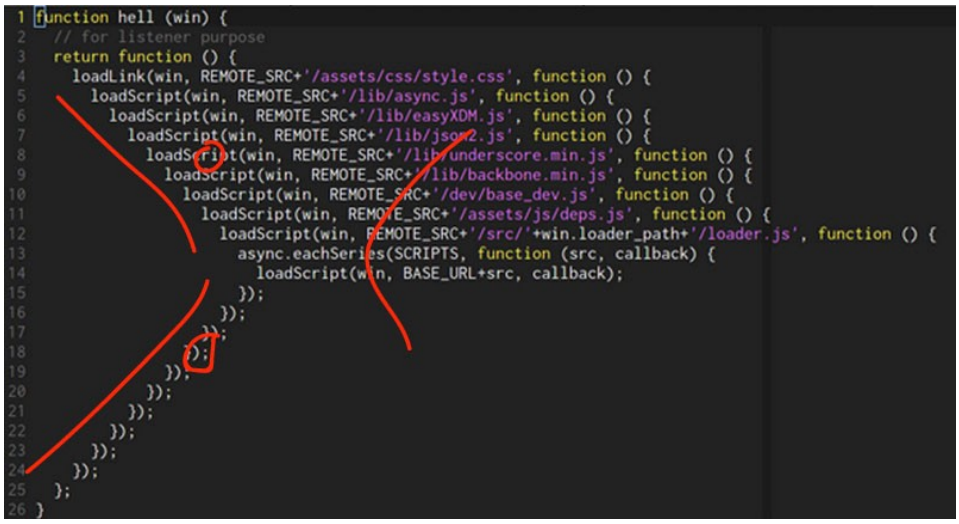
Silvana Goberdhan-Vigle Follow
Sep 12 · 4 min read

Promises

Javascript is a *synchronous, single-threaded* language, that is to say that it doesn't allow for multiple commands to run at the same time. Ruby is the same in this regard but the order of execution is exactly as it has been written in the program. In JavaScript, this is not always the case. As we have seen, this can be quite limiting, and so we use callbacks to be able to write Javascript asynchronously. We've already looked at callbacks, but just to refresh quickly callbacks are simply functions passed as parameters. This is particularly important when doing anything that requires waiting (e.g. some kind of user input, stuff loading). A promise is tool to help us write callbacks more concisely, and avoid what is sometimes referred to as callback hell. Promises were introduced in ES6 but versions have existed in various frameworks for a while (Q, RSVP.js etc.)



A promise represents a value that may or may not become available in the future (or that might be available already). In the UK, universities make offers based on students' expected A-level results, since offers are made before exams are sat. They might agree to take a student if she gets, say, at least two 'A's and a 'B'. If the student makes the offer, then she gets accepted. If she doesn't, then she won't. Promises work in a similar way.

A promise is an object that represents an operation that hasn't finished yet, but is expected to (in the past, promises were known as futures). It has three states: pending, fulfilled, rejected. When the promise is first created, it's in the pending state. It will then change to either fulfilled or rejected. At that point it becomes settled, and will stay in this state permanently.

Promises are a good tool to have because they enable us to make our code more “composable” when used correctly. Broadly speaking, composability means evolving more complex functions (higher-order functions) from simpler ones (perhaps involving several layers), which is good because it makes our code more DRY.

More complex callback structures without promises can often turn into complex, deeply nested structure that are hard to read and also to debug. One way we can avoid this is by abstracting out the callbacks into their own functions. Another way we can refine our code by using promises.

Basic Syntax

```
let iamapromise = new Promise(function(resolve, reject){
 // asynchronous code goes here
 // call resolve() if task successfully completed
 // call reject() if task has failed
})
```

The .then and .catch Methods

The .catch and .then methods are built into promises and called automatically on success or failure of the promise.

Going back to our earlier example, the student getting three ‘A*’s (i.e. success), would result in the promise’s success callback of the .then method being called. The .then method takes callback functions for both success and failure of the promise. Both of these are optional. The student failing all her exams and receiving no A-levels (i.e. rejection), would result in the .catch method being called. The .catch method gets called with a callback function to deal with the error and has one argument (reason). Remember that rejections happen when a promise is explicitly rejected, but can also happen implicitly if an error is thrown in the constructor callback.

```
getImage('pineapple.jpg').then(
    function(successurl){
        document.getElementById('tropical-fruit').innerHTML = '<img src="' + successurl + '" />'
    },
    function(errorurl){
        console.log('Error loading ' + errorurl)
    }
)
```

.then with success and failure

```
getImage('prawn.jpg').then(function(successurl){
    document.getElementById('alien').innerHTML = '<img src="' + successurl + '" />'
}).catch(function(errorurl){
    console.log('Error loading ' + errorurl)
})
```

.then with success, and .catch

Promises in Action

```
loadImage('images/ca3.jpg',
(error, img) => {
  let imgElement = document.createElement("img")
  imgElement.src = img.src
  document.body.appendChild(imgElement)
})
```

Image loading with normal callback

```
function loadImage(url, callback) {
    let image = new Image()

    Image.onload = function() {
        callback(null, image)
    }

    Image.onerror = function() {
        let message = `Unable to load image at ${url}`
        callback(new Error(msg))
    }
    image.src = url
}

export default loadImage
```

Normal callback

```
loadImage('images/ca3.jpg')
.then((img) => {
  var imgElement = document.createElement("img")
  imgElement.src = img.src
  document.body.appendChild(imgElement)
})
```

Image loading with promises

```
function loadImage(url, callback) {
  return new Promise(resolve, reject) => {
    let image = new Image()

    image.onload = function() {
      resolve(image)
    }

    image.onerror = function() {
      let message = `Unable to load image at ${url}`
      reject(new Error(msg))
    }

  }
}
```

Promise

```
let addImg = (src) => {
  let imgElement = document.createElement("img")
  imgElement.src = src
  document.body.appendChild(imgElement)
}

loadImageCallbacked('images/ca1.jpg', (error, img1) => {
  if(error) throw error;
  addImg(img1.src)
  loadImageCallbacked('images/ca2.jpg', (error, img2) => {
    if(error) throw error;
    addImg(img2.src)
    loadImageCallbacked('images/ca3.jpg', (error, img3) => {
      if(error) throw error;
      addImg(img3.src)
    })
  })
})
```

Image loading with multiple images

This is much harder to read than the example with only one image, and is also not running in parallel—each loadImageCallbacked function must wait for the one before it to load.

```
let addImg = (src) => {
  let imgElement = document.createElement("img")
  imgElement.src = src
  document.body.appendChild(imgElement)
}

Promise.all([
  loadImage('images/ca1.jpeg'),
  loadImage('images/ca2.jpeg'),
  loadImage('images/ca3.jpeg')
]).then((images) => {
  images.forEach(img => addImg(img.src))
}).catch((error) => {
  //error-handling stuff
})
```

Promise image loading with multiple images

This is much cleaner than the callbacks example with multiple images, and has a flat structure (not a pyramid). It also allows us to only handle errors once, because we are chaining the .then and .catch methods. This chaining is known as composition.

The .all() method was introduced in ES6. It takes an array of promise objects, and waits for all the promises to settle before it moves on to the .then method (if applicable).

```
function loadImage(url, callback) {
    let image = new Image()

    Image.onload = function() {
        callback(null, image)
    }

    Image.onerror = function() {
        let message = `Unable to load image at ${url}`
        callback(new Error(msg))
    }
    image.src = url
}

export default loadImage
```

Standard callback

A nifty little tool: http://stuk.github.io/promise-me/