# 콜백 지옥

*비동기 JavaScript 프로그램 작성 가이드*

### " *콜백 지옥* "이란 무엇입니까?

비동기 JavaScript 또는 콜백을 사용하는 JavaScript는 직관적으로 이해하기 어렵습니다. 많은 코드가 다음과 같이 표시됩니다.

```
fs.readdir(source, function (err, files) {
  if (err) {
    console.log('Error finding files: ' + err)
  } else {
    files.forEach(function (filename, fileIndex) {
      console.log(filename)
      gm(source + filename).size(function (err, values) {
        if (err) {
          console.log('Error identifying file size: ' + err)
        } else {
          console.log(filename + ' : ' + values)
          aspect = (values.width / values.height)
          widths.forEach(function (width, widthIndex) {
            height = Math.round(width / aspect)
            console.log('resizing ' + filename + 'to ' + height + 'x' + height)
            this.resize(width, height).write(dest + 'w' + width + '_' + filename, function(err) {
              if (err) console.log('Error writing file: ' + err)
            })
          }.bind(this))
        }
      })
    })
  }
})
```

피라미드 모양과 `})` 끝에 있는 모든 것이 보이십니까? 에잇! 이것은 **콜백 지옥** 으로 잘 알려져 있습니다 .

콜백 지옥의 원인은 사람들이 위에서 아래로 시각적으로 실행되는 방식으로 JavaScript를 작성하려고 할 때입니다. 많은 사람들이 이런 실수를 합니다! C, Ruby 또는 Python과 같은 다른 언어에서는 1행에서 발생하는 모든 일이 2행의 코드가 실행을 시작하기 전에 완료될 것으로 예상되며 파일 아래로 계속됩니다. 배우게 되겠지만 JavaScript는 다릅니다.

## 콜백이란 무엇입니까?

콜백은 JavaScript 함수를 사용하기 위한 규칙의 이름일 뿐입니다. JavaScript 언어에는 '콜백'이라는 특별한 것이 없으며 단지 관례일 뿐입니다. 대부분의 함수처럼 일부 결과를 즉시 반환하는 대신 콜백을 사용하는 함수는 결과를 생성하는 데 시간이 걸립니다. '비동기' 또는 '비동기'라는 단어는 '시간이 좀 걸린다' 또는 '지금 당장이 아니라 미래에 일어날 것'을 의미합니다. 일반적으로 콜백은 I/O를 수행할 때만 사용됩니다(예: 항목 다운로드, 파일 읽기, 데이터베이스 통신 등).

일반 함수를 호출할 때 반환 값을 사용할 수 있습니다.

```
var result = multiplyTwoNumbers(5, 10)
console.log(result)
// 50 gets printed out
```

그러나 비동기적이고 콜백을 사용하는 함수는 아무 것도 즉시 반환하지 않습니다.

```
var photo = downloadPhoto('http://coolcats.com/cat.gif')
// photo is 'undefined'!
```

이 경우 gif를 다운로드하는 데 시간이 오래 걸릴 수 있으며 다운로드가 완료되기를 기다리는 동안 프로그램이 일시 중지(일명 '차단')되는 것을 원하지 않습니다.

대신 다운로드가 완료된 후 실행해야 하는 코드를 함수에 저장합니다. 콜백입니다! 함수에 제공하면 `downloadPhoto` 다운로드가 완료될 때 콜백(예: '나중에 다시 전화')을 실행하고 사진을 전달합니다(또는 문제가 발생한 경우 오류).

```
downloadPhoto('http://coolcats.com/cat.gif', handlePhoto)

function handlePhoto (error, photo) {
  if (error) console.error('Download error!', error)
  else console.log('Download finished', photo)
}

console.log('Download started')
```

사람들이 콜백을 이해하려고 할 때 가장 큰 장애물은 프로그램이 실행될 때 실행되는 순서를 이해하는 것입니다. 이 예에서는 크게 세 가지 일이 발생합니다. 먼저 `handlePhoto` 함수가 선언되고 함수가 호출되어 콜백으로 `downloadPhoto` 전달되고 마지막으로 출력됩니다. `handlePhoto` `'Download started'`

은 `handlePhoto` 아직 호출되지 않았으며 단지 생성되어 에 콜백으로 전달되었을 뿐입니다 `downloadPhoto` . 그러나 작업을 완료할 때까지 실행되지 않으며 `downloadPhoto` 인터넷 연결 속도에 따라 시간이 오래 걸릴 수 있습니다.

