Generators and Koa

Modern Flow Control and Middleware in Node

Will Conant

What you should get out of this presentation:

Generators are a new JavaScript feature that can be used as an alternative to the callback pattern in Node

Part 1: Generators

What Are Generators?

Defined in ES6 Harmony

Available in Node 0.11.x with the harmony flag:

```
$ node --harmony your-script.js
```

What Are Generators?

A generator is a function that can pause its own execution and then later be resumed by some caller

Whenever a generator pauses, it may yield a value to the caller that resumed it

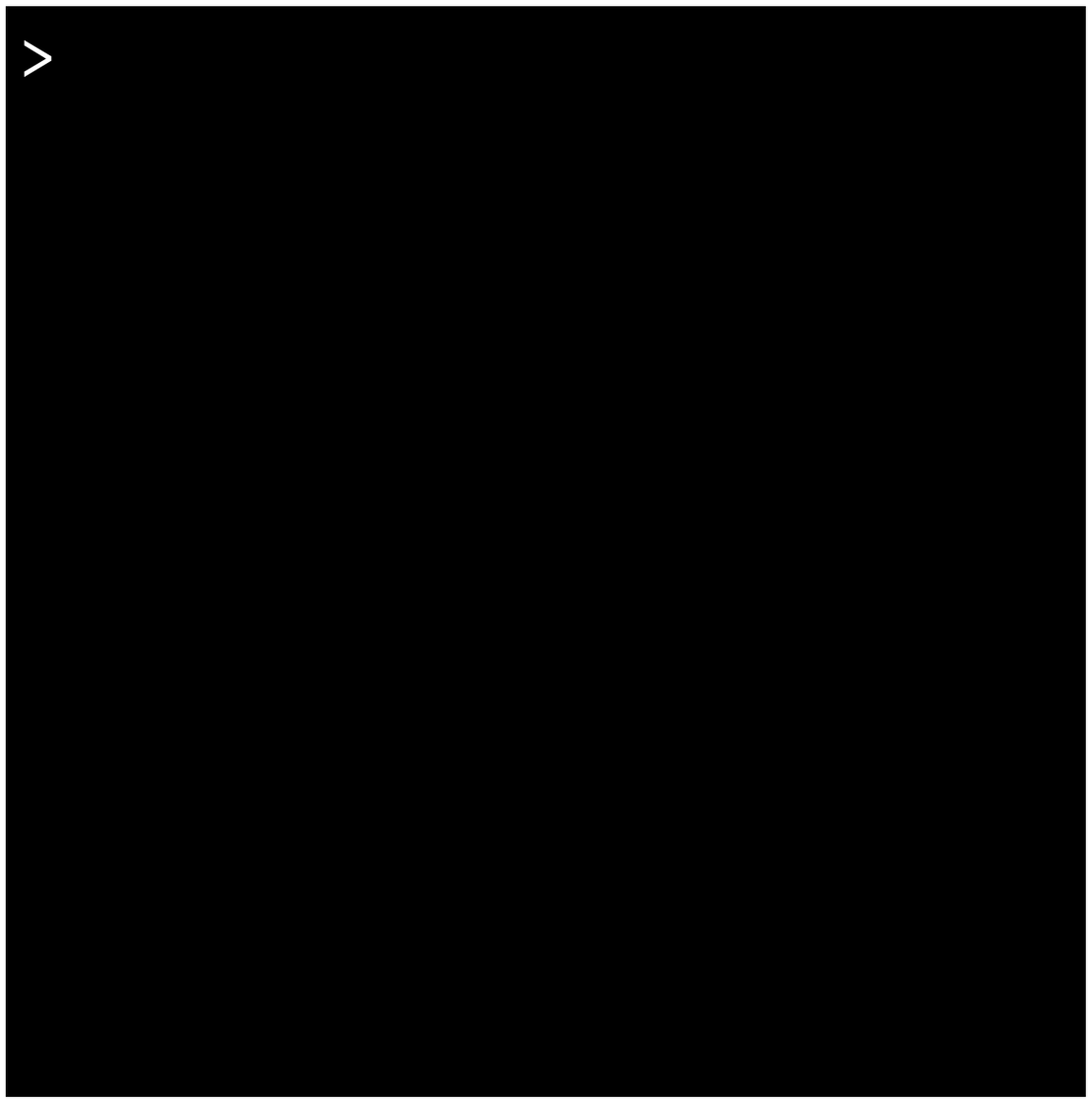
When a caller resumes a generator, the caller may pass a value into the generator

