other words, a future value. You treat the promise as if it were the value you're waiting for, and write your code as if you already had it.

Why we require Promise

- To understand need of promise we need to understand whats Synchronous and Asynchronous

```
// readfile_sync.js

"use strict";

// This example uses Node, and so won't run in the browser.
const filename = 'text.txt',
       fs       = require('fs');

console.log('Reading file . . . ');

// readFileSync BLOCKS execution until it returns.
//   The program will wait to execute anything else until this operation finishes.
const file = fs.readFileSync(`${__dirname}/${filename}`);

// This will ALWAYS print after readFileSync returns. . .
console.log('Done reading file.');

// . . . And this will ALWAYS print the contents of 'file'.
console.log(`Contents: ${file.toString()}`);
```

JAVASCRIPT

```javascript
// readfile_async.js

"use strict";

// This example uses Node, so it won't run in the browser.
const filename      = 'text.txt',
        fs            = require('fs'),
        getContents = function printContent (file) {
        try {
          return file.toString();
        } catch (TypeError) {
          return file;
        }
      }

console.log('Reading file . . . ');
console.log("=".repeat(76));

// readFile executes ASYNCHRONOUSLY.
//   The program will continue to execute past LINE A while
//   readFile does its business. We'll talk about callbacks in detail
//   soon -- for now, just pay mind to the the order of the log
//   statements.
let file;
fs.readFile(`${__dirname}/${filename}`, function (err, contents) {
  file = contents;
  console.log( `Uh, actually, now I'm done. Contents are: ${ getContents(file) }`);
}); // LINE A

// These will ALWAYS print BEFORE the file read is complete.

// Well, that's both misleading and useless.
console.log(`Done reading file. Contents are: ${getContents(file)}`);
console.log("=".repeat(76));
```

JAVASCRIPT

```javascript
"use strict";

// This example uses Node, and so won't run in the browser.
const filename = 'throwaway.txt',
        fs       = require('fs');

console.log('Reading file . . . ');

fs.readFile(`${__dirname}/${filename + 'a'}`, function (err, contents) {
  if (err) { // catch
    console.log( `There was a/n ${err}.` );
  } else   { // try
    console.log( `Got it. File contents are: '${file}'`);
  }
});
```

THE INFAMOUS PYRAMID OF DOOM.

```
step1(function (value1) {
    step2(value1, function(value2) {
        step3(value2, function(value3) {
            step4(value3, function(value4) {
                // Do something with value4
            });
        });
    });
});
```

Promise assurance

- Enter Promises to get the future data i.e writing code in assurance that data will be recieved
- Difference between promise and callback is callback are functions and promise is an object
- Promise API have 4 important methods