이 예는 두 가지 중요한 개념을 설명하기 위한 것입니다.

- 콜백 `handlePhoto` 은 나중에 할 일을 저장하는 방법일 뿐입니다.
- 일이 일어나는 순서는 위에서 아래로 읽히지 않고 일이 완료되는 시점을 기준으로 건너뜁니다.

## 콜백 지옥을 어떻게 수정합니까?

콜백 지옥은 잘못된 코딩 관행으로 인해 발생합니다. 운 좋게도 더 나은 코드를 작성하는 것은 그렇게 어렵지 않습니다!

**세 가지 규칙** 만 따르면 됩니다 .

## 1. 코드를 얕게 유지하라

다음은 브라우저 요청을 사용하여 서버에 AJAX 요청을 하는 지저분한 브라우저 JavaScript입니다 .

```
var form = document.querySelector('form')
form.onsubmit = function (submitEvent) {
  var name = document.querySelector('input').value
  request({
    uri: "http://example.com/upload",
    body: name,
    method: "POST"
  }, function (err, response, body) {
    var statusMessage = document.querySelector('.status')
    if (err) return statusMessage.value = err
    statusMessage.value = body
  })
}
```

이 코드에는 두 가지 익명 함수가 있습니다. 그들에게 이름을 지어줍시다!

```
var form = document.querySelector('form')
form.onsubmit = function formSubmit (submitEvent) {
  var name = document.querySelector('input').value
  request({
    uri: "http://example.com/upload",
    body: name,
    method: "POST"
  }, function postResponse (err, response, body) {
    var statusMessage = document.querySelector('.status')
    if (err) return statusMessage.value = err
    statusMessage.value = body
  })
}
```

보시다시피 이름 지정 기능은 매우 쉽고 몇 가지 즉각적인 이점이 있습니다.

- 설명이 포함된 함수 이름 덕분에 코드를 더 쉽게 읽을 수 있습니다.
- 예외가 발생하면 "익명" 대신 실제 함수 이름을 참조하는 스택 추적을 얻게 됩니다.
- 기능을 이동하고 이름으로 참조할 수 있습니다.

이제 함수를 프로그램의 최상위 수준으로 옮길 수 있습니다.

```
document.querySelector('form').onsubmit = formSubmit

function formSubmit (submitEvent) {
  var name = document.querySelector('input').value
  request({
    uri: "http://example.com/upload",
    body: name,
    method: "POST"
  }, postResponse)
}

function postResponse (err, response, body) {
  var statusMessage = document.querySelector('.status')
  if (err) return statusMessage.value = err
  statusMessage.value = body
}
```

여기서 선언 `function` 은 파일 맨 아래에 정의되어 있습니다. 이는 호이스팅 기능 덕분입니다 .

## 2. 모듈화

이것은 가장 중요한 부분입니다. **누구나 모듈(일명 라이브러리)을 만들 수 있습니다** . (node.js 프로젝트의) **Isaac Schlueter의** 말을 인용하자면 : *"각각 한 가지 일을 하는 작은 모듈을 작성하고 더 큰 일을 하는 다른 모듈로 조립하십시오. 거기에 가지 않으면 콜백 지옥에 들어갈 수 없습니다. ."*

위에서 상용구 코드를 꺼내서 두 개의 파일로 분할하여 모듈로 만들어 봅시다. 브라우저 코드 또는 서버 코드(또는 둘 모두에서 작동하는 코드)에서 작동하는 모듈 패턴을 보여드리겠습니다.

`formuploader.js` 다음은 이전의 두 가지 기능을 포함하는 새 파일입니다 .

```
module.exports.submit = formSubmit

function formSubmit (submitEvent) {
  var name = document.querySelector('input').value
  request({
    uri: "http://example.com/upload",
    body: name,
    method: "POST"
  }, postResponse)
}

function postResponse (err, response, body) {
  var statusMessage = document.querySelector('.status')
  if (err) return statusMessage.value = err
  statusMessage.value = body
}
```

비트 는 node, Electron 및 browserify를 `module.exports` 사용하는 브라우저에서 작동하는 node.js 모듈 시스템의 예입니다 . 저는 이 스타일의 모듈을 매우 좋아합니다. 왜냐하면 이 모듈은 모든 곳에서 작동하고 이해하기 매우 간단하며 복잡한 구성 파일이나 스크립트가 필요하지 않기 때문입니다.