A caller may also throw an exception into a generator

New Syntax

```
function *countTo(x) {
  var i = 1
  while (i < x) {
    i += yield i
  }
  return i
}
```

Running a Generator




```
function *countTo(x) {
  var i = 1
  while (i < x) {
    i += yield i
  }
  return i
}
```

Running a Generator

```
> var g = countTo(5)
```

```
function *countTo(x) {
  var i = 1
  while (i < x) {
    i += yield i
  }
  return i
}
```

Running a Generator

```
> var g = countTo(5)
```

```
function *countTo(x) {
  var i = 1
  while (i < x) {
    i += yield i
  }
  return i
}
```

Running a Generator

```
> var g = countTo(5)
```

```
function *countTo(x) {
  var i = 1
  while (i < x) {
    i += yield i
  }
  return i
}
```

x = 5, i = undefined

Running a Generator

```
> var g = countTo(5)
> g.next()
```

```
function *countTo(x) {
  var i = 1
  while (i < x) {
    i += yield i
  }
  return i
}
```

x = 5, i = undefined

Running a Generator

```
> var g = countTo(5)
> g.next()
```

```
function *countTo(x) {
  var i = 1
  while (i < x) {
    i += yield i
  }
  return i
}
```

x = 5, i = 1

Running a Generator

```
> var g = countTo(5)
> g.next()
```

```
function *countTo(x) {
  var i = 1
  while (i < x) {
    i += yield i
  }
  return i
}
```

x = 5, i = 1

Running a Generator

```
> var g = countTo(5)
> g.next()
```

```
function *countTo(x) {
  var i = 1
  while (i < x) {
    i += yield i
  }
  return i
}
```

x = 5, i = 1

Running a Generator

```
> var g = countTo(5)
> g.next()
{value: 1, done: false}
```

```
function *countTo(x) {
  var i = 1
  while (i < x) {
    i += yield i
  }
  return i
}
```

x = 5, i = 1

Running a Generator

```
> var g = countTo(5)
> g.next()
{value: 1, done: false}
```

```
function *countTo(x) {
  var i = 1
  while (i < x) {
    i += yield i
  }
  return i
}
```

x = 5, i = 1

Running a Generator

```
> var g = countTo(5)
> g.next()
{value: 1, done: false}

> g.next(1)
```

```
function *countTo(x) {
  var i = 1
  while (i < x) {
    i += yield i
  }
  return i
}
```

x = 5, i = 1

Running a Generator

```
> var g = countTo(5)
> g.next()
{value: 1, done: false}

> g.next(1)
```

```
function *countTo(x) {
  var i = 1
  while (i < x) {
    i += yield i
  }
  return i
}
```

x = 5, i = 1

Running a Generator

```
> var g = countTo(5)
> g.next()
{value: 1, done: false}

> g.next(1)
```

```
function *countTo(x) {
  var i = 1
  while (i < x) {
    i += yield i
  }
  return i
}
```

x = 5, i = 2

Running a Generator

```
> var g = countTo(5)
> g.next()
{value: 1, done: false}

> g.next(1)
```

```
function *countTo(x) {
  var i = 1
  while (i < x) {
    i += yield i
  }
  return i
}
```

x = 5, i = 2

Running a Generator

```
> var g = countTo(5)
> g.next()
{value: 1, done: false}

> g.next(1)
```

```
function *countTo(x) {
  var i = 1
  while (i < x) {
    i += yield i
  }
  return i
}
```