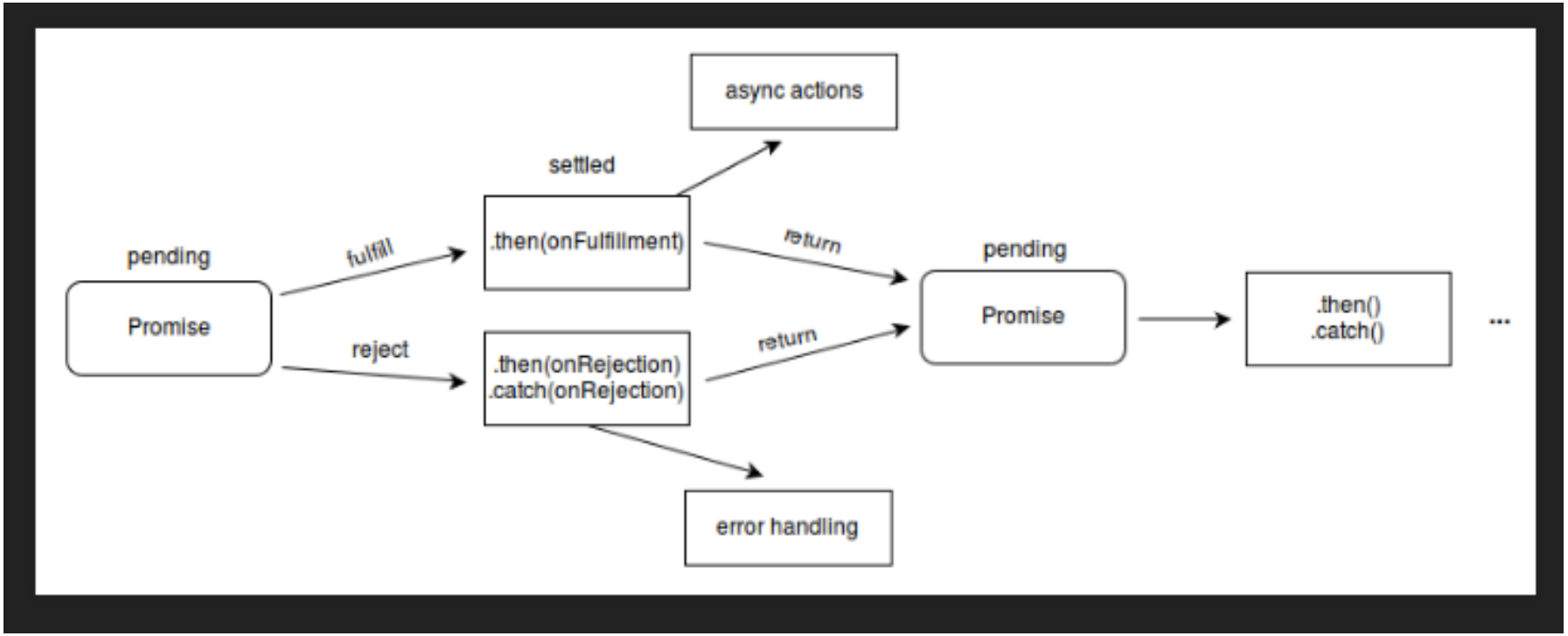
  resolve,reject,then,all

THE MORE MANAGEABLE TOWER OF DO AWESOME.

```
var step1 = new Promise(...);

step1.then(step2)
    .then(step3)
    .then(step4)
    .then(function (value4) {
        // Do something with value4
    })
    .catch(function (error) {
        // Handle any error from any above steps
        throw new Error('fatal error in promises:' + error.message);
    });
```

Promise Lifecycle



- Basics

```JS
'use strict';

const fs = require('fs');

const text =
  new Promise(function (resolve, reject) {
      // Does nothing
  })
```

JS

```
// constructor.js

const resolve = console.log,
      reject = console.log;
```

```
// constructor.js

const text =
  new Promise(function (resolve, reject) {
    fs.readFile('text.txt', function (err, text) {
      if (err)
        reject(err);
      else
        resolve(text.toString());
    })
  })
  .then(resolve, reject);
```

JS

```
const text =
  new Promise(function (resolve, reject) {
    fs.readFile('tex.txt', function (err, text) {
      if (err)
        reject(err);
      else
        resolve(text.toString());
    })
  })
  .then(resolve)
  .catch(reject);
```

Promises for API calls

JAVASCRIPT

```
function getData() {
    return new Promise((resolve, reject)=>{
        $.ajax({
            url: `http://www.omdbapi.com/?t=The+Matrix`,
            method: 'GET'
            }).done((response)=>{
                //this means my api call suceeded, so I will call resolve on the respo
                resolve(response);
            }).fail((error)=>{
                //this means the api call failed, so I will call reject on the error
                reject(error);
            });
    });
}
```

JAVASCRIPT

```
getData()
    .then(data => console.log(data))
    .catch(error => console.log(error));
```

Parallel Promises

- Sometimes we're working with multiple Promises and we need to be able to start our processing when all of them are fulfilled. This is where Promise.all() comes in. Promise.all() takes an array of Promises and once all of them are fulfilled it fulfills its returned Promise with an array of their fulfilled values.

```
var fetchJSON = function(url) {
  return new Promise((resolve, reject) => {
    $.getJSON(url)
      .done((json) => resolve(json))
      .fail((xhr, status, err) => reject(status + err.message));
  });
}
```

- Now we can setup an array of promises which will fulfill with the JSON results of fetching the response from each of the urls in our itemUrls array. Promise.all() will not fulfill until all the Promises in the array have fulfilled. If any of those promises are rejected (or throw an exception) then the Promise.all() Promise will reject and the .catch() below will be triggered.

```
var itemUrls = {
    'http://www.api.com/items/1234',
    'http://www.api.com/items/4567'
  },
  itemPromises = itemUrls.map(fetchJSON);

Promise.all(itemPromises)
  .then(function(results) {
     // we only get here if ALL promises fulfill
     results.forEach(function(item) {
       // process item
     });
  })
  .catch(function(err) {
    // Will catch failure of first failed promise
    console.log("Failed:", err);
  });
```

```
// A Promise that times out after ms milliseconds
function delay(ms) {
  return new Promise((resolve, reject) => {
    setTimeout(resolve, ms);
  });
}

// Which ever Promise fulfills first is the result passed to our handler
Promise.race([
  fetchJSON('http://www.api.com/profile/currentuser'),
  delay(5000).then(() => { user: 'guest' })
])
.then(function(json) {
   // this will be 'guest' if fetch takes longer than 5 sec.
   console.log("user:", json.user);
})
.catch(function(err) {
  console.log("error:", err);
});
```

