Você vai sair usando Promises

Derek Stavis
Engenheiro de Software

Who?

then

Derek Stavis

github.com/derekstavis

Full-stack engineer

Healfies

Ruby, JavaScript, Python, C

Rails, AngularJS, ReactJS, NodeJS, Webpack

then

How have you been doing asynchronous JavaScript?

then

How have you been doing continuation passing?

then

ASYNC
IS
COMING

Using callback functions

then

```
// In the browser
setTimeout(function () {
  // Will be called in the future
}, 2000);


// In the server
fs.readFile('file.txt', function () {
  // Will be called when file.txt is read
});
```

Node.js callback standard

then

```
fs.readFile('file.txt', function (err, data) {
  // If an error occurred, err will have a value
  // Always check for errors using if clauses
})
```

Node.js callback scenario

then

- Let's say we have a `fetch` function
- It does plain HTTP GET
- Accepts a URL and a callback
- Callback receives error and response

```
fetch ('url', function (err, res) { })
```

Node.js callback scenario

then

```
fetch('/users/session', function (sessionError, user) {
  if (sessionError) {
    alert('Error fetching session')
    return
  }
  fetch('/users/' + user.id + '/posts', function (userErr, posts) {
    if (userErr) {
      alert('Error fetching user posts')
      return
    }
    renderUserPosts(posts)
  })
})
```

Node.js callback hell

then

Benjamin ☆
@winterbe_

Follow

If #nodejs would have existed in 1995

node95.js

```
var floppy = require('floppy');

floppy.load('disk1', function (data1) {
    floppy.prompt('Please insert disk 2', function () {
        floppy.load('disk2', function (data2) {
            floppy.prompt('Please insert disk 3', function () {
                floppy.load('disk3', function (data3) {
                    floppy.prompt('Please insert disk 4', function () {
                        floppy.load('disk4', function (data4) {
                            floppy.prompt('Please insert disk 5', function () {
                                floppy.load('disk5', function (data5) {
                                    // if node.js would have existed in 1995
                                });
                            });
                        });
                    });
                });
            });
        });
    });
});

```

then

How could we flatten that tree?

then

```
new Promise()
```

Formal definition

then

"A promise represents the eventual result of an asynchronous operation."

Formal definition

then

"A promise represents the eventual result of an asynchronous operation."