x = 5, i = 2

Running a Generator

```
> var g = countTo(5)
> g.next()
{value: 1, done: false}

> g.next(1)
{value: 2, done: false}
```

```
function *countTo(x) {
  var i = 1
  while (i < x) {
    i += yield i
  }
  return i
}
```

x = 5, i = 2

Running a Generator

```
> var g = countTo(5)
> g.next()
{value: 1, done: false}

> g.next(1)
{value: 2, done: false}

> g.next(0)
```

```
function *countTo(x) {
  var i = 1
  while (i < x) {
    i += yield i
  }
  return i
}
```

x = 5, i = 2

Running a Generator

```
> var g = countTo(5)
> g.next()
{value: 1, done: false}

> g.next(1)
{value: 2, done: false}

> g.next(0)
```

```
function *countTo(x) {
  var i = 1
  while (i < x) {
    i += yield i
  }
  return i
}
```

x = 5, i = 2

Running a Generator

```
> var g = countTo(5)
> g.next()
{value: 1, done: false}

> g.next(1)
{value: 2, done: false}

> g.next(0)
```

```
function *countTo(x) {
  var i = 1
  while (i < x) {
    i += yield i
  }
  return i
}
```

x = 5, i = 2

Running a Generator

```
> var g = countTo(5)
> g.next()
{value: 1, done: false}

> g.next(1)
{value: 2, done: false}

> g.next(0)
```

```
function *countTo(x) {
  var i = 1
  while (i < x) {
    i += yield i
  }
  return i
}
```

x = 5, i = 2

Running a Generator

```
> var g = countTo(5)
> g.next()
{value: 1, done: false}

> g.next(1)
{value: 2, done: false}

> g.next(0)
{value: 2, done: false}
```

```
function *countTo(x) {
  var i = 1
  while (i < x) {
    i += yield i
  }
  return i
}
```

x = 5, i = 2

Running a Generator

```
> var g = countTo(5)
> g.next()
{value: 1, done: false}

> g.next(1)
{value: 2, done: false}

> g.next(0)
{value: 2, done: false}

> g.next(3)
```

```
function *countTo(x) {
  var i = 1
  while (i < x) {
    i += yield i
  }
  return i
}
```

x = 5, i = 2

Running a Generator

```
> var g = countTo(5)
> g.next()
{value: 1, done: false}

> g.next(1)
{value: 2, done: false}

> g.next(0)
{value: 2, done: false}

> g.next(3)
```

```
function *countTo(x) {
  var i = 1
  while (i < x) {
    i += yield i
  }
  return i
}
```

x = 5, i = 5

Running a Generator

```
> var g = countTo(5)
> g.next()
{value: 1, done: false}

> g.next(1)
{value: 2, done: false}

> g.next(0)
{value: 2, done: false}

> g.next(3)
```

```
function *countTo(x) {
  var i = 1
  while (i < x) {
    i += yield i
  }
  return i
}
```

x = 5, i = 5

Running a Generator

```
> var g = countTo(5)
> g.next()
{value: 1, done: false}

> g.next(1)
{value: 2, done: false}

> g.next(0)
{value: 2, done: false}

> g.next(3)
```

```
function *countTo(x) {
  var i = 1
  while (i < x) {
    i += yield i
  }
  return i
}
```

x = 5, i = 5

Running a Generator

```
> var g = countTo(5)
> g.next()
{value: 1, done: false}

> g.next(1)
{value: 2, done: false}

> g.next(0)
{value: 2, done: false}

> g.next(3)
{value: 5, done: true}
```

```
function *countTo(x) {
  var i = 1
  while (i < x) {
    i += yield i
  }
  return i
}
```

DONE

Okay, ready for the cool part?

What if you could yield for asynchronous work and be resumed when that work is done?