Promises Summary

- These are the basics of the ES6 Promise specification in a nutshell.
- Promises give us the ability to write asynchronous code in a synchronous fashion, with flat indentation and a single exception channel.
- Promises help us unify asynchronous APIs and allow us to wrap non-spec compliant Promise APIs or callback APIs with real Promises.
- Promises give us guarantees of no race conditions and immutability of the future value represented by the Promise (unlike callbacks and events).
- But, Promises aren't without some drawbacks as well:
- You can't cancel a Promise, once created it will begin execution. If you don't handle rejections or exceptions, they get swallowed.
- You can't determine the state of a Promise, ie whether it's pending, fulfilled or rejected. Or even determine where it is in it's processing while in pending state.
- If you want to use Promises for recurring values or events, there is a better mechanism/pattern for this scenario called streams.

Generators

- Generator is another Powerful feature introduced in ES6

Generators

- Generator is another Powerful feature introduced in ES6
- Generators can pause a function and only continue when they are ordered to

Generators

- Generator is another Powerful feature introduced in ES6
- Generators can pause a function and only continue when they are ordered to
- Functions start behaving lazily and provide non blocking IO calls

Generators

- Generator is another Powerful feature introduced in ES6
- Generators can pause a function and only continue when they are ordered to
- Functions start behaving lazily and provide non blocking IO calls
- What this mean for Implementation in Real world?

Generators

- Generator is another Powerful feature introduced in ES6
- Generators can pause a function and only continue when they are ordered to
- Functions start behaving lazily and provide non blocking IO calls
- What this mean for Implementation in Real world?
- When used with promises lot of Ugly Nested code suddenly looks as straight as Synchronous but still as capability of Asynchronous

Basic Syntax

```
function *foo() {
  var x = 1 + (yield "foo");
   console.log(x);
}
var runner = foo()
runner.next() --- //Value : foo ,done false
//1
runner.next(0)--//Value : undefined ,done true
```

Generator Iterator

- Iterators are a special kind of behavior, a design pattern actually, where we step through an ordered set of values one at a time by calling next()
- Imagine for example using an iterator on an array that has five values in it: [1,2,3,4,5]. The first next() call would return 1, the second next() call would return 2, and so on.

Basic Syntax

- function *foo() {
    yield 1;
    yield 2;
    yield 3;
    yield 4;
    yield 5;
  }
  var it = foo()
  console.log(it.next()); // { value:2, done:false }
  console.log(it.next()); // { value:3, done:false }
  console.log(it.next()); // { value:4, done:false }
  console.log(it.next()); // { value:5, done:false }
  console.log(it.next()); // { value:undefined, done:true }

Avoid returns

It may not be a good idea to rely on the return value from generators, because when iterating generator functions with for..of loops (see below), the final returned value would be thrown away.

```
function *foo() {
    yield 1;
    yield 2;
    yield 3;
    yield 4;
    yield 5;
    return 6;
}

for (var v of foo()) {
    console.log( v );
}
// 1 2 3 4 5

console.log( v ); // still `5`, not `6` :(
```

Decode this

```
function *foo(x) {
    var y = 2 * (yield (x + 1));
    var z = yield (y / 3);
    return (x + y + z);
}

var it = foo( 5 );

// note: not sending anything into `next()` here
console.log( it.next() );       // { value:6, done:false }
console.log( it.next( 12 ) );   // { value:8, done:false }
console.log( it.next( 13 ) );   // { value:42, done:true }
```

Error Handling

```
function *foo() { }

var it = foo();
try {
    it.throw( "Oops!" );
}
catch (err) {
    console.log( "Error: " + err ); // Error: Oops!
}
```