이제 `formuploader.js` (브라우저화된 후 스크립트 태그로 페이지에 로드되었으므로) 요구하고 사용하기만 하면 됩니다! 애플리케이션별 코드는 다음과 같습니다.

```
var formUploader = require('formuploader')
document.querySelector('form').onsubmit = formUploader.submit
```

이제 우리의 애플리케이션은 단 두 줄의 코드이며 다음과 같은 이점이 있습니다.

- `formuploader` 새로운 개발자가 이해하기 더 쉬움 -- 모든 기능을 읽어야 하는 수렁에 빠지지 않을 것입니다.
- `formuploader` 코드를 복제하지 않고 다른 곳에서 사용할 수 있으며 github 또는 npm에서 쉽게 공유할 수 있습니다.

## 3. 모든 오류 처리

오류에는 여러 가지 유형이 있습니다. 프로그래머에 의해 발생한 구문 오류(일반적으로 프로그램을 처음 실행하려고 할 때 발견됨), 프로그래머에 의해 발생한 런타임 오류(코드가 실행되었지만 문제를 일으키는 버그가 있음), 플랫폼 오류 잘못된 파일 권한, 하드 드라이브 오류, 네트워크 연결 없음 등으로 인해 발생합니다. 이 섹션은 이 마지막 오류 클래스를 해결하기 위한 것입니다.

처음 두 규칙은 주로 코드를 읽을 수 있게 만드는 것에 관한 것이지만 이 규칙은 코드를 안정적으로 만드는 것에 관한 것입니다. 콜백을 처리할 때 정의에 따라 디스패치되고 백그라운드에서 작업을 수행한 다음 성공적으로 완료되거나 실패로 인해 중단되는 작업을 처리하는 것입니다. 숙련된 개발자라면 이러한 오류가 언제 발생하는지 알 수 없으므로 항상 발생하도록 계획해야 한다고 말할 것입니다.

콜백을 사용하여 오류를 처리하는 가장 일반적인 방법은 콜백에 대한 첫 번째 인수가 항상 오류에 대해 예약되는 Node.js 스타일입니다.

```
var fs = require('fs')

fs.readFile('/Does/not/exist', handleFile)

function handleFile (error, file) {
  if (error) return console.error('Uhoh, there was an error', error)
  // otherwise, continue on and use `file` in your code
}
```

첫 번째 인수를 the로 지정하는 것은 `error` 오류를 처리하는 것을 기억하도록 권장하는 간단한 규칙입니다. 두 번째 인수인 경우 다음과 같은 코드를 작성 하고 오류를 더 쉽게 무시할 수 있습니다. `function handleFile (file) { }`

콜백 오류 처리를 기억하는 데 도움이 되도록 코드 린터를 구성할 수도 있습니다. **사용하기 가장 간단한 것은 standard** 입니다 . 코드 폴더에서 실행하기만 하면 `$ standard` 처리되지 않은 오류가 있는 코드의 모든 콜백이 표시됩니다.

### 요약

1. 함수를 중첩하지 마십시오. 이름을 지정하고 프로그램의 최상위 레벨에 배치하십시오.
2. 함수 호이스팅을 사용하여 함수를 '접힌 부분 아래'로 이동하세요.
3. 모든 콜백에서 **모든 단일 오류를** 처리합니다 . 표준 과 같은 린터를 사용 하면 도움이 됩니다.
4. 재사용 가능한 함수를 만들고 모듈에 배치하여 코드를 이해하는 데 필요한 인지 부하를 줄입니다. 이와 같이 코드를 작은 조각으로 분할하면 오류를 처리하고, 테스트를 작성하고, 코드에 대해 안정적이고 문서화된 공개 API를 생성하고, 리팩토링에 도움이 됩니다.

콜백 지옥을 피하는 가장 중요한 측면은 **함수를 방해하지 않도록 이동하여** 초보자가 프로그램이 수행하려는 작업의 핵심에 도달하기 위해 함수의 모든 세부 사항을 샅샅이 뒤지지 않고도 프로그램 흐름을 더 쉽게 이해할 수 있도록 하는 것입니다. .