Look, Ma!
No Callbacks

```
function *getUserAndItems() {
  var userId = this.session.userId

  try {
    var user = yield db.getUser(userId)
    var items = yield db.getItems(userId)

    for (var i = 0; i < items.length; i++) {
      items[i].metadata = yield db.getMetadata(items[i])
    }

    this.body = {user: user, items: items}
  }
  catch (err) {
    this.status = 500
    this.body = {error: err.message}
  }
}
```

Look, Ma!
No Callbacks

```
function *getUserAndItems() {
  var userId = this.session.userId

  try {
    var user = yield db.getUser(userId)
    var items = yield db.getItems(userId)

    for (var i = 0; i < items.length; i++) {
      items[i].metadata = yield db.getMetadata(items[i])
    }

    this.body = {user: user, items: items}
  }
  catch (err) {
    this.status = 500
    this.body = {error: err.message}
  }
}
```

Look, Ma!
No Callbacks

```
function *getUserAndItems() {
  var userId = this.session.userId

  try {
    var user = yield db.getUser(userId)
    var items = yield db.getItems(userId)

    for (var i = 0; i < items.length; i++) {
      items[i].metadata = yield db.getMetadata(items[i])
    }

    this.body = {user: user, items: items}
  }
  catch (err) {
    this.status = 500
    this.body = {error: err.message}
  }
}
```

Look, Ma!
No Callbacks

```
function *getUserAndItems() {
  var userId = this.session.userId

  try {
    var user = yield db.getUser(userId)
    var items = yield db.getItems(userId)

    for (var i = 0; i < items.length; i++) {
      items[i].metadata = yield db.getMetadata(items[i])
    }

    this.body = {user: user, items: items}
  }
  catch (err) {
    this.status = 500
    this.body = {error: err.message}
  }
}
```

How do we yield for asynchronous work?

Thunks

What is a Thunk?

A thunk is a function that accepts a callback as its only argument and arranges for that callback to be called on some future turn of the event loop

```
function delayOneSecond(callback) {
  setTimeout(callback, 1000)
}
```

Thunks Encapsulate Asynchronous Work

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

// create a thunk that will
// read the specified file
function readFileThunk(path) {
  return function(callback) {
    fs.readFile(path, callback)
  }
}
```

Thunks Encapsulate Asynchronous Work

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

// create a thunk that will
// read the specified file
function readFileThunk(path) {
  return function(callback) {
    fs.readFile(path, callback)
  }
}
```