Obviously, the reverse direction of error handling also works:

```
function *foo() {
    var x = yield 3;
    var y = x.toUpperCase(); // could be a TypeError error!
    yield y;
}

var it = foo();

it.next(); // { value:3, done:false }

try {
    it.next( 42 ); // `42` won't have `toUpperCase()`
}
catch (err) {
    console.log( err ); // TypeError (from `toUpperCase()` call)
}
```

Delegating

```
function *foo() {
    yield 3;
    yield 4;
}

function *bar() {
    yield 1;
    yield 2;
    yield *foo(); // `yield *` delegates iteration control to `foo()`
    yield 5;
}

for (var v of bar()) {
    console.log( v );
}
// 1 2 3 4 5
```

```
function *foo() {
    var z = yield 3;
    var w = yield 4;
    console.log( "z: " + z + ", w: " + w );
}

function *bar() {
    var x = yield 1;
    var y = yield 2;
    yield *foo(); // `yield*` delegates iteration control to `foo()`
    var v = yield 5;
    console.log( "x: " + x + ", y: " + y + ", v: " + v );
}

var it = bar();

it.next();      // { value:1, done:false }
it.next( "X" ); // { value:2, done:false }
it.next( "Y" ); // { value:3, done:false }
it.next( "Z" ); // { value:4, done:false }
it.next( "W" ); // { value:5, done:false }
// z: Z, w: W

it.next( "V" ); // { value:undefined, done:true }
// x: X, y: Y, v: V
```

```
function *foo() {
    yield 2;
    yield 3;
    return "foo"; // return value back to `yield*` expression
}

function *bar() {
    yield 1;
    var v = yield *foo();
    console.log( "v: " + v );
    yield 4;
}

var it = bar();

it.next(); // { value:1, done:false }
it.next(); // { value:2, done:false }
it.next(); // { value:3, done:false }
it.next(); // "v: foo"   { value:4, done:false }
it.next(); // { value:undefined, done:true }
```

Go Async

- As we saw basics how Generators and Promises work.Now we dwell into more deep and see how JS can be made Async neatly
- The main strength of generators is that they provide a single-threaded, synchronous-looking code style, while allowing you to hide the asynchronicity away as an implementation detail.
- The result? All the power of asynchronous code, with all the ease of reading and maintainability of synchronous(-looking) code.

How to do ?

- Simplest Async

```
function makeAjaxCall(url,cb) {
    // do some ajax fun
    // call `cb(result)` when complete
}

makeAjaxCall( "http://some.url.1", function(result1){
    var data = JSON.parse( result1 );

    makeAjaxCall( "http://some.url.2/?id=" + data.id, function(result2){
        var resp = JSON.parse( result2 );
        console.log( "The value you asked for: " + resp.value );
    });
} );
```

```
function request(url) {
    // this is where we're hiding the asynchronicity,
    // away from the main code of our generator
    // `it.next(..)` is the generator's iterator-resume
    // call
    makeAjaxCall( url, function(response){
        it.next( response );
    } );
    // Note: nothing returned here!
}


  function *main() {
      var result1 = yield request( "http://some.url.1" );
      var data = JSON.parse( result1 );

      var result2 = yield request( "http://some.url.2?id=" + data.id );
      var resp = JSON.parse( result2 );
      console.log( "The value you asked for: " + resp.value );
  }

  var it = main();
  it.next(); // get it all started
```

Real world Implementation

- We all want our code to be really neat like

```
runGenerator( function *main(){
    var result1 = yield request( "http://some.url.1" );
    var data = JSON.parse( result1 );

    var result2 = yield request( "http://some.url.2?id=" + data.id );
    var resp = JSON.parse( result2 );
    console.log( "The value you asked for: " + resp.value );
} );
```

```
async(function *() {
  try {
    var resultPromise1 = $.ajax("/request1");
    var resultPromise2 = $.ajax("/request2");
    var resultPromise3 = $.ajax("/request3");
    // Do something with the results
    let results = {"1": yield resultPromise1, "2": yield resultPromise2, "3": yiel
    console.log(results);
  } catch(xhr) {
    console.log("Error: " + xhr);
  }
});
```