함수를 파일 맨 아래로 이동하여 시작한 다음 관련 요구 사항을 사용하여 로드하는 다른 파일로 이동한 다음 마지막으로 와 같은 독립형 모듈로 이동할 수 있습니다 . `require('./photo-helpers.js')` `require('image-resize')`

다음은 모듈을 만들 때 적용되는 몇 가지 규칙입니다.

- 반복적으로 사용되는 코드를 함수로 이동하여 시작
- 함수(또는 동일한 테마와 관련된 함수 그룹)가 충분히 커지면 다른 파일로 옮기고 `module.exports` . 상대 요구를 사용하여 로드할 수 있습니다.
- 여러 프로젝트에서 사용할 수 있는 코드가 있는 경우 자체 readme, 테스트를 제공하고 `package.json` github 및 npm에 게시합니다. 이 특정 접근 방식에는 여기에 나열하기에는 너무 많은 놀라운 이점이 있습니다!
- 좋은 모듈은 작고 하나의 문제에 집중합니다.
- 모듈의 개별 파일은 약 150줄의 JavaScript를 초과할 수 없습니다.
- 모듈에는 JavaScript 파일로 가득 찬 한 수준 이상의 중첩 폴더가 없어야 합니다. 그렇다면 아마도 너무 많은 일을 하고 있는 것입니다.
- 당신이 알고 있는 경험이 많은 코더에게 그들이 어떻게 생겼는지에 대한 좋은 아이디어를 얻을 때까지 좋은 모듈의 예를 보여달라고 요청하십시오. 무슨 일이 일어나고 있는지 이해하는 데 몇 분 이상 걸린다면 그다지 좋은 모듈이 아닐 수 있습니다.

### 더 읽어보기

**callbacks에 대한 더 긴 소개를** 읽거나 **nodeschool** 튜토리얼을 시도해 보십시오 .

또한 모듈식 코드 작성 예제는 **browserify-handbook을** 확인하십시오 .

### 약속/제너레이터/ES6 등은 어떻습니까?

고급 솔루션을 살펴보기 전에 콜백은 JavaScript의 기본 부분이며(그냥 함수일 뿐이므로) 고급 언어 기능으로 이동하기 전에 콜백을 읽고 쓰는 방법을 배워야 합니다. 콜백. 유지 관리 가능한 콜백 코드를 아직 작성할 수 없다면 계속 작업하십시오!

비동기 코드가 위에서 아래로 읽기를 *정말로* 원한다면 시도해 볼 수 있는 몇 가지 멋진 일이 있습니다. 이로 인해 **성능 및/또는 교차 플랫폼 런타임 호환성 문제가 발생할 수 있으므로** 반드시 조사하십시오.

**Promise는** 여전히 하향식으로 실행되는 것처럼 보이는 비동기 코드를 작성하는 방법이며 권장되는 스타일 오류 처리 사용으로 인해 더 많은 유형의 오류를 처리합니다. `try/catch`

**생성기를 사용** 하면 전체 프로그램의 상태를 일시 중지하지 않고 개별 함수를 '일시 중지'할 수 있습니다. 코드를 이해하기가 약간 더 복잡해지는 대신 비동기 코드가 하향식 방식으로 실행되는 것처럼 보입니다. 이 접근 방식의 예는 **watt를** 확인하십시오 .

**비동기 함수는** 더 높은 수준의 구문에서 생성기와 약속을 추가로 래핑하는 제안된 ES7 기능입니다. 그것이 당신에게 흥미롭게 들린다면 그것들을 확인하십시오.

**개인적으로 저는 제가 작성하는 비동기 코드의 90%에 대해 콜백을 사용하고 상황이 복잡해지면 run-parallel** 또는 **run-series** 와 같은 것을 가져옵니다 . 나는 콜백 대 약속 대 나에게 정말로 차이를 만드는 것이 무엇인지 생각하지 않습니다. 가장 큰 영향은 코드를 중첩하지 않고 작은 모듈로 분할하지 않고 단순하게 유지하는 데 있습니다.

어떤 방법을 선택하든 항상 **모든 오류를 처리** 하고 **코드를 단순하게 유지하십시오** .

### 콜백지옥과 산불은 *당신* 만이 막을 수 있다는 것을 기억하세요.

이에 대한 소스는 **github에서** 찾을 수 있습니다 .