Using a Thunk

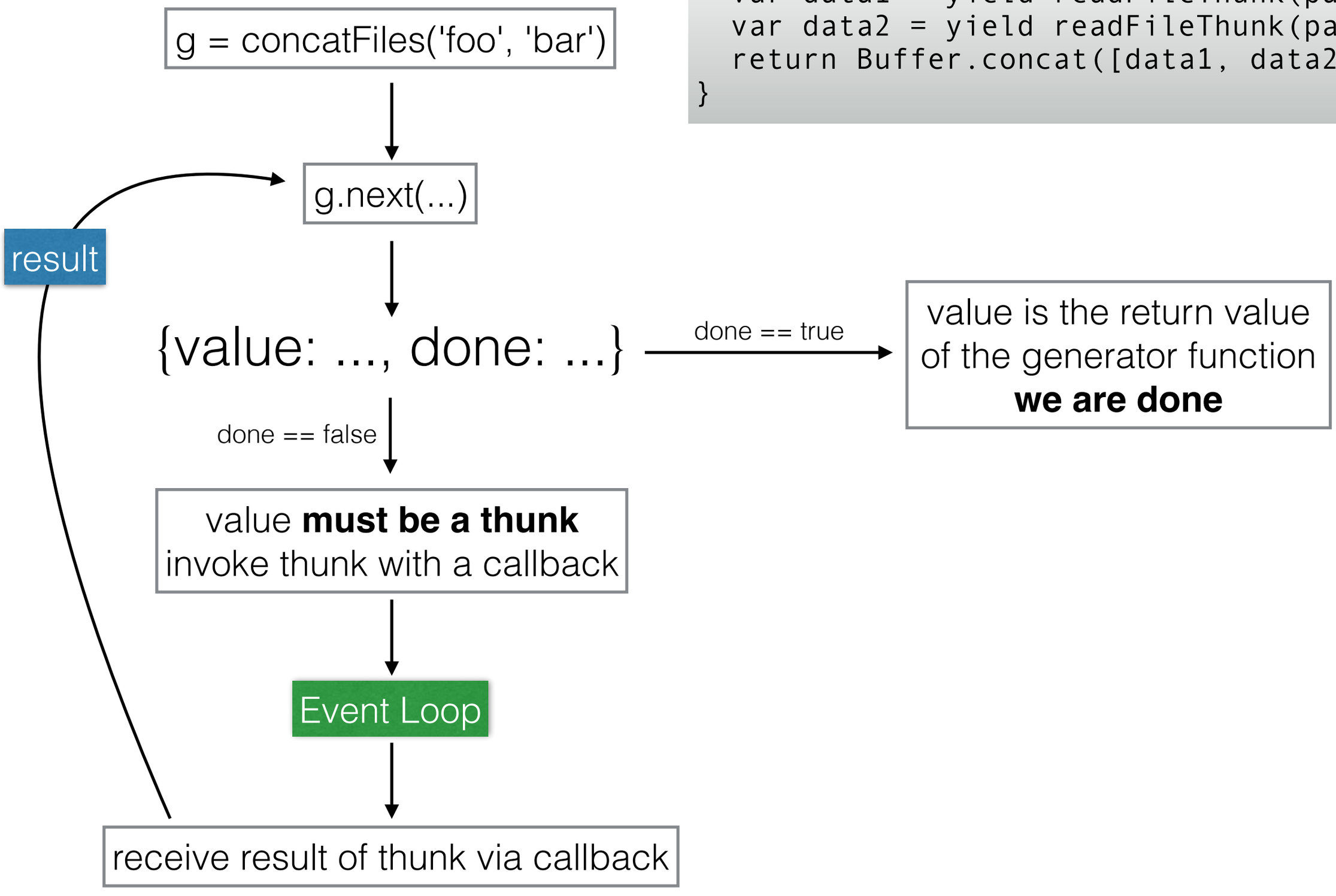
```
// traditional callback
fs.readFile('foo.txt', function(err, data) {
  // use the data
})

// thunk
readFileThunk('foo.txt')(function(err, data) {
  // use the data
})
```

Yielding Thunks From Generator Functions

```
// read and concatenate the contents
// of the files at path1 and path2
function *concatFiles(path1, path2) {
  var data1 = yield readFileThunk(path1)
  var data2 = yield readFileThunk(path2)
  return Buffer.concat([data1, data2])
}
```

How would we run a generator function that yields thunks?

```
function *concatFiles(path1, path2) {
  var data1 = yield readFileThunk(path1)
  var data2 = yield readFileThunk(path2)
  return Buffer.concat([data1, data2])
}
```



```
function runGenerator(g, callback) {
  setImmediate(resume);

  function resume(resumeErr, resumeValue) {
    var yieldOutput;

    try {
      if (resumeErr) {
        yieldOutput = g.throw(resumeErr);
      }
      else {
        yieldOutput = g.next(resumeValue);
      }
    }
    catch (e) {
      callback(e);
      return;
    }

    if (yieldOutput.done) {
      callback(null, yieldOutput.value);
      return;
    }

    var yieldedFunction = yieldOutput.value;

    if (typeof yieldedFunction !== 'function') {
      throw new Error('you must yield a function');
    }

    yieldedFunction(resume);
  }
}
```

Using Our Harness

```
runGenerator(
  concatFiles('foo', 'bar'),
  function(err, result) {
    // do something with result
  }
)

function *concatFiles(path1, path2) {
  var data1 = yield readFileThunk(path1)
  var data2 = yield readFileThunk(path2)
  return Buffer.concat([data1, data2])
}
```