- To Acheive this we have to use Promises and a Utility function like runGenerator in our example

```
function request(url) {
    // Note: returning a promise now!
    return new Promise( function(resolve,reject){
        makeAjaxCall( url, resolve );
    } );
}
```

```
// run (async) a generator to completion
// Note: simplified approach: no error handling here
function runGenerator(g) {
    var it = g(), ret;

    // asynchronously iterate over generator
    (function iterate(val){
        ret = it.next( val );

        if (!ret.done) {
            // poor man's "is it a promise?" test
            if ("then" in ret.value) {
                // wait on the promise
                ret.value.then( iterate );
            }
            // immediate value: just send right back in
            else {
                // avoid synchronous recursion
                setTimeout( function(){
                    iterate( ret.value );
                }, 0 );
            }
        }
    })();
}
```

```
function async(genertorFactory) {
  var generator = genertorFactory.apply(this, arguments);
  var handleResult = function(result) {
    if(result.done) return result.value;
      // In our example, the result.value would be a jqXHR object, which has a
      // then() method that is similar in its contract to the Promise objects
      // specified in A+ promises (for ex. https://www.promisejs.org/)
      return result.value.then(function(nextResult) {
        // Push the result back to the generator. This will be the
        // return value of the corresponding yield operation.
        return handleResult(generator.next(nextResult));
    }, function(error) {
      // Propagate the error back to the generator. This exception will be
      // thrown from the current suspended context of the generator, as if
      // the yield statement that is currently suspended were a `throw`
      // statement.
      generator.throw(error);
    })
  };
  return handleResult(generator.next());
}
```

To use it

```
function request(url) {
    return new Promise( function(resolve,reject){
        // pass an error-first style callback
        makeAjaxCall( url, function(err,text){
            if (err) reject( err );
            else resolve( text );
        } );
    } );
}

runGenerator( function *main(){
    try {
        var result1 = yield request( "http://some.url.1" );
    }
    catch (err) {
        console.log( "Error: " + err );
        return;
    }
    var data = JSON.parse( result1 );

    try {
        var result2 = yield request( "http://some.url.2?id=" + data.id );
    } catch (err) {
        console.log( "Error: " + err );
        return;
    }
    var resp = JSON.parse( result2 );
    console.log( "The value you asked for: " + resp.value );
} );
```

Parallell Execution

```
function request(url) {
    return new Promise( function(resolve,reject){
        makeAjaxCall( url, resolve );
    } )
    // do some post-processing on the returned text
    .then( function(text){
        // did we just get a (redirect) URL back?
        if (/^https?:\/\/.+/.test( text )) {
            // make another sub-request to the new URL
            return request( text );
        }
        // otherwise, assume text is what we expected to get back
        else {
            return text;
        }
    } );
}

runGenerator( function *main(){
    var search_terms = yield Promise.all( [
        request( "http://some.url.1" ),
        request( "http://some.url.2" ),
        request( "http://some.url.3" )
    ] );

    var search_results = yield request(
        "http://some.url.4?search=" + search_terms.join( "+" )
    );
    var resp = JSON.parse( search_results );

    console.log( "Search results: " + resp.value );
} );
```

```
async(function *() {
  try {
    var resultPromise1 = $.ajax("/request1");
    var resultPromise2 = $.ajax("/request2");
    var resultPromise3 = $.ajax("/request3");
    // Do something with the results
    let results = {"1": yield resultPromise1, "2": yield resultPromise2, "3": yiel
    console.log(results);
  } catch(xhr) {
    console.log("Error: " + xhr);
  }
});
```

ES7 Async

- In Future versions of ES7 we dont have to use such Utility functions

```
async function main() {
    var result1 = await request( "http://some.url.1" );
    var data = JSON.parse( result1 );

    var result2 = await request( "http://some.url.2?id=" + data.id );
    var resp = JSON.parse( result2 );
    console.log( "The value you asked for: " + resp.value );
}

main();
```

```
async function productReviewsByProductAllGet(request, response) {
  try {
    const { productId } = request.params;
    const reviews = await productReviewsQuery.selectAllByProductId(productId);
    const reviewPromises = reviews.map(
      (review) => accountsQuery.selectByAccountId(review.accountId)
                  .then(
                    ({firstName, lastName}) => Object.assign({}, review, { firstName, lastName })
                  )
                  .catch (error => null)
    )
    const result = await Promise.all(reviewPromises)
    response.status(200).json(result);
  } catch (error) {
    response.status(200).json({  error: 'No Results' });
  }
}
```

Summary

- Put simply: a generator + yielded promise(s) combines the best of both worlds to get really powerful and elegant sync(-looking) async flow control expression capabilities.
- With simple wrapper utilities (which many libraries are already providing), we can automatically run our generators to completion, including sane and sync(-looking) error handling!