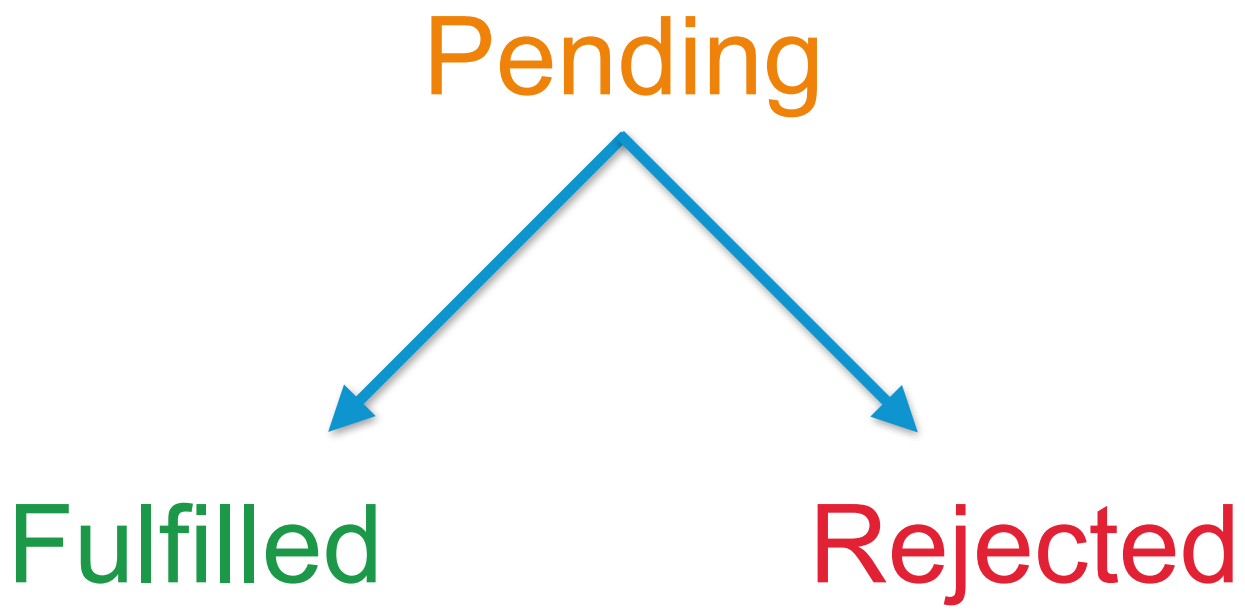
Enter Promises

then

An object which represents and manage the lifecycle of a future result

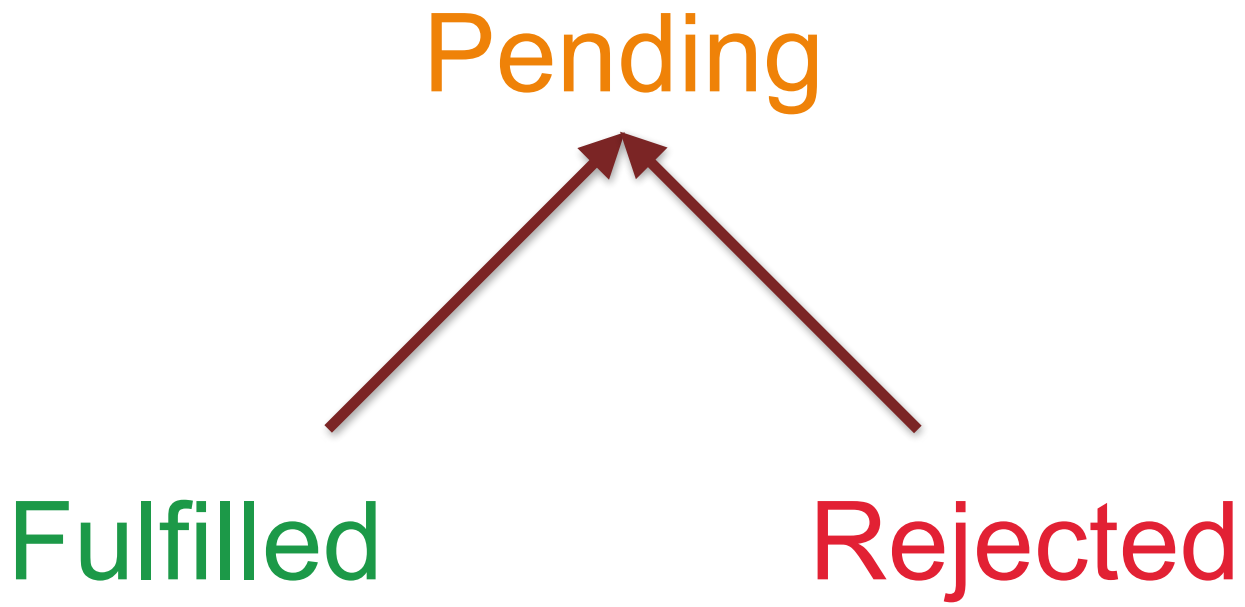
Promise states

then



Promise states

then



Promise API

then

```
// New Promises start in "Pending" state
new Promise(function (resolve, reject) {

  // Transition to "Rejected" state
  reject(new Error('A meaningful error'))

  // Transition to "Fulfilled" state
  resolve({ my: 'data' })

})
```

Promise API

then

```
var promise = new Promise(...)

promise.then(function (result) {
  console.log(result)
})

=> { my: "data" }
```

Promise API

then

```
var promise = new Promise(...)

promise.catch(function (error) {
  console.log(error.message)
})

=> A meaningful error
```

Promise API

then

Node.js callbacks can be easily wrapped in promises

Promise API

then

```
function fetchAsync (url) {
  return new Promise(function (resolve, reject) {
    fetch(url, function (err, data) {
      if (err) {
        reject(err)
      } else {
        resolve(data)
      }
  })
}
```

Promise API

then

Every `.then` and `.catch` returns a new promise, so promises are chainable

Flattening the tree

then

```
function fetchPosts (user) {
  return fetch('/users/' + user.id + '/posts')
}

function fetchSession () {
  return fetch('/users/session')
}

fetchSession()
  .catch(handleSessionError)
  .then(fetchPosts)
  .catch(handlePostsError)
  .then(renderUserPosts)
```

Flattening the tree

then

Chaining allows flattening the callback hell and make continuation passing look sequential

Chaining (a.k.a. sequence monad)

then

```
const makeObject   = e => ({ l: e[0], r: e[1] })
const attachPlus   = e => merge(e, { plus: e.l + e.r })
const attachMinus  = e => merge(e, { minus: e.l - e.r })
const attachTimes  = e => merge(e, { times: e.l * e.r })
const attachDivide = e => merge(e, { divide: e.l / e.r })

fetchTuples()
  .then(makeObject)
  .then(attachPlus)
  .then(attachMinus)
  .then(attachTimes)
  .then(attachDivide)
  .then(console.log.bind(console))
```

Promise Libraries

then

There are a handful of Promise implementations

Solving different issues, focusing on different areas

Promise Libraries

then

So I have to be tied to a single implementation?

Promise Libraries

then

Promises/A+ Contract

https://promisesaplus.com

Promises/A+ Contract

then

Promises/A+ provides interface and behaviour specification for interoperable promises

Promises/A+ Contract

then

So you are free to use the implementation which better fit your needs while keeping your code compatible

Promises/A+ Contract

then

This contract was created because there was no native Promise specification in ECMAScript

ECMAScript 2015 Promise

then

Since ECMAScript 2015 the Promise object was included in the spec

https://tc39.github.io/ecma262/#sec-promise-constructor

ECMAScript 2015 Promise

then

It allows more fun stuff do be done

ECMAScript 2015 Promise

then

Waiting for multiple Promises

Waiting for multiple Promises

then

```
var promises = [
  new Promise(function (resolve, reject) {
    setTimeout(resolve, 1000);
  }),
  new Promise(function (resolve, reject) {
    setTimeout(resolve, 2000);
  })
]

Promise.all(promises).then(function () {
  console.log('Ran after 2 seconds')
})
```

ECMAScript 2015 Promise

then

Racing multiple Promises

Racing multiple Promises

then

```
var promises = [
  new Promise(function (resolve, reject) {
    setTimeout(resolve, 1000);
  }),
  new Promise(function (resolve, reject) {
    setTimeout(resolve, 2000);
  })
]

Promise.race(promises).then(function () {
  console.log('Ran after 2 seconds')
})
```

then

You should definitely look into Promises

then

Bluebird

A complete Promise library

http://bluebirdjs.com

then

HTML Fetch

A Promise approach to HTTP requests

https://fetch.spec.whatwg.org

then

Demo

Fetching stuff from Github

https://github.com/derekstavis/promises-on-the-browser

then

Thanks for watching

Questions?

github.com/derekstavis

twitter.com/derekstavis

facebook.com/derekstavis