Don’t Write Your Own
Use co

co is an npm module that is **way** better than our example harness

In fact, Koa depends on co

Co Supports Several Types of Yield-able Values

- Thunks
- Generator Objects
- Arrays of either
- A few other types

Calling One Generator Function From Another

```
function *concatFiles(path1, path2) {
  var data1 = yield readFileThunk(path1)
  var data2 = yield readFileThunk(path2)
  return Buffer.concat([data1, data2])
}

function *concatAndReverse(a, b) {
  var data = yield concatFiles(a, b)
  return data.toString().reverse()
}
```

Yielding Arrays for Parallel Evaluation

```
function *concatFiles(path1, path2) {
  var results = yield [
    readFileThunk(path1),
    readFileThunk(path2)
  ]
  return Buffer.concat(results)
}
```

Using Co

```
var co = require('co')

co(
  concatFiles('foo', 'bar')
)(
  function(err, result) {
    // do something with result
  }
)
```

thunkify

Use the thunkify npm module to wrap conventional callback functions

Gotchas

No Yield in forEach

```
function *processAll(items) {
  items.forEach(function(item) {
    yield processItem(item)
  })
}

function *processItem(item) {
  // use your imagination
}
```

No Yield in forEach

```
function *processAll(items) {
  items.forEach(function(item) {
    yield processItem(item)
  })
}

function *processItem(item) {
  // use your imagination
}
```

Usually map() Is Better

```
function *processAll(items) {
  yield items.map(processItem)
}

function *processItem(item) {
  // use your imagination
}
```

Don’t Forget To Yield

```
co(function *() {
  var result =
    concatAndReverse('foo', 'bar')

  console.log(result)
})()
```

Don’t Forget To Yield

```
co(function *() {
  var result =
    yield concatAndReverse('foo', 'bar')

  console.log(result)
})()
```

Part 2: Koa

"Next Generation" Web Framework

- Designed around generators and co
- Created by the authors of Express
- Feature set right in the Goldilocks zone

Trivial Example

```
var koa = require('koa')
var app = koa()

app.use(function *(){
  this.body = 'Hello World'
})

app.listen(3000)
```

Middleware

```
var koa = require('koa')
var app = koa()

app.use(function *(){
  this.body = 'Hello World'
})

app.listen(3000)
```

Cascading Middleware

A Koa App is an object containing an array of generator functions referred to as **middleware**

Each middleware function has the option of **yielding** to the middleware below it. When the downstream middleware completes, control returns back to the upstream middleware.

Example: Logging With Elapsed Time

```
// logger
app.use(function *(next){
  var start = new Date

  yield next

  var ms = new Date - start

  console.log('%s %s - %s’,this.method, this.url, ms)
})

// response
app.use(function *(){
  this.body = 'Hello World'
})
```

The Context

In each middleware function, the **this** value is bound to a Koa Context

A new Context is created for each request

The Context wraps node's built-in request and response objects in a useful interface with just the right amount of sugar

The Context

- Signed cookies
- Request URL parsing
- Sane response rendering and content type detection
- Content negotiation
- Not much else

Everything Else Is Community Middleware

Koa has no built-in middleware

There are close to 100 npm modules providing middleware for request body parsing, routing, file serving, compression, caching, authentication, sessions, templating, logging, and more

Useful Middleware

koa-body

```
var koa = require('koa')
var app = koa()

app.use(require('koa-body')())

app.use(function *(){
  console.log(this.request.body)
})
```

koa-session

```
var koa = require('koa')
var app = koa()

app.keys = ['some secret hurr']

app.use(require('koa-session')())

app.use(function *(){
  var n = this.session.views || 0
  this.session.views = ++n
  this.body = n + ' views'
})
```

koa-static

```
var koa = require('koa')
var static = require('koa-static')
var app = koa()

app.use(static(__dirname + '/static'))
```

koa-router

```
var koa = require('koa')
var router = require('koa-router')
var app = koa()

app.use(router(app))

app.get('/users/:id', function *(next) {
  var user =
    yield getUser(this.params.id)

  this.body = user
})
```

```
// the end
process.exit